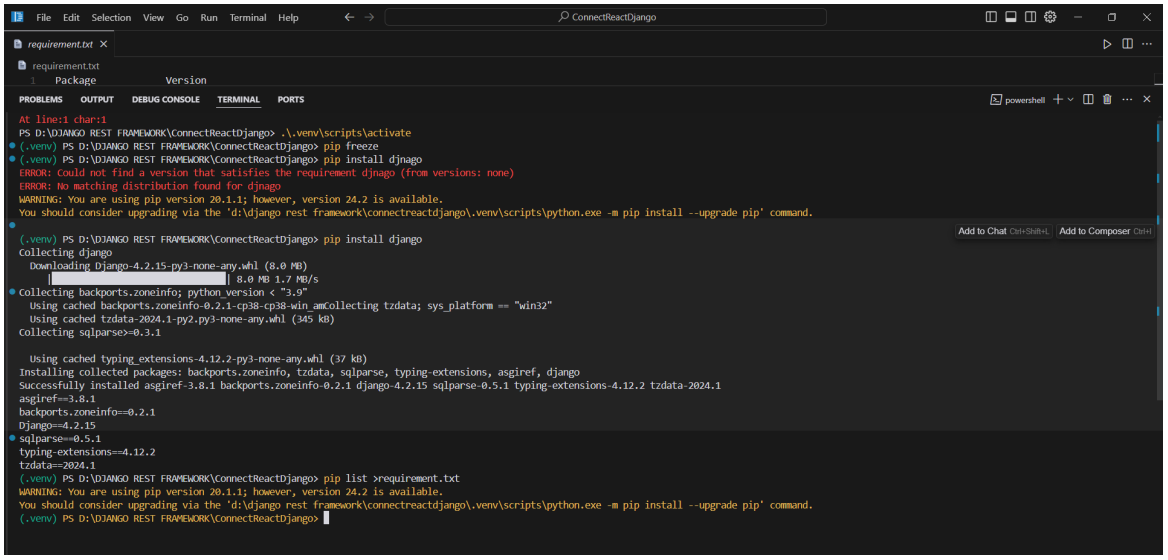
# Django REST Framework

**Disclaimer:**

These notes have been created by *Khushal Savalakha* (khushal.sav@gmail.com) and are based on practical applications and personal understanding of the concepts. Although I have thoroughly checked and verified the information and code provided, there is a possibility that some aspects may contain errors or inaccuracies. Please use this content as a reference and verify details independently as needed. I am sharing these notes as part of a GitHub repository to help others learn, and I hope they are beneficial.

Django REST Framework (DRF) is a powerful and flexible toolkit for building Web APIs in Django. It provides a comprehensive set of features, including serialization, authentication, and authorization, making it easier to develop robust and performant RESTful APIs. DRF integrates seamlessly with Django's ORM and other core features, allowing developers to quickly create APIs that can handle complex data structures and business logic. Additionally, DRF offers extensive documentation, a browsable API interface for testing endpoints, and a large community of users and contributors, making it an excellent choice for both new and experienced developers working on Django projects.

## ▼ Python Virual Environment




It looks like you're trying to create and activate a virtual environment in your current directory but encountering issues.

## Creating and Activating a Virtual Environment in the Current Folder

1. **Use `venv` to Create the Virtual Environment:**

   Since `virtualenv` isn't recognized and you're using Anaconda, you can use Python's built-in `venv` module instead. You tried the right command, but it seems like there was a misunderstanding with the syntax.

   In your current folder, run:

   ```
   python -m venv .venv
   ```

   This command creates a virtual environment in the `.venv` directory within your current folder.

2. **Activate the Virtual Environment:**

   After creating the environment, activate it:

   - **For PowerShell** (which you are using in VS Code):

     ```
     .\\.venv\\Scripts\\Activate
     ```

   - **For Command Prompt (cmd.exe)**:

     ```
     .venv\\Scripts\\activate
     ```

3. **Verify Activation:**

   If successful, you should see the name of the virtual environment (likely `.venv`) appear in parentheses at the start of your terminal prompt, indicating that the environment is active.

## Notes:

- The `.venv` directory is often used for virtual environments and is generally ignored by version control systems (like Git) by default.
- If you're still encountering issues, make sure your Python installation is correctly set up and accessible from your terminal.

This setup should allow you to create and work within a virtual environment directly in your current directory.

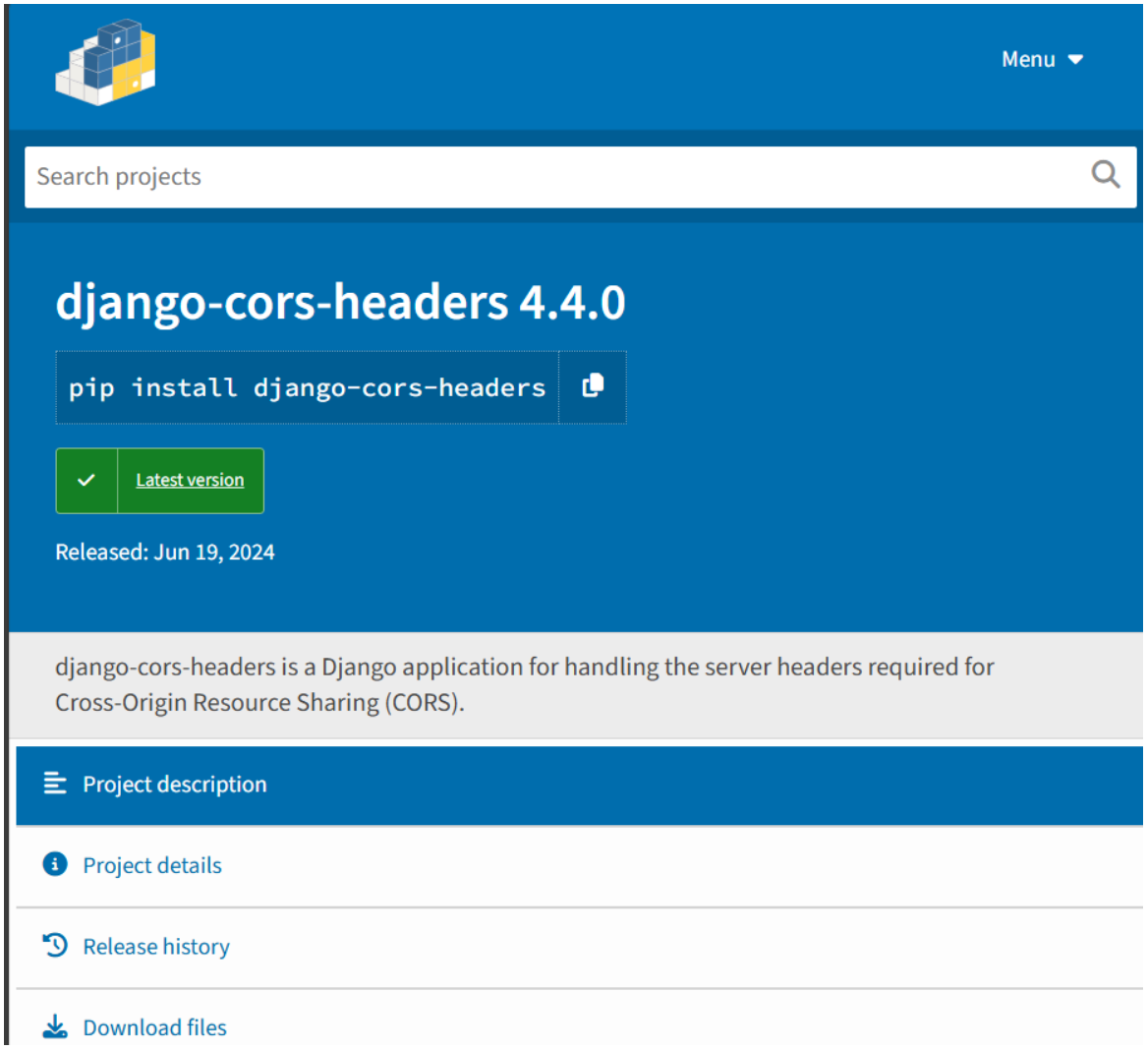

Khushal Savalakha

and how to deactivate it ?

To deactivate the virtual environment, simply run the following command in your terminal:

```
deactivate
```

This command will deactivate the current virtual environment and return you to your global Python environment. After deactivation, you should no longer see the virtual environment name in parentheses at the beginning of your terminal prompt.



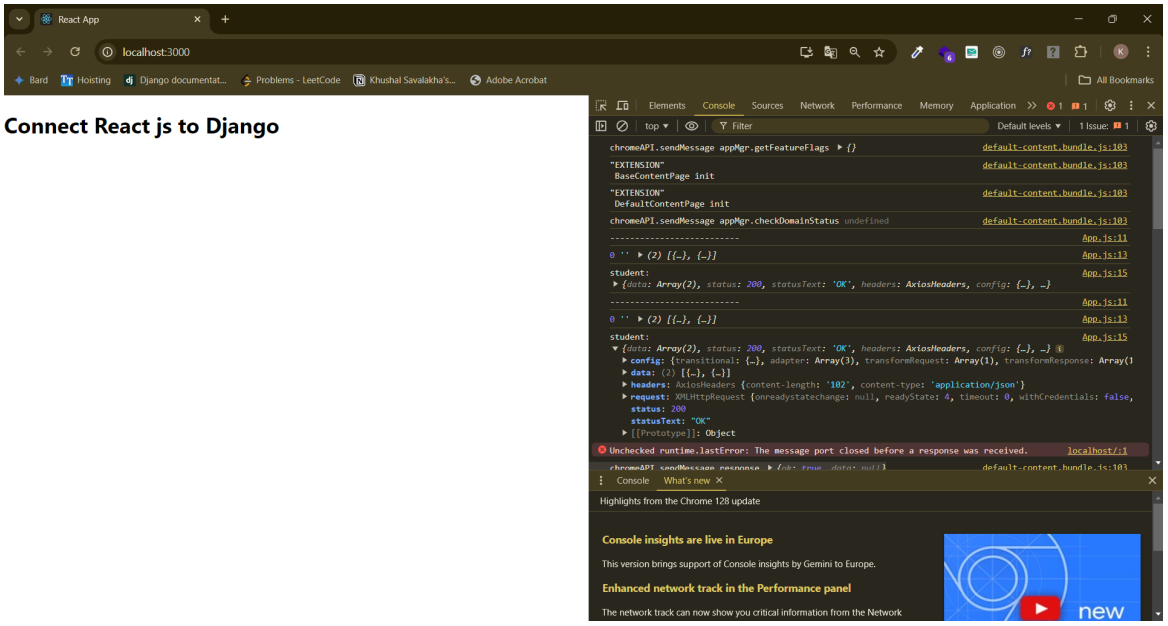

**App.js**

```
import "./App.css";
import { useState, useEffect } from "react";
import  axios from "axios";
function App() {
  const [students, setStudents] = useState([]);
  const [count, setCount] = useState(0);
  useEffect(() => {
    async function getAllStudent() {
      try {
        const students = await axios.get("http://127.0.0.1:
        console.log('--------------------------');
        setCount(count+1)
        console.log(count,'',students.data);
        setStudents(students.data);
        console.log('student:',students);
      } catch (error) {
        console.log(error);
      }
    }
    getAllStudent();
  },[]);


  return (
    <>
      <h1>Connect React js to Django</h1>
      {
        students.map((student,i)=>{
          return(
            <h2>{student.stuname}</h2>
          )
        })
      }
    </>
  );
}
```


Khushal Savalakha

```
export default App;
```

[django-cors-headers · PyPI]




asgiref==3.8.1
backports.zoneinfo==0.2.1
Django==4.2.15
django-cors-headers==4.4.0
djangorestframework==3.15.2
sqlparse==0.5.1
typing-extensions==4.12.2
tzdata==2024.1

## ▼ What is the need of Django Rest Framework

> 💡 In Django Rest framework why we have to convert our Query set data into JSON and then render it in json format and why we need serialization and deserialization and why in simple Django we don't have to do this types of things explain me with simple example depthly by which i can understand all the cocepts.

In Django and Django REST Framework (DRF), the way data is handled differs because the two frameworks serve different purposes. Let’s break down the concepts step by step

with an example to help you understand why serialization, deserialization, and JSON conversion are necessary in DRF but not in simple Django.

## 1. Simple Django (without Django REST Framework)

In a typical Django application, you're usually dealing with HTML templates. For example, let's say you have a model `Book`:

```
from django.db import models

class Book(models.Model):
    title = models.CharField(max_length=100)
    author = models.CharField(max_length=100)
    published_date = models.DateField()
```

In a view, you might retrieve a list of books and pass it to a template:

```
from django.shortcuts import render
from .models import Book

def book_list(request):
    books = Book.objects.all()  # QuerySet
    return render(request, 'books/book_list.html', {'boo
ks': books})
```

In this case, Django handles the data by passing the `books` QuerySet to the template. The template then iterates over the books and displays the data in HTML:

```
<ul>
  {% for book in books %}
    <li>{{ book.title }} by {{ book.author }} ({{ book.p
ublished_date }})</li>
  {% endfor %}
</ul>
```

**Why No JSON or Serialization?**

- **HTML Response:** The data is rendered as HTML, which is intended to be viewed in a web browser.


Khushal Savalakha

- **No Need for Serialization:** You’re not transferring data between different systems or applications; everything stays within the server and the browser. Django templates can directly use QuerySets and model instances.

## 2. Django REST Framework (DRF)

In DRF, you're typically building an API that other applications or front-end frameworks can interact with. Here, data is usually exchanged in JSON format, which is a common data format that can be easily parsed by various programming languages.

Let's use the same `Book` model, but now we're building an API endpoint to return the list of books in JSON format:

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

class BookList(APIView):
    def get(self, request):
        books = Book.objects.all()  # QuerySet
        serializer = BookSerializer(books, many=True)
        return Response(serializer.data)
```

**Key Concepts:**

- **Serialization:** The `BookSerializer` is used to convert the `Book` model instances (which are Python objects) into JSON data. Serialization is the process of converting complex data types (like QuerySets and model instances) into a format that can be easily rendered into JSON, XML, or other content types.
- **Deserialization:** When you receive data in JSON format (e.g., from a POST request), deserialization is the process of converting that JSON data back into a Python object, such as a model instance.

**Why Do We Need Serialization/Deserialization?**

- **Interoperability:** JSON is a lightweight data format that is easy to parse in various programming languages. By converting data to JSON, you make your API accessible to different clients, like mobile apps, front-end frameworks (e.g., React, Angular), or even other backend services.


Khushal Savalakha

- **Data Transfer:** Unlike simple Django, where data stays within the server and the browser, DRF is often used to build APIs where data needs to be transferred over the internet. JSON is the most common format for this kind of data exchange.

**Example of Serializer:**

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

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

Here's what happens in the `BookList` API view:

1. **QuerySet Retrieval:** The `Book.objects.all()` fetches all the books from the database.
2. **Serialization:** The `BookSerializer` converts each `Book` instance into a JSON-compatible Python dictionary.
3. **JSON Response:** The `Response(serializer.data)` returns the serialized data as a JSON response.

**Why Not in Simple Django?**

- In simple Django, the primary goal is to render data into HTML for web pages, not to expose data to other systems.
- The built-in Django views handle data within the context of a web application, while DRF is specifically designed to handle data exchange between different systems using a standardized format like JSON.

## Summary

- **Simple Django:** Focuses on rendering data into HTML for web browsers, no need for JSON conversion.
- **Django REST Framework:** Designed for creating APIs, requires serialization to convert data into JSON format for interoperability and data transfer.
- **Serialization/Deserialization:** Essential in DRF for converting data between Python objects and JSON, enabling communication between different systems.


Khushal Savalakha

Understanding these concepts is crucial when moving from traditional web applications to building APIs.

> 💡 if we use simple Django then end user send data from html page then that data is in which format ?

When an end user submits data from an HTML page in a simple Django application, the data is sent to the server in one of two formats, depending on the method used in the form submission: **URL-encoded (form data)** or **multipart/form-data**.

## 1. URL-encoded (Form Data)

This is the most common format used when submitting a form with the `POST` method in HTML. Here's how it works:

- **Form Submission:** When a user fills out a form and submits it, the data is sent to the server as key-value pairs. The keys correspond to the `name` attributes of the form fields, and the values are the data entered by the user.
- **Encoding:** The data is URL-encoded, meaning that special characters are converted into a format that can be safely transmitted over the internet.
- **Content-Type:** The request is sent with the `Content-Type` header set to `application/x-www-form-urlencoded`.

**Example:**
Consider a form where a user submits their name and email:

```html
<form method="POST" action="/submit/">
    <label for="name">Name:</label>
    <input type="text" id="name" name="name">

    <label for="email">Email:</label>
    <input type="email" id="email" name="email">

    <button type="submit">Submit</button>
</form>
```

When this form is submitted, the data might be sent to the server as:


Khushal Savalakha

```
name=Khushal&email=khushal@example.com
```

## 2. Multipart/form-data

This format is used when a form includes file uploads. It allows for binary data (like images or documents) to be sent along with the regular text data.

- **Content-Type:** The request is sent with the `Content-Type` header set to `multipart/form-data`.
- **Boundary:** The data is separated by a boundary string, which indicates where one form field ends, and the next begins.

**Example:**
Consider a form where a user uploads a profile picture:

```html
<form method="POST" action="/upload/" enctype="multipar
t/form-data">
    <label for="name">Name:</label>
    <input type="text" id="name" name="name">

    <label for="profile_pic">Profile Picture:</label>
    <input type="file" id="profile_pic" name="profile_pi
c">

    <button type="submit">Upload</button>
</form>
```

The data might be sent to the server as a multipart message, which looks like this (simplified example):

```
--boundary
Content-Disposition: form-data; name="name"

Khushal
--boundary
Content-Disposition: form-data; name="profile_pic"; file
name="profile.jpg"
Content-Type: image/jpeg
```


Khushal Savalakha

```
(binary image data)
--boundary--
```

## How Django Handles This Data

In a Django view, you can access the submitted form data using `request.POST` and `request.FILES`:

- `request.POST` contains the text fields from the form (key-value pairs).
- `request.FILES` contains any uploaded files.

```
def submit_form(request):
    if request.method == 'POST':
        name = request.POST.get('name')
        email = request.POST.get('email')
        profile_pic = request.FILES.get('profile_pic')
        # Process the data as needed
```

## Summary

- **URL-encoded (Form Data):** Used for regular text form submissions; data is sent as key-value pairs.
- **Multipart/form-data:** Used for forms that include file uploads; data is sent in a format that handles both text and binary data.
- **In Django:** The data is easily accessible via `request.POST` and `request.FILES`, and Django takes care of parsing and handling these formats internally, so you don't need to worry about the underlying data format.

💡 here data is already in complex form so we don't do any conversion of it ?

In the context of simple Django, when a user submits form data via an HTML page, the data is indeed in a relatively straightforward format—either URL-encoded form data or multipart/form-data. Django automatically handles the parsing and processing of this data, so you, as the developer, typically don’t need to perform any additional conversions. Here's a breakdown of why this is the case:

## 1. URL-encoded (Form Data)


Khushal Savalakha

- **Simple Structure:** The data is submitted as key-value pairs (e.g., `name=Khushal&email=khushal@example.com`). This structure is simple and easy for Django to parse.
- **Django's Handling:** Django automatically parses this data and makes it available in `request.POST`, which is a dictionary-like object. You can access the data using the keys, like `request.POST['name']`.

## 2. Multipart/form-data

- **File Handling:** When files are uploaded, they are sent in a multipart/form-data format, which separates different parts of the data using boundaries.
- **Django's Handling:** Django parses the multipart/form-data and stores the uploaded files in `request.FILES`. It also parses the text fields and stores them in `request.POST`.

## Why No Additional Conversion?

- **Data is Already in a Usable Form:** When data arrives at your Django view, it is already in a form that is easy to work with—either as text in `request.POST` or files in `request.FILES`. There's no need to convert it further because Django has already handled the necessary parsing.
- **Python Native Types:** The data is already parsed into Python native types like strings, dictionaries, and file objects. For instance, when you access `request.POST['name']`, you get a string directly, and when you access `request.FILES['profile_pic']`, you get a file object that you can work with.

## Contrast with Django REST Framework (DRF)

In DRF, serialization is necessary because you're dealing with complex Python objects (like QuerySets or model instances) that need to be converted into a format (like JSON) that can be easily transmitted over the internet and understood by different systems.

- **Serialization:** Converts complex Python objects into a simpler format (like JSON) for transmission.
- **Deserialization:** Converts received JSON data back into complex Python objects.

## In Simple Django: No Complex Conversion Needed

- **Form Data:** The form data sent from the user is already in a simple format that Django can immediately use. There's no need for complex conversions because the data is already in a usable state when it reaches the server.


Khushal Savalakha

- **Automatic Parsing:** Django automatically parses the incoming data into Python types, making it easy for you to handle form submissions without worrying about data conversion.

### Summary

In simple Django, data submitted from an HTML form is already in a straightforward format that Django can automatically parse and convert into Python native types. This is why you don't need to perform additional conversions like serialization or deserialization. The data is directly accessible and usable within your views, allowing you to focus on processing the data rather than converting it.

## ▼ Virtual Enviornment In python

# virtualenv

pypi v20.26.3 | implementation cpython | pypy | python 3.7 | 3.8 | 3.9 | 3.10 | 3.11 | 3.12 | docs passing | chat 117 online

downloads 135M/month | license MIT | issues 25 open | pull requests 2 open | stars 4.8k

`virtualenv` is a tool to create isolated Python environments. Since Python `3.3`, a subset of it has been integrated into the standard library under the venv module. The `venv` module does not offer all features of this library, to name just a few more prominent:

- is slower (by not having the `app-data` seed method),
- is not as extendable,
- cannot create virtual environments for arbitrarily installed python versions (and automatically discover these),
- is not upgrade-able via pip,
- does not have as rich programmatic API (describe virtual environments without creating them).

The basic problem being addressed is one of dependencies and versions, and indirectly permissions. Imagine you have an application that needs version `1` of `LibFoo`, but another application requires version `2`. How can you use both these libraries? If you install everything into your host python (e.g. `python3.8`) it's easy to end up in a situation where two packages have conflicting requirements.

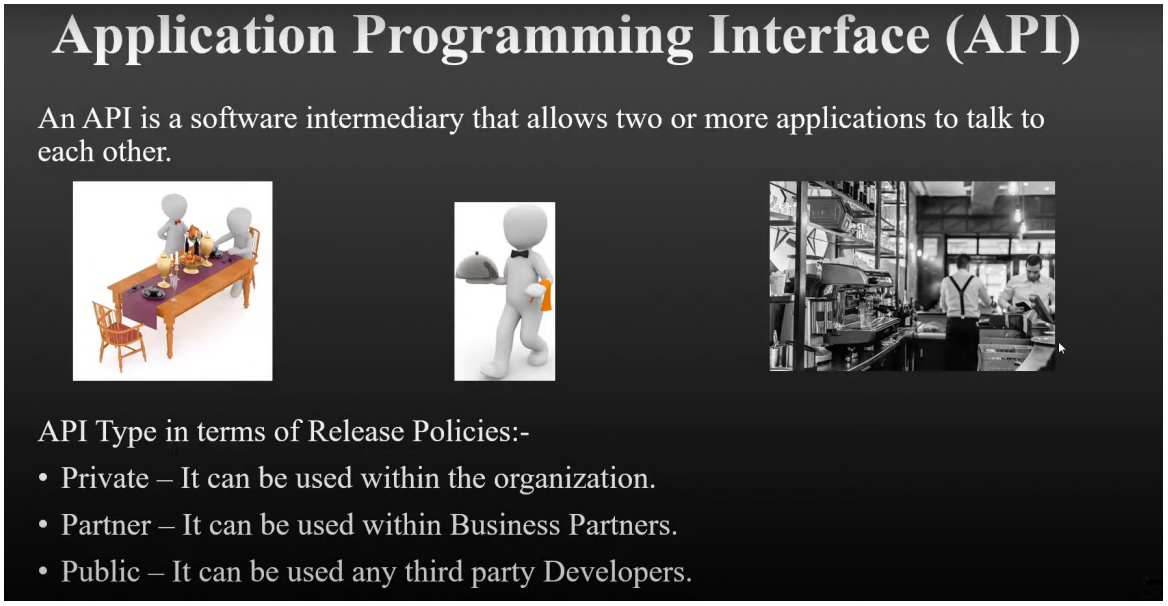
Or more generally, what if you want to install an application *and leave it be*? If an application works, any change in its libraries or the versions of those libraries can break the application. Also, what if you can't install packages into the global `site-packages` directory, due to not having permissions to change the host python environment?

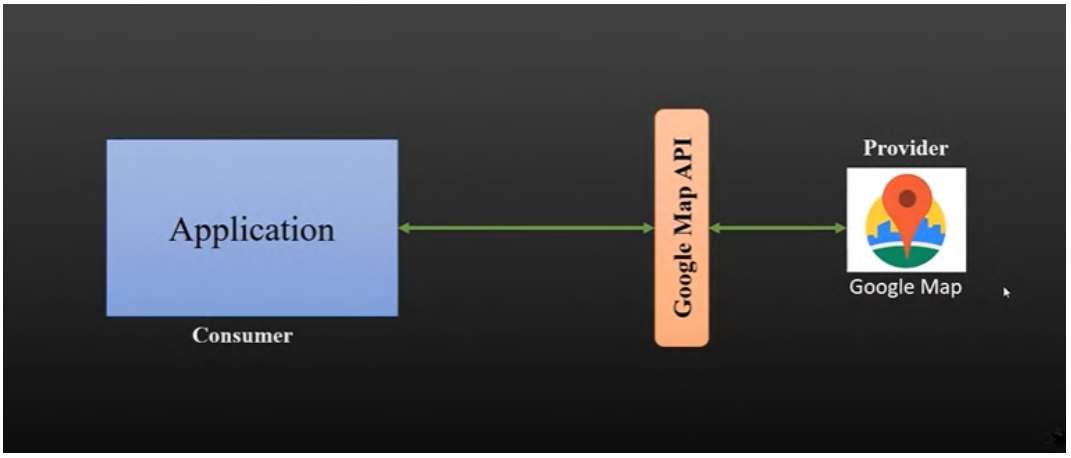
In all these cases, `virtualenv` can help you. It creates an environment that has its own installation directories, that doesn't share libraries with other virtualenv environments (and optionally doesn't access the globally installed libraries either).

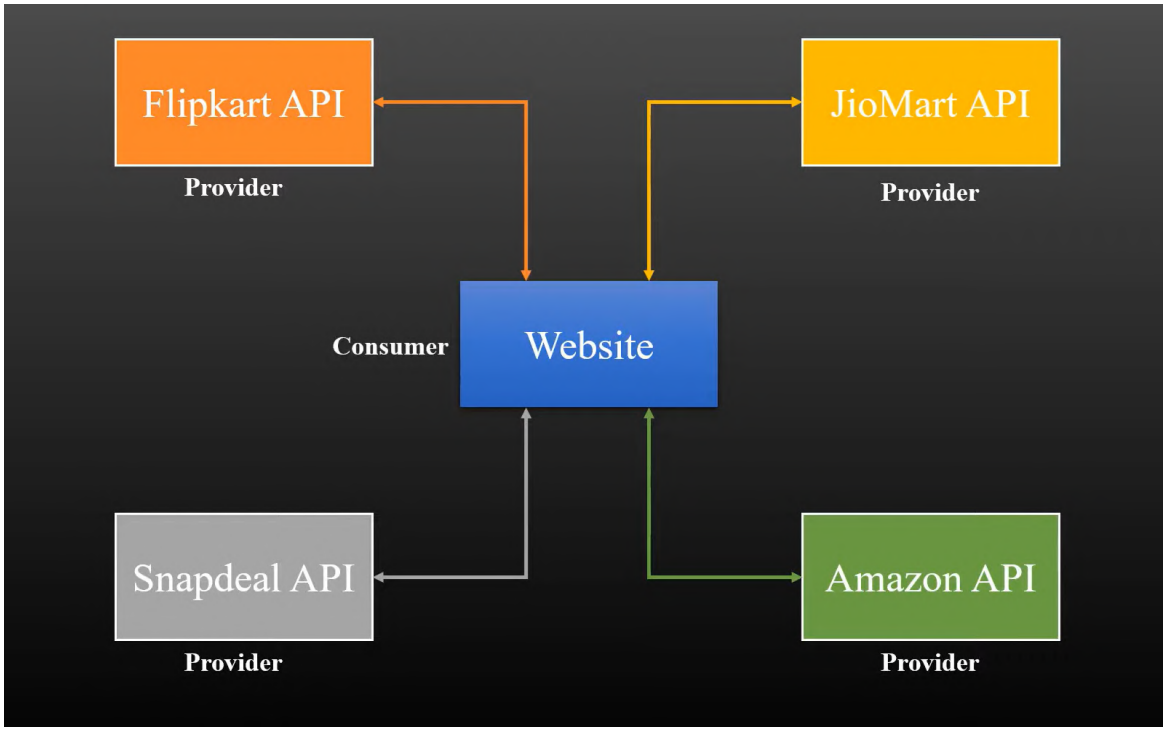

Khushal Savalakha

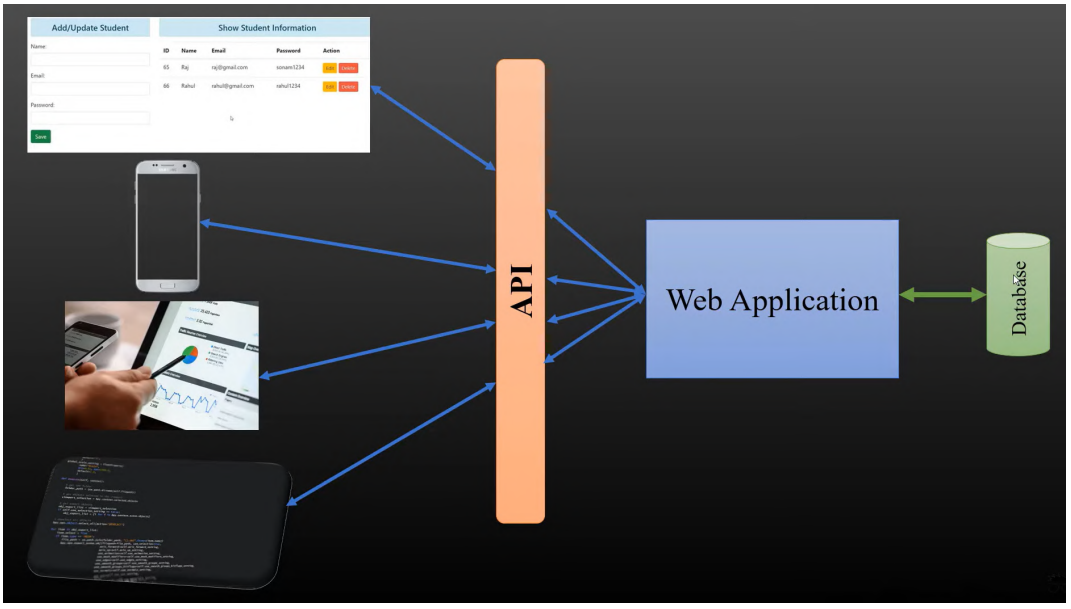
## ▼ API(Application Programming Interface)

- API  is not language dependent.






In the image, the API (Application Programming Interface) acts as an intermediary between the user interface and the backend logic of the web application. Here's a breakdown of how the API is functioning:

1. **User Inputs (Frontend):**
    - The image shows different user interfaces (like a web form for adding/updating student information, mobile phone applications, data visualizations, and possibly code-related interfaces) that interact with the system.

- These interfaces send requests to the API when a user performs actions like adding a student, viewing data, or interacting with the application.

2. **API Handling Requests:**
   - The API is represented as a vertical bar in the center. It acts as a bridge between the user interfaces and the web application.
   - When the user performs an action on the frontend (e.g., submitting a form, requesting data, etc.), the API processes these requests by communicating with the web application.
3. **Web Application Logic:**
   - The web application, depicted on the right side of the API, contains the business logic. It processes the requests received from the API, executes necessary operations (like fetching data, updating records, etc.), and prepares a response.
4. **Database Interaction:**
   - The web application communicates with the database, as shown by the connection between the web application and the database.
   - The API facilitates data retrieval and updates, allowing the web application to store, retrieve, and manipulate data in the database.
5. **Response Back to the User Interface:**
   - Once the web application processes the request, the API sends the response back to the user interface. This could be in the form of updated student information, data for visualizations, or other content.

In summary, the API serves as a mediator that enables communication between different user interfaces and the backend system (web application and database), ensuring that requests are properly handled and responses are delivered back to the user.

> 💡 means 1 web application also has app so app has to communicate with API then API communicate with  webapplication it will communicate with database ?am i true ?

Yes, you are correct. Here's a breakdown of the communication flow:

1. **Mobile App/User Interface:**


Khushal Savalakha

- The mobile app (or any other user interface like a web browser) sends a request to the **API** when the user performs an action (e.g., logging in, viewing data, submitting forms, etc.).

2. **API:**
   - The **API** receives the request from the app. It acts as a middleman and forwards the request to the **Web Application**.
   - The API also ensures that the request is properly formatted, secure, and meets any necessary requirements before forwarding it.
3. **Web Application:**
   - The **Web Application** processes the request received from the API. This is where the core business logic resides. For example, it might verify user credentials, fetch data from the database, or perform other necessary computations.
   - If the request involves interacting with the **Database**, the web application communicates with the database to retrieve or update data.
4. **Database:**
   - The **Database** stores the data. The web application queries the database for information or updates it based on the API's request.
   - Once the database operation is complete, the web application prepares a response.
5. **API Response:**
   - The **API** receives the response from the web application, which could include data or confirmation of an operation (e.g., a successful login).
   - The API then sends this response back to the mobile app or user interface.
6. **User Interface (App):**
   - Finally, the **mobile app** or user interface receives the response from the API and displays it to the user.

So in summary, the communication flow goes like this:

**App/User Interface → API → Web Application → Database → Web Application → API → App/User Interface**

This process allows the app and the web application to work together, with the API serving as the communication bridge.


Khushal Savalakha

💡 why we directly don't connect app with common database ?

Directly connecting an app to a common database might seem like a simpler approach, but it has several significant drawbacks compared to using an API. Here are some reasons why using an API is preferred:

## 1. Security:

- **Data Exposure:** Direct access to the database from an app can expose the database's structure and data, making it vulnerable to SQL injection attacks or unauthorized access.
- **Sensitive Operations:** APIs allow you to control what data and operations are exposed to the app. You can enforce authentication, authorization, and validation layers to protect sensitive data.

## 2. Separation of Concerns:

- **Business Logic Separation:** The web application (and API) contains the business logic that determines how data is processed. If the app directly accesses the database, the business logic would have to be duplicated in each app, leading to inconsistencies and maintenance challenges.
- **Centralized Logic:** With an API, the logic is centralized in the web application, ensuring that all interactions with the database follow the same rules and procedures.

## 3. Data Integrity and Validation:

- **Data Consistency:** APIs ensure that data is validated and processed consistently before reaching the database. Direct access by multiple apps could lead to incorrect data entries, violating constraints or causing inconsistencies.
- **Transaction Management:** APIs can manage complex transactions (e.g., if multiple changes need to be made simultaneously), ensuring that either all changes succeed, or none do (rollback). This is difficult to enforce with direct database access.

## 4. Scalability:

- **Handling Load:** APIs can manage connections more efficiently, pooling resources, caching data, and distributing load across multiple servers. If every app directly connects to the database, it could overwhelm the database with connections and requests.


Khushal Savalakha

- **Horizontal Scaling:** APIs allow for the web application to scale independently from the database, handling more users or operations without directly impacting database performance.

## 5. Version Control and Updates:

- **Versioning:** With APIs, you can create different versions of your endpoints. This allows you to update the backend logic without breaking the functionality of older versions of the app.
- **Easier Updates:** Updating business logic or the database schema is easier when apps interact with an API, as you don't need to update the app every time you make changes.

## 6. Platform Independence:

- **Multiple Clients:** An API can serve multiple types of clients (e.g., web apps, mobile apps, IoT devices) without the need for each client to understand the underlying database. This makes it easier to support multiple platforms with consistent data.
- **Unified Interface:** APIs provide a unified interface for all clients, meaning that any changes to the backend are abstracted away from the frontend apps.

## 7. Maintenance and Manageability:

- **Centralized Maintenance:** By using an API, you can make changes to the database structure, optimize queries, or migrate data without affecting the app directly. This reduces downtime and simplifies maintenance.
- **Reduced Complexity:** Direct database connections from multiple apps increase complexity, especially if different apps need different data or query logic. An API abstracts this complexity.

## 8. Audit and Logging:

- **Tracking:** APIs can log requests and responses, providing a detailed audit trail of what operations were performed and by whom. This is more challenging to manage with direct database access.

## 9. Consistency in Data Access:

- **Controlled Access:** APIs ensure consistent access to data. Direct connections from multiple apps can lead to issues like data conflicts or inconsistencies, especially if different apps perform updates independently.


Khushal Savalakha

In summary, using an API instead of direct database access provides better security, scalability, maintainability, and flexibility. It ensures that the app interacts with the database through a controlled, consistent, and secure interface, which is crucial for building robust and scalable applications.

> 💡 Business Logic Separation: The web application (and API) contains the business logic that determines how data is processed. If the app directly accesses the database, the business logic would have to be duplicated in each app, leading to inconsistencies and maintenance challenges.
> Centralized Logic: With an API, the logic is centralized in the web application, ensuring that all interactions with the database follow the same rules and procedures.

> 💡 if apps are directly connected the there is a threat of SQL injection then what about web application it is connected with database then why it does not has threat of SQL injection give me explanation about all the topics mentioned above.

Great question! Let's explore the topics mentioned—specifically, **Business Logic Separation**, **Centralized Logic**, and **SQL Injection Threats**—and how these concepts are addressed in modern web development with APIs and web applications.

## 1. Business Logic Separation:

- **Explanation:** Business logic refers to the rules and processes that govern how data is handled and manipulated within an application. It includes everything from how user input is validated, to how certain computations are performed, to how data is stored in the database.
- **Why Separation is Important:** When the business logic is separated from the app (i.e., placed in a web application or API layer), it creates a centralized point for managing these rules. This means:
    - **Consistency:** All apps, whether mobile, web, or desktop, interact with the same set of rules. This consistency reduces the risk of data being handled differently in different places.
    - **Maintainability:** Updates to business logic only need to be made in one place (the API/web application) rather than in every app. This simplifies updates and


Khushal Savalakha

reduces the risk of errors.

**Example:** Imagine an e-commerce app where the business logic includes calculating discounts for customers based on their purchase history. If this logic is spread across multiple apps (e.g., the mobile app and the web app), any changes to the discount rules would require updating both apps. If the logic is centralized in the API, you only need to update it once.

## 2. Centralized Logic:

- **Explanation:** Centralized logic means that all interactions with the database and business processes are handled by a single entity, usually the web application or API.
- **Benefits:**
  - **Uniform Data Handling:** All data validation, authorization, and business rules are applied consistently across all platforms and clients.
  - **Security:** Centralized logic allows you to enforce security measures such as input validation, authorization, and data encryption at a single point before data reaches the database.
  - **Reduced Duplication:** Developers don't need to write the same logic multiple times for different apps. Instead, the logic is written once in the API or web application, and all clients use it.

**Example:** If your API has logic that ensures a user can only view their own data, that logic is applied consistently whether the user accesses the data from a mobile app, a web app, or any other platform.

## 3. SQL Injection Threats and Why Web Applications are Safer:

**SQL Injection Overview:**

- **What is SQL Injection?** SQL injection is a security vulnerability that occurs when user input is improperly handled, allowing attackers to execute malicious SQL queries directly against the database.
- **How it Happens:** If an app directly connects to a database and uses raw SQL queries without proper validation or escaping, attackers can inject SQL code through user input fields (e.g., a login form) to manipulate the database.

**Why Web Applications and APIs Reduce the Risk of SQL Injection:**

- **Input Validation and Sanitization:**


Khushal Savalakha

- **Web Application Layer:** Modern web applications and APIs include layers of input validation and sanitization that check and clean user input before it reaches the database. This prevents malicious input from being treated as part of a SQL query.
  - **Example:** If a user enters something like `'; DROP TABLE users; --` into a form field, the web application should sanitize this input to ensure it is treated as a harmless string rather than a SQL command.

- **Use of Parameterized Queries and ORM:**
  - **Parameterized Queries:** Web applications and APIs often use parameterized queries, which separate SQL code from user input. This prevents user input from being interpreted as SQL code.
  - **Example:** In a parameterized query, user input is treated as data, not executable code, making SQL injection impossible.
  - **ORM (Object-Relational Mapping):** Many modern web frameworks use ORM libraries, which abstract database operations into high-level functions. ORM tools generate SQL queries safely, further reducing the risk of SQL injection.

- **Centralized Security Practices:**
  - **API Layer Security:** APIs and web applications are usually built with multiple layers of security, including authentication, authorization, and rate limiting. These measures ensure that only authorized users and requests can interact with the database.
  - **Example:** An API might require users to authenticate before accessing sensitive data, reducing the chance of unauthorized access and potential SQL injection attacks.

- **Security Best Practices in APIs and Web Applications:**
  - **Web frameworks like Django, Express, and others have built-in protection mechanisms against SQL injection, making it harder for developers to accidentally introduce vulnerabilities.**
  - **Centralized Error Handling:** APIs and web applications centralize error handling, logging, and auditing, making it easier to detect and respond to potential security threats.

**Why Direct Database Access is Riskier:**


Khushal Savalakha

- **Lack of Centralized Control:** When an app directly accesses a database, it’s harder to enforce security measures uniformly across all clients. Each app might handle input validation differently, increasing the risk of SQL injection.
- **Increased Attack Surface:** If multiple apps connect directly to the database, each app represents a potential entry point for attackers. With an API, you reduce the attack surface by funneling all database interactions through a single, controlled interface.

## 4. Why Using an API/Web Application is Safer and Better:

- **Centralized Logic and Security:** With an API/web application handling all database interactions, you can apply consistent security measures, business logic, and validation across all clients.
- **Reduced Risk of SQL Injection:** By using techniques like input validation, parameterized queries, and ORMs, web applications significantly reduce the risk of SQL injection compared to direct database access.
- **Maintainability and Scalability:** Centralizing logic in an API/web application simplifies maintenance, allows for easier updates, and ensures consistent behavior across all platforms.

In summary, by using an API or web application to handle business logic and database interactions, you achieve better security (especially against SQL injection), maintainability, and consistency across different platforms. Direct database access bypasses many of these protections, making it riskier and harder to manage in the long run.

> 💡 but we can also do that approach like bussiness logic in an app and web application has to communicate with API and API connect with APP and it has to communicate with database am i true ?

Yes, your approach is technically possible, but it introduces some complexities and potential issues. Let's explore this alternative architecture where the business logic resides in the app, and the web application communicates with the API, which then connects to the app and ultimately interacts with the database.

## 1. How the Architecture Would Work:


Khushal Savalakha

- **Business Logic in the App:** The app would contain the business logic that governs how data is processed. It would validate user input, enforce rules, and handle computations.
- **Web Application → API → App:** The web application would communicate with an API, which then interacts with the app to perform operations.
- **App → Database:** After processing the request and applying business logic, the app would communicate directly with the database to retrieve or update data.

## 2. Challenges with This Approach:

## 1. Duplication of Business Logic:

- **Complexity in Maintenance:** If the business logic is in the app, you must ensure that every app version (e.g., mobile app, desktop app) implements the same logic. This can lead to duplication and potential inconsistencies if different versions of the app don't stay in sync.
- **Updates:** If you need to update the business logic, you'll have to update every app, which can be cumbersome, especially if users don't update their apps promptly. This can lead to different versions of the app behaving differently.

## 2. Security Risks:

- **Data Validation:** Since the app is responsible for business logic, it also needs to handle data validation. If an attacker compromises the app, they could manipulate the logic or bypass validations, leading to potential security vulnerabilities.
- **Increased Attack Surface:** Exposing business logic in the app increases the attack surface, making it more vulnerable to reverse engineering, tampering, or other attacks. The app would need to securely handle sensitive operations, which is harder to control on client-side devices compared to a server.

## 3. Dependency on App for Core Functions:

- **Tightly Coupled Systems:** If the web application needs to perform operations that rely on the app, the two systems become tightly coupled. This can introduce latency and make the architecture harder to manage.
- **Offline Scenarios:** If the app is unavailable (e.g., offline or facing issues), the entire system might be disrupted. This could impact the availability of the web application and other services that rely on the app.


Khushal Savalakha

## 4. Limited Flexibility:

- **Cross-Platform Challenges:** If your system needs to support multiple platforms (e.g., iOS, Android, web), duplicating business logic across different platforms can be challenging. This could lead to inconsistencies in how data is processed.
- **Scalability Issues:** Handling business logic on the client side (e.g., in the app) can limit scalability. Server-side logic in the API allows you to scale more effectively, as you can optimize server resources to handle increased traffic.

## 5. Data Integrity and Consistency:

- **Lack of Centralized Control:** If each app handles business logic independently, there's a risk of data inconsistency. Different versions of the app might process data differently, leading to discrepancies in the database.
- **Transaction Management:** Handling complex transactions across multiple apps and ensuring data consistency can be challenging. Centralized logic in an API or web application makes it easier to manage transactions and ensure data integrity.

## 3. Benefits of the Traditional Approach (Business Logic in API/Web Application):

## 1. Centralized Business Logic:

- **Consistency Across Platforms:** When the business logic is centralized in an API or web application, all platforms (e.g., mobile apps, web apps) follow the same rules and procedures, reducing the risk of inconsistencies.
- **Simplified Maintenance:** You only need to update the logic in one place (the API or web application) rather than updating each app individually.

## 2. Enhanced Security:

- **Server-Side Control:** Keeping business logic on the server side (in the API/web application) reduces the risk of tampering and unauthorized access, as the logic is not exposed to client-side devices.
- **Data Validation:** Centralized validation and security measures ensure that data is handled consistently and securely, reducing the risk of vulnerabilities like SQL injection or data breaches.

## 3. Better Scalability:


Khushal Savalakha

- **Optimized Resources:** Centralizing logic in the API/web application allows you to optimize server resources, handle increased traffic more effectively, and scale the system as needed.
- **Reduced Dependency on Client Devices:** The system is less dependent on client devices (e.g., apps), making it more resilient and available even if individual clients face issues.

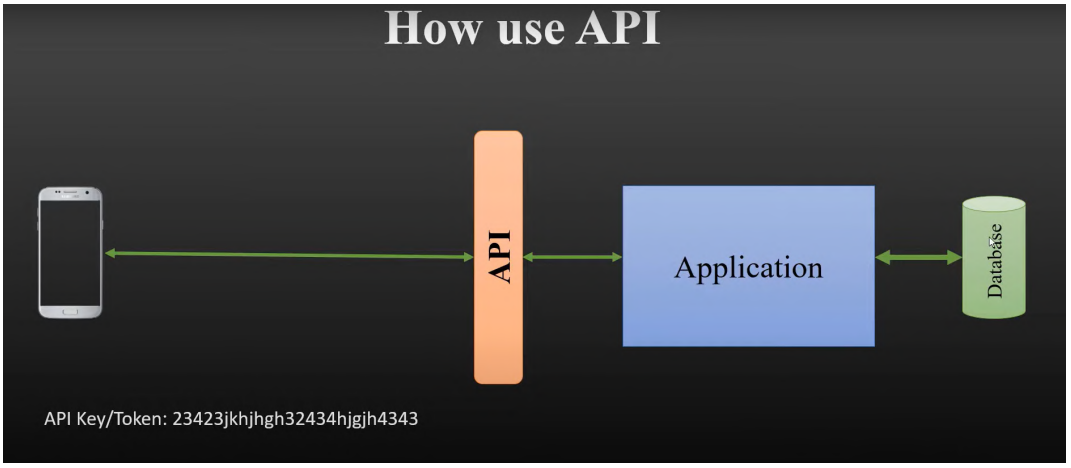
## 4. Flexibility and Extensibility:

- **Easier Integration:** Centralized logic in the API/web application makes it easier to integrate new platforms or services (e.g., adding a new mobile app or IoT device) without duplicating business logic.
- **Version Control:** APIs can handle versioning, allowing you to introduce new features or changes without breaking existing clients.

## 4. Conclusion:

While the approach of placing business logic in the app and having the web application communicate with the API (and then the app) is technically possible, it introduces several complexities and risks, especially in terms of security, maintainability, and consistency. Centralizing business logic in an API or web application provides better control, security, and scalability, making it the preferred approach in most cases.

By centralizing the logic in the API or web application, you ensure that all clients interact with the same set of rules and procedures, reducing the risk of errors, inconsistencies, and security vulnerabilities.



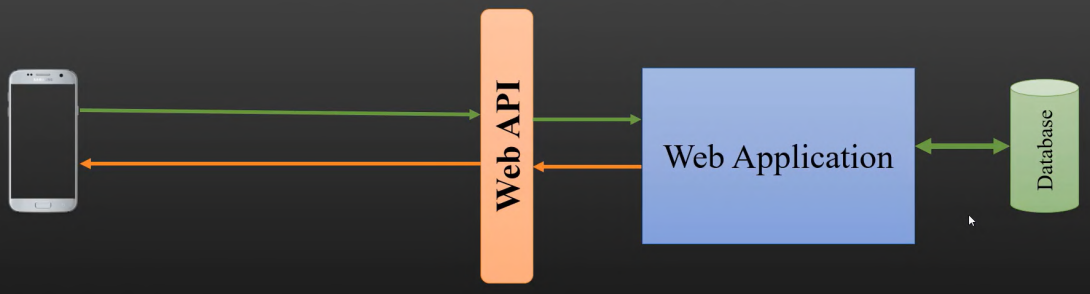
## ▼ Web API

# Web API

An API, which is interface for web is called as Web API.

It may consist of one or more endpoints to define request and response.

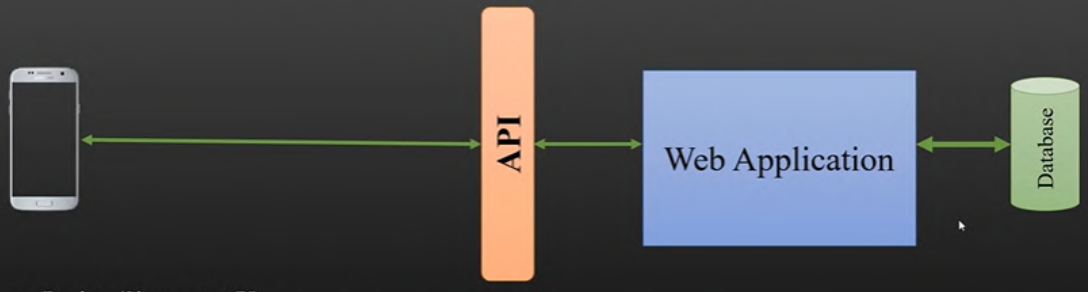
# How Web API Works




- Client makes HTTP Request to API
- API will communicate to Web Application/Database (If needed)
- Web Application/Database provides required data to API
- API returns Data to Client

Note – Json Data, XML Data

# How to use Web API




- Register/Sign-up to API
- API may provide API Key for Authentication purpose
  API Key/Token: *23423jkhjhgh32434hjgjh4343*
- Whenever you need to communicate with server make Request to API with API Key
  http://geekyshows.com/?key=*23423jkhjhgh32434hjgjh4343*
- If API Key authentication succeed, API will provide required Data



Khushal Savalakha

💡 what is the difference between API and Web API ?

Here’s a concise explanation of the difference between API and Web API:

- **API (Application Programming Interface)**:
  - **Definition**: A set of rules and tools for building software applications. It defines how different software components should interact.
  - **Scope**: APIs can be used for any type of interaction between software components, including libraries, operating systems, or services.
  - **Examples**: Database APIs, OS-level APIs, and hardware APIs.
- **Web API (Web Application Programming Interface)**:
  - **Definition**: A specific type of API that operates over the web. It allows different web-based applications to communicate with each other using standard protocols like HTTP/HTTPS.
  - **Scope**: Web APIs are used for web services and are accessed through web requests. They typically return data in formats like JSON or XML.
  - **Examples**: REST APIs, SOAP APIs, and GraphQL APIs.

In summary, all Web APIs are APIs, but not all APIs are Web APIs. Web APIs are a subset of APIs designed specifically for web-based interactions.

💡 i don't understand difference properly explain depthly with example.

Sure, let's dive deeper into the difference between API and Web API with examples:

## 1. API (Application Programming Interface)

**Definition**: An API is a set of rules and protocols that allows different software components to communicate with each other. It defines methods and data formats for how different software systems should interact. APIs can be used in various contexts, not just over the web.

**Example**:

- **Library API**: Imagine you have a math library with a function to calculate the square root. The API of this library might define a method `sqrt(number)` which returns the


Khushal Savalakha

square root of the given number. The interaction with this API happens within your program.

```
// Example in Java
import java.lang.Math;

public class MathExample {
    public static void main(String[] args) {
        double result = Math.sqrt(16);  // Calls the
sqrt method from the Math API
        System.out.println("Square root: " + result);
    }
}
```

- **Operating System API**: APIs provided by an operating system allow programs to perform tasks such as file operations or memory management. For example, the Windows API allows applications to interact with the Windows OS to perform tasks like opening files or managing windows.

## 2. Web API

**Definition**: A Web API is a type of API that is accessed over the web using HTTP/HTTPS protocols. Web APIs allow different web-based applications to interact with each other. They are designed to handle requests and responses over the internet and are often used to enable integration between different systems or services.

**Example**:

- **REST API**: Suppose you want to fetch weather data from a weather service. You would use a RESTful Web API to make HTTP requests to a specific URL provided by the weather service. The API might offer endpoints like `/weather` that you can query to get weather information.

  ```
  // Example HTTP GET request to a REST API
  GET <https://api.weather.com/v3/weather/forecast?apiK
  ey=YOUR_API_KEY&location=New_York>
  ```

  **Response Example**:


Khushal Savalakha

```
{
    "location": "New York",
    "temperature": "75°F",
    "condition": "Sunny"
}
```

**Code Example (Python using `requests` library)**:

```
import requests

response = requests.get('<https://api.weather.com/v3/
weather/forecast>', params={
    'apiKey': 'YOUR_API_KEY',
    'location': 'New_York'
})
data = response.json()
print(f"Temperature: {data['temperature']}")
```

## Summary

- **API**: Can be used for various types of software interactions. Examples include library APIs or OS APIs, which work within the system or application context.
- **Web API**: A specific kind of API that operates over the internet using HTTP/HTTPS. It facilitates communication between web-based systems and is commonly used for integrating services or accessing web resources.

In essence, Web APIs are a subset of APIs designed for web-based communication, enabling different web services to interact and exchange data over the internet.

> 💡 But as i know API is work as mediater like mobile application and web application?

You're correct that APIs can act as intermediaries between different applications, such as mobile apps and web apps. Let's clarify how APIs function as intermediaries with some examples:

## Role of APIs as Mediators


Khushal Savalakha

**1. Mobile Applications and Web Applications**:

- **Scenario**: Consider a mobile application that allows users to view and manage their tasks. This mobile app might need to interact with a web application that provides a task management service.
- **How APIs Work**: The mobile app uses a Web API provided by the web application to send and receive data. The Web API acts as the intermediary that facilitates communication between the mobile app and the web application.

  **Example**:

  - **Mobile App**: Sends a request to the Web API to fetch a list of tasks.

    ```
    GET <https://api.taskmanager.com/tasks?userId=123>
    ```

  - **Web API**: Processes this request, interacts with the web application's database to retrieve the tasks, and returns the data in a format (like JSON) that the mobile app can understand.

    ```
    [
        {"taskId": "1", "taskName": "Buy groceries",
    "status": "Completed"},
        {"taskId": "2", "taskName": "Call the bank",
    "status": "Pending"}
    ]
    ```

  - **Mobile App**: Receives the data from the Web API and displays the list of tasks to the user.

**2. Interacting with External Services**:

- **Scenario**: A web application needs to integrate with a third-party service, like a payment gateway or weather information provider.
- **How APIs Work**: The web application makes requests to the Web API of the third-party service. The API facilitates the communication between the web application and the external service.

  **Example**:

  - **Web Application**: Makes a request to a payment gateway's Web API to process a payment.


Khushal Savalakha

```
POST <https://api.paymentgateway.com/process>
Content-Type: application/json

{
    "amount": "100.00",
    "currency": "USD",
    "cardNumber": "4111111111111111",
    "expiryDate": "12/24"
}
```

- **Web API**: Processes the payment request and returns a response indicating whether the transaction was successful.

```
{
    "transactionId": "abc123",
    "status": "Success"
}
```

## Summary

- **API as Mediator**: APIs indeed act as intermediaries by allowing different systems or applications to communicate and exchange data. They provide a standardized way for mobile apps, web apps, and other systems to interact with each other, regardless of their underlying implementations.
- **Web API**: Specifically refers to APIs that are accessible over the web, facilitating communication between web-based systems and services.

In essence, APIs are the bridge that allows disparate systems to work together and exchange information seamlessly.

💡 make a table to explain difference between API and Web API .

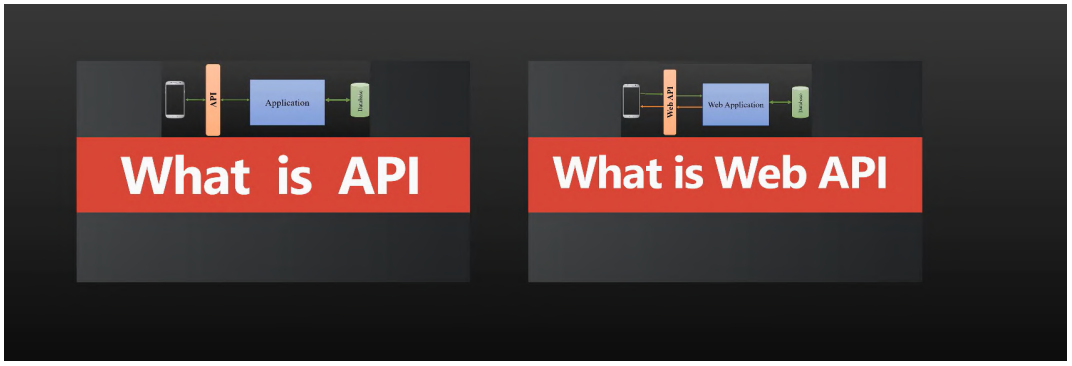
Here's a table summarizing the differences between API and Web API:

| Aspect | API | Web API |
|---|---|---|
| **Definition** | A set of rules and protocols for interacting with software components. | A specific type of API that operates over the web using HTTP/HTTPS |


Khushal Savalakha

| | | |
|---|---|---|
| | | protocols. |
| **Scope** | Can be used for various types of interactions (e.g., libraries, OS, hardware). | Specifically designed for web-based interactions and services. |
| **Protocols** | May use various protocols depending on the context (e.g., system calls, library functions). | Uses HTTP/HTTPS protocols for communication. |
| **Access** | Typically used within the same system or application. | Accessible over the internet by different systems or services. |
| **Data Format** | Data interchange formats can vary (e.g., binary, JSON). | Typically uses JSON or XML for data interchange. |
| **Examples** | - Library API (e.g., Math API in Java) | - REST API (e.g., weather service API) |
| | - OS API (e.g., Windows API) | - SOAP API (e.g., payment gateway API) |
| **Usage** | Often used for integrating different components within the same application or system. | Used to connect and interact with web-based services and external systems. |
| **Authentication** | May use various methods depending on the context. | Commonly uses web authentication methods like OAuth. |
| **Data Exchange** | Often used within applications or between local components. | Facilitates data exchange over the web between different applications or services. |

This table highlights key differences and helps in understanding how APIs and Web APIs are used in various contexts.




The image you've provided highlights the basic difference between an API (Application Programming Interface) and a Web API:

1. **API (Application Programming Interface):**


Khushal Savalakha

- An API is a set of rules and protocols that allows different software applications to communicate with each other.
- It is used to access the functionality of an application or a service.
- In the image, the API connects a mobile device (or another application) to an application, which then interacts with a database. This setup is common in software where the API is not necessarily web-based, and the communication can occur within a single system or across different systems without relying on the web.

2. **Web API:**
   - A Web API is a specific type of API that is accessible over the web using HTTP/HTTPS protocols.
   - It enables different web services or applications to interact over the internet. The Web API is often used in web development to allow communication between a client (like a web browser or a mobile app) and a web server.
   - In the image, the Web API connects a mobile device to a web application, which then interacts with a database. This emphasizes that the communication occurs over the web, typically using REST or SOAP protocols.

**Key Differences:**

- **Communication Protocol:** APIs can use a variety of protocols (e.g., function calls, messaging), whereas Web APIs specifically use web protocols like HTTP/HTTPS.
- **Accessibility:** Web APIs are accessible over the internet, while other APIs may only be accessible within a local environment or network.
- **Use Case:** APIs might be used for general application communication within a system, while Web APIs are designed specifically for web-based interactions.

If you have more questions or need further details, feel free to ask!

> 💡 it means API is for interact library and software and web API includes interaction with application like using HTTP/HTTPS protocols here app has not that types of protocol so here REST API is working as mediator am i true ?

You're on the right track, but let's clarify a few points to ensure a better understanding:

1. **API (General):**


Khushal Savalakha

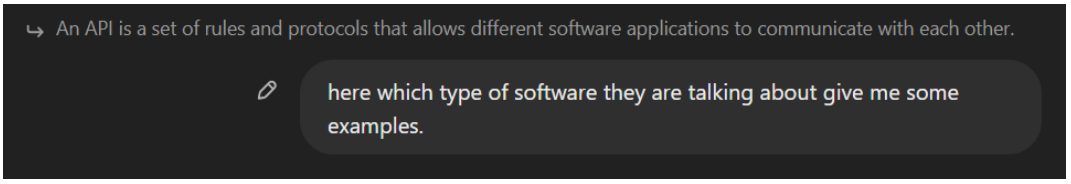
- An API is indeed a way for different software components, libraries, or systems to interact with each other. It defines the methods and data structures that developers can use to interact with the software, service, or library. APIs aren't limited to a specific protocol—they can be based on various mechanisms like function calls, messaging, or even file-based communication.

2. **Web API:**
   - A Web API specifically uses web protocols (like HTTP/HTTPS) to allow different systems to communicate over the internet. This type of API is typically used when you need to expose the functionality of your application to other applications or services over the web.
   - **REST API** (which is a type of Web API) is based on REST (Representational State Transfer) principles and is one of the most common types of Web APIs. REST APIs use standard HTTP methods (GET, POST, PUT, DELETE) and are stateless, meaning each call from the client to the server must contain all the necessary information for the server to fulfill the request.
3. **REST API as a Mediator:**
   - When you say "REST API is working as a mediator," you're correct in a sense. A REST API often acts as an intermediary between a client (like a mobile app, web browser, or another server) and a backend service (like a database or another service). The REST API processes requests from the client, interacts with the backend service, and then returns the appropriate response to the client.

So, to summarize:

- **API**: A broader term for any interface that allows interaction between software components.
- **Web API**: A type of API accessible over the web using HTTP/HTTPS.
- **REST API**: A specific type of Web API that follows REST principles, often acting as an intermediary between a client and a server application.

This means that in a web environment, the Web API (including REST APIs) serves as a bridge for communication between the client and the server, using standard web protocols.





Khushal Savalakha

When the explanation mentions that APIs allow different software applications to communicate with each other, it refers to various types of software. Here are some examples:

1. **Operating Systems**:
   - **Windows API**: Allows software to interact with the Windows operating system, like handling files, processes, and other OS-level operations.
   - **POSIX API**: Used in Unix-like operating systems, enabling software to manage tasks such as input/output, file handling, and process control.
2. **Libraries and Frameworks**:
   - **OpenGL**: A graphics API that enables software to render 2D and 3D vector graphics, often used in video games and simulations.
   - **TensorFlow**: A machine learning library API that allows developers to create and deploy machine learning models.
3. **Database Management Systems**:
   - **SQL API**: Provides a way for software to interact with databases, like retrieving, inserting, updating, or deleting data from a database system like MySQL, PostgreSQL, or Oracle.
   - **MongoDB API**: Allows applications to interact with MongoDB databases to perform operations such as querying, inserting, or updating documents.
4. **Software Development Kits (SDKs)**:
   - **Android SDK**: Provides APIs for developers to build applications for Android devices, allowing the app to interact with the device's hardware, like the camera or GPS.
   - **iOS SDK**: Contains APIs for developing iOS applications, enabling interaction with the device’s functionalities, such as touch input, sensors, or networking.
5. **Web APIs**:
   - **Twitter API**: Allows software to interact with Twitter, enabling actions like posting tweets, reading timelines, or managing accounts.
   - **Google Maps API**: Lets developers integrate maps and location data into their applications, providing services like geocoding, directions, and map embedding.

These examples show how APIs act as intermediaries, allowing different types of software to work together or integrate with each other to extend functionality or interact

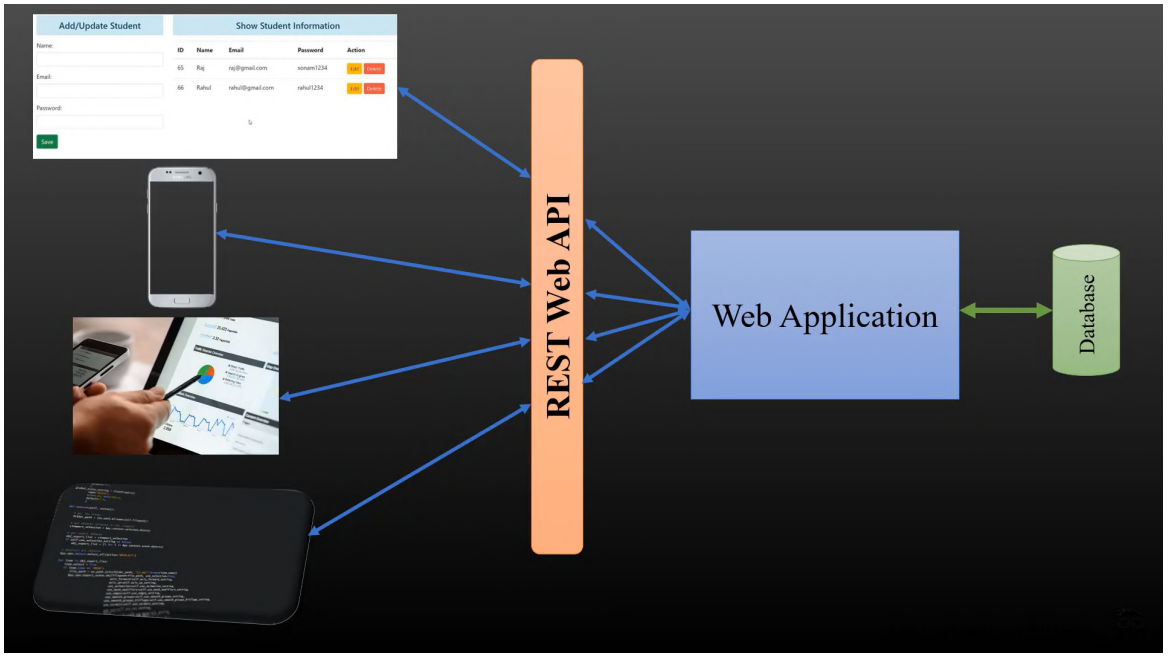

Khushal Savalakha

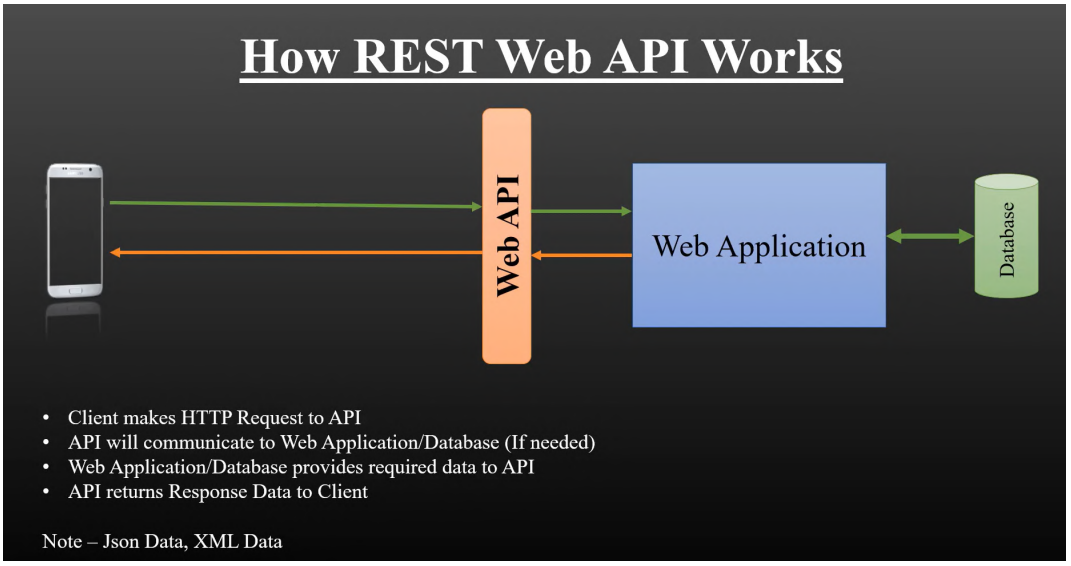
with external systems.

## ▼ REST AND REST API

REST:It is an architectural guideline to develop web API.

REST API:The API which is developed using REST is known  as REST API/RESTful API.



# REST API

CRUD Operations:-

| Operation | HTTP Methods | Description |
|---|---|---|
| Create | POST | Creating/Posting/Inserting Data |
| Read | GET | Reading/Getting/Retrieving Data |
| Update | PUT, PATCH | Updating Data<br>Complete Update - PUT<br>Partial Update - PATCH |
| Delete | DELETE | Deleting Data |

# Students API Resource

http://geekyshows.com/api/students

1 2 3

1. Base URL
2. Naming Convention
3. Resource of API or End-Point

# Request - Response

- Request for Updating Data, id = 1

**Request**

PUT or PATCH: /api/students/1

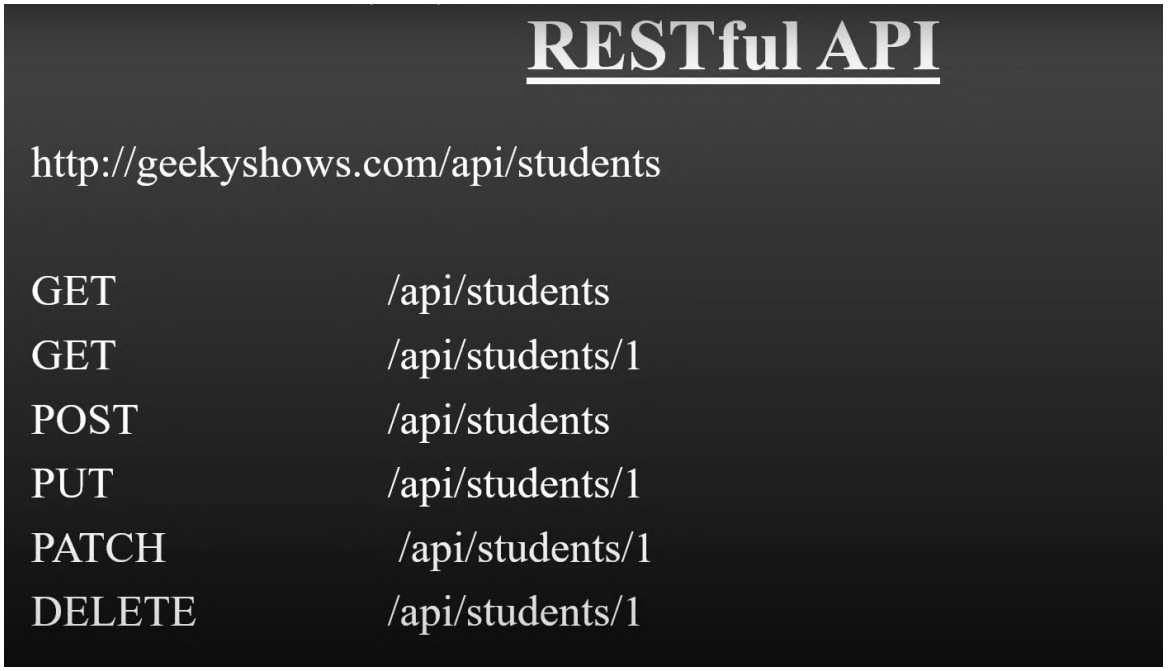
{"name": "Raj" }

**Response**

```
[
  {"id":1, "name": "Raj" }
]
```



Khushal Savalakha




💡 what is REST and REST api explain simply and depthly with example ?

## What is REST?

**REST** (Representational State Transfer) is an architectural style used in designing networked applications. It relies on a stateless, client-server communication model where interactions are based on standard HTTP methods like GET, POST, PUT, DELETE, etc. REST is not a protocol but a set of guidelines or principles that guide the design of web services.

**Key Principles of REST:**

1. **Statelessness**: Each request from a client to a server must contain all the information needed to understand and process the request. The server doesn't store any client context between requests. This means each request is independent and self-contained.
2. **Client-Server Architecture**: The client and server are separated and interact through a uniform interface. This separation allows each to evolve independently.
3. **Uniform Interface**: REST uses standard HTTP methods and status codes, making it simpler to understand and use. This interface simplifies interactions between systems.
4. **Layered System**: A REST system can be composed of multiple layers, with each layer having a specific role, such as handling security, caching, or load balancing, without affecting the client-server interaction.


Khushal Savalakha

5. **Cacheable**: Responses from the server can be marked as cacheable or non-cacheable, improving efficiency by allowing clients to reuse responses.
6. **Resource-Based**: REST treats server data as resources that can be created, read, updated, or deleted. Each resource is identified by a unique URL.

## What is a REST API?

A **REST API** (RESTful API) is an API that conforms to the principles of REST. It allows interaction with web services by using HTTP methods to perform CRUD (Create, Read, Update, Delete) operations on resources.

## Example of a REST API

Let's say you have a REST API for managing a collection of books in a library system. Here's how you might interact with it:

1. **GET Request (Retrieve a List of Books)**:
   - **Request**: `GET /api/books`
   - **Description**: This request fetches a list of all books from the server.
   - **Response**: A JSON array containing all book details.
2. **GET Request (Retrieve a Specific Book)**:
   - **Request**: `GET /api/books/1`
   - **Description**: This request fetches the details of a book with the ID 1.
   - **Response**: A JSON object containing details of the book with ID 1.
3. **POST Request (Create a New Book)**:
   - **Request**: `POST /api/books`
   - **Description**: This request creates a new book in the system.
   - **Body**: JSON data containing details of the book, like title, author, etc.
   - **Response**: A JSON object of the newly created book, including its ID.
4. **PUT Request (Update a Book)**:
   - **Request**: `PUT /api/books/1`
   - **Description**: This request updates the details of the book with ID 1.
   - **Body**: JSON data with the updated information (e.g., new title, author).


Khushal Savalakha

- **Response**: A JSON object with the updated book details.

5. **DELETE Request (Delete a Book)**:
   - **Request**: `DELETE /api/books/1`
   - **Description**: This request deletes the book with ID 1 from the system.
   - **Response**: Typically, no content or a success status.

## Real-World Example: GitHub REST API

GitHub provides a RESTful API for interacting with its platform. For instance:

- **GET /repos/{owner}/{repo}**: Retrieves information about a specific repository.
- **POST /repos/{owner}/{repo}/issues**: Creates a new issue in a repository.
- **PATCH /repos/{owner}/{repo}/issues/{issue_number}**: Updates an existing issue.

By following the REST principles, GitHub ensures that its API is easy to use, understand, and integrate with different clients and systems.

## Summary

REST is a set of principles for designing networked applications, and a REST API is an implementation of these principles using HTTP methods. REST APIs are widely used because they are simple, flexible, and scalable, making them ideal for web services.

💡 then what about PATCH ?

## What is the PATCH Method?

The **PATCH** method in HTTP is used to apply partial modifications to a resource. Unlike the PUT method, which replaces the entire resource with the updated content, PATCH is more efficient when only a few fields need to be updated.

## How PATCH Works

- **Purpose**: PATCH is used when you only need to update part of a resource, rather than the entire resource. This can save bandwidth and processing time.
- **Request Format**: The request typically includes the specific changes you want to apply. This could be in the form of a JSON object containing only the fields you wish to update.


Khushal Savalakha

- **Idempotency**: PATCH is not necessarily idempotent, meaning that applying the same PATCH request multiple times could yield different results (though in practice, many implementations do strive to make PATCH idempotent).

## Example of PATCH

Let's say you have a user profile resource:

- **Resource**: `/api/users/1`
- **Current Data**:

```
{
    "username": "john_doe",
    "email": "john@example.com",
    "firstName": "John",
    "lastName": "Doe"
}
```

Now, you want to update only the `email` and `firstName` fields:

- **PATCH Request**:

```
PATCH /api/users/1
Content-Type: application/json

{
    "email": "john.new@example.com",
    "firstName": "Johnny"
}
```

- **Result**: After the PATCH request, the resource may look like this:

```
{
    "username": "john_doe",
    "email": "john.new@example.com",
    "firstName": "Johnny",
    "lastName": "Doe"
}
```


Khushal Savalakha

### When to Use PATCH vs. PUT

- **PATCH** is ideal when you need to update just a few fields in a resource.
- **PUT** is better when you need to completely replace the resource with a new version.

### Real-World Example: GitHub API

GitHub uses PATCH for updating specific properties of a resource. For instance, you can update an issue title or body in a repository:

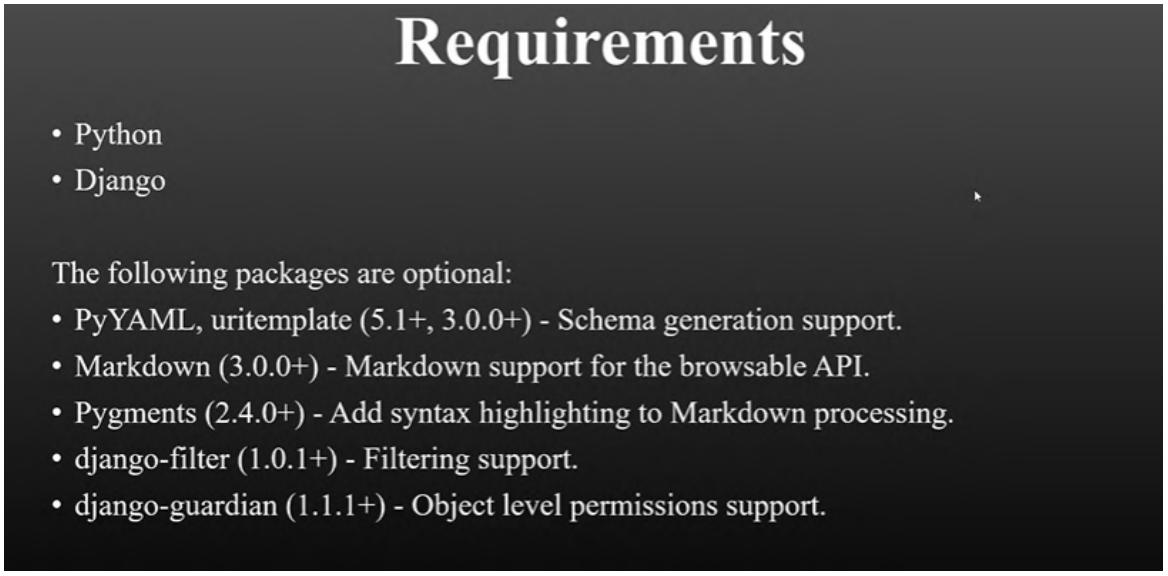
- **PATCH /repos/{owner}/{repo}/issues/{issue_number}**:
  This request allows you to modify just the title or body of an issue without affecting the other fields.

### Summary

PATCH is a powerful HTTP method for making partial updates to a resource. It’s more efficient than PUT when only specific parts of a resource need to be updated. This makes it useful in many real-world applications, like user profile updates or modifying specific fields in a database entry.

## ▼ Django REST Framework(DRF) Installation and Explanation

Khushal Savalakha




## ▼ Serializer and Serializer Class


Khushal Savalakha

# Python JSON

Python has a built in package called json, which is used to work with json data.

dumps(data) – This is used to convert python object into json string.

Example:-

To use json package First we have to import it.

```
import json
python_data = {'name': 'Sonam', 'roll':101 }
json_data = json.dumps(python_data)
print(json_data)
{"name" : "Sonam", "roll" : 101}
```

# Python JSON

loads(data) – This is used to parse json string.

Example:-

```
import json
json_data = {"name" : "Sonam", "roll" : 101}
parsed_data = json.loads(json_data)
print(parsed_data)
{'name' : 'Sonam', 'roll' : 101}
```

P → J = dumps()
J → P = loads

The `json.loads()` function in Python is used to parse a JSON (JavaScript Object Notation) formatted string and convert it into a corresponding Python object, such as a dictionary or list. The `loads()` function is part of the `json` module, which provides methods for working with JSON data.

## Key Points About `json.loads()`:

1. **JSON to Python Conversion:**
   - The primary purpose of `json.loads()` is to take a JSON string (which is essentially text) and convert it into a Python object that you can work with programmatically.


Khushal Savalakha

- For example, a JSON string representing a dictionary can be converted into a Python dictionary.

2. **Common Use Cases:**
   - **API Responses:** When working with web APIs, data is often returned in JSON format. You can use `json.loads()` to convert this data into Python objects for further processing.
   - **Reading JSON from Files:** When you read JSON data from a file as a string, you can use `json.loads()` to parse it into a Python object.
3. **Syntax:**

```
import json

json_string = '{"name": "Khushal", "age": 21, "skill
s": ["Python", "Java"]}'
python_object = json.loads(json_string)
```

   - Here, `python_object` will be a Python dictionary: `{'name': 'Khushal', 'age': 21, 'skills': ['Python', 'Java']}`.
4. **Difference from `json.load()`:**
   - `json.loads()` is used to parse a JSON string.
   - `json.load()` is used to parse JSON data from a file object.

   Example:

```
import json

with open('data.json', 'r') as file:
    data = json.load(file)  # Parses the JSON data fr
om the file into a Python object
```

## Why Use `json.loads()`?

1. **Data Interchange:** JSON is a common format for data interchange between systems, especially in web applications. `json.loads()` allows you to convert JSON data received as a string into native Python data types.


Khushal Savalakha

2. **Ease of Use:** Once the JSON string is converted to a Python object, you can easily access and manipulate the data using Python's built-in data structures like dictionaries and lists.
3. **Serialization and Deserialization:** JSON is often used to serialize and deserialize data. `json.loads()` is the deserialization step, converting JSON strings back into Python objects.

## Example in Context:

Suppose you receive the following JSON string from an API:

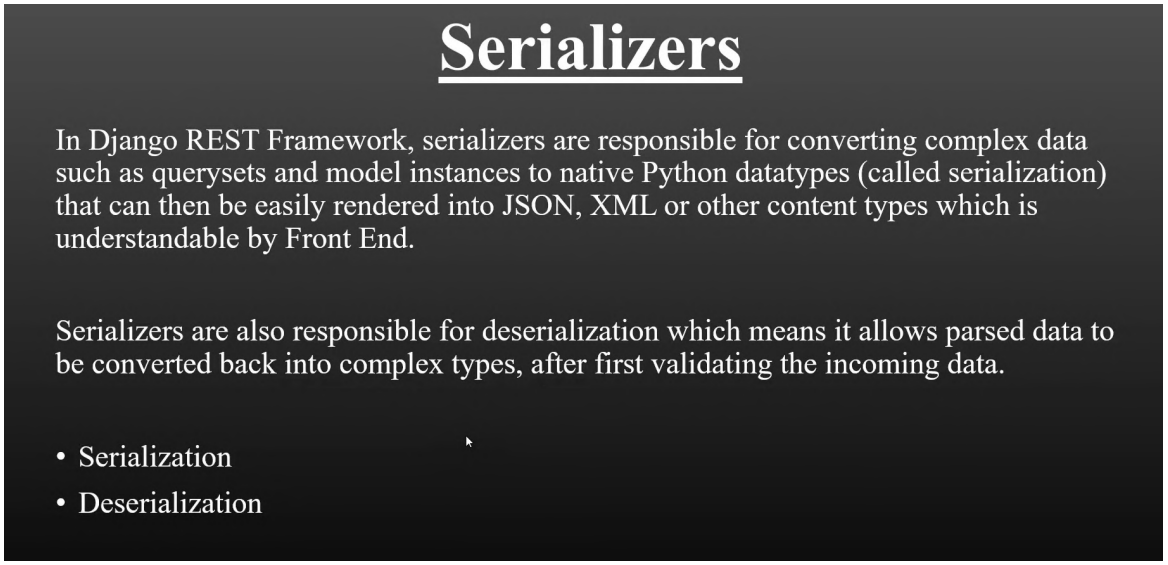
```
{
    "name": "Khushal",
    "age": 21,
    "languages": ["Python", "JavaScript"]
}
```

You can parse this JSON string into a Python dictionary using `json.loads()`:

```
import json

json_string = '{"name": "Khushal", "age": 21, "language
s": ["Python", "JavaScript"]}'
data = json.loads(json_string)

print(data['name'])  # Output: Khushal
print(data['languages'])  # Output: ['Python', 'JavaScri
pt']
```

Now, `data` is a Python dictionary that you can work with just like any other dictionary.


Khushal Savalakha




The image provides an overview of **serializers** in the Django REST Framework, explaining their role in converting data between complex types like querysets and model instances into formats that can be easily rendered (like JSON or XML) and vice versa.

## Explanation of Serializers:

1. **Serialization:**
   - Serialization refers to the process of converting complex data types, such as Django querysets and model instances, into native Python data types like dictionaries or lists. These data types can then be easily converted into JSON, XML, or other content types that can be sent to the front end.
   - **Example:**
     Suppose you have a Django model instance representing a user:

     ```
     class User(models.Model):
         username = models.CharField(max_length=100)
         email = models.EmailField()
     ```

     If you have an instance of this model:

     ```
     user_instance = User(username='Khushal', email='kh
     ushal@example.com')
     ```

     You can serialize this instance into JSON using a serializer:


Khushal Savalakha

```
from rest_framework import serializers

class UserSerializer(serializers.Serializer):
    username = serializers.CharField(max_length=10
0)
    email = serializers.EmailField()

serializer = UserSerializer(user_instance)
json_data = serializer.data  # This will be a Pyth
on dictionary: {'username': 'Khushal', 'email': 'k
hushal@example.com'}
```

You can then convert this dictionary to a JSON string:

```
import json
json_string = json.dumps(json_data)
# Result: '{"username": "Khushal", "email": "khush
al@example.com"}'
```

2. **Deserialization:**

- Deserialization is the reverse process where data received in JSON or another format is converted back into complex Python types (e.g., Django model instances). During this process, the data is validated to ensure it meets certain criteria before being accepted.

- **Example:**
  Consider you receive JSON data from an API request:

```
{
    "username": "Khushal",
    "email": "khushal@example.com"
}
```

You can deserialize this data to create or update a `User` instance:

```
from rest_framework.parsers import JSONParser
from io import BytesIO
```


Khushal Savalakha

```
json_data = '{"username": "Khushal", "email": "khu
shal@example.com"}'
stream = BytesIO(json_data.encode('utf-8'))
data = JSONParser().parse(stream)

serializer = UserSerializer(data=data)
if serializer.is_valid():
    user_instance = User(**serializer.validated_da
ta)
    # user_instance now has the username and email
from the JSON data
```

In this example, `serializer.is_valid()` checks if the incoming data adheres to the required format and types. If it does, the data is deserialized into a Python object, such as a Django model instance.

## Key Takeaways:

- **Serialization** makes it possible to convert complex data types into simpler formats like JSON, which can be easily sent to clients.
- **Deserialization** allows you to take data received from clients (like JSON) and convert it into complex Python objects, validating the data in the process.

These operations are essential in building APIs with Django REST Framework, where data is often exchanged between a front-end client and the back-end server.

## Serializer Class

A serializer class is very similar to a Django Form and ModelForm class, and includes similar validation flags on the various fields, such as required, max_length and default.

DRF provides a Serializer class which gives you a powerful, generic way to control the output of your responses, as well as a ModelSerializer class which provides a useful shortcut for creating serializers that deal with model instances and querysets.


Khushal Savalakha

# How to Create Serializer Class

- Create a separate seriealizers.py file to write all serializers

```
from rest_framework import serializers
class StudentSerializer(serializers.Serializer):
    name = serializers.CharField(max_length=100)
    roll = serializers.IntegerField()
    city = serializers.CharField(max_length=100)
```

**models.py**

```
from django.db import models
class Student(models.Model):
    name = models.CharField(max_length=100)
    roll = models.IntegerField()
    city = models.CharField(max_length=100)
```

Run makemigrations and migrate command

| ID | NAME | ROLL | CITY |
|---|---|---|---|
| 1 | Sonam | 101 | Ranchi |
| 2 | Rahul | 102 | Ranchi |
| 3 | Raj | 103 | Bokaro |

JSON Data

| ID | NAME | ROLL | CITY |
|---|---|---|---|
| 1 | Sonam | 101 | Ranchi |
| 2 | Rahul | 102 | Ranchi |
| 3 | Raj | 103 | Bokaro |

Model Object 1
Model Object 2
Model Object 3

Complex DataType → Serialization → Python Native DataType → Render into Json → Json Data

Sure! Here are examples of complex and native Python data types:

- **Complex Data Type:**
  - Django Model Instance (e.g., a `User` object representing a row in a database).
  - QuerySet (e.g., a collection of `User` objects retrieved from the database).


Khushal Savalakha

- DateTime object (e.g., a specific date and time, like `2024-08-26 14:30:00`).

- **Python Native Data Type:**
  - Dictionary (e.g., `{"username": "Khushal", "email": "khushal@example.com"}`).
  - List (e.g., `["apple", "banana", "cherry"]`).
  - String (e.g., `"Hello, World!"`).

# Serialization

The process of converting complex data such as querysets and model instances to native Python datatypes are called as Serialization in DRF.

- Creating model instance stu

  stu = Student.objects.get(id = 1)

- Converting model instance stu to Python Dict / Serializing Object

  serializer = StudentSerializer(stu)

# serializer.data

This is the serialized data.

serializer.data

# JSONRenderer

This is used to render Serialized data into JSON which is understandable by Front End.

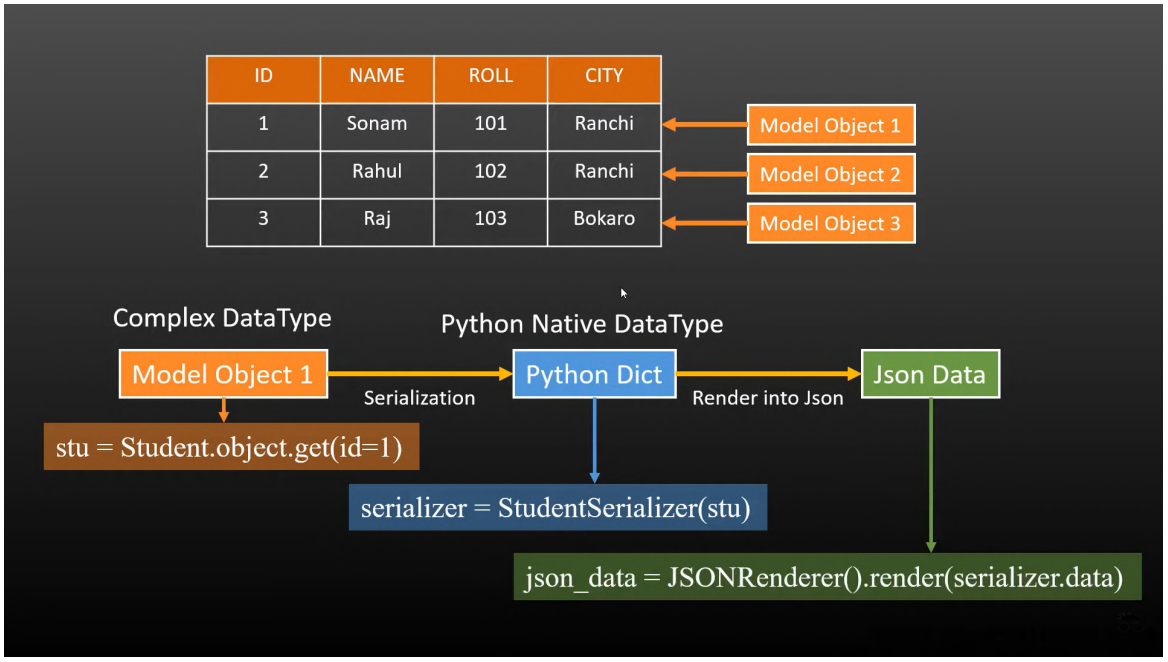
Importing JSONRenderer

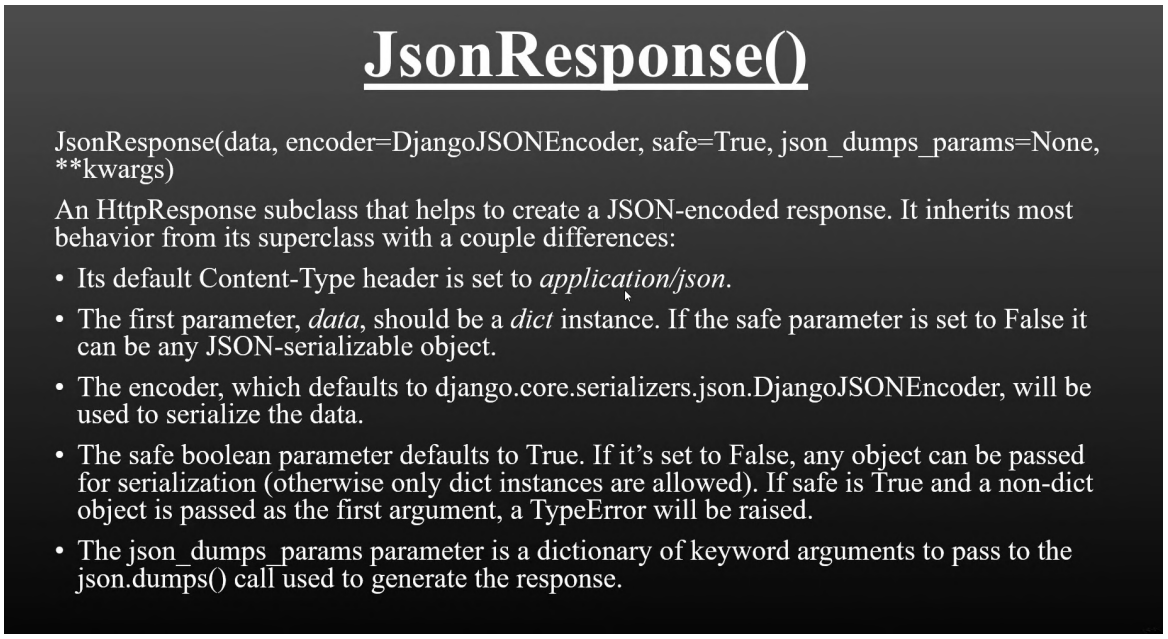
from rest_framework.renderers import JSONRenderer

Render the Data into Json

json_data = JSONRenderer().render(serializer.data)


Khushal Savalakha




The image provides an overview of the `JsonResponse` class in Django, which is a subclass of `HttpResponse`. Here's a simple explanation:

## What is `JsonResponse`?

- **Purpose**: `JsonResponse` is used in Django to create an HTTP response that contains JSON-encoded data. It's particularly useful when building APIs where the server needs to send data back to the client in JSON format.

## Key Points:


Khushal Savalakha

1. **Content-Type Header**:
   - When you use `JsonResponse`, the response automatically has its `Content-Type` set to `application/json`, which tells the client that the response is in JSON format.
2. **Data Parameter**:
   - The first parameter, `data`, is where you provide the data you want to return as JSON. This should be a dictionary (`dict`) by default.
3. **Safe Parameter**:
   - By default, `safe=True`, which means that only dictionaries can be passed as data. If you set `safe=False`, you can pass any JSON-serializable object (like a list or a string). However, if you pass something other than a dictionary without setting `safe=False`, you'll get an error.
4. **Encoder**:
   - The `encoder` parameter, which defaults to `DjangoJSONEncoder`, is responsible for converting the Python data into JSON.
5. **json_dumps_params**:
   - This parameter allows you to pass additional options to the underlying `json.dumps()` function, giving you more control over how the JSON is generated.

## Example:

If you want to send a response with a dictionary containing user information, you would use `JsonResponse` like this:

```
return JsonResponse({"username": "Khushal", "email": "kh
ushal@example.com"})
```

This would return a JSON response with the content:

```
{
  "username": "Khushal",
  "email": "khushal@example.com"
}
```

The response will have the correct `Content-Type` header, and the data will be properly formatted in JSON.


Khushal Savalakha

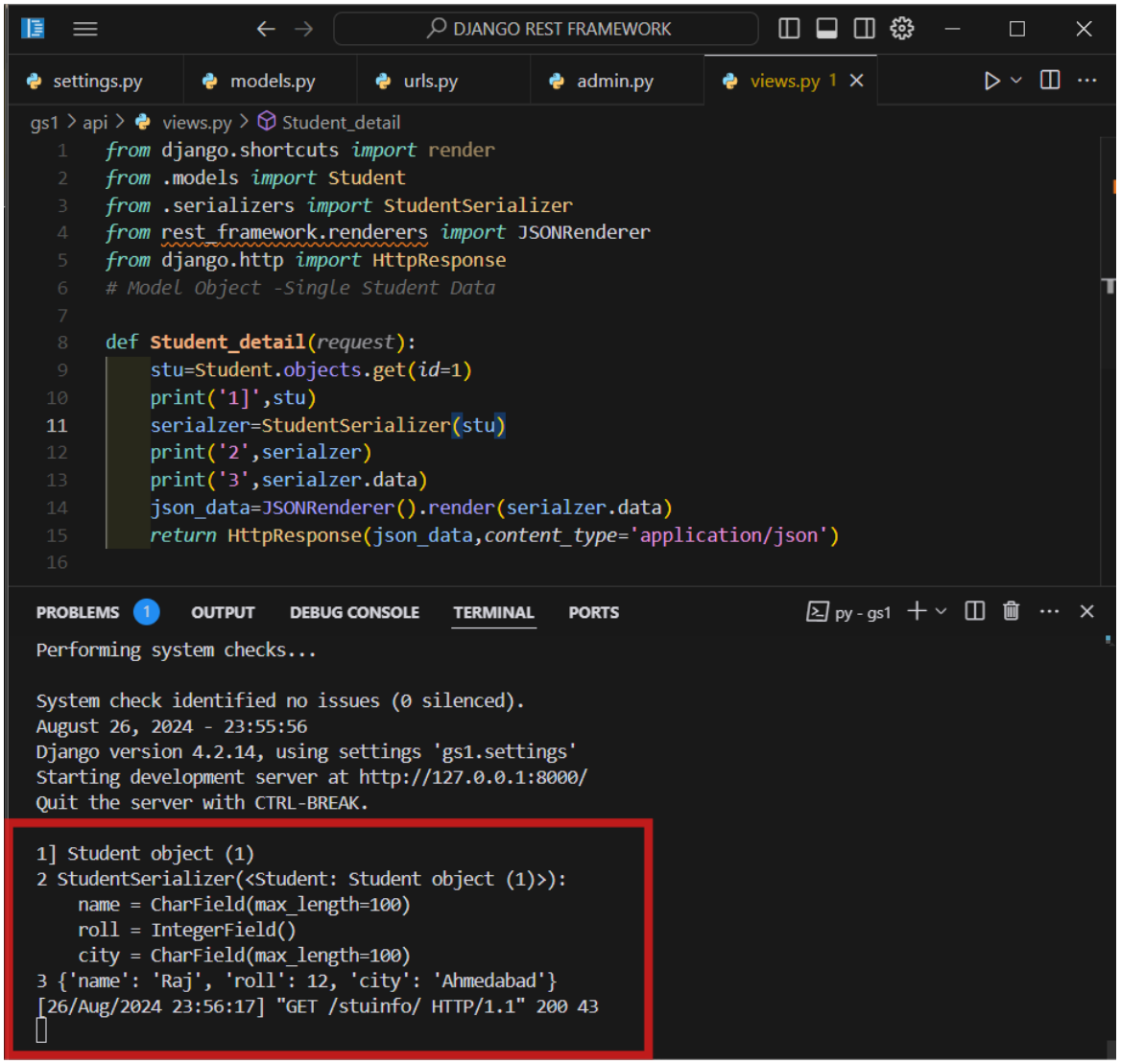
```python
from rest_framework import serializers
class StudentSerializer(serializers.Serializer):
    name = serializers.CharField(max_length=100)
    roll = serializers.IntegerField()
    city = serializers.CharField(max_length=100)


class STudentForm(forms.Form):
    name = forms.CharField(max_length=100)
    roll = forms.IntegerField()
    city = forms.CharField(max_length=100)
```

```python
from django.shortcuts import render
from .models import Student
from .serializers import StudentSerializer
from rest_framework.renderers import JSONRenderer
from django.http import HttpResponse
# Model Object -Single Student Data

def Student_detail(request):
    stu=Student.objects.get(id=1)
    serializer=StudentSerializer(stu)
    json_data=JSONRenderer().render(serializer.data)
    return HttpResponse(json_data,content_type='application/json')
```

```python
om rest_framework import serializers

ass StudentSerializer(serializers.Serializer):
    name=serializers.CharField(max_length=100)
    roll=serializers.IntegerField()
    city=serializers.CharField(max_length=100)
```

```
{"name":"Raj","roll":12,"city":"Ahmedabad"}
```

```
    stu=Student.object.get(id=1)
AttributeError: type object 'Student' has no attribute 'object'
[26/Aug/2024 23:50:34] "GET /stuinfo/ HTTP/1.1" 500 66045
D:\DJANGO REST FRAMEWORK\gs1\api\views.py changed, reloading.
Watching for file changes with StatReloader
Performing system checks...

System check identified no issues (0 silenced).
August 26, 2024 - 23:52:23
Django version 4.2.14, using settings 'gs1.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.

[26/Aug/2024 23:52:24] "GET /stuinfo/ HTTP/1.1" 200 43
```




```
from django.shortcuts import render
from .models import Student
from .serializers import StudentSerializer
from rest_framework.renderers import JSONRenderer
from django.http import HttpResponse
# Model Object -Single Student Data

def Student_detail(request,pk):
    stu=Student.objects.get(id=pk)
    # print('1]',stu)
    serialzer=StudentSerializer(stu)
    # print('2',serialzer)
    # print('3',serialzer.data)
```


Khushal Savalakha

```
    json_data=JSONRenderer().render(serialzer.data)
    return HttpResponse(json_data,content_type='applicatior
```

Yes, you're correct! In the provided Django view, you are dealing with complex data (a single `Student` object, which is a model instance).

1. **Serializer**: The `StudentSerializer` converts this complex data into a Python native data type, specifically a dictionary.
2. **JSONRenderer**: Then, the `JSONRenderer` converts this dictionary into JSON format.
3. **HttpResponse**: Finally, the JSON data is sent back to the user as an HTTP response with the `application/json` content type.

This process is essential for ensuring that the data from your Django application can be easily consumed by frontend applications or other APIs.

💡 what is difference between model instance and querySet ?

In Django, both model instances and querysets are used to interact with the database, but they serve different purposes and represent data in different forms. Here's a breakdown of the differences:

## 1. Model Instance:

- **Definition:** A model instance is a single, specific record in the database. It's an object of the model class.
- **Representation:** It represents one row of data from a database table.
- **Usage:** You typically work with a model instance when you want to retrieve, update, or delete a specific record in the database.
- **Example:** If you have a model named `Student`, a model instance would represent one student. For example:
  Here,
  `student` is a model instance representing the student with `id=1`.

  ```
  student = Student.objects.get(id=1)
  ```

## 2. QuerySet:


Khushal Savalakha

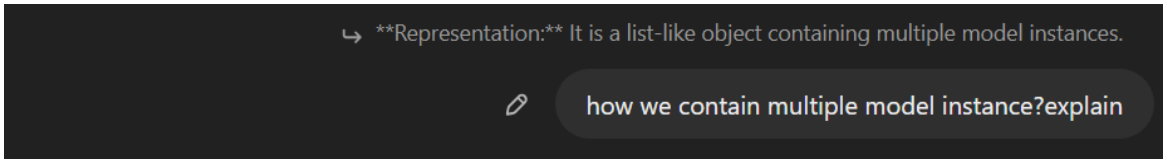
- **Definition:** A queryset is a collection of multiple model instances, representing zero, one, or many rows of data from the database.
- **Representation:** It is a list-like object containing multiple model instances.
- **Usage:** You use a queryset when you need to retrieve multiple records from the database, perform filtering, or execute bulk operations.
- **Example:** A queryset can be used to get all students or filter them based on certain criteria:
  Here,
  `students` is a queryset containing all students who are 20 years old. Even if it contains only one student, it's still considered a queryset.

  ```
  students = Student.objects.filter(age=20)
  ```

## Key Differences:

- **Number of Records:** A model instance represents a single record, while a queryset can represent multiple records (or none).
- **Operations:** With a model instance, you perform operations like updating or deleting a specific record. With a queryset, you can perform bulk operations on multiple records at once.
- **Methods:** Model instances have methods for working with a single record (e.g., `save()`, `delete()`), while querysets have methods for working with multiple records (e.g., `filter()`, `exclude()`).

Understanding the distinction between a model instance and a queryset is crucial for effectively managing and querying data in Django applications.




In Django, when you want to handle multiple records from the database, you use a **QuerySet**. A QuerySet is essentially a collection of multiple **model instances**. Think of it as a list-like object that contains several individual records, where each record is an instance of a model.

## How QuerySets Contain Multiple Model Instances


Khushal Savalakha

When you perform a query in Django to retrieve multiple records from the database, the result is a QuerySet. Each item in this QuerySet is a model instance representing one record from the database. Here's how it works:

## Example:

Suppose you have a `Student` model in your Django project, which represents students in a database table:

```
from .models import Student
```

Now, let's say you want to get all students from the database:

```
students = Student.objects.all()
```

- `students` is a QuerySet.
- Each item in `students` is a model instance of the `Student` model.
- If there are 10 students in the database, the QuerySet `students` will contain 10 model instances, one for each student.

You can loop through this QuerySet to access each model instance individually:

```
for student in students:
    print(student.name, student.age)  # Accessing attrib
utes of each model instance
```

## Key Points:

- **QuerySet:** Acts like a list that holds multiple model instances.
- **Model Instance:** Each item in the QuerySet is an individual model instance, representing one record in the database.
- **Operations:** You can iterate over a QuerySet, filter it, or perform bulk actions on it.

## Filtering Example:

If you want to filter students who are 20 years old:

```
students_20 = Student.objects.filter(age=20)
```


Khushal Savalakha

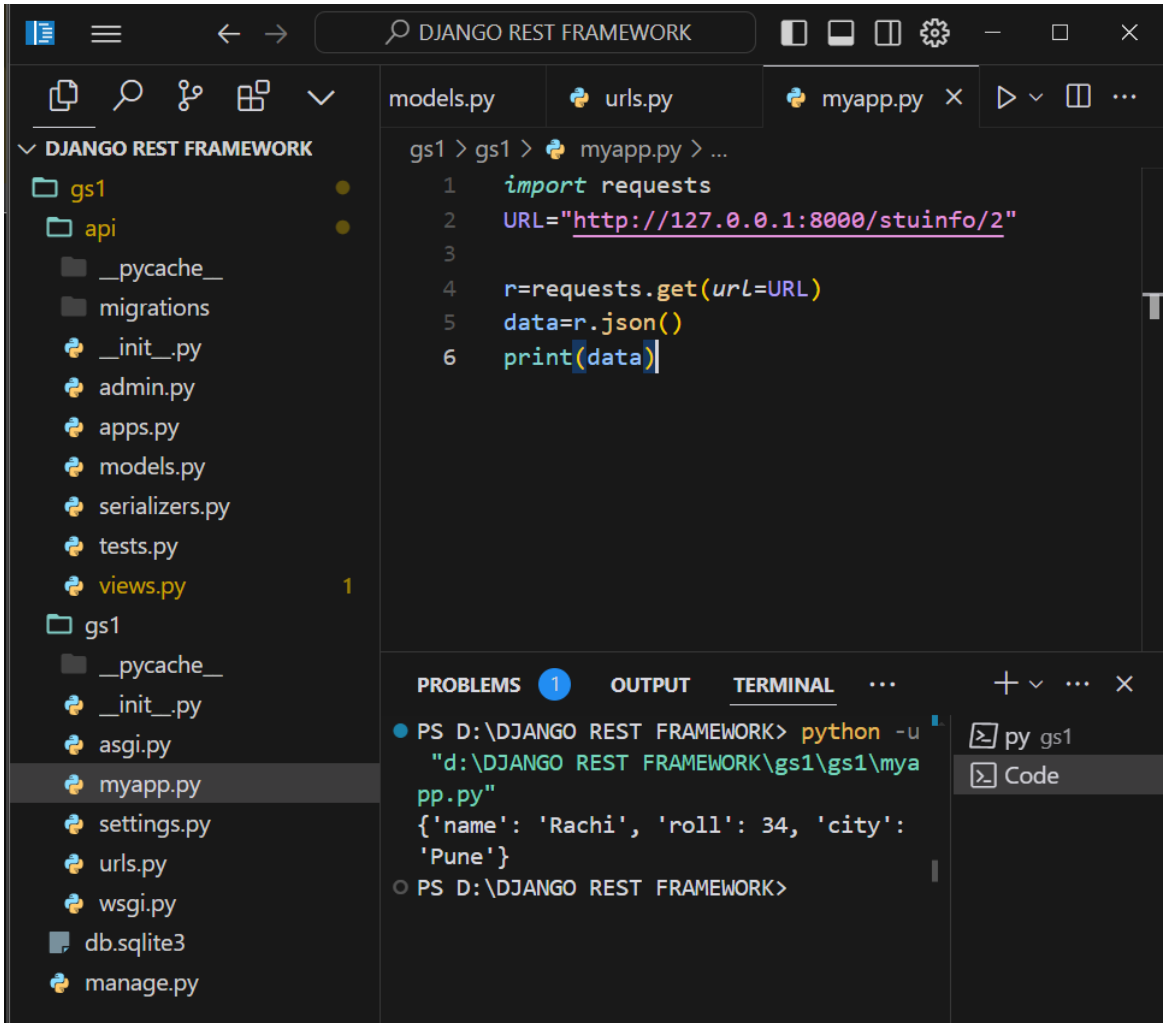
- `students_20` is a QuerySet containing all `Student` model instances where the `age` attribute is 20.

Each model instance in the QuerySet can be treated as a regular Python object, and you can interact with it just like any other object in your code.

This is how Django allows you to manage and interact with multiple records efficiently using QuerySets.

myapp.py

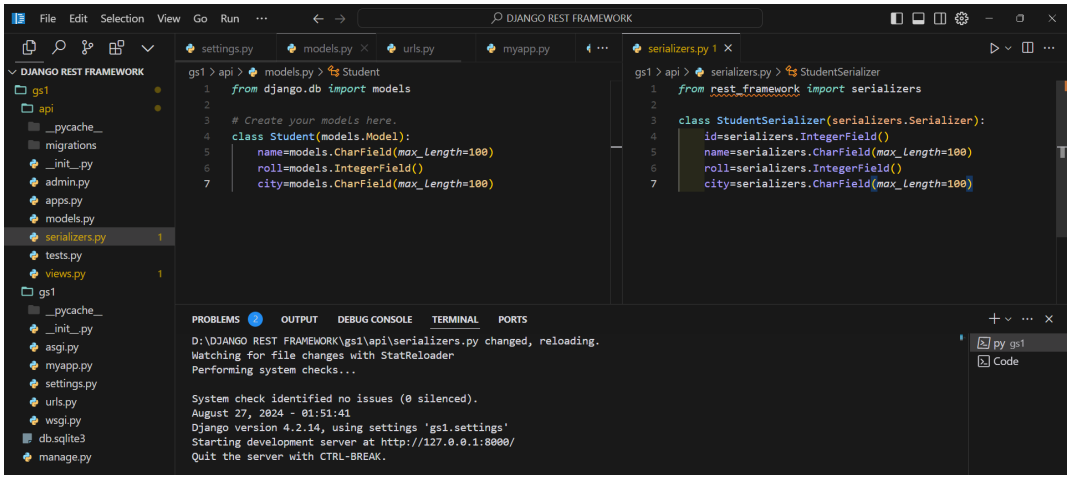
```
import requests
URL="http://127.0.0.1:8000/stuinfo/2"

r=requests.get(url=URL)
data=r.json()
print(data)
```





Khushal Savalakha

```
from django.shortcuts import render
from .models import Student
from .serializers import StudentSerializer
from rest_framework.renderers import JSONRenderer
from django.http import HttpResponse,JsonResponse
# Model Object -Single Student Data
def Student_detail(request,pk):
    stu=Student.objects.get(id=pk)
    serializer=StudentSerializer(stu)
    return JsonResponse(serializer.data)

#Query Set-All Student Data
def Student_list(request):
    stu=Student.objects.all()
    serializer=StudentSerializer(stu,many=True)
    #non dict data (dict data in form of list like [{},{}]
    return JsonResponse(serializer.data,safe=False)
```




explain me here how serializer flow works here with models.py and views.py file?

In the Django project setup you provided, the `serializers.py`, `models.py`, and `views.py` files work together to convert model instances into JSON data and send them as a response to the client. Here's how the flow works:


Khushal Savalakha

## 1. `models.py` (Defining the Data Structure)

- You define a `Student` model, which represents a student in the database.
- This model has fields like `name`, `roll`, and `city`.

```
from django.db import models

class Student(models.Model):
    name = models.CharField(max_length=100)
    roll = models.IntegerField()
    city = models.CharField(max_length=100)
```

- When a `Student` object is created, it corresponds to a record in the database with the specified fields.

## 2. `serializers.py` (Transforming the Model into JSON-serializable Format)

- You define a `StudentSerializer`, which is responsible for converting the `Student` model instances into a format that can be rendered into JSON.
- The serializer defines how each field in the `Student` model should be represented in the serialized output.

```
from rest_framework import serializers

class StudentSerializer(serializers.Serializer):
    id = serializers.IntegerField()
    name = serializers.CharField(max_length=100)
    roll = serializers.IntegerField()
    city = serializers.CharField(max_length=100)
```

- Each field in the `StudentSerializer` matches the corresponding field in the `Student` model.

## 3. `views.py` (Handling the Request and Response)

- In the `Student_detail` view, you fetch a single `Student` instance from the database using `Student.objects.get(id=pk)`.
- You then serialize this instance using the `StudentSerializer`.


Khushal Savalakha

```
from django.shortcuts import render
from .models import Student
from .serializers import StudentSerializer
from rest_framework.renderers import JSONRenderer
from django.http import HttpResponse

def Student_detail(request, pk):
    stu = Student.objects.get(id=pk)  # Fetches the Stud
ent instance by ID
    serializer = StudentSerializer(stu)  # Serializes th
e instance
    json_data = JSONRenderer().render(serializer.data)
# Converts the serialized data to JSON
    return HttpResponse(json_data, content_type='applica
tion/json')  # Returns the JSON response
```

- **Flow:**
  1. **Retrieve the Model Instance:** `Student.objects.get(id=pk)` fetches the specific `Student` model instance from the database.
  2. **Serialize the Data:** `StudentSerializer(stu)` converts the model instance into a Python dictionary (native data type).
  3. **Render to JSON:** `JSONRenderer().render(serializer.data)` converts the Python dictionary to JSON format.
  4. **Return the Response:** The JSON data is returned to the client as an `HttpResponse` with the content type set to `'application/json'`.

## Summary:

- The `models.py` defines the data structure (how your data looks in the database).
- The `serializers.py` defines how to convert this data into a format that can be easily rendered to JSON.
- The `views.py` handles the request, fetches the model data, uses the serializer to convert the data, and then returns the JSON response to the client.

This flow allows Django to efficiently manage, serialize, and return data from the server to the client in a JSON format, making it suitable for APIs.


Khushal Savalakha

💡 here first we add data into models.py and then when we making view we add that data to serializers.py which converts complex data into python native and then we convert it and send the JSON response am i true ?

Yes, your understanding is correct, but let's refine it slightly:

1. **Define the Data Structure in `models.py`:**
   - First, you define the structure of your data by creating a model in `models.py`. This model defines the fields and their types, representing the database table.
2. **Add Data to the Model (Optional):**
   - After defining the model, you can add data to this model either through Django's admin interface, Django shell, or directly through the database. This step involves populating the database with instances of the model (i.e., records).
3. **Create a Serializer in `serializers.py`:**
   - Next, you create a serializer in `serializers.py`. The serializer's job is to convert complex data types (like model instances) into Python native data types (like dictionaries) that can be easily rendered into JSON, XML, or other content types.
4. **Use the Serializer in `views.py`:**
   - In your view (`views.py`), you fetch the model instance (or instances) from the database.
   - You pass this instance (or instances) to the serializer, which converts the model data into a Python dictionary.
   - Finally, you render this dictionary into JSON format and send it as an HTTP response to the client.

## Flow Recap:

- **Model Definition (`models.py`):** You define the data structure.
- **Data Addition (Optional):** You populate the database with instances of this model.
- **Serialization (`serializers.py`):** You create a serializer to convert model instances into a format that can be rendered into JSON.
- **View (`views.py`):** You retrieve data, serialize it, and send it as a JSON response.


Khushal Savalakha

## So to summarize:

- **Model:** Defines the structure of the data.
- **Serializer:** Converts model instances into Python native data types.
- **View:** Handles requests, uses the serializer to prepare data, and sends it as a JSON response.

You're on the right track!

> 💡 it means like in models.py database in forms.py we are creating cleaned for every validation like this we are creating function create and update for every validation in serializers.py am i true  then explain how forms.py and serializers.py structure match with each other by making table and also give example that how validations are created in them and also tell me when code executes is_valid then how it knows which function executes for validation.?

Great question! Let’s break down how `forms.py` and `serializers.py` in Django handle validation, and how their structures are similar and different.

## Overview

Both `forms.py` and `serializers.py` are used to validate and process user inputs. While `forms.py` is used in traditional Django forms, `serializers.py` is used in Django REST Framework (DRF) for API data processing.

## Structure Comparison

| Feature | `forms.py` | `serializers.py` |
|---|---|---|
| **Purpose** | Validate and process HTML form submissions | Validate and process API request/response data |
| **Validation** | Use `clean_<fieldname>()` methods and validators | Use `validate_<fieldname>()` methods and `validate()` method |
| **Data Storage** | Data is processed and stored in models via forms | Data is processed and stored in models via serializers |
| **Methods for Validation** | `clean()` method for overall form validation | `validate()` method for overall data validation |

## How It Works


Khushal Savalakha

## 1. Django Forms ( `forms.py` ):

- **Structure:**

```
from django import forms
from .models import Student

class StudentForm(forms.ModelForm):
    class Meta:
        model = Student
        fields = ['name', 'roll', 'city']

    def clean_roll(self):
        roll = self.cleaned_data.get('roll')
        if roll < 1:
            raise forms.ValidationError("Roll number
must be positive.")
        return roll
```

- **Validation:**
  - **Field-specific Validation:** Use methods like `clean_<fieldname>` to validate specific fields.
  - **Form-wide Validation:** Use the `clean()` method to validate data across multiple fields.

## 2. DRF Serializers ( `serializers.py` ):

- **Structure:**

```
from rest_framework import serializers
from .models import Student

class StudentSerializer(serializers.ModelSerializer):
    class Meta:
        model = Student
        fields = ['name', 'roll', 'city']

    def validate_roll(self, value):
        if value < 1:
```


Khushal Savalakha

```
            raise serializers.ValidationError("Roll n
umber must be positive.")
        return value
```

- **Validation:**
  - **Field-specific Validation:** Use methods like `validate_<fieldname>` to validate specific fields.
  - **Object-wide Validation:** Use the `validate()` method to validate the entire data object.

## How Validations are Executed

When you call `is_valid()` in Django forms or serializers, the following process occurs:

1. **Field Validation:**
   - For each field, Django or DRF will call the field-specific validation methods (e.g., `clean_roll()` in forms, `validate_roll()` in serializers).
2. **Form/Serializer-Wide Validation:**
   - After field-specific validations, Django or DRF will call the overall validation method (`clean()` in forms, `validate()` in serializers) to perform any cross-field validations or additional checks.

## Example Flow

1. **For Django Forms:**
   - User submits a form.
   - `StudentForm` processes the data:
     - Calls `clean_roll()` to validate the `roll` field.
     - Calls `clean()` to validate the entire form.
   - If all validations pass, the form is considered valid and can be saved.
2. **For DRF Serializers:**
   - API request is made with data.
   - `StudentSerializer` processes the data:
     - Calls `validate_roll()` to validate the `roll` field.


Khushal Savalakha

    - Calls `validate()` to validate the entire data.
  - If all validations pass, the serializer is considered valid and can be used to save the data.

## Summary

- **Django Forms** handle validation for HTML forms and integrate with Django models via the `ModelForm` class. Validation methods are specific to fields or the whole form.
- **DRF Serializers** handle validation for API data and integrate with Django models via the `ModelSerializer` class. Validation methods are specific to fields or the entire data object.

Both mechanisms ensure that data meets the required criteria before saving or processing.

> 💡 Is there any difference between serializers.ModelSerializer and serializers.Serializer explain simply and depthly with example.

In Django REST Framework (DRF), both `serializers.ModelSerializer` and `serializers.Serializer` are used for serializing and deserializing data (i.e., converting between complex data types such as Django QuerySets and Python data types or JSON).

## Key Differences

1. **Automatic field generation**:
   - `ModelSerializer`: Automatically generates fields based on the Django model, so you don't need to define each field manually.
   - `Serializer`: You need to define each field explicitly, giving you full control over the structure of the serialized data.
2. **Ease of use**:
   - `ModelSerializer`: Easier and faster to use when you already have a Django model and just want a quick serializer for it.
   - `Serializer`: Requires more manual work but provides flexibility for custom or non-model data.
3. **Customization**:


Khushal Savalakha

- `ModelSerializer`: Can be customized, but it's tied to a specific model. Great for working directly with model data.
- `Serializer`: Ideal for handling custom data structures, not tied to any model, giving you full control.

## Example Explanation

### `serializers.ModelSerializer` Example:

Imagine you have a `Movie` model.

```
# models.py
class Movie(models.Model):
    title = models.CharField(max_length=100)
    description = models.TextField()
    release_date = models.DateField()
    rating = models.DecimalField(max_digits=3, decimal_p
laces=1)
```

With `ModelSerializer`, you can create a serializer like this:

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

class MovieSerializer(serializers.ModelSerializer):
    class Meta:
        model = Movie
        fields = ['title', 'description', 'release_dat
e', 'rating']
```

Here, the `MovieSerializer` will automatically create the fields based on the `Movie` model. You don't need to manually define each field.

When you serialize a `Movie` object:

```
# views.py
movie_instance = Movie.objects.get(id=1)
serializer = MovieSerializer(movie_instance)
print(serializer.data)  # Output: {'title': 'Inception',
```


Khushal Savalakha

```
'description': 'A mind-bending thriller', 'release_dat
e': '2010-07-16', 'rating': 8.8}
```

## `serializers.Serializer` Example:

If you want to serialize custom data or don't want to use a model, you can use `Serializer`.

```
# serializers.py
from rest_framework import serializers

class CustomMovieSerializer(serializers.Serializer):
    title = serializers.CharField(max_length=100)
    description = serializers.CharField()
    release_date = serializers.DateField()
    rating = serializers.DecimalField(max_digits=3, deci
mal_places=1)

    # You can add custom validation logic
    def validate_rating(self, value):
        if value < 0 or value > 10:
            raise serializers.ValidationError("Rating mu
st be between 0 and 10.")
        return value
```

Here, you manually define all the fields. It gives you more flexibility, allowing custom validation and handling non-model data.

When you serialize data:

```
# views.py
movie_data = {
    'title': 'Inception',
    'description': 'A mind-bending thriller',
    'release_date': '2010-07-16',
    'rating': 8.8
}
serializer = CustomMovieSerializer(data=movie_data)
if serializer.is_valid():
```

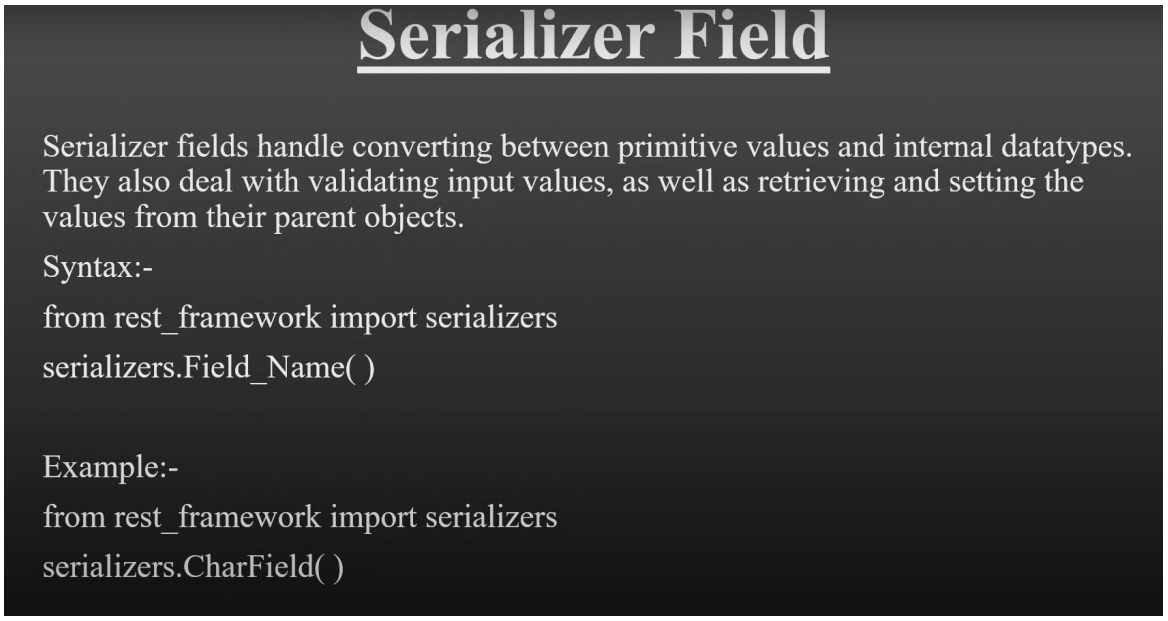

Khushal Savalakha

```
    print(serializer.data)  # Output: {'title': 'Incepti
on', 'description': 'A mind-bending thriller', 'release_
date': '2010-07-16', 'rating': 8.8}
else:
    print(serializer.errors)
```

### When to Use Each?

- **Use** `ModelSerializer` when you need to quickly create a serializer for a model, and you don't need complex customization.
- **Use** `Serializer` when you need more control, for example, if you want to serialize custom data that isn't tied to a model or need custom validation logic.

## ▼ Serializer Fields and Core Arguments





Khushal Savalakha

# Serializer Field

CharField - A text representation. Optionally validates the text to be shorter than max_length and longer than min_length.

Syntax:- CharField(max_length=None, min_length=None, allow_blank=False, trim_whitespace=True)

- max_length - Validates that the input contains no more than this number of characters.
- min_length - Validates that the input contains no fewer than this number of characters.
- allow_blank - If set to True then the empty string should be considered a valid value. If set to False then the empty string is considered invalid and will raise a validation error. Defaults to False.
- trim_whitespace - If set to True then leading and trailing whitespace is trimmed. Defaults to True.
- The allow_null option is also available for string fields, although its usage is discouraged in favor of allow_blank.

# Serializer Field

IntegerField - An integer representation.

Syntax:- IntegerField(max_value=None, min_value=None)

- max_value Validate that the number provided is no greater than this value.
- min_value Validate that the number provided is no less than this value.

FloatField - A floating point representation.

Syntax:- FloatField(max_value=None, min_value=None)

- max_value Validate that the number provided is no greater than this value.
- min_value Validate that the number provided is no less than this value.

# Serializer Field

DecimalField - A decimal representation, represented in Python by a Decimal instance.

Syntax:- DecimalField(max_digits, decimal_places, coerce_to_string=None, max_value=None, min_value=None)

- max_digits The maximum number of digits allowed in the number. It must be either None or an integer greater than or equal to decimal_places.
- decimal_places The number of decimal places to store with the number.
- coerce_to_string Set to True if string values should be returned for the representation, or False if Decimal objects should be returned. Defaults to the same value as the COERCE_DECIMAL_TO_STRING settings key, which will be True unless overridden. If Decimal objects are returned by the serializer, then the final output format will be determined by the renderer. Note that setting localize will force the value to True.
- max_value Validate that the number provided is no greater than this value.
- min_value Validate that the number provided is no less than this value.


Khushal Savalakhia

# Serializer Field

localize Set to True to enable localization of input and output based on the current locale. This will also force coerce_to_string to True. Defaults to False. Note that data formatting is enabled if you have set USE_L10N=True in your settings file.

rounding Sets the rounding mode used when quantising to the configured precision. Valid values are decimal module rounding modes. Defaults to None.

SlugField - A RegexField that validates the input against the pattern [a-zA-Z0-9_-]+

Syntax:- SlugField(max_length=50, min_length=None, allow_blank=False)

# Serializer Field

EmailField - A text representation, validates the text to be a valid e-mail address.

Syntax:- EmailField(max_length=None, min_length=None, allow_blank=False)

BooleanField - A boolean representation.

Syntax:- BooleanField()

NullBooleanField - A boolean representation that also accepts None as a valid value.

Syntax:- NullBooleanField()

# Serializer Field

URLField - A RegexField that validates the input against a URL matching pattern. Expects fully qualified URLs of the form http://<host>/<path>.

Syntax:- URLField(max_length=200, min_length=None, allow_blank=False)

FileField - A file representation. Performs Django's standard FileField validation.

Syntax:- FileField(max_length=None, allow_empty_file=False, use_url=UPLOADED_FILES_USE_URL)

max_length - Designates the maximum length for the file name.

allow_empty_file - Designates if empty files are allowed.

use_url - If set to True then URL string values will be used for the output representation. If set to False then filename string values will be used for the output representation. Defaults to the value of the UPLOADED_FILES_USE_URL settings key, which is True unless set otherwise.


Khushal Savalakha

# Serializer Field

DateField - A date representation.

Syntax:- DateField(format=api_settings.DATE_FORMAT, input_formats=None)

format - A string representing the output format. If not specified, this defaults to the same value as the DATE_FORMAT settings key, which will be 'iso-8601' unless set. Setting to a format string indicates that to_representation return values should be coerced to string output. Format strings are described below. Setting this value to None indicates that Python date objects should be returned by to_representation. In this case the date encoding will be determined by the renderer.

input_formats - A list of strings representing the input formats which may be used to parse the date. If not specified, the DATE_INPUT_FORMATS setting will be used, which defaults to ['iso-8601'].

# Serializer Field

DateTimeField - A date and time representation.

Syntax:- DateTimeField(format=api_settings.DATETIME_FORMAT, input_formats=None, default_timezone=None)

format - A string representing the output format. If not specified, this defaults to the same value as the DATETIME_FORMAT settings key, which will be 'iso-8601' unless set. Setting to a format string indicates that to_representation return values should be coerced to string output. Format strings are described below. Setting this value to None indicates that Python datetime objects should be returned by to_representation. In this case the datetime encoding will be determined by the renderer.

input_formats - A list of strings representing the input formats which may be used to parse the date. If not specified, the DATETIME_INPUT_FORMATS setting will be used, which defaults to ['iso-8601'].

default_timezone - A pytz.timezone representing the timezone. If not specified and the USE_TZ setting is enabled, this defaults to the current timezone. If USE_TZ is disabled, then datetime objects will be naive.

# Serializer Field

DurationField - A Duration representation. The validated_data for these fields will contain a datetime.timedelta instance. The representation is a string following this format '[DD] [HH:[MM:]]ss[.uuuuuu]'.

Syntax:- DurationField(max_value=None, min_value=None)

max_value Validate that the duration provided is no greater than this value.

min_value Validate that the duration provided is no less than this value.

RegexField - A text representation, that validates the given value matches against a certain regular expression.

Syntax:- RegexField(regex, max_length=None, min_length=None, allow_blank=False)

regex argument may either be a string, or a compiled python regular expression object.


Khushal Savalakha

## Serializer Field

UUIDField - A field that ensures the input is a valid UUID string. The to_internal_value method will return a uuid.UUID instance. On output the field will return a string in the canonical hyphenated format.

Syntax:- UUIDField(format='hex_verbose')

format: Determines the representation format of the uuid value

'hex_verbose' - The canonical hex representation, including hyphens: "5ce0e9a5-5ffa-654b-cee0-1238041fb31a"

'hex' - The compact hex representation of the UUID, not including hyphens: "5ce0e9a55ffa654bcee01238041fb31a"

'int' - A 128 bit integer representation of the UUID: "123456789012312313134124512351145145114"

'urn' - RFC 4122 URN representation of the UUID: "urn:uuid:5ce0e9a5-5ffa-654b-cee0-1238041fb31a" Changing the format parameters only affects representation values. All formats are accepted by to_internal_value

## Serializer Field

FilePathField -A field whose choices are limited to the filenames in a certain directory on the filesystem.

Syntax:- FilePathField(path, match=None, recursive=False, allow_files=True, allow_folders=False, required=None, **kwargs)

path - The absolute filesystem path to a directory from which this FilePathField should get its choice.

match - A regular expression, as a string, that FilePathField will use to filter filenames.

recursive - Specifies whether all subdirectories of path should be included. Default is False.

allow_files - Specifies whether files in the specified location should be included. Default is True. Either this or allow_folders must be True.

allow_folders - Specifies whether folders in the specified location should be included. Default is False. Either this or allow_files must be True.

and further more in video: https://youtu.be/Dr5C9vbcSbc?feature=shared

## Core Arguments

label - A short text string that may be used as the name of the field in HTML form fields or other descriptive elements.

validators - A list of validator functions which should be applied to the incoming field input, and which either raise a validation error or simply return. Validator functions should typically raise serializers.ValidationError, but Django's built-in ValidationError is also supported for compatibility with validators defined in the Django codebase or third party Django packages.

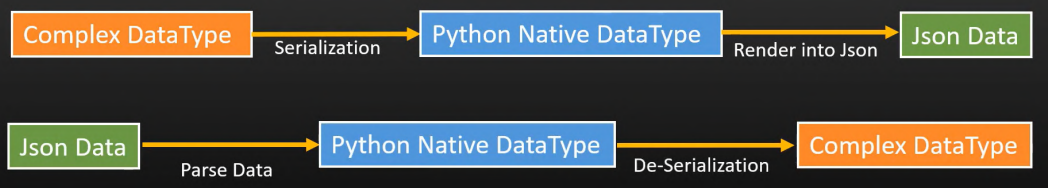
error_messages - A dictionary of error codes to error messages.


Khushal Savalakha

## ▼ DeSerialization and Insert Data

# De-serialization

Serializers are also responsible for deserialization which means it allows parsed data to be converted back into complex types, after first validating the incoming data.




# BytesIO( )

A stream implementation using an in-memory bytes buffer. It inherits BufferedIOBase. The buffer is discarded when the close() method is called.

```
import io
stream = io.BytesIO(json_data)
```

# JSONParser( )

This is used to parse json data to python native data type.

```
from rest_framework.parsers import JSONParser
parsed_data = JSONParser().parse(stream)
```


Khushal Savalakha

# De-serialization

Deserialization allows parsed data to be converted back into complex types, after first validating the incoming data.

Creating Serializer Object

serializer = StudentSerializer(data = parsed_data)

Validated Data

serializer.is_valid()

serializer.validated_data

serializer.errors

# serializer.validated_data

This is the Valid data.

serializer.validated_data

# Create Data/Insert Data

```
from rest_framework import serializers
class StudentSerializer(serializers.Serializer):
    name = serializers.CharField(max_length=100)
    roll = serializers.IntegerField()
    city = serializers.CharField(max_length=100)

   def create(self, validate_data):
          return Student.objects.create(**validate_data)
```

**models.py**

```
from django.db import models

# Create your models here.
class Student(models.Model):
```



Khushal Savalakha

```
    name=models.CharField(max_length=100)
    roll=models.IntegerField()
    city=models.CharField(max_length=100)
```

**views.py**

```
from django.shortcuts import render
import io
from rest_framework.parsers import JSONParser
from .serializers import StudentSerializer
from rest_framework.renderers import JSONRenderer
from django.http import HttpResponse
from django.views.decorators.csrf import csrf_exempt

@csrf_exempt
def student_create(request):
    if request.method == 'POST':
        json_data = request.body
        stream = io.BytesIO(json_data)
        python_data = JSONParser().parse(stream)
        serializer = StudentSerializer(data=python_data)

        if serializer.is_valid():
            try:
                serializer.save()
                res = {'msg': 'Data Created'}
                json_data = JSONRenderer().render(res)
                return HttpResponse(json_data, content_type
            except Exception as e:
                # Log the exception for debugging purposes
                print(f"Exception occurred: {e}")
                res = {'errors': str(e)}
                json_data = JSONRenderer().render(res)
                return HttpResponse(json_data, content_type
        else:
            res = {'errors': serializer.errors}
            json_data = JSONRenderer().render(res)
```


Khushal Savalakha

```
            return HttpResponse(json_data, content_type='ap
    return HttpResponse({'message': 'Method not allowed'},
```

**serializers.py**

```
from rest_framework import serializers
from .models import Student

class StudentSerializer(serializers.ModelSerializer):
    class Meta:
        model = Student
        fields = ['name', 'roll', 'city']


# class StudentSerializer(serializers.Serializer):
#     name=serializers.CharField(max_length=100)
#     roll=serializers.IntegerField()
#     city=serializers.CharField(max_length=100)


def create(self,validate_data):
    return Student.objects.create(**validate_data)
```

**myapp.py**

```
import requests
import json

URL = "http://127.0.0.1:8000/stucreate/"

data = {
    'name': 'sonam',
    'roll': 101,
    'city': 'Ranchi'
}

headers = {
    'Content-Type': 'application/json'
```


Khushal Savalakha

```
}

response = requests.post(url=URL, data=json.dumps(data), he

print("Response Status Code:", response.status_code)
print("Response Content:", response.text)
print("Response Headers:", response.headers)

# Only try to decode JSON if the response is valid
if response.headers.get('Content-Type') == 'application/jsc
    data = response.json()
    print(data)
else:
    print("Response is not JSON format.")
```

💡 it means like in models.py database in forms.py we are creating cleaned for every validation like this we are creating function create and update for every validation in serializers.py am i true then explain how forms.py and serializers.py structure match with each other by making table and also give example that how validations are created in them and also tell me when code executes is_valid then how it knows which function executes for validation?

Great question! Let's break down how `forms.py` and `serializers.py` in Django handle validation, and how their structures are similar and different.

## Overview

Both `forms.py` and `serializers.py` are used to validate and process user inputs. While `forms.py` is used in traditional Django forms, `serializers.py` is used in Django REST Framework (DRF) for API data processing.

## Structure Comparison

| Feature | `forms.py` | `serializers.py` |
|---|---|---|
| **Purpose** | Validate and process HTML form submissions | Validate and process API request/response data |
| **Validation** | Use `clean_<fieldname>()` methods and validators | Use `validate_<fieldname>()` methods and `validate()` method |


Khushal Savalakha

| | | |
|---|---|---|
| **Data Storage** | Data is processed and stored in models via forms | Data is processed and stored in models via serializers |
| **Methods for Validation** | `clean()` method for overall form validation | `validate()` method for overall data validation |

## How It Works

### 1. Django Forms ( `forms.py` ):

- **Structure:**

```
from django import forms
from .models import Student

class StudentForm(forms.ModelForm):
    class Meta:
        model = Student
        fields = ['name', 'roll', 'city']

    def clean_roll(self):
        roll = self.cleaned_data.get('roll')
        if roll < 1:
            raise forms.ValidationError("Roll number
must be positive.")
        return roll
```

- **Validation:**
  - **Field-specific Validation:** Use methods like `clean_<fieldname>` to validate specific fields.
  - **Form-wide Validation:** Use the `clean()` method to validate data across multiple fields.

### 2. DRF Serializers ( `serializers.py` ):

- **Structure:**

```
from rest_framework import serializers
from .models import Student
```


Khushal Savalakha

```
class StudentSerializer(serializers.ModelSerializer):
    class Meta:
        model = Student
        fields = ['name', 'roll', 'city']

    def validate_roll(self, value):
        if value < 1:
            raise serializers.ValidationError("Roll n
umber must be positive.")
        return value
```

- **Validation:**
  - **Field-specific Validation:** Use methods like `validate_<fieldname>` to validate specific fields.
  - **Object-wide Validation:** Use the `validate()` method to validate the entire data object.

## How Validations are Executed

When you call `is_valid()` in Django forms or serializers, the following process occurs:

1. **Field Validation:**
   - For each field, Django or DRF will call the field-specific validation methods (e.g., `clean_roll()` in forms, `validate_roll()` in serializers).
2. **Form/Serializer-Wide Validation:**
   - After field-specific validations, Django or DRF will call the overall validation method (`clean()` in forms, `validate()` in serializers) to perform any cross-field validations or additional checks.

## Example Flow

1. **For Django Forms:**
   - User submits a form.
   - `StudentForm` processes the data:
     - Calls `clean_roll()` to validate the `roll` field.
     - Calls `clean()` to validate the entire form.


Khushal Savalakha

- If all validations pass, the form is considered valid and can be saved.

2. **For DRF Serializers:**
   - API request is made with data.
   - `StudentSerializer` processes the data:
     - Calls `validate_roll()` to validate the `roll` field.
     - Calls `validate()` to validate the entire data.
   - If all validations pass, the serializer is considered valid and can be used to save the data.

## Summary

- **Django Forms** handle validation for HTML forms and integrate with Django models via the `ModelForm` class. Validation methods are specific to fields or the whole form.
- **DRF Serializers** handle validation for API data and integrate with Django models via the `ModelSerializer` class. Validation methods are specific to fields or the entire data object.

Both mechanisms ensure that data meets the required criteria before saving or processing.

> 💡 explain here how io.BytesIO works and in which form it converts the data and also for json parser with example depthly in simple way.here deserilization is used to compare json data with complex data am i true ?

## Understanding `io.BytesIO` and `JSONParser` in Django

1. **What is `io.BytesIO`?**
   - `io.BytesIO` is a Python class that creates an in-memory stream that behaves like a file. Instead of writing to or reading from a physical file on disk, you can store bytes in memory, which is useful for handling small chunks of data, like JSON in this case.

   When `request.body` is accessed in Django, it contains the raw HTTP request body (like the JSON data) as bytes. However, for further processing (e.g., converting it into a Python dictionary), this needs to be wrapped in a stream-like object that can be read by parsers.

   **Example:**


Khushal Savalakha

```
import io

# Simulating a raw JSON byte string from a request
json_bytes = b'{"name": "John", "age": 25}'

# Wrapping it in a BytesIO object
stream = io.BytesIO(json_bytes)

# Now, stream behaves like a file object, and parsers
can read from it
print(stream.read())  # Output: b'{"name": "John", "a
ge": 25}'
```

Here, the `BytesIO` object allows reading byte data as if it were coming from a file or a network connection. It acts as an intermediary between the raw `request.body` and the next step, `JSONParser`.

---

1. **What is `JSONParser`?**

   - `JSONParser` is part of the Django REST Framework, and it helps deserialize JSON data (which comes as a byte stream from `request.body`) into a Python dictionary. Deserialization means converting a format like JSON into Python's native data structures so that it can be processed or validated.

   In this case, after wrapping the raw byte data in `BytesIO`, the `JSONParser` reads the stream and converts it into a Python dictionary.

   **Example:**

```
from rest_framework.parsers import JSONParser
import io

# Simulating a raw JSON byte string from a request
json_bytes = b'{"name": "John", "age": 25}'

# Wrapping the JSON bytes in BytesIO stream
stream = io.BytesIO(json_bytes)

# Parsing the byte stream into Python dictionary usin
g JSONParser
```


Khushal Savalakha

```
python_data = JSONParser().parse(stream)

# Now python_data is a dictionary
print(python_data)  # Output: {'name': 'John', 'age':
25}
```

In this step, the JSONParser converts the incoming JSON data into a Python dictionary ( `{'name': 'John', 'age': 25}` ) that can be passed to Django serializers for validation and further operations.

---

## Deserialization and Validation

Yes, you are right! **Deserialization** is the process where the JSON data (which is a string format) is converted into complex Python data types (like dictionaries, lists, etc.). In your code, the process follows this pattern:

1. **Deserialization**: JSON data from the request is parsed into a Python dictionary using `JSONParser`.
2. **Validation**: Once the data is in Python format, the `StudentSerializer` checks whether it meets the expected structure and data types. If the data is valid (i.e., it passes all the checks), it can then be saved to the database.
3. **Serialization**: If the data is valid, the serializer creates a response (in this case, `{'msg': 'Data Created'}` ), which is then serialized back into JSON using `JSONRenderer`.

**In simple terms:**

- **Deserialization**: Converts JSON (a simple data format) into complex Python data (like dictionaries).
- **Validation**: Checks whether the data structure and content match the expected format (e.g., is the age a number, is the name a string, etc.).
- **Serialization**: Converts the response data (e.g., success message) back into JSON to send to the client.

---

## How Deserialization Works in the Code:

1. **Incoming Request**: JSON data is sent in the body of the request (as bytes).
2. **BytesIO**: The byte data ( `request.body` ) is wrapped in a `BytesIO` stream, which makes it readable as a file-like object.

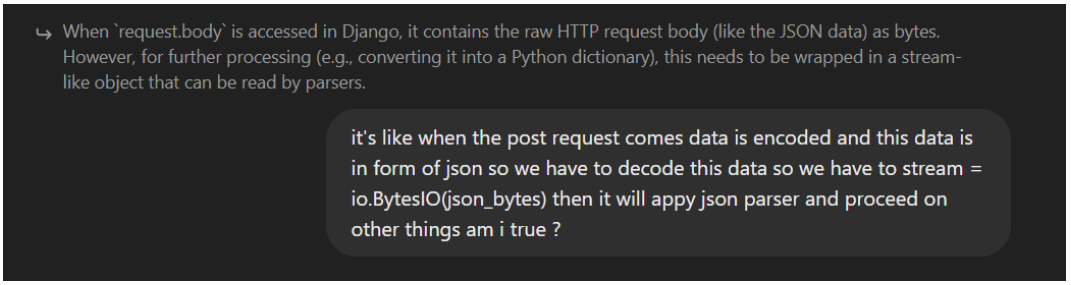

Khushal Savalakha

3. **JSONParser**: The stream is parsed into a Python dictionary using `JSONParser`.
4. **Serializer**: The `StudentSerializer` takes the Python dictionary and checks if it's valid. If it is, it saves the data to the database; if not, it returns validation errors.
5. **Response**: The server sends back either a success message (`Data Created`) or errors, both serialized into JSON format.

---

## Summary

- `io.BytesIO` wraps the raw byte data from `request.body` so that it can be read by the `JSONParser`.
- `JSONParser` converts JSON (in byte format) into Python dictionaries (deserialization).
- **Deserialization** helps compare or validate the incoming JSON data with complex Python data types to ensure it meets the expected schema.




Yes, you are correct!

When a POST request comes in, the data in the request body is typically encoded, and if it's JSON data, it's encoded as bytes. Here's the process step-by-step:

1. **Encoded Data in Request Body**: The POST request contains the JSON data, but this data is in byte format when it arrives (e.g., `b'{"name": "John", "age": 25}'`).
2. `io.BytesIO`: To decode and work with this data, you first need to wrap it in a `BytesIO` object. This effectively creates an in-memory stream that mimics a file, allowing the data to be read and processed by parsers.

   ```
   stream = io.BytesIO(request.body)
   ```

3. **JSON Parsing**: Once you have the data wrapped in `BytesIO`, you can pass it to a `JSONParser`, which will decode (or deserialize) the JSON byte data into a Python dictionary. This makes the data usable in your Python code.


Khushal Savalakha

```
python_data = JSONParser().parse(stream)
```

4. **Further Processing**: After decoding the JSON data into a Python dictionary, you can pass it to serializers for validation and saving to the database, or for other processing as needed.

So, in summary:

- **Request comes in** → Data is encoded in JSON byte format.
- `io.BytesIO` wraps this byte data in a stream.
- `JSONParser` decodes the byte stream into a Python dictionary.
- **Further actions** (like validation, saving, or responding) are performed on the decoded data.

This process ensures that your JSON data is correctly decoded and ready for further processing in Django.

## ▼ Quick Explanation about forms,model,Serilizer,ModelSerilizer


Khushal Savalakha

> 💡 1]it means first we making class in forms.py for making html form (not mendatory to make table of that data in models.py) forms.Form is for only taking data not to save and2] when we use forms.ModelForm that time first we create class and create table then we are making same type of input fields or making class meta in which first take instance of class present in models.py then we define fields which we want in generated {{form}} and some regarding validations then in views.py we are importing that forms.py class which data we want from user and save it in main model .3]like this model process of making table is same but In serilization process serilizers.Serilizer if we want to convert data in native python we send object instance or query set in class instance present in serilizers.py and then we further create that data into json and render it if want to 4]deserilization in which user sends data which are in json form then we convert it into byteio stream and then parse and send that native data in serializers.ModelSerializer instance then it will be stored in Model also am i true for all things mentioned here write true and false for every question if i am wrong then explain it properly with simple way depthly.

Let's go through each of your points and clarify them step by step.

## 1. Forms in Django:

- **Your Statement:** "First, we create a class in `forms.py` for making an HTML form (not mandatory to make a table for that data in `models.py`). `forms.Form` is for only taking data, not for saving."
- **True:** ✅
  - **Explanation:** `forms.Form` is used to create forms that are not necessarily tied to any model. It’s used when you need to capture data through an HTML form but don’t need to save it directly to the database. It is mainly for validation and processing of form data.

## 2. forms.ModelForm in Django:

- **Your Statement:** "When we use `forms.ModelForm`, we first create a class and create a table, then we are making the same type of input fields or creating a `class Meta` in which we first take an instance of the class present in `models.py`. Then we define fields which we want in the generated `{{form}}` and some validations. Then in


Khushal Savalakha

`views.py`, we import that `forms.py` class, get data from the user, and save it in the main model."

- **True:** ✅
  - **Explanation:** `forms.ModelForm` is a form tied to a model. It automatically generates form fields that correspond to the fields in the model. The `class Meta` in `ModelForm` defines which model to use and which fields to include in the form. This form can be used to capture data and save it directly to the database via the associated model.

### 3. Serialization Process with `serializers.Serializer`:

- **Your Statement:** "In the serialization process, `serializers.Serializer` is used if we want to convert data into native Python. We send an object instance or a queryset to the class instance in `serializers.py`, and then we further convert that data into JSON and render it."
- **True:** ✅
  - **Explanation:** `serializers.Serializer` is used to convert complex data like model instances or querysets into Python native data types (like dictionaries) that can then be easily converted into JSON. This is helpful when you need to serialize model instances for APIs.

### 4. Deserialization Process with `serializers.ModelSerializer`:

- **Your Statement:** "In deserialization, the user sends data in JSON form, then we convert it into a byte stream and parse it. We send that native data to the `serializers.ModelSerializer` instance, and then it will be stored in the Model."
- **True:** ✅
  - **Explanation:** Deserialization involves converting incoming JSON data into native Python data types, then validating and saving that data to the model. This process uses `serializers.ModelSerializer` to automatically map the data to the corresponding model fields and handle validation and saving.

---

### Summary of Key Points:

1. **forms.Form:**
   - Used to create forms that are not tied to any model.
   - Mainly for validation and data processing.


Khushal Savalakha

2. **forms.ModelForm:**
    - Tied to a model.
    - Automatically creates form fields based on the model's fields.
    - Used for both displaying forms and saving data to the database.
3. **serializers.Serializer:**
    - Used for manually serializing and deserializing data.
    - Ideal for complex or custom data structures not directly tied to a model.
4. **serializers.ModelSerializer:**
    - Tied to a model.
    - Automatically handles serialization and deserialization of model instances.
    - Simplifies the process of converting data to and from JSON.

This structure ensures that data flows smoothly between forms, models, and APIs, each playing its part in either presenting data to users or handling data from users.

> 💡 here in serialization we don't add data in model but in deserilization we add data through serilizers.ModelSerilizer am i true ?

Yes, you are correct, but let's clarify it a bit more:

## Serialization:

- **Purpose:** To convert complex data types (like model instances or querysets) into native Python data types that can be easily converted into JSON, XML, or other content types.
- **Common Usage:** You typically use `serializers.Serializer` or `serializers.ModelSerializer` to convert your model instances into JSON when you want to send this data to the client (e.g., in an API response).
- **Key Point:** During serialization, you are **not** adding or modifying data in the model. Instead, you are simply converting existing data from the model into a format that can be transmitted over the network (like JSON).

## Deserialization:


Khushal Savalakha

- **Purpose:** To convert incoming data (usually in JSON format) into native Python data types and then validate and save it to the database.
- **Common Usage:** You typically use `serializers.ModelSerializer` to convert the incoming JSON data into a model instance and save it to the database when you receive data from the client (e.g., in an API request).
- **Key Point:** During deserialization, you **are** adding or modifying data in the model. The deserialized data is validated and then saved to the database, creating or updating records.

## Summary:

- **Serialization**: Converting model data to JSON (or other formats) to send to the client. **No data is added to the model.**
- **Deserialization**: Converting incoming JSON data into a model instance and saving it to the database. **Data is added or updated in the model.**

So yes, your understanding is correct!

> 💡 according to you all things are true then make something visulizable like flow chart by which i don't have to read too much just see the flow chart and understand.

Sure! I'll create a simple and clear flowchart to help you visualize how `forms.py` and `serializers.py` work with `models.py` in Django. This will include both the data handling for forms and serialization/deserialization processes.

## Flowchart: How Forms, ModelForms, Serializers, and ModelSerializers Work

## 1. Handling HTML Forms with `forms.py`

### Forms without Model (forms.Form)

```
User Input (HTML Form)
        |
        v
   forms.Form
   (Class in forms.py)
```


Khushal Savalakha

```
        |
        v
  Validation
  (Custom validation can be applied here)
        |
        v
 Use Data (e.g., for display, emails, etc.)
  (Data is not saved to the database)
```

**Forms with Model (forms.ModelForm)**

```
User Input (HTML Form)
        |
        v
  forms.ModelForm
  (Class in forms.py)
        |
        v
    Linked to Model (Meta Class)
        |
        v
  Validation
  (Based on model fields & custom validation)
        |
        v
Save Data to Model (Database)
   (Data is automatically saved in the linked model)
```

💡 here you made things very complex  answer me for below things:
"""1]serialization is just for converting complex data and in python native data no any process with model database of adding new data and
2]while ModelSerilizer has end user input in json format which we can't directly add in model so we first stream it and add python native data in ModelSerilizer instance which directly add it to main model """
am i true for all above question ?and if there is further use of them then explain me in simple way.


Khushal Savalakha

Let's clarify the two points:

## 1. Serialization

**Statement:** Serialization is just for converting complex data into Python native data and not directly related to adding new data to the model database.

**Explanation:**

- **True**: Serialization is the process of converting complex data types (like model instances or querysets) into simpler data types (like Python dictionaries) that can be easily rendered into formats like JSON or XML.
- **Serialization** is **not** directly related to adding new data to the database. It only transforms data into a format that can be sent to the client (e.g., in response to a request).

## 2. Deserialization with `ModelSerializer`

**Statement:** While `ModelSerializer` handles end-user input in JSON format, it cannot directly add this data to the model. Instead, the data is first converted to Python native data and then added to the model.

**Explanation:**

- **True**: `ModelSerializer` is used for deserialization, which involves converting JSON data received from the client into Python native data types.
- The process involves:
  1. **Deserialization**: Converting the JSON data into a Python object.
  2. **Validation**: Ensuring the data is correct and meets any constraints.
  3. **Saving**: Adding or updating the data in the model (database).

## Simplified Summary

1. **Serialization:**
   - Converts complex data (model instances) into simpler formats (like JSON) for sending to clients.
   - **No interaction with the database** (no data creation or updates).
2. **Deserialization with `ModelSerializer`:**
   - Converts JSON data from clients into Python native types.


Khushal Savalakha

- Validates the data and then **adds or updates** it in the model (database).

## Further Use

- **Serialization**: Useful for sending data to clients or APIs. You can customize the output format or include additional computed fields.
- **Deserialization with** `ModelSerializer`: Useful for processing and saving data sent by clients. Ensures that data is validated and correctly mapped to model fields before saving.

## Example Flow:

1. **Serialization (Server Side):**
   - **Data:** Model instance
   - **Process:** Convert to JSON
   - **Output:** Send JSON to client
2. **Deserialization (Server Side):**
   - **Data:** JSON from client
   - **Process:** Convert JSON to Python data, validate, save to model
   - **Output:** Data is saved or updated in the database

This separation helps keep the concerns of data representation (serialization) and data management (deserialization and saving) distinct.

**DESERILIZATION**

*models.py*

```
from django.db import models

# Create your models here.
class Student(models.Model):
    name=models.CharField(max_length=100)
    roll=models.IntegerField()
    city=models.CharField(max_length=100)
```

***serializers.py***


Khushal Savalakha

```
from rest_framework import serializers
from .models import Student

class StudentSerializer(serializers.ModelSerializer):
    class Meta:
        model = Student
        fields = ['name', 'roll', 'city']


# class StudentSerializer(serializers.Serializer):
#     name=serializers.CharField(max_length=100)
#     roll=serializers.IntegerField()
#     city=serializers.CharField(max_length=100)


def create(self,validate_data):
    return Student.objects.create(**validate_data)
```

***views.py***

```
from django.shortcuts import render
import io
from rest_framework.parsers import JSONParser
from .serializers import StudentSerializer
from rest_framework.renderers import JSONRenderer
from django.http import HttpResponse
from django.views.decorators.csrf import csrf_exempt

@csrf_exempt
def student_create(request):
    if request.method == 'POST':
        json_data = request.body
        stream = io.BytesIO(json_data)
        python_data = JSONParser().parse(stream)
        serializer = StudentSerializer(data=python_data)

        if serializer.is_valid():
            try:
```


Khushal Savalakha

```
                serializer.save()
                res = {'msg': 'Data Created'}
                json_data = JSONRenderer().render(res)
                return HttpResponse(json_data, content_type
            except Exception as e:
                # Log the exception for debugging purposes
                print(f"Exception occurred: {e}")
                res = {'errors': str(e)}
                json_data = JSONRenderer().render(res)
                return HttpResponse(json_data, content_type
        else:
            res = {'errors': serializer.errors}
            json_data = JSONRenderer().render(res)
            return HttpResponse(json_data, content_type='ap
    return HttpResponse({'message': 'Method not allowed'},
```

💡 here we add data in main model by StudentSerializer(data=python_data) am i true ?

Yes, you are correct. Here's a breakdown of how data is added to the model using `StudentSerializer`:

## Flow of Data Addition Using `StudentSerializer`

1. **Data Parsing:**
   - **Incoming Data:** The data is received in JSON format through the `request.body`.
   - **Conversion:** The JSON data is parsed into a Python dictionary using `JSONParser()`.
2. **Serialization and Validation:**
   - **Serializer Initialization:** `StudentSerializer(data=python_data)` is used to create an instance of the `StudentSerializer` class with the parsed data.
   - **Validation:** The `is_valid()` method checks if the data meets the serializer's validation rules defined in the `StudentSerializer` class.
3. **Saving Data:**
   - **Data Creation:** If the data is valid, `serializer.save()` is called.


Khushal Savalakha

- **Internally:** This invokes the `create()` method of the `ModelSerializer` (if defined), which handles the process of saving the data to the model (i.e., the database).
- **Automatic Creation:** If you use `ModelSerializer` without overriding `create()`, it uses the default behavior to create and save the model instance.

4. **Response:**
   - **Success:** On successful creation, a JSON response with a success message is returned.
   - **Failure:** If there are errors or exceptions, appropriate error messages are returned.

## Summary of Key Points:

- `StudentSerializer`: This serializer is used to validate and serialize data. When you call `serializer.save()`, it handles saving the data to the `Student` model.
- `create()` **Method**: This method is used to customize the way data is saved to the database. In your case, if you had overridden this method, it would be used. Otherwise, `ModelSerializer` uses its default implementation.

So, the data is added to the main model (`Student`) through the `StudentSerializer` when `serializer.save()` is called. This process involves converting the validated data into a model instance and then saving it to the database.

> 💡 here while using StudentSerializer(data=python_data) we are just doing validation in python native data after validation we call create which saves data in main models.py usable class table am i true ?

Yes, you are correct. Here's a concise summary of the process:

1. **Initialization and Validation:**
   - **Data Parsing:** When you call `StudentSerializer(data=python_data)`, you are initializing the serializer with the parsed Python native data.
   - **Validation:** The `serializer.is_valid()` method is used to validate this data against the rules defined in the `StudentSerializer` class.
2. **Data Creation:**


Khushal Savalakha

- **Save Method:** If the data is valid, calling `serializer.save()` will trigger the `create()` method of `ModelSerializer` (if you have overridden it) or use the default implementation provided by `ModelSerializer`.
- **Saving to Model:** The `create()` method is responsible for creating a new instance of the `Student` model with the validated data and saving it to the database.

## Detailed Flow:

- **Step 1: Initialization**

```
serializer = StudentSerializer(data=python_data)
```

- **Step 2: Validation**

```
if serializer.is_valid():
    # Proceed to save
```

- **Step 3: Data Creation**

```
serializer.save()
```

  - **Default Behavior:** If `create()` is not overridden, `ModelSerializer` will use its default method to save the validated data to the database.
  - **Custom Behavior:** If `create()` is overridden, it will use your custom logic to save the data.

## Example

Here's a simple example to illustrate:

**`serializers.py`:**

```
from rest_framework import serializers
from .models import Student

class StudentSerializer(serializers.ModelSerializer):
    class Meta:
        model = Student
```


Khushal Savalakha

```
        fields = ['name', 'roll', 'city']

    # Custom create method (optional)
    def create(self, validated_data):
        return Student.objects.create(**validated_data)
```

`views.py`:

```
from django.http import HttpResponse
from rest_framework.parsers import JSONParser
from rest_framework.renderers import JSONRenderer
from .serializers import StudentSerializer
from django.views.decorators.csrf import csrf_exempt
import io

@csrf_exempt
def student_create(request):
    if request.method == 'POST':
        json_data = request.body
        stream = io.BytesIO(json_data)
        python_data = JSONParser().parse(stream)
        serializer = StudentSerializer(data=python_data)

        if serializer.is_valid():
            try:
                serializer.save()  # Creates and saves t
he model instance
                res = {'msg': 'Data Created'}
                json_data = JSONRenderer().render(res)
                return HttpResponse(json_data, content_t
ype='application/json')
            except Exception as e:
                res = {'errors': str(e)}
                json_data = JSONRenderer().render(res)
                return HttpResponse(json_data, content_t
ype='application/json', status=400)
        else:
            res = {'errors': serializer.errors}
```


Khushal Savalakha

```
            json_data = JSONRenderer().render(res)
            return HttpResponse(json_data, content_type
='application/json', status=400)
    return HttpResponse({'message': 'Method not allowe
d'}, status=405)
```

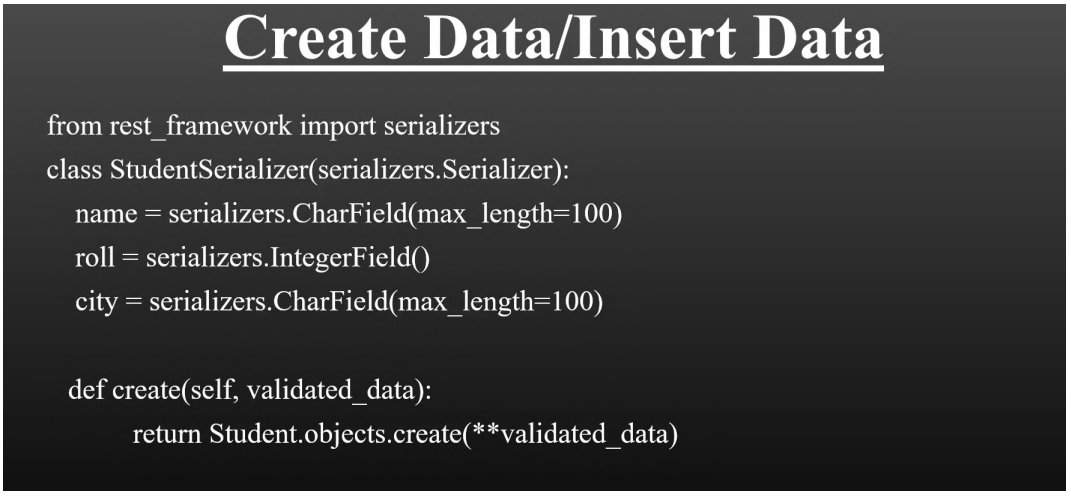
### Summary

- `StudentSerializer(data=python_data)` : Initializes the serializer with data and prepares it for validation.
- `serializer.is_valid()` : Validates the data.
- `serializer.save()` : Saves the validated data to the `Student` model, either using the **default** `create()` method or a custom one if defined.

# ▼ CRUD API using Function Based View and Class Based View

## ▼ Function Based Views

Previously





Khushal Savalakha

# Update Data

```
from rest_framework import serializers
class StudentSerializer(serializers.Serializer):
    name = serializers.CharField(max_length=100)
    roll = serializers.IntegerField()
    city = serializers.CharField(max_length=100)
    def update(self, instance, validated_data):
        instance.name = validated_data.get('name', instance.name)
        instance.roll = validated_data.get('roll', instance.roll)
        instance.city = validated_data.get('city', instance.city)
        instance.save()
        return instance
```

New Data from user for updation

Old Data stored in Database

# Complete Update Data

By default, serializers must be passed values for all required fields or they will raise validation errors.

Required All Data from Front End/Client

```
serializer = StudentSerializer(stu, data=pythondata)
if serializer.is_valid():
    serializer.save()
```

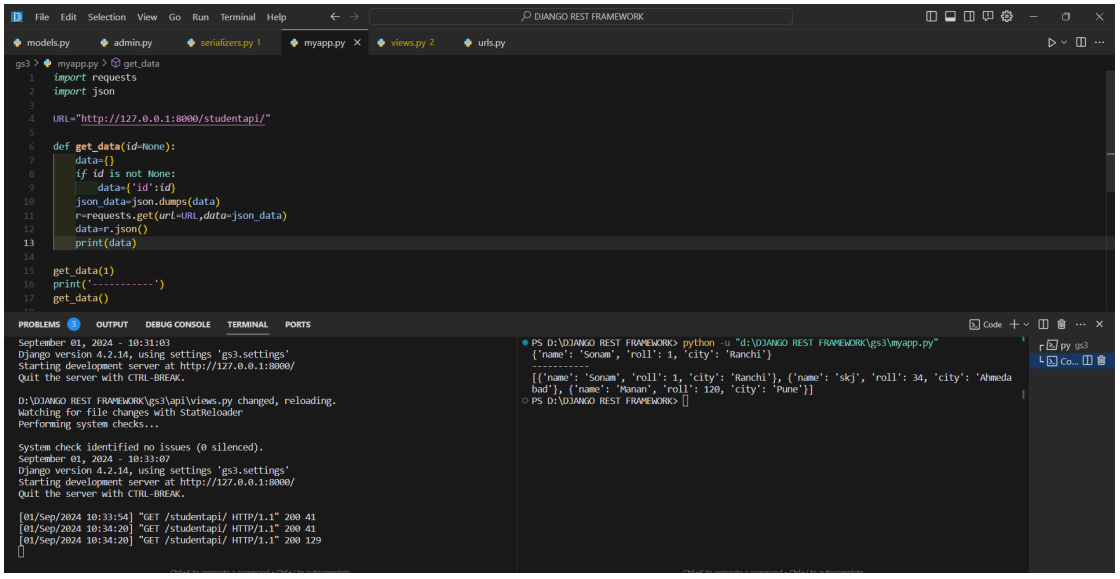
# Partial Update Data

Partial Update - All Data not required

```
serializer = StudentSerializer(stu, data=pythondata, partial=True)
if serializer.is_valid():
    serializer.save()
```



Khushal Savalakha




```
# from django.shortcuts import render
# import io
# from .models import Student
# from rest_framework.parsers import JSONParser
# from rest_framework.renderers import JSONRenderer
# from django.http import HttpResponse
# from .serializers import StudentSerializer
# # Create your views here.
# def student_api(request):
#     if request.method =='GET':
#         json_data=request.body
#         stream=io.ByteIo(json_data)
#         pythondata=JSONparser().parse(stream)
#         #it converts binary form data into python dict
#         # (python native data)
#         id=pythondata.get('id',None)
#         if id is not None:
#             stu=Student.objects.get(id=id)
#             serializer=StudentSerializer(stu)
#             json_data=JSONRenderer().render(serializer
#             return HttpResponse(json_data,content_type=


from django.shortcuts import render
import io
```


Khushal Savalakha

```
from .models import Student
from rest_framework.parsers import JSONParser
from rest_framework.renderers import JSONRenderer
from django.http import HttpResponse
from .serializers import StudentSerializer

def student_api(request):
    if request.method == 'GET':
        # Get the raw JSON data from the request body
        json_data = request.body

        # Convert the JSON data (in bytes) into a stream
        stream = io.BytesIO(json_data)
        print(stream)

        # Parse the byte stream to convert it into a Pyt
        pythondata = JSONParser().parse(stream)

        # Extract the 'id' from the parsed data, if avai
        id = pythondata.get('id', None)

        if id is not None:
            # Fetch the student object based on the prov
            stu = Student.objects.get(id=id)

            # Serialize the student object into a JSON-c
            serializer = StudentSerializer(stu)

            # Render the serialized data as JSON
            json_data = JSONRenderer().render(serializer

            # Return the JSON data as an HTTP response
            return HttpResponse(json_data, content_type=

    stu=Student.objects.all()
    serializer=StudentSerializer(stu,many=True)
    # Render the serialized data as JSON
    json_data = JSONRenderer().render(serializer.data)
```


Khushal Savalakha

```
    # Return the JSON data as an HTTP response
    return HttpResponse(json_data, content_type='applica
```

> 💡 1]here first of all when get request comes that times data is in json format but enccoded so on that time io.BytesIO(json_data) converts that data in <_io.BytesIO object at 0x000001FF08FB4450> after this we convert this data in python native datatype like dixtionary2] then we are getting id and fetch data in complex form like Query set or object instance and this data we are converting in python native format and 3] after this using JSONRenderer().render(serializer.data) we are converting this data in json format and return HttpResponse  it am i true for all questions ? if not then explain with example in simple way.

**Code Breakdown**

1. **Receiving and Handling the JSON Data:**
   - **What you wrote:** "First of all, when a GET request comes, the data is in JSON format but encoded. At that time, `io.BytesIO(json_data)` converts that data into a `<_io.BytesIO object at 0x000001FF08FB4450>`."
   - **Explanation:**
     - The data in `request.body` is indeed in raw byte format (which could be JSON or another type of encoded data).
     - `io.BytesIO(json_data)` creates a byte stream from the raw data. This stream can be read by the `JSONParser` to convert it into a Python-native format like a dictionary.
2. **Parsing the JSON Data:**
   - **What you wrote:** "After this, we convert this data into a Python native datatype like a dictionary."
   - **Explanation:**
     - The `JSONParser().parse(stream)` takes the byte stream and parses it into a Python dictionary.
     - For example, if the incoming JSON data is `{"id": 1}`, after parsing, `pythondata` would be a Python dictionary: `{'id': 1}`.


Khushal Savalakha

3. **Fetching Data Based on ID:**
   - **What you wrote:** "We are getting `id` and fetching data in complex form like QuerySet or object instance."
   - **Explanation:**
     - Once you extract the `id` from the dictionary, you use it to query your database.
     - `stu = Student.objects.get(id=id)` retrieves a single `Student` object (not a QuerySet) from the database where the `id` matches.
     - If no `id` is provided or found, the code will retrieve all `Student` objects using `Student.objects.all()`.
4. **Serializing the Data:**
   - **What you wrote:** "This data we are converting into Python native format."
   - **Explanation:**
     - The `StudentSerializer(stu)` takes the `Student` object (or objects) and serializes it into a Python dictionary that can be easily converted to JSON.
     - If you're serializing a single object, the output might look like `{'id': 1, 'name': 'John Doe', 'age': 21}`.
     - If you're serializing multiple objects, the output would be a list of dictionaries.
5. **Rendering the JSON Response:**
   - **What you wrote:** "After this, using `JSONRenderer().render(serializer.data)`, we are converting this data into JSON format and returning an `HttpResponse`."
   - **Explanation:**
     - `JSONRenderer().render(serializer.data)` converts the serialized data (a Python dictionary or list of dictionaries) into a JSON string.
     - The `HttpResponse` then returns this JSON string to the client, which the client (like a browser or API consumer) can interpret as JSON.

Example for Clarity:

Let's say your `Student` model has fields `id`, `name`, and `age`.


Khushal Savalakha

- **Incoming JSON Request Body:**

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

- **After Parsing:**

```
pythondata = {'id': 1}
```

- **Fetching the Student Object:**

```
stu = Student.objects.get(id=1)
# Let's assume the student with id=1 is John Doe, a
ge 21.
```

- **Serializing the Object:**

```
serializer = StudentSerializer(stu)
# The serializer's data would be: {'id': 1, 'name':
'John Doe', 'age': 21}
```

- **Rendering the Data to JSON:**

```
json_data = JSONRenderer().render(serializer.data)
# The json_data would be: b'{"id": 1, "name": "John
Doe", "age": 21}'
```

- **Returning the Response:**

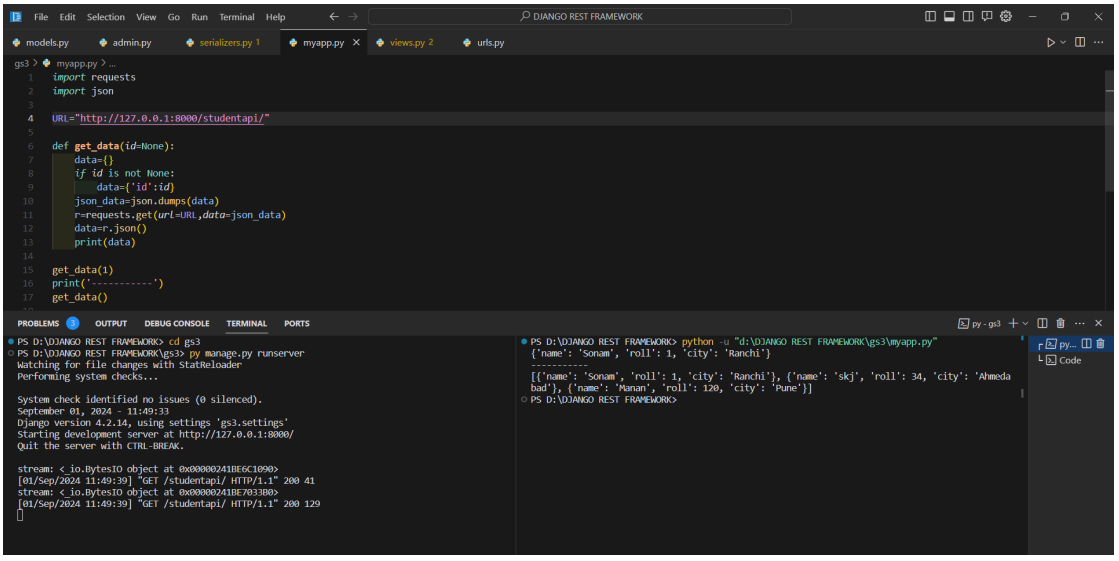
```
return HttpResponse(json_data, content_type='applic
ation/json')
```

Summary:

- **Yes, you're correct** in your explanation: you're taking JSON-encoded data, parsing it into a Python dictionary, fetching data from the database based on the `id`, serializing that data into a JSON-compatible format, and then rendering it back into JSON to send as an HTTP response.

If any part of this is still unclear or if you need further clarification, feel free to ask!


Khushal Savalakha




**views.py**

```
from django.shortcuts import render
import io
from api.models import Student
from rest_framework.parsers import JSONParser
from rest_framework.renderers import JSONRenderer
from django.http import HttpResponse
from .serializers import StudentSerializer
from django.views.decorators.csrf import csrf_exempt

@csrf_exempt
def student_api(request):
    if request.method == 'GET':
        # Get the raw JSON data from the request body
        json_data = request.body

        # Convert the JSON data (in bytes) into a stream
        stream = io.BytesIO(json_data)
        print('stream:',stream)
        # Parse the byte stream to convert it into a Pyt
        pythondata = JSONParser().parse(stream)

        # Extract the 'id' from the parsed data, if avai
        id = pythondata.get('id', None)
```


Khushal Savalakha

```
        if id is not None:
            # Fetch the student object based on the prov
            stu = Student.objects.get(id=id)

            # Serialize the student object into a JSON-c
            serializer = StudentSerializer(stu)

            # Render the serialized data as JSON
            json_data = JSONRenderer().render(serializer

            # Return the JSON data as an HTTP response
            return HttpResponse(json_data, content_type=
        else:
            stu=Student.objects.all()
            serializer=StudentSerializer(stu,many=True)
            # Render the serialized data as JSON
            json_data = JSONRenderer().render(serializer
            # Return the JSON data as an HTTP response
            return HttpResponse(json_data,content_type='
    if request.method=='POST':
        json_data=request.body
        stream=io.BytesIO(json_data)
        pythondata=JSONParser().parse(stream)
        serializer=StudentSerializer(data=pythondata)
        if serializer.is_valid():
            serializer.save()
            res={'msg':'Data Created'}
            json_data=JSONRenderer().render(res)
            return HttpResponse(json_data,content_type='
        else:
            json_data=JSONRenderer().render(serializer.e
            return HttpResponse(json_data,content_type='
```

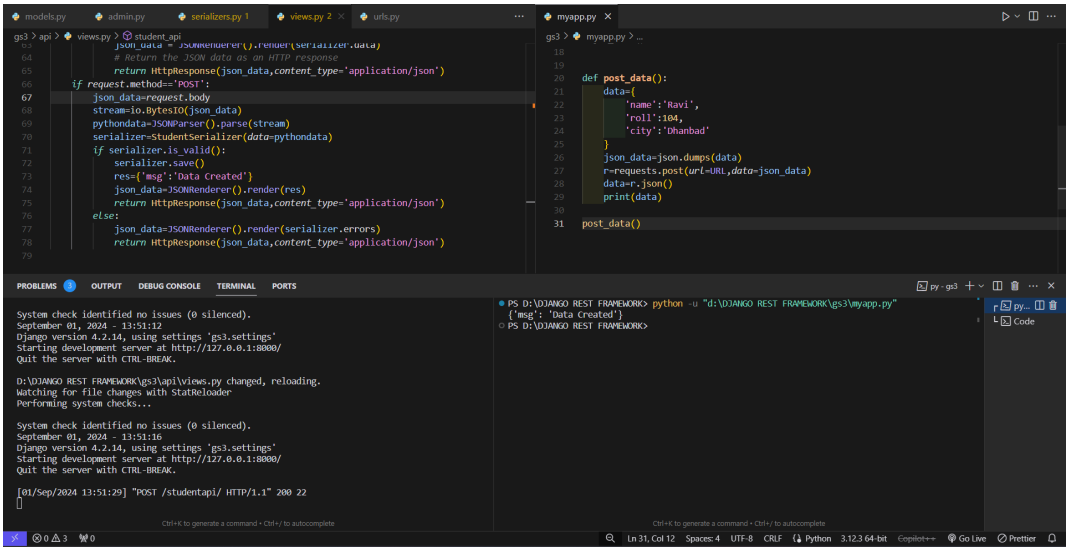
**my_app.py**

```
import requests
import json
```


Khushal Savalakha

```
URL="http://127.0.0.1:8000/studentapi/"

def get_data(id=None):
    data={}
    if id is not None:
        data={'id':id}
    json_data=json.dumps(data)
    r=requests.get(url=URL,data=json_data)
    data=r.json()
    print(data)

# get_data(1)
# print('-----------')
# get_data()


def post_data():
    data={
        'name':'Ravi',
        'roll':104,
        'city':'Dhanbad'
    }
    json_data=json.dumps(data)
    r=requests.post(url=URL,data=json_data)
    data=r.json()
    print(data)

post_data()
```


Khushal Savalakha




## serializers.py

```
from rest_framework import serializers
from .models import Student
class StudentSerializer(serializers.Serializer):
    name=serializers.CharField(max_length=100)
    roll=serializers.IntegerField()
    city=serializers.CharField(max_length=100)

    def create(self,validated_data):
        return Student.objects.create(**validated_data)

    def update(self,instance,validated_data):
        print(instance.name)
        instance.name=validated_data.get('name',instance
        print(instance.name)
        instance.roll=validated_data.get('roll',instance
        instance.city=validated_data.get('city',instance
        instance.save()
        return instance
```

## views.py

```
from django.shortcuts import render
import io
```

```
from api.models import Student
from rest_framework.parsers import JSONParser
from rest_framework.renderers import JSONRenderer
from django.http import HttpResponse
from .serializers import StudentSerializer
from django.views.decorators.csrf import csrf_exempt

@csrf_exempt
def student_api(request):
    if request.method == 'GET':
        # Get the raw JSON data from the request body
        json_data = request.body

        # Convert the JSON data (in bytes) into a stream
        stream = io.BytesIO(json_data)
        print('stream:',stream)
        # Parse the byte stream to convert it into a Pyt
        pythondata = JSONParser().parse(stream)

        # Extract the 'id' from the parsed data, if avai
        id = pythondata.get('id', None)

        if id is not None:
            # Fetch the student object based on the prov
            stu = Student.objects.get(id=id)

            # Serialize the student object into a JSON-c
            serializer = StudentSerializer(stu)

            # Render the serialized data as JSON
            json_data = JSONRenderer().render(serializer

            # Return the JSON data as an HTTP response
            return HttpResponse(json_data, content_type=
        else:
            stu=Student.objects.all()
            serializer=StudentSerializer(stu,many=True)
            # Render the serialized data as JSON
```


Khushal Savalakha

```
            json_data = JSONRenderer().render(serializer
            # Return the JSON data as an HTTP response
            return HttpResponse(json_data,content_type='
    if request.method=='POST':
        json_data=request.body
        stream=io.BytesIO(json_data)
        pythondata=JSONParser().parse(stream)
        serializer=StudentSerializer(data=pythondata)
        if serializer.is_valid():
            serializer.save()
            res={'msg':'Data Created'}
            json_data=JSONRenderer().render(res)
            return HttpResponse(json_data,content_type='
        else:
            json_data=JSONRenderer().render(serializer.e
            return HttpResponse(json_data,content_type='
    if request.method=='PUT':
        json_data=request.body
        stream=io.BytesIO(json_data)
        pythondata=JSONParser().parse(stream)
        id=pythondata.get('id')
        stu=Student.objects.get(id=id)
        serializer=StudentSerializer(stu,data=pythondata
        if serializer.is_valid():
            serializer.save()
            res={'msg':'Data Updated'}
            json_data=JSONRenderer().render(res)
            return HttpResponse(json_data,content_type='
        else:
            json_data=JSONRenderer().render(serializer.e
            return HttpResponse(json_data,content_type='
```

**myapp.py**

```
import requests
import json

URL="http://127.0.0.1:8000/studentapi/"
```


Khushal Savalakha

```
def get_data(id=None):
    data={}
    if id is not None:
        data={'id':id}
    json_data=json.dumps(data)
    r=requests.get(url=URL,data=json_data)
    data=r.json()
    print(data)

# get_data(1)
# print('-----------')
# get_data()


def post_data():
    data={
        'name':'Ravi',
        'roll':104,
        'city':'Dhanbad'
    }
    json_data=json.dumps(data)
    r=requests.post(url=URL,data=json_data)
    data=r.json()
    print(data)

# post_data()


def update_data():
    data={
        'id':5,
        'name':'Rohit',
        'roll':104,
        'city':'Ranchi'
    }
    json_data=json.dumps(data)
    r=requests.put(url=URL,data=json_data)
```


Khushal Savalakha

```
    data=r.json()
    print(data)

update_data()
```

💡 why we have to write multiple save () like if request.method=='POST':
json_data=request.body
stream=io.BytesIO(json_data)
pythondata=JSONParser().parse(stream)
serializer=StudentSerializer(data=pythondata)
if serializer.is_valid():
serializer.save()
res={'msg':'Data Created'}
json_data=JSONRenderer().render(res)
return HttpResponse(json_data,content_type='application/json')
else:
json_data=JSONRenderer().render(serializ

and here below code instance.save()
def update(self,instance,validated_data):
print(
instance.name)
instance.name=validated_data.get('name',
instance.name)
print(
instance.name)
instance.roll=validated_data.get('roll',instance.roll)
instance.city=validated_data.get('city',instance.city)
instance.save()
return instance

is serializer.save() and instance.save() do different role if they are then explains simply and depthly .

In Django REST Framework, the `serializer.save()` and `instance.save()` methods serve different purposes in the context of handling data in your API. Here's a detailed explanation:


Khushal Savalakha

## `serializer.save()`

- **Purpose**: The `serializer.save()` method is used to create or update an instance of a model based on the validated data provided to the serializer.
- **Usage**: This method is invoked after `serializer.is_valid()` returns `True`, indicating that the data has passed all validation checks.
- **How It Works**:
  - If you are creating a new instance, `serializer.save()` calls the `create()` method defined in the serializer.
  - If you are updating an existing instance, it calls the `update()` method defined in the serializer.
- **Code Example**:

```
def create(self, validated_data):
    return Student.objects.create(**validated_data)

def update(self, instance, validated_data):
    instance.name = validated_data.get('name', inst
ance.name)
    instance.roll = validated_data.get('roll', inst
ance.roll)
    instance.city = validated_data.get('city', inst
ance.city)
    instance.save()
    return instance
```

## `instance.save()`

- **Purpose**: The `instance.save()` method is used to save changes to a specific model instance. It directly interacts with the Django model's save method.
- **Usage**: This method is called on an individual instance of a model to save any changes made to that instance.
- **How It Works**:
  - When you call `instance.save()`, Django performs an SQL `UPDATE` operation if the instance already exists or an `INSERT` operation if it's a new instance.


Khushal Savalakha

- This method is used inside the `update()` method of the serializer to persist changes to an existing model instance.

## Comparison of `serializer.save()` vs. `instance.save()`

- `serializer.save()`:
  - **Purpose**: To handle the logic for both creating and updating instances based on the data provided.
  - **Workflow**: It determines whether to call `create()` or `update()` based on whether the instance is new or existing.
  - **Use Case**: You use `serializer.save()` in views or other parts of your application where you want to handle the creation or updating of model instances based on the validated data.
- `instance.save()`:
  - **Purpose**: To actually save the instance to the database.
  - **Workflow**: It is used to persist changes to a single model instance, typically called within `update()` or `create()` methods.
  - **Use Case**: You use `instance.save()` within custom methods to persist changes to a model instance after you have made updates to its fields.

## Example in Context

Consider the following `update()` method inside a serializer:

```
def update(self, instance, validated_data):
    instance.name = validated_data.get('name', instanc
e.name)
    instance.roll = validated_data.get('roll', instanc
e.roll)
    instance.city = validated_data.get('city', instanc
e.city)
    instance.save()  # Save changes to the database
    return instance
```

- `instance.save()`: Here, it saves the updated instance to the database after modifying its fields.

When you call `serializer.save()` in your view:


Khushal Savalakha

```
if serializer.is_valid():
    serializer.save()  # This will call `create()` or
`update()`
    return Response({"msg": "Data Created"}, status=st
atus.HTTP_201_CREATED)
```

- `serializer.save()`: It decides whether to create a new instance or update an existing one based on whether the instance already exists and calls the appropriate method (`create()` or `update()`) internally.

In summary, `serializer.save()` is a higher-level method that abstracts away the details of creating or updating model instances, while `instance.save()` is a lower-level method used to actually persist changes to a specific model instance.

💡 def create(self,validated_data):
return Student.objects.create(**validated_data)

what is the use of this ? is this called by StudentSerilizers here like serializer=StudentSerializer(data=pythondata)?

Yes, the `create(self, validated_data)` method in your `StudentSerializer` class is used when creating a new instance of the `Student` model based on the validated data provided to the serializer. Let me break this down:

## What is the use of `create(self, validated_data)`?

The `create()` method is responsible for creating a new instance of the `Student` model using the data that has been validated by the serializer.

When you pass data to the serializer using `StudentSerializer(data=pythondata)`, and call `serializer.is_valid()`, the serializer first validates the data. If the data is valid, calling `serializer.save()` triggers the `create()` method to create a new record in the database.

## Is this called by `StudentSerializer`?

Yes, it is. When you call `serializer.save()` after validating the data (`serializer.is_valid()`), the `save()` method internally checks whether the serializer has been passed an existing instance (for updates) or not (for creation).


Khushal Savalakha

- **If no instance is provided**: It assumes you're creating a new object and calls the `create()` method defined in the serializer.
- **If an instance is provided**: It assumes you're updating an existing object and calls the `update()` method instead.

## Flow of Data Creation

Here's the typical flow when you create a new object:

1. **Data Input**: You provide input data (e.g., from a POST request in your view) to the serializer like this:

   ```
   serializer = StudentSerializer(data=pythondata)
   ```

2. **Validation**: You call `serializer.is_valid()` to validate the input data.
3. **Save/Creation**: If the data is valid, calling `serializer.save()` will invoke the `create()` method if no existing instance is passed. The `create()` method looks like this:

   ```
   def create(self, validated_data):
       # Create a new Student instance using the valid
   ated data
       return Student.objects.create(**validated_data)
   ```

4. **Database Insertion**: The `Student.objects.create(**validated_data)` is executed, which inserts a new `Student` record into the database.

## Code Example in Context

```
# serializers.py
from rest_framework import serializers
from .models import Student

class StudentSerializer(serializers.ModelSerializer):
    class Meta:
        model = Student
        fields = ['name', 'roll', 'city', 'email', 'pa
ssword']
```


Khushal Savalakha

```
    # Called when saving new data
    def create(self, validated_data):
        # Creating a new Student instance in the datab
ase
        return Student.objects.create(**validated_dat
a)
```

### How the `create()` Method is Called in Views

In your view, you would typically handle the request and use the serializer to create a new `Student` object, like so:

```
# views.py
from rest_framework.response import Response
from rest_framework import status
from .serializers import StudentSerializer

@api_view(['POST'])
def student_create(request):
    # Parse the incoming data and initialize the seria
lizer
    serializer = StudentSerializer(data=request.data)

    # Validate the data
    if serializer.is_valid():
        # Call save(), which internally calls create()
        serializer.save()
        return Response({"msg": "Data Created"}, statu
s=status.HTTP_201_CREATED)

    return Response(serializer.errors, status=status.H
TTP_400_BAD_REQUEST)
```

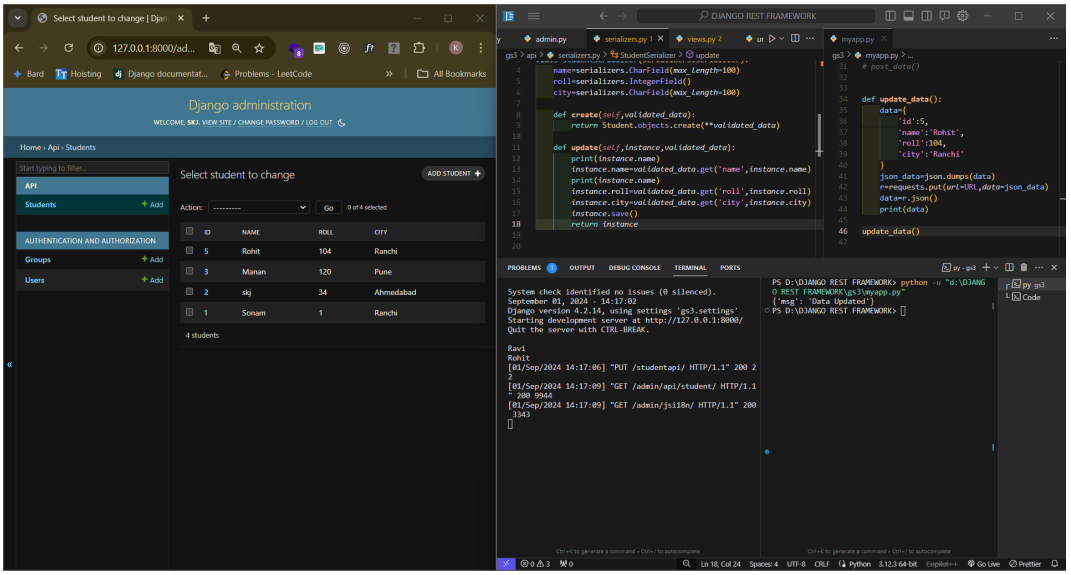
### Summary

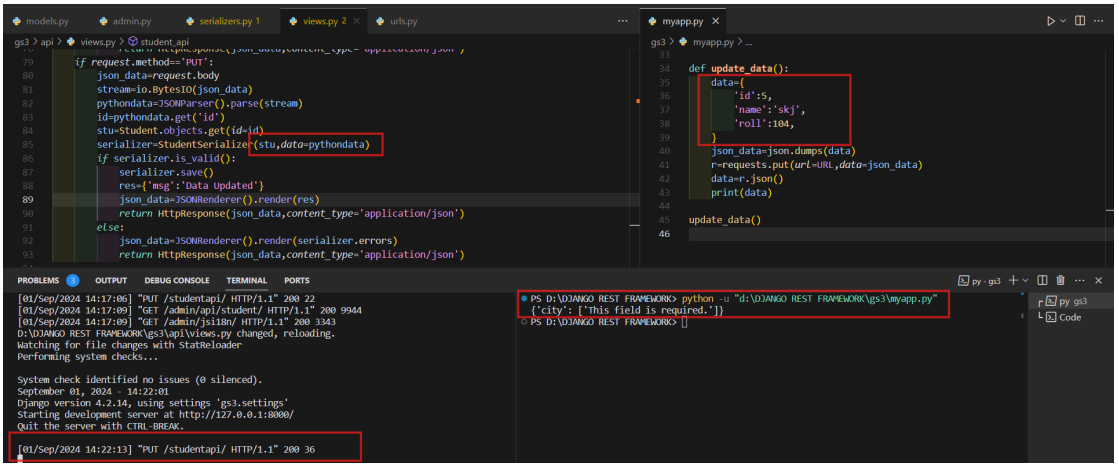
- The `create(self, validated_data)` method is automatically called when you use `serializer.save()` to create a new object.


Khushal Savalakha

- The serializer handles validation (`serializer.is_valid()`), and if the data is valid, `save()` either calls `create()` (for new objects) or `update()` (for existing ones).
- This method creates a new `Student` instance using the validated data and saves it to the database.

So, the `create()` method is essential when the serializer is used to add new data to the database.



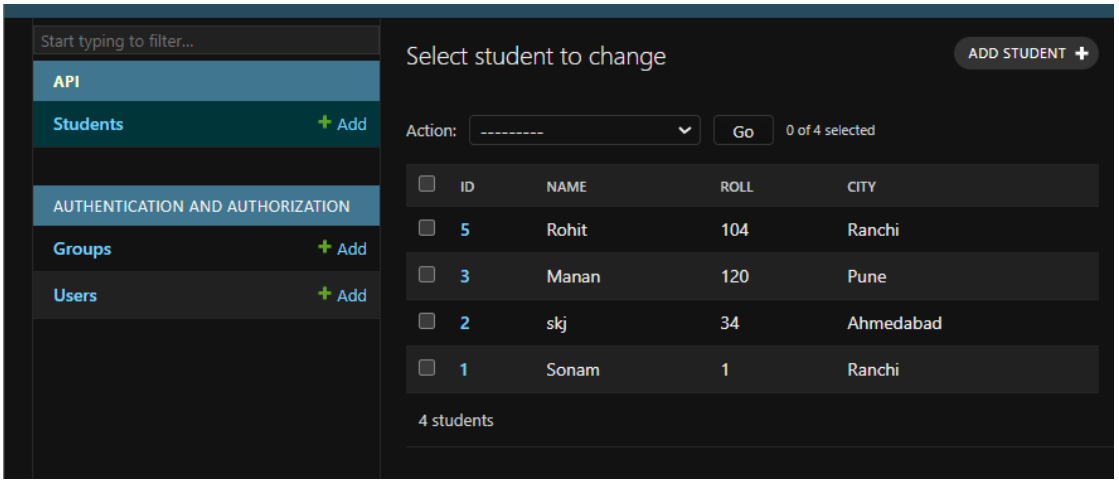

here if we don't want to update all data we have to write below line

serializer=StudentSerializer(stu,*data*=pythondata,*partial*=True)

```
data={
        'id':5,
        'name':'skj',
```


Khushal Savalakha

```
        'roll':104,
    }
```




```
43  stream = io.BytesIO(json_data)
44  pythondata = JSONParser().parse(stream)
45  id = pythondata.get('id')
46  stu = Student.objects.get(id=id)
47  #  Complete Update - Required All Data from Front End/Client
48  #  serializer = StudentSerializer(stu, data=pythondata)
49  #  Partial Update - All Data not required
50  serializer = StudentSerializer(stu, data=pythondata,
    partial=True)
51  if serializer.is_valid():
```

in delete operations we don't have to make any function.

**views.py**

```
from django.shortcuts import render
import io
from api.models import Student
from rest_framework.parsers import JSONParser
from rest_framework.renderers import JSONRenderer
from django.http import HttpResponse, JsonResponse
from .serializers import StudentSerializer
from django.views.decorators.csrf import csrf_exempt

@csrf_exempt
def student_api(request):
    if request.method == 'GET':
```


Khushal Savalakha

```
        # Get the raw JSON data from the request body
        json_data = request.body

        # Convert the JSON data (in bytes) into a stream
        stream = io.BytesIO(json_data)
        print('stream:',stream)
        # Parse the byte stream to convert it into a Pyt
        pythondata = JSONParser().parse(stream)

        # Extract the 'id' from the parsed data, if avai
        id = pythondata.get('id', None)

        if id is not None:
            # Fetch the student object based on the prov
            stu = Student.objects.get(id=id)

            # Serialize the student object into a JSON-c
            serializer = StudentSerializer(stu)

            # Render the serialized data as JSON
            json_data = JSONRenderer().render(serializer

            # Return the JSON data as an HTTP response
            return HttpResponse(json_data, content_type=
        else:
            stu=Student.objects.all()
            serializer=StudentSerializer(stu,many=True)
            # Render the serialized data as JSON
            json_data = JSONRenderer().render(serializer
            # Return the JSON data as an HTTP response
            return HttpResponse(json_data,content_type='
    if request.method=='POST':
        json_data=request.body
        stream=io.BytesIO(json_data)
        pythondata=JSONParser().parse(stream)
        serializer=StudentSerializer(data=pythondata)
        if serializer.is_valid():
            serializer.save()
```


Khushal Savalakha

```
            res={'msg':'Data Created'}
            json_data=JSONRenderer().render(res)
            return HttpResponse(json_data,content_type='
        else:
            json_data=JSONRenderer().render(serializer.e
            return HttpResponse(json_data,content_type='
    if request.method=='PUT':
        json_data=request.body
        stream=io.BytesIO(json_data)
        pythondata=JSONParser().parse(stream)
        id=pythondata.get('id')
        stu=Student.objects.get(id=id)
        #Complete Update - Required All Data from Front
        #serializer = StudentSerializer(stu, data=python
        #Partial Update - All Data not required
        serializer=StudentSerializer(stu,data=pythondata
        if serializer.is_valid():
            serializer.save()
            res={'msg':'Data Updated'}
            json_data=JSONRenderer().render(res)
            return HttpResponse(json_data,content_type='
        else:
            json_data=JSONRenderer().render(serializer.e
            return HttpResponse(json_data,content_type='
    if request.method=='DELETE':
        json_data=request.body
        stream=io.BytesIO(json_data)
        pythondata=JSONParser().parse(stream)
        id=pythondata.get('id')
        stu=Student.objects.get(id=id)
        stu.delete()
        res={'msg':'Data Deleted!!'}
        # json_data=JSONRenderer().render(res)
        # return HttpResponse(json_data,content_type='ap
        return JsonResponse(res,safe=False)
```

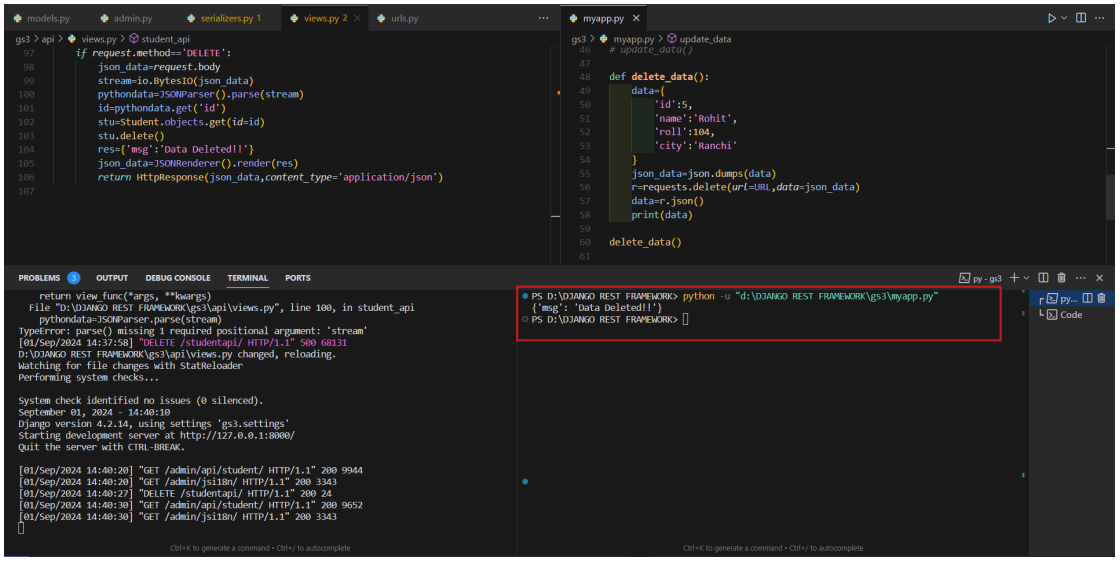
**my_app.py**


Khushal Savalakha

```
import requests
import json

URL="http://127.0.0.1:8000/studentapi/"

def get_data(id=None):
    data={}
    if id is not None:
        data={'id':id}
    json_data=json.dumps(data)
    r=requests.get(url=URL,data=json_data)
    data=r.json()
    print(data)

# get_data(1)
# print('-----------')
# get_data()


def post_data():
    data={
        'name':'Ravi',
        'roll':104,
        'city':'Dhanbad'
    }
    json_data=json.dumps(data)
    r=requests.post(url=URL,data=json_data)
    data=r.json()
    print(data)

# post_data()


def update_data():
    data={
        'id':5,
        'name':'Rohit',
        'roll':104,
```


Khushal Savalakha

```
        'city':'Ranchi'
    }
    json_data=json.dumps(data)
    r=requests.put(url=URL,data=json_data)
    data=r.json()
    print(data)

# update_data()

def delete_data():
    data={
        'id':2,
    }
    json_data=json.dumps(data)
    r=requests.delete(url=URL,data=json_data)
    data=r.json()
    print(data)

delete_data()
```



```
models.py   admin.py   serializers.py 1   views.py 2   urls.py
gs3 > api > views.py > ...
    if request.method=='DELETE':
        json_data=request.body
        stream=io.BytesIO(json_data)
        pythondata=JSONParser().parse(stream)
        id=pythondata.get('id')
        stu=Student.objects.get(id=id)
        stu.delete()
        res={'msg':'Data Deleted!!'}
        # json_data=JSONRenderer().render(res)
        # return HttpResponse(json_data,content_type='application/json')
        return JsonResponse(res,safe=False)
```

```
myapp.py
gs3 > myapp.py > delete_data
# update_data()

def delete_data():
    data={
        'id':2,
    }
    json_data=json.dumps(data)
    r=requests.delete(url=URL,data=json_data)
    data=r.json()
    print(data)

delete_data()
```

```
PROBLEMS 1   OUTPUT   DEBUG CONSOLE   TERMINAL   PORTS

changed, reloading.
Watching for file changes with StatReloader
Performing system checks...

System check identified no issues (0 silenced).
September 01, 2024 - 14:44:19
Django version 4.2.14, using settings 'gs3.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.

[01/Sep/2024 14:44:28] "DELETE /studentapi/ HTTP/1.1" 200 25
[01/Sep/2024 14:44:32] "GET /admin/api/student/ HTTP/1.1" 200 9362
[01/Sep/2024 14:44:32] "GET /admin/jsi18n/ HTTP/1.1" 200 3343

PS D:\DJANGO REST FRAMEWORK> python -u "d:\DJANGO REST FRAMEWORK\gs3\myapp.py"
{'msg': 'Data Deleted!!'}
PS D:\DJANGO REST FRAMEWORK>
```

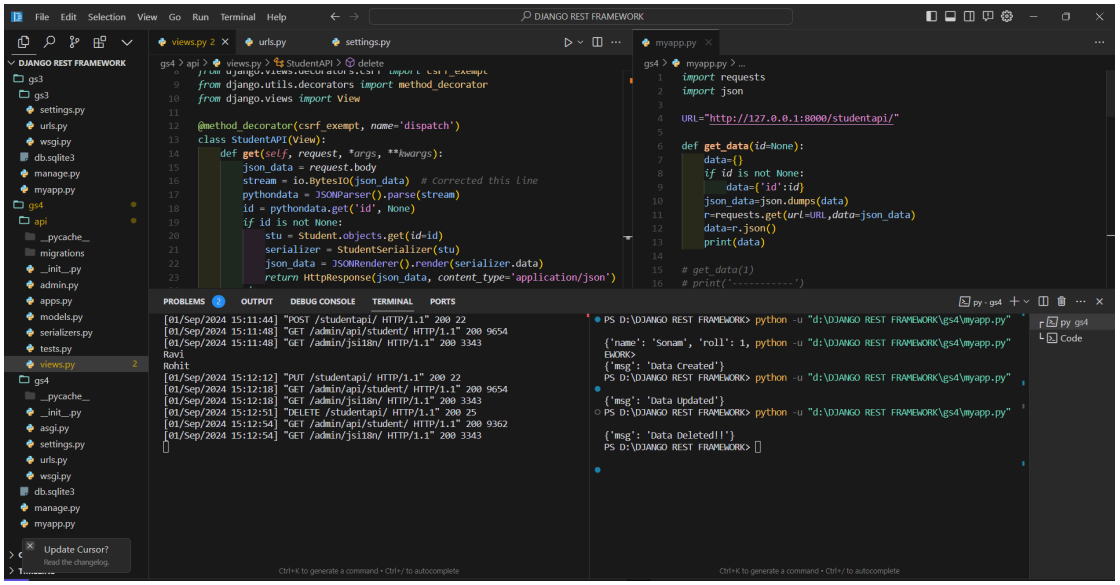
```
DRF CODE
  .vscode
  gs1
  gs2
  gs3
    api
      __pycache__
      migrations
      __init__.py
      admin.py
      apps.py
      models.py
      serializers.py
      tests.py
      views.py
    gs3
      __pycache__
      __init__.py
      asgi.py
      settings.py
      urls.py
      wsgi.py
    db.sqlite3
    manage.py
    myapp.py
```

```
def create(self, validated_data):
    return Student.objects.create(**validated_data)

def update(self, instance, validated_data):
    print(instance.name)
    instance.name = validated_data.get('name', instance.name)
    print(instance.name)
    instance.roll = validated_data.get('roll', instance.roll)
    instance.city = validated_data.get('city', instance.city)
    instance.save()
    return instance
```

```
TERMINAL   PROBLEMS   OUTPUT   DEBUG CONSOLE   1: python

[09/Oct/2020 18:44:15] "GET /admin/jsi18n/ HTTP/1.1" 200 3187
[09/Oct/2020 18:44:17] "GET /admin/api/student/?o=1 HTTP/1.1" 200 7205
[09/Oct/2020 18:44:17] "GET /admin/jsi18n/ HTTP/1.1" 200 3187
Rahul
Kunal
```

## ▼ Class Based Views


Khushal Savalakha




## views.py

```
from django.shortcuts import render
import io
from api.models import Student
from rest_framework.parsers import JSONParser
from rest_framework.renderers import JSONRenderer
from django.http import HttpResponse, JsonResponse
from .serializers import StudentSerializer
from django.views.decorators.csrf import csrf_exempt
from django.utils.decorators import method_decorator
from django.views import View

@method_decorator(csrf_exempt, name='dispatch')
class StudentAPI(View):
    def get(self, request, *args, **kwargs):
        json_data = request.body
        stream = io.BytesIO(json_data)  # Corrected this
        pythondata = JSONParser().parse(stream)
        id = pythondata.get('id', None)
        if id is not None:
            stu = Student.objects.get(id=id)
            serializer = StudentSerializer(stu)
            json_data = JSONRenderer().render(serializer
            return HttpResponse(json_data, content_type=
```


Khushal Savalakha

```
        else:
            stu = Student.objects.all()
            serializer = StudentSerializer(stu, many=Tru
            json_data = JSONRenderer().render(serializer
            return HttpResponse(json_data, content_type=

    def post(self, request, *args, **kwargs):
        json_data = request.body
        stream = io.BytesIO(json_data)  # Corrected this
        pythondata = JSONParser().parse(stream)
        serializer = StudentSerializer(data=pythondata)
        if serializer.is_valid():
            serializer.save()
            res = {'msg': 'Data Created'}
            json_data = JSONRenderer().render(res)
            return HttpResponse(json_data, content_type=
        else:
            json_data = JSONRenderer().render(serializer
            return HttpResponse(json_data, content_type=

    def put(self, request, *args, **kwargs):
        json_data = request.body
        stream = io.BytesIO(json_data)  # Corrected this
        pythondata = JSONParser().parse(stream)
        id = pythondata.get('id')
        stu = Student.objects.get(id=id)
        serializer = StudentSerializer(stu, data=pythond
        if serializer.is_valid():
            serializer.save()
            res = {'msg': 'Data Updated'}
            json_data = JSONRenderer().render(res)
            return HttpResponse(json_data, content_type=
        else:
            json_data = JSONRenderer().render(serializer
            return HttpResponse(json_data, content_type=

    def delete(self, request, *args, **kwargs):
        json_data = request.body
```


Khushal Savalakha

```
        stream = io.BytesIO(json_data)  # Corrected this
        pythondata = JSONParser().parse(stream)
        id = pythondata.get('id')
        stu = Student.objects.get(id=id)
        stu.delete()
        res = {'msg': 'Data Deleted!!'}
        return JsonResponse(res, safe=False)
```

**urls.py**

```
from django.contrib import admin
from django.urls import path
from api import views

urlpatterns = [
    path('admin/', admin.site.urls),
    path('studentapi/',views.StudentAPI.as_view()),
]
```

**myapp.py**

```
import requests
import json

URL="http://127.0.0.1:8000/studentapi/"

def get_data(id=None):
    data={}
    if id is not None:
        data={'id':id}
    json_data=json.dumps(data)
    r=requests.get(url=URL,data=json_data)
    data=r.json()
    print(data)

# get_data(1)
# print('-----------')
# get_data()
```


Khushal Savalakha

```
def post_data():
    data={
        'name':'Ravi',
        'roll':104,
        'city':'Dhanbad'
    }
    json_data=json.dumps(data)
    r=requests.post(url=URL,data=json_data)
    data=r.json()
    print(data)

# post_data()


def update_data():
    data={
        'id':6,
        'name':'Rohit',
        'roll':104,
        'city':'Ranchi'
    }
    json_data=json.dumps(data)
    r=requests.put(url=URL,data=json_data)
    data=r.json()
    print(data)

# update_data()

def delete_data():
    data={
        'id':6,
    }
    json_data=json.dumps(data)
    r=requests.delete(url=URL,data=json_data)
    data=r.json()
    print(data)
```


Khushal Savalakha

```
delete_data()
```

## Solution:

To fix this error, you need to create an instance of `JSONParser` and then call the `parse()` method on that instance.

Here's the corrected code for handling the `DELETE` request:

```
if request.method == 'DELETE':
    json_data = request.body
    stream = io.BytesIO(json_data)
    pythondata = JSONParser().parse(stream)  # Correct
ed this line
    id = pythondata.get('id')

    # Retrieve the student object and delete it
    try:
        stu = Student.objects.get(id=id)
        stu.delete()
        res = {'msg': 'Data Deleted!!'}
    except Student.DoesNotExist:
        res = {'msg': 'Student not found'}

    json_data = JSONRenderer().render(res)
    return HttpResponse(json_data, content_type='appli
cation/json')
```

## Explanation:

- **`JSONParser().parse(stream)`**: This correctly creates an instance of `JSONParser` and calls its `parse()` method with the `stream` as an argument, converting the JSON byte stream into a Python dictionary.
- **Exception Handling**: The code includes a `try-except` block to handle the case where the `Student` object with the given `id` does not exist. This prevents the server from crashing if an invalid `id` is provided.


Khushal Savalakha

## Summary:

This change ensures that the `DELETE` request is correctly handled by parsing the incoming JSON data, retrieving the appropriate student object, deleting it, and responding with an appropriate message.

> 💡 by above CRUD opertion i am able to cnnect react and django ?

Yes, the code you provided is a basic CRUD (Create, Read, Update, Delete) API built using Django. This API can be connected to a React frontend for data exchange. Here's a brief overview of how you can connect your React frontend with this Django backend:

## 1. Django REST API (Backend)

- **GET**: Fetches data from the database (e.g., list of students or a single student by ID).
- **POST**: Creates new data in the database (e.g., adding a new student).
- **PUT**: Updates existing data in the database (e.g., updating student details).
- **DELETE**: Deletes data from the database (e.g., removing a student by ID).

## 2. React (Frontend)

React will interact with your Django API using HTTP requests. Here’s how each operation can be implemented:

- **GET Request**: To fetch data from the Django API, you would use `fetch` or a library like `axios` in React.
- **POST Request**: To send data to the Django API to create a new record, you would again use `fetch` or `axios` with a `POST` request.
- **PUT Request**: For updating data, you would use a `PUT` request, sending the updated data along with the specific ID.
- **DELETE Request**: To delete a record, you would use a `DELETE` request, specifying the ID of the record to be deleted.

## 3. Example Code in React


Khushal Savalakha

Below is a simple example of how you can use `fetch` to interact with your Django API:

- **Fetching Data (GET):**

```
useEffect(() => {
  fetch('<http://127.0.0.1:8000/studentapi/>')
    .then(response => response.json())
    .then(data => console.log(data))
    .catch(error => console.error('Error:', erro
r));
}, []);
```

- **Creating Data (POST):**

```
const createStudent = () => {
  fetch('<http://127.0.0.1:8000/studentapi/>', {
    method: 'POST',
    headers: {
      'Content-Type': 'application/json',
    },
    body: JSON.stringify({
      name: 'John Doe',
      age: 25,
      grade: 'A',
    }),
  })
    .then(response => response.json())
    .then(data => console.log('Success:', data))
    .catch(error => console.error('Error:', erro
r));
};
```

- **Updating Data (PUT):**

```
const updateStudent = (id) => {
  fetch(`http://127.0.0.1:8000/studentapi/`, {
    method: 'PUT',
    headers: {
```


Khushal Savalakha

```
      'Content-Type': 'application/json',
    },
    body: JSON.stringify({
      id: id,
      name: 'Jane Doe',
      age: 26,
      grade: 'B',
    }),
  })
    .then(response => response.json())
    .then(data => console.log('Success:', data))
    .catch(error => console.error('Error:', erro
r));
};
```

- **Deleting Data (DELETE):**

```
const deleteStudent = (id) => {
  fetch(`http://127.0.0.1:8000/studentapi/`, {
    method: 'DELETE',
    headers: {
      'Content-Type': 'application/json',
    },
    body: JSON.stringify({ id: id }),
  })
    .then(response => response.json())
    .then(data => console.log('Deleted:', data))
    .catch(error => console.error('Error:', erro
r));
};
```

## 4. Cross-Origin Resource Sharing (CORS)

To enable React to communicate with Django on different ports (e.g., React on `localhost:3000` and Django on `localhost:8000`), you might need to configure CORS in your Django settings:

1. Install the CORS headers package:


Khushal Savalakha

```
pip install django-cors-headers
```

2. Add it to your `INSTALLED_APPS` and `MIDDLEWARE` in `settings.py`:

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

MIDDLEWARE = [
    ...
    'corsheaders.middleware.CorsMiddleware',
    ...
]
```

3. Allow CORS for all domains or specific domains:

```
CORS_ALLOWED_ORIGINS = [
    "<http://localhost:3000>",
]
```

## 5. Testing the Integration

- Start your Django server using `python manage.py runserver`.
- Start your React development server using `npm start` or `yarn start`.
- Test your API interactions from the React frontend.

With these steps, your React frontend should be able to communicate effectively with your Django backend, performing all CRUD operations seamlessly.


Khushal Savalakha

# ▼ API View

## ▼ Function Based API View

### Function Based api_view

This wrapper provide a few bits of functionality such as making sure you receive Request instances in your view, and adding context to Response objects so that content negotiation can be performed.

The wrapper also provide behaviour such as returning 405 Method Not Allowed responses when appropriate, and handling any ParseError exceptions that occur when accessing request.data with malformed input.

By default only GET methods will be accepted. Other methods will respond with "405 Method Not Allowed".

```
@api_view()

@api_view(['GET', 'POST', 'PUT', 'DELETE'])
def function_name(request):
    ……————————————————
    ……————————————————
```

### api_view

```
from rest_framework.decorators import api_view
from rest_framework.response import Response
@api_view(['GET'])
def student_list(request):
    if request.method == 'GET':
        stu = Student.objects.all()
        serializer = StudentSerializer(stu, many=True)
        return Response(serializer.data)
```


Khushal Savalakha

# api_view

```
from rest_framework.decorators import api_view
from rest_framework.response import Response
from rest_framework import status
@api_view(['POST'])
def student_create(request):
  if request.method == 'POST':
      serializer = StudentSerializer(data = request.data)
      if serializer.is_valid():
          serializer.save()
          res = {'msg': 'Data Created'}
         return Response(res, status=status.HTTP_201_CREATED)
      return Response(serializer.error, status=status.HTTP_400_BAD_REQUEST)
```

# Methods

- GET
- POST
- PUT
- PATCH
- DELETE

**Patch** :refers to an HTTP method used for making partial updates to a resource. Unlike PUT which replaces the entire resource, PATCH allows modifying specific fields of an existing resource without affecting other parts.


Khushal Savalakha

# Request

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.

request.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 in the same way that you handle incoming form data.

In Django REST Framework (DRF), `request.data` and `request.method` are used to handle incoming data and identify the type of HTTP request being made.

1. `request.data`:
   - **Purpose**: It contains the parsed data of the request body. This is typically used to handle data sent in **POST**, **PUT**, or **PATCH** requests.
   - **Usage**: When a client sends data (like JSON, form data, etc.) to the server, you can access that data using `request.data`.
   - **Example**:

     ```
     from rest_framework.response import Response
     from rest_framework.decorators import api_view

     @api_view(['POST'])
     def create_user(request):
         # Access the data sent in the request body
         username = request.data.get('username')
         email = request.data.get('email')
         # Perform user creation logic here
         return Response({"message": "User created",
     "username": username})
     ```

   - **Note**: Unlike `request.POST`, which only handles form data, `request.data` can handle multiple data formats (e.g., JSON, form data, etc.).


Khushal Savalakha

2. `request.method`:
   - **Purpose**: It holds the HTTP method used to make the request (e.g., **GET**, **POST**, **PUT**, **DELETE**, etc.).
   - **Usage**: You can use `request.method` to check the type of the request and handle different behaviors for different HTTP methods.
   - **Example**:

```
@api_view(['GET', 'POST'])
def user_view(request):
    if request.method == 'GET':
        return Response({"message": "GET request
received"})
    elif request.method == 'POST':
        return Response({"message": "POST reques
t received"})
```

In summary:

- `request.data`: Used to access the request body data in formats like JSON or form-encoded data.
- `request.method`: Used to determine the type of HTTP request (like GET, POST, PUT, DELETE) being made.

> 💡 it means requests.data convert json data in python native data type ?

Yes, exactly!

In Django REST Framework (DRF), `request.data` automatically converts the incoming **JSON data** (or other formats) from the request body into **Python native data types** like dictionaries, lists, strings, etc.

For example, when a client sends a JSON payload like this:

```
{
    "name": "John",
```


Khushal Savalakha

```
    "age": 25
}
```

Using `request.data`, DRF will automatically parse that JSON data into a Python dictionary:

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

## Key Points:

- **JSON** data is converted into Python native types: `str`, `int`, `list`, `dict`, etc.
- DRF handles this conversion internally, so you don't need to manually parse the JSON.

## Example:

```
@api_view(['POST'])
def create_user(request):
    # Assuming the incoming JSON data looks like {"nam
e": "John", "age": 25}
    data = request.data  # This will be a Python dicti
onary
    name = data.get('name')  # "John"
    age = data.get('age')  # 25

    # You can now use this data to create a user or pe
rform other operations
    return Response({"message": "User created", "nam
e": name, "age": age})
```

So, `request.data` acts as a bridge between the JSON data sent by the client and Python's native data types for easy processing.


Khushal Savalakha

# Request

request.method – request.method returns the uppercased string representation of the request's HTTP method.

Browser-based PUT, PATCH and DELETE forms are transparently supported.

request.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.

# 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.

Response objects are initialized 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.

Response class simply provides a nicer interface for returning content-negotiated Web API responses, that can be rendered to multiple formats.

Syntax:- Response(data, status=None, template_name=None, headers=None, content_type=None)

- data: The unrendered, serialized data for the response.
- status: A status code for the response. Defaults to 200.
- template_name: A template name to use only if HTMLRenderer or some other custom template renderer is the accepted renderer for the response.
- 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.

In Django REST Framework (DRF), the `Response()` function is used to return data to the client in a standardized format, typically in **JSON** or another content type (like XML). It is similar to Django's standard `HttpResponse` but offers more features, particularly for working with APIs.

## Key Features of `Response()`:

1. **Automatic Content Negotiation**:
   - It automatically formats the data (e.g., JSON, XML) based on the client's request and the `Accept` headers.
   - For example, if the client requests `application/json`, DRF will return the response as JSON.
2. **Data Serialization**:


Khushal Savalakha

- `Response()` can accept native Python data types (like `dict`, `list`) and convert them into the appropriate format (e.g., JSON).
- You don't need to manually serialize data into JSON; `Response()` handles that for you.

3. **Status Codes**:
   - You can specify an HTTP status code (e.g., 200 OK, 404 Not Found, 201 Created) as an optional argument.
   - By default, it returns a 200 status code for success, but you can change it if necessary.

## Syntax:

```
from rest_framework.response import Response

# Syntax for using Response()
Response(data, status=None, template_name=None, header
s=None, content_type=None)
```

- `data`: The data to be returned, usually in the form of a dictionary or list. DRF will serialize this into JSON (or another format).
- `status`: Optional HTTP status code (e.g., `status=200`, `status=201`).
- `headers`: Optional additional HTTP headers (e.g., `headers={'Custom-Header': 'value'}`).
- `content_type`: The type of content being returned, like `application/json` (optional, as it's usually inferred).

## Example:

```
from rest_framework.decorators import api_view
from rest_framework.response import Response

@api_view(['GET'])
def example_view(request):
    data = {
        "message": "Hello, World!",
        "status": "success"
```


Khushal Savalakha

```
    }
    return Response(data)  # Automatically returns as
JSON with status 200
```

### Specifying a Status Code:

```
@api_view(['POST'])
def create_user(request):
    # Assuming the user creation is successful
    data = {"message": "User created successfully"}
    return Response(data, status=201)  # Return 201 Cr
eated status
```

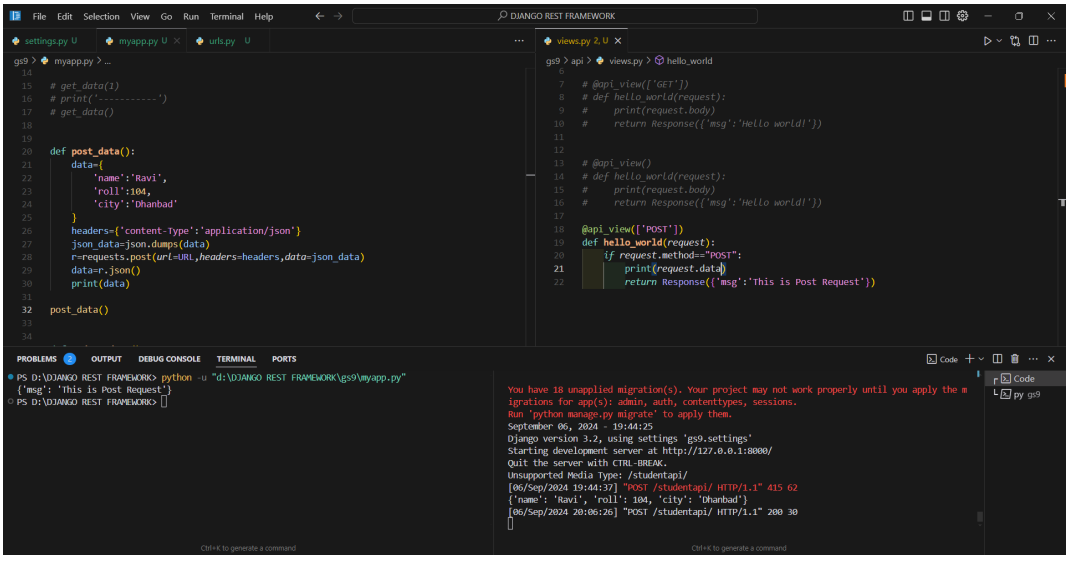
### Why Use `Response()` Instead of `HttpResponse()`?

- **Automatic Serialization**: You can directly return Python objects (lists, dictionaries) without manually converting them to JSON.
- **Content Negotiation**: It automatically decides the correct content type based on the request (JSON, XML, etc.).
- **More Features**: Easier handling of status codes, headers, and templates.

In summary, `Response()` is a convenient way to return API responses in Django REST Framework, handling data serialization, content negotiation, and status codes efficiently.

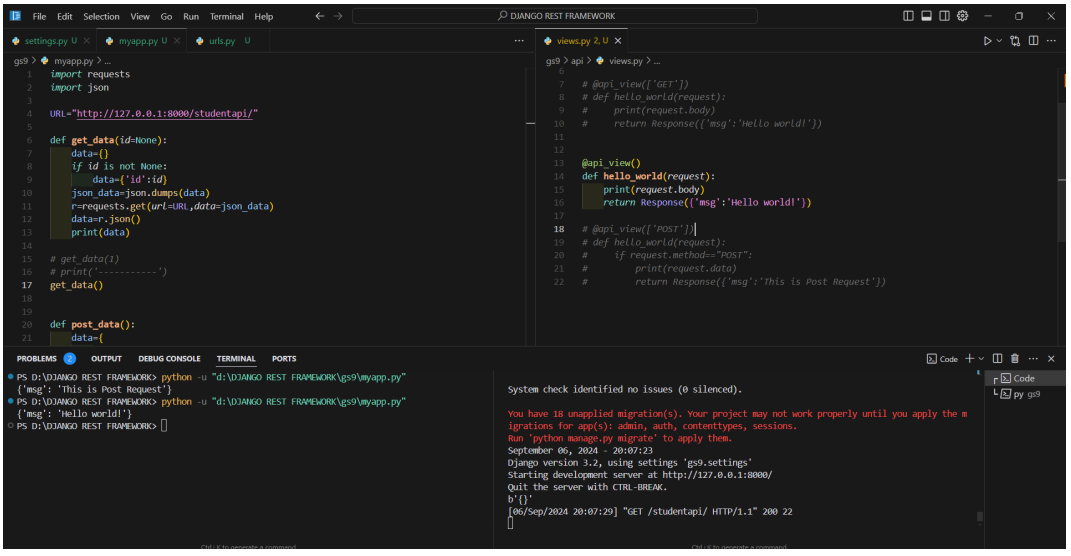
```
from django.shortcuts import render
from rest_framework.response import Response
from rest_framework.decorators import api_view
@api_view(['POST'])
def hello_world(request):
    if request.method=="POST":
        print(request.data)
        return Response({'msg':'This is Post Request'})
```


Khushal Savalakha




we must have to write headers otherwise it will gives error.

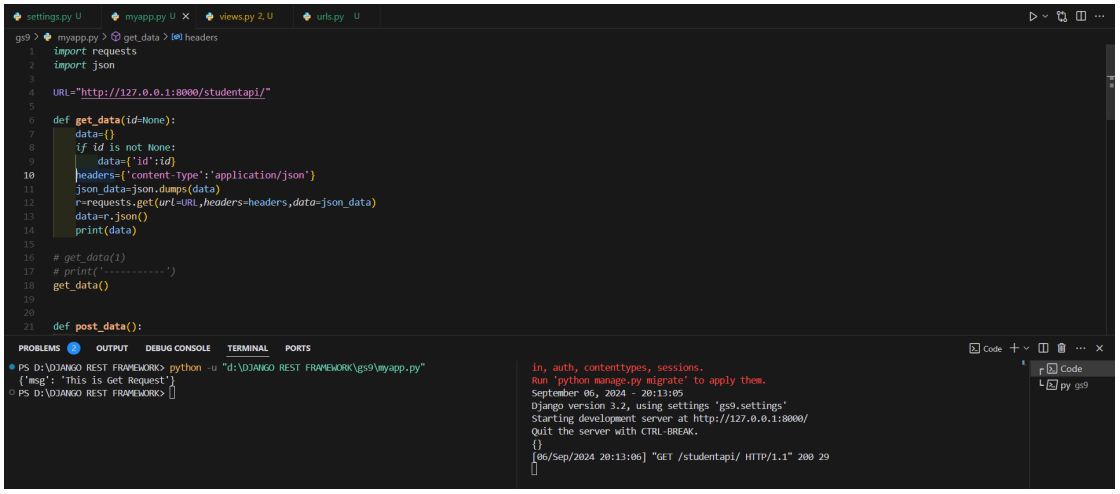
```
from django.shortcuts import render
from rest_framework.response import Response
from rest_framework.decorators import api_view
@api_view(['GET'])
def hello_world(request):
    print(request.body)
    return Response({'msg':'Hello world!'})
```

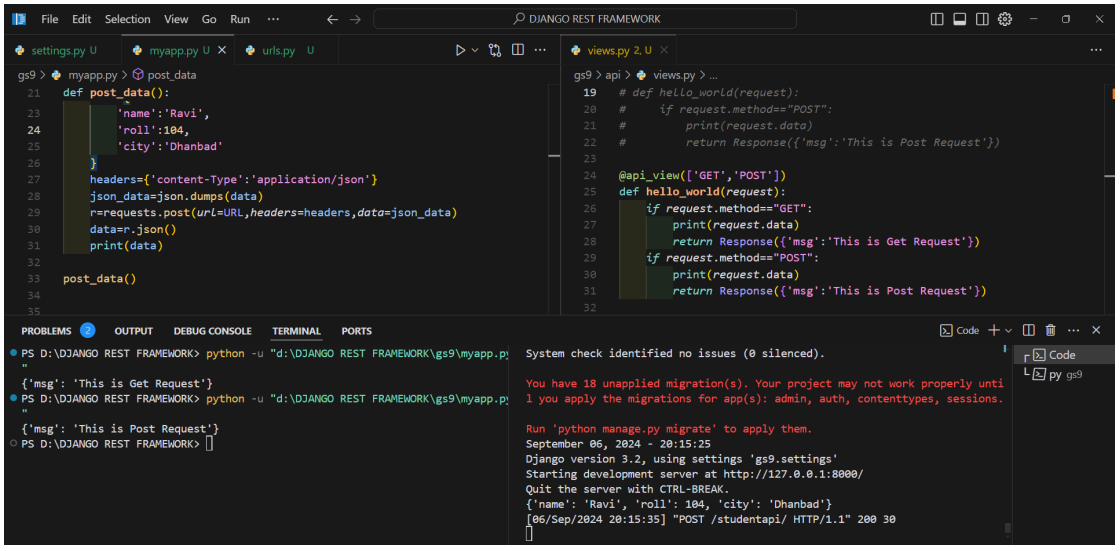




Khushal Savalakha

```python
from django.shortcuts import render
from rest_framework.response import Response
from rest_framework.decorators import api_view
@api_view(['GET','POST'])
def hello_world(request):
    if request.method=="GET":
        print(request.data)
        return Response({'msg':'This is Get Request'})
    if request.method=="POST":
        print(request.data)
        return Response({'msg':'This is Post Request'})
```





Khushal Savalakha

```python
from rest_framework.response import Response
from rest_framework.decorators import api_view
# Create your views here.


@api_view(['GET'])
def hello_world(request):
    print(request.body)
    return Response({'msg':'Hello world!'})


# @api_view()
# def hello_world(request):
#     print(request.body)
```

```python
def post_data():
        'name':'Ravi',
        'roll':104,
        'city':'Dhanbad'
    }
    headers={'content-Type':'application/json'}
    json_data=json.dumps(data)
    r=requests.post(url=URL,headers=headers,data=json_data)
    data=r.json()
    print(data)

post_data()
```

```
PS D:\DJANGO REST FRAMEWORK> python -u "d:\DJANGO REST FRAMEWORK\gs9\myapp.py
"
{'detail': 'Method "POST" not allowed.'}
PS D:\DJANGO REST FRAMEWORK>
```

```
in, auth, contenttypes, sessions.
Run 'python manage.py migrate' to apply them.
September 06, 2024 - 20:23:39
Django version 3.2, using settings 'gs9.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.
Method Not Allowed: /studentapi/
[06/Sep/2024 20:23:46] "POST /studentapi/ HTTP/1.1" 405 41
```

Hello World

GET /studentapi/

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

{
    "msg": "This is Get Request"
}
```

Media type: application/json

Content:


Khushal Savalakha

Hello World

# Hello World

OPTIONS GET

```
GET /studentapi/
```

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

{
    "msg": "This is Get Request"
}
```

**Media type:** application/json

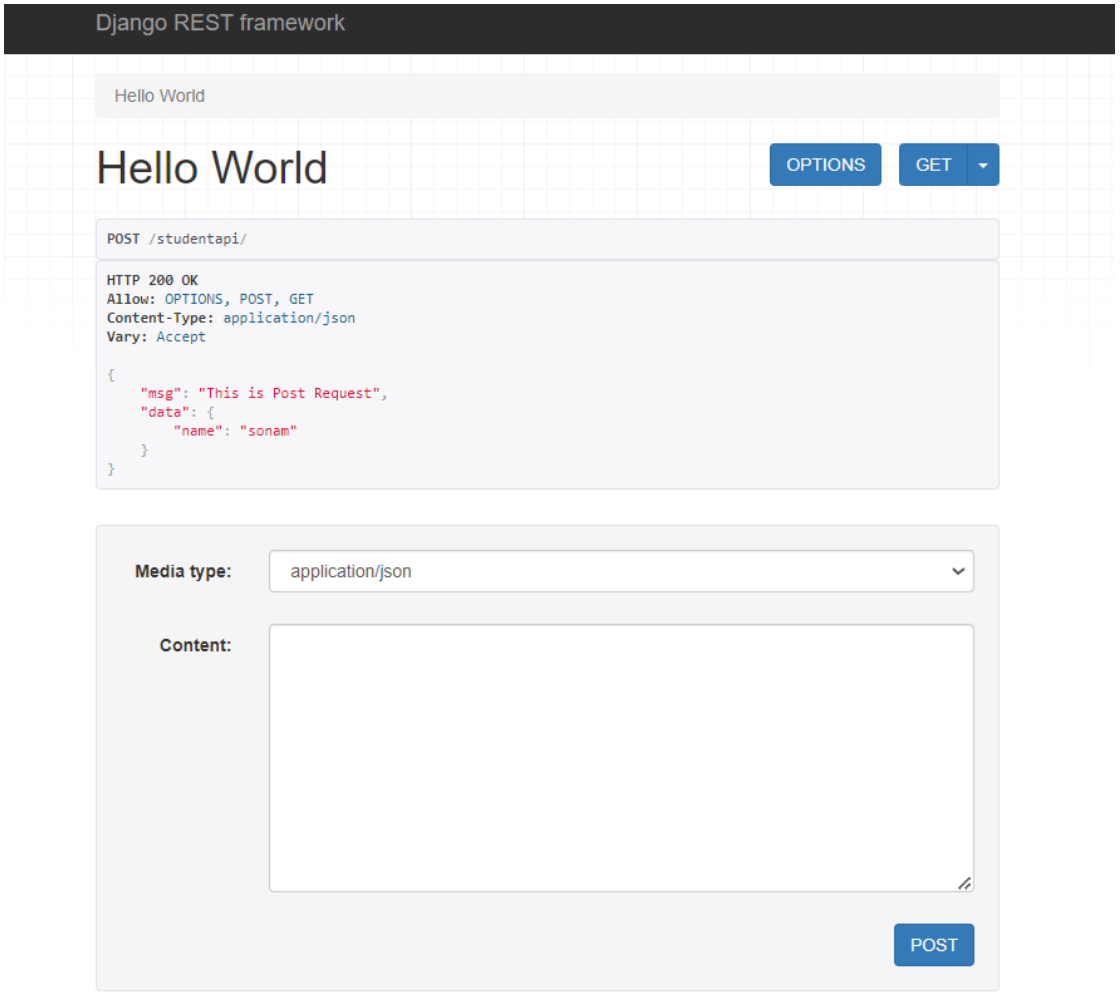
**Content:** {"name":"sonam"}

Make a POST request on the Hello World resource

POST



Khushal Savalakha

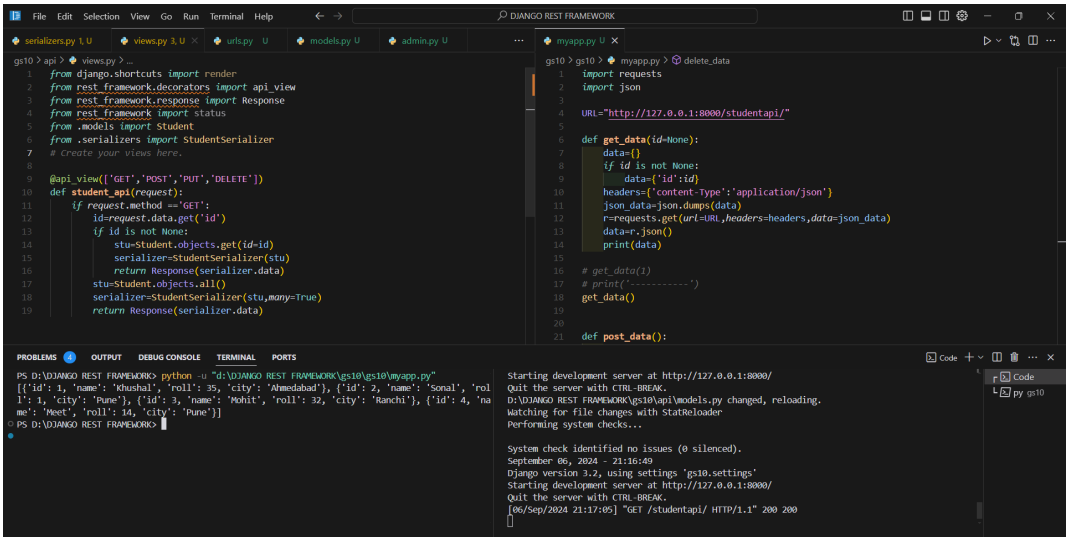



**views.py**

```
from django.shortcuts import render
from rest_framework.decorators import api_view
from rest_framework.response import Response
from rest_framework import status
from .models import Student
from .serializers import StudentSerializer
# Create your views here.

@api_view(['GET','POST','PUT','DELETE'])
def student_api(request):
    if request.method =='GET':
        id=request.data.get('id')
        if id is not None:
            stu=Student.objects.get(id=id)
```


Khushal Savalakha

```
            serializer=StudentSerializer(stu)
            return Response(serializer.data)
        stu=Student.objects.all()
        serializer=StudentSerializer(stu,many=True)
        return Response(serializer.data)
```

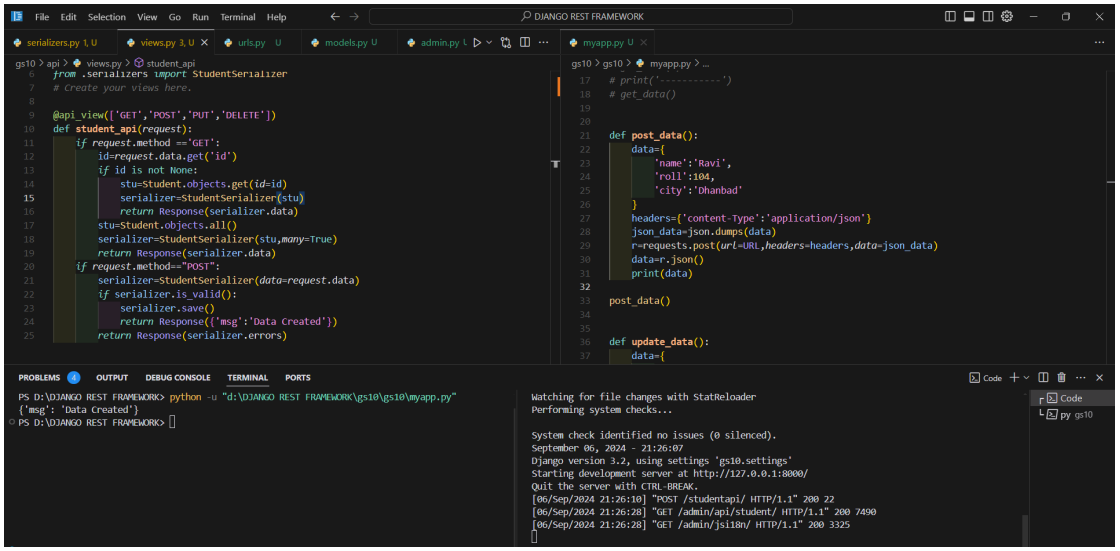


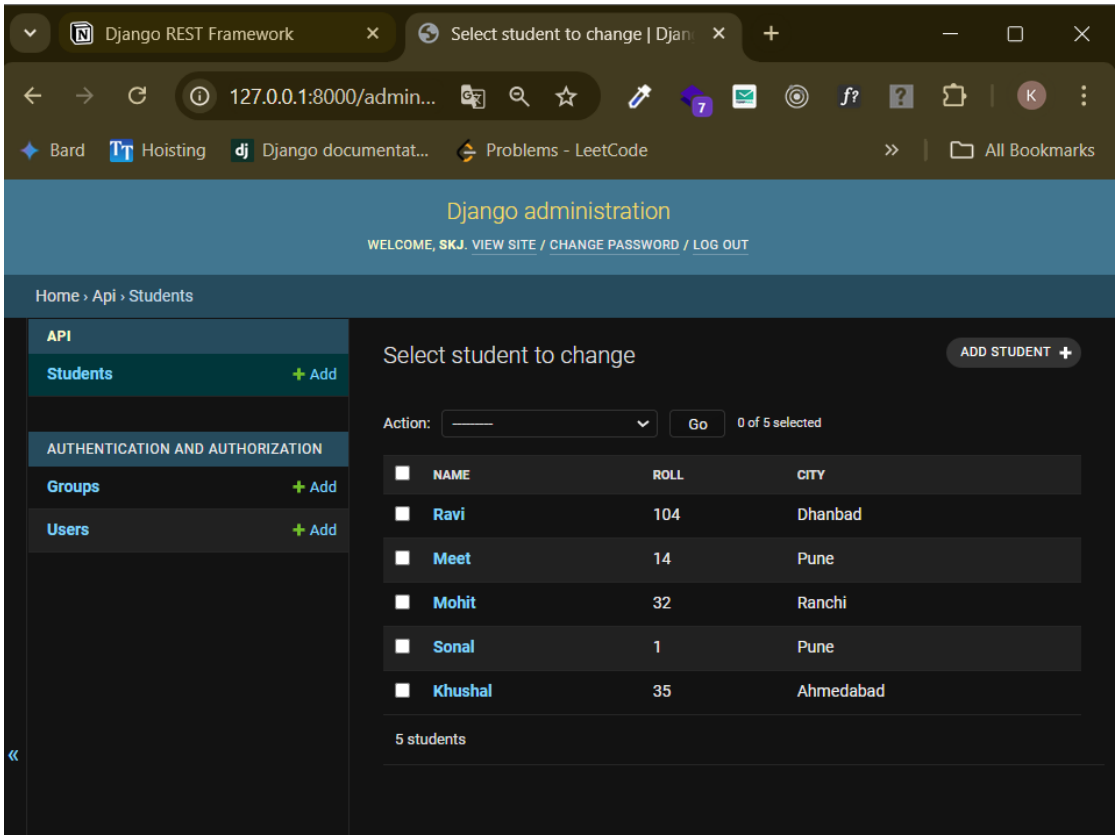

**views.py**

```
from django.shortcuts import render
from rest_framework.decorators import api_view
from rest_framework.response import Response
from rest_framework import status
from .models import Student
from .serializers import StudentSerializer
# Create your views here.

@api_view(['GET','POST','PUT','DELETE'])
def student_api(request):
    if request.method =='GET':
        id=request.data.get('id')
        if id is not None:
            stu=Student.objects.get(id=id)
            serializer=StudentSerializer(stu)
            return Response(serializer.data)
        stu=Student.objects.all()
        serializer=StudentSerializer(stu,many=True)
```


Khushal Savalakha

```
        return Response(serializer.data)
    if request.method=="POST":
        serializer=StudentSerializer(data=request.data)
        if serializer.is_valid():
            serializer.save()
            return Response({'msg':'Data Created'})
        return Response(serializer.errors)
```





Khushal Savalakha

```
def student_api(request):
        return Response(serializer.data)
    stu=Student.objects.all()
    serializer=StudentSerializer(stu,many=True)
    return Response(serializer.data)
if request.method=="POST":
    serializer=StudentSerializer(data=request.data)
    if serializer.is_valid():
        serializer.save()
        return Response({'msg':'Data Created'})
    return Response(serializer.errors)
if request.method=='PUT':
    id=request.data.get('id')
    stu=Student.objects.get(pk=id)
    serializer=StudentSerializer(stu,data=request.data,partial=True)
    if serializer.is_valid():
        return Response({'msg':'Data Updated'})
    return Response(serializer.errors)
```

```
def update_data():
    data={
        'id':5,
        'name':'Rohit',
        'roll':104,
        'city':'Ranchi'
    }
    headers={'content-Type':'application/json'}
    json_data=json.dumps(data)
    r=requests.put(url=URL,headers=headers,data=json_data)
    data=r.json()
    print(data)

update_data()

def delete_data():
    data={
        'id':6,
    }
    json_data=json.dumps(data)
    r=requests.delete(url=URL,data=json_data)
```

```
PS D:\DJANGO REST FRAMEWORK> python -u "d:\DJANGO REST FRAMEWORK\gs10\gs10\myapp.py"
{'msg': 'Data Updated'}
PS D:\DJANGO REST FRAMEWORK>
```

```
UnboundLocalError: local variable 'stu' referenced before assignment
[06/Sep/2024 21:35:42] "PUT /studentapi/ HTTP/1.1" 500 93909
D:\DJANGO REST FRAMEWORK\gs10\api\views.py changed, reloading.
Watching for file changes with StatReloader
Performing system checks...

System check identified no issues (0 silenced).
September 06, 2024 - 21:37:22
Django version 3.2, using settings 'gs10.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.
[06/Sep/2024 21:37:27] "PUT /studentapi/ HTTP/1.1" 200 22
```

Select student to change

Action: ——— Go 0 of 5 selected

| | ID | NAME | ROLL | CITY |
|---|---|---|---|---|
| ☐ | 5 | Rohit | 104 | Ranchi |
| ☐ | 4 | Meet | 14 | Pune |
| ☐ | 3 | Mohit | 32 | Ranchi |
| ☐ | 2 | Sonal | 1 | Pune |
| ☐ | 1 | Khushal | 35 | Ahmedabad |

5 students

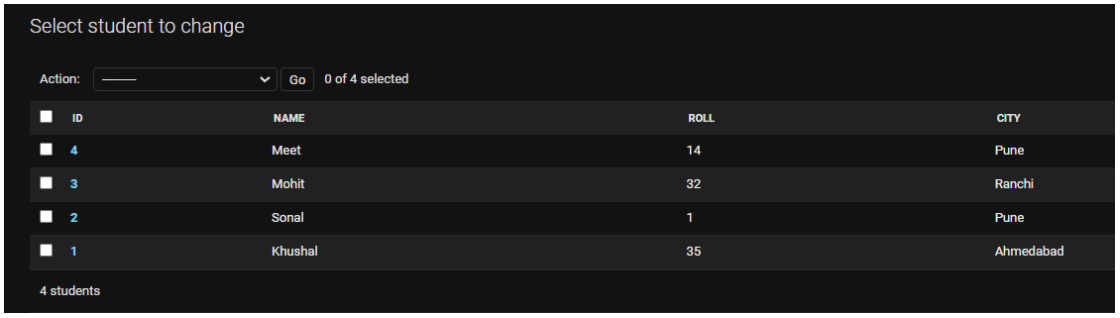
```
def student_api(request):
        return Response({'msg':'Data Created'})
    return Response(serializer.errors)
if request.method=='PUT':
    id=request.data.get('id')
    stu=Student.objects.get(pk=id)
    serializer=StudentSerializer(stu,data=request.data,partial=True)
    if serializer.is_valid():
        serializer.save()
        return Response({'msg':'Data Updated'})
    return Response(serializer.errors)
if request.method=='DELETE':
    id=request.data.get('id')
    stu=Student.objects.get(pk=id)
    stu.delete()
    return Response({'msg':'Data Deleted'})
```

```
def delete_data():
    data={
        'id':5,
    }
    headers={'content-Type':'application/json'}
    json_data=json.dumps(data)
    r=requests.delete(url=URL,headers=headers,data=json_data)
    data=r.json()
    print(data)

delete_data()
```

```
PS D:\DJANGO REST FRAMEWORK> python -u "d:\DJANGO REST FRAMEWORK\gs10\gs10\myapp.py"
{'msg': 'Data Deleted'}
PS D:\DJANGO REST FRAMEWORK>
```

```
Watching for file changes with StatReloader
Performing system checks...

System check identified no issues (0 silenced).
September 06, 2024 - 21:46:03
Django version 3.2, using settings 'gs10.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.
[06/Sep/2024 21:46:17] "DELETE /studentapi/ HTTP/1.1" 200 22
[06/Sep/2024 21:46:24] "GET /admin/api/student/ HTTP/1.1" 200 7475
[06/Sep/2024 21:46:24] "GET /admin/jsi18n/ HTTP/1.1" 200 3325
```


Khushal Savalakha




```
from django.shortcuts import render
from rest_framework.decorators import api_view
from rest_framework.response import Response
from rest_framework import status
from .models import Student
from .serializers import StudentSerializer
# Create your views here.

@api_view(['GET','POST','PUT','DELETE'])
def student_api(request):
    if request.method =='GET':
        id=request.data.get('id')
        if id is not None:
            stu=Student.objects.get(id=id)
            serializer=StudentSerializer(stu)
            return Response(serializer.data)
        stu=Student.objects.all()
        serializer=StudentSerializer(stu,many=True)
        return Response(serializer.data)
    if request.method=="POST":
        serializer=StudentSerializer(data=request.data)
        if serializer.is_valid():
            serializer.save()
            return Response({'msg':'Data Created'})
        return Response(serializer.errors)
    if request.method=='PUT':
        id=request.data.get('id')
        stu=Student.objects.get(pk=id)
        serializer=StudentSerializer(stu,data=request.da
        if serializer.is_valid():
```

```
            serializer.save()
            return Response({'msg':'Data Updated'})
        return Response(serializer.errors)
    if request.method=='DELETE':
        id=request.data.get('id')
        stu=Student.objects.get(pk=id)
        stu.delete()
        return Response({'msg':'Data Deleted'})
```

Certainly! Let's break down the `@api_view` decorator and then walk through the complete code for your `student_api` view.

## `@api_view` Decorator

The `@api_view` decorator is part of the Django REST Framework (DRF). It is used to mark a function-based view as an API view, allowing it to handle HTTP requests such as GET, POST, PUT, and DELETE.

**Key Points:**

- **Purpose**: It simplifies the creation of views for API endpoints by automatically providing request and response handling specific to REST APIs.
- **Arguments**: It takes a list of HTTP methods that the view will handle, such as `['GET', 'POST', 'PUT', 'DELETE']`. The view function will only respond to those specified methods.

## Full Code Explanation

Here's a step-by-step breakdown of your `student_api` view:

```
from django.shortcuts import render
from rest_framework.decorators import api_view
from rest_framework.response import Response
from rest_framework import status
from .models import Student
from .serializers import StudentSerializer
```

1. **Imports**:
    - `render`: A Django shortcut to render templates (not used in this code).
    - `api_view`: A decorator to specify HTTP methods allowed for the view.


Khushal Savalakha

- `Response`: A DRF class to return HTTP responses with JSON data.
- `status`: A module for HTTP status codes (e.g., 200 OK, 404 Not Found).
- `Student`: The model representing a student in the database.
- `StudentSerializer`: A serializer to convert model instances to JSON and vice versa.

```
@api_view(['GET','POST','PUT','DELETE'])
def student_api(request):
```

1. `@api_view` **Decorator**:
   - This decorator allows the `student_api` function to handle GET, POST, PUT, and DELETE requests.

```
    if request.method =='GET':
        id=request.data.get('id')
        if id is not None:
            stu=Student.objects.get(id=id)
            serializer=StudentSerializer(stu)
            return Response(serializer.data)
        stu=Student.objects.all()
        serializer=StudentSerializer(stu,many=True)
        return Response(serializer.data)
```

1. **GET Method**:
   - `id = request.data.get('id')`: Retrieves the `id` from the request data.
   - `if id is not None:`: Checks if an `id` was provided.
     - `stu = Student.objects.get(id=id)`: Retrieves a single student object with the given `id`.
     - `serializer = StudentSerializer(stu)`: Serializes the student object.
     - `return Response(serializer.data)`: Returns the serialized data as a JSON response.
   - `stu = Student.objects.all()`: Retrieves all student objects if no `id` is provided.


Khushal Savalakha

- `serializer = StudentSerializer(stu, many=True)`: Serializes the list of student objects.
- `return Response(serializer.data)`: Returns the serialized list as a JSON response.

```
    if request.method=="POST":
        serializer=StudentSerializer(data=request.dat
a)
        if serializer.is_valid():
            serializer.save()
            return Response({'msg':'Data Created'})
        return Response(serializer.errors)
```

1. **POST Method**:
   - `serializer = StudentSerializer(data=request.data)`: Initializes the serializer with the incoming data.
   - `if serializer.is_valid():`: Validates the data.
     - `serializer.save()`: Saves the valid data to the database.
     - `return Response({'msg': 'Data Created'})`: Returns a success message.
   - `return Response(serializer.errors)`: Returns validation errors if the data is invalid.

```
    if request.method=='PUT':
        id=request.data.get('id')
        stu=Student.objects.get(pk=id)
        serializer=StudentSerializer(stu,data=request.
data,partial=True)
        if serializer.is_valid():
            serializer.save()
            return Response({'msg':'Data Updated'})
        return Response(serializer.errors)
```

1. **PUT Method**:
   - `id = request.data.get('id')`: Retrieves the `id` from the request data.


Khushal Savalakha

- `stu = Student.objects.get(pk=id)`: Retrieves the student object with the given `id`.
- `serializer = StudentSerializer(stu, data=request.data, partial=True)`: Initializes the serializer with the updated data and specifies `partial=True` to allow partial updates.
- `if serializer.is_valid():`: Validates the updated data.
  - `serializer.save()`: Saves the updated data to the database.
  - `return Response({'msg': 'Data Updated'})`: Returns a success message.
- `return Response(serializer.errors)`: Returns validation errors if the update fails.

```
if request.method=='DELETE':
    id=request.data.get('id')
    stu=Student.objects.get(pk=id)
    stu.delete()
    return Response({'msg':'Data Deleted'})
```

1. **DELETE Method**:
   - `id = request.data.get('id')`: Retrieves the `id` from the request data.
   - `stu = Student.objects.get(pk=id)`: Retrieves the student object with the given `id`.
   - `stu.delete()`: Deletes the student object from the database.
   - `return Response({'msg': 'Data Deleted'})`: Returns a success message.

## Summary

- `GET`: Retrieve one or all student records.
- `POST`: Create a new student record.
- `PUT`: Update an existing student record.
- `DELETE`: Delete a student record.

Each method uses the `StudentSerializer` to handle serialization and validation of the data. The `Response` class is used to return data and status messages in JSON format.


Khushal Savalakha

💡 here it is necessory to use pk instead of this we can use id or not ?

Yes, you can use `id` instead of `pk` in Django, but it's important to understand the context in which these terms are used.

## `id` vs `pk`

- `id`: This is a common convention for primary key fields in models, but it's just a field name. In Django, it defaults to being the primary key if you don't specify otherwise.
- `pk`: This stands for "primary key" and is a more generic way to refer to the primary key of a model. It is a Django ORM attribute that works with whatever field is set as the primary key.

## When to Use Each

- **Using `id`:**
  - If your model's primary key field is explicitly named `id`, you can use `id` to refer to it.
  - Example: `Student.objects.get(id=1)`
- **Using `pk`:**
  - This is more flexible and works regardless of what the primary key field is named. It's a Django ORM attribute that refers to the primary key field, whatever it's called.
  - Example: `Student.objects.get(pk=1)`

## Code Examples

Both `id` and `pk` can be used in your views. Here's how you can use them interchangeably:

## Using `id`:

```
if request.method == 'GET':
    id = request.data.get('id')
    if id is not None:
        stu = Student.objects.get(id=id)
```


Khushal Savalakha

```
        serializer = StudentSerializer(stu)
        return Response(serializer.data)
    stu = Student.objects.all()
    serializer = StudentSerializer(stu, many=True)
    return Response(serializer.data)
```

### Using `pk`:

```
if request.method == 'GET':
    id = request.data.get('id')
    if id is not None:
        stu = Student.objects.get(pk=id)
        serializer = StudentSerializer(stu)
        return Response(serializer.data)
    stu = Student.objects.all()
    serializer = StudentSerializer(stu, many=True)
    return Response(serializer.data)
```

In summary, both `id` and `pk` are valid and can be used based on your preference or naming conventions. `pk` is often preferred in Django because it is more generic and works with any primary key field name, making the code more adaptable if the field name changes in the future.


Khushal Savalakha

Django REST framework skj

Student Api

# Student Api

DELETE OPTIONS GET

```
PUT /studentapi/2
```

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

{
    "name": [
        "This field is required."
    ]
}
```

**Media type:** application/json

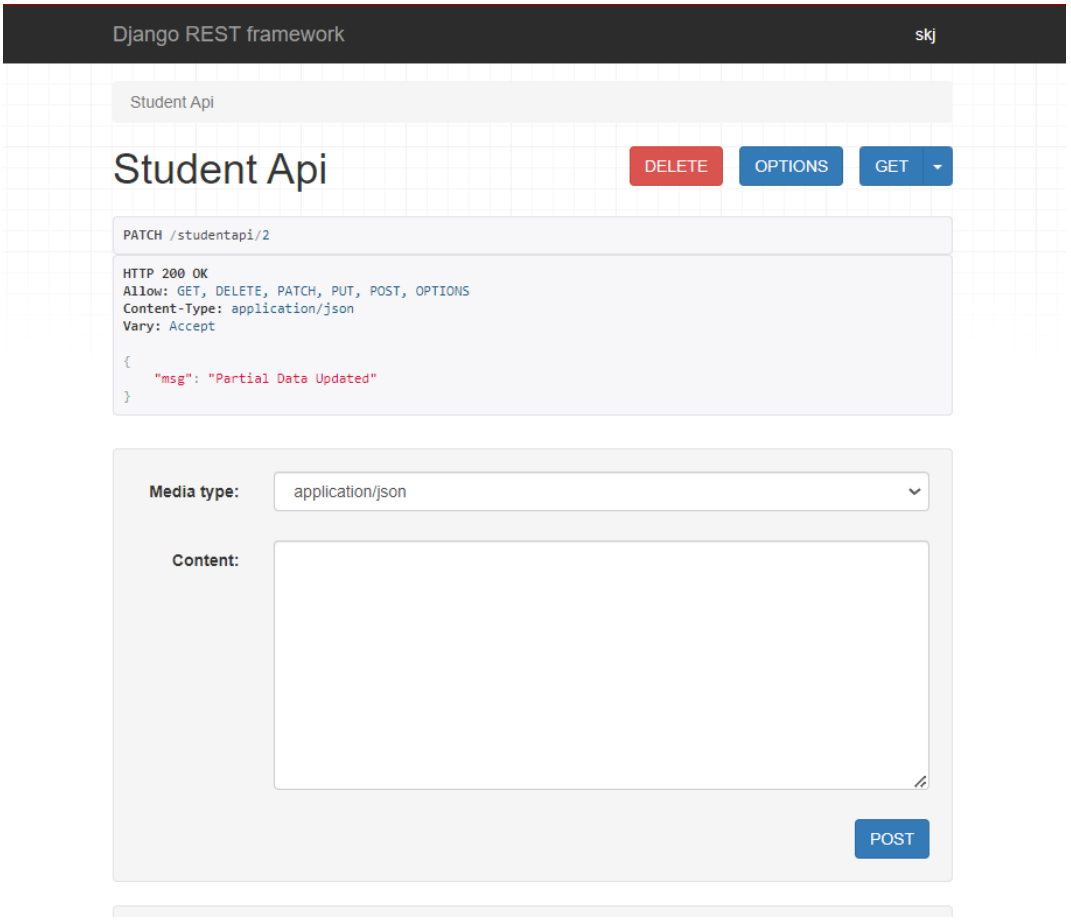
**Content:**

POST

**Media type:** application/json



Khushal Savalakha




## ▼ Class Based API VIew

# Class Based APIView

REST framework provides an APIView class, which subclasses Django's View class.

APIView classes are different from regular View classes in the following ways:

- Requests passed to the handler methods will be REST framework's Request instances, not Django's HttpRequest instances.
- Handler methods may return REST framework's Response, instead of Django's HttpResponse. The view will manage content negotiation and setting the correct renderer on the response.
- Any APIException exceptions will be caught and mediated into appropriate responses.
- Incoming requests will be authenticated and appropriate permission and/or throttle checks will be run before dispatching the request to the handler method.

# Class Based APIView

```
from rest_framework.views import APIView
class StudentAPI(APIView):
  def get(self, request, format=None):
      stu = Student.objects.all()
      serializer = StudentSerializer(stu, many=True)
      return Response(serializer.data)

   def post(self, request, format=None):
      serializer = StudentSerializer(data=request.data)
      if serializer.is_valid():
         serializer.save()
         return Response({'msg': 'Data Created' }, status=status.HTTP_201_CREATED)
      return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)
```

**urls.py**

```
from django.contrib import admin
from django.urls import path
from api import views

urlpatterns = [
    path('admin/', admin.site.urls),
    path('studentapi/',views.StudentAPI.as_view()),
    path('studentapi/<int:pk>',views.StudentAPI.as_view(
]
```

**views.py**

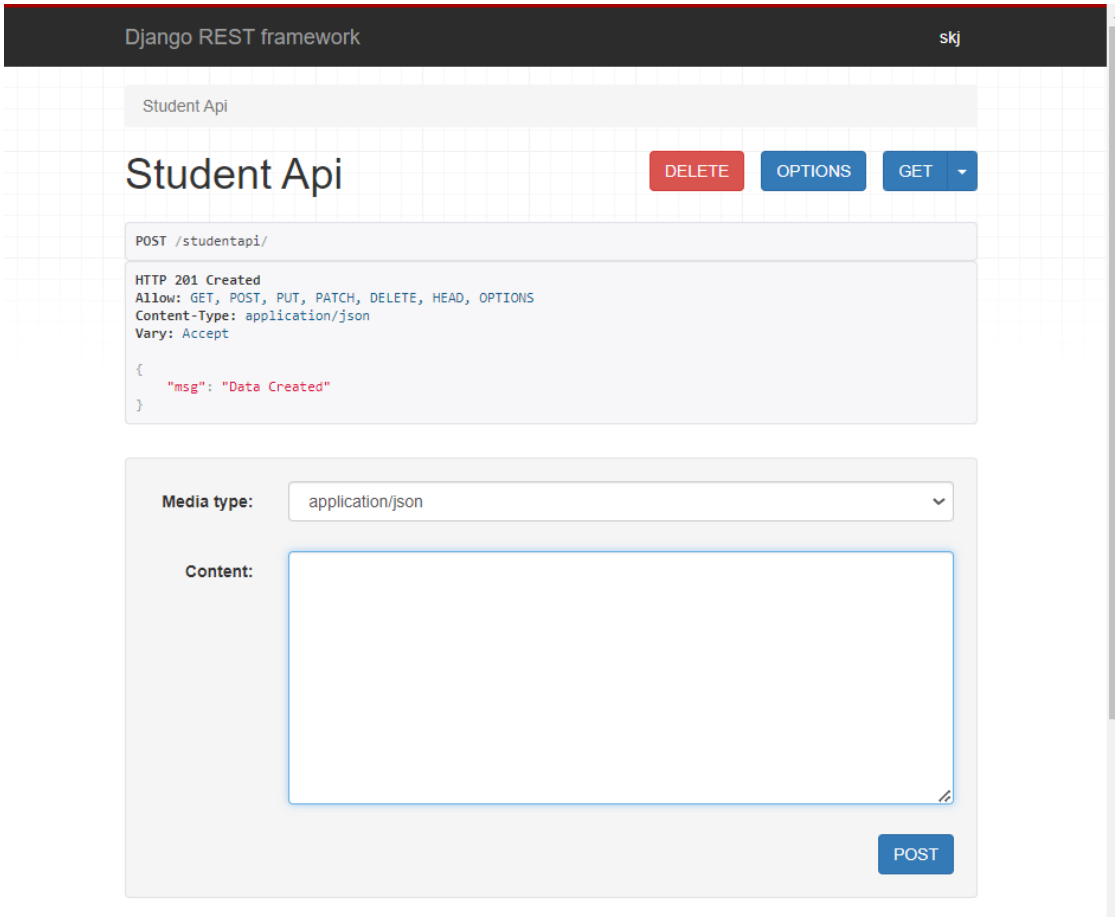
```
from django.shortcuts import render
from rest_framework.response import Response
from rest_framework import status
from .models import Student
from .serializers import StudentSerializer
from rest_framework.views import APIView
# Create your views here.


class StudentAPI(APIView):
    def get(self,request,pk=None,format=None):
```

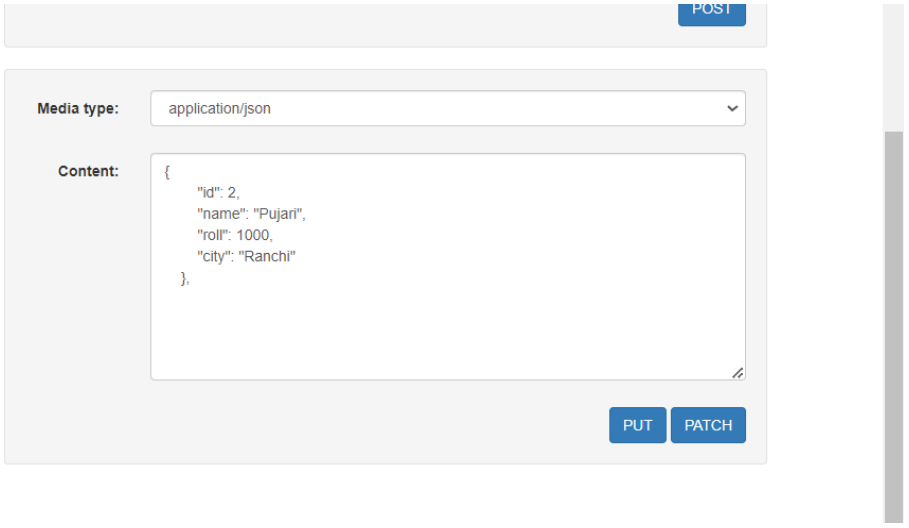

Khushal Savalakha

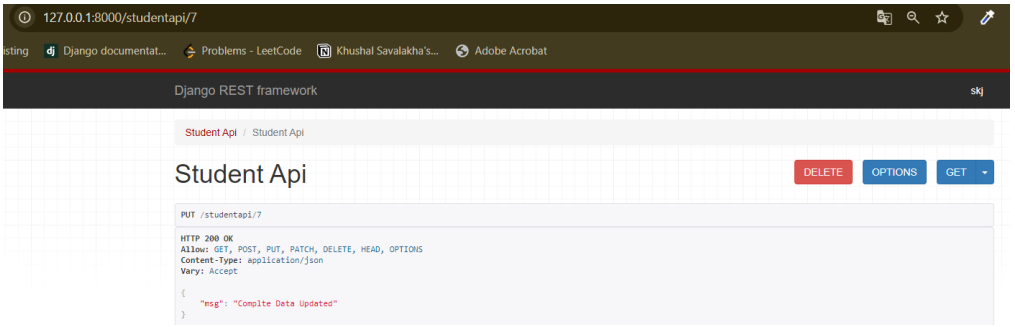
```
        id=pk
        if id is not None:
            stu=Student.objects.get(id=id)
            serializer=StudentSerializer(stu)
            return Response(serializer.data)
        stu=Student.objects.all()
        serializer=StudentSerializer(stu,many=True)
        return Response(serializer.data)

    def post(self,request,format=None):
        serializer=StudentSerializer(data=request.data)
        if serializer.is_valid():
            serializer.save()
            return Response({'msg':'Data Created'},statu
        return Response(serializer.errors,status=status.

    def put(self,request,pk,format=None):
        id=pk
        stu=Student.objects.get(pk=id)
        serializer=StudentSerializer(stu,data=request.da
        if serializer.is_valid():
            serializer.save()
            return Response({'msg':'Complte Data Updated
        return Response(serializer.errors,status=status.

    def patch(self,request,pk,format=None):
        id=pk
        stu=Student.objects.get(pk=id)
        serializer=StudentSerializer(stu,data=request.da
        if serializer.is_valid():
            serializer.save()
            return Response({'msg':'Partial Data Updated
        return Response(serializer.errors)

    def delete(self,request,pk,format=None):
        id=pk
        stu=Student.objects.get(pk=id)
```


Khushal Savalakha

```
        stu.delete()
        return Response({'msg':'Data Deleted'})
```



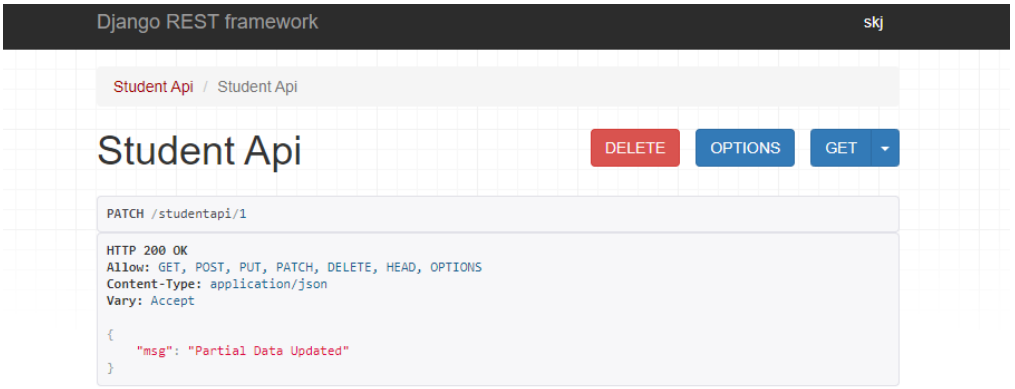



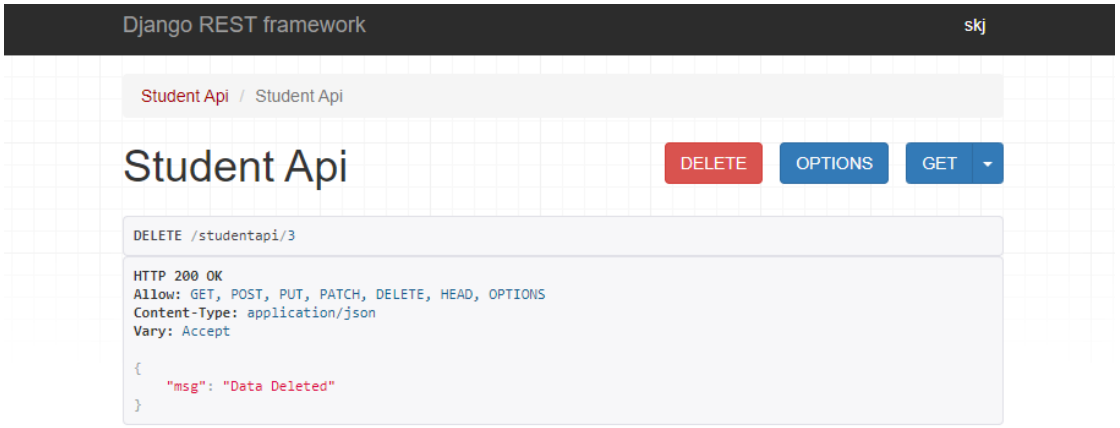



**views.py**

```
def put(self,request,format=None):
    id=request.data.get('id')
    stu=Student.objects.get(pk=id)
    serializer=StudentSerializer(stu,data=request.da
    if serializer.is_valid():
        serializer.save()
        return Response({'msg':'Complte Data Updated
    return Response(serializer.errors,status=status.
```

if we don't want to send id through pk.






## ▼ Generic APIView and Mixins

# GenericAPIView

This class extends REST framework's APIView class, adding commonly required behavior for standard list and detail views.

Attributes

queryset - The queryset that should be used for returning objects from this view. Typically, you must either set this attribute, or override the get_queryset() method. If you are overriding a view method, it is important that you call get_queryset() instead of accessing this property directly, as queryset will get evaluated once, and those results will be cached for all subsequent requests.

serializer_class - The serializer class that should be used for validating and deserializing input, and for serializing output. Typically, you must either set this attribute, or override the get_serializer_class() method.


Khushal Savalakha

# GenericAPIView

lookup_field - The model field that should be used to for performing object lookup of individual model instances. Defaults to 'pk'.

lookup_url_kwarg - The URL keyword argument that should be used for object lookup. The URL conf should include a keyword argument corresponding to this value. If unset this defaults to using the same value as lookup_field.

pagination_class - The pagination class that should be used when paginating list results. Defaults to the same value as the DEFAULT_PAGINATION_CLASS setting, which is 'rest_framework.pagination.PageNumberPagination'. Setting pagination_class=None will disable pagination on this view.

filter_backends - A list of filter backend classes that should be used for filtering the queryset. Defaults to the same value as the DEFAULT_FILTER_BACKENDS setting.

# GenericAPIView

Methods

get_queryset(self) - It returns the queryset that should be used for list views, and that should be used as the base for lookups in detail views. Defaults to returning the queryset specified by the queryset attribute.

This method should always be used rather than accessing self.queryset directly, as self.queryset gets evaluated only once, and those results are cached for all subsequent requests.

get_object(self) - It returns an object instance that should be used for detail views. Defaults to using the lookup_field parameter to filter the base queryset.

get_serializer_class(self) - It returns the class that should be used for the serializer. Defaults to returning the serializer_class attribute.


Khushal Savalakha

# GenericAPIView

get_serializer_context(self) – It returns a dictionary containing any extra context that should be supplied to the serializer. Defaults to including 'request', 'view' and 'format' keys.

get_serializer(self, instance=None, data=None, many=False, partial=False) – It returns a serializer instance.

get_paginated_response(self, data) – It returns a paginated style Response object.

paginate_queryset(self, queryset) - Paginate a queryset if required, either returning a page object, or None if pagination is not configured for this view.

# GenericAPIView

filter_queryset(self, queryset) - Given a queryset, filter it with whichever filter backends are in use, returning a new queryset.

# Mixins

One of the big wins of using class-based views is that it allows us to easily compose reusable bits of behaviour.

The create/retrieve/update/delete operations that we've been using so far are going to be pretty similar for any model-backed API views we create.

Those bits of common behaviour are implemented in REST framework's mixin classes.

The mixin classes provide the actions that are used to provide the basic view behavior.

Note that the mixin classes provide action methods rather than defining the handler methods, such as get() and post(), directly. This allows for more flexible composition of behavior.

The mixin classes can be imported from rest_framework.mixins


Khushal Savalakha

# Mixins

- ListModelMixin
- CreateModelMixin
- RetrieveModelMixin
- UpdateModelMixin
- DestroyModelMixin

# ListModelMixin

It provides a list(request, *args, **kwargs) method, that implements listing a queryset.

If the queryset is populated, this returns a 200 OK response, with a serialized representation of the queryset as the body of the response. The response data may optionally be paginated.

```
from rest_framework.mixins import ListModelMixin
from rest_framework.generics import GenericAPIView
```

# ListModelMixin

```
from rest_framework.mixins import ListModelMixin
from rest_framework.generics import GenericAPIView
class StudentList(ListModelMixin, GenericAPIView):
    queryset = Student.objects.all()
    serializer_class = StudentSerializer
    def get(self, request, *args, **kwargs):
        return self.list(request, *args, **kwargs)
```


Khushal Savalakha

# CreateModelMixin

It provides a create(request, *args, **kwargs) method, that implements creating and saving a new model instance.

If an object is created this returns a 201 Created response, with a serialized representation of the object as the body of the response. If the representation contains a key named url, then the Location header of the response will be populated with that value.

If the request data provided for creating the object was invalid, a 400 Bad Request response will be returned, with the error details as the body of the response.

# CreateModelMixin

```
from rest_framework.mixins import CreateModelMixin
from rest_framework.generics import GenericAPIView
class StudentCreate(CreateModelMixin, GenericAPIView):
    queryset = Student.objects.all()
    serializer_class = StudentSerializer
    def post(self, request, *args, **kwargs):
        return self.create(request, *args, **kwargs)
```

# RetrieveModelMixin

It provides a retrieve(request, *args, **kwargs) method, that implements returning an existing model instance in a response.

If an object can be retrieved this returns a 200 OK response, with a serialized representation of the object as the body of the response. Otherwise it will return a 404 Not Found.


Khushal Savalakha

# RetrieveModelMixin

```
from rest_framework.mixins import RetrieveModelMixin
from rest_framework.generics import GenericAPIView
class StudentRetrieve(RetrieveModelMixin, GenericAPIView):
    queryset = Student.objects.all()
    serializer_class = StudentSerializer
    def get(self, request, *args, **kwargs):
        return self.retrieve(request, *args, **kwargs)
```

# UpdateModelMixin

It provides a update(request, *args, **kwargs) method, that implements updating and saving an existing model instance.

It also provides a partial_update(request, *args, **kwargs) method, which is similar to the update method, except that all fields for the update will be optional. This allows support for HTTP PATCH requests.

If an object is updated this returns a 200 OK response, with a serialized representation of the object as the body of the response.

If the request data provided for updating the object was invalid, a 400 Bad Request response will be returned, with the error details as the body of the response.

# UpdateModelMixin

```
from rest_framework.mixins import UpdateModelMixin
from rest_framework.generics import GenericAPIView
class StudentUpdate(UpdateModelMixin, GenericAPIView):
    queryset = Student.objects.all()
    serializer_class = StudentSerializer
    def put(self, request, *args, **kwargs):
        return self.update(request, *args, **kwargs)
```


Khushal Savalakha

# DestroyModelMixin

It provides a destroy(request, *args, **kwargs) method, that implements deletion of an existing model instance.

If an object is deleted this returns a 204 No Content response, otherwise it will return a 404 Not Found.

# DestroyModelMixin

```
from rest_framework.mixins import DestroyModelMixin
from rest_framework.generics import GenericAPIView
class StudentDestroy(DestroyModelMixin, GenericAPIView):
    queryset = Student.objects.all()
    serializer_class = StudentSerializer
    def delete(self, request, *args, **kwargs):
        return self.destroy(request, *args, **kwargs)
```

```
#GenericAPIView and Model Mixin
from .models import Student
from .serializers import StudentSerializer
from rest_framework.generics import  GenericAPIView
from rest_framework.mixins import ListModelMixin,CreateMode


class StudentList(GenericAPIView,ListModelMixin):
    queryset=Student.objects.all()
    serializer_class=StudentSerializer
    def get(self,request,*args,**kwargs):
        return self.list(request,*args,**kwargs)

class StudentCreate(GenericAPIView,CreateModelMixin):
    queryset=Student.objects.all()
    serializer_class=StudentSerializer
    def post(self,request,*args,**kwargs):
        return self.create(request,*args,**kwargs)
```


Khushal Savalakha

Django REST framework

Student Create

# Student Create

OPTIONS

```
POST /studentapi/
```

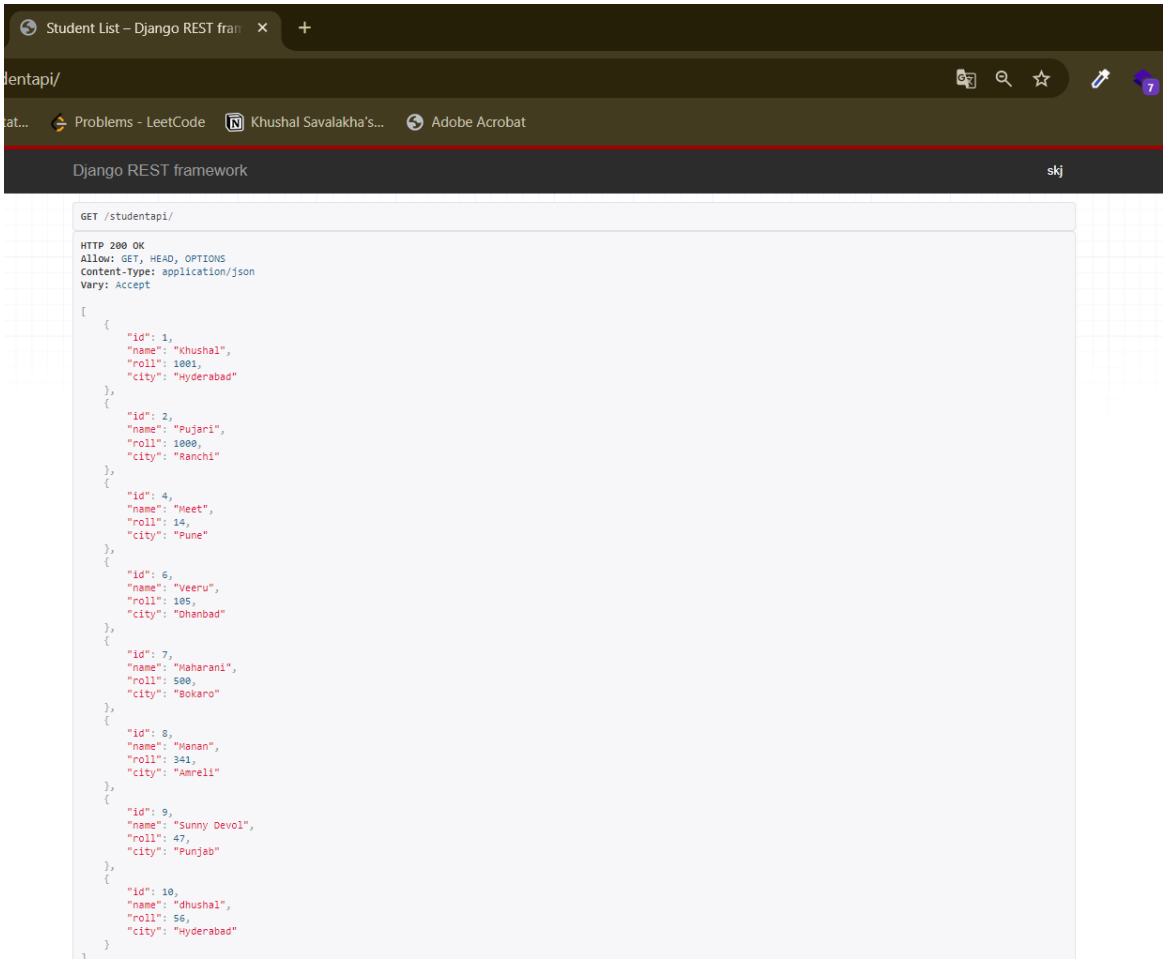
```
HTTP 201 Created
Allow: POST, OPTIONS
Content-Type: application/json
Vary: Accept

{
    "id": 10,
    "name": "dhushal",
    "roll": 56,
    "city": "Hyderabad"
}
```

Raw data | HTML form

| | |
|---|---|
| **Name** | dhushal |
| **Roll** | 56 |
| **City** | Hyderabad |

POST

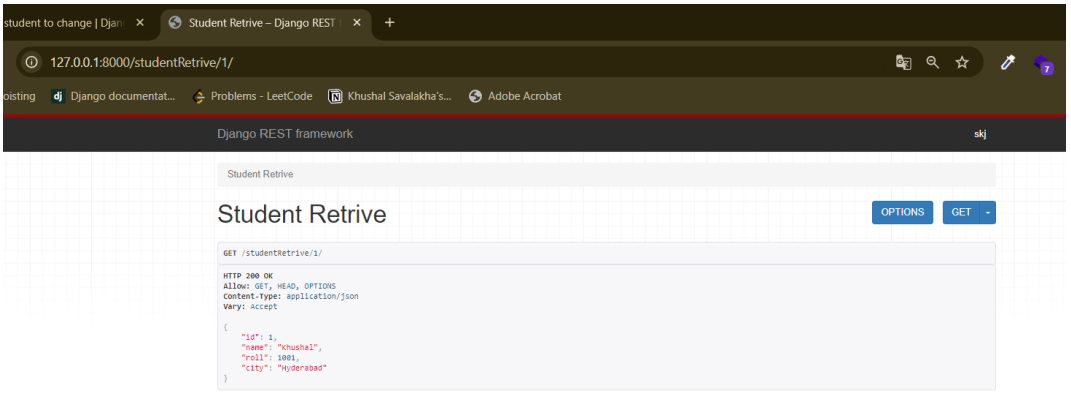

Khushal Savalakha

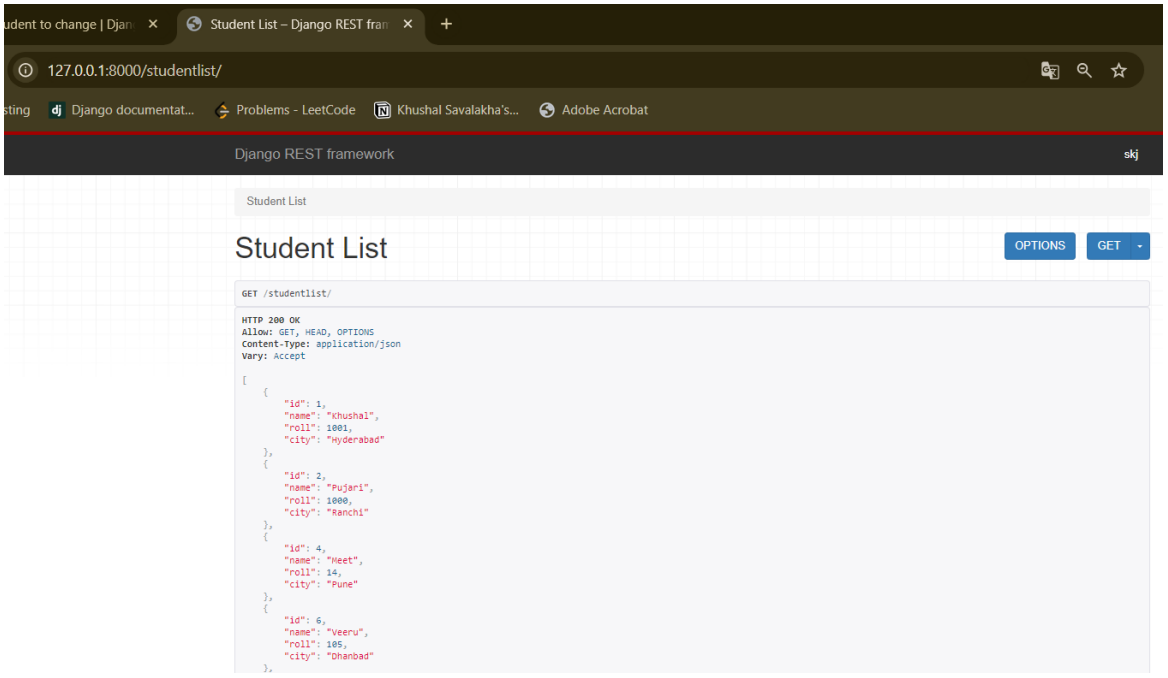



**views.py**

```
#GenericAPIView and Model Mixin
from .models import Student
from .serializers import StudentSerializer
from rest_framework.generics import  GenericAPIView
from rest_framework.mixins import ListModelMixin, CreateMod


class StudentList(GenericAPIView,ListModelMixin):
    queryset=Student.objects.all()
    serializer_class=StudentSerializer
    def get(self,request,*args,**kwargs):
        return self.list(request,*args,**kwargs)

class StudentCreate(GenericAPIView,CreateModelMixin):
    queryset=Student.objects.all()
```

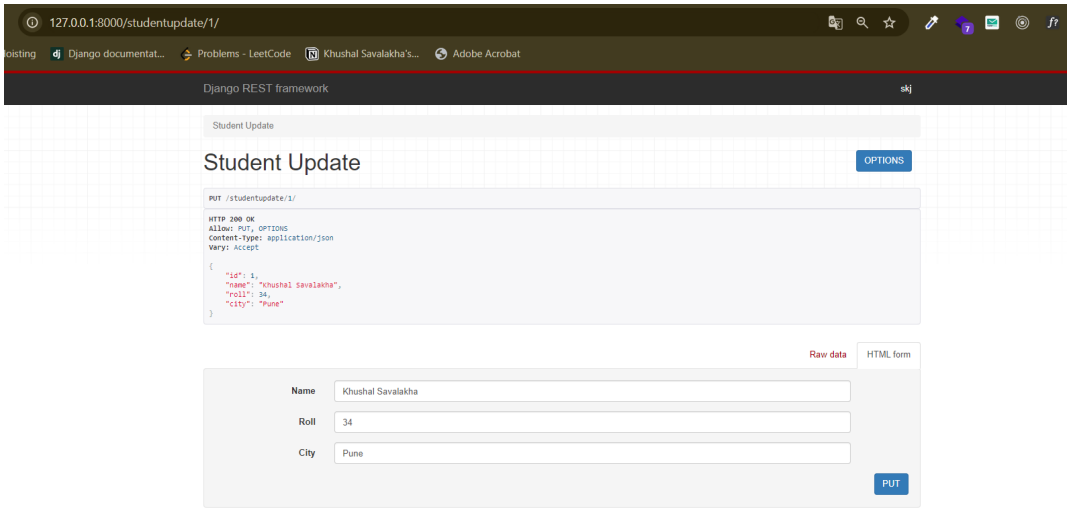
```
    serializer_class=StudentSerializer
    def post(self,request,*args,**kwargs):
        return self.create(request,*args,**kwargs)

class StudentRetrive(GenericAPIView,RetrieveModelMixin):
    queryset=Student.objects.all()
    serializer_class=StudentSerializer
    def get(self,request,*args,**kwargs):
     return self.retrieve(request,*args,**kwargs)
```

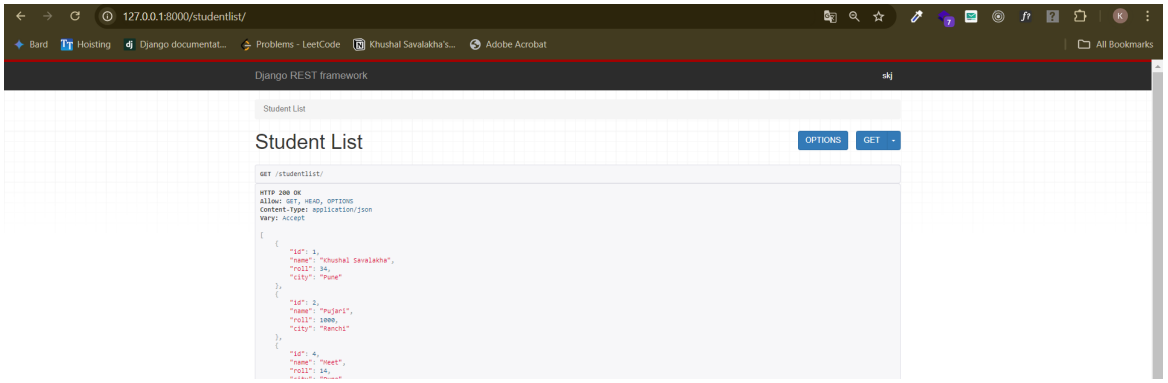




Khushal Savalakha




**urls.py**

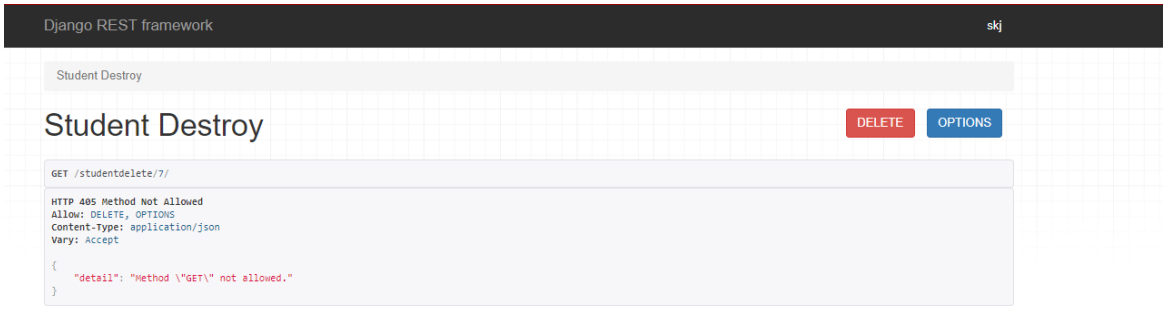
```
from django.contrib import admin
from django.urls import path
from api import views

urlpatterns = [
    path('admin/', admin.site.urls),
    path('studentlist/',views.StudentList.as_view()),
    path('studentcreate/',views.StudentCreate.as_view()),
    path('studentRetrive/<int:pk>/',views.StudentRetrive.as
    path('studentupdate/<int:pk>/',views.StudentUpdate.as_v
    path('studentdelete/<int:pk>/',views.StudentDestroy.as_
    # path('studentapi/<int:pk>',views.StudentList.as_view(
]
```

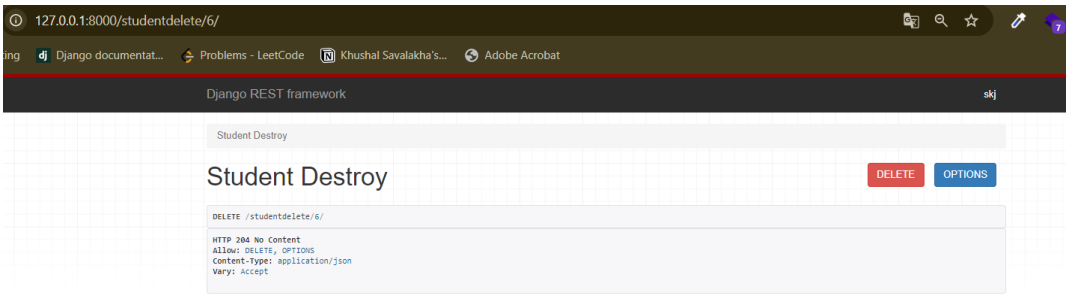
**views.py**


Khushal Savalakha

```
#GenericAPIView and Model Mixin
from .models import Student
from .serializers import StudentSerializer
from rest_framework.generics import  GenericAPIView
from rest_framework.mixins import ListModelMixin,CreateMode
from rest_framework.mixins import RetrieveModelMixin,Update


class StudentList(GenericAPIView,ListModelMixin):
    queryset=Student.objects.all()
    serializer_class=StudentSerializer
    def get(self,request,*args,**kwargs):
        return self.list(request,*args,**kwargs)

class StudentCreate(GenericAPIView,CreateModelMixin):
    queryset=Student.objects.all()
    serializer_class=StudentSerializer
    def post(self,request,*args,**kwargs):
        return self.create(request,*args,**kwargs)

class StudentRetrive(GenericAPIView,RetrieveModelMixin):
    queryset=Student.objects.all()
    serializer_class=StudentSerializer
    def get(self,request,*args,**kwargs):
     return self.retrieve(request,*args,**kwargs)

class StudentUpdate(GenericAPIView,UpdateModelMixin):
    queryset=Student.objects.all()
    serializer_class=StudentSerializer
    def put(self,request,*args,**kwargs):
     return self.update(request,*args,**kwargs)

class StudentDestroy(GenericAPIView,DestroyModelMixin):
    queryset=Student.objects.all()
    serializer_class=StudentSerializer
    def delete(self,request,*args,**kwargs):
     return self.destroy(request,*args,**kwargs)
```


Khushal Savalakha



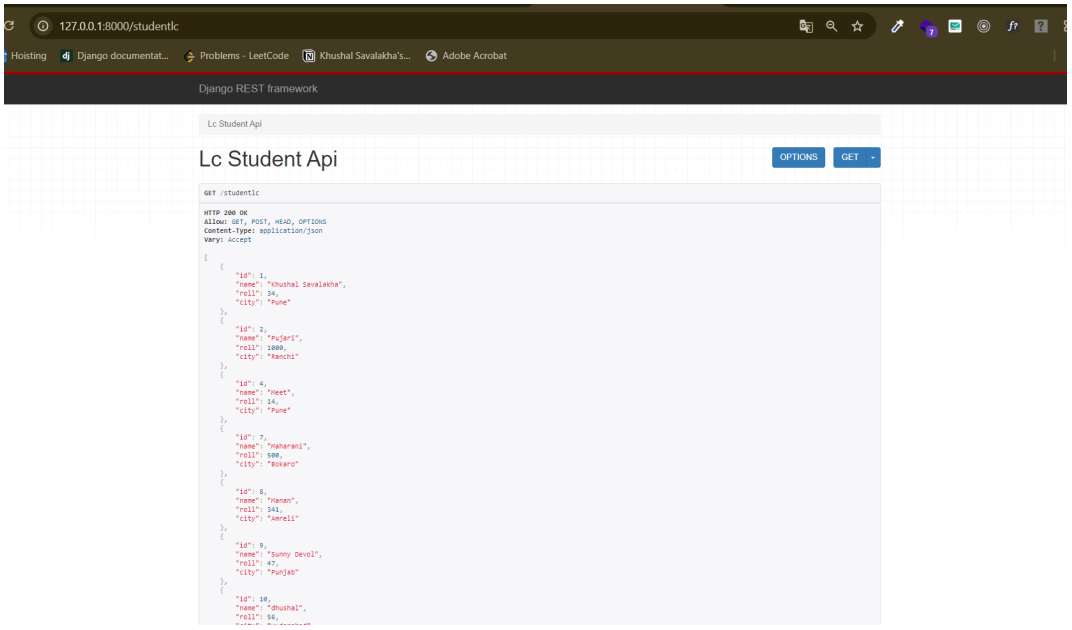




Merging getList and Post

```
#List and Create-PK Not Required
```

**views.py**

```
class LCStudentAPI(GenericAPIView,ListModelMixin,CreateMode
    queryset=Student.objects.all()
    serializer_class=StudentSerializer
    def get(self,request,*args,**kwargs):
        return self.list(request,*args,**kwargs)

    def post(self,request,*args,**kwargs):
        return self.create(request,*args,**kwargs)
```




💡 Merge Retrive,Update,Delete

```
#Retrive Update and Delete-PK required
```

**urls.py**

```
from django.contrib import admin
from django.urls import path
from api import views

urlpatterns = [
    path('admin/', admin.site.urls),
    path('studentlc/',views.LCStudentAPI.as_view()),
    path('studentrud/<int:pk>/',views.RUDStudentAPI.as_view
]
```

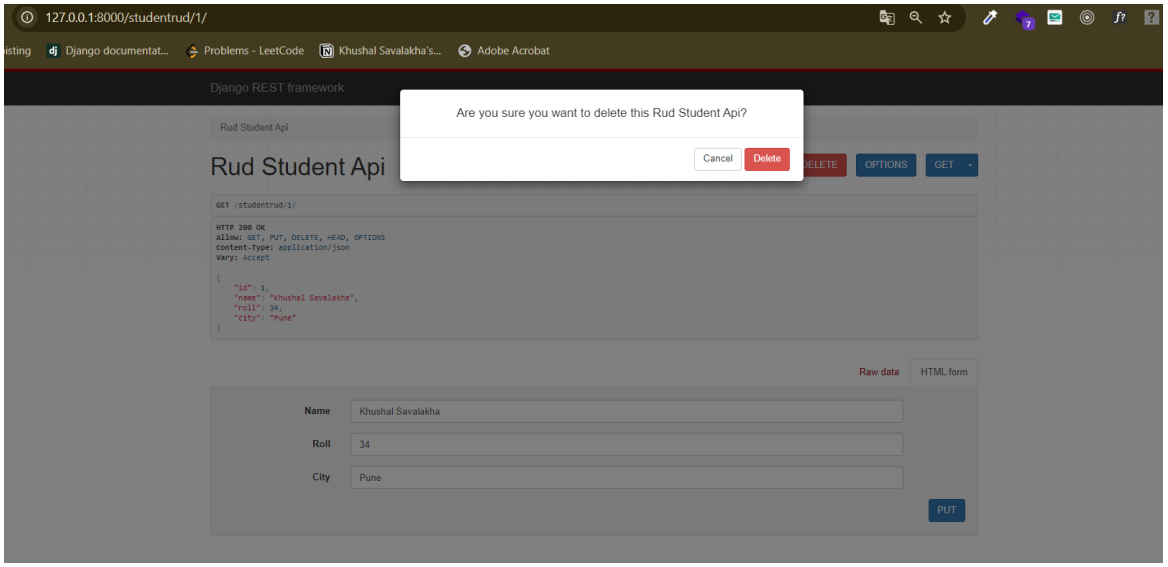
**views.py**

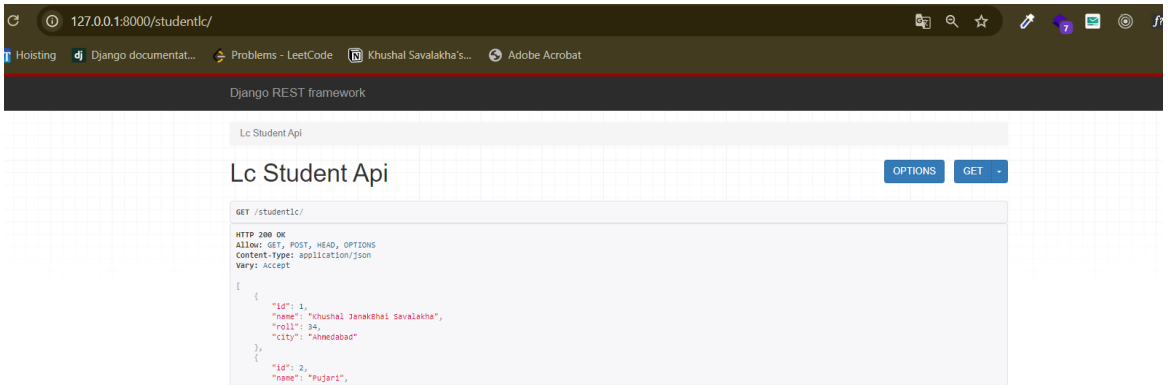
```
#GenericAPIView and Model Mixin
from .models import Student
from .serializers import StudentSerializer
```

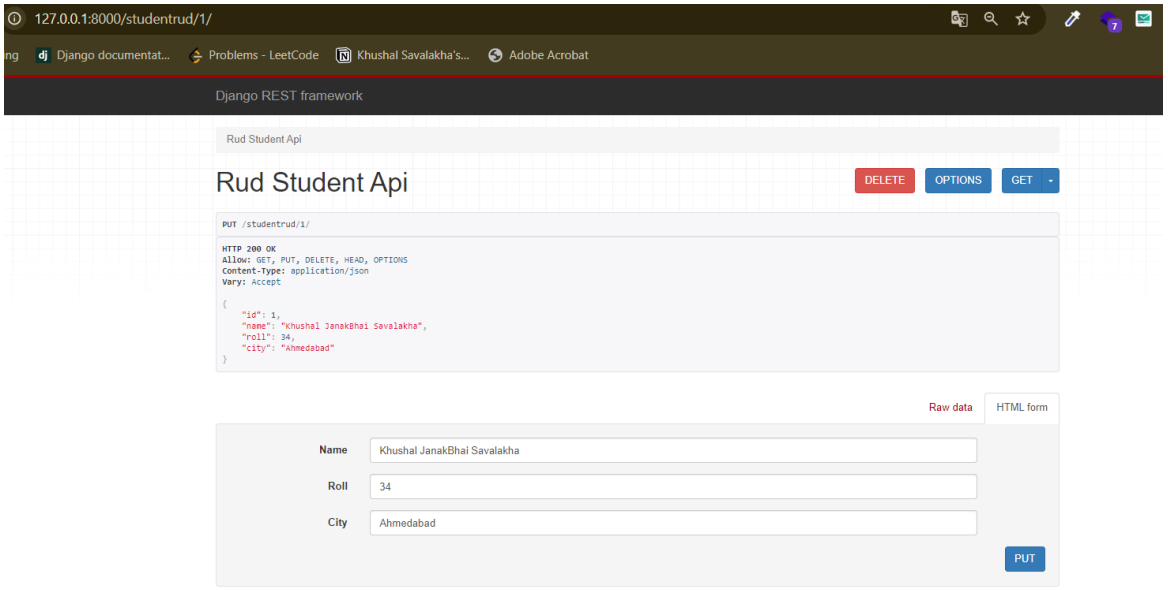

Khushal Savalakha

```
from rest_framework.generics import  GenericAPIView
from rest_framework.mixins import ListModelMixin,CreateMode
from rest_framework.mixins import RetrieveModelMixin,Update


#List and Create-PK Not Required

class LCStudentAPI(GenericAPIView,ListModelMixin,CreateMode
    queryset=Student.objects.all()
    serializer_class=StudentSerializer
    def get(self,request,*args,**kwargs):
        return self.list(request,*args,**kwargs)

    def post(self,request,*args,**kwargs):
        return self.create(request,*args,**kwargs)

#Retrive Update and Delete-PK required
class RUDStudentAPI(GenericAPIView,RetrieveModelMixin,Updat
    queryset=Student.objects.all()
    serializer_class=StudentSerializer
    def get(self,request,*args,**kwargs):
     return self.retrieve(request,*args,**kwargs)

    def put(self,request,*args,**kwargs):
     return self.update(request,*args,**kwargs)

    def delete(self,request,*args,**kwargs):
     return self.destroy(request,*args,**kwargs)
```


Khushal Savalakha




# ▼ Concrete view class

# Concrete View Class

The following classes are the concrete generic views.

If you're using generic views this is normally the level you'll be working at unless you need heavily customized behavior.

The view classes can be imported from rest_framework.generics.

- ListAPIView
- CreateAPIView
- RetrieveAPIView
- UpdateAPIView
- DestroyAPIView
- ListCreateAPIView
- RetrieveUpdateAPIView
- RetrieveDestroyAPIView
- RetrieveUpdateDestroyAPIView

# ListAPIView

It is used for read-only endpoints to represent a collection of model instances. It provides a get method handler.

Extends: GenericAPIView, ListModelMixin

```
from rest_framework.generics import ListAPIView
class StudentList(ListAPIView):
    queryset = Student.objects.all()
    serializer_class = StudentSerializer
```

# CreateAPIView

It is used for create-only endpoints. It provides a post method handler.

Extends: GenericAPIView, CreateModelMixin

```
from rest_framework.generics import CreateAPIView
class StudentCreate(CreateAPIView):
    queryset = Student.objects.all()
    serializer_class = StudentSerializer
```


Khushal Savalakha

# RetrieveAPIView

It is used for read-only endpoints to represent a single model instance. It provides a get method handler.

Extends: GenericAPIView, RetrieveModelMixin

```
from rest_framework.generics import RetrieveAPIView
class StudentRetrieve(RetrieveAPIView):
    queryset = Student.objects.all()
    serializer_class = StudentSerializer
```

# UpdateAPIView

It is used for update-only endpoints for a single model instance. It provides put and patch method handlers.

Extends: GenericAPIView, UpdateModelMixin

```
from rest_framework.generics import UpdateAPIView
class StudentUpdate(UpdateAPIView):
    queryset = Student.objects.all()
    serializer_class = StudentSerializer
```

# DestroyAPIView

It is used for delete-only endpoints for a single model instance. It provides a delete method handler.

Extends: GenericAPIView, DestroyModelMixin

```
from rest_framework.generics import DestroyAPIView
class StudentDestroy(DestroyAPIView):
    queryset = Student.objects.all()
    serializer_class = StudentSerializer
```


Khushal Savalakha

# ListCreateAPIView

It is used for read-write endpoints to represent a collection of model instances. It provides get and post method handlers.

Extends: GenericAPIView, ListModelMixin, CreateModelMixin

```
from rest_framework.generics import ListCreateAPIView
class StudentListCreate(ListCreateAPIView):
    queryset = Student.objects.all()
    serializer_class = StudentSerializer
```

# RetrieveUpdateAPIView

It is used for read or update endpoints to represent a single model instance. It provides get, put and patch method handlers.

Extends: GenericAPIView, RetrieveModelMixin, UpdateModelMixin

```
from rest_framework.generics import RetrieveUpdateAPIView
class StudentRetrieveUpdate(RetrieveUpdateAPIView):
    queryset = Student.objects.all()
    serializer_class = StudentSerializer
```

# RetrieveDestroyAPIView

It is used for read or delete endpoints to represent a single model instance. It provides get and delete method handlers.

Extends: GenericAPIView, RetrieveModelMixin, DestroyModelMixin

```
from rest_framework.generics import RetrieveDestroyAPIView
class StudentRetrieveDestroy(RetrieveDestroyAPIView):
    queryset = Student.objects.all()
    serializer_class = StudentSerializer
```


Khushal Savalakha

# RetrieveUpdateDestroyAPIView

It is used for read-write-delete endpoints to represent a single model instance. It provides get, put, patch and delete method handlers.

Extends: GenericAPIView, RetrieveModelMixin, UpdateModelMixin, DestroyModelMixin

```
from rest_framework.generics import RetrieveUpdateDestroyAPIView
class StudentRetrieveUpdateDestroy(RetrieveUpdateDestroyAPIView):
    queryset = Student.objects.all()
    serializer_class = StudentSerializer
```

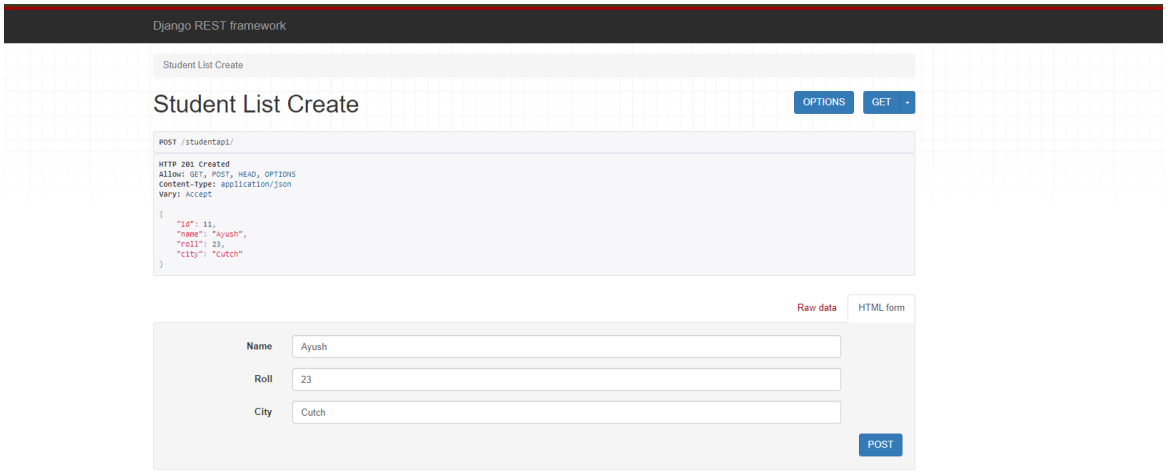
**urls.py**

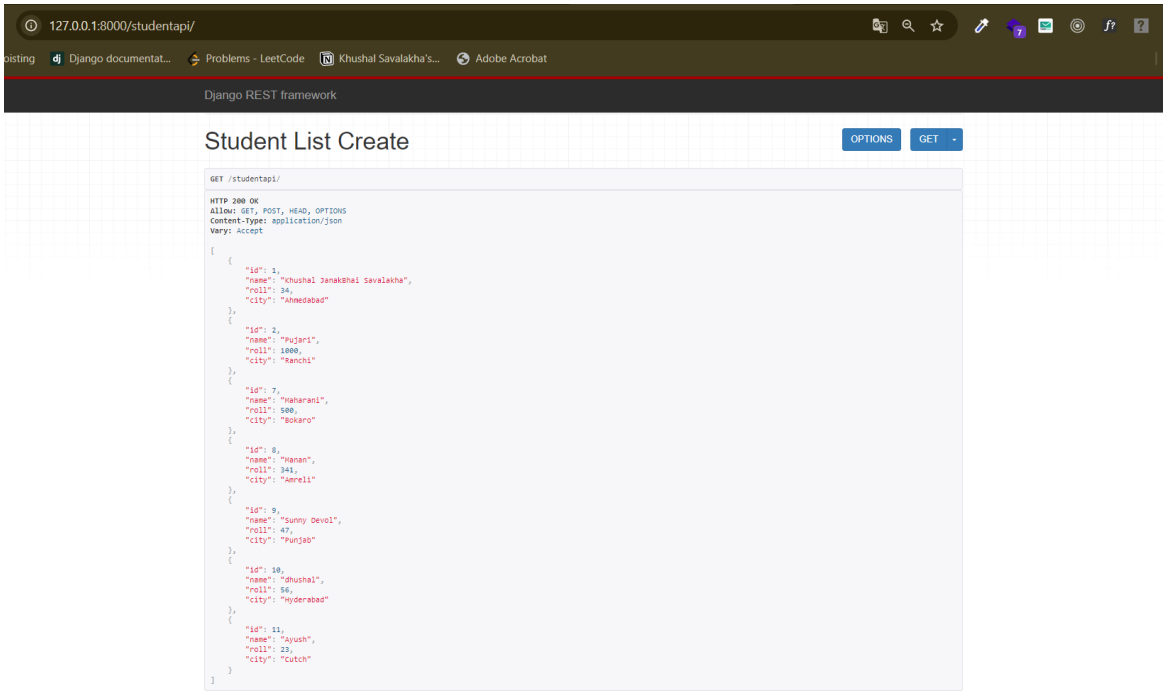
```
from django.contrib import admin
from django.urls import path
from api import views

urlpatterns = [
    path('admin/', admin.site.urls),
    path('studentapi/', views.StudentListCreate.as_view()),
    path('studentapi/<int:pk>/', views.StudentRetrieveUpdat
]
```

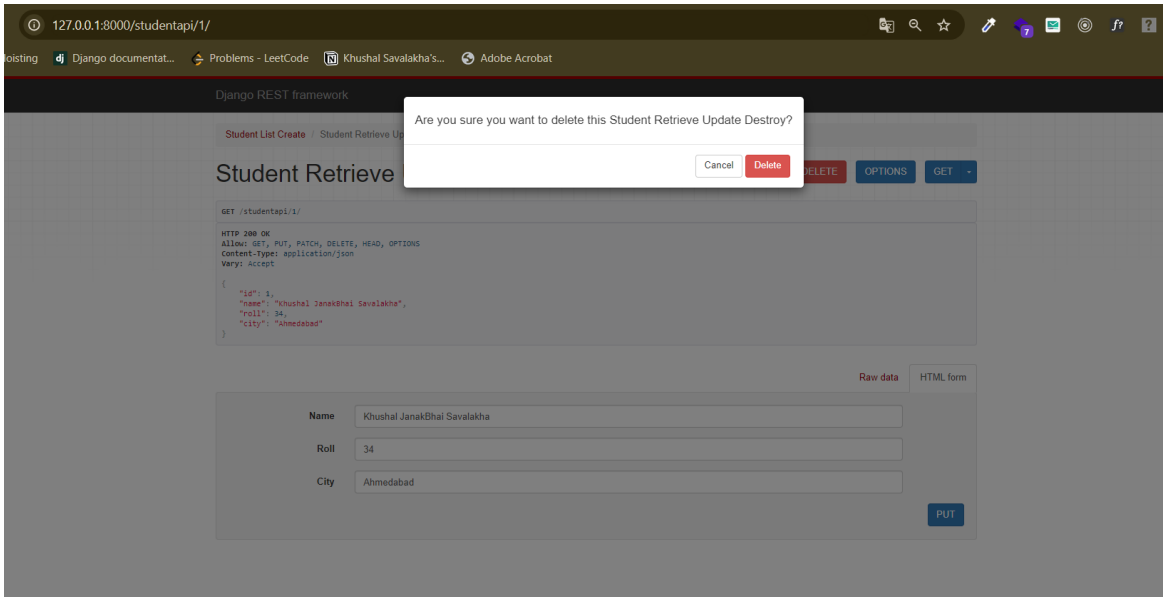
**views.py**

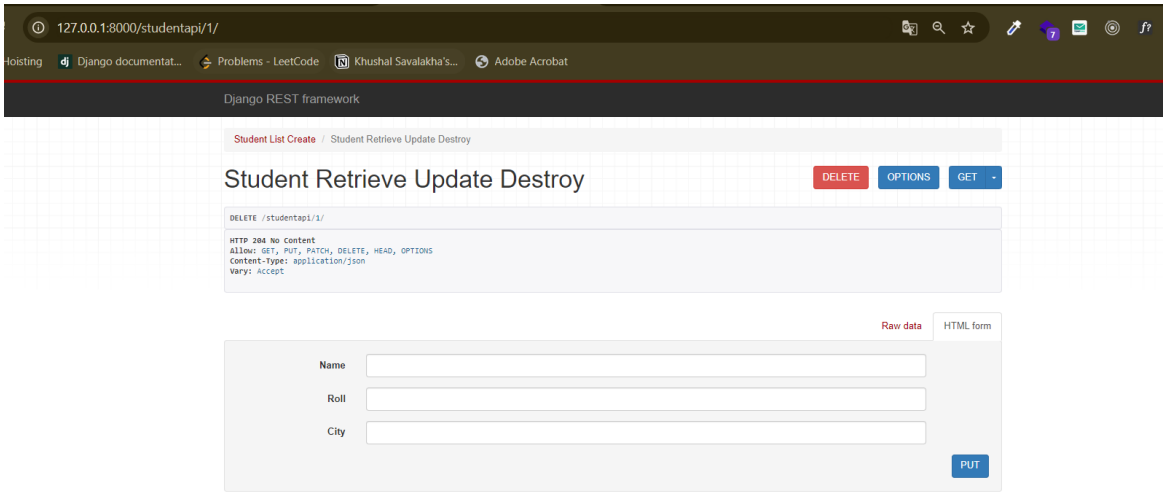
```
from .models import Student
from .serializers import StudentSerializer
from rest_framework.generics import ListCreateAPIView, Retr

class StudentListCreate(ListCreateAPIView):
    queryset = Student.objects.all()
    serializer_class = StudentSerializer  # Corrected typo

class StudentRetrieveUpdateDestroy(RetrieveUpdateDestroyAPI
    queryset = Student.objects.all()
    serializer_class = StudentSerializer  # Corrected typo
```

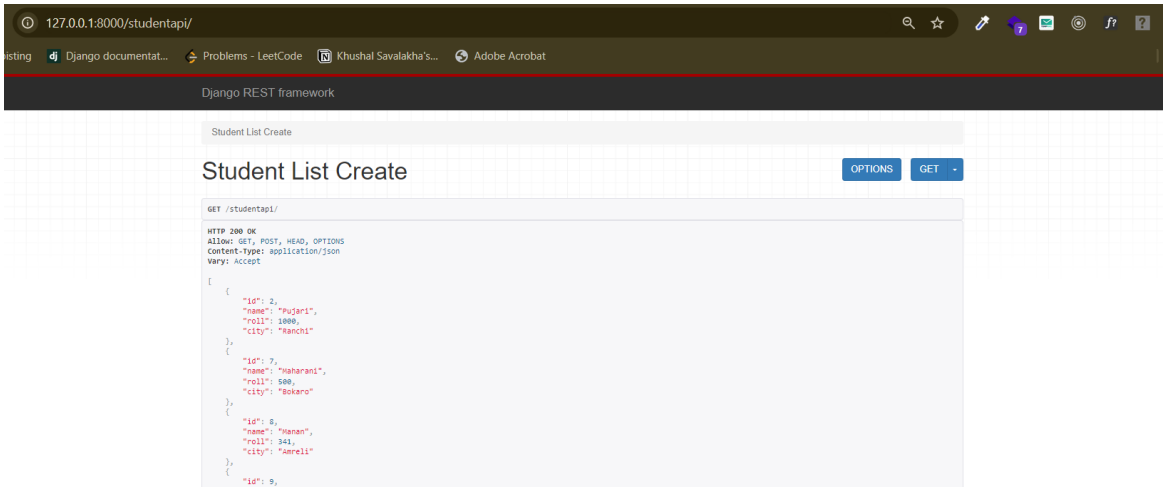

Khushal Savalakha


Django REST framework
Student List Create
Student List Create
OPTIONS
GET
POST /studentapi/
HTTP 201 Created
Allow: GET, POST, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept
{
"id": 11,
"name": "Ayush",
"roll": 23,
"city": "Cutch"
}
Raw data
HTML form
Name
Ayush
Roll
23
City
Cutch
POST


127.0.0.1:8000/studentapi/
Django documentat...
Problems - LeetCode
Khushal Savalakha's...
Adobe Acrobat
Django REST framework
Student List Create
OPTIONS
GET
GET /studentapi/
HTTP 200 OK
Allow: GET, POST, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept
[
{
"id": 1,
"name": "Khushal Janakbhai Savalakha",
"roll": 34,
"city": "Ahmedabad"
},
{
"id": 2,
"name": "Pujari",
"roll": 1000,
"city": "Ranchi"
},
{
"id": 7,
"name": "Maharani",
"roll": 500,
"city": "Bokaro"
},
{
"id": 8,
"name": "Hanan",
"roll": 341,
"city": "Amreli"
},
{
"id": 9,
"name": "Sunny Devol",
"roll": 47,
"city": "Punjab"
},
{
"id": 10,
"name": "dhushal",
"roll": 56,
"city": "Hyderabad"
},
{
"id": 11,
"name": "Ayush",
"roll": 23,
"city": "Cutch"
}
]


127.0.0.1:8000/studentapi/1/
Hoisting
Django documentat...
Problems - LeetCode
Khushal Savalakha's...
Adobe Acrobat
Django REST framework
Are you sure you want to delete this Student Retrieve Update Destroy?
Cancel
Delete
Student List Create / Student Retrieve Up
Student Retrieve
DELETE
OPTIONS
GET
GET /studentapi/1/
HTTP 200 OK
Allow: GET, PUT, PATCH, DELETE, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept
{
"id": 1,
"name": "Khushal JanakBhai Savalakha",
"roll": 34,
"city": "Ahmedabad"
}
Raw data
HTML form
Name
Khushal JanakBhai Savalakha
Roll
34
City
Ahmedabad
PUT


127.0.0.1:8000/studentapi/1/
Hoisting
Django documentat...
Problems - LeetCode
Khushal Savalakha's...
Adobe Acrobat
Django REST framework
Student List Create / Student Retrieve Update Destroy
Student Retrieve Update Destroy
DELETE
OPTIONS
GET
DELETE /studentapi/1/
HTTP 204 No Content
Allow: GET, PUT, PATCH, DELETE, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept
Raw data
HTML form
Name
Roll
City
PUT


127.0.0.1:8000/studentapi/
Hoisting
Django documentat...
Problems - LeetCode
Khushal Savalakha's...
Adobe Acrobat
Django REST framework
Student List Create
Student List Create
OPTIONS
GET
GET /studentapi/
HTTP 200 OK
Allow: GET, POST, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept
[
{
"id": 2,
"name": "Pujari",
"roll": 1000,
"city": "Ranchi"
},
{
"id": 7,
"name": "Maharani",
"roll": 500,
"city": "Bokaro"
},
{
"id": 8,
"name": "Manan",
"roll": 341,
"city": "Amreli"
},
{
"id": 9,

# ▼ corsheaders and csrf_excempt

> 💡 what is the role of coreheaders ?

**CORS (Cross-Origin Resource Sharing)** headers, often managed by middleware like `django-cors-headers`, play a significant role in handling **cross-origin requests** in Django applications, especially when using APIs.

## What are CORS Headers?

By default, web browsers block requests from one domain to another domain for security reasons. For example, if your frontend (say, a React app) is running on `http://localhost:3000` and your Django backend is running on `http://localhost:8000`, browsers will block such requests unless your Django server explicitly allows it by adding **CORS headers** to the response.

CORS headers define which origins (domains) are allowed to interact with your server.

## How do CORS headers work?

If your Django API or web application will be accessed by a frontend running on a different domain, you need to allow cross-origin requests by sending the appropriate **CORS headers** in responses. These headers indicate to the browser that it can trust requests from other origins.

## Using `django-cors-headers` in Django:

You can manage CORS headers in Django using the popular package `django-cors-headers`. It adds the appropriate CORS headers to your responses to allow cross-origin requests.

## Steps to Enable CORS in Django:

1. **Install `django-cors-headers`:**

   Run the following command to install the package:

   ```
   pip install django-cors-headers
   ```

2. **Add it to `INSTALLED_APPS`:**


Khushal Savalakha

Add `corsheaders` to your `INSTALLED_APPS` in `settings.py`:

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

3. **Add the Middleware:**

   Add `corsheaders.middleware.CorsMiddleware` to the top of your `MIDDLEWARE` settings in `settings.py`:

```
MIDDLEWARE = [
    'corsheaders.middleware.CorsMiddleware',  # Add t
his at the top
    'django.middleware.common.CommonMiddleware',
    ...
]
```

4. **Configure CORS in `settings.py`:**

   You can allow specific origins or all origins, depending on your use case.

   - To allow **all** origins:

```
CORS_ALLOW_ALL_ORIGINS = True
```

   - To allow **specific** origins:

```
CORS_ALLOWED_ORIGINS = [
    "<http://localhost:3000>",
    "<https://yourfrontend.com>",
]
```

   - If you're working with cookies and authentication across domains, you might also need to allow credentials:

```
CORS_ALLOW_CREDENTIALS = True
```


Khushal Savalakha

- You can also specify which **HTTP methods** are allowed in cross-origin requests:

```
CORS_ALLOW_METHODS = [
    "GET",
    "POST",
    "PUT",
    "DELETE",
    "PATCH",
    "OPTIONS",
]
```

## CORS Headers in Action:

When a browser makes a cross-origin request (e.g., a `fetch` request from the frontend running on a different domain), the Django backend will return a response with appropriate **CORS headers** (such as `Access-Control-Allow-Origin`). These headers let the browser know that the server permits requests from that specific domain.

## Example of a CORS request:

If you allow `http://localhost:3000` to make requests to your Django API at `http://localhost:8000`, the response will include a header like this:

```
Access-Control-Allow-Origin: <http://localhost:3000>
```

## Role in APIs and CSRF tokens:

- **CORS** is crucial when building APIs, especially in single-page applications (SPAs) where the frontend and backend run on different domains.
- **CSRF tokens and CORS**: If you're using CSRF tokens for security, and the frontend and backend are on different domains, it's essential to correctly configure both CORS headers and CSRF token settings. CORS does not replace CSRF protection, but it helps browsers determine whether they should even allow the request to be sent.

## Example Use Case in Django:

Imagine you're building a React app and a Django REST API. The React app sends a `POST` request to the Django API to create a new record:


Khushal Savalakha

```
fetch('<http://localhost:8000/api/create/>', {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json',
  },
  body: JSON.stringify(data),
})
```

Without the proper CORS configuration, the browser will block this request because it is cross-origin (React app on `localhost:3000`, Django API on `localhost:8000`). By configuring CORS headers in Django, you ensure that the browser allows this request to go through.

## In Conclusion:

CORS headers manage which external domains can interact with your Django backend. Using the `method_decorator` like `@csrf_exempt` may disable CSRF protection for certain views, but if you're using APIs with cross-origin requests, you also need to handle **CORS headers** to ensure the requests are not blocked by the browser due to security policies.

💡 how to use csrf excempt and decorators?

💡 method_decorator is it used for csrf token checking process not to do ?

Yes, when `method_decorator` is used with `@csrf_exempt`, it effectively **disables the CSRF token checking process** for the specific method or entire class-based view (CBV).

In Django, CSRF (Cross-Site Request Forgery) protection is enabled by default for views that handle `POST`, `PUT`, `PATCH`, or `DELETE` requests to prevent malicious requests. Using `@csrf_exempt` tells Django **not to check for a CSRF token** for that specific view or method, meaning CSRF protection is skipped.

## How it works:

- **Without `csrf_exempt`:** Django expects a valid CSRF token in `POST` requests to ensure the request is coming from a trusted source.


Khushal Savalakha

- **With `csrf_exempt`:** Django will **not validate** the CSRF token for the decorated method, meaning the view will accept `POST` requests without checking the CSRF token.

### Example:

If you want to disable CSRF protection for the `post` method in a class-based view, you would do something like this:

```
from django.views.decorators.csrf import csrf_exempt
from django.utils.decorators import method_decorator
from django.http import JsonResponse
from django.views import View

class MyView(View):
    @method_decorator(csrf_exempt)  # Disables CSRF chec
k for the 'post' method
    def post(self, request, *args, **kwargs):
        return JsonResponse({'message': 'CSRF check is s
kipped for this request'})
```

### What does this do?

- The `csrf_exempt` decorator tells Django not to check for a CSRF token in `POST` requests handled by this method (`post` in this case).
- If someone makes a `POST` request to this view, Django will not throw a "403 Forbidden" error due to missing or invalid CSRF tokens.

### When to use it?

- **API endpoints**: Sometimes, when you're building APIs (especially if you're using something like Django REST Framework), CSRF tokens may not be necessary, and you might want to exempt certain views from CSRF protection.
- **Testing**: You may want to temporarily disable CSRF protection while testing or during development.

### Example of Applying to the Entire Class:


Khushal Savalakha

```
from django.utils.decorators import method_decorator
from django.views.decorators.csrf import csrf_exempt
from django.views import View

@method_decorator(csrf_exempt, name='dispatch')
class MyView(View):
    def post(self, request, *args, **kwargs):
        return JsonResponse({'message': 'Post request wi
thout CSRF token check'})

    def get(self, request, *args, **kwargs):
        return JsonResponse({'message': 'Get request'})
```

In this case, CSRF protection is disabled for **all** methods (`post`, `get`, etc.) in the class.

## Caution:

- Disabling CSRF protection can expose your app to security vulnerabilities, especially in web forms where user data is submitted. Be careful when using `@csrf_exempt`, and only use it when you are sure that CSRF protection is not needed (e.g., for certain APIs).

💡 @api_view(['POST'])
def insert_booking_data(request):
# Insert new booking data
serializer = BookingDataSerializer(data=request.data)
if serializer.is_valid():
serializer.save()
return Response(serializer.data, status=status.HTTP_201_CREATED)
return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)

this is my views.py function how to use csrf excempt and decorators ?

To make your `insert_booking_data` function exempt from CSRF protection, you can use the `@csrf_exempt` decorator in combination with the `@api_view` decorator from Django REST framework. However, **Django REST framework**'s `@api_view` is a function-based


Khushal Savalakha

decorator that only works on views expecting JSON and does not perform CSRF checks by default. So, in most cases, you don't need to explicitly exempt it from CSRF.

That said, if you're still facing CSRF issues (for example, in scenarios where you're sending data via forms rather than JSON), you can use the `@csrf_exempt` decorator explicitly.

### To apply `@csrf_exempt`:

You can use the `method_decorator` to apply `@csrf_exempt` on the `@api_view` function.

Here's how you can modify your view:

### Solution 1: Use `method_decorator` with `@csrf_exempt`

```
from django.utils.decorators import method_decorator
from django.views.decorators.csrf import csrf_exempt
from rest_framework.decorators import api_view
from rest_framework.response import Response
from rest_framework import status

# Exempt CSRF for POST request
@method_decorator(csrf_exempt, name='dispatch')
@api_view(['POST'])
def insert_booking_data(request):
    # Insert new booking data
    serializer = BookingDataSerializer(data=request.dat
a)
    if serializer.is_valid():
        serializer.save()
        return Response(serializer.data, status=status.H
TTP_201_CREATED)
    return Response(serializer.errors, status=status.HTT
P_400_BAD_REQUEST)
```

### Solution 2: Use `@csrf_exempt` directly above `@api_view`

If you prefer to use the `@csrf_exempt` directly, you can do so as well.

```
from django.views.decorators.csrf import csrf_exempt
from rest_framework.decorators import api_view
```


Khushal Savalakha

```
from rest_framework.response import Response
from rest_framework import status

# Exempt CSRF for POST request
@csrf_exempt
@api_view(['POST'])
def insert_booking_data(request):
    # Insert new booking data
    serializer = BookingDataSerializer(data=request.dat
a)
    if serializer.is_valid():
        serializer.save()
        return Response(serializer.data, status=status.H
TTP_201_CREATED)
    return Response(serializer.errors, status=status.HTT
P_400_BAD_REQUEST)
```

## Important Notes:

1. **Security Consideration**: Be cautious when using `@csrf_exempt`. CSRF protection is a critical part of web security. Only disable it if you're sure that it won't expose your application to vulnerabilities. For example, if you are using **token-based authentication** (like JWT or OAuth), CSRF might not be necessary.
2. **Token Authentication**: If you're using Django REST framework with **JWT authentication** or **OAuth**, CSRF might not be relevant. In such cases, you can keep your views without CSRF checks.
3. **API Views and CSRF**: Django REST framework's `@api_view` is commonly used for API endpoints that accept JSON requests (which are not typically prone to CSRF attacks). So often, CSRF checks are not needed when using `@api_view`.

If you're using AJAX or front-end libraries (React, Vue, etc.), and encountering CSRF issues, you might also want to look into **sending the CSRF token** with your AJAX requests instead of disabling CSRF altogether.


Khushal Savalakha

Khushal Savalakha