

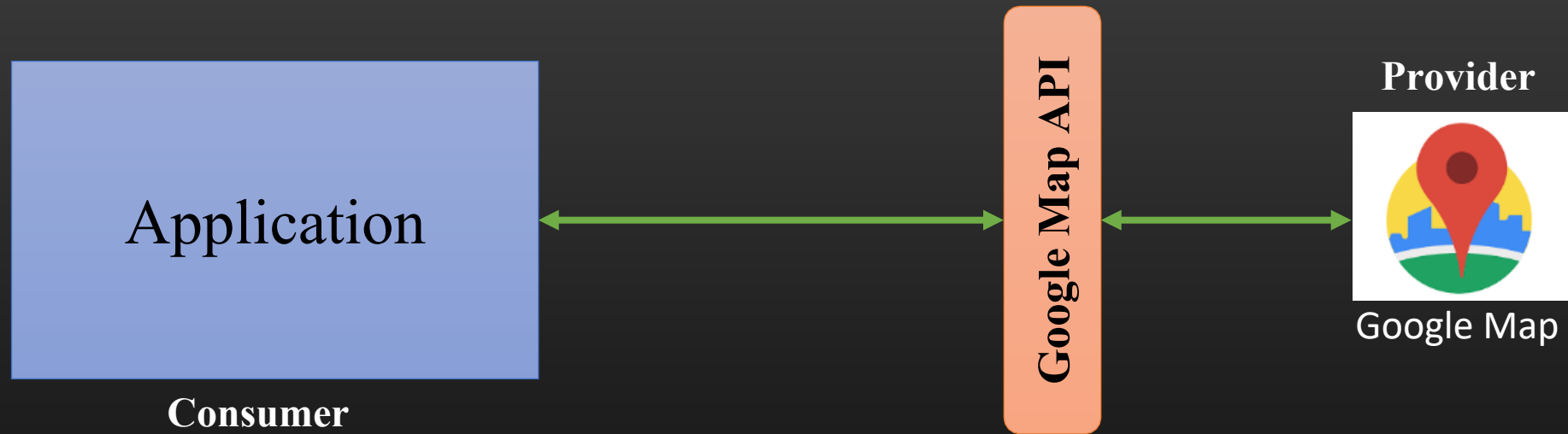
Application Programming Interface (API)

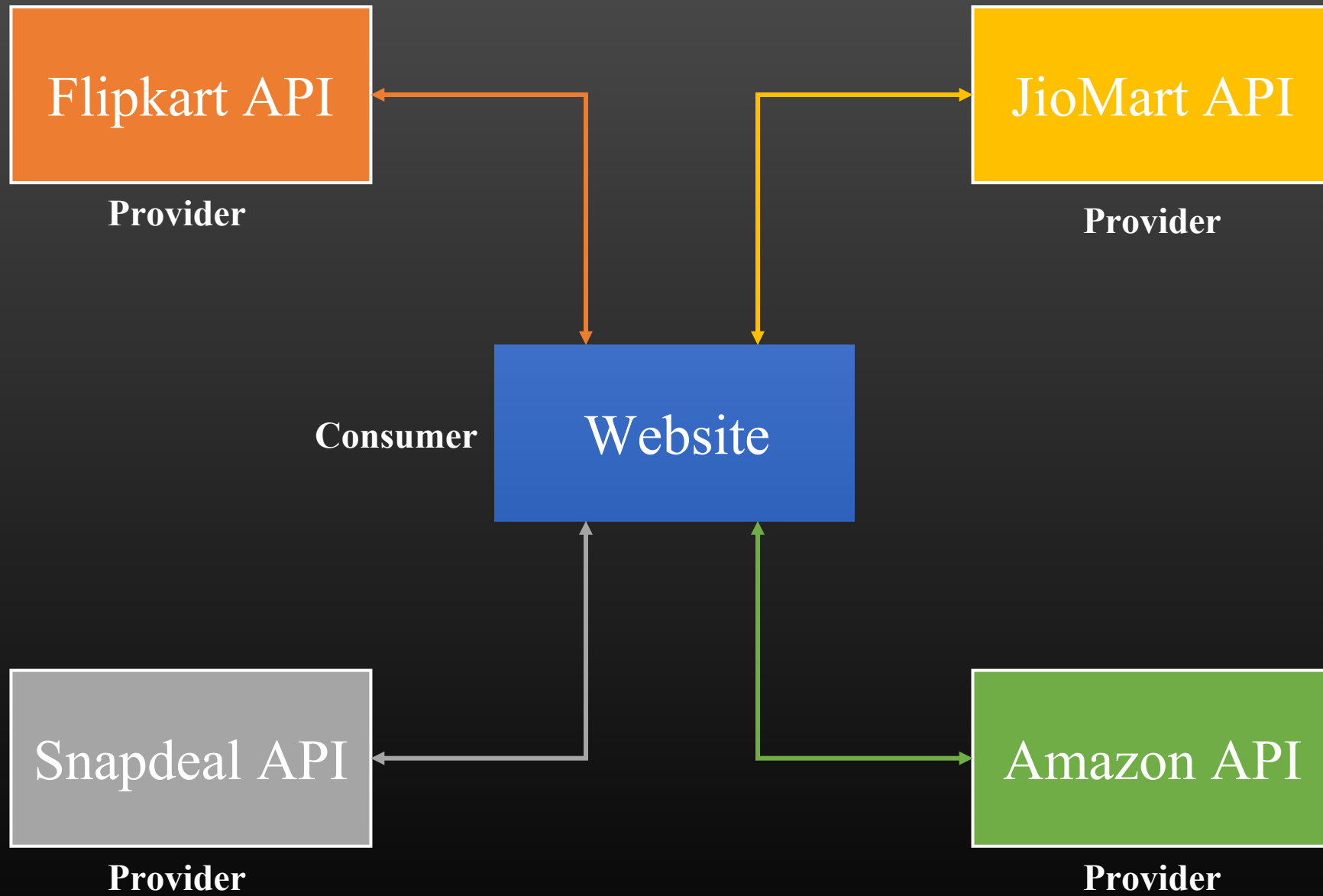
An API is a software intermediary that allows two or more applications to talk to each other.



API Type in terms of Release Policies:-

- Private – It can be used within the organization.
- Partner – It can be used within Business Partners.
- Public – It can be used any third party Developers.





Add/Update Student

Name:

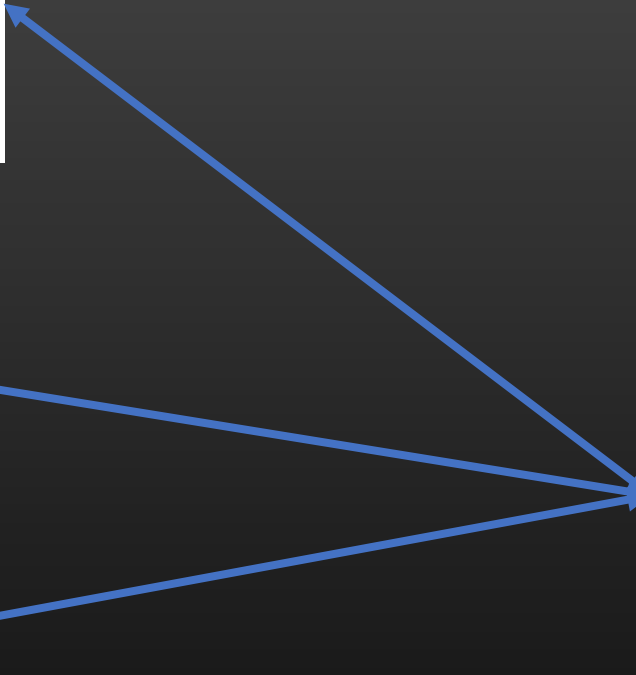
Email:

Password:

Save

Show Student Information

ID	Name	Email	Password	Action
65	Raj	raj@gmail.com	sonam1234	<div>Edit</div> <div>Delete</div>
66	Rahul	rahul@gmail.com	rahul1234	<div>Edit</div> <div>Delete</div>



Add/Update Student

Name:

Email:

Password:

Save

Show Student Information

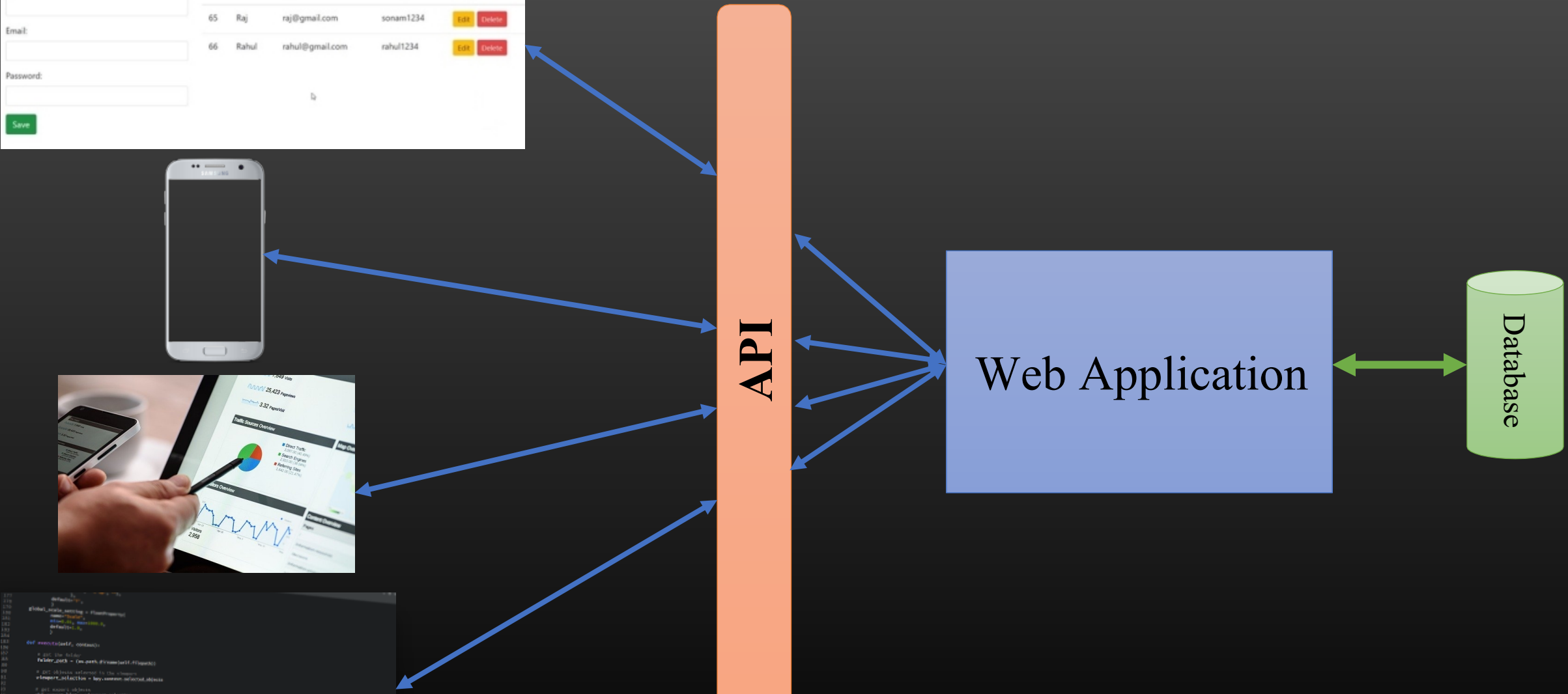
ID	Name	Email	Password	Action
65	Raj	raj@gmail.com	sonam1234	<div>EditDelete</div>
66	Rahul	rahul@gmail.com	rahul1234	<div>EditDelete</div>



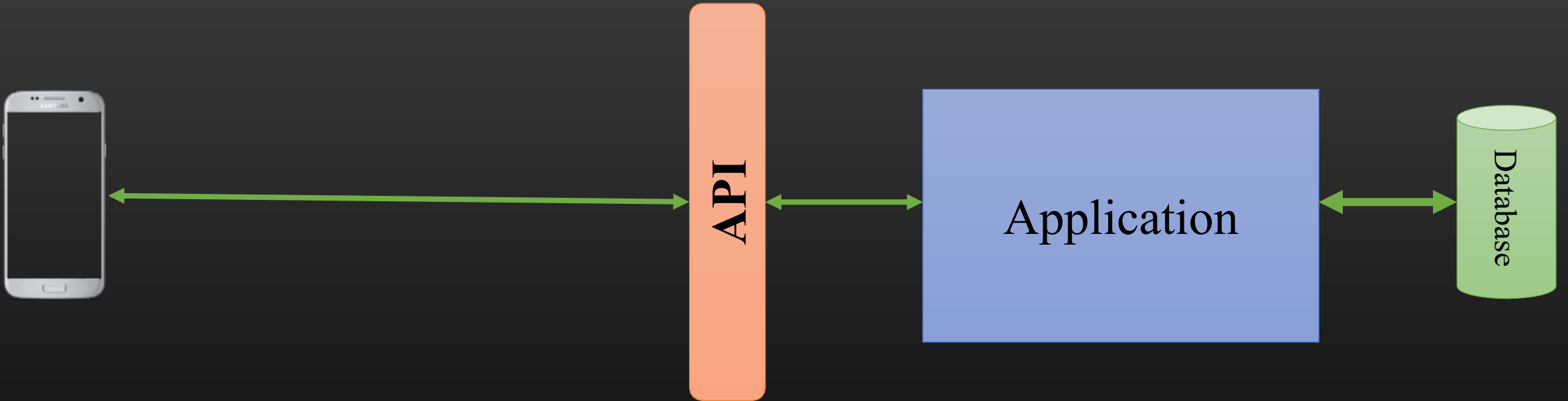
```
210 def __init__(self):
211     self._name = None
212     self._email = None
213     self._password = None
214
215 def __str__(self):
216     return f'Name: {self._name}, Email: {self._email}, Password: {self._password}'
217
218 def __repr__(self):
219     return f'<__main__.Student object at {hex(id(self))}>'
220
221 def __setattr__(self, name, value):
222     if name in ['_name', '_email', '_password']:
223         self.__dict__[name] = value
224     else:
225         raise AttributeError(f'Attribute {name} not found')
226
227 def __getattr__(self, name):
228     if name in ['_name', '_email', '_password']:
229         return self.__dict__[name]
230     else:
231         raise AttributeError(f'Attribute {name} not found')
232
233 def __del__(self):
234     print(f'Deleting {self._name}')
235
236 def __main__(self):
237     # Create a new student
238     student = Student('Raj', 'raj@gmail.com', 'sonam1234')
239     # Add student to the database
240     student.save()
241     # Print student information
242     print(student)
```

API

Web Application



How use API



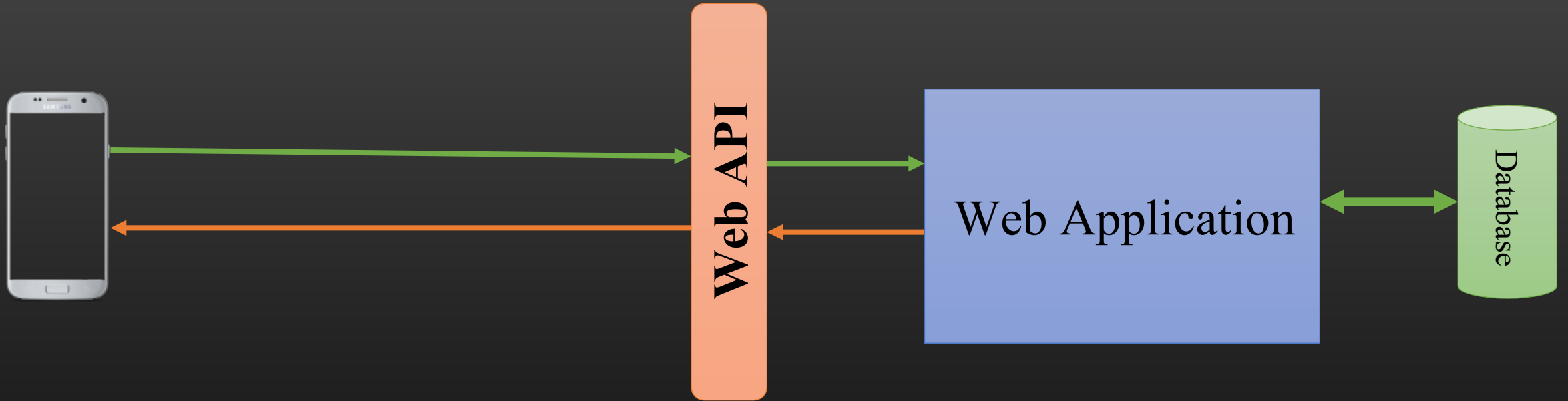
API Key/Token: 23423jkhjhgh32434hjgjh4343

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.

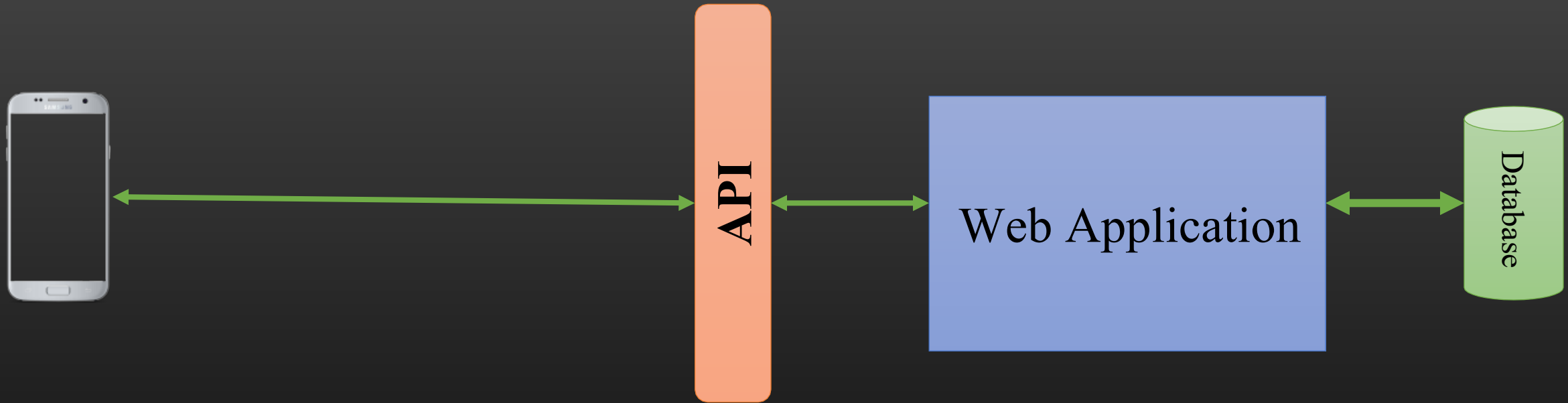
How Web API Works



- 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

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.

Add/Update Student

Name:

Email:

Password:

Save

Show Student Information

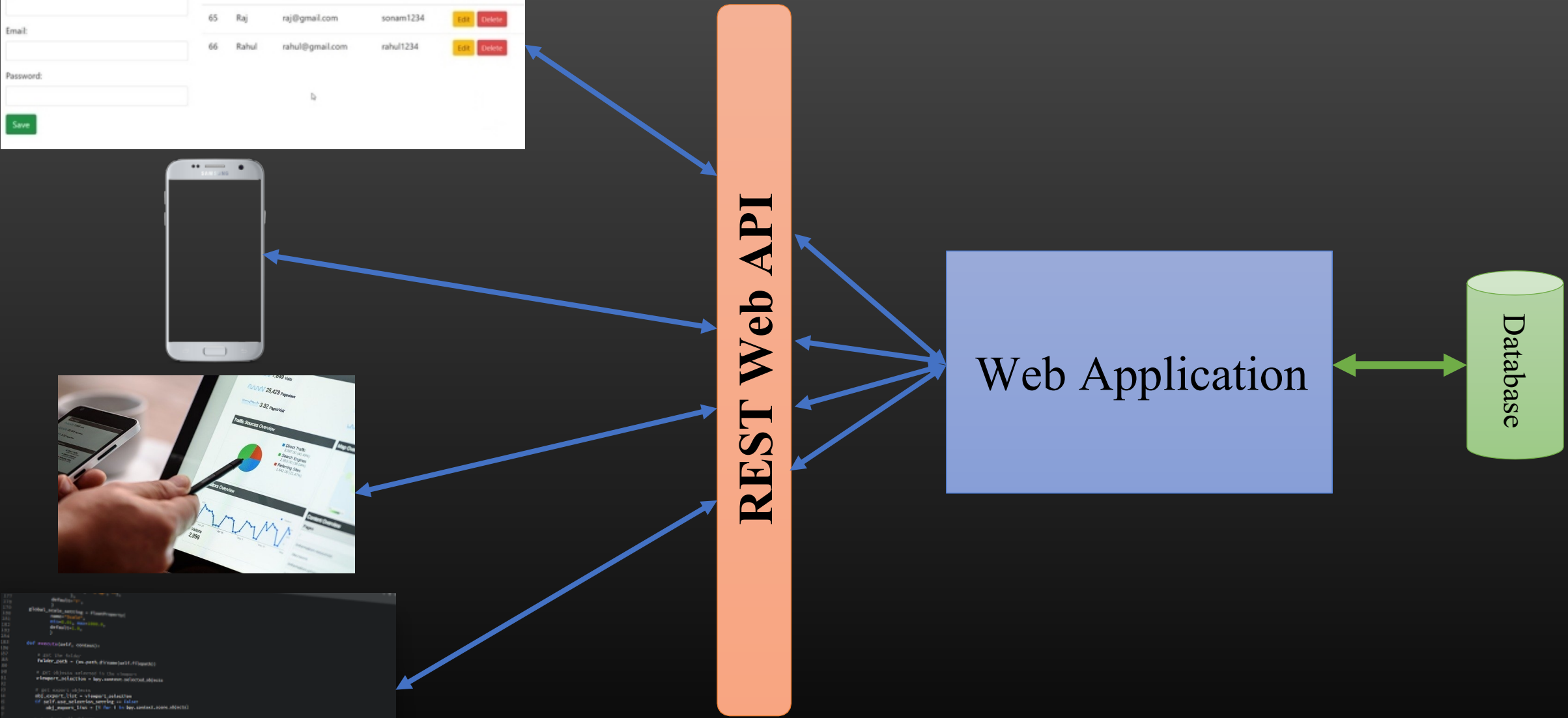
ID	Name	Email	Password	Action
65	Raj	raj@gmail.com	sonam1234	<div>EditDelete</div>
66	Rahul	rahul@gmail.com	rahul1234	<div>EditDelete</div>



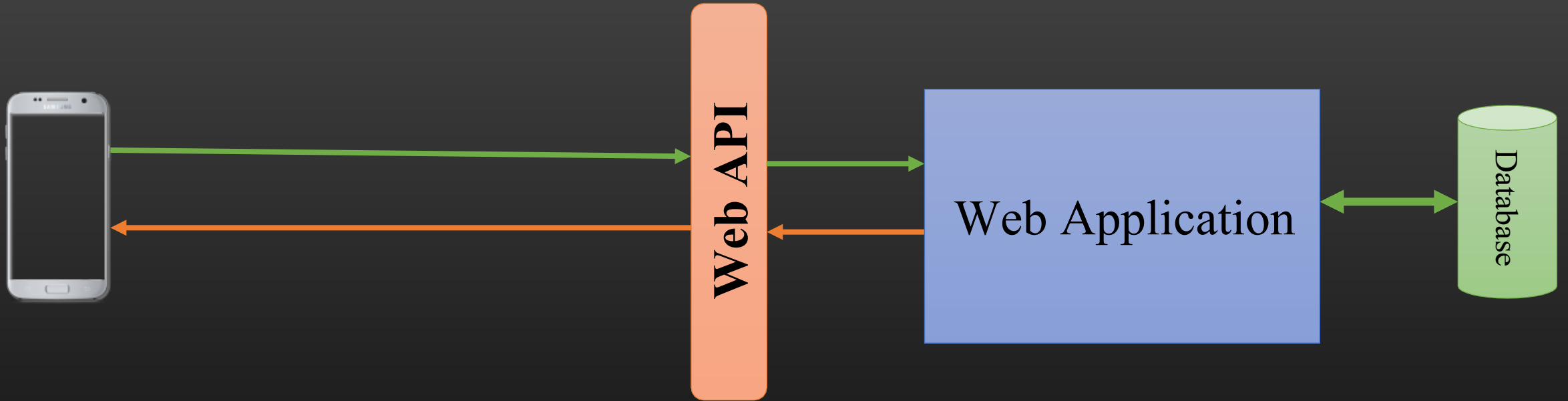
```
210 def __init__(self):
211     self._name = None
212     self._email = None
213     self._password = None
214
215 def __str__(self):
216     return f'Name: {self._name}, Email: {self._email}, Password: {self._password}'
217
218 def __repr__(self):
219     return f'<__main__.Student object at {hex(id(self))}>'
220
221 def __setattr__(self, name, value):
222     if name in ['_name', '_email', '_password']:
223         self.__dict__[name] = value
224     else:
225         raise AttributeError(f'Attribute {name} not found')
226
227 def __getattr__(self, name):
228     if name in ['_name', '_email', '_password']:
229         return self.__dict__[name]
230     else:
231         raise AttributeError(f'Attribute {name} not found')
232
233 def __del__(self):
234     print(f'Deleting {self._name}')
235
236 def __main__():
237     s = Student('Raj', 'raj@gmail.com', 'sonam1234')
238     print(s)
239     print(repr(s))
240     s._email = 'rahul@gmail.com'
241     print(s)
242     print(repr(s))
243     del s
244     print('Done')
```

REST Web API

Web Application



How REST 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 Response Data to Client

Note – Json Data, XML Data

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 Complete Update - PUT Partial Update - PATCH
Delete	DELETE	Deleting Data

Students API Resource

http://geekyshows.com/api/students



1. Base URL

2. Naming Convention

3. Resource of API or End-Point

Request - Response

- Request for All Students

Request

GET: /api/students

Response

```
[  
  {"id":1, "name": "Rahul"},  
  {"id":2, "name" : "Sonam"},  
  ....  
]
```


Request - Response

- Request for One Student having id = 1

Request

GET: /api/students/1

Response

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

Request - Response

- Request Posting/Creating/Inserting Data

Request

POST: api/students

{"name": "Sumit"}

Response

{"id":11, "name": "Sumit"}

Request - Response

- Request for Updating Data, id = 1

Request

PUT or PATCH: /api/students/1

{"name": "Raj" }

Response

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

Request - Response

- Request for Deleting Data, id = 1

Request

DELETE: /api/students/1

Response

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

RESTful API

<http://geekyshows.com/api/students>

GET	/api/students
GET	/api/students/1
POST	/api/students
PUT	/api/students/1
PATCH	/api/students/1
DELETE	/api/students/1

Django REST Framework

Django REST framework is a powerful and flexible toolkit for building Web APIs.

- The Web browsable API is a huge usability win for your developers.
- Authentication policies including packages for OAuth1 and OAuth2.
- Serialization that supports both ORM and non-ORM data sources.
- Customizable all the way down - just use regular function-based views if you don't need the more powerful features.
- Extensive documentation, and great community support.
- Used and trusted by internationally recognized companies including Mozilla, Red Hat, Heroku, and Eventbrite.

Requirements

- Python
- Django

The following packages are optional:

- PyYAML, uritemplate (5.1+, 3.0.0+) - Schema generation support.
- Markdown (3.0.0+) - Markdown support for the browsable API.
- Pygments (2.4.0+) - Add syntax highlighting to Markdown processing.
- django-filter (1.0.1+) - Filtering support.
- django-guardian (1.1.1+) - Object level permissions support.

How to Install/Uninstall DRF

- Install using pip

```
pip install djangorestframework
```

- Uninstall using pip

```
pip uninstall djangorestframework
```


Installing DRF to Django Project

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

URL to useBrowsable API

```
urlpatterns = [  
    ...  
    path(api-auth/, include('rest_framework.urls'))  
]
```

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}
```

Serializers

In Django REST Framework, serializers are responsible for converting complex data such as querysets and model instances to native Python datatypes (called serialization) that can then be easily rendered into JSON, XML or other content types which is understandable by Front End.

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.

- Serialization
- Deserialization

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.

How to Create Serializer Class

- Create a separate serializers.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



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)
```

Serialization

- Creating Query Set

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

- Converting Query Set stu to List of Python Dict / Serializing Query Set

```
serializer = StudentSerializer(stu, many=True)
```

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.

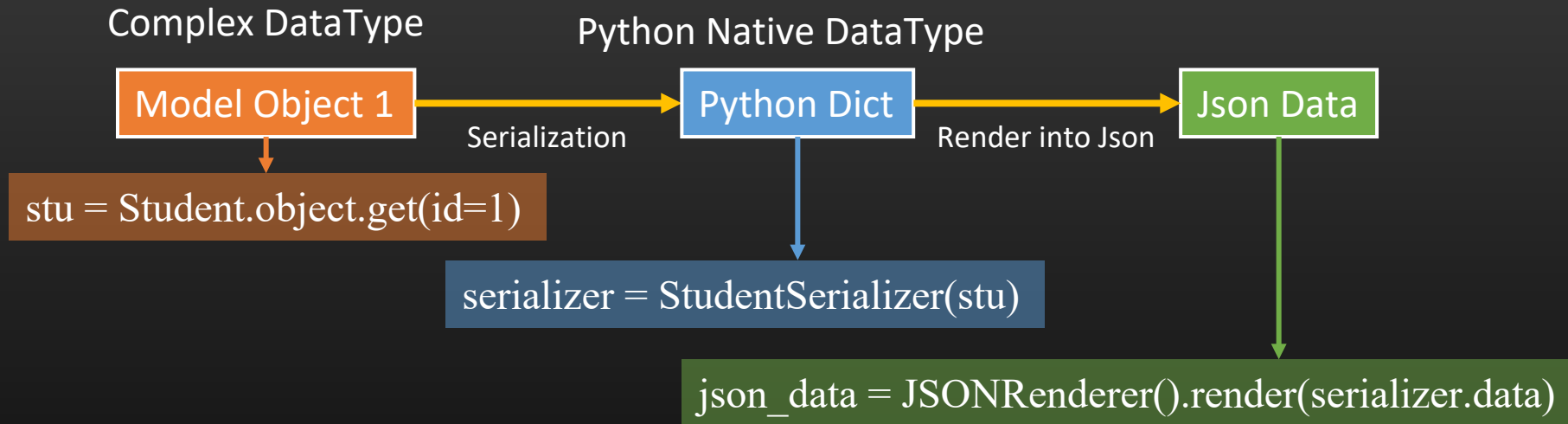
Importing JSONRenderer

```
from rest_framework.renderers import JSONRenderer
```

Render the Data into Json

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

ID	NAME	ROLL	CITY	
1	Sonam	101	Ranchi	← Model Object 1
2	Rahul	102	Ranchi	← Model Object 2
3	Raj	103	Bokaro	← Model Object 3



JsonResponse()

`JsonResponse(data, encoder=DjangoJSONEncoder, safe=True, json_dumps_params=None, **kwargs)`

An `HttpResponse` subclass that helps to create a JSON-encoded response. It inherits most behavior from its superclass with a couple differences:

- Its default Content-Type header is set to *application/json*.
- The first parameter, *data*, should be a *dict* instance. If the *safe* parameter is set to `False` it can be any JSON-serializable object.
- The encoder, which defaults to `django.core.serializers.json.DjangoJSONEncoder`, will be used to serialize the data.
- The *safe* boolean parameter defaults to `True`. If it's set to `False`, any object can be passed for serialization (otherwise only `dict` instances are allowed). If *safe* is `True` and a non-`dict` object is passed as the first argument, a `TypeError` will be raised.
- The `json_dumps_params` parameter is a dictionary of keyword arguments to pass to the `json.dumps()` call used to generate the response.

Serializer Class

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


Serializer Field

Serializer fields handle converting between primitive values and internal datatypes. They also deal with validating input values, as well as retrieving and setting the values from their parent objects.

Syntax:-

```
from rest_framework import serializers  
serializers.Field_Name( )
```

Example:-

```
from rest_framework import serializers  
serializers.CharField( )
```

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.

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 RegexpField 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 RegexpField 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.

Serializer Field

ImageField - An image representation. Validates the uploaded file content as matching a known image format.

Syntax:- ImageField(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.

Note:-

The FileField and ImageField classes are only suitable for use with MultiPartParser or FileUploadParser. Most parsers, such as e.g. JSON don't support file uploads.

Requires either the Pillow package.

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

TimeField - A time representation.

Syntax:- TimeField(format=api_settings.TIME_FORMAT, input_formats=None)

format - A string representing the output format. If not specified, this defaults to the same value as the TIME_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 time objects should be returned by to_representation. In this case the time 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 TIME_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]'.
[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.

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.

Serializer Field

IPAddressField - A field that ensures the input is a valid IPv4 or IPv6 string.

Syntax:- IPAddressField(protocol='both', unpack_ipv4=False, **options)

protocol Limits valid inputs to the specified protocol. Accepted values are 'both' (default), 'IPv4' or 'IPv6'. Matching is case insensitive.

unpack_ipv4 Unpacks IPv4 mapped addresses like ::ffff:192.0.2.1. If this option is enabled that address would be unpacked to 192.0.2.1. Default is disabled. Can only be used when protocol is set to 'both'.

Serializer Field

ChoiceField - A field that can accept a value out of a limited set of choices.

Used by ModelSerializer to automatically generate fields if the corresponding model field includes a `choices=...` argument.

Syntax:- `ChoiceField(choices)`

`choices` - A list of valid values, or a list of (key, `display_name`) tuples.

`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. `allow_blank` should be preferred for textual choices. Defaults to `False`.

`html_cutoff` - If set this will be the maximum number of choices that will be displayed by a HTML select drop down. Can be used to ensure that automatically generated ChoiceFields with very large possible selections do not prevent a template from rendering. Defaults to `None`.

`html_cutoff_text` - If set this will display a textual indicator if the maximum number of items have been cutoff in an HTML select drop down. Defaults to "More than {count} items..."

`allow_null` - `allow_null` should be preferred for numeric or other non-textual choices.

Serializer Field

`MultipleChoiceField` - A field that can accept a set of zero, one or many values, chosen from a limited set of choices. Takes a single mandatory argument. `to_internal_value` returns a set containing the selected values.

Syntax:- `MultipleChoiceField(choices)`

`choices` - A list of valid values, or a list of (key, `display_name`) tuples.

`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. `allow_blank` should be preferred for textual choices. Defaults to `False`. Defaults to `False`.

`html_cutoff` - If set this will be the maximum number of choices that will be displayed by a HTML select drop down. Can be used to ensure that automatically generated `ChoiceFields` with very large possible selections do not prevent a template from rendering. Defaults to `None`.

`html_cutoff_text` - If set this will display a textual indicator if the maximum number of items have been cutoff in an HTML select drop down. Defaults to "More than {count} items..."

`allow_null` - `allow_null` should be preferred for numeric or other non-textual choices.

Serializer Field

ListField - A field class that validates a list of objects.

Sntax:- ListField(child=<A_FIELD_INSTANCE>, allow_empty=True, min_length=None, max_length=None)

child - A field instance that should be used for validating the objects in the list. If this argument is not provided then objects in the list will not be validated.

allow_empty - Designates if empty lists are allowed.

min_length - Validates that the list contains no fewer than this number of elements.

max_length - Validates that the list contains no more than this number of elements.

Serializer Field

DictField - A field class that validates a dictionary of objects. The keys in DictField are always assumed to be string values.

Syntax:- DictField(child=<A_FIELD_INSTANCE>, allow_empty=True)

child - A field instance that should be used for validating the values in the dictionary. If this argument is not provided then values in the mapping will not be validated.

allow_empty - Designates if empty dictionaries are allowed.

Serializer Field

HStoreField - A preconfigured DictField that is compatible with Django's postgres HStoreField.

Syntax:- HStoreField(child=<A_FIELD_INSTANCE>, allow_empty=True)

child - A field instance that is used for validating the values in the dictionary. The default child field accepts both empty strings and null values.

allow_empty - Designates if empty dictionaries are allowed.

Note that the child field must be an instance of CharField, as the hstore extension stores values as strings.

Serializer Field

JSONField - A field class that validates that the incoming data structure consists of valid JSON primitives. In its alternate binary mode, it will represent and validate JSON-encoded binary strings.

Syntax: `JSONField(binary, encoder)`

binary - If set to True then the field will output and validate a JSON encoded string, rather than a primitive data structure. Defaults to False.

encoder - Use this JSON encoder to serialize input object. Defaults to None.

ReadOnlyField - A field class that simply returns the value of the field without modification.

This field is used by default with ModelSerializer when including field names that relate to an attribute rather than a model field.

Syntax: `ReadOnlyField()`

Serializer Field

HiddenField - A field class that does not take a value based on user input, but instead takes its value from a default value or callable. The HiddenField class is usually only needed if you have some validation that needs to run based on some pre-provided field values, but you do not want to expose all of those fields to the end user.

Syntax:- HiddenField()

Serializer Field

ModelField - A generic field that can be tied to any arbitrary model field. The ModelField class delegates the task of serialization/deserialization to its associated model field. This field can be used to create serializer fields for custom model fields, without having to create a new custom serializer field.

This field is used by ModelSerializer to correspond to custom model field classes.

Syntax:- `ModelField(model_field=<Django ModelField instance>)`

The ModelField class is generally intended for internal use, but can be used by your API if needed. In order to properly instantiate a ModelField, it must be passed a field that is attached to an instantiated model. For example:

`ModelField(model_field=MyModel()._meta.get_field('custom_field'))`

Serializer Field

SerializerMethodField - This is a read-only field. It gets its value by calling a method on the serializer class it is attached to. It can be used to add any sort of data to the serialized representation of your object.

Syntax: `SerializerMethodField(method_name=None)`

`method_name` - The name of the method on the serializer to be called. If not included this defaults to `get_<field_name>`.

The serializer method referred to by the `method_name` argument should accept a single argument (in addition to `self`), which is the object being serialized. It should return whatever you want to be included in the serialized representation of the object.

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.

`help_text` - A text string that may be used as a description of the field in HTML form fields or other descriptive elements.

Core Arguments

required - Normally an error will be raised if a field is not supplied during deserialization. Set to false if this field is not required to be present during deserialization.

Setting this to False also allows the object attribute or dictionary key to be omitted from output when serializing the instance. If the key is not present it will simply not be included in the output representation.

Defaults to True.

default - If set, this gives the default value that will be used for the field if no input value is supplied. If not set the default behaviour is to not populate the attribute at all.

The default is not applied during partial update operations. In the partial update case only fields that are provided in the incoming data will have a validated value returned.

Note that setting a default value implies that the field is not required. Including both the default and required keyword arguments is invalid and will raise an error.

Core Arguments

initial - A value that should be used for pre-populating the value of HTML form fields.

style - A dictionary of key-value pairs that can be used to control how renderers should render the field.

Example:-

```
password = serializers.CharField(  
    max_length=100,  
    style={'input_type': 'password', 'placeholder': 'Password'}  
)
```

Core Arguments

`read_only` - Read-only fields are included in the API output, but should not be included in the input during create or update operations. Any 'read_only' fields that are incorrectly included in the serializer input will be ignored.

Set this to True to ensure that the field is used when serializing a representation, but is not used when creating or updating an instance during deserialization.

Defaults to False

`write_only` - Set this to True to ensure that the field may be used when updating or creating an instance, but is not included when serializing the representation.

Defaults to False

Core Arguments

`allow_null` - Normally an error will be raised if `None` is passed to a serializer field. Set this keyword argument to `True` if `None` should be considered a valid value.

Note that, without an explicit default, setting this argument to `True` will imply a default value of `null` for serialization output, but does not imply a default for input deserialization.

Defaults to `False`

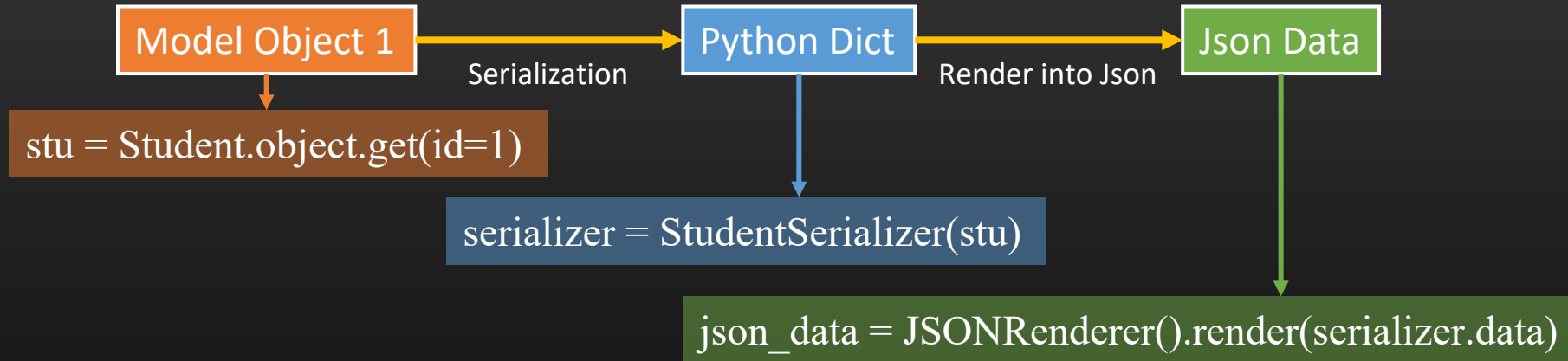
`source` - The name of the attribute that will be used to populate the field. May be a method that only takes a `self` argument, such as `URLField(source='get_absolute_url')`, or may use dotted notation to traverse attributes, such as `EmailField(source='user.email')`. When serializing fields with dotted notation, it may be necessary to provide a default value if any object is not present or is empty during attribute traversal.

The value `source='*'` has a special meaning, and is used to indicate that the entire object should be passed through to the field. This can be useful for creating nested representations, or for fields which require access to the complete object in order to determine the output representation.

Defaults to the name of the field.

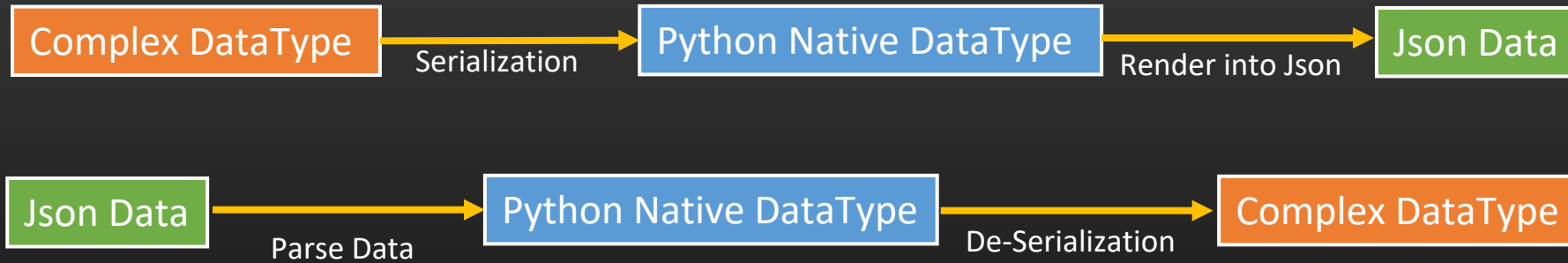
Serialization

ID	NAME	ROLL	CITY	
1	Sonam	101	Ranchi	← Model Object 1
2	Rahul	102	Ranchi	← Model Object 2
3	Raj	103	Bokaro	← Model Object 3



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)
```

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

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)
```

New Data from user for updation

```
    def update(self, instance, validated_data):
```

Old Data stored in Database

```
        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
```

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()
```

Validation

- Field Level Validation
- Object Level Validation
- Validators

Field Level Validation

We can specify custom field-level validation by adding *validate_fieldName* methods to your Serializer subclass.

These are similar to the *clean_fieldName* methods on Django forms.

validate_fieldName methods should return the validated value or raise a `serializers.ValidationError`


Syntax:- `def validate_fieldname(self, value)`

Example:- `def validate_roll(self, value)`

Where, value is the field value that requires validation.

Field Level Validation

```
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 validate_roll(self, value):  
        if value >= 200 :  
            raise serializers.ValidationError('Seat Full')  
        return value
```



This Method is automatically invoked
when is_valid() method is called

Object Level Validation

When we need to do validation that requires access to multiple fields we do object level validation by adding a method called *validate()* to Serializer subclass.

It raises a serializers.ValidationError if necessary, or just return the validated values.

Syntax:- `def validate (self, data)`

Example:- `def validate (self, data)`

Where, data is a dictionary of field values.

Object Level Validation

```
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 validate(self, data):
```

```
        nm = data.get('name')
```

```
        ct = data.get('city')
```

```
        if nm.lower() == 'rohit' and ct.lower() != 'ranchi' :
```

```
            raise serializers.ValidationError('City must be Ranchi')
```

```
        return data
```

data is a python dictionary of field values.

Validators

Most of the time you're dealing with validation in REST framework you'll simply be relying on the default field validation, or writing explicit validation methods on serializer or field classes.

However, sometimes you'll want to place your validation logic into reusable components, so that it can easily be reused throughout your codebase. This can be achieved by using validator functions and validator classes.

Validators

REST framework the validation is performed entirely on the serializer class. This is advantageous for the following reasons:

- It introduces a proper separation of concerns, making your code behavior more obvious.
- It is easy to switch between using shortcut `ModelSerializer` classes and using explicit `Serializer` classes. Any validation behavior being used for `ModelSerializer` is simple to replicate.
- Printing the *repr()* of a serializer instance will show you exactly what validation rules it applies. There's no extra hidden validation behavior being called on the model instance.
- When you're using `ModelSerializer` all of this is handled automatically for you. If you want to drop down to using `Serializer` classes instead, then you need to define the validation rules explicitly.

Validators

```
from rest_framework import serializers

def starts_with_r(value):
    if value[0].lower() != 'r' :
        raise serializers.ValidationError('Name should start with R')

class StudentSerializer(serializers.Serializer):
    name = serializers.CharField(max_length=100, validators=[starts_with_r])
    roll = serializers.IntegerField()
    city = serializers.CharField(max_length=100)
```

Validation Priority

- Validators
- Field Level Validation
- Object Level Validation

ModelSerializer Class

The ModelSerializer class provides a shortcut that lets you automatically create a Serializer class with fields that correspond to the Model fields.

The ModelSerializer class is the same as a regular Serializer class, except that:

- It will automatically generate a set of fields for you, based on the model.
- It will automatically generate validators for the serializer, such as `unique_together` validators.
- It includes simple default implementations of `create()` and `update()`.

Create ModelSerializer Class

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

```
from rest_framework import serializers
```

```
class StudentSerializer(serializers.ModelSerializer):
```

```
    class Meta:
```

```
        model = Student
```

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

- fields = '__all__'
- exclude = ['roll']

Create ModelSerializer Class

```
from rest_framework import serializers

class StudentSerializer(serializers.ModelSerializer):
    name = serializers.CharField(read_only=True)

    class Meta:
        model = Student
        fields = ['id', 'name', 'roll', 'city']

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

Create ModelSerializer Class

```
from rest_framework import serializers

class StudentSerializer(serializers.ModelSerializer):
    class Meta:
        model = Student
        fields = ['id', 'name', 'roll', 'city']
        extra_kwargs = {'name': {'read_only': True}}
```

ModelSerializer Validation

```
from rest_framework import serializers

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

    def validate_roll(self, value):
        if value >= 200 :
            raise serializers.ValidationError('Seat Full')
        return value
```

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)
```

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

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.

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.

DRF Status

REST framework includes a set of named constants that you can use to make your code more obvious and readable.

The full set of HTTP status codes included in the status module.

Example:-

```
from rest_framework import status

def student_create(request):
    res = {'msg': 'Data Created'}
    return Response(res, status=status.HTTP_201_CREATED)
```

Status Code – 1xx

Informational - 1xx

This class of status code indicates a provisional response. There are no 1xx status codes used in REST framework by default.

HTTP_100_CONTINUE

HTTP_101_SWITCHING_PROTOCOLS

Status Code – 2xx

Successful - 2xx

This class of status code indicates that the client's request was successfully received, understood, and accepted.

HTTP_200_OK

HTTP_201_CREATED

HTTP_202_ACCEPTED

HTTP_203_NON_AUTHORITATIVE_INFORMATION

HTTP_204_NO_CONTENT

HTTP_205_RESET_CONTENT

HTTP_206_PARTIAL_CONTENT

HTTP_207_MULTI_STATUS

HTTP_208_ALREADY_REPORTED

HTTP_226_IM_USED

Status Code – 3xx

Redirection - 3xx

This class of status code indicates that further action needs to be taken by the user agent in order to fulfill the request.

HTTP_300_MULTIPLE_CHOICES

HTTP_301_MOVED_PERMANENTLY

HTTP_302_FOUND

HTTP_303_SEE_OTHER

HTTP_304_NOT_MODIFIED

HTTP_305_USE_PROXY

HTTP_306_RESERVED

HTTP_307_TEMPORARY_REDIRECT

HTTP_308_PERMANENT_REDIRECT

Status Code – 4xx

Client Error - 4xx

The 4xx class of status code is intended for cases in which the client seems to have erred. Except when responding to a HEAD request, the server SHOULD include an entity containing an explanation of the error situation, and whether it is a temporary or permanent condition.

HTTP_400_BAD_REQUEST

HTTP_401_UNAUTHORIZED

HTTP_402_PAYMENT_REQUIRED

HTTP_403_FORBIDDEN

HTTP_404_NOT_FOUND

HTTP_405_METHOD_NOT_ALLOWED

HTTP_406_NOT_ACCEPTABLE

HTTP_407_PROXY_AUTHENTICATION_REQUIRED

HTTP_408_REQUEST_TIMEOUT

HTTP_409_CONFLICT

Status Code – 4xx

HTTP_410_GONE

HTTP_411_LENGTH_REQUIRED

HTTP_412_PRECONDITION_FAILED

HTTP_413_REQUEST_ENTITY_TOO_LARGE

HTTP_414_REQUEST_URI_TOO_LONG

HTTP_415_UNSUPPORTED_MEDIA_TYPE

HTTP_416_REQUESTED_RANGE_NOT_SATISFIABLE

HTTP_417_EXPECTATION_FAILED

HTTP_422_UNPROCESSABLE_ENTITY

HTTP_423_LOCKED

HTTP_424_FAILED_DEPENDENCY

HTTP_426_UPGRADE_REQUIRED

HTTP_428_PRECONDITION_REQUIRED

HTTP_429_TOO_MANY_REQUESTS

Status Code – 4xx

HTTP_431_REQUEST_HEADER_FIELDS_TOO_LARGE

HTTP_451_UNAVAILABLE_FOR_LEGAL_REASONS

Status Code – 5xx

Server Error - 5xx

Response status codes beginning with the digit "5" indicate cases in which the server is aware that it has erred or is incapable of performing the request. Except when responding to a HEAD request, the server SHOULD include an entity containing an explanation of the error situation, and whether it is a temporary or permanent condition.

HTTP_500_INTERNAL_SERVER_ERROR

HTTP_501_NOT_IMPLEMENTED

HTTP_502_BAD_GATEWAY

HTTP_503_SERVICE_UNAVAILABLE

HTTP_504_GATEWAY_TIMEOUT

HTTP_505_HTTP_VERSION_NOT_SUPPORTED

HTTP_506_VARIANT_ALSO_NEGOTIATES

HTTP_507_INSUFFICIENT_STORAGE

HTTP_508_LOOP_DETECTED

Status Code – 5xx

HTTP_509_BANDWIDTH_LIMIT_EXCEEDED

HTTP_510_NOT_EXTENDED

HTTP_511_NETWORK_AUTHENTICATION_REQUIRED

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)
```

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.

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.

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`

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)
```


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.

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)
```

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)
```


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
```

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
```

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
```

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
```

ViewSet

Django REST framework allows you to combine the logic for a set of related views in a single class, called a ViewSet.

There are two main advantages of using a ViewSet over using a View class.

- Repeated logic can be combined into a single class.
- By using routers, we no longer need to deal with wiring up the URL conf ourselves.

ViewSet Class

A ViewSet class is simply a type of class-based View, that does not provide any method handlers such as `get()` or `post()`, and instead provides actions such as `list()` and `create()`.

- `list()` – Get All Records.
- `retrieve()` – Get Single Record
- `create()` – Create/Insert Record
- `update()` – Update Record Completely
- `partial_update()` – Update Record Partially
- `destroy()` – Delete Record

ViewSet Class

```
from rest_framework import viewsets

class StudentViewSet(viewsets.ViewSet):
    def list(self, request): .....
    def create(self, request): .....
    def retrieve(self, request, pk=None): .....
    def update(self, request, pk=None): .....
    def partial_update(self, request, pk=None): .....
    def destroy(self, request, pk=None): .....
```

ViewSet Class

During dispatch, the following attributes are available on the ViewSet:-

- `basename` - the base to use for the URL names that are created.
- `action` - the name of the current action (e.g., `list`, `create`).
- `detail` - boolean indicating if the current action is configured for a list or detail view.
- `suffix` - the display suffix for the viewset type - mirrors the `detail` attribute.
- `name` - the display name for the viewset. This argument is mutually exclusive to `suffix`.
- `description` - the display description for the individual view of a viewset.

ViewSet – URL Config

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

```
from api import views
```

```
from rest_framework.routers import DefaultRouter
```

```
router = DefaultRouter()
```

Creating Default Router Object




Register StudentViewSet with Router



```
router.register('studentapi', views.StudentViewSet, basename='student')
```

```
urlpatterns = [  
    path("", include(router.urls)),  
]
```

The API URLs are now determined automatically by the router.



ModelViewSet Class

The ModelViewSet class inherits from GenericAPIView and includes implementations for various actions, by mixing in the behavior of the various mixin classes.

The actions provided by the ModelViewSet class are list(), retrieve(), create(), update(), partial_update(), and destroy(). You can use any of the standard attributes or method overrides provided by GenericAPIView

```
class StudentModelViewSet(viewsets.ModelViewSet):
```

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

```
    serializer_class = StudentSerializer
```


ReadOnlyModelViewSet Class

The ReadOnlyModelViewSet class also inherits from GenericAPIView. As with ModelViewSet it also includes implementations for various actions, but unlike ModelViewSet only provides the 'read-only' actions, list() and retrieve(). You can use any of the standard attributes and method overrides available to GenericAPIView

```
class StudentReadOnlyModelViewSet(viewsets.ReadOnlyModelViewSet):  
    queryset = Student.objects.all()  
    serializer_class = StudentSerializer
```

Why use Authentication & Permission?

Currently our API doesn't have any restrictions on who can edit or delete Data. We'd like to have some more advanced behavior in order to make sure that:

- Data is always associated with a creator.
- Only authenticated users may create Data.
- Only the creator of a Data may update or delete it.
- Unauthenticated requests should have full read-only access.

Authentication

Authentication is the mechanism of associating an incoming request with a set of identifying credentials, such as the user the request came from, or the token that it was signed with. The permission and throttling policies can then use those credentials to determine if the request should be permitted.

Authentication is always run at the very start of the view, before the permission and throttling checks occur, and before any other code is allowed to proceed.

Authentication

REST framework provides a number of authentication schemes out of the box, and also allows you to implement custom schemes.

- BasicAuthentication
- SessionAuthentication
- TokenAuthentication
- RemoteUserAuthentication
- Custom authentication

BasicAuthentication

This authentication scheme uses HTTP Basic Authentication, signed against a user's username and password.

Basic authentication is generally only appropriate for testing.

If successfully authenticated, BasicAuthentication provides the following credentials.

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

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

WWW-Authenticate: Basic realm="api"

BasicAuthentication

Note: If you use BasicAuthentication in production you must ensure that your API is only available over https.

You should also ensure that your API clients will always re-request the username and password at login, and will never store those details to persistent storage.

Permission

Permissions are used to grant or deny access for different classes of users to different parts of the API.

Permission checks are always run at the very start of the view, before any other code is allowed to proceed.

Permission checks will typically use the authentication information in the *request.user* and *request.auth* properties to determine if the incoming request should be permitted.

Permission Classes

Permissions in REST framework are always defined as a list of permission classes.

- AllowAny
- IsAuthenticated
- IsAdminUser
- IsAuthenticatedOrReadOnly
- DjangoModelPermissions
- DjangoModelPermissionsOrAnonReadOnly
- DjangoObjectPermissions
- Custom Permissions

AllowAny

The AllowAny permission class will allow unrestricted access, regardless of if the request was authenticated or unauthenticated.

This permission is not strictly required, since you can achieve the same result by using an empty list or tuple for the permissions setting, but you may find it useful to specify this class because it makes the intention explicit.

IsAuthenticated

The IsAuthenticated permission class will deny permission to any unauthenticated user, and allow permission otherwise.

This permission is suitable if you want your API to only be accessible to registered users.

IsAdminUser

The IsAdminUser permission class will deny permission to any user, unless `user.is_staff` is True in which case permission will be allowed.

This permission is suitable if you want your API to only be accessible to a subset of trusted administrators.

Authentication

REST framework provides a number of authentication schemes out of the box, and also allows you to implement custom schemes.

- BasicAuthentication
- SessionAuthentication
- TokenAuthentication
- RemoteUserAuthentication
- Custom authentication

SessionAuthentication

This authentication scheme uses Django's default session backend for authentication. Session authentication is appropriate for AJAX clients that are running in the same session context as your website.

If successfully authenticated, SessionAuthentication provides the following credentials.

`request.user` will be a Django User instance.

`request.auth` will be `None`.

Unauthenticated responses that are denied permission will result in an HTTP 403 Forbidden response.

SessionAuthentication

If you're using an AJAX style API with SessionAuthentication, you'll need to make sure you include a valid CSRF token for any "unsafe" HTTP method calls, such as PUT, PATCH, POST or DELETE requests.

Warning: Always use Django's standard login view when creating login pages. This will ensure your login views are properly protected.

CSRF validation in REST framework works slightly differently to standard Django due to the need to support both session and non-session based authentication to the same views. This means that only authenticated requests require CSRF tokens, and anonymous requests may be sent without CSRF tokens. This behaviour is not suitable for login views, which should always have CSRF validation applied.

Permission

Permissions are used to grant or deny access for different classes of users to different parts of the API.

Permission checks are always run at the very start of the view, before any other code is allowed to proceed.

Permission checks will typically use the authentication information in the *request.user* and *request.auth* properties to determine if the incoming request should be permitted.

Permission Classes

Permissions in REST framework are always defined as a list of permission classes.

- AllowAny
- IsAuthenticated
- IsAdminUser
- IsAuthenticatedOrReadOnly
- DjangoModelPermissions
- DjangoModelPermissionsOrAnonReadOnly
- DjangoObjectPermissions
- Custom Permissions

AllowAny

The AllowAny permission class will allow unrestricted access, regardless of if the request was authenticated or unauthenticated.

This permission is not strictly required, since you can achieve the same result by using an empty list or tuple for the permissions setting, but you may find it useful to specify this class because it makes the intention explicit.

IsAuthenticated

The IsAuthenticated permission class will deny permission to any unauthenticated user, and allow permission otherwise.

This permission is suitable if you want your API to only be accessible to registered users.

IsAdminUser

The IsAdminUser permission class will deny permission to any user, unless `user.is_staff` is True in which case permission will be allowed.

This permission is suitable if you want your API to only be accessible to a subset of trusted administrators.

IsAuthenticatedOrReadOnly

The `IsAuthenticatedOrReadOnly` will allow authenticated users to perform any request. Requests for unauthorised users will only be permitted if the request method is one of the "safe" methods; GET, HEAD or OPTIONS.

This permission is suitable if you want to your API to allow read permissions to anonymous users, and only allow write permissions to authenticated users.

DjangoModelPermissions

This permission class ties into Django's standard `django.contrib.auth` model permissions. This permission must only be applied to views that have a *queryset* property set. Authorization will only be granted if the user is authenticated and has the relevant model permissions assigned.

- POST requests require the user to have the add permission on the model.
- PUT and PATCH requests require the user to have the change permission on the model.
- DELETE requests require the user to have the delete permission on the model.

The default behaviour can also be overridden to support custom model permissions. For example, you might want to include a view model permission for GET requests.

To use custom model permissions, override `DjangoModelPermissions` and set the *perms_map* property.

DjangoModelPermissionsOrAnonReadOnly

Similar to DjangoModelPermissions, but also allows unauthenticated users to have read-only access to the API.

DjangoObjectPermissions

This permission class ties into Django's standard object permissions framework that allows per-object permissions on models. In order to use this permission class, you'll also need to add a permission backend that supports object-level permissions, such as `django-guardian`.

As with `DjangoModelPermissions`, this permission must only be applied to views that have a `queryset` property or `get_queryset()` method. Authorization will only be granted if the user is authenticated and has the relevant per-object permissions and relevant model permissions assigned.

- POST requests require the user to have the add permission on the model instance.
- PUT and PATCH requests require the user to have the change permission on the model instance.
- DELETE requests require the user to have the delete permission on the model instance.

Custom Permissions

To implement a custom permission, override `BasePermission` and implement either, or both, of the following methods:

- `has_permission(self, request, view)`
- `has_object_permission(self, request, view, obj)`

The methods should return `True` if the request should be granted access, and `False` otherwise.

Custom Permissions

custompermissions.py

```
class MyPermission(BasePermission):  
    def has_permission(self, request, view)
```

Permission

Third party packages:-

- DRF - Access Policy
- Composed Permissions
- REST Condition
- DRY Rest Permissions
- Django Rest Framework Roles
- Django REST Framework API Key
- Django Rest Framework Role Filters
- Django Rest Framework PSQ

Authentication

- BasicAuthentication
- SessionAuthentication
- TokenAuthentication
- RemoteUserAuthentication
- Custom Authentication

TokenAuthentication

This authentication scheme uses a simple token-based HTTP Authentication scheme. Token authentication is appropriate for client-server setups, such as native desktop and mobile clients.

To use the TokenAuthentication scheme you'll need to configure the authentication classes to include TokenAuthentication, and additionally include `rest_framework.authtoken` in your `INSTALLED_APPS` setting:

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

Note: Make sure to run `manage.py migrate` after changing your settings. The `rest_framework.authtoken` app provides Django database migrations.

TokenAuthentication

You'll also need to create tokens for your users.

```
from rest_framework.authtoken.models import Token  
token = Token.objects.create(user=...)  
print(token.key)
```

For clients to authenticate, the token key should be included in the Authorization HTTP header. The key should be prefixed by the string literal "Token", with whitespace separating the two strings. For example:

Authorization: Token 9944b09199c62bcf9418ad846dd0e4bbdfc6ee4b

TokenAuthentication

If successfully authenticated, TokenAuthentication provides the following credentials.

request.user will be a Django User instance.

request.auth will be a `rest_framework.authtoken.models.Token` instance.

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

WWW-Authenticate: Token

The http command line tool may be useful for testing token authenticated APIs. For example:

```
http http://127.0.0.1:8000/studentapi/ 'Authorization: Token
9944b09199c62bcf9418ad846dd0e4bbdfc6ee4b'
```

TokenAuthentication

Note: If you use TokenAuthentication in production you must ensure that your API is only available over https.

Generate Token

- Using Admin Application
- Using Django manage.py command

python manage.py drf_create_token <username> - This command will return API Token for the given user or Creates a Token if token doesn't exist for user.

- By exposing an API endpoint
- Using Signals

How Client can Ask/Create Token

When using TokenAuthentication, you may want to provide a mechanism for clients to obtain a token given the username and password.

REST framework provides a built-in view to provide this behavior. To use it, add the `obtain_auth_token` view to your `URLconf`:

```
from rest_framework.authtoken.views import obtain_auth_token

urlpatterns = [
    path('gettoken/', obtain_auth_token)
]
```

The `obtain_auth_token` view will return a JSON response when valid username and password fields are POSTed to the view using form data or JSON:

```
http POST http://127.0.0.1:8000/gettoken/ username="name" password="pass"
{ 'token' : '9944b09199c62bcf9418ad846dd0e4bbdfc6ee4b' }
```

How Client can Ask/Create Token

It also generates token if the token is not generated for the provided user.

Custom Auth Token

We can customize auth token to provide additional details to the client.

```
from rest_framework.authtoken.views import ObtainAuthToken
from rest_framework.authtoken.models import Token
from rest_framework.response import Response

class CustomAuthToken(ObtainAuthToken):
    def post(self, request, *args, **kwargs):
        serializer = self.serializer_class(data=request.data, context={'request': request})
        serializer.is_valid(raise_exception=True)
        user = serializer.validated_data['user']
        token, created = Token.objects.get_or_create(user=user)
        return Response({
            'token': token.key,
            'user_id': user.pk,
            'email': user.email
        })
```

Generate Token by Signals

If you want every user to have an automatically generated Token, you can simply catch the User's post_save signal.

```
from django.conf import settings
from django.db.models.signals import post_save
from django.dispatch import receiver
from rest_framework.authtoken.models import Token
@receiver(post_save, sender=settings.AUTH_USER_MODEL)
def create_auth_token(sender, instance=None, created=False, **kwargs):
    if created:
        Token.objects.create(user=instance)
```

Permission Classes

Permissions in REST framework are always defined as a list of permission classes.

- AllowAny
- IsAuthenticated
- IsAdminUser
- IsAuthenticatedOrReadOnly
- DjangoModelPermissions
- DjangoModelPermissionsOrAnonReadOnly
- DjangoObjectPermissions
- Custom Permissions

httpie

HTTPie (pronounced aitch-tee-tee-pie) is a command line HTTP client. Its goal is to make CLI interaction with web services as human-friendly as possible. It provides a simple http command that allows for sending arbitrary HTTP requests using a simple and natural syntax, and displays colorized output. HTTPie can be used for testing, debugging, and generally interacting with HTTP servers.

Syntax:- http [flags] [METHOD] URL [ITEM [ITEM]]

How to Install

```
pip install httpie
```

Use httpie

GET Request

```
http http://127.0.0.1:8000/studentapi/
```

GET Request with Auth

```
http http://127.0.0.1:8000/studentapi/ 'Authorization:Token  
621cdf999d9151f9aea8e52f00eb436aa680fa24'
```

POST Request/ Submitting Form

```
http -f POST http://127.0.0.1:8000/studentapi/ name=Jay roll=104 city=Dhanbad  
'Authorization:Token 621cdf999d9151f9aea8e52f00eb436aa680fa24'
```

Use httpie

PUT Request

```
http PUT http://127.0.0.1:8000/studentapi/4/ name=Kunal roll=109 city=Bokaro  
'Authorization:Token 621cdf999d9151f9aea8e52f00eb436aa680fa24'
```

Delete Request

```
http DELETE http://127.0.0.1:8000/studentapi/4/ 'Authorization:Token  
621cdf999d9151f9aea8e52f00eb436aa680fa24'
```


Custom Authentication

To implement a custom authentication scheme, subclass `BaseAuthentication` and override the `authenticate(self, request)` method.

The method should return a two-tuple of `(user, auth)` if authentication succeeds, or `None` otherwise.

Authentication

- BasicAuthentication
- SessionAuthentication
- TokenAuthentication
- RemoteUserAuthentication
- Custom authentication

Authentication

Third party packages:-

- Django OAuth Toolkit
- JSON Web Token Authentication
- Hawk HTTP Authentication
- HTTP Signature Authentication
- Djoser
- django-rest-auth / dj-rest-auth
- django-rest-framework-social-oauth2
- django-rest-knox
- drfpasswordless

JSON Web Token (JWT)

JSON Web Token is a fairly new standard which can be used for token-based authentication. Unlike the built-in TokenAuthentication scheme, JWT Authentication doesn't need to use a database to validate a token.

<https://jwt.io/>

Simple JWT

Simple JWT provides a JSON Web Token authentication backend for the Django REST Framework. It aims to cover the most common use cases of JWTs by offering a conservative set of default features. It also aims to be easily extensible in case a desired feature is not present.

<https://django-rest-framework-simplejwt.readthedocs.io/en/latest/>

How to Install Simple JWT

```
pip install djangorestframework-simplejwt
```

Configure Simple JWT

settings.py

```
REST_FRAMEWORK = {  
    'DEFAULT_AUTHENTICATION_CLASSES': (  
        'rest_framework_simplejwt.authentication.JWTAuthentication',  
    ) }
```

urls.py

```
from rest_framework_simplejwt.views import TokenObtainPairView, TokenRefreshView  
urlpatterns = [  
    path('gettoken/', TokenObtainPairView.as_view(), name='token_obtain_pair'),  
    path('refreshtoken/', TokenRefreshView.as_view(), name='token_refresh'),  
]
```

Configure Simple JWT

You can also include a route for Simple JWT's TokenVerifyView if you wish to allow API users to verify HMAC-signed tokens without having access to your signing key.

urls.py

```
from rest_framework_simplejwt.views import TokenObtainPairView, TokenRefreshView,
TokenVerifyView

urlpatterns = [
    path('gettoken/', TokenObtainPairView.as_view(), name='token_obtain_pair'),
    path('refresh token/', TokenRefreshView.as_view(), name='token_refresh'),
    path('verifytoken/', TokenVerifyView.as_view(), name='token_verify'),
]
```


JWT Default Settings

```
from datetime import timedelta
```

```
SIMPLE_JWT = {
```

```
    'ACCESS_TOKEN_LIFETIME': timedelta(minutes=5),
```

```
    'REFRESH_TOKEN_LIFETIME': timedelta(days=1),
```

```
    'ROTATE_REFRESH_TOKENS': False,
```

```
    'BLACKLIST_AFTER_ROTATION': True,
```

```
    'ALGORITHM': 'HS256',
```

```
    'SIGNING_KEY': settings.SECRET_KEY,
```

```
    'VERIFYING_KEY': None,
```

```
    'AUDIENCE': None,
```

```
    'ISSUER': None,
```

JWT Default Settings

```
'AUTH_HEADER_TYPES': ('Bearer',),  
'USER_ID_FIELD': 'id',  
'USER_ID_CLAIM': 'user_id',  
  
'AUTH_TOKEN_CLASSES': ('rest_framework_simplejwt.tokens.AccessToken',),  
'TOKEN_TYPE_CLAIM': 'token_type',  
  
'JTI_CLAIM': 'jti',  
  
'SLIDING_TOKEN_REFRESH_EXP_CLAIM': 'refresh_exp',  
'SLIDING_TOKEN_LIFETIME': timedelta(minutes=5),  
'SLIDING_TOKEN_REFRESH_LIFETIME': timedelta(days=1),  
}
```

JWT Default Settings

`ACCESS_TOKEN_LIFETIME` - A `datetime.timedelta` object which specifies how long access tokens are valid.

`REFRESH_TOKEN_LIFETIME` - A `datetime.timedelta` object which specifies how long refresh tokens are valid.

Use JWT

GET Token

```
http POST http://127.0.0.1:8000/gettoken/ username="user1"
password="geekyshows"
```

Verify Token

```
http POST http://127.0.0.1:8000/verifytoken/  
token="eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ0b2t1bl90eXB1IjoiYWNjZ  
XNzIiwiaXhwIjoxNjAzNjIxOTAxLCJqdGkiOiI2NmM4ZWJiYjUwMWMM0MzA3  
YWJjMGJNTY2ZmNmNTJiMyIsInVzZXJfaWQiOiJ9.cwUSWrkFnFdO8fP45aE  
P6GDa3yaybSVYAG6vGUlkFOo"
```

Use JWT

Refresh Token

```
http POST http://127.0.0.1:8000/refresh token/  
refresh="eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ0b2t1b190eXB1IjoicmVmcmVzaCI6ImV4cCI6MTYwMzcwODAwMSwianRpljoiYzYzODBmYjVjMDk3NDVhNjkyYzA5YWRmMGI1ZDQ5OWIiLCJ1c2VyX2lkIjoyfQ.Q-E-8N8VvSZof5IjoNIL-2KECRLqlYzBojbTCj_4dBc"
```

Permission Classes

Permissions in REST framework are always defined as a list of permission classes.

- AllowAny
- IsAuthenticated
- IsAdminUser
- IsAuthenticatedOrReadOnly
- DjangoModelPermissions
- DjangoModelPermissionsOrAnonReadOnly
- DjangoObjectPermissions
- Custom Permissions

Throttling

Throttling is similar to permissions, in that it determines if a request should be authorized. Throttles indicate a temporary state, and are used to control the rate of requests that clients can make to an API.

Your API might have a restrictive throttle for unauthenticated requests, and a less restrictive throttle for authenticated requests.

Throttling

The default throttling policy may be set globally, using the `DEFAULT_THROTTLE_CLASSES` and `DEFAULT_THROTTLE_RATES` settings. For example.

```
REST_FRAMEWORK = {  
    'DEFAULT_THROTTLE_CLASSES': [  
        'rest_framework.throttling.AnonRateThrottle',  
        'rest_framework.throttling.UserRateThrottle'  
    ],  
    'DEFAULT_THROTTLE_RATES': {  
        'anon': '100/day',  
        'user': '1000/day'  
    }  
}
```

Note:- The rate descriptions used in `DEFAULT_THROTTLE_RATES` may include second, minute, hour or day as the throttle period.

AnonRateThrottle

The AnonRateThrottle will only ever throttle unauthenticated users. The IP address of the incoming request is used to generate a unique key to throttle against.

The allowed request rate is determined from one of the following (in order of preference).

The rate property on the class, which may be provided by overriding AnonRateThrottle and setting the property.

The DEFAULT_THROTTLE_RATES['anon'] setting.

AnonRateThrottle is suitable if you want to restrict the rate of requests from unknown sources.

UserRateThrottle

The UserRateThrottle will throttle users to a given rate of requests across the API. The user id is used to generate a unique key to throttle against. Unauthenticated requests will fall back to using the IP address of the incoming request to generate a unique key to throttle against.

The allowed request rate is determined from one of the following (in order of preference).

The rate property on the class, which may be provided by overriding UserRateThrottle and setting the property.

The DEFAULT_THROTTLE_RATES['user'] setting.

ScopedRateThrottle

The ScopedRateThrottle class can be used to restrict access to specific parts of the API. This throttle will only be applied if the view that is being accessed includes a *throttle_scope* property. The unique throttle key will then be formed by concatenating the "scope" of the request with the unique user id or IP address.

The allowed request rate is determined by the DEFAULT_THROTTLE_RATES setting using a key from the request "scope".

Filtering

The simplest way to filter the queryset of any view that subclasses `GenericAPIView` is to override the `.get_queryset()` method.

Filtering against the current user

Generic Filtering

REST framework also includes support for generic filtering backends that allow you to easily construct complex searches and filters.

DjangoFilterBackend

The django-filter library includes a DjangoFilterBackend class which supports highly customizable field filtering for REST framework.

To use DjangoFilterBackend, first install django-filter.

pip install django-filter

Then add 'django_filters' to Django's INSTALLED_APPS:

```
INSTALLED_APPS = [  
    'django_filters',  
]
```

<https://django-filter.readthedocs.io/en/latest/index.html>

Global Setting

Settings.py

```
REST_FRAMEWORK = {  
    'DEFAULT_FILTER_BACKENDS':  
    ['django_filters.rest_framework.DjangoFilterBackend']  
}
```

Per View Setting

You can set the filter backends on a per-view, or per-viewset basis, using the GenericAPIView class-based views.

```
from django_filters.rest_framework import DjangoFilterBackend  
class StudentListView(ListAPIView):  
    queryset = Student.objects.all()  
    serializer_class = StudentSerializer  
    filter_backends = [DjangoFilterBackend]
```


DjangoFilterBackend

If all you need is simple equality-based filtering, you can set a `filterset_fields` attribute on the view, or viewset, listing the set of fields you wish to filter against.

```
class StudentList(ListAPIView):  
    queryset = Student.objects.all()  
    serializer_class = StudentSerializer  
    filter_backends = [DjangoFilterBackend]  
    filterset_fields = ['name', 'city']
```

<http://127.0.0.1:8000/studentapi/?name=Sonam&city=Ranchi>

SearchFilter

The SearchFilter class supports simple single query parameter based searching, and is based on the Django admin's search functionality.

The SearchFilter class will only be applied if the view has a *search_fields* attribute set. The search_fields attribute should be a list of names of text type fields on the model, such as CharField or TextField.

SearchFilter

```
from rest_framework.filters import SearchFilter  
class StudentListView(ListAPIView):  
    queryset = Student.objects.all()  
    serializer_class = StudentSerializer  
    filter_backends = [SearchFilter]  
    search_fields = ['city']
```

<http://127.0.0.1:8000/studentapi/?search=Ranchi>

SearchFilter

- '^' Starts-with search.
- '=' Exact matches.
- '@' Full-text search. (Currently only supported Django's PostgreSQL backend.)
- '\$' Regex search.

Example:-

```
search_fields = ['^name',]
```

```
http://127.0.0.1:8000/studentapi/?search=r
```

OrderingFilter

The OrderingFilter class supports simple query parameter controlled ordering of results.

<http://127.0.0.1:8000/studentapi/?ordering=name>

The client may also specify reverse orderings by prefixing the field name with '-', like so:

<http://127.0.0.1:8000/studentapi/?ordering=-name>

Multiple orderings may also be specified:

<http://example.com/api/users?ordering=account,username>

OrderingFilter

It's recommended that you explicitly specify which fields the API should allow in the ordering filter. You can do this by setting an `ordering_fields` attribute on the view, like so:

```
class StudentListView(generics.ListAPIView):
```

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

```
    serializer_class = StudentSerializer
```

```
    filter_backends = [OrderingFilter]
```

```
    ordering_fields = ['name']
```

```
    ordering_fields = ['name', 'city']
```

```
    ordering_fields = '__all__'
```

Pagination

REST framework includes support for customizable pagination styles. This allows you to modify how large result sets are split into individual pages of data.

Pagination

- PageNumberPagination
- LimitOffsetPagination
- CursorPagination

Pagination Global Setting

The pagination style may be set globally, using the `DEFAULT_PAGINATION_CLASS` and `PAGE_SIZE` setting keys.

```
REST_FRAMEWORK = {  
    'DEFAULT_PAGINATION_CLASS': 'rest_framework.pagination.  
    PageNumberPagination',  
    'PAGE_SIZE': 5  
}
```

Pagination Per View

You can set the pagination class on an individual view by using the *pagination_class* attribute.

```
class StudentList(ListAPIView):  
    queryset = Student.objects.all()  
    serializer_class = StudentSerializer  
    pagination_class = PageNumberPagination
```

PageNumberPagination

This pagination style accepts a single number page number in the request query parameters.

To enable the PageNumberPagination style globally, use the following configuration, and set the PAGE_SIZE as desired:

```
REST_FRAMEWORK = {  
    'DEFAULT_PAGINATION_CLASS':  
    'rest_framework.pagination.PageNumberPagination',  
    'PAGE_SIZE': 5  
}
```

<http://127.0.0.1:8000/studentapi/?page=3>

PageNumberPagination

The PageNumberPagination class includes a number of attributes that may be overridden to modify the pagination style.

To set these attributes you should override the PageNumberPagination class, and then enable your custom pagination class.

- `django_paginator_class` - The Django Paginator class to use. Default is `django.core.paginator.Paginator`, which should be fine for most use cases.
- `page_size` - A numeric value indicating the page size. If set, this overrides the `PAGE_SIZE` setting. Defaults to the same value as the `PAGE_SIZE` settings key.
- `page_query_param` - A string value indicating the name of the query parameter to use for the pagination control.
- `page_size_query_param` - If set, this is a string value indicating the name of a query parameter that allows the client to set the page size on a per-request basis. Defaults to `None`, indicating that the client may not control the requested page size.

PageNumberPagination

- `max_page_size` - If set, this is a numeric value indicating the maximum allowable requested page size. This attribute is only valid if `page_size_query_param` is also set.
- `last_page_strings` - A list or tuple of string values indicating values that may be used with the `page_query_param` to request the final page in the set. Defaults to ('last',)
- `template` - The name of a template to use when rendering pagination controls in the browsable API. May be overridden to modify the rendering style, or set to `None` to disable HTML pagination controls completely. Defaults to `"rest_framework/pagination/numbers.html"`.

PageNumberPagination

```
class MyPageNumberPagination(PageNumberPagination):
```

```
    page_size = 5
```

```
    page_size_query_param = 'records'
```

```
    max_page_size = 7
```

```
class StudentList(ListAPIView):
```

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

```
    serializer_class = StudentSerializer
```

```
    pagination_class = MyPageNumberPagination
```

LimitOffsetPagination

This pagination style mirrors the syntax used when looking up multiple database records. The client includes both a "limit" and an "offset" query parameter. The limit indicates the maximum number of items to return, and is equivalent to the `page_size` in other styles. The offset indicates the starting position of the query in relation to the complete set of unpaginated items.

To enable the `LimitOffsetPagination` style globally, use the following configuration:

```
REST_FRAMEWORK = {  
    'DEFAULT_PAGINATION_CLASS':  
    'rest_framework.pagination.LimitOffsetPagination'  
}
```

<http://127.0.0.1:8000/studentapi/?limit=4&offset=6>

LimitOffsetPagination

The LimitOffsetPagination class includes a number of attributes that may be overridden to modify the pagination style.

To set these attributes you should override the LimitOffsetPagination class, and then enable your custom pagination class.

- `default_limit` - A numeric value indicating the limit to use if one is not provided by the client in a query parameter. Defaults to the same value as the `PAGE_SIZE` settings key.
- `limit_query_param` - A string value indicating the name of the "limit" query parameter. Defaults to 'limit'.
- `offset_query_param` - A string value indicating the name of the "offset" query parameter. Defaults to 'offset'.

LimitOffsetPagination

- `max_limit` - If set this is a numeric value indicating the maximum allowable limit that may be requested by the client. Defaults to `None`.
- `template` - The name of a template to use when rendering pagination controls in the browsable API. May be overridden to modify the rendering style, or set to `None` to disable HTML pagination controls completely. Defaults to `"rest_framework/pagination/numbers.html"`.

CursorPagination

The cursor-based pagination presents an opaque "cursor" indicator that the client may use to page through the result set. This pagination style only presents forward and reverse controls, and does not allow the client to navigate to arbitrary positions.

Cursor based pagination requires that there is a unique, unchanging ordering of items in the result set. This ordering might typically be a creation timestamp on the records, as this presents a consistent ordering to paginate against.

The default is to order by "-created". This assumes that there must be a 'created' timestamp field on the model instances, and will present a "timeline" style paginated view, with the most recently added items first.

CursorPagination

The CursorPagination class includes a number of attributes that may be overridden to modify the pagination style.

To set these attributes you should override the CursorPagination class, and then enable your custom pagination class.

- `page_size` = A numeric value indicating the page size. If set, this overrides the `PAGE_SIZE` setting. Defaults to the same value as the `PAGE_SIZE` settings key.
- `cursor_query_param` = A string value indicating the name of the "cursor" query parameter. Defaults to 'cursor'.
- `ordering` = This should be a string, or list of strings, indicating the field against which the cursor based pagination will be applied. For example: `ordering = 'slug'`. Defaults to `-created`. This value may also be overridden by using `OrderingFilter` on the view.

CursorPagination

template = The name of a template to use when rendering pagination controls in the browsable API. May be overridden to modify the rendering style, or set to None to disable HTML pagination controls completely. Defaults to "rest_framework/pagination/previous_and_next.html".

HyperlinkedModelSerializer

The HyperlinkedModelSerializer class is similar to the ModelSerializer class except that it uses hyperlinks to represent relationships, rather than primary keys.

By default the serializer will include a url field instead of a primary key field.

The url field will be represented using a HyperlinkedIdentityField serializer field, and any relationships on the model will be represented using a HyperlinkedRelatedField serializer field.

Serializer Relations

When using the ModelSerializer class, serializer fields and relationships will be automatically generated for you.

`repr(serializer)` – This shows automatically generated fields and relations.

Serializer Relations

StringRelatedField - StringRelatedField may be used to represent the target of the relationship using its `__str__` method. This field is read only.

Arguments:

many - If applied to a to-many relationship, you should set this argument to True.

Serializer Relations

PrimaryKeyRelatedField - PrimaryKeyRelatedField may be used to represent the target of the relationship using its primary key.

By default this field is read-write, although you can change this behavior using the `read_only` flag.

Arguments:

`queryset` - The queryset used for model instance lookups when validating the field input. Relationships must either set a queryset explicitly, or set `read_only=True`.

`many` - If applied to a to-many relationship, you should set this argument to `True`.

`allow_null` - If set to `True`, the field will accept values of `None` or the empty string for nullable relationships. Defaults to `False`.

`pk_field` - Set to a field to control serialization/deserialization of the primary key's value. For example, `pk_field=UUIDField(format='hex')` would serialize a UUID primary key into its compact hex representation.

Serializer Relations

HyperlinkedRelatedField - HyperlinkedRelatedField may be used to represent the target of the relationship using a hyperlink. By default this field is read-write, although you can change this behavior using the read_only flag.

Arguments:

view_name - The view name that should be used as the target of the relationship. If you're using the standard router classes this will be a string with the format <modelname>-detail. required.

queryset - The queryset used for model instance lookups when validating the field input. Relationships must either set a queryset explicitly, or set read_only=True.

many - If applied to a to-many relationship, you should set this argument to True.

allow_null - If set to True, the field will accept values of None or the empty string for nullable relationships. Defaults to False.

Serializer Relations

`lookup_field` - The field on the target that should be used for the lookup. Should correspond to a URL keyword argument on the referenced view. Default is 'pk'.

`lookup_url_kwarg` - The name of the keyword argument defined in the URL conf that corresponds to the lookup field. Defaults to using the same value as `lookup_field`.

`format` - If using format suffixes, hyperlinked fields will use the same format suffix for the target unless overridden by using the `format` argument.

Serializer Relations

SlugRelatedField - SlugRelatedField may be used to represent the target of the relationship using a field on the target.

By default this field is read-write, although you can change this behavior using the `read_only` flag.

When using SlugRelatedField as a read-write field, you will normally want to ensure that the slug field corresponds to a model field with `unique=True`.

Arguments:

`slug_field` - The field on the target that should be used to represent it. This should be a field that uniquely identifies any given instance. For example, `username`. `required`

`queryset` - The queryset used for model instance lookups when validating the field input. Relationships must either set a queryset explicitly, or set `read_only=True`.

`many` - If applied to a to-many relationship, you should set this argument to `True`.

`allow_null` - If set to `True`, the field will accept values of `None` or the empty string for nullable relationships. Defaults to `False`.

Serializer Relations

HyperlinkedIdentityField - This field can be applied as an identity relationship, such as the 'url' field on a HyperlinkedModelSerializer. It can also be used for an attribute on the object. This field is always read-only.

Arguments:

view_name - The view name that should be used as the target of the relationship. If you're using the standard router classes this will be a string with the format <model_name>-detail. required.

lookup_field - The field on the target that should be used for the lookup. Should correspond to a URL keyword argument on the referenced view. Default is 'pk'.

lookup_url_kwarg - The name of the keyword argument defined in the URL conf that corresponds to the lookup field. Defaults to using the same value as lookup_field.

format - If using format suffixes, hyperlinked fields will use the same format suffix for the target unless overridden by using the format argument.

Nested relationships

As opposed to previously discussed references to another entity, the referred entity can instead also be embedded or nested in the representation of the object that refers to it. Such nested relationships can be expressed by using serializers as fields.

If the field is used to represent a to-many relationship, you should add the `many=True` flag to the serializer field.

What Next ?

- Update Yourself by reading Django Doc and DRF Doc
- Build Applications
- Apply for Job or Internship
- Try to use various packages with Django - www.pypi.org
- Learn other Web App Framework/Library – Flask, Web2Py, TurboGears
Pyramid