💛

Django

Tags

Django is a high-level web framework for building web applications. It is built on Python and follows the model-view-controller (MVC) architectural pattern. It is known for its robustness, scalability, and ease of use.

Django provides a number of features that make it easy to build web applications quickly and efficiently. These include:

- Object-relational mapper (ORM): Django's ORM allows you to interact with your database using Python code, rather than writing raw SQL. This makes it easy to work with different databases and to switch between them if needed.
- Automatic admin interface: Django provides an in-built admin interface that makes it easy to manage your data.
- Templating system: Django provides a powerful templating system that allows you to separate the presentation of your data from the logic of your application.
- URL routing: Django provides a powerful URL routing system that makes it easy to map URLs to views (Business Logic).

Django REST framework (DRF) is a powerful tool for building RESTful APIs on top of Django. It provides a number of features that make it easy to build APIs, including:

- Serialization: DRF provides a powerful serialization framework that allows you to easily convert between Python objects and JSON.
- Authentication and permissions: DRF provides a number of built-in authentication and permission classes that make it easy to control access to your API.
- Pagination: DRF provides built-in support for pagination, making it easy to handle large datasets.
- Filtering: DRF provides built-in support for filtering, making it easy to retrieve specific subsets of data.

In this tutorial, we will walk through the process of building a simple RESTful API using Django and DRF. Specifically, we will cover the following steps:

1. Installation of Django and DRF
2. Creating a new Django project
3. Creating a new Django app
4. Defining a model
5. Serializing the model
6. Creating views
7. URL routing
8. Testing the API

By following this tutorial, you will be able to create a simple RESTful API using Django and DRF, and be familiar with the concepts of Django and how it works

Installation

The first step in building a Django and DRF based API is to install both libraries. You can do this by running the following command in your command prompt or terminal:

```
# Install Django
pip install django
```

```
# Install Django Rest Framework (DRF)
pip install django djangorestframework
```

The above cmds will install the django libraries, now we need to start a new Django project with the help of `django-admin`

#TODO: Add DRF into installed app

`django-admin` is a command-line utility that comes with Django. It allows you to perform various tasks related to your Django projects, such as creating a new project, creating new apps, running management commands, and more.

A **`Django project`** is a collection of settings and configurations for a specific web application. It contains one or more apps, each of which can be used to build a specific functionality or feature of the web application. A project can be created using the **`django-admin startproject`** command, which creates a new directory with the basic file structure of a Django project.

Creating a new Django Project

Next, you will need to create a new Django app within your project. You can do this by running the following command:

```
python manage.py startapp appname
```

This command will create a new directory within your project with the name of your app and the basic file structure of a Django app.

File Structure of Django

The basic file structure of a Django app typically includes the following:

- **`models.py`**: This file defines the data models for the app, including the fields and behavior of the data. It is used to interact with the database using the built-in Django

ORM.

- `views.py`: This file defines the views for the app, which handle specific requests and return a response. A view is a Python function that defines how to handle a specific URL.
- `urls.py`: This file defines the URLs for the app, including the mapping between URLs and views. It is used to control how URLs are handled by the app.
- `forms.py`: This file defines the forms for the app, which are used to handle form submissions and validation.
- `admin.py`: This file defines the admin interface for the app, which is used to manage the data through the built-in Django admin interface.
- `migrations/`: This directory contains the files related to database migrations. It is used to keep track of changes to the models and apply them to the database.
- `templates/`: This directory contains the templates for the app, which define the presentation of the data.
- `static/`: This directory contains the static files for the app, such as CSS, JavaScript, and images.

It's worth mentioning that some apps may not require all of these files, for example if you're building an API you may not need the `templates/` and `forms.py` files. You can also include other files and directories as per your requirement.

The basic file structure of a Django app is essential to understand and use correctly because it gives a clear overview of how the different components of an app fit together and interact with each other.

Here as we are just building up the APIs so the files of our interest are `models.py` , `views.py` `admin.py` , `urls.py` and the `migrations/` directory.

Django App

A Django app is a self-contained module that can be reused in multiple projects. It is designed to contain all the necessary files and logic for a specific functionality or feature of a web application. An app typically includes models, views, and templates, and can also include other files such as forms, serializers, and custom management commands. An app can be created using the `python manage.py startapp` command.

the basic file structure of a Django app, including `models.py`, `views.py`, `urls.py`, and `migrations/` directory.

A Django project, on the other hand, is a collection of settings and configurations for a specific web application. It contains one or more apps, and it can also include other files such as the `manage.py` file, which is used to manage the project, and the `settings.py` file, which contains the settings for the project, including the database settings, installed apps, and other configurations. A project can be created using the `django-admin startproject` command.

In simple words, a `Django project` is a collection of apps and configurations that make up a web application, while a `Django app` is a self-contained module that can be reused in multiple projects and contains the code for a specific functionality or feature of the web application.

It's worth noting that while a project can have multiple apps, an app can only belong to one project. This allows you to create reusable apps that can be easily plugged into different projects, making development faster and more efficient.

Create a new Django app

1. Next, you will need to create a new Django app within your project. You can do this by running the following command:

```
python manage.py startapp appname
```

This command will create a new directory within your project with the name of your app and the basic file structure of a Django app.

2. Add the app to `INSTALLED_APPS` list In the `settings.py` file of your project . This tells Django to include the app in the project.

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

3. Create Models: Create the models for the app in the `models.py` file, these models will be used to create the database tables.
4. Create views: Create views for the app in the `views.py` file, views handle the request and return the response.
5. Create URL routing: Create URL routing for the app in the `urls.py` file, define the URLs that the app should handle and map them to the views.

Django Model

In Django, a model is a Python class that defines the fields and behavior of the data you will be storing. It is used to interact with the database using the built-in Object-relational mapper (ORM).

A model defines the structure of the data, including the fields and their types, as well as any constraints or validation rules. It also defines the behavior of the data, such as how it is stored and retrieved from the database, and any relationships it has with other models.

For example, you might have a model called `Book` that has fields for the title, author, and publication date, and methods for retrieving all books by a specific author or finding the average rating for all books.

Models are defined in the `models.py` file of an app, and Django automatically creates a database table for each model. The ORM then allows you to interact with the database using Python code, rather than writing raw SQL, which makes it easy to work with different databases and to switch between them if needed.

In summary, a model in Django is a Python class that defines the structure and behavior of the data you will be storing, and it's used to interact with the database using the built-in Object-relational mapper (ORM), it allows you to interact with the database using Python code, rather than writing raw SQL, making it easy to work with different databases and to switch between them if needed.

Now, We will define model of the API

```
from django.db import models

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

- You can define multiple models for your API as per your requirement.
- The Model class should be inherited from the `model.Model` class which comes out of the box of the Django Library.

In this example, the `Book` model has three fields: `title`, `author`, and `publication_date`. The `CharField` and `DateField` options are used to define the data type of the fields.

`CharField` is used for storing character data and takes an optional `max_length` parameter. `DateField` is used for storing date data, it doesn't have any specific parameter.

Django provides many other field options such as `IntegerField`, `FloatField`, `TextField`, `BooleanField`, `EmailField`, and more.

The `models.Model` class also provides a number of other options and methods that can be used to define the behavior of the model, such as:

- `objects` : The manager for the model, it's an instance of django.db.models.Manager.
- `Meta` : A class that allows you to set additional (meta) options on the model.
- `save()` : Saves the model instance to the database.
- `delete()` : Deletes the model instance from the database.

The `save()` `delete()` and `objects` methods can be used via the Django ORM

In summary, A model in Django is defined as a Python class that inherits from `django.db.models.Model`. The class defines the fields and behavior of the data you will be storing, where the fields are defined with the available options such as `CharField`, `IntegerField`, `DateField` and more. The class also provides a number of other options and methods that can be used to define the behavior of the model, such as `objects`, `Meta` and more.

Database Migration in Django

In Django, database migration refers to the process of creating, modifying, and deleting database tables and fields to match the models defined in your code.

Django uses a built-in database migration system called `django.db.migrations` which automatically generates the necessary SQL commands to create, modify, and delete database tables and fields based on the models defined in your code.

When you make changes to your models, such as adding, removing, or modifying fields, you need to create a new migration to reflect those changes in the database. You can create a new migration by running the `python manage.py makemigrations` command, which will generate a new migration file in the `migrations` directory of your app.

Once the migration file is created, you can apply the migration to the database by running the `python manage.py migrate` command. This command will execute the SQL commands in the migration file to create, modify, or delete the necessary tables and fields in the database.

It's worth noting that Django also keeps track of all the migrations that have been applied to the database, this allows you to roll back to a previous state of the database if needed.

In summary, database migration in Django is the process of creating, modifying, and deleting database tables and fields to match the models defined in your code. Django uses a built-in database migration system that automatically generates the necessary SQL commands based on the models defined in your code. You can create a new migration by running the `python manage.py makemigrations` command and apply the migration to the database by running the `python manage.py migrate` command.

Now run the below cmds in your terminal to migrate the DB for the Book model we have created above

```
# Create migration files
python manage.py makemigrations

# Execute SQL quries on DB
python manage.py migrate
```

Serialization

Serialization in Django Rest Framework (DRF) exists to handle the conversion of complex data types, such as Django models and querysets, into a format that can be easily rendered into JSON, XML or other content types. This is important because when building an API, it's common to want to return data from your views and API endpoints in a format that can be easily consumed by other applications.

JSON, XML, and other formats are commonly used to represent data in an API because they are lightweight, easy to parse, and widely supported. However, these formats cannot natively handle complex data types such as Django models and querysets. Serialization solves this problem by providing a way to convert these data types into a format that can be easily rendered into JSON, XML, or other formats.

Serialization also provides a way to validate the data that is being sent to the API, this is useful because it ensures that the data is in the correct format and meets certain constraints before it is processed. This can help to prevent errors and improve the overall reliability of the API.

In summary, Serialization in Django Rest Framework (DRF) exists to handle the conversion of complex data types, such as Django models and querysets, into a format that can be easily rendered into JSON, XML or other content types. It also provides a way to validate the data that is being sent to the API, this ensures that the data is in the correct format and meets certain constraints before it is processed, it makes the API more reliable and easy to consume

DRF provides a built-in serializer class that can be used to serialize and deserialize data. The serializer class defines the fields that should be included in the serialized data, as well as any validation rules that should be applied.

To create a serializer in Django Rest Framework (DRF), you would typically follow these steps:

1. Import the serializer class: In a new file called `serializers.py` in your app directory, import the serializer class from the `rest_framework` module.

```
from rest_framework import serializers
```

1. Create a new serializer class: Define a new class that inherits from the `serializers.Serializer` or `serializers.ModelSerializer` class.

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

1. Define the fields: Use class attributes to define the fields that should be included in the serialized data. You can use the built-in field classes such as `CharField`, `IntegerField`, and `DateField`, or you can create custom fields.

```
class BookSerializer(serializers.ModelSerializer):
    title = serializers.CharField(max_length=100)
    author = serializers.CharField(max_length=100)
    publication_date = serializers.DateField()
```

Views

A view in Django and Django Rest Framework (DRF) is a Python function or class that handles a specific request from a client and returns a response. The view is responsible for performing the logic and business logic of the application, such as handling data validation, authentication, and authorization.

In Django, views can be defined as Python functions or as class-based views. To define a view as a Python function, you would typically create a new file called `views.py` in your app directory and define a function that takes a request object as an input and returns a response.

In DRF, views are defined as class-based views that inherit from `APIView` class or other classes that inherit from it such as `ListAPIView`, `CreateAPIView` etc. These classes provide built-in functionality for handling common tasks such as data validation, authentication, and authorization. Or you can implement function based views. They are easy to start with.

Lets handle CRUD operations on a model using a serializer in Django Rest Framework (DRF). Here's an example of how you might implement CRUD operations on a `Book` model using function-based views and the `BookSerializer` serializer:

```
from rest_framework.decorators import api_view, permission_classes
from rest_framework.response import Response
from rest_framework import status
from .models import Book
from .serializers import BookSerializer

@api_view(['GET', 'POST'])
def book_list(request):
    """
    List all books or create a new book
    """
    if request.method == 'GET':
        books = Book.objects.all()
        serializer = BookSerializer(books, many=True)
        return Response(serializer.data)
    elif request.method == 'POST':
        serializer = BookSerializer(data=request.data)
        if serializer.is_valid():
            serializer.save()
            return Response(serializer.data, status=status.HTTP_201_CREATED)
        return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)

@api_view(['GET', 'PUT', 'DELETE'])
def book_detail(request, pk):
    """
    Retrieve, update or delete a book
    """
    try:
        book = Book.objects.get(pk=pk)
    except Book.DoesNotExist:
        return Response(status=status.HTTP_404_NOT_FOUND)

    if request.method == 'GET':
        serializer = BookSerializer(book)
        return Response(serializer.data)
    elif request.method == 'PUT':
        serializer = BookSerializer(book, data=request.data
```

The code above defines two function-based views, `book_list` and `book_detail`, to handle CRUD operations on a `Book` model using the `BookSerializer` serializer.

`book_list` function handles the `GET` and `POST` requests:

- `GET` request: The function retrieves all the books from the database using the `Book.objects.all()` method and serializes the data using the `BookSerializer`. The `many=True` argument is used to indicate that the serializer is handling multiple instances of the model. The serialized data is then returned in the response.
- `POST` request: The function receives the data from the request using the `request.data` attribute, and the data is passed to the serializer using the `BookSerializer(data=request.data)` constructor. The `is_valid()` method is used to

check if the data is valid. If the data is valid, the serializer's `save()` method is called to save the data to the database and the serialized data is returned in the response. If the data is invalid, the serializer's `errors` attribute is returned in the response.

`book_detail` function handles the `GET`, `PUT` and `DELETE` requests:

- `GET` request: The function retrieves a single book from the database using the `Book.objects.get(pk=pk)` method, where `pk` is the primary key of the book passed as a parameter in the URL. The book data is then serialized using the `BookSerializer` and the serialized data is returned in the response.
- `PUT` request: The function retrieves a single book from the database using the `Book.objects.get(pk=pk)` method and deserializes the data using the `BookSerializer(book, data=request.data)` constructor. The `is_valid()` method is used to check if the data is valid. If the data is valid, the serializer's `save()` method is called to save the data to the database and the serialized data is returned in the response. If the data is invalid, the serializer's `errors` attribute is returned in the response.
- `DELETE` request: The function retrieves a single book from the database using the `Book.objects.get(pk=pk)` method, and the `book.delete()` method is called to delete the book from the database. The response will have a status code of 204 (No Content) if the deletion was successful.

The `api_view` decorator is used to specify that these views should be handled as API views, and the `permission_classes` decorator is used to specify that these views should not require any permissions.

In summary, the code defines two function-based views, `book_list` and `book_detail`, that handle CRUD operations on a `Book` model using the `BookSerializer` serializer. `book_list` handles `GET` and `POST` requests, and `book_detail` handles `GET`, `PUT`, and `DELETE` requests. The views use the `api_view` and `permission_classes` decorators to specify that these views should be handled as API views and do not require any permissions.

URL Routing

In Django, URL routing is the process of mapping a URL pattern to a view function or class-based view. This is done by including a URL pattern in the `urls.py` file of your app that maps the request URL to the view function.

Here's an example of how you might map a URL pattern to a view function:

```
Copy code
from django.urls import path
from . import views

urlpatterns = [
    path('books/', views.book_list, name='book_list'),
    path('books/<int:pk>/', views.book_detail, name='book_detail'),
]
```

In this example, we are using the `path` function from `django.urls` to map URL patterns to view functions. The first argument of the `path` function is the URL pattern, and the second argument is the view function. The `name` argument is used to give the URL pattern a name, which can be used to reverse the URL later.

The `<int:pk>` in the path function is a URL parameter and in this case it represents the primary key of the book.

Once you've defined your URL patterns, you need to include them in your project's `urls.py` file so that they can be used by the Django URL dispatcher. You can do this by using the `include` function from `django.urls`:

```
from django.urls import path, include

urlpatterns = [
    path('api/v1/', include('books.urls')),
]
```

In summary, URL routing in Django is the process of mapping a URL pattern to a view function or class-based view. This is done by including a URL pattern in the `urls.py` file of your app that maps the request URL to the view function using the `path` function.

Now as your developement is complete you can start the Django server my running.

`python manage.py runserver` at the root of your project. After that your API will be assible via localhost:8080/api/v1/books

Task Manager Project

Now let's create backend APIs for a small Task Manager App in Django.

We will have a Model (Table in DB) for storing tasks and have to do CRUD operations on them.

This is a fairly small project and a good point to get started with Django and DRF.

1. Start by creating a new Django project and app for your task manager. In your project's root directory, run the command `django-admin startproject taskmanager`
2. Navigate inside the newly created taskmanager directory created by django-admin and create a new app for your project

```
# Navigate inside the directory
cd taskmanager

# Create a new Django App
python manage.py startapp tasks
```

Here's how the directory structure would look like after executing the above cmds

```
.
├── manage.py
├── taskmanager
│   ├── __init__.py
│   ├── asgi.py
│   ├── settings.py
│   ├── urls.py
│   └── wsgi.py
└── tasks
    ├── __init__.py
    ├── admin.py
    ├── apps.py
    ├── migrations
    │   └── __init__.py
    ├── models.py
    ├── tests.py
    └── views.py
```

And also register the app and `rest_framework` in the `INSTALLED_APPS` list which resides in `taskmanager/setting.py`. Also It would now look something like this

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'rest_framework',
    'tasks',
]
```

3. Now we have to create models for our `Task`

   inside `tasks/models.py` add the following

```
from django.db import models

class Task(models.Model):
    name = models.CharField(max_length=255)
    description = models.TextField()
    due_date = models.DateTimeField()
    completed = models.BooleanField(default=False)
    created_at = models.DateTimeField(auto_now_add=True)
```

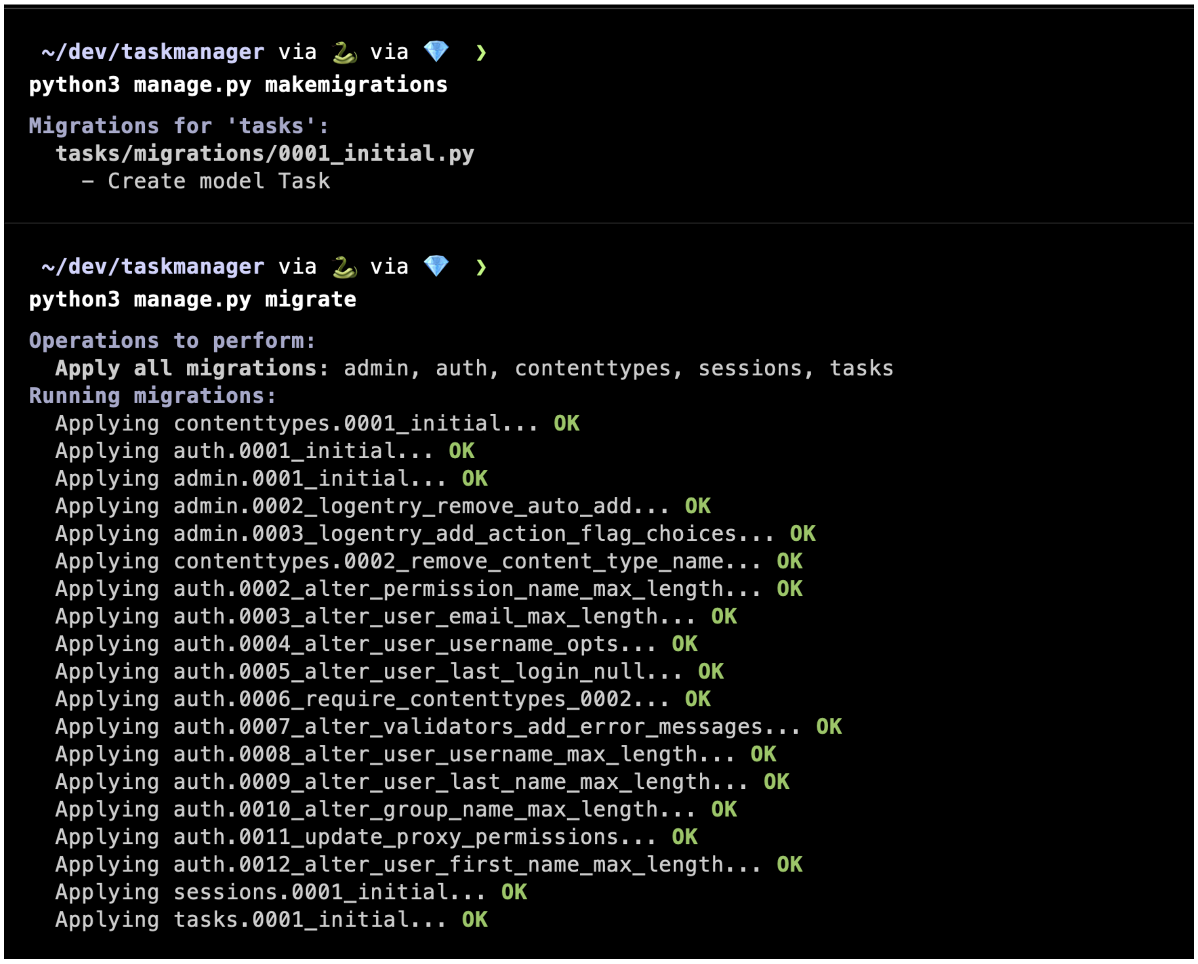
4. Now register the task model in `tasks/admin.py` file so that it is visible in the in-built Admin interface provided by Django

```
from django.contrib import admin

from .models import Task
admin.site.register(Task)
```

5. Now the Task model only exist as a python class in your code, there's no actual table in DB for it. To create one we have to run migrations against our DB. Django comes with an inbuilt configurations for a SQLite DB so we just have to run the cmds no worry about provisioning a DB.

```
# Create migration files
python3 manage.py makemigrations

# Run the migrations (Execute SQL quries against the DB)
python3 manage.py migrate
```

You should be able to see a output something similar below. Also you'll notice that a new `db.sqlite3` file is created in the root of your project. This is the default SQLite DB that Django offers.



```
 ~/dev/taskmanager via 🐍 via 💎  ❯
python3 manage.py makemigrations
Migrations for 'tasks':
  tasks/migrations/0001_initial.py
    - Create model Task

 ~/dev/taskmanager via 🐍 via 💎  ❯
python3 manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, sessions, tasks
Running migrations:
  Applying contenttypes.0001_initial... OK
  Applying auth.0001_initial... OK
  Applying admin.0001_initial... OK
  Applying admin.0002_logentry_remove_auto_add... OK
  Applying admin.0003_logentry_add_action_flag_choices... OK
  Applying contenttypes.0002_remove_content_type_name... OK
  Applying auth.0002_alter_permission_name_max_length... OK
  Applying auth.0003_alter_user_email_max_length... OK
  Applying auth.0004_alter_user_username_opts... OK
  Applying auth.0005_alter_user_last_login_null... OK
  Applying auth.0006_require_contenttypes_0002... OK
  Applying auth.0007_alter_validators_add_error_messages... OK
  Applying auth.0008_alter_user_username_max_length... OK
  Applying auth.0009_alter_user_last_name_max_length... OK
  Applying auth.0010_alter_group_name_max_length... OK
  Applying auth.0011_update_proxy_permissions... OK
  Applying auth.0012_alter_user_first_name_max_length... OK
  Applying sessions.0001_initial... OK
  Applying tasks.0001_initial... OK
```

6. Our next step is to create a serializer that would be used to validate the data in our models. create a new file `serializers.py` in the tasks app directory (`tasks/serializers.py`) and add the following.

```
# tasks/serializers.py

from rest_framework import serializers

class TaskSerializer(serializers.Serializer):
    id = serializers.IntegerField(read_only=True)
    name = serializers.CharField(max_length=255)
    description = serializers.CharField()
    due_date = serializers.DateTimeField()
    completed = serializers.BooleanField()
    created_at = serializers.DateTimeField(read_only=True)
```

the `id` and `created_at` fields as read_only as these fields are automatically generated.

We can add validation on different feilds as well. Below is an example of a serializer with validation on the `name` and `due_date` feilds

```
from rest_framework import serializers
from datetime import datetime

class TaskSerializer(serializers.Serializer):
    id = serializers.IntegerField(read_only=True)
    name = serializers.CharField(max_length=255)
    description = serializers.CharField()
    due_date = serializers.DateTimeField()
    completed = serializers.BooleanField()
    created_at = serializers.DateTimeField(read_only=True)

    def validate_name(self, value):
        if value == "":
            raise serializers.ValidationError("Name field is required")
        return value

    def validate_due_date(self, value):
        if value < datetime.now():
            raise serializers.ValidationError("Due date should be in the future")
        return value
```

We have defined two validation methods `validate_name` and `validate_due_date` to check if the `name` field is not empty and if the `due_date` is in the future. If these validations fail, the serializer will raise a `serializers.ValidationError` with the appropriate error message.

Any one of the above two serializer will do the Job.

7. Now it’s time to create Views for the API. Add the below code in `tasks/views.py` file. We will use class based views instead of function based views.

```
from rest_framework import views, status
from rest_framework.response import Response
from .models import Task
from .serializers import TaskSerializer

class TaskView(views.APIView):
    def get(self, request):
        tasks = Task.objects.all()
        serializer = TaskSerializer(tasks, many=True)
        return Response(serializer.data)

    def post(self, request):
        serializer = TaskSerializer(data=request.data)
        if serializer.is_valid():
            serializer.save()
            return Response(serializer.data, status=status.HTTP_201_CREATED)
        return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)

class TaskDetailView(views.APIView):
    def get_object(self, pk):
        try:
            return Task.objects.get(pk=pk)
        except Task.DoesNotExist:
            raise Http404

    def get(self, request, pk):
        task = self.get_object(pk)
        serializer = TaskSerializer(task)
        return Response(serializer.data)

    def put(self, request, pk):
        task = self.get_object(pk)
        serializer = TaskSerializer(task, data=request.data)
        if serializer.is_valid():
            serializer.save()
            return Response(serializer.data)
        return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)

    def delete(self, request, pk):
        task = self.get_object(pk)
        task.delete()
        return Response(status=status.HTTP_204_NO_CONTENT)
```

The `TaskView` class is a subclass of `views.APIView` and it handles the GET and POST operations of the API.

The `get` method retrieves a list of all tasks from the `Task` model using the `Task.objects.all()` query, and it then passes this queryset to the `TaskSerializer` to be serialized. The serialized data is then returned in the response using the `Response` class.

The `post` method is used to create a new task. It takes the data passed in the request and passes it to the `TaskSerializer` for validation. If the data is valid, the `save()` method is called on the serializer, creating a new task in the database. The serialized data is then returned in the response, along with a status code of 201 (HTTP_201_CREATED) to indicate that the task was successfully created. If the data is not valid, the serializer's `errors` field will contain the validation errors, and the response will include a status code of 400 (HTTP_400_BAD_REQUEST)

The `TaskDetailView` class is also a subclass of `views.APIView` and it handles the GET, PUT, and DELETE operations for a single task.

The `get_object` method is a helper method that retrieves a task from the database using the primary key passed in the URL. If the task does not exist, it raises a Http404 exception.

The `get` method retrieves a single task from the database using the `get_object` method, then it uses the `TaskSerializer` to serialize the data and returns the serialized data in the response.

The `put` method updates an existing task. It retrieves the task to be updated using the `get_object` method, then it takes the data passed in the request and passes

8. Last step is to add Routes for our API endpoint. It determines which code should be executed when an endpoint is called. Create a new file `urls.py` inside the `tasks` directory and add the below code

```
from django.urls import path
from .views import TaskView, TaskDetailView

urlpatterns = [
    path('tasks/', TaskView.as_view(), name='task-list'),
    path('tasks/<int:pk>/', TaskDetailView.as_view(), name='task-detail'),
]
```

And add this code inside the main `urls.py` file which is situated at `taskmanager/urls.py`

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

urlpatterns = [
    path('admin/', admin.site.urls),
    path('api/v1/', include('tasks.urls')),
]
```

Now we have completed developement of APIs, Just run `python3 manage.py runserver` inside the root directory, where `manage.py` file is situated and your API is is available at

http://127.0.0.1:8000/api/v1/tasks/

How to use Postgres Database with Django

Django comes up with an by default SQLite Database but you might want to use a Postgres Database instead as it is more robust solution and is being widely used in industry.

Here are the steps to be followed in order to use postgres database instead of SQLite database:

1. Install the **`psycopg2`** **library**

   Django and postgres requires psycopg2 library to function together. Install it in your environment via **`pip install psycopg2`**

2. Change configuration in your settings.py file

   inside your settings.py file change the `DATABASE` configurations to

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': 'your_db_name',
        'USER': 'your_db_user',
        'PASSWORD': 'your_password',
        'HOST': 'localhost',
        'PORT': '5432',
    }
}
```

Replace the above values with appropriate values for your setup.

3. Create the database: Run the following command to create the database:

   ```
   python manage.py migrate
   ```

You can now start using your PostgreSQL database in your Django application. To access the database, you can use the Django ORM (Object-Relational Mapping) or raw SQL queries.