Initial Setup & Running Project

docker compose run --rm django python manage.py makemigrations
docker compose run --rm django python manage.py migrate
docker compose up

Supported API Endpoints

List View for Movie
GET: localhost:8080/api/movies/ (returns all the movies)
POST: localhost:8080/api/movies/ (create a new movie)

Detail View for Movie
GET: localhost:8080/api/movies/1 (select movie with id = 1)
PUT: localhost:8080/api/movies/1 (update movie with id = 1)
PATCH: localhost:8080/api/movies/1 (partially update movie with id = 1)
DELETE: localhost:8080/api/movies/1 (delete movie with id = 1)

List View for Review
GET: localhost:8080/api/movies/review/ (returns all the reviews)
POST: localhost:8080/api/movies/review/ (create a new review)

Detail View for Review
GET: localhost:8080/api/movies/review/1 (select review with id = 1)
PUT: localhost:8080/api/movies/review/1 (update review with id = 1)
PATCH: localhost:8080/api/movies/review/1 (partially review with id = 1)
DELETE: localhost:8080/api/movies/ review/1 (delete review with id = 1)

Filter by Runtime
filter_type: exact | greater | less (by default: exact)
GET: localhost:8080/api/movies/?runtime=141
GET: localhost:8080/api/movies/?runtime=120&filter_type=greater
GET: localhost:8080/api/movies/?runtime=150&filter_type=less
GET: localhost:8080/api/movies/?runtime=141&filter_type=exact

Task 1:

Add Detail View for Movie to support following operations:

GET: localhost:8080/api/movies/1
PUT: localhost:8080/api/movies/1
PATCH: localhost:8080/api/movies/1
DELETE: localhost:8080/api/movies/1

Solution

In **coding_challenge/movies/views** created a new file **'movie_detail.py'**, added the following code:

```
from rest_framework.generics import RetrieveUpdateDestroyAPIView

from movies.models import Movie
from movies.serializers import MovieSerializer


class MovieDetailView(RetrieveUpdateDestroyAPIView):
    queryset = Movie.objects.all()
    serializer_class = MovieSerializer
```

Created **MovieDetailView** class which inherits from **RetrieveUpdateDestroyAPIView**. The **RetrieveUpdateDestroyAPIView** implements the ability to select a specific movie, update a movie, partially update the movie and delete the movie.

Add url to support **MovieDetailView:**

Update the **coding_challenge\movies\urls.py** to add **Detail** endpoint for **Movie**:

```
path("<int:pk>", MovieDetailView.as_view(), name="MovieDetailView"),
```

Task 2:

Add **runtime_formatted** in the API that returns the runtime as a string in the format **H:MM**. For example: if runtime is 120, the runtime_formatted should be 2:0

Solution

In **movies/serializers/movie_serializer.py**, made the following changes:

```
class MovieSerializer(serializers.ModelSerializer):
    runtime_formatted = serializers.SerializerMethodField()
    class Meta:
        model = Movie
        fields = (
            "id",
            "title",
            "runtime",
            "release_date",
            "Runtime_formatted",
        )
      def get_runtime_formatted(self, obj):
        hours, minutes = divmod(obj.runtime, 60)
        return f"{hours}:{minutes}"
```

The following code is saying that get the value of **runtime_formatted** field from **get_runtime_formatted** method:

runtime_formatted = serializers.SerializerMethodField()

get_runtime_formatted explanation:

- The get_runtime_formatted method takes the **obj** as argument. If you send a request with id = 1.
- It will select the movie with **id = 1** and **obj** is equal to that movie.
- From **obj**, we're selecting **runtime** (can be 120 etc) and passing it into **divmod function.**
-]**divmod function** splits the runtime into hours and minutes. Finally, returning the runtime_formatted value in **H:MM** format.

Task 3:

Add second Model for **Reviews** that is **many:one** related to **Movies**. At least add **name** and **rating** fields.

Solution:

In **coding_challenge/movies/models**, create **review.py** with following code:

```
from django.db import models
from django.core.validators import MinValueValidator, MaxValueValidator
from .movie import Movie

class Review(models.Model):
    movie = models.ForeignKey(Movie, related_name='reviews',
on_delete=models.CASCADE)
    name = models.CharField(max_length=100)
    rating = models.PositiveIntegerField(
        validators=[MinValueValidator(1), MaxValueValidator(5)]
    )
```

Explanation:

```
from django.db import models
```

Importing the models module which contains Model class

```
from django.core.validators import MinValueValidator, MaxValueValidator
```

Importing the **MinValueValidator** to set the minimum value of rating to be 1, and importing **MaxValueValidator** for maximum value of rating to be 5.

```
from .movie import Movie
```

Importing Movie model class to add the many to one relationship.

```
class Review(models.Model):
    movie = models.ForeignKey(Movie, related_name='reviews',
on_delete=models.CASCADE)
    name = models.CharField(max_length=100)
    rating = models.PositiveIntegerField(
```

```
        validators=[MinValueValidator(1), MaxValueValidator(5)]
    )
```

Created the **Review** model class with **foreign key movie, name** (person name), rating (between 1 to 5)

Setting the min and max rating to 1 and 5 using MinValueValidator and MaxValueValidator.

In **coding_challenge/movies/serializers**, create **review_serializer.py** with following code:

```
from rest_framework import serializers
from movies.models import Review, Movie

class ReviewSerializer(serializers.ModelSerializer):
    class Meta:
        model = Review
        fields = ('id', 'movie', 'name', 'rating')
```

This is implemented exactly like MovieSerializer only the name of class is different and the field names are different. This class is used for serializing (converting to JSON) the review objects.

Adding support for these operations:

List View for Review
GET: localhost:8080/api/movies/review/ (returns all the reviews)
POST: localhost:8080/api/movies/review/ (create a new review)

Create **review_list.py** inside **coding_challenge/movies/views** and add following code:

```
from rest_framework.generics import ListCreateAPIView

from movies.models import Review
from movies.serializers import ReviewSerializer


class ReviewListView(ListCreateAPIView):
    queryset = Review.objects.order_by("id")
    serializer_class = ReviewSerializer
```

This is similar to **MovieListView**, only the view class name is **ReviewListView** and the serializer class is **ReviewSerializer**. This view supports selecting all the reviews and inserting a new review for a movie.

Adding support for these operations:

Detail View for Review
GET: localhost:8080/api/movies/review/1 (select review with id = 1)
PUT: localhost:8080/api/movies/review/1 (update review with id = 1)
PATCH: localhost:8080/api/movies/review/1 (partially review with id = 1)
DELETE: localhost:8080/api/movies/ review/1 (delete review with id = 1)

Create **review_detail.py** inside **coding_challenge/movies/views** and add following code:

```
from rest_framework.generics import RetrieveUpdateDestroyAPIView

from movies.models import Review
from movies.serializers import ReviewSerializer


class ReviewDetailView(RetrieveUpdateDestroyAPIView):
    queryset = Review.objects.all()
    serializer_class = ReviewSerializer
```

This is same as **MovieDetailView.** Only the serializer class is different. We're using ReviewSerializer. This view supports:

- Select a specific review
- Update a review
- Partially update a review
- Delete a review

Add urls to support **ReviewListView** and **ReviewDetailView:**

Update the **coding_challenge\movies\urls.py** to add **List** and **Detail** endpoints for **Review**:

```
path("review/", ReviewListView.as_view(), name="ReviewListView"),
path("review/<int:pk>", ReviewDetailView.as_view(), name="ReviewDetailView"),
```

Task 4:

Add Reviews to both the **List** and **Detail Movie Views**. What it means is that right now, when we select a movie, it doesn't send back the reviewers for that movie. So, we have to integrate Review model with Movie to make sure the reviewers are also sent with a movie.

Solution:

Make the following changes in **coding_challenge/movies/serializers/movie_serializer.py**:

```
class MovieSerializer(serializers.ModelSerializer):
    runtime_formatted = serializers.SerializerMethodField()
    reviewers = serializers.SerializerMethodField()
    avg_rating = serializers.SerializerMethodField()
    class Meta:
        model = Movie
        fields = (
            "id",
            "title",
            "runtime",
            "release_date",
            "runtime_formatted",
            "reviewers",
            "avg_rating"
        )

    def get_runtime_formatted(self, obj):
        hours, minutes = divmod(obj.runtime, 60)
        return f"{hours}:{minutes}"

    def get_reviewers(self, obj):
        reviews = Review.objects.filter(movie=obj.id)
        serializer = ReviewSerializer(reviews, many=True)
        return serializer.data
```

Explanation:

1. Add `"reviewers"` in the fields list so that it returns the **reviewers** as well.
2. Add `reviewers = serializers.SerializerMethodField()` which means get the reviewers value from **get_reviewers** method.
3. Add **get_reviewers** method:

```
def get_reviewers(self, obj):
        reviews = Review.objects.filter(movie=obj.id)
        serializer = ReviewSerializer(reviews, many=True)
        return serializer.data
```

If the id = 1, it will select the movie with id = 1 which will be stored in **obj** parameter. Selecting all the reviews for that movie id. Using ReviewSerializer to convert the review objects to JSON. Return the JSON formatted reviewers list in the end.

So, the reviewers field will return all the reviews as a list when we select a movie.

Task 5:

Add a new field to the API: **avg_rating** that returns the **average rating** of a movie by all the **reviewers**

Solution:

Make the following changes in **coding_challenge/movies/serializers/movie_serializer.py**:

```
class MovieSerializer(serializers.ModelSerializer):
    runtime_formatted = serializers.SerializerMethodField()
    reviewers = serializers.SerializerMethodField()
    avg_rating = serializers.SerializerMethodField()
    class Meta:
        model = Movie
        fields = (
            "id",
            "title",
            "runtime",
            "release_date",
            "runtime_formatted",
            "reviewers",
            "avg_rating"
        )

    def get_runtime_formatted(self, obj):
        hours, minutes = divmod(obj.runtime, 60)
```

```
        return f"{hours}:{minutes}"

    def get_reviewers(self, obj):
        reviews = Review.objects.filter(movie=obj.id)
        serializer = ReviewSerializer(reviews, many=True)
        return serializer.data

    def get_avg_rating(self, obj):
        ratings = list(Review.objects.filter(movie=obj.id).values_list("rating", flat=True))
        if len(ratings) == 0:
            return None
        avg_rating = sum(ratings)/len(ratings)
        return round(avg_rating, 2)
```

Explanation:

4. Add `"avg_rating"` in the fields list so that it returns the **average rating** as well.
5. Add `avg_rating = serializers.SerializerMethodField()` which means get the average rating value from **get_avg_rating** method.
6. Add **get_avg_rating** method:

```
    def get_avg_rating(self, obj):
        ratings = list(Review.objects.filter(movie=obj.id).values_list("rating", flat=True))
        if len(ratings) == 0:
            return None
        avg_rating = sum(ratings)/len(ratings)
        return round(avg_rating, 2)
```

If the movie id = 1, it will select the movie with id = 1 and save it in **obj**.

The following line inside **get_avg_rating method** selects all the reviews with movie id = 1 and only get the rating values as a list.

```
ratings = list(Review.objects.filter(movie=obj.id).values_list("rating", flat=True))
```

The following code checks is there is at least one rating for the movie. If there are no ratings, we cannot calculate average rating, so it returns None. However, if there is at least one rating, it sums the ratings and divide by total no of ratings to calculate the average rating. Finally, it rounds off average rating by 2 decimal places and returns the avg_rating:

```
if len(ratings) == 0:
            return None
        avg_rating = sum(ratings)/len(ratings)
        return round(avg_rating, 2)
```

Task 6:

Add **Query Parameters** to the **List View** that allows **filtering** on the **movie runtime**. This will implement the following endpoints:

Filter by Runtime
filter_type: exact | greater | less (by default: exact)
GET: localhost:8080/api/movies/?runtime=141
GET: localhost:8080/api/movies/?runtime=120&filter_type=greater
GET: localhost:8080/api/movies/?runtime=150&filter_type=less
GET: localhost:8080/api/movies/?runtime=141&filter_type=exact

Solution:

Implementing filtering based on greater, less and exact matches for runtime.

Update **coding_challenge/movies/views\movie_list.py**:

```
from rest_framework.generics import ListCreateAPIView
from rest_framework.exceptions import ValidationError

from movies.models import Movie
from movies.serializers import MovieSerializer


class MovieListView(ListCreateAPIView):
    serializer_class = MovieSerializer

    def get_queryset(self):
        queryset = Movie.objects.order_by("id")
        runtime_filter = self.request.query_params.get('runtime', None)
        filter_type = self.request.query_params.get('filter_type', 'exact')
        if runtime_filter:
            try:
                runtime = int(runtime_filter)
                if filter_type == 'greater':
                    queryset = queryset.filter(runtime__gt=runtime)
                elif filter_type == 'less':
                    queryset = queryset.filter(runtime__lt=runtime)
                elif filter_type == 'exact':
                    queryset = queryset.filter(runtime=runtime)
                else:
                    # Handle invalid filter_type
                    raise ValidationError("Invalid filter type. Must be 'greater', 'less', or
'exact'.")
            except ValueError:
                # Handle invalid runtime_filter value
                raise ValidationError("Invalid runtime value. Must be an integer.")
        return queryset
```

The idea here that, if the url includes **runtime** and **filter_type** query parameters, the MovieListView should filter the movies based on that and return the movies.

The following code tries to select **runtime** and **filter_type** from the **url**. If the runtime is not present, it is set to **None.** If the filter_type is not given, it is set to **exact**:

```
runtime_filter = self.request.query_params.get('runtime', None)
filter_type = self.request.query_params.get('filter_type', 'exact')
```

Next, we check if the runtime_filter has a value. If it is None, it is going to skip the filtration based on runtime and return the movies. If there is a runtime_filter, based on filter_type, it filters the movies and returns the movies. If the url contains runtime which is not an integer or filter_type which is not valid, it throws an error as the response:

```
if runtime_filter:
            try:
                runtime = int(runtime_filter)
                if filter_type == 'greater':
                    queryset = queryset.filter(runtime__gt=runtime)
                elif filter_type == 'less':
                    queryset = queryset.filter(runtime__lt=runtime)
                elif filter_type == 'exact':
                    queryset = queryset.filter(runtime=runtime)
                else:
                    # Handle invalid filter_type
                    raise ValidationError("Invalid filter type. Must be 'greater', 'less', or
'exact'.")
            except ValueError:
                # Handle invalid runtime_filter value
                raise ValidationError("Invalid runtime value. Must be an integer.")
        return queryset
```

Also update the **__init__.py** files in **models, serializers,** and **views** to include all model classes, serializer classes and view classes. This will allow them to be importables in other files.

movies/models/__init__.py

```
from .movie import Movie
from .review import Review

__all__ = ["Movie", "Review"]
```

movies/views/__init__.py

```
from .movie_list import MovieListView
from .movie_detail import MovieDetailView
from .review_list import ReviewListView
from .review_detail import ReviewDetailView

__all__ = ["MovieListView", "MovieDetailView", "ReviewListView",
"ReviewDetailView"]
```

movies/serializers/__init__.py

```
from .movie_serializer import MovieSerializer
from .review_serializer import ReviewSerializer

__all__ = ["MovieSerializer", "ReviewSerializer"]
```