Question - 1

Spring Boot: Filter Microservice

Implement REST APIs to perform filter and sort operations on a collection of Products.

Each event is a JSON entry with the following keys:

- `barcode` : the unique id of the product (String)
- `price` : the price of the product (Integer)
- `discount` : the discount % available on the product(Integer)
- `available` : the availability status of the product (0 or 1)

Here is an example of a product JSON object:

```
[
    {
      "barcode": "74001755",
      "item": "Ball Gown",
      "category": "Full Body Outfits",
      "price": 3548,
      "discount": 7,
      "available": 1
    },
    {
      "barcode": "74002423",
      "item": "Shawl",
      "category": "Accessories",
      "price": 758,
      "discount": 12,
      "available": 1
    }
]
```

You are provided with the implementation of the models required for all the APIs. The task is to implement a set of REST services that exposes the endpoints and allows for filtering and sorting the collection of product records in the following ways:

`GET` request to `/filter/price/{initial_range}/{final_range}` :

- returns a collection of all products whose price is between the initial and the final range supplied
- The response code is 200, and the response body is an array of products in the price range provided.
- In case there are no such products return status code 400.

`GET` request to `/sort/price` :

- returns a collection of all products sorted by their pricing
- The response code is 200 and the response body is an array of the product names sorted in ascending order of price.

Complete the given project so that it passes all the test cases when running the provided unit tests.

▼ Example requests and responses

`GET` request to `/filter/price/{initial_range}/{final_range}`

The response code is 200, and when converted to JSON, the response body is as follows for filter/750/900:

```
[
  {
    "barCode": "74002423"
  }
]
```

`GET` request to `/sort/price`

The response code is 200 and the response body, when converted to JSON, is as follows:

```
[
  {
    "barCode": "74002423"
  },
  {
    "barCode": "74001755"
  }
]
```

Question - 2

Spring Boot: Github Events API

In this challenge, your task is to implement a simple REST API to manage a collection of GitHub events.

Each event is a JSON entry with the following keys:

- `id`: the unique ID of the event (Integer)
- `type`: the type of the event, written in PascalCase (String)
- `public`: whether the event is public, either true or false (Boolean)
- `repoId`: the ID of the repository the event belongs to (Integer)
- `actorId`: the ID of the user who created the event (Integer)

Here is an example of a trade JSON object:

```
{
      "type": "PushEvent",
      "public": true,
      "repoId": 1,
      "actorId": 1,
}
```

You are provided with the implementation of the Event model. The task is to implement a REST service that exposes the `/events` endpoint, which allows for managing the collection of events records in the following way:

`POST` request to `/events`:

- creates a new event
- expects a JSON event object without an id property as the body payload. You can assume that the given object is always valid.
- adds the given event object to the collection of events and assigns a unique integer id to it. The first created event must have id 1, the second one 2, and so on.
- you can assume that payload given to create the object is always valid.
- the response code is 201 and the response body is the created event object, including its id

`GET` request to `/events`:

- returns a collection of all events

- the response code is 200, and the response body is an array of all events ordered by their ids in increasing order

`GET` request to `/repos/{repoId}/events`:

- Returns a collection of events related to the given repository, ordered by their ids in increasing order.
- The response code is 200 if the repository exists, even if there are no events for that repository.
- The response code is 404 if the repository doesn't exist.

`GET` request to `/events/{eventId}`:

- returns an event with the given id
- if the matching event exists, the response code is 200 and the response body is the matching event object
- if there is no event in the collection with the given id, the response code is 404

You should complete the given project so that it passes all the test cases when running the provided *unit* tests. The project by default supports the use of the H2 database. Implement the POST request to `/events` first because testing the other methods requires POST to work correctly.

▼ Example requests and responses

`POST` request to `/events`

Request body:

```
{
          "type": "PushEvent",
          "public": True,
          "repoId": 1,
          "actorId": 1,
    }
```

The response code is 201, and when converted to JSON, the response body is:

```
{
        "id" : 1,
        "type": "PushEvent",
        "public": True,
        "repoId": 1,
        "actorId": 1,
  }
```

This adds a new object to the collection with the given properties and id 1.

`GET` request to `/events`

The response code is 200, and when converted to JSON, the response body (assuming that the below objects are all objects in the collection) is as follows:

```
[
        {
          "id": 1,
          "type": "PushEvent",
          "public": true,
          "repoId": 1,
          "actorId": 1
        },
        {
          "id": 2,
          "type": "ReleaseEvent",
          "public": true,
          "repoId": 1,
          "actorId": 1
        },
        {
          "id": 3,
          "type": "PushEvent",
          "public": true,
          "repoId": 2,
          "actorId": 1
        }
```

```
]
```

`GET` request to `/repos/1/events`

The response code is 200, and when converted to JSON, the response body (assuming that the below objects are all objects with *repoId* 1) is as follows:

```
[
        {
          "id": 1,
          "type": "PushEvent",
          "public": true,
          "repoId": 1,
          "actorId": 1
        },
        {
          "id": 2,
          "type": "ReleaseEvent",
          "public": true,
          "repoId": 1,
          "actorId": 1
        }
  ]
```

`GET` request to `/events/1`

Assuming that the object with id 1 exists, then the response code is 200 and the response body, when converted to JSON, is as follows:

```
{
    "id": 1,
    "type": "PushEvent",
    "public": true,
    "repoId": 1,
    "actorId": 1
  }
```

If an object with id 1 doesn't exist, then the response code is 404 and there are no particular requirements for the response body.

Question - 3

Spring Boot: Stock Trades API

In this challenge, your task is to implement a simple REST API to manage a collection of stock trades.

Each trade is a JSON entry with the following keys:

- `id`: The unique trade ID. (Integer)
- `type`: The trade type, either 'buy' or 'sell'. (String)
- `userId`: The unique user ID. (Integer):
- `symbol`: The stock symbol. (String)
- `shares`: The total number of shares traded. The traded shares value is between 10 and 30 shares, inclusive. (Integer)
- `price`: The price of one share of stock at the time of the trade. (Integer)
- `timestamp`: The epoch time of the stock trade in milliseconds. (Long)

Here is an example of a trade JSON object:

```
{
    "id":1,
    "type": "buy",
    "userId": 23,
    "symbol": "ABX",
```

```
    "shares": 30,
    "price": 134,
    "timestamp": 1531522701000
}
```

You are provided with the implementation of the Trade model. The task is to implement the REST service that exposes the `/trades` endpoint, which allows for managing the collection of trade records in the following way:

`POST` request to `/trades`:

- creates a new trade
- expects a JSON trade object without an id property as a body payload. You can assume that the given object is always valid.
- adds the given trade object to the collection of trades and assigns a unique integer id to it. The first created trade must have id 1, the second one 2, and so on.
- the response code is 201, and the response body is the created trade object

`GET` request to `/trades`:

- return a collection of all trades
- the response code is 200, and the response body is an array of all trade objects ordered by their ids in increasing order

`GET` request to `/trades/<id>`:

- returns a trade with the given id
- if the matching trade exists, the response code is 200 and the response body is the matching trade object
- if there is no trade with the given id in the collection, the response code is 404

`DELETE`, `PUT`, `PATCH` request to `/trades/<id>`:

- the response code is 405 because the API does not allow deleting or modifying trades for any id value

You should complete the given project so that it passes all the test cases when running the provided *unit* tests. The project by default supports the use of the H2 database.

▼ Example requests and responses

`POST` request to `/trades`

Request body:

```
{
   "type": "buy",
   "userId": 1,
   "symbol": "AC",
   "shares": 28,
   "price": 162,
   "timestamp" : 1591514264000
}
```

The response code is 201, and when converted to JSON, the response body is:

```
{
   "id": 1,
   "type": "buy",
   "userId": 1,
   "symbol": "AC",
   "shares": 28,
   "price": 162,
   "timestamp" : 1591514264000
}
```

This adds a new object to the collection with the given properties and id 1.

`GET` request to `/trades`

The response code is 200, and when converted to JSON, the response body (assuming that the below objects are all objects in the collection) is as follows:

```
[
   {
      "id": 1,
      "type": "buy",
      "userId": 1,
      "symbol": "AC",
      "shares": 28,
      "price": 162,
      "timestamp" : 1591514264000
   },
   {
      "id": 2,
      "type": "sell",
      "userId": 1,
      "symbol": "AC",
      "shares": 28,
      "price": 162,
      "timestamp" : 1591514264000
   }
]
```

`GET` request to `/trades/1`

Assuming that the object with id 1 exists, then the response code is 200 and the response body, when converted to JSON, is as follows:

```
{
   "id": 1,
   "type": "buy",
   "userId": 1,
   "symbol": "AC",
   "shares": 28,
   "price": 162,
   "timestamp" : 1591514264000
}
```

If an object with id 1 doesn't exist, then the response code is 404 and there are no particular requirements for the response body.

`DELETE` request to `/trades/1`

The response code is 405 and there are no particular requirements for the response body.

Question - 4

Spring Boot: Weather API

A team is building a travel company platform. One requirement is for a REST API service to provide weather information. Add functionality to add and delete the information as well as to perform some queries. It must handle typical information for weather data like latitude, longitude, temperature, etc.

The definitions and detailed requirements list follow. The submission is graded on whether the application performs data retrieval and manipulation based on given use cases exactly as described in the requirements.

Each weather data is a JSON object describing daily temperature recorded at a given location on a given date. Each such object has the following properties:

- `id`: the unique integer ID of the object
- `date`: the date, in `YYYY-MM-DD` format, denoting the date of the record
- `lat`: the latitude (up to 4 decimal places) of the location of the record
- `lon`: the longitude (up to 4 decimal places) of the location of the record
- `city`: the name of the city of the record
- `state`: the name of the state of the record
- `temperature`: a Double value, up to one decimal place, denoting the daily temperature of the record in Celsius

Here is an example of a weather data JSON object:

```
{
   "id": 1,
   "date": "1985-01-01",
   "lat": 36.1189,
   "lon": -86.6892,
   "city": "Nashville",
   "state": "Tennessee",
   "temperature": 17.3
}
```

The REST service must expose the `/weather` endpoint, which allows for managing the collection of weather records in the following way:

`POST` request to `/weather` :

- creates a new weather data record
- expects a valid weather data object as its body payload, except that it does not have an id property; assume that the given object is always valid
- adds the given object to the database and assigns a unique integer id to it
- the response code is 201, and the response body is the created record, including its unique id

`GET` request to `/weather`:

- the response code is 200
- the response body is an array of matching records, ordered by their ids in increasing order

`GET` request to `/weather/<id>`:

- returns a record with the given id
- if the matching record exists, the response code is 200 and the response body is the matching object
- if there is no record in the database with the given id, the response code is 404

`DELETE` request to `/weather/<id>`:

- deletes the record with the given id from the database
- if a matching record existed, the response code is 204
- if there was no record in the database with the given id, the response code is 404

Complete the project so that it passes all the test cases when running the provided *unit* tests. By default it supports the use of the H2 database. Implement the `POST` request to `/weather` first because testing the other methods requires `POST` to work correctly.

▼ Example requests and responses

`POST` request to `/weather`

Request body:

```
{
   "date": "2019-06-11",
   "lat": 41.8818,
   "lon": -87.6231,
   "city": "Chicago",
   "state": "Illinois",
   "temperature": 24.0
}
```

The response code is 201, and when converted to JSON, the response body is:

```
{
   "id": 1,
   "date": "2019-06-11",
   "lat": 41.8818,
   "lon": -87.6231,
   "city": "Chicago",
   "state": "Illinois",
```

```
    "temperature": 24.0
}
```

This adds a new object to the database with the given properties and id 1.

`GET` request to `/weather`

The response code is 200, and when converted to JSON, the response body (assuming that the below objects are all objects in the database) is as follows:

```
[
   {
      "id": 1,
      "date": "2019-06-11",
      "lat": 41.8818,
      "lon": -87.6231,
      "city": "Chicago",
      "state": "Illinois",
      "temperature": 24.0
   },
   {
      "id": 2,
      "date": "2019-06-12",
      "lat": 37.8043,
      "lon": -122.2711,
      "city": "Oakland",
      "state": "California",
      "temperature": 24.0
   }
]
```

`GET` request to `/weather/1`

Assuming that the object with id 1 exists, then the response code is 200 and the response body, when converted to JSON, is as follows:

```
{
   "id": 1,
   "date": "2019-06-11",
   "lat": 41.8818,
   "lon": -87.6231,
   "city": "Chicago",
   "state": "Illinois",
   "temperature": 24.0
}
```

When an object with id 1 doesn't exist, then the response code is 404 and there are no particular requirements for the response body.

`DELETE` request to `/weather/1`

Assuming that the object with id 1 exists, then the response code is 204 and there are no particular requirements for the response body. This causes the object with id 1 to be removed from the database.

When an object with id 1 doesn't exist, then the response code is 404 and there are no particular requirements for the response body.

Question - 1

Spring Boot: Filter Microservice

Implement REST APIs to perform filter and sort operations on a collection of Products.

Each event is a JSON entry with the following keys:

- `barcode` : the unique id of the product (String)
- `price` : the price of the product (Integer)
- `discount` : the discount % available on the product(Integer)
- `available` : the availability status of the product (0 or 1)

Here is an example of a product JSON object:

```
[
    {
      "barcode": "74001755",
      "item": "Ball Gown",
      "category": "Full Body Outfits",
      "price": 3548,
      "discount": 7,
      "available": 1
    },
    {
      "barcode": "74002423",
      "item": "Shawl",
      "category": "Accessories",
      "price": 758,
      "discount": 12,
      "available": 1
    }
]
```

You are provided with the implementation of the models required for all the APIs. The task is to implement a set of REST services that exposes the endpoints and allows for filtering and sorting the collection of product records in the following ways:

`GET` request to `/filter/price/{initial_range}/{final_range}` :

- returns a collection of all products whose price is between the initial and the final range supplied
- The response code is 200, and the response body is an array of products in the price range provided.
- In case there are no such products return status code 400.

`GET` request to `/sort/price` :

- returns a collection of all products sorted by their pricing
- The response code is 200 and the response body is an array of the product names sorted in ascending order of price.

Complete the given project so that it passes all the test cases when running the provided unit tests.

▼ Example requests and responses

`GET` request to `/filter/price/{initial_range}/{final_range}`

The response code is 200, and when converted to JSON, the response body is as follows for filter/750/900:

```
[
  {
    "barCode": "74002423"
  }
]
```

`GET` request to `/sort/price`

The response code is 200 and the response body, when converted to JSON, is as follows:

```
[
  {
    "barCode": "74002423"
  },
  {
    "barCode": "74001755"
  }
]
```

Question - 2

Spring Boot: Github Events API

In this challenge, your task is to implement a simple REST API to manage a collection of GitHub events.

Each event is a JSON entry with the following keys:

- `id`: the unique ID of the event (Integer)
- `type`: the type of the event, written in PascalCase (String)
- `public`: whether the event is public, either true or false (Boolean)
- `repoId`: the ID of the repository the event belongs to (Integer)
- `actorId`: the ID of the user who created the event (Integer)

Here is an example of a trade JSON object:

```
{
      "type": "PushEvent",
      "public": true,
      "repoId": 1,
      "actorId": 1,
}
```

You are provided with the implementation of the Event model. The task is to implement a REST service that exposes the `/events` endpoint, which allows for managing the collection of events records in the following way:

`POST` request to `/events`:

- creates a new event
- expects a JSON event object without an id property as the body payload. You can assume that the given object is always valid.
- adds the given event object to the collection of events and assigns a unique integer id to it. The first created event must have id 1, the second one 2, and so on.
- you can assume that payload given to create the object is always valid.
- the response code is 201 and the response body is the created event object, including its id

`GET` request to `/events`:

- returns a collection of all events

- the response code is 200, and the response body is an array of all events ordered by their ids in increasing order

`GET` request to `/repos/{repoId}/events`:

- Returns a collection of events related to the given repository, ordered by their ids in increasing order.
- The response code is 200 if the repository exists, even if there are no events for that repository.
- The response code is 404 if the repository doesn't exist.

`GET` request to `/events/{eventId}`:

- returns an event with the given id
- if the matching event exists, the response code is 200 and the response body is the matching event object
- if there is no event in the collection with the given id, the response code is 404

You should complete the given project so that it passes all the test cases when running the provided *unit* tests. The project by default supports the use of the H2 database. Implement the POST request to `/events` first because testing the other methods requires POST to work correctly.

▼ Example requests and responses

`POST` request to `/events`

Request body:

```
{
          "type": "PushEvent",
          "public": True,
          "repoId": 1,
          "actorId": 1,
    }
```

The response code is 201, and when converted to JSON, the response body is:

```
{
        "id" : 1,
        "type": "PushEvent",
        "public": True,
        "repoId": 1,
        "actorId": 1,
  }
```

This adds a new object to the collection with the given properties and id 1.

`GET` request to `/events`

The response code is 200, and when converted to JSON, the response body (assuming that the below objects are all objects in the collection) is as follows:

```
[
        {
          "id": 1,
          "type": "PushEvent",
          "public": true,
          "repoId": 1,
          "actorId": 1
        },
        {
          "id": 2,
          "type": "ReleaseEvent",
          "public": true,
          "repoId": 1,
          "actorId": 1
        },
        {
          "id": 3,
          "type": "PushEvent",
          "public": true,
          "repoId": 2,
          "actorId": 1
        }
```

```
]
```

`GET` request to `/repos/1/events`

The response code is 200, and when converted to JSON, the response body (assuming that the below objects are all objects with *repoId* 1) is as follows:

```
[
        {
          "id": 1,
          "type": "PushEvent",
          "public": true,
          "repoId": 1,
          "actorId": 1
        },
        {
          "id": 2,
          "type": "ReleaseEvent",
          "public": true,
          "repoId": 1,
          "actorId": 1
        }
  ]
```

`GET` request to `/events/1`

Assuming that the object with id 1 exists, then the response code is 200 and the response body, when converted to JSON, is as follows:

```
{
    "id": 1,
    "type": "PushEvent",
    "public": true,
    "repoId": 1,
    "actorId": 1
  }
```

If an object with id 1 doesn't exist, then the response code is 404 and there are no particular requirements for the response body.

Question - 3

Spring Boot: Stock Trades API

In this challenge, your task is to implement a simple REST API to manage a collection of stock trades.

Each trade is a JSON entry with the following keys:

- `id`: The unique trade ID. (Integer)
- `type`: The trade type, either 'buy' or 'sell'. (String)
- `userId`: The unique user ID. (Integer):
- `symbol`: The stock symbol. (String)
- `shares`: The total number of shares traded. The traded shares value is between 10 and 30 shares, inclusive. (Integer)
- `price`: The price of one share of stock at the time of the trade. (Integer)
- `timestamp`: The epoch time of the stock trade in milliseconds. (Long)

Here is an example of a trade JSON object:

```
{
    "id":1,
    "type": "buy",
    "userId": 23,
    "symbol": "ABX",
```

```
    "shares": 30,
    "price": 134,
    "timestamp": 1531522701000
}
```

You are provided with the implementation of the Trade model. The task is to implement the REST service that exposes the `/trades` endpoint, which allows for managing the collection of trade records in the following way:

`POST` request to `/trades`:

- creates a new trade
- expects a JSON trade object without an id property as a body payload. You can assume that the given object is always valid.
- adds the given trade object to the collection of trades and assigns a unique integer id to it. The first created trade must have id 1, the second one 2, and so on.
- the response code is 201, and the response body is the created trade object

`GET` request to `/trades`:

- return a collection of all trades
- the response code is 200, and the response body is an array of all trade objects ordered by their ids in increasing order

`GET` request to `/trades/<id>`:

- returns a trade with the given id
- if the matching trade exists, the response code is 200 and the response body is the matching trade object
- if there is no trade with the given id in the collection, the response code is 404

`DELETE`, `PUT`, `PATCH` request to `/trades/<id>`:

- the response code is 405 because the API does not allow deleting or modifying trades for any id value

You should complete the given project so that it passes all the test cases when running the provided *unit* tests. The project by default supports the use of the H2 database.

▼ Example requests and responses

`POST` request to `/trades`

Request body:

```
{
   "type": "buy",
   "userId": 1,
   "symbol": "AC",
   "shares": 28,
   "price": 162,
   "timestamp" : 1591514264000
}
```

The response code is 201, and when converted to JSON, the response body is:

```
{
   "id": 1,
   "type": "buy",
   "userId": 1,
   "symbol": "AC",
   "shares": 28,
   "price": 162,
   "timestamp" : 1591514264000
}
```

This adds a new object to the collection with the given properties and id 1.

`GET` request to `/trades`

The response code is 200, and when converted to JSON, the response body (assuming that the below objects are all objects in the collection) is as follows:

```
[
   {
      "id": 1,
      "type": "buy",
      "userId": 1,
      "symbol": "AC",
      "shares": 28,
      "price": 162,
      "timestamp" : 1591514264000
   },
   {
      "id": 2,
      "type": "sell",
      "userId": 1,
      "symbol": "AC",
      "shares": 28,
      "price": 162,
      "timestamp" : 1591514264000
   }
]
```

`GET` request to `/trades/1`

Assuming that the object with id 1 exists, then the response code is 200 and the response body, when converted to JSON, is as follows:

```
{
   "id": 1,
   "type": "buy",
   "userId": 1,
   "symbol": "AC",
   "shares": 28,
   "price": 162,
   "timestamp" : 1591514264000
}
```

If an object with id 1 doesn't exist, then the response code is 404 and there are no particular requirements for the response body.

`DELETE` request to `/trades/1`

The response code is 405 and there are no particular requirements for the response body.

Question - 4

Spring Boot: Weather API

A team is building a travel company platform. One requirement is for a REST API service to provide weather information. Add functionality to add and delete the information as well as to perform some queries. It must handle typical information for weather data like latitude, longitude, temperature, etc.

The definitions and detailed requirements list follow. The submission is graded on whether the application performs data retrieval and manipulation based on given use cases exactly as described in the requirements.

Each weather data is a JSON object describing daily temperature recorded at a given location on a given date. Each such object has the following properties:

- `id`: the unique integer ID of the object
- `date`: the date, in `YYYY-MM-DD` format, denoting the date of the record
- `lat`: the latitude (up to 4 decimal places) of the location of the record
- `lon`: the longitude (up to 4 decimal places) of the location of the record
- `city`: the name of the city of the record
- `state`: the name of the state of the record
- `temperature`: a Double value, up to one decimal place, denoting the daily temperature of the record in Celsius

Here is an example of a weather data JSON object:

```
{
   "id": 1,
   "date": "1985-01-01",
   "lat": 36.1189,
   "lon": -86.6892,
   "city": "Nashville",
   "state": "Tennessee",
   "temperature": 17.3
}
```

The REST service must expose the `/weather` endpoint, which allows for managing the collection of weather records in the following way:

`POST` request to `/weather` :

- creates a new weather data record
- expects a valid weather data object as its body payload, except that it does not have an id property; assume that the given object is always valid
- adds the given object to the database and assigns a unique integer id to it
- the response code is 201, and the response body is the created record, including its unique id

`GET` request to `/weather`:

- the response code is 200
- the response body is an array of matching records, ordered by their ids in increasing order

`GET` request to `/weather/<id>`:

- returns a record with the given id
- if the matching record exists, the response code is 200 and the response body is the matching object
- if there is no record in the database with the given id, the response code is 404

`DELETE` request to `/weather/<id>`:

- deletes the record with the given id from the database
- if a matching record existed, the response code is 204
- if there was no record in the database with the given id, the response code is 404

Complete the project so that it passes all the test cases when running the provided *unit* tests. By default it supports the use of the H2 database. Implement the `POST` request to `/weather` first because testing the other methods requires `POST` to work correctly.

▼ Example requests and responses

`POST` request to `/weather`

Request body:

```
{
   "date": "2019-06-11",
   "lat": 41.8818,
   "lon": -87.6231,
   "city": "Chicago",
   "state": "Illinois",
   "temperature": 24.0
}
```

The response code is 201, and when converted to JSON, the response body is:

```
{
   "id": 1,
   "date": "2019-06-11",
   "lat": 41.8818,
   "lon": -87.6231,
   "city": "Chicago",
   "state": "Illinois",
```

```
    "temperature": 24.0
}
```

This adds a new object to the database with the given properties and id 1.

`GET` request to `/weather`

The response code is 200, and when converted to JSON, the response body (assuming that the below objects are all objects in the database) is as follows:

```
[
   {
      "id": 1,
      "date": "2019-06-11",
      "lat": 41.8818,
      "lon": -87.6231,
      "city": "Chicago",
      "state": "Illinois",
      "temperature": 24.0
   },
   {
      "id": 2,
      "date": "2019-06-12",
      "lat": 37.8043,
      "lon": -122.2711,
      "city": "Oakland",
      "state": "California",
      "temperature": 24.0
   }
]
```

`GET` request to `/weather/1`

Assuming that the object with id 1 exists, then the response code is 200 and the response body, when converted to JSON, is as follows:

```
{
   "id": 1,
   "date": "2019-06-11",
   "lat": 41.8818,
   "lon": -87.6231,
   "city": "Chicago",
   "state": "Illinois",
   "temperature": 24.0
}
```

When an object with id 1 doesn't exist, then the response code is 404 and there are no particular requirements for the response body.

`DELETE` request to `/weather/1`

Assuming that the object with id 1 exists, then the response code is 204 and there are no particular requirements for the response body. This causes the object with id 1 to be removed from the database.

When an object with id 1 doesn't exist, then the response code is 404 and there are no particular requirements for the response body.

Question - 1

Springboot: Playlist Management APIs

Implement a simple REST API to manage a collection of music playlist data.

Each PlayList is a JSON entry with the following keys:

- id: the unique ID of the event (Long).
- name: name of the play list (String).
- tracksCount: initial number of tracks in the play list (Integer).

Here is an example of a PlayList JSON object:

```
{
	"name": "Henry1s list",
	"tracksCount": 10
}
```

An implementation of the PlayList model is provided. Implement a REST service that exposes the v1/playlists endpoints, which allows for managing the collection of PlayList records in the following way:

POST request to /v1/playlists:

- Create a new playlist data record.
- The response code is 201, and the response body is the created record, including its unique *id*.

GET request to /v1/playlists:

- The response code is 200.
- The response body is a list of matching records, ordered by their ids in increasing order.

GET request to /v1/playlists/{playListId}:

- Return a record with the given *id* and status code 200.
- If there is no record with the given *id*, the response code is null.

DELETE request to /v1/playlists/{playListId}:

- Delete the record with the given *id* and return status code 204.

Complete the project so that it passes all the test cases when running the provided unit tests. The project, by default, supports the use of the H2 database.

▼ Example requests and responses

`POST` request to /v1/playlists

Request body:

```
{
	"name": "Henry1s list",
	"tracksCount": 10
}
```

The response code is 201, and when converted to JSON, the response body is:

```
{
	"id": 1,
```

```
    "name": "Henry1s list",
    "tracksCount": 10
}
```

This adds a new object to the collection with the given properties and id 1.

`GET` request to /v1/playlists

The response code is 200, and when converted to JSON, the response body (assuming that the objects are all in the collection) is as follows:

```
[
   {
      "id":1,
      "name":"Henry1s list",
      "tracksCount":10
   },
   {
      "id":2,
      "name":"Bera",
      "tracksCount":10
   },
   {
      "id":3,
      "name":"Namal",
      "tracksCount":10
   }
]
```

`GET` request to /v1/playlists/1

The response code is 200, and when converted to JSON, the response body is as follows:

```
{
   "id":1,
   "name":"Henry1s list",
   "tracksCount":10
}
```

DELETE request to /v1/playlists/1

Assuming that the object with id 1 exists, the response code is 204, and the response body is empty.

Question - 2

Springboot: Track Information APIs

Develop a REST API that shares details about music tracks for a music streaming platform. This API can receive, show, and query data like the track title, album, description, and play count. The work is judged based on how accurately and efficiently the application handles and processes this data.

Music track data is represented as a JSON object, each describing various properties of a music track. The following properties are included:

- *id:* The unique integer ID of the object (Long).
- *title:* The title of the track (String).
- *albumName:* The corresponding album name of the track (String).
- *releaseDate:* The release date of the track, formatted as YYYY-MM-DD (Date).
- *playCount:* An integer value denoting how many times a user has played this track (Integer).

Example of a music track data JSON object:

```
{
  "title": "Lost in Echoes",
```

```
  "albumName": "Echoes of the Unknown",
  "releaseDate": "2021-07-15",
  "playCount": 5000
}
```

The REST service must expose the endpoint /music/platform/v1/tracks, enabling the management of music track records as follows:

POST request to /music/platform/v1/tracks:

- Creates a new music track record.
- It expects a valid music track data object as its body payload, excluding the id property.
- The service assigns a unique long id to the added object.
- The response includes the created record with its unique id, and the response code is 201.

GET request to /music/platform/v1/tracks:

- Responds with a list of all music track records and a response code of 200.

DELETE request to /music/platform/v1/tracks/{trackId}:

- Deletes the record with the specified track id if it exists in the database.

GET request to /music/platform/v1/tracks/sorted:

- Provides a list of music track records sorted by title.
- The response code is 200.
- The system defaults to sorting by title in ascending order.

Ensure the project meets all test case criteria when running the provided rspec tests. It uses the H2 database by default. Start by implementing the POST request to music/platform/v1/tracks, as testing other methods requires the POST functionality to work correctly.

▼ Example requests and responses

`POST` **request to /music/platform/v1/tracks**

Request body:

```
{
   "title":"Lost in Echoes",
   "albumName":"Echoes of the Unknown",
   "releaseDate":"2021-07-15",
   "playCount":5000
}
```

The response code is 201, and when converted to JSON, the response body is:

```
{
   "id":1,
   "title":"Lost in Echoes",
   "albumName":"Echoes of the Unknown",
   "releaseDate":"2021-07-15",
   "playCount":5000
}
```

This adds a new object to the collection with the given properties and id 1.

`GET` **request to /music/platform/v1/tracks**

The response code is 200, and when converted to JSON, the response body, assuming these objects are in the collection, is as follows:

```
[
   {
      "id":1,
      "title":"Lost in Echoes",
      "albumName":"Echoes of the Unknown",
      "releaseDate":"2021-07-15",
      "playCount":5000
```

```
    },
    {
        "id":2,
        "title":"Bravos",
        "albumName":"Unknown",
        "releaseDate":"2021-07-16",
        "playCount":100
    }
]
```

DELETE **request to /music/platform/v1/tracks/1**

Assuming that the object with id 1 exists, the response code is 204, and the response body is empty.

`GET` request to /music/platform/v1/tracks/sorted

The response code is 200, and when converted to JSON, the response body, the returned collection sorted by its title ascending order, is as follows:

```
[
    {
        "id":1,
        "title":"Abacus",
        "albumName":"Echoes of the Unknown",
        "releaseDate":"2021-07-15",
        "playCount":5000
    },
    {
        "id":2,
        "title":"Bravos",
        "albumName":"Unknown",
        "releaseDate":"2021-07-16",
        "playCount":100
    }
]
```

Question - 3

Spring Boot: Setting up the Trading Platform

A team is building a trading platform that allows users to be registered as traders. One requirement is to implement REST API service using the Spring Boot Framework to provide options to create an account, update account information, add money, etc. It must deal with the typical information for a trader: name, email, account balance, account creation time, and account update time.

Each trader record is a JSON object with the following keys:

- `id`: The unique ID assigned at the time of registration.
- `name`: The name of the trader.
- `email`: The email of the trader.
- `balance`: The account balance of the trader.
- `createdAt`: The timestamp when the registration was completed described by the string `yyyy-MM-dd HH:mm:ss`.
- `updatedAt`: The timestamp when the trader account got updated, i.e., either the name was updated or money was added described by the string `yyyy-MM-dd HH:mm:ss`.

Example of a trader JSON object:

```
{
    "id": 1,
    "name": "Elizabeth Small",
    "email": "susanchandler.wchurch@buck.com",
    "balance": 62.0,
```

```
    "createdAt": "2018-04-16 04:56:28",
    "updatedAt": ""
}
```

The `REST` service must expose the `/trading/traders` endpoint, which allows for managing the data records in the following way:

`POST` request to `/trading/traders/register`:

- Registers a new trader record.
- Expects a JSON trader object with missing `id`, `createdAt`, `updatedAt`. Assume that the given object is always valid.
- If a trader with the same email already exists, the response code is 400. Otherwise, the response code is 201.

`GET` request to `/trading/traders/all`:

- Returns all the records with status code 200.
- Records should be sorted by ID in ascending order.

`GET` request to `/trading/traders?email={email}`:

- Returns a record with the given email and status code 200.
- If there is no record in the database with the given email, the response code is 404.

`PUT` request to `/trading/traders`:

- Updates the trader's name by email. The trader JSON object sent in the request body will have the keys `email, name`.
- If the trader with the requested email does not exist, the response code is 404. Otherwise, the response code is 200.

`PUT` request to `/trading/traders/add`:

- Adds money to the trader's account by email. The trader JSON sent in the request body will have the keys `email, amount`.
- If the trader with the requested email does not exist, the response code is 404. Otherwise, the response code is 200.

Note that:

- The default timezone of the application is set to *UTC*, which should not be modified.
- We allow an absolute error of 10^{-3} in the expected and returned `balance` field of the trader data.
- We allow a maximum difference of one second between the returned `createdAt` value in any *GET* request and the expected `createdAt`, i.e., the timestamp when the registration request was sent from the JUnit test.
- We allow a maximum difference of one second between the returned `updatedAt` value in any *GET* request and the expected `updatedAt`, i.e., the timestamp when the update request was sent from the JUnit test.

Most of the implementation is provided, but the expected behavior is not achieved as there are some bugs in the given project. Find and fix the bugs in order to get the expected behavior which is validated by executing a set of JUnit tests. By default, the project supports the use of the H2 database.

▼ Example requests and responses

▼ POST to /trading/traders/register

Consider the following *POST* requests:

1.
```
{
   "name":"Elizabeth Small",
   "email":"susanchandler.wchurch@buck.com",
   "balance":62.0
}
```

2.
```
{
   "name":"Susan Adams",
   "email":"jeremyortega.smithpatricia@coleman.biz",
   "balance":67.0
}
```

It adds these records to database and returns status code 201.

▼ GET to /trading/traders?email=susanchandler.wchurch@buck.com

The response of the *GET* request is the following *JSON* with the *HTTP* response code *200*:

```
{
   "id":1,
   "name":"Elizabeth Small",
   "email":"susanchandler.wchurch@buck.com",
   "balance":62.0,
   "createdAt":"2018-04-16 05:46:18",
   "updatedAt":""
}
```


▼ PUT to /trading/traders

The following *JSON* is sent to update the name:


```
{
   "name":"Susan Wood",
   "email":"jeremyortega.smithpatricia@coleman.biz"
}
```


The name is successfully updated for the trader with the *HTTP* response code *200*.

▼ PUT to /trading/traders/add

The following *JSON* is sent to add money:


```
{
   "email":"mbooker.jacobsmith@hotmail.com",
   "amount":73.0
}
```


The amount is added to the trader's account with the *HTTP* response code *200*.

▼ GET to /trading/traders/all

The response of the *GET* request is the following *JSON* array with the *HTTP* response code *200*:


```
[
   {
      "id":2,
      "name":"Susan Wood",
      "email":"jeremyortega.smithpatricia@coleman.biz",
      "balance":67.0,
      "created_at":"2018-04-16 05:46:27",
      "updated_at":"2018-04-16 05:49:18"
   },
   {
      "id":3,
      "name":"Rebecca Carter",
      "email":"adkinsjason.paul57@collins.com",
      "balance":151.0,
      "created_at":"2018-04-16 05:46:34",
      "updated_at":"2018-04-16 05:53:40"
   }
]
```


Question - 4

Spring Boot: Rest Controller Advice Setup

The `@ControllerAdvice` annotation in Spring Boot is used as an interceptor to handle exceptions thrown from the controller's `@RequestMapping` methods. It acts as a global exception-handling component for the entire application. This question is about setting up one such controller advice component using the `@ControllerAdvice` and `@ExceptionHandler` annotations.

The definitions and detailed requirements are listed as follows.

Given a rest controller class `FizzBuzzController.java`, there is a `GET` API endpoint to be modified, which can throw three types of runtime exceptions:

1. FizzException
2. BuzzException
3. FizzBuzzException

Below is the endpoint for the `GET` API with a single path parameter:

`GET /controller_advice/{code}`
Exceptions are to be thrown based on the value of the path param {code} passed to the REST API.

Here is a series of requests and their corresponding expected responses:

`GET` request to `/controller_advice/fizz`:

- Should trigger the `FizzException`
- Should return the response code 500 and the `GlobalError` as the response body

`GET` request to `/controller_advice/buzz`:

- Should trigger the `BuzzException`
- Should return the response code 400 and the `GlobalError` as the response body

`GET` request to `/controller_advice/fizzbuzz`:

- Should trigger the `FizzBuzzException`
- Should return the response code 507 and the `GlobalError` as the response body

`GET` request to `/controller_advice/success`:

- Should not trigger any exceptions
- Should return the response code 200 and the `FizzBuzzResponse` as the response body

Complete the project so that it passes all the test cases when running the provided unit tests. To do so, implement a controller advice global exception handler `FizzBuzzExceptionHandler`, intercept the runtime exceptions thrown from the `GET` API endpoint, and return a response entity wrapped in the `GlobalError` class.

The runtime exceptions and error response classes are already predefined in the project. Complete the implementation controller `GET` API endpoint and the controller advice classes.

▼ Example requests and responses

`GET` request to `/controller_advice/fizz`
The response code is 500, and when converted to JSON, the response body is:

```
{
    "message": "Fizz Exception has been thrown",
    "errorReason" : "Internal Server Error"
}
```

`GET` request to `/controller_advice/buzz`
The response code is 400, and when converted to JSON, the response body is:

```
{
    "message": "Buzz Exception has been thrown",
    "errorReason" : "Bad Request"
}
```

`GET` request to `/controller_advice/fizzbuzz`

The response code is 507, and when converted to JSON, the response body is:

```
{
    "message": "FizzBuzz Exception has been thrown",
    "errorReason" : "Insufficient Storage"
}
```

`GET` request to `/controller_advice/success`

The response code is 200, and when converted to JSON, the response body is:

```
{
    "message": "Successfully completed fizzbuzz test",
    "statusCode": "200"
}
```

Question - 5

Spring Boot: Custom Request Validator

There is a tiny `employee information system` built using Spring Boot. Currently, the system accepts employee information as Employee objects from the client requests, but it lacks the validation of the Employee object attributes that checks for correctness. The application owner doesn't want invalid data to be saved in the system, meaning the employee information should be validated according to the given constraints. The REST API request is validated by using the `@Valid` request Bean Validation or programmatically by implementing a custom request validator. This question deals with setting up a custom request validator in order to validate a RESTful API request.

The given project has all the classes defined for accepting requests from clients and saving employee information into the database. It also uses a custom request validator `EmployeeValidator` to validate each `Employee` request object before saving it into the database. However, the validator currently does nothing, meaning it doesn't report any errors upon receiving invalid data. Your task is to complete the implementation of this custom request validator class `EmployeeValidator` so that it reports errors.

The validation constraints are given below:

1. `fullName:`
   - validation: check if it's null or empty
   - message: The fullName is a mandatory field
2. `mobileNumber:`
   - validation: check if it's null or not of 10 digits
   - message: The mobileNumber is a mandatory field
3. `emailId:`
   - validation: check if it's null or empty
   - message: The emailId is a mandatory field
   - validation: check if it doesn't contain the @ sign
   - message: The emailId should be in a valid email format
4. `emailId:`
   - validation: check if it's null or empty
   - message: The dateOfBirth is a mandatory field
   - validation: check if it's not in YYYY-MM-DD format
   - message: The dateOfBirth should be in YYYY-MM-DD format

NOTE: All the validation error messages must be reported in the same order as the corresponding field declaration order.

Here is an example of a valid employee data JSON object:

```
{
  "fullName": "Foo Bar",
  "mobileNumber": 9837465792,
  "emailId": "test@gmail.com",
  "dateOfBirth": "1990-01-01"
}
```

There is a single REST API endpoint exposed for receiving employee information.

`POST` request to `/employee`:

- accepts an `Employee` object as a request body
- calls employee validator
- if there are no errors, then it returns status code 200
- if there are any errors, then `EmployeeValidationErrorHandler` returns a list of validation error messages `List<FieldValidationMessage>` with status code 400

Your task is to complete the given project so that it passes all the test cases when running the provided *unit* tests.

▼ Example requests and responses

`POST` request to `/employee`

Request body:

```
{
  "fullName": null,
  "mobileNumber": 0,
  "emailId": null,
  "dateOfBirth": null
}
```

The response code is 400, and when converted to JSON, the response body is:

```
[
  {
    "message": "The fullName is a mandatory field"
  },
  {
    "message": "The mobileNumber is a mandatory field"
  },
  {
    "message": "The emailId is a mandatory field"
  },
  {
    "message": "The dateOfBirth is a mandatory field"
  }
]
```

`POST` request to `/employee`

Request body:

```
{
  "fullName": "",
  "mobileNumber": 0,
  "emailId": "",
  "dateOfBirth": ""
}
```

The response code is 400, and when converted to JSON, the response body is:

```
[
  {
    "message": "The fullName is a mandatory field"
  },
  {
    "message": "The mobileNumber is a mandatory field"
  },
  {
    "message": "The emailId is a mandatory field"
  },
  {
    "message": "The dateOfBirth is a mandatory field"
  }
]
```

`POST` request to `/employee`

Request body:

```
{
  "fullName": "Foo Bar",
  "mobileNumber": 123456789,
  "emailId": "test@gmail.com",
  "dateOfBirth": "1990-01-01"
}
```

The response code is 400, and when converted to JSON, the response body is:

```
[
  {
    "message": "The mobileNumber is a mandatory field"
  }
]
```

`POST` request to `/employee`

Request body:

```
{
  "fullName": "Foo Bar",
  "mobileNumber": 1234567891,
  "emailId": "test-At-gmail.com",
  "dateOfBirth": "1990-01-01"
}
```

The response code is 400, and when converted to JSON, the response body is:

```
[
  {
    "message": "The emailId should be in a valid email format"
  }
]
```

`POST` request to `/employee`

Request body:

```
{
  "fullName": "Foo Bar",
  "mobileNumber": 1234567891,
  "emailId": "test@gmail.com",
  "dateOfBirth": "1990/01/01"
}
```

The response code is 400, and when converted to JSON, the response body is:

```
[
  {
    "message": "The dateOfBirth should be in YYYY-MM-DD format"
```

```
  }
]
```

`POST` request to `/employee`

Request body:

```
{
  "fullName": "Foo Bar",
  "mobileNumber": 1234567891,
  "emailId": "test@gmail.com",
  "dateOfBirth": "1990-01-01"
}
```

The response code is 200. This adds a new object to the database with the given properties and id 1.

Question - 1
Springboot: Music Streaming

This project is a practical assessment designed to evaluate understanding and skills in working with RESTful APIs using Spring Boot.

Below are examples of the JSON data for this project.:

Example of Artist data JSON Object

```
{
  "id": 1,
  "artistName": "Alex Doe",
  "bio": "Alex Doe is a popular artist...",
  "genre": "Pop",
  "origin": "USA",
  "formedYear": "2000",
  "socialLink": "https://www.alexdoe.com",
  "image": "https://www.alexdoe.com/image.jpg",
  "tracksProduced": [
    {
      "id": 1,
      "title": "Track 1",
      "albumName": "Album 1",
      "releaseDate": "2022-01-01",
      "duration": "3:30",
      "genre": "Pop",
      "description": "This is a description of Track 1",
      "playCount": 1000,
      "fileUrl": "https://www.alexdoe.com/track1.mp3",
      "coverImage": "https://www.alexdoe.com/track1.jpg"
    }
  ]
}
```

Example of Playlist data JSON Object

```
{
  "id": 1,
  "name": "My Playlist",
  "description": "This is my favorite playlist",
  "tracks": [
    {
      "id": 1,
      "name": "Track 1",
      "duration": "3:45",
      "artist": {
        "id": 1,
        "name": "Artist 1"
      }
    },
    {
      "id": 2,
      "name": "Track 2",
      "duration": "4:30",
      "artist": {
        "id": 2,
        "name": "Artist 2"
```

```
      }
    }
  ]
}
```

Example of Track data JSON Object

```
{
  "id": 1,
  "title": "Track 1",
  "albumName": "Album 1",
  "releaseDate": "2022-01-01",
  "duration": "3:30",
  "genre": "Pop",
  "description": "This is a description of Track 1",
  "playCount": 1000,
  "fileUrl": "https://www.alexdoe.com/track1.mp3",
  "coverImage": "https://www.alexdoe.com/track1.jpg",
  "artist": {
    "id": 1,
    "artistName": "Alex Doe",
    "bio": "Alex Doe is a popular artist...",
    "genre": "Pop",
    "origin": "USA",
    "formedYear": "2000",
    "socialLink": "https://www.alexdoe.com",
    "image": "https://www.alexdoe.com/image.jpg"
  }
}
```

Implement the following APIs:

ArtistController

- *POST /music/platform/v1/artists* - Create a new artist.
- *GET /music/platform/v1/artists* - Get all artists.
- *GET /music/platform/v1/artists/{artistId}* - Get an artist by ID.
- *PUT /music/platform/v1/artists/{artistId}* - Update an artist by ID.
- *DELETE /music/platform/v1/artists/{artistId}* - Delete an artist by ID.

PlayListController

- *GET /music/platform/v1/playlists/{playlistId}* - Get a playlist by ID.
- *POST /music/platform/v1/playlists* - Create a new playlist.
- *DELETE /music/platform/v1/playlists/{playlistId}* - Delete a playlist by ID.

TrackController

- *GET /music/platform/v1/tracks* - Get all tracks.
- *POST /music/platform/v1/tracks* - Create a new track.
- *GET /music/platform/v1/tracks/{trackId}* - Get a track by ID.
- *PUT /music/platform/v1/tracks/{trackId}* - Update a track by ID.
- *DELETE /music/platform/v1/tracks/{trackId}* - Delete a track by ID.

Complete the project so that it passes all the test cases when running the provided unit tests. The project, by default, supports the use of the H2 database.

▼ Example Requests and Responses

POST request to /music/platform/v1/artists

Request data JSON Object:

```
{
  "artistName": "Alex Doe",
```

```
  "bio": "Alex Doe is a popular artist...",
  "genre": "Pop",
  "origin": "USA",
  "formedYear": "2000",
  "socialLink": "https://www.alexdoe.com",
  "image": "https://www.alexdoe.com/image.jpg"
}
```

Response JSON Object:

```
{
  "id": 1,
  "artistName": "Alex Doe",
  "bio": "Alex Doe is a popular artist...",
  "genre": "Pop",
  "origin": "USA",
  "formedYear": "2000",
  "socialLink": "https://www.alexdoe.com",
  "image": "https://www.alexdoe.com/image.jpg"
}
```

GET request to /music/platform/v1/artists

Response JSON Object:

```
[
  {
    "id": 1,
    "artistName": "Alex Doe",
    "bio": "Alex Doe is a popular artist...",
    "genre": "Pop",
    "origin": "USA",
    "formedYear": "2000",
    "socialLink": "https://www.alexdoe.com",
    "image": "https://www.alexdoe.com/image.jpg"
  }
]
```

GET request to /music/platform/v1/artists/{artistId}

Response JSON Object:

```
{
  "id": 1,
  "artistName": "Alex Doe",
  "bio": "Alex Doe is a popular artist...",
  "genre": "Pop",
  "origin": "USA",
  "formedYear": "2000",
  "socialLink": "https://www.alexdoe.com",
  "image": "https://www.alexdoe.com/image.jpg"
}
```

PUT request to /music/platform/v1/artists/{artistId}

Request data JSON Object:

```
{
  "artistName": "Alex Doe",
  "bio": "Alex Doe is a popular artist...",
  "genre": "Pop",
  "origin": "USA",
  "formedYear": "2000",
  "socialLink": "https://www.alexdoe.com",
  "image": "https://www.alexdoe.com/image.jpg"
}
```

Response JSON Object:

```
{
  "id": 1,
  "artistName": "Alex Doe",
  "bio": "Alex Doe is a popular artist...",
  "genre": "Pop",
  "origin": "USA",
  "formedYear": "2000",
  "socialLink": "https://www.alexdoe.com",
  "image": "https://www.alexdoe.com/image.jpg"
}
```

DELETE request to /music/platform/v1/artists/{artistId}

No response data. Returns a 204 No Content status.

POST request to /music/platform/v1/playlists:

Request data JSON Object:

```
{
  "name": "My Playlist",
  "description": "This is my favorite playlist"
}
```

Response JSON Object:

```
{
  "id": 1,
  "name": "My Playlist",
  "description": "This is my favorite playlist"
}
```

GET request to /music/platform/v1/playlists/{playlistId}

Response JSON Object:

```
{
  "id": 1,
  "name": "My Playlist",
  "description": "This is my favorite playlist"
}
```

DELETE request to /music/platform/v1/playlists/{playlistId}

No response data. Returns a 204 No Content status.

POST request to /music/platform/v1/tracks

Request data JSON Object:

```
{
  "title": "Track 1",
  "albumName": "Album 1",
  "releaseDate": "2022-01-01",
  "duration": "3:30",
  "genre": "Pop",
  "description": "This is a description of Track 1",
  "playCount": 1000,
  "fileUrl": "https://www.alexdoe.com/track1.mp3",
  "coverImage": "https://www.alexdoe.com/track1.jpg"
}
```

Response JSON Object:

```
{
  "id": 1,
  "title": "Track 1",
  "albumName": "Album 1",
  "releaseDate": "2022-01-01",
  "duration": "3:30",
```

```
  "genre": "Pop",
  "description": "This is a description of Track 1",
  "playCount": 1000,
  "fileUrl": "https://www.alexdoe.com/track1.mp3",
  "coverImage": "https://www.alexdoe.com/track1.jpg"
}
```

GET request to /music/platform/v1/tracks

Response JSON Object:

```
[
  {
    "id": 1,
    "title": "Track 1",
    "albumName": "Album 1",
    "releaseDate": "2022-01-01",
    "duration": "3:30",
    "genre": "Pop",
    "description": "This is a description of Track 1",
    "playCount": 1000,
    "fileUrl": "https://www.alexdoe.com/track1.mp3",
    "coverImage": "https://www.alexdoe.com/track1.jpg"
  }
]
```

GET request to /music/platform/v1/tracks/{trackId}

Response JSON Object:

```
{
  "id": 1,
  "title": "Track 1",
  "albumName": "Album 1",
  "releaseDate": "2022-01-01",
  "duration": "3:30",
  "genre": "Pop",
  "description": "This is a description of Track 1",
  "playCount": 1000,
  "fileUrl": "https://www.alexdoe.com/track1.mp3",
  "coverImage": "https://www.alexdoe.com/track1.jpg"
}
```

PUT request to /music/platform/v1/tracks/{trackId}

Request data JSON Object:

```
{
  "title": "Track 1",
  "albumName": "Album 1",
  "releaseDate": "2022-01-01",
  "duration": "3:30",
  "genre": "Pop",
  "description": "This is a description of Track 1",
  "playCount": 1000,
  "fileUrl": "https://www.alexdoe.com/track1.mp3",
  "coverImage": "https://www.alexdoe.com/track1.jpg"
}
```

Response JSON Object:

```
{
  "id": 1,
  "title": "Track 1",
  "albumName": "Album 1",
  "releaseDate": "2022-01-01",
  "duration": "3:30",
  "genre": "Pop",
  "description": "This is a description of Track 1",
  "playCount": 1000,
```

```
  "fileUrl": "https://www.alexdoe.com/track1.mp3",
  "coverImage": "https://www.alexdoe.com/track1.jpg"
}
```

DELETE request to /music/platform/v1/tracks/{trackId}

No response data. Returns a 204 No Content status.

Question - 2

Springboot: Sports Health Coaching

This project is a practical assessment designed to evaluate understanding and skills in working with RESTful APIs using Spring Boot.

Here is an example of a customer JSON object:

```
{
  "id": 1,
  "height": 180,
  "weight": 80,
  "coach": {
    "id": 1,
    "name": "Alex Doe"
  }
}
```

Here is an example of a coach JSON object:

```
{
  "id": 1,
  "name": "Alex Doe"
}
```

The project consists of two empty controller classes, *CustomerController* and *CoachController*, that must be implemented. The controllers should have the following endpoints:

CustomerController

POST request to /api/customer:

- Create a new customer.
- Accepts a JSON body with height, weight, and coach_id fields.
- Return the created record with a 201 Created status.

GET request to /api/customer:

- Return a list of all customer records in the database sorted by id in ascending order.

GET request to /api/customer/{id}:

- Return a customer record with the given id.
- If the customer record exists, return the record with a 200 OK status.
- If the customer record does not exist, return a 404 Not Found status.

CoachController

POST request to /api/coach:

- Create a new coach.
- Accepts a JSON body with name field.
- Return the created record with a 201 Created status.

GET request to /api/coach:

- Return a list of all coach records in the database sorted by id in ascending order.
- Return the list with a 200 OK status.

GET request to /api/coach/{id}:

- Return a coach record with the given id.
- If the coach record exists, return the record with a 200 OK status.
- If the coach record does not exist, return a 404 Not Found status.

Please ensure that all endpoints handle errors appropriately and return the correct HTTP status codes so that all unit tests pass.

▼ **Example Requests and Responses**

POST request to /api/customer:

Request data JSON Object:

```
{
  "height": 180,
  "weight": 80,
  "coach_id": 1
}
```

Response JSON Object:

```
{
  "id": 1,
  "height": 180,
  "weight": 80,
  "coach": {
    "id": 1,
    "name": "Alex Doe"
  }
}
```

GET request to /api/customer:

Response JSON Object:

```
[
  {
    "id": 1,
    "height": 180,
    "weight": 80,
    "coach": {
      "id": 1,
      "name": "Alex Doe"
    }
  }
]
```

GET request to /api/customer/{id}:

Response JSON Object:

```
{
  "id": 1,
  "height": 180,
  "weight": 80,
  "coach": {
    "id": 1,
    "name": "Alex Doe"
  }
}
```

POST request to /api/coach:

Request data JSON Object:

```
{
  "name": "Alex Doe"
```

```
}
```

Response JSON Object:

```
{
  "id": 1,
  "name": "Alex Doe"
}
```

GET request to /api/coach:

Response JSON Object:

```
[
  {
    "id": 1,
    "name": "Alex Doe"
  },
  {
    "id": 2,
    "name": "Sam"
  }
]
```

GET request to /api/coach/{id}:

Response JSON Object:

```
{
  "id": 1,
  "name": "Alex Doe"
}
```

Question - 3

Springboot: Artist Management APIs

In this challenge, implement a simple REST API to manage a collection of Artist data.

Each artist is a JSON entry with the following keys:

- id: the unique ID of the event (Long).
- firstName: first name of the artist (String).
- lastName: last name of the artist (String).

Here is an example of an artist JSON object:

```
{
      "firstName": "Dasun",
      "lastName": "Anushka"
}
```

An implementation of the Artist model is provided. Implement a REST service that exposes the /v1/artists endpoints, which allows for managing the collection of artists' records in the following way:

POST request to /v1/artists:

- CCreates a new artist data record.
- The response code is 201, and the response body is the created record, including its unique id.

GET request to /v1/artists:

- The response code is 200.
- The response body is a list of matching records, ordered by their ids in increasing order.

GET request to /v1/artists/{artistId}:

- Returns a record with the given id and status code 200.
- If there is no record with the given id, the response code is null.

DELETE request to /v1/artists/{artistId}:

- Delete the record with the given id and return status code 204.

Complete the project so that it passes all the test cases when running the provided unit tests. The project, by default, supports the use of the H2 database.

▼ Example requests and responses

`POST` request to /v1/artists

Request body:

```
{
          "firstName": "Dasun",
          "lastName": "Anushka"
}
```

The response code is 201, and when converted to JSON, the response body is:

```
{
        "id" : 1,
        "firstName": "Dasun",
        "lastName": "Anushka"
}
```

This adds a new object to the collection with the given properties and id 1.

`GET` request to /v1/artists

The response code is 200, and when converted to JSON, the response body (assuming that the objects are in the collection) is as follows:

```
[
        {
          "id" : 1,
          "firstName": "Dasun",
          "lastName": "Anushka"
        },
        {
          "id": 2,
          "firstName": "Anushka",
          "lastName": "Lakmal"
        },
        {
          "id": 3,
          "firstName": "Era",
          "lastName": "Jaye"
        }
  ]
```

`GET` request to /v1/artists/1

The response code is 200, and when converted to JSON, the response body is as follows:

```
[
        {
          "id" : 1,
          "firstName": "Dasun",
          "lastName": "Anushka"
        }
  ]
```

If an object with id 1 does not exist, the response is null.

DELETE request to /v1/artists/1
Assuming that an object with id 1 exists, the response code is 204, and the response body is empty.

Question - 4
Springboot: Music Track APIs

In this project, you'll be working with a team to create a music streaming platform. Your main task is to develop a REST API that shares details about music tracks. You need to ensure this API can receive, show, and query data like track title, album, description, and play count. Your work will be judged on how accurately and efficiently the application handles and processes this data.

Music track data is represented as a JSON object, each describing various properties of a music track. The following properties are included:

- id: The unique integer ID of the object(Long).
- title: The title of the track(String).
- albumName: The corresponding album name of the track(String).
- releaseDate: The release date of the track, formatted as YYYY-MM-DD(Date).
- playCount: An integer value denoting how many times a user has played this track(Integer).

Example of a music track data JSON object:

```
{
  "title": "Lost in Echoes",
  "albumName": "Echoes of the Unknown",
  "releaseDate": "2021-07-15",
  "playCount": 5000
}
```

The REST service must expose the endpoint /music/platform/v1/tracks, enabling the management of music track records as follows:

POST request to /music/platform/v1/tracks:

- Creates a new music track record.
- It expects a valid music track data object as its body payload, excluding the id property.
- The service assigns a unique long id to the added object.
- The response includes the created record with its unique id, and the response code is 201.

GET request to /music/platform/v1/tracks:

- Responds with a list of all music track records and a response code of 200.

GET request to /music/platform/v1/tracks/search:

- Responds with music track records filtered by title, Consider track title is unique. (`Eg - /music/platform/v1/tracks/search?title=Henry`)
- The response code is 200. It accepts query string parameter title.
- Records are returned based on matching title.

DELETE request to /music/platform/v1/tracks/{trackId}:

- Deletes the record with the specified track id if it exists in the database with the status code of 204.

Your task is to ensure the project meets all test case criteria when running the provided rspec tests. The project uses H2 database by default. Start by implementing the POST request to music/platform/v1/tracks, as testing other methods requires the POST functionality to work correctly.

▼ Example requests and responses

`POST` request to /music/platform/v1/tracks

Request body:

```
{
   "title":"Lost in Echoes",
   "albumName":"Echoes of the Unknown",
   "releaseDate":"2021-07-15",
   "playCount":5000
}
```

The response code is 201, and when converted to JSON, the response body is:

```
{
   "id":1,
   "title":"Lost in Echoes",
   "albumName":"Echoes of the Unknown",
   "releaseDate":"2021-07-15",
   "playCount":5000
}
```

This adds a new object to the collection with the given properties and id 1.

`GET` request to /music/platform/v1/tracks

The response code is 200, and when converted to JSON, the response body (assuming that the below objects are all objects in the collection) is as follows:

```
[
   {
      "id":1,
      "title":"Lost in Echoes",
      "albumName":"Echoes of the Unknown",
      "releaseDate":"2021-07-15",
      "playCount":5000
   },
   {
      "id":2,
      "title":"Bravos",
      "albumName":"Unknown",
      "releaseDate":"2021-07-16",
      "playCount":100
   }
]
```

`GET` request to /music/platform/v1/tracks/search

The response code is 200, and when converted to JSON, the response body (assuming that the below objects are all objects with title is "Lost in Echoes" also unique) is as follows:

```
{
   "id":1,
   "title":"Lost in Echoes",
   "albumName":"Echoes of the Unknown",
   "releaseDate":"2021-07-15",
   "playCount":5000
}
```

DELETE request to music/platform/v1/tracks/1

Assuming that the object with id 1 exists, then the response code is 204 and the response body is empty.

Question - 5

Springboot: Playlist Management APIs

Implement a simple REST API to manage a collection of music playlist data.

Each PlayList is a JSON entry with the following keys:

- id: the unique ID of the event (Long).
- name: name of the play list (String).
- tracksCount: initial number of tracks in the play list (Integer).

Here is an example of a PlayList JSON object:

```
{
      "name": "Henry1s list",
      "tracksCount": 10
}
```

An implementation of the PlayList model is provided. Implement a REST service that exposes the v1/playlists endpoints, which allows for managing the collection of PlayList records in the following way:

POST request to /v1/playlists:

- Create a new playlist data record.
- The response code is 201, and the response body is the created record, including its unique *id*.

GET request to /v1/playlists:

- The response code is 200.
- The response body is a list of matching records, ordered by their ids in increasing order.

GET request to /v1/playlists/{playListId}:

- Return a record with the given *id* and status code 200.
- If there is no record with the given *id*, the response code is null.

DELETE request to /v1/playlists/{playListId}:

- Delete the record with the given *id* and return status code 204.

Complete the project so that it passes all the test cases when running the provided unit tests. The project, by default, supports the use of the H2 database.

▼ Example requests and responses

`POST` request to /v1/playlists

Request body:

```
{
      "name": "Henry1s list",
      "tracksCount": 10
}
```

The response code is 201, and when converted to JSON, the response body is:

```
{
      "id": 1,
      "name": "Henry1s list",
      "tracksCount": 10
}
```

This adds a new object to the collection with the given properties and id 1.

`GET` request to /v1/playlists

The response code is 200, and when converted to JSON, the response body (assuming that the objects are all in the collection) is as follows:

```
[
   {
      "id":1,
      "name":"Henry1s list",
      "tracksCount":10
   },
   {
      "id":2,
      "name":"Bera",
      "tracksCount":10
   },
   {
      "id":3,
      "name":"Namal",
      "tracksCount":10
   }
]
```


`GET` request to /v1/playlists/1

The response code is 200, and when converted to JSON, the response body is as follows:


```
{
   "id":1,
   "name":"Henry1s list",
   "tracksCount":10
}
```


DELETE request to /v1/playlists/1

Assuming that the object with id 1 exists, the response code is 204, and the response body is empty.