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.