

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": "Henryls 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": "Henryls 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.