Back exam fall 2024 - Trip Planning Application

Exercise Guidelines

- Allowed resources: written materials, personal computers, laptops, extra monitor, and internet resources. Headphones, and listening to music.
- Prohibited: communication with anyone other than the examiner, censor, and proctor. So no use of social media, forums, emails, sms, chatrooms, etc.
- Do not store solutions on external networks or drives/hosts like Facebook, OneDrive, Google Drive, etc. And don't share your code on Github until the end of the exam.
- Duration: 4 hours. Restroom breaks only. No smoking.

Consider your problem solving strategy (important)

1. Read the entire exercise before starting.
2. Sometimes you need to interpret the tasks. If you are unsure, make a decision and document it by adding a comment in the code or the `README.md` file.
3. If you get stuck on a task, move on to the next one.
4. Focus on demonstrating your approach to solving the tasks.
5. You will be asked to create entities and JPA DAOs from the beginning. If you get totally stuck with JPA, and that makes it difficult to continue, you can create a mock DAO instead. The mock DAO should implement the same interface as the JPA DAO and have the data hardcoded in the class in a list or map. BUT, if most of your JPA code is working, you should continue with JPA. Most likely, you will not need to implement a mock DAO today. Consider it the last resort.

Percentage distribution of the tasks

This is a breakdown of the distribution for each task.

Task	Topic	%
2	JPA and DAOs	25%
3	Building a REST Service Provider with Javalin	25%
4	REST Error Handling	5%
5	Streams	10%
6	Getting additional data from API	15%
7	Testing REST Endpoints	15%
8	Security	5%
	Total	100%

Hand in on Wiseflow

1. A zip file containing your whole project, including the `README.md` file with answers to the theoretical questions.
2. A link to your GitHub repository. Don't push your solutions until the very end of the exam. Do not copy the clone link from GitHub, but grab the link from the browser address bar and paste it into Wiseflow.

Introduction

Build a backend system for an e-commerce platform offering trip planning services. Tasks include managing trips and guides. Theoretical questions are part of the exercise.

Domain Description

The application facilitates the booking of guided trips with these properties:

1. **Trip**: starttime, endtime, longitude, latitude, name, price, id, category. (Categories are `beach`, `city`, `forest`, `lake`, `sea`, and `snow`).
2. **Guide**: firstname, lastname, email, phone, yearsOfExperience. A guide can offer multiple trips, but each trip is led by only one guide.

Task 1: Setup

1.1 Create a new Java Project for javalin and JPA.

1.2 Document your work in a `README.md` file.

Task 2: JPA and DAOs (25%)

2.1 Establish a **HibernateConfig** class with a method that returns an **EntityManagerFactory**.

2.2 Implement a **Trip** entity class with the following properties: starttime, endtime, startposition, name, price, id, category. Use an enum for the category of the trip.

2.3 Implement a **Guide** entity class with the following properties: firstname, lastname, email, phone, yearsOfExperience, and a **OneToMany** relationship to trips.

2.4. Implement the DAOs for **Trip** and **Guide**.

2.4.1 Implement a **TripDTO** and a **GuideDTO** class. Use an enum for the category of the trip as in the entity class.

2.4.2 Create a generic Interface **IDAO** with CRUD operations (create, getAll, getById, update, delete), that uses DTOs as arguments and return types.

2.4.3 Create 2 new DAO classes **TripDAO** and **GuideDAO** using JPA and Hibernate. The new DAO classes should implement the **IDAO** interface. You will need to implement the CRUD operations for **TripDAO**, but you should only implement the CRUD operations for **GuideDAO** that you need for this exercise. So wait and see what you need.

2.4.4 Let the **TripDAO** also implement another interface: **ITripGuideDAO** with these additional methods:

- `void addGuideToTrip(int tripId, int guideId)`
- `Set<TripDTO> getTripsByGuide(int guideId)`

2.5 Create a **Populator** class and populate the database with trips and their guides.

Task 3: Building a REST Service Provider with Javalin (25%)

3.1 Develop a REST API with Javalin for trips.

3.2 Create a **TripController** that uses the TripDAO to persist data in the database. Use DTOs to transfer data between the controller and the DAO.

3.3 Create a **TripRoutes** file that uses the **TripController** to handle the API requests.

3.3.1 Implement routes in **TripRoutes** file to handle the API requests. The routes should match the controller methods. That would be someting like this:

Method	Route	Description
GET	/trips	Get all trips.
GET	/trips/{id}	Get a trip by its id.
POST	/trips	Create a new trip. Add guide later.

Method	Route	Description
PUT	/trips/{id}	Update information about a trip.
DELETE	/trips/{id}	Delete a trip.
PUT	/trips/{tripId}/guides/{guideId}	Add an existing guide to and existing trip.
POST	/trips/populate	Populate the database with trips and guides.

3.3.2 Test the endpoints using a **dev.http** file. Document the output in your `README.md` file to verify the functionality.

3.3.3 As a minimum you should request all endpoints once to get all trips, get a trip by id, adding a trip, updating a trip, and delete a trip. Also add a guide to a trip. For each request, document the response in your `README.md` file by copying the response.

3.3.4 When getting a trip by id, the response should include the guide information.

3.3.5 Theoretical question: Why do we suggest a PUT method for adding a guide to a trip instead of a POST method? Write the answer in your `README.md` file.

Task 4: REST Error Handling (5%)

4.1 Return exceptions as JSON. At least for:

- Getting a trip by id, if the trip does not exist
- Deleting a trip that does not exist.

Feel free to add more error handling as needed.

Task 5: Streams and queries (10%)

5.1 Create a method in **TripController** to filter trips by category, and add a new route to the **TripRoutes** file to handle the request.

5.2 In a similar manner, find a way to get an overview with each guide, and the total sum price of all trips offered by each guide. Like this:

```
[
{
  "guideId": 1,
  "totalPrice": 1000
},
{
  "guideId": 2,
  "totalPrice": 2000
}
]
```

And so on.

Use JPA and / or streams to solve the task as you see fit. The result should be returned as a JSON object in a new endpoint. Call it `trips/guides/totalprice`.

Task 6: Getting additional data from API (15%)

6.1 Depending on the trip category get packing items from external api.

6.1.1 The external API is available at `https://packingapi.cphbusinessapps.dk/packinglist/{category}`.

6.1.2 The available categories are `beach`, `city`, `forest`, `lake`, `sea` and `snow`.

6.1.3 The API returns a JSON object with a list of items to pack for the trip in this format:

```
{
  "items": [
    {
      "name": "Beach Umbrella",
      "weightInGrams": 1200,
      "quantity": 1,
      "description": "Sunshade umbrella for beach outings.",
      "category": "beach",
      "createdAt": "2024-10-30T17:44:58.547Z",
      "updatedAt": "2024-10-30T17:44:58.547Z",
      "buyingOptions": [
        {
          "shopName": "Sunny Store",
          "shopUrl": "https://shop3.com",
          "price": 50
        },
        {
          "shopName": "Beach Essentials",
          "shopUrl": "https://shop4.com",
          "price": 55
        }
      ]
    },
    ...
  ]
}
```

NB: The date format for `createdAt` and `updatedAt` is `ZonedDateTime` format, like `2024-10-30T17:44:58.547Z`. Jackson might need an extra dependency to handle this format, and this custom configuration of the ObjectMapper:

```
ObjectMapper objectMapper = new ObjectMapper();
objectMapper.registerModule(new JavaTimeModule());
```

```
objectMapper.disable(SerializationFeature.WRITE_DATES_AS_TIMESTAMPS);
```

6.2 Implement a method in the **TripController** that fetches the packing items for a trip based on the category.

6.3 Add the packing items to the response of the endpoint for getting a trip by id.

6.4 Add a new endpoint to get the sum of the weights of all packing items for a trip.

Task 7: Testing REST Endpoints (15%)

7.1 Create a test class for the REST endpoints in your **TripRoutes** file.

7.2 Set up `@BeforeAll` to create the Javalin server, the **TripController**, **TripRoutes**, and the **EntityManagerFactory** for testing.

7.3 Configure the `@BeforeEach` methods to create the test objects (Trips and Guides).

7.4 Create a test method for each of the endpoints.

7.5 Test the "trip by id" endpoint to verify that the packing items are returned.

Task 8: Security (5%)

8.1 Implement an authentication mechanism for the REST API using JWT (with login and protected endpoints).

8.2 Add allowed roles for each endpoint (make sure everyone can use at least the login endpoint).

8.3 Adding security roles to the endpoints will make the corresponding Rest Assured Test fail. Now the request will return a `401` **Unauthorized response**. Describe how you would fix the failing tests in your `README.md` file, or if time permits, implement the solution so your tests pass.
