# Exam 2025 - Ski Instructor

## Exercise Guidelines

- Allowed resources: written materials, personal computers, laptops, extra monitors, and internet resources. Headphones, and listening to music.
- Prohibited: communication with anyone. 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: 5 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 a class using a list or map. However, 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

| Task | Topic | % |
|---|---|---|
| 1 | Setup | 5 |
| 2 | JPA and DAOs | 25 |
| 3 | Building a REST Service Provider with Javalin | 25 |
| 4 | REST Error Handling | 5 |
| 5 | Streams and Queries | 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 ski instructor services. The platform should manage ski instructor details, and the ski lessons that they lead. This system will include managing instructor profiles, their lesson schedules, and client bookings.

---

## Domain Description

The application facilitates the booking of ski lessons with these properties:

1. **SkiLesson**: starttime, endtime, location (longitude, latitude), name, price, id, level. Levels are `beginner`, `intermediate`, `advanced`.
2. **Instructor**: firstname, lastname, email, phone, yearsOfExperience. An instructor can offer multiple lessons, but each lesson is led by only one instructor.

---

## 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 **SkiLesson** entity class with the following properties: starttime, endtime, location (latitude, longitude), name, price, id, level. Use an enum for the level of the ski lesson.

2.3 Implement an **Instructor** entity class with the following properties: firstname, lastname, email, phone, yearsOfExperience, and a **OneToMany** relationship to ski lessons.

2.4 Implement the DAOs for **SkiLesson** and **Instructor**.

2.4.1 Implement a **SkiLessonDTO** and a **InstructorDTO** class. Use an enum for the level of the ski lesson 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 **SkiLessonDAO** and **InstructorDAO** using JPA and Hibernate. The new DAO classes should implement the **IDAO** interface. You will need to implement the CRUD operations for

**SkiLessonDAO**, but you should only implement the CRUD operations for **InstructorDAO** that you need for this exercise. So wait and see what you need.

2.4.4 Let the **SkiLessonDAO** also implement another interface: **ISkiLessonInstructorDAO** with these additional methods:

- `void addInstructorToSkiLesson(int lessonId, int instructorId)`
- `Set<SkiLessonDTO> getSkiLessonsByInstructor(int instructorId)`

2.5 Create a **Populator** class and populate the database with ski lessons and their instructors.

---

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

3.1 Develop a REST API with Javalin for ski lessons.

3.2 Create a **SkiLessonController** that uses the SkiLessonDAO to persist data in the database.

3.3 Create a **SkiLessonRoutes** file that uses the **SkiLessonController** to handle the API requests.

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

| Method | Route | Description |
|---|---|---|
| GET | /skilessons | Get all ski lessons. |
| GET | /skilessons/{id} | Get a ski lesson by its id. |
| POST | /skilessons | Create a new ski lesson. Add instructor later. |
| PUT | /skilessons/{id} | Update information about a ski lesson. |
| DELETE | /skilessons/{id} | Delete a ski lesson. |
| PUT | /skilessons/{lessonId}/instructors/{instructorId} | Add an existing instructor to an existing ski lesson. |
| POST | /skilessons/populate | Populate the database with ski lessons and instructors. |

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 ski lessons, get a ski lesson by id, adding a ski lesson, updating a ski lesson, and deleting a ski lesson. Also add an instructor to a ski lesson. For each request, document the response in your `README.md` file by copying the response.

3.3.4 When getting a ski lesson by id, the response should include the instructor information.

3.3.5 Theoretical question: Why do we suggest a PUT method for adding an instructor to a ski lesson 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 ski lesson by id, if the lesson does not exist.
- Deleting a ski lesson 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 **SkiLessonController** to filter ski lessons by level, and add a new route to the **SkiLessonRoutes** file to handle the request.

5.2 In a similar manner, find a way to get an overview with each instructor.

- Choose between these two outputs: (The latter might is a little harder)

1. The total sum price of all lessons offered by each instructor. Like this:

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

or 2. The sum of lesson time for each instructor. Like this:

```
[
{
  "instructorId": 1,
  "totalTime": 120
},
{
  "instructorId": 2,
  "totalTime": 240
}
]
```

## Task 6: Getting Additional Data from API (15%)

6.1 Depending on the ski lesson level, get ski instruction data from an external API.

6.1.1 The external API is available at `https://apiprovider.cphbusinessapps.dk/skilesson/{level}`.

6.1.2 The available levels are `beginner`, `intermediate`, and `advanced`.

6.1.3 The API returns a JSON object with a list of instructions and tips for each level in the following format:

```
[
    {
      "title": "Basic Skiing Stance",
      "description": "Learn the proper stance for skiing to maintain
balance and control.",
      "level": "Beginner",
      "durationMinutes": 30,
      "createdAt": "2024-10-30T17:44:58.547Z",
      "updatedAt": "2024-10-30T17:44:58.547Z"
    },
    {
      "title": "Turning Techniques",
      "description": "Master the art of turning on the slopes to control
your speed.",
      "level": "Intermediate",
      "durationMinutes": 45,
      "createdAt": "2024-10-30T17:44:58.547Z",
      "updatedAt": "2024-10-30T17:44:58.547Z"
    },
    {
      "title": "Advanced Off-Piste Techniques",
      "description": "Learn advanced techniques for skiing off-piste,
focusing on safety and maneuvering.",
      "level": "Advanced",
      "durationMinutes": 60,
      "createdAt": "2024-10-30T17:44:58.547Z",
      "updatedAt": "2024-10-30T17:44:58.547Z"
    },
    ...
]
```

6.2 Implement a method in the SkiLessonController that fetches ski lesson instructions for lessons based on the level (beginner, intermediate, advanced).

- The method should return a list of ski lesson instructions for the given level.

6.3 Add the ski lesson instructions to the response of the endpoint for getting a ski lesson by id.

6.4 Add a new endpoint to get an overview of the total duration of instructions for a ski lesson based on its level.

## Task 7: Testing REST Endpoints (15%)

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

7.2 Set up @BeforeAll to create the Javalin server, the SkiLessonController, SkiLessonRoutes, and the EntityManagerFactory for testing.

7.3 Configure the @BeforeEach methods to create the test objects (SkiLessons and Instructors).

7.4 Create a test method for each of the endpoints.

7.5 Test the "ski lesson 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 Make creating, updating and deleting ski lessons require a role of `admin`.

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.