Mock Exam: Programming and Technology

Exam guidelines

All written materials, PCs, laptops, and internet resources are allowed during the exam.

We expect you to use code from your previous assignments and projects, otherwise you will not have time to complete the exam.

Mobile phones and communication with other individuals other than communication with the examiner, censor, and proctor are prohibited.

You are not allowed to save your solutions on external networks, drives/hosts such as GitHub, Facebook, Google Drive, DropBox, OneDrive or similar. Violation of this rule will result in expulsion from the exam, and appropriate sanctions will be imposed on both the sender/uploader and the receiver.

At the end of the exam, you must upload your entire solution to Wiseflow. Your upload should be in the form of a zip file containing all your solutions and the document with your answers to the theoretical questions (a `README.md` file).

The exam duration is 4 hours. You may only leave the exam room for restroom breaks. Smoking is not allowed.

When the 4 hours have passed, an individual assesment round will take place. This will take approximately 10 minutes per student.

Introduction

You are required to program parts of a backend for an online web shop, including adding new items to the web shop and more.

In addition to programming this system, there will be theoretical questions along the way where you will be asked to explain considerations and provide explanations. These should be written in a document (a `README.md` file), which should be uploaded to Wiseflow along with your code.

Domain Description

Lyngby Garden Center and other Plant Resellers want to sell garden plants online. Garden plants are displayed with information such as plant type (Roses, Rhododendrons, shrubs, ...), name, size, and price.

More specifically, you need to program a system that can handle the following properties for plants:

- PlantId, a unique identifier
- PlantType
- PlantName
- Price
- MaxHeight

Plant properties/data can be displayed as in the table below:

PlantId	PlantType	PlantName	MaxHeight	Price
1	Rose	Albertine	400	199.50
2	Bush	Aronia	200	169.50
3	FruitAndBerries	AromaApple	350	399.50
4	Rhododendron	Astrid	40	269.50
5	Rose	The DarkLady	100	199.50

There are, of course, many more plants, but they are not shown here.

Task 1: Build a REST Service Provider with Javalin

1.1 Create a Java application using Javalin, named `PlantShopService`

1.2 Create a **`README.md`** file in your project. This file should contain your answers to the questions, that need a written answer. We have marked those questions with a **`README.md`** tag. Please add task numbers for each answer.

1.3 Implement a `PlantDTO` class with properties: PlantId, PlantType, PlanteName, MaxHeight, Price.

1.4 Develop an API in Javalin with the following endpoints:

HTTP method	REST Ressource	json	Comment
GET	`/api/plants`	`response:` [{"id": 1, "planttype": "Rose", "plantname": "Albertine", "maxheight": 400, "price": 199.50}, ...]	Retrieve all plants
GET	`/api/plants/{id}`	`response:` {"id": 1, "planttype": "Rose", "plantname": "Albertine", "maxheight": 400, "price": 199.50}	Retrieve a plant by its ID
GET	`/api/plants/type/{type}`	`response:` [{"id": 1, "planttype": "Rose", "plantname": "Albertine", "maxheight": 400, "price": 199.50}, ...]	Retrieve plants by type
POST	`/api/plants`	`request payload:` {"planttype": "Rose", "plantname": "Gallicanae", "maxheight": 350, "price": 299.0} `response:` {"id": 6, "planttype": "Rose",	Add a new plant. The created plant object should be returned with the assigned ID

		"plantname": "Gallicanae", "maxheight": 350, "price": 299.0}	

The solution should include:

1.4.1 Routing

1.4.2 A controller, `PlantController`, based on an interface `IPlantController`. The controller methods should each return a `Handler` and each handler should return a `json` string.

1.4.3 To begin with, the data should be held in an in-memory Java datastructure, which means that we "mock" the database. For this, create an `iPlantDAO` interface and implement the interface as a `PlantDAOMock` class. Manage the list of plants in the `PlantDAOMock` as a static arraylist or a hashmap. You decide as long a the interface contract is fulfilled.
The `iPlantDAO` should have these abstract methods:

- List getAllPlants()
- PlantDTO getPlantById(int id)
- List getPlantsByType(String type)
- PlantDTO addPlant(PlantDTO plant)

1.5 Create a `dev.http` file and test the endpoints. Copy the output to your **`README.md`** file.

Task 2: REST Errorhandling

2.1 In your implementation various exceptions can occur. Think about where these exceptions can happen, and how to handle them. Note in your `REAMDME.md` file for each endpoint which errors you handle, and which HTTP status codes you wish to return. Like this:

HTTP method	REST Ressource	Exceptions and status(es)
GET	`/api/plants`	
GET	`/api/plants/{id}`	
GET	`/api/plants/type/{type}`	
POST	`/api/plants`	

(feel free to cut'n paste this markdown and fill out):

```
|HTTP method | REST Ressource | Exceptions and status(es) |
|---|---|---|
|GET| `/api/plants`|  |
|GET|`/api/plants/{id}`|  |
|GET|`/api/plants/type/{type}` |  |
|POST| `/api/plants` |  |
```

2.2 Implement a REST error handler that returns a JSON object with the following properties:

- status: The HTTP status code.
- message: A message describing the error.
- timestamp: The time of the error.

2.3 Implement one or more Exception mappers that maps exceptions to the appropriate HTTP status code.

Task 3: Streams and Generics

The easiest way to manually test the methods below is probably through a main method in its own class. You could also do it through unit-tests.

Now add methods in the `PlantDAOMock` class that:

3.1 returns a list of plants with a maximum height of 100 cm using the `stream API`, `filter()` and a `predicate function`.

3.2 maps / converts a list of PlantDTOs to a list of Strings containing the plant names. Again use the `stream API` and the `map` function.

3.3 sorts a list of PlantDTOs by name using `streams`, `sorted()`, and a `Comparator`.

3.4 Please note in your **`README.md`** file which programming paradigm the `stream API` is inspired by.

The next step is introducing generics:

3.5. Create a new interface to generalize `iPlantDAO` by using generics, so it can handle any type of DTO (and change its name to something more generic like `iDAO`).

3.6. Create a new DTO class: `ResellerDTO` with the following properties: id, name, address, phone. This is a suggestion for reseller data:

Id	Name	Address	Phone
1	Lyngby Plantecenter	Firskovvej 18	33212334
2	Glostrup Planter	Tværvej 35	32233232
3	Holbæk Planteskole	Stenhusvej 49	59430945

3.7. Implement two new DAOs: `ResellerDAOGeneric` and `PlantDAOgeneric` using the generic DAO interface.

Task 4: JPA

NOTE: Task 6 is about testing. You have the option to do task 5 and 6 together as TDD.

4.1 Setup a `HibernateConfig` class with a method that returns a `EntityManagerFactory`.

4.2 Implement a `Plant` entity class with the following properties: id, type, name, maxHeight, price.

4.3 Implement a `Reseller` entity class with the following properties: id, name, address, phone, and a `OneToMany` relationship to `Plant`. This means that a reseller (Plant Shop) can have many plants in stock.

4.4 Make a `IPlantCenterDAO` interface with the following 7 methods (disregard the interfaces from previous tasks):

- List getAllPlants()
- Plant getPlantById(int id)
- List getPlantsByType(String type)
- Plant addPlant(PlantDTO plant)
- Plant deletePlant(int id)
- Reseller addPlantToReseller(int resellerId, int plantId)
- List getPlantsByReseller(int resellerId)

Note that the methods are returning JPA entities and receiving DTO types or primitive datatypes.

4.5 Implement the `IPlantCenterDAO` interface in a `PlantCenterDAO` class using JPA and Hibernate.

4.6 The last step is to change the endpoints to persist data in the database instead of the mock-version we used earlier. Create a new controller called `PlantControllerDB` to replace `PlantController` - and hook up the handlers to your `PlantCenterDAO`.

4.7 Run the `dev.http` file and test the endpoints again. They should still work. Copy the output to your **`README.md`** file.

4.8 If time permits, then add the remaining endpoints to the routing and the controller (deletePlant, addPlantToReseller, and getPlantsByReseller)

Task 5: Create automated tests for the PlantCenterDAO class

5.1 Setup `@BeforeAll` to create the `EntityManagerFactory`.
5.2 Setup the `@BeforeEach` and `@AfterEach` methods to create the test objects (Plants and Resellers).
5.3 Create a test method for each of the methods in the `PlantCenterDAO` class.
5.4 Please describe in you own words the main differences between regular unit-tests and tests done in this task in your **`README.md`** file.

Task 6: Create a Test to test the REST endpoints

6.1 Create a test class for the REST endpoints.
6.2 Setup `@BeforeAll` to create the Javalin server, the `PlantControllerDB` and the `EntityManagerFactory` for test.
6.3 Setup the `@BeforeEach` and `@AfterEach` methods to create the test objects (Plants and Resellers).
6.4 Create a test method for each of the endpoints in the `PlantControllerDB` class.
6.5 Please describe in your own words why testing REST endpoints is different from the tests you did in Task 5. Write you answer in your **`README.md`** file.