Programming Exam - Medical Appointments

This mock exam is set to 4 hours

Exam guidelines

All written materials, PCs, laptops and internet resources are permitted during the examination.

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 anyone other than the examiner, censor and proctor are prohibited.

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

At the end of the exam, you must upload your entire project 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.

Later in the week, an individual assesment round will take place. This will take 30 minutes per student.

OBS! Since this is a Mock-exam - the real exam will be different, but in the naborhood of this mock-exam.

Introduction

You are required to program parts of a backend for a medical clinic, that has doctors and appointments.

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

"Medical Appointments", is a platform that keep track of medical doctors and their appointments.

More specifically, you need to program an API that can handle the following properties for a doctor:

- id, a unique identifier
- name
- date of birth
- year of graduation from medical school
- name of clinic
- speciality. Can be surgery, family medicin, psychiatry, pediatrics or geriatrics.

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

ID	Name	Date of Birth	Year of Graduation	Name of Clinic	Speciality
1	Dr. Alice Smith	1975-04-12	2000	City Health Clinic	FAMILY_MEDICINE
2	Dr. Bob Johnson	1980-08-05	2005	Downtown Medical Center	SURGERY
3	Dr. Clara Lee	1983-07-22	2008	Green Valley Hospital	PEDIATRICS
4	Dr. David Park	1978-11-15	2003	Hillside Medical Practice	PSYCHIATRY
5	Dr. Emily White	1982-09-30	2007	Metro Health Center	GERIATRICS
6	Dr. Fiona Martinez	1985-02-17	2010	Riverside Wellness Clinic	SURGERY
7	Dr. George Kim	1979-05-29	2004	Summit Health Institute	FAMILY_MEDICINE

These rows include fictitious names, dates of birth, graduation years, clinic names, and specialties.

- The data type for the `Date of birth` is `LocalDate`.
- The data type for the speciality should be an `enum`.

Task 1: Build a REST Service Provider with Javalin

1.1 Create a Java project using the Javalin framework.

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 Create an `enum` called `Speciality` with the following types: SURGERY, FAMILY_MEDICINE, PSYCHIATRY, PEDIATRICS or GERIATRICS

1.4 Implement a `DoctorDTO` class with attributes: `id`, `name`, `dateOfBirth`, `yearOfGraduation`, `nameOfClinic`, and `speciality` (enum).

1.5 Develop an API in Javalin with the following endpoints:

	HTTP method	REST Resource	Status (ok)	Not okay
1.	GET	`/api/doctors`	200	500
2.	GET	`/api/doctors/{id}`	200	404 Not found / 500
3.	GET	`/api/doctor/speciality/{speciality}`	200	404 Not found / 500
4.	GET	`/api/doctor/birthdate/range`	200	400 / 404 / 500
5.	POST	`/api/doctors`	201	400 / 500
6.	PUT	`/api/doctors/{id}`	200	400 / 404 / 500

The details for each endpoint can be found in Appendix A.

The solution should include:

1.5.1 Routing

1.5.2 Create a controller called `DoctorMockController`. The controller methods should each generate a `JSON` response.

1.5.3 To begin with, the data should be held in an in-memory Java data structure, which means that we "mock" the database. For this, create a DAO class, `DoctorMockDAO` (or use your own naming). Manage the list of doctors in the `DoctorMockDAO` as a static ArrayList. Populate the array with the fake data in the table above.

The `DoctorDAOMock` should have these methods:

```
List<DoctorDTO> readAll();
DoctorDTO read(int id); // *) use streams
List<DoctorDTO> doctorBySpeciality(Speciality speciality);  // *) use
streams
List<DoctorDTO> doctorByBirthdateRange(LocalDate from, LocalDate to); //
*) use streams
DoctorDTO create(DoctorDTO doctor);
DoctorDTO update(int id, DoctorDTO doctor);
// *) use streams to implement these methods
```

- Use `java streams` where indicated to build the reponse.
- The `doctorBySpeciality` method should return a list of doctors with a specific speciality.
- The `doctorByBirthdateRange` method should return a list of doctors that is born within a specified date range. For example between 2004 and 2009, both dates inclusive.

1.5.4 Create a `doctor.http` file and test the endpoints. Copy the output to your **`README.md`** file to document that the endpoints are working as expected.

Task 2: REST Error Handling

2.1 In your implementation various exceptions may occur. Each endpoint will also need to return a HTTP Statuscode. Please make sure that your endpoints will live up to these requirements:

	HTTP method	REST Resource	Status (ok)	Not okay
1.	GET	`/api/doctors`	200	500
2.	GET	`/api/doctors/{id}`	200	404 Not found / 500
3.	GET	`/api/doctor/speciality/{speciality}`	200	404 Not found / 500
4.	GET	`/api/doctor/birthdate/range`	200	400 / 404 / 500
5.	POST	`/api/doctors`	201	400 / 500
6.	PUT	`/api/doctors/{id}`	200	400 / 404 / 500

Prioritize to implement exception handling for endpoint number 2, 3, 4, and 5.

2.2 An exception should be returned from an endpoint request as `JSON` with these properties:

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

Example:

```
{
  "status": 404,
  "message": "Doctor not found - /api/doctor/34",
  "timestamp": "2023-11-24 10:57:19.373"
}
```

Use your `doctor.http` file to provoke these errors to occur.

Task 3: Generics

3.1 Create a generic interface named `iDAO` that works with both `DoctorDAOMock` and other DTOs. You should refactor `DoctorDAOMock` to work with this generic interface.

3.2 Explain in the `README.md` file the purpose of generics in this exercise. Why can it be helpful?

Task 4: JPA and Persistence

4.1 Set up a `HibernateConfig` class with a method that returns an `EntityManagerFactory` for managing doctor-related entities in your project.

4.2 Create a new `jpa entity class` called `Doctor` with these attributes: `id`, `name`, `dateOfBirth`, `yearOfGraduation`, `nameOfClinic`, `speciality` (enum), `createdAt`, and `updatedAt`. The `createdAt` and `updatedAt` attributes should be of type `LocalDateTime` - and should not be included in the constructor.

4.3 Use the `@PrePersist` and `@PreUpdate` annotations to set the `createdAt` and `updatedAt` properties when a new doctor is created or updated.

4.4 Create a new `jpa entity class` called `Appointment` with the following properties: `id`, `clientName`, `date`, `time`, `comment`. Here are some examples for appointments:

ID	ClientName	Date	Time	Comment
1	John Smith	2023-11-24	09:45	First visit
2	Alice Johnson	2023-11-27	10:30	Follow up
3	Bob Anderson	2023-12-12	14:00	General check
4	Emily White	2023-12-15	11:00	Consultation
5	David Martinez	2023-12-18	15:30	Routine checkup
6	Clara Lee	2023-12-20	08:45	Vaccine shot

These rows include fictitious names, dates, times, and comments for appointments.

4.5 Create an `AppointmentDTO` class with the following properties: `id`, `clientName`, `date`, `time`, `comment`.

4.6 The relation between `Doctor` and `Appointment` should be a `OneToMany` relationship. This means that a doctor can have many appointments, but an appointment can only be assigned to one doctor.

4.7 The only `Doctor` attributes that are required for update operations are `dateOfBirth`, `yearOfGraduation`, `nameOfClinic`, and `speciality`.

4.8 Create a DAO class called `DoctorDAO`, using JPA and Hibernate. The new DAO should implement the `iDAO` interface and have these methods:

```
        List<Doctor> readAll()
        Doctor read(int id)
        List<Doctor> doctorBySpeciality(Speciality speciality);
        List<Doctor> doctorByBirthdateRange(LocalDate from, LocalDate to);
        Doctor create(DoctorDTO route)
        Doctor update(int id, DoctorDTO doctor)
```

You have the option to follow a Test-Driven Development (TDD) approach for tasks 4 and 5 together or manually test the DAO class in a static method.

4.9 Create a Populator class and populate the database with at least to doctors with 2 appointments each.

4.10 In the final step, replace the existing endpoints to persist data in the database instead of the mock version used earlier. Create a new controller named `DoctorControllerDB` to replace `DoctorMockController` and connect the handlers to your `DoctorDAO`.

4.11 Run the `train.http` file and test the endpoints to ensure they work as expected. Copy the output to your **`README.md`** file to document the results.

Task 5: Automated Tests for DoctorDAO

6.1 Set up the `@BeforeAll` method to create the `EntityManagerFactory` for managing doctor-related entities.

6.2 Configure the `@BeforeEach` and `@AfterEach` methods to create and clean up the test objects, including doctors, appointments, or any other relevant entities.

6.3 Create a test method for each of the methods in the `DoctorDAO` class.

6.4 In your **`README.md`** file, please describe the main differences between regular unit tests and tests performed in this task.

Task 6: Testing the Doctor API with REST Assured

This task is a theoretical one. Please type brief answers in the `README.md` file.

7.1 Describe the purpose of Rest Assured and why we want to test the endpoints in this way.

7.2 Describe in words how we set up our database for tests.

7.3 Please describe why testing REST endpoints is different from the tests you performed in Task 5.

END OF TASKS

Appendix A: Endpoints for doctor API

1. **List All Doctors**
    - Endpoint: `/api/doctor/`
    - Method: `GET`
    - Description: Retrieves a list of all doctors.
    - json response example:
    - ```
      [
         {
           "id": 1,
           "name": "Dr. Alice Smith",
           "dateOfBirth": "1975-04-12",
           "yearOfGraduation": 2000,
           "nameOfClinic": "City Health Clinic",
           "speciality": "FAMILY_MEDICINE"
         } .....
         // More doctors...
       ]
      ```
2. **Get Doctor by ID**
    - Endpoint: `/api/doctor/{id}`
    - Method: `GET`
    - Description: Retrieves details of a specific doctor based on their ID.
    - json response example:
    - ```
      {
        "id": 1,
        "name": "Dr. Alice Smith",
        "dateOfBirth": "1975-04-12",
        "yearOfGraduation": 2000,
        "nameOfClinic": "City Health Clinic",
        "speciality": "FAMILY_MEDICINE"
      }
      ```

3. **Search Doctors by Speciality**
    - Endpoint: `/api/doctor/speciality/{speciality}`
    - Method: `GET`
    - Description: Retrieves a list of doctors who specialize in a specific area (e.g., surgery, psychiatry).
    - The json response is similar to `List All Doctors`.
4. **Get Doctors by Date of Birth Range**
    - Endpoint: `/api/doctor/birthdate/range`
    - Method: GET
    - Description: Retrieves doctors born within a specified date range. The range is provided as query parameters. Like this: `/api/doctors/birthdate/range?from=1975-01-01&to=1979-01-01`
    - The json response is similar to `List All Doctors`.
5. **Create New Doctor**
    - Endpoint: `/api/doctor/`
    - Method: `POST`
    - Description: Adds a new doctor. The doctor's details are sent in the request body.
    - json request body example:

```
{
  "name": "Dr. Sophus Olsson",
  "dateOfBirth": "1980-05-21",
  "yearOfGraduation": 2008,
  "nameOfClinic": "Green Valley Hospital",
  "speciality": "PEDIATRICS"
}
```

    - json response example:

```
{
  "id": 6, // newly created id from database
  "name": "Dr. Sophus Olsson",
  "dateOfBirth": "1980-05-21",
  "yearOfGraduation": 2008,
  "nameOfClinic": "Green Valley Hospital",
  "speciality": "PEDIATRICS"
}
```

6. **Update Doctor Information**

   - Endpoint: `/api/doctor/{id}`
   - Method: `PUT`
   - Description: Updates the information of an existing doctor. The updated details are sent in the request body.
   - json request body example:

   ```
   {
     "name": "Dr. Alice Smith",
     "dateOfBirth": "1975-04-12",
     "yearOfGraduation": 2001, // new graduation year
     "nameOfClinic": "City Health Clinic",
     "speciality": "FAMILY_MEDICINE"
   }
   ```

   - json response example (the updated entity):

   ```
   {
     "id": 1,
     "name": "Dr. Alice Smith",
     "dateOfBirth": "1975-04-12",
     "yearOfGraduation": 2001,
     "nameOfClinic": "City Health Clinic",
     "speciality": "FAMILY_MEDICINE"
   }
   ```