# Programming and E-commerce Exercise

November 2023, the exercise is designed for completion within 4 hours.

## Exercise Guidelines (for the real exam)

- You are allowed to use all written materials, personal computers, laptops, and internet resources.
- Mobile phones and communication with anyone other than the examiner, censor, and proctor are prohibited.
- You must not store your solutions on external networks, drives/hosts such as GitHub, Facebook, Google Drive, Dropbox, OneDrive, or similar. Violation of this rule will result in disqualification and appropriate sanctions.
- **Start by skim reading the entire exercise.** This will give you an overview of the tasks and help you plan your time. **If you get stuck with a certain task**, move on to the next one to not waste time. And remember it is **not important to complete all tasks**, but rather to show how you approach the task.
- At the end of the exercise, upload your project to wiseflow in the form of a zip file. This should include all your solutions and a document containing your answers to the theoretical questions (a `README.md` file).
- The exercise duration is 4 hours. You may only leave the exercise area for restroom breaks. Smoking is not allowed.

## Introduction

In this exercise, you are tasked with building a backend system for an e-commerce platform that sells health products. Your responsibilities include adding new products to the store and more.

In addition to programming this system, there will be theoretical questions where you will need to provide explanations. These should be documented in a `README.md` file, along with your code.

## Domain Description

Lyngby Health Store and other Health Product Retailers aim to sell a variety of health products online. These health products are characterized by their category (Vitamins, Supplements, Personal Care, ...), name, price, calories, and description.


You will be working with a system that handles the following properties for health products:

- id, a unique identifier
- category
- name
- price
- calories
- description

Health product data can be displayed in a tabular format as shown below:

| id | category | name | calories | price | description | expireDate |
|---|---|---|---|---|---|---|
| 1 | Vitamins | Multivitamin | 20 | 25.99 | A comprehensive daily multivitamin | 2024-12-31 |
| 2 | Supplements | Omega-3 | 15 | 19.50 | Fish oil supplement rich in omega-3 | 2025-06-30 |
| 3 | Personal Care | Aloe Vera Gel | 5 | 12.99 | Soothing and moisturizing aloe vera gel | 2023-10-15 |
| 4 | Vitamins | Vitamin C | 0 | 9.99 | Immune system support with vitamin C | 2024-08-20 |
| 5 | Supplements | Protein Powder | 120 | 29.99 | Whey protein powder for muscle recovery | 2023-11-30 |

There are many more health products, but they are not included here.

## Task 1: Building a REST Service Provider with Javalin

1.1 Start by creating a Java project using the Javalin framework.

1.2 Document your work in a `README.md` file in your project. This file should contain your answers to questions that require written explanations. These questions are marked with a `README.md` tag. Be sure to include task numbers for each answer.

1.3 Implement a `HealthProductDTO` class that contains properties: `id, category, name, price, calories, and description`.

1.4 Develop a REST API in Javalin with the following endpoints for health products:

| HTTP Method | REST Resource | JSON | Description |
|---|---|---|---|
| GET | `/api/healthproducts` | `response:` [{"id": 1, "category": "Vitamins", "name": "Multivitamin", "calories": 20, "price": 25.99, "description": "A comprehensive daily multivitamin", "expireDate": "2024-12-31"}, ...] | Retrieve all health products |

| HTTP Method | REST Resource | JSON | Description |
|---|---|---|---|
| GET | `/api/healthproducts/{id}` | `response:` {"id": 1, "category": "Vitamins", "name": "Multivitamin", "calories": 20, "price": 25.99, "description": "A comprehensive daily multivitamin", "expireDate": "2024-12-31"} | |
| POST | `/api/healthproducts` | `request payload:` {"category": "Vitamins", "name": "Vitamin D", "calories": 5, "price": 12.50, "description": "Supports bone health"} `response:` {"id": 6, "category": "Vitamins", "name": "Vitamin D", "calories": 5, "price": 12.50, "description": "Supports bone health", "expireDate": "2025-12-31"} | Add a new health product. The created product object should be returned with the assigned `id` and an `expireDate` field. |
| PUT | `/api/healthproducts/{id}` | `request payload:` {"category": "Vitamins", "name": "Vitamin D Plus", "calories": 10, "price": 15.50, "description": "Enhanced formula for bone health", "expireDate": "2025-12-31"} `response:` {"id": 6, "category": "Vitamins", "name": "Vitamin D Plus", "calories": 10, "price": 15.50, "description": "Enhanced formula for bone health", "expireDate": "2025-12-31"} | Update an existing health product by ID. The updated product object should be returned with the same `id` and updated fields. |

Your solution should include:

1.4.1 A controller, `HealthProductController`, based on an interface `IHealthProductController`. The interface should contain at least the following methods:

```
    Handler getAll();
    Handler getById();
    Handler create();
    Handler update();
    Handler delete();
```

In case you use the double colon syntax `::` in the endpoint handlers, the interface should look like this instead:

```
    void getAll(Contex ctx);
    void getById(Contex ctx);
    void create(Contex ctx);
    void update(Contex ctx);
    void delete(Contex ctx);
```

1.4.2 Routing

- Set all routes in a separate class, `HealthProductRoutes`, that has a method with the signature: `EndpointGroup getHealthProductRoutes()`
- Make sure that all routes are prefixed with `healthshop`. For example, the route for retrieving all health products should be like `/healthshop/api/healthproducts`.
- Register the routes with the Javalin server.

1.4.3 Initially, the data should be stored in an in-memory Java data structure to mock the database. Create a DAO class, `HealthProductDAOMock`, or use your preferred naming. Manage the list of health products in the `HealthProductDAOMock` as a static ArrayList or a HashMap.

The `HealthProductDAOMock` should include these methods:

```
Set<HealthProductDTO> getAll()
HealthProductDTO getById(int id)
Set<HealthProductDTO> getByCategory(String category)
HealthProductDTO create(HealthProductDTO healthProduct)
HealthProductDTO update(HealthProductDTO healthProduct)
HealthProductDTO delete(int id)
Set<HealthProductDTO> getTwoWeeksToExpire() // returns all health products
that expire within two weeks
```

1.5 Create a `dev.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, there may be various exceptions. Consider where these exceptions might occur and how to handle them. Document in your **`README.md`** file, for each endpoint, the errors you handle and the HTTP status codes you intend to return.

Example:

| HTTP Method | REST Resource | Exceptions and Status Codes |
|---|---|---|
| GET | `/api/healthproducts` | |

2.2 Demonstrate how to handle some of the documented exceptions. Return an exception 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": "Not found - /api/healthproducts/34",
  "timestamp": "2023-11-02 08:57:19.373"
}
```

2.3 Implement a logger, that can write to a file, to log the exceptions.

- Log the HTTP status code and the timestamp.
- Log the http request method and the path.
- Log the IP address of the client See docs here.
- Document in your `README.md` file how you implemented the logger with path to the logfile(s).
- If you fail to create the file logger, you can send the above data to the json response properties instead.

## Task 3: Streams and Generics

3.1 Create a method in the `HealthProductDAOMock` class that use the `stream API` and `filter()` to return a list of health products with a calorie count less than 50.

3.2 Create a method that utilizes the `stream API` and the `map` function to convert a list of `HealthProductDTOs` to a list of Strings containing the product names.

3.3 Use the `stream API` with groupby to group health products by category.

- Get the total price of each category.

### Introducing Generics

3.4 Create an interface named `iDAO` using generics, so it functions with `HealthProductDAOMock`. Refactor `HealthProductDAOMock` to work with `iDAO`. The interface should provide methods for: `getAll`, `getById`, `create`, `update` and `delete`.

Example:

```
Set<T> getAll()
...
```

## Task 4: JPA

4.1 Establish a `HibernateConfig` class with a method that returns an `EntityManagerFactory`.

4.2 Implement a `HealthProduct` entity class with the following properties: `id`, `category`, `name`, `calories`, `price`, `description` and `storage`.

4.3 Implement a `Storage` entity class with properties: `id`, `updatedTimeStamp`, `totalAmount`, `shelfNumber`, and a `OneToMany` relationship to `HealthProduct`.

4.4 Create a DAO class `storageDAO` using JPA and Hibernate. The new DAO should implement `iDAO` and include additional methods:

```
void addProductToStorage(int storageId, int productId)
Set<HealthProduct> getProductsByStorageShelf(int storageId)
```

4.5 Create a Populator class and populate the database with health products (according to the table a the start of this exercise). Add 2 storage objects with 2 health products each.

4.6 Modify the HealthProductController to persist data in the database.

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

## Task 5: Testing REST Endpoints

5.1 Create a test class for the REST endpoints in your `healthproductroutes` file.

5.2 Set up `@BeforeAll` to create the Javalin server, the `HealthProductController`, `healthproductroutes`and the `EntityManagerFactory` for testing.

5.3 Configure the `@BeforeEach` and `@AfterEach` methods to create the test objects (HealthProducts and Storage).

5.4 Create a test method for each of the endpoints.

5.5 Explain the differences between testing REST endpoints and the tests conducted in Task 5 in your **`README.md`** file.

Please tailor the content, images, and examples as necessary for the health product domain. If you need any specific modifications or details, feel free to ask.