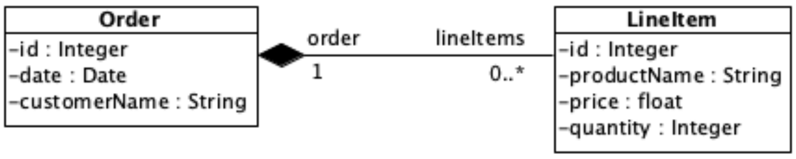
Exercise 6: Spring Boot REST Controller

Presentation of progress and results in next two labs /
Submission in AULIS until 11.12.25, 9:45am

Assignment: Development of a Spring Boot REST Controller

This assignment is about extending our order system by designing and implementing a Spring Boot REST controller in the clean architecture's *interface adapters* ring. The REST controller will provide access to two resources and make use of the application logic implemented in the existing order service:



The assignment is subdivided into four subtasks.
The results of the subtasks shall be submitted as *one* upload to AULIS.

Preparation:

Set up a new Spring Boot project, e.g. using the *Spring Initializr* (https://start.spring.io).

- Choose a *Maven* project, Java (or Kotlin), and the current version of Spring Boot (4.0.0 at the time of writing)
- Supply your project metadata:
    - Group: package name without project base package, e.g. de.hs-bremen.mkss
    - Artifact: base package name, e.g. orderService or shopBackend or …
    - Name: Name of your project
    - Description: Short description
    - Package name: (results from *Group* and *Artifact*)
    - Packaging: jar
    - Java: java version of your choice (I currently use 21)
- Specify the following dependencies:
    - Spring Web
    - Spring Data JPA
    - H2 Database
    - (Depending on your implementation you might need further dependencies.)
- Generate the project, extract the generated zip file, import the project into your IDE, and build the project.

Subtask 1: Persisting Orders and LineItems

Setup a database for persisting *Order* and associated *LineItem* objects. Insert test data into the database using a repository interface. Verify that persisted objects can be retrieved from the database with intact associations.

Hints:

- Use a relational database, e.g. H2 (in-memory or file-based)
- Use Spring Data JPA for
    - mapping objects and their properties to the database (JPA annotations), and
    - persisting objects to / retrieving objects from the database (e.g. using *JpaRepository*)
- Definition of the database schema in SQL (confer *schema.sql* in PetClinic) is not required, but common practice. JPA can create the schema based on annotations in your domain objects (have a look at property *spring.jpa.hibernate.ddl-auto* in application.properties).
- Definition of sample data in SQL (confer *data.sql* in PetClinic) is not required, but recommended if you define the database schema using a *schema.sql* file. If you decide to let JPA create the schema, sample data may be provided *in code* using a @Configuration component which creates data and inserts these into the (injected) repository.

Subtask 2: Design and Implementation of REST API

Our online shop would like to expose a REST API to its clients. Design and implement a REST API ("restful" resource paths and appropriate HTTP methods) for

- Retrieving all orders (including associated line items)
- Retrieving an order with a given id
- Retrieving all line items of an order with a given id
- Creating a new order (with the customer's name)
- Adding a line item to an order
- Removing a line item from an order
- Deleting an order.

The REST API needs to process and return JSON documents.

You might have to extend your existing order service with additional methods.
The implementation of HATEOAS is not required.

Test your API using the tool Postman (https://www.postman.com). Submit the resulting Postman collection of REST requests as part of your solution.

Note: In the previous assignment you were asked to design a clean architecture which would return the results of service (use case) calls via an OutputBoundary interface. This, however, would require the REST controller to be somehow split into two parts (one for receiving a REST call and invoking a service and another for receiving the service's result via the OutputBoundary and formulating a REST response). Therefore, you may change your clean architecture such that the services directly return their results, i.e. you may delete the OutputBoundary. Make sure that at least your CLI still works properly in parallel with the REST API!

Hints:

- Implement a @RestController component.
- Have a look at the slides and potentially do further research on appropriate resource paths and HTTP methods.

Subtask 3: Order Status and Committing Orders

Of course, the online shop wants to sell stuff – being able to purchase an order therefore is a must!

a) Add a property holding an order status to the *Order* class. An order always has one of the following statuses:
   - EMPTY (initial status: no line items / “empty shopping cart”)
   - IN_PREPARATION (at least one line item, but not purchased)
   - COMMITTED (purchased order; order not processed by warehouse)
   - ACCEPTED (order has been processed and accepted by warehouse)
   - REJECTED (order has been processed and rejected by warehouse, e.g. due to insufficient stock)

   You may implement the property either using a Java *enum* or a plain String attribute. If you are using an enum, you may want to use a @Converter component that implements the *AttributeConverter* interface for mapping status codes to and from the database.

b) Business rules for the order status:
   - The status may need to be updated whenever a line item is added or removed from an order (toggle between EMPTY and IN_PREPARATION status).
   - Line items may only be added or removed as long as the order has not been committed.

c) Extend the REST API (path+method) for purchasing an order. Purchasing will—for the moment—only set the COMMITTED status (and of course the order’s checkout date) *and* return the committed order. Be aware that an order may only be purchased in its IN_PREPARATION status! Argue for your decision regarding choice of HTTP method and resource path.

Test your solution using Postman! Submit the resulting Postman collection of REST requests as part of your solution.

Subtask 4: Error Handling and HTTP Status Codes

How does your REST service respond if you try to retrieve an order or a line item with an id that does not exist? What is its reaction if you try to purchase an empty order (or one that has already been purchased)?

Implement error handling including descriptive error messages and adequate HTTP status codes for at least the situations outlined above (do some research on proper ways to achieve that!). Test your solution using Postman!