**A simple system for a machine selling factory**

We started this project by setting up everything we needed. We went to https://start.spring.io/ and selected Maven as the project type, Java as the language, and selected the latest Spring Boot version, 3.2.0. Next, we added some parts to the project:

* Spring Web: Creating web apps and services that use REST, a way of talking between computers.
* Spring Data JPA: This helps work with databases and saves a lot of time.
* H2 Database: A simple in-memory database, great for testing.
* Spring Boot DevTools: Make development faster, especially when we need to restart the app or see changes live.
* Lombok: A cool tool that makes Java programming less repetitive.
* Validation: To check whether the data in the app is correct.
* Spring Security: To keep the app safe and secure.
* Actuator: This helps us see how the app is doing and how we can manage it better.
* Flyway: For when we need to change the database, but still keep everything in order.

After setting up these parts, we opened the project in IntelliJ, started writing and managing the code.

We then created a UML diagram, to visually plan the project. We used PlantUML, a plugin that we found in IntelliJ, it generates a UML based on the text you type. We started by adding all necessary parts of the project: Customer, Address, Order, Machine, Subassembly and Part. We then proceeded to pair them up and give them a relationship. Here's how we planned them:

* Customer class: This has a customer ID, name and email address. It is linked to orders and addresses.
* Address class: This includes an address ID, street, city and zip code. It is connected to customers.
* Order class: This has an order ID, date and status. It is linked to customers and machines.
* Machine class: This includes a machine ID, type and model. It is connected to subunits.
* Subassembly class: This has an ID and name and is associated with parts.
* Part class: This includes a part ID, name, and who created it.

After creating the diagram, we made our first 'git commit', to save our progress in Git, and to track changes to code.

**Build the project:**

We focused on creating the entire customer aspect with its devices, controls, services and repository. Before moving on to the next work on the Address aspect of the project

**Dive into development**

With the plan in place, we started building the project, implementing one feature at a time, starting with the customer functionality. This meant creating all the bits and pieces (entities, controllers, services and repositories) that would handle customer information. We wanted to get this part right before moving on to the next phase, which involved creating and running tests to ensure the application behaved as expected. We first wrote the Customer entity first. Next, we proceeded to create the CustomerRepository and then started with the CustomerControllerTest.

**Test-driven development**

Focusing on TDD, we first wrote tests to guide development. This included testing the customer entity and CustomerRepository. CustomerControllerTest was particularly decisive, here we went and created several test cases to validate the functionality of CustomerController. These tests were designed to check if the application could handle various scenarios, such as finding all users, creating new customers and handling invalid customer data.

**Security and validation troubleshooting**

During testing, we discovered an issue where the application did not return the expected 400 Bad Request response for invalid customer data, instead returning a status of 201 Created. To resolve this, we revisited the validation setup in the customer entity and made sure the @Validated annotation was correctly used in the controller method. This involved double-checking the use of annotations such as @NotBlank, @Email, where we found we had the wrong import installed.

**Refinement of customer administration**

Before moving forward, we refactored the CustomerController to introduce a CustomerService class, with the business logic to improve maintainability. The service class handled all business operations, allowing the controller to focus on HTTP requests and responses.

We kept the optional <Customer> in CustomerService from the controller class, thus maintaining a clear and flexible approach to handling customer presence or absence, which was crucial for scenarios such as order handling.

To further refine the architecture, a CustomerValidation class was introduced. This class was responsible for all client-related validation logic, providing a clear separation of concerns and improving application robustness. It was created to validate new and existing customers, as well as customer identification, before any processing in the customer service. he ensures that all customer-related operations are carried out only after data checks.

CustomerControllerTest was updated to mock the service layer instead of the repository methods, aligning the tests with the controller's interaction with CustomerService. This change ensured that the tests accurately assessed the behavior of the controller with the updated service layer.

**Customer Management Completion:**

The customer management functions in the project are now comprehensive and robust. This includes creating, updating, deleting and retrieving customer data, with thorough validation controls integrated to ensure data integrity and security.

In this way, improvements in customer management used best practices in software design, consistent with the principles of clean architecture and effective testing strategies. With this, the customer components were complete, and the focus shifted to developing the Address class. This new component will manage address-related information, essential for comprehensive customer management.

**Troubleshooting and problem solving**

We encountered a challenging situation with the 406 Not Acceptable error during the integration testing of the createCustomer endpoint. This necessitated a deep dive into the controller's configurations and request handling mechanisms.

Key actions and insights

• MediaType configuration: We adjusted @PostMapping annotations and CustomerController to explicitly produce defines = MediaType.APPLICATION\_JSON\_VALUE. This ensured clarity in the response type expectation.

• Request and response compatibility: We examined the serialization and deserialization processes. Ensuring compatibility between the request text format and controlling expected format was necessary to resolve the mismatch that led to the 406 error.

• Test refinement: The integrated tests were refined to use MockMvc exclusively, consistent with the controller's primary interaction mode. This approach is consistent in testing HTTP request/response scenarios.

**Review of program configuration**

We have also reviewed the application's configuration settings, special data sources and JPA settings. Ensuring these configurations were optimal for our in-memory H2 database was essential for seamless operation.

**Comprehensive testing strategy**

Testing played a central role in this phase. We used a lot of unit and integration testing to identify and fix issues.

• Unit Testing: Focused on individual components to ensure they function properly in isolation.

• Integration Testing: Aims to test the interaction between components and the overall behavior of the system.

**Troubleshooting and problem solving - detailed handling and solutions**

Solves 406 does not accept errors

1. Explicit MediaType configuration:
   1. The 406 error was fixed by specifying produces = MediaType.APPLICATION\_JSON\_VALUE in the @PostMapping annotation. This provided the expected response type to the createCustomer endpoint.
2. Improved handling of requests and responses:
   1. Fixed serialization issues by ensuring that the format of the JSON sent in the request matches the expected format to be checked.
   2. Performed extensive testing to verify compatibility of request and response formats.
3. Add the password criteria:
4. from the InvalidCustomerDataException that sent an http status of not accepted, by changing it to bad request and adding a message string to send why it was thrown, we found that it was because the validation asked for a strong password and it was not one. as a criterion, adding it solved the problem.

**Addressing Entity State Management and test errors**

1. Limited unit administration in service layer:

Fixed issues related to Hibernate device state management by adjusting service methods for updating and adding addresses. This involved proper handling of loose entities and conflicts such as multiple representations of the same entity.

1. Test adjustment with controller logic:

Updated test cases to reflect the actual logic in the controller. For example, adjusted tests to delete a customer by email according to the controller's method, instead of using an ID.

1. Password management in tests:

Ensured correct handling of BCrypt hashed passwords in tests. This involved using plain text passwords in test cases that match the hashed versions stored in the database.

1. Troubleshooting and log analysis:

Leveraged verbose logging to diagnose issues such as NullPointerExceptions and authentication failures.

Performed step-by-step analyzes of test errors, which led to adjustments in test setup and data conditions.

1. Comprehensive testing strategy

Adopted a rigorous testing methodology, using both unit tests and integration tests to uncover and fix problems.

Iteratively refined tests to reflect the actual operating scenarios of the application.

Used MockMvc extensively to test HTTP request and response scenarios, to ensure consistency and reliability in tests.

**Manual testing:**

In addition to automated tests, we also performed manual testing with tools such as Postman. This helped us verify the application's behavior in a more interactive way and provided a clearer understanding of how the endpoints responded to different requests.

**The rest of the project:**

After the customer handler was established, worked and tested both integrated and as a unit, finally with postman, we used it as a blue print to create the rest of the classes for our project, we did minimal unit testing on the rest of the classes as we used most of the same logic from the customer manager, to create the rest.

**Documentation and code management:**

Throughout the development process, we have regularly updated project documentation to reflect changes and improvements. This included clear comments in the code and ensuring that the Git commit captured each new change.