Reflection Document

To get some insight into my microservice structure I will provide the same information as I did in the documentation

Application Workflow For how the microservice application works. When everything of the data is set at the company for luxury cars, it will initiate both synchronous calls and asynchronous calls when an order from the client is sent. This order will then check if the userId exists with a synchronous REST api call and also send the OrderEvent if the user exists to the InventoryService and PaymentService with asynchronous event driven communication using RabbitMQ. In the inventory we will see if there is stock for the productId given by the order, and then return true if we do have stock as well as remove one from the quantity for that product. This inventory service also sends an event to the product service and updates the availability of the product to false if we are out of stock. The orderevent is also sent from the order to the payment, the payment does a rest call to the inventory service with the productId given and checks for stock with a REST call, if we do indeed have stock we will process the payment and send it back to order. In order after payment, the order is saved as completed. The call from payment to inventory to check for stock should in theory not happen but I will talk more about this in the reflection documentation. The gateway is load balancing, health checking and has centrally configuration.

AMQPOrder Service, Manages order-related operations.Communicates with the Inventory Service to check stock availability and with the AMQPPayment Service to process payments.Performs REST calls to the Springtest (User Service) to validate the legitimacy of the userId associated with each order.

AMQPPayment Service. Processes payment events related to orders. Assesses if users have sufficient funds and communicates the outcome back to the AMQPOrder Service. Makes REST calls to the Inventory Service to confirm stock availability before processing orders.

Gateway Service. Acts as the primary entry point to the microservices ecosystem. Routes requests to the appropriate microservices, managing load balancing and service discovery. Implements global configurations such as CORS settings and common filters like retry logic.

Inventory Service. Responds to order events by checking stock levels. Adjusts stock quantities based on order details and informs the AMQPOrder Service about stock availability. Sends events to the ProductService to update real-time product availability.

Product Service. Represents the actual products, mainly cars, in the system. Receives updates from the Inventory Service to reflect real-time availability of products.

Springtest (User Service). Provides functionalities for creating and managing mock users.Ensures that user IDs provided in orders correspond to actual, legitimate users in the system.

The user stories from the arbeidskrav:

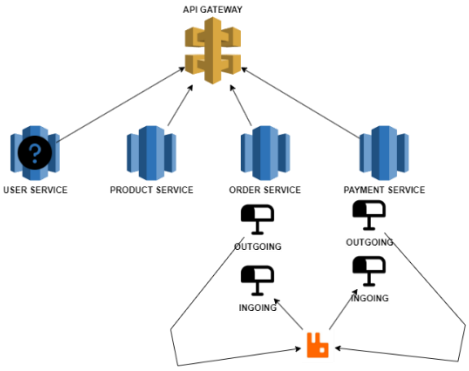
As a potential car buyer. I want to view a list of available cars. So that I can select a car to purchase.

As a registered user/mock user. I want to be able to place an order for a selected car. So that I can initiate the car purchase process.

As a user placing an order/mock user. I want to be notified if my order is successfully processed or canceled. So that I am aware of the status of my car purchase.

As a user making a purchase/mock user. I want to have a receipt for my completed purchases. So that I can keep a record of my transactions.

All of these are implemented, but the details for them could be nicer in the frontend. And this was my expectations for the project in the arbeidskrav



while this was how we thought the project would look like with a doubt if we would implement the user service or not, we both added the user service, made a inventory service, and made the inventory service communicate with the product service. The product service and the order service also have functionality for adding equipment with some small updates. So I would say the project was a success, the order can already have equipment when saved, but this is not properly handled in the inventory and product yet. But the main order of a car works as intended. The only challenge with the project was keeping the services decoupled, some docker related issues on my pc specifically, and the implementation could have been better between the communication from order to inventory and payment. At the moment inventory and payment listen to the same order queue, so the payment does a rest call to inventory to check if we have stock, it would be better for this structure to work with order being sent to inventory, inventory replying back to order, order sending the processed event to payment and payment sending it back to order. The reason this was not done was because the inventory was neither in the mvp or the realistic scenario but rather in the best case scenario, meaning it was made after the order and payment communication was made already. Meaning if we did implement it the way I want to have it now, it would cause a huge refactoring of the application and change a lot of the main functionality designed. If I would continue to work on this project that's what I would refactor, and I would create a user service that isn't just mock users.