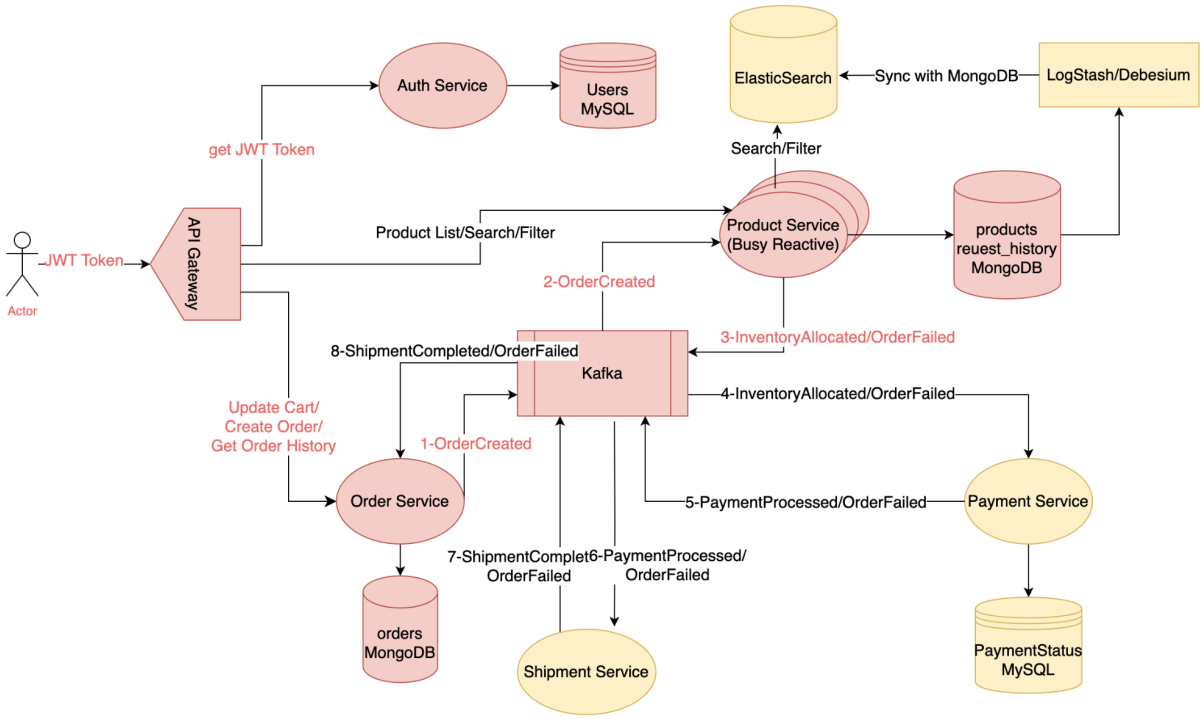
1. High Level Design

Planned Design is as Follows. As of today Pink Parts are implemented. Yellow parts are to be implemented.



1.1. Design Decisions

1.1.1. Architecture

- **Microservice Architecture** is selected for scaling heavy tasks independently from the others.
- **Product Service** will be used to display, search, update product info and store inventory information (It would have been better to have a separate Inventory Service but I combined Product and Inventory services to make implementation easier at this step).

  It will be the **busiest** service. That's why I preferred **Reactive approach (Webflux)** to prevent requests from blocking the OS threads, so that we can maximize the cpu utilization and maximize the number of concurrent requests it can handle.

The service will be stateless. So that we can safely **auto scale** it **horizontally** during peak loads to perform **load balancing**.

Also since the amount of data to be stored by the Product Service will be large, we will need to have a DB that is suitable for **horizontal scaling** as well. So we preferred (No SQL) **Mongo DB** for the DB side.

I selected the product id and brand (assume that includes product category info) fields as the **sharding key** to distribute products evenly to different nodes. If we selected the category alone as the sharding key it might have created hotspots.
In this approach searches by product category won't be efficient since products with the same category will be on different nodes. Traversing shards and combining the results will increase latency. So, I plan to perform category like searches using **ElasticSearch**.

MongoDb will be used to display the details of products by product Id, and it will be used as the source of truth for ElasticSearch. **ElasticSearch will be populated from MongoDb** using technologies like **Logstash/Debesium**.

For **Order Service** I've chosen **MongoDDB** to store Order and OrderItems together in a document. If I had chosen SQL I would have to perform a join of Order and OrderItems tables to get a complete Order. Considering OrderItems table might get large eventually, it seemed to be a good idea to store OrderItems within their Order Document using Mongo. I didn't select Reactive Mongo and WebFlux for this service since this service won't be as busy as Product Service, and it is easier to implement with plain Spring Boot.

Since **Payment** is the most crucial step which might need Atomic operations, I will go with **SQL** Database for this service.

Shipment service can also be **MongoDB**.

1.1.2. Order Processing

When performing Order Processing steps using Order, Product/Inventory, Payment and Shippinng Microservices, we need to coordinate success and failure cases of these services to ensure **consistency** of the system about the **status of an order**. Saga pattern is used for this purpose. There are two alternatives to implement this pattern:

Orchestration

Needs a **centralized service** which tracks and logs each step of the process. If one of the step fails (Payment failure etc), it performs necessary **rollbacks** performed by the other services to keep system in a **consistent** state **(order successful for all services, or order failed for all services).**

Choreography

There is **no centralized service**. Each service performs its own part and emits an event to tell its state to the **next service/s** in the chain. When one of the services fail to perform its part (eg. Payment Service fail to process payment), it emits a **failure event** so that previous services in the chain do their **rollback**.

Fault tolerance/Consistency:

A service might **update its database** but might **crash before emitting its event**. When it restarts, it should emit the event or it should rollback its changes and send a failure event. For this, we need a table where we can log the state of our steps (**Transaction Log**: DB_UPDATED, EVENT_PUBLISHED etc), and **retry/rollback** them when possible.

We should log each transaction and order processing step atomically to the DB. Atomicity of db+Kafka operations together is not possible. We can use Outbox pattern (Debesium) to solve this problem. I haven't used this technology yet, and I don't know if it's mature. I might try it in the future. For now we **ignore db-kafka inconsistency** possibility.

In Choreography all services have to perform fault tolerance/consistency checks themselves. Orchestration moves these complex parts to a dedicated service and makes the implementation of other services easier.

I've chosen Choreography pattern since it seemed to be easier at the beginning (requires less event collaboration). But when implementing the Product Service I realized that I am going to solve the same fault tolerance/consistency problems at all services with different technologies. For Product Service using Reactive Mongo, for Order Service: JPA Mongo, for Payment: SQL. In Orchestration, implementing the logic once with 1 technology stack (probably SQL) would be sufficient.

Implementation of Order Processing is not finished yet. After finishing it with Choreography, I might consider implementing the Orchestrator approach and compare them.

1.1.3. Events

- OrderService: tx: OrderCreated
  rx: ShipmentCompleted
- ProductService: rx: OrderCreated
  tx: InventoryAllocated
- PaymentService: rx: InventoryAllocated
  tx: PaymentProcessed
- ShipmentService: rx: PaymentProcessed
  tx: ShipmentCompleted
- All Services rollback their operation, if they receive OrderFailed for an order

1.1.4. Deployment

- A **docker-compose** file is available at **OrderService/compose.yaml** for running **kafka, mysql, mongo and mongo-express**.
- I plan to deploy the services to Kubernetes using minikube first, then AWS (Not implemented yet).
- For local development I will try to provide a docker-compose file for all services.
- For now all microservices share the same MySQL and Mongo instances. I will try to create distinct instances for each service when docker-compose file for the services is ready.

2. Project Status

Api-Gateway:

- **Authenticates** user using the Auth Service.
- Gets **JWT** from Auth Service and sends back to the Customer.
- Performs **authentication** of logged in users by **validating their JWT.**
- **Redirects** incoming traffic to related services.
  Todos:
- Currently users send their passwords in plain text. I will **enable HTTPS** for this service. Communication of API gateway to other services will still be HTTP.
- **Routes** for order, product, payment, shipment services will be added. For now only Auth Service routes are available.
- JWT token of the user will be **forwarded** to other services in request header, so that services can utilize Spring Security features to perform Role based authorization.

Auth Service:

- **Registers** new users with username and password.
- Saves their **passwords** in **encrypted** format.
- **Authenticates users** by comparing their password hashes with the ones in the DB (performed authomatically via **Spring Security**).
- Creates **JWTs** for all authenticated users.
- JWT includes **Role** info (USER or ADMIN).
- Provides an API endpoint for the **Public Key** which is used to sign the JWTS. All services can get this Public Key from Auth Service and validate user JWTs themselves. When Auth Service is restarted other services should be **notified** to update their Public Key cache (we can use Kafka for this).
- Has **Role Authorization** implementation example (PUT /users mapping):
    - Users are not allowed to change other user's info.
    - Users are not allowed to change their Roles.
    - ADMIN users are allowed to change other user's info and Roles.

Order Service:

- Implemented **CRUD** operations using JPA Mongo .
- Implemented **Kafka** Integration.
- Implemented **Shopping Cart**.
- An **order is created** with the contents of Shopping Cart and Cart is **cleared**. And an ORDER_CREATED event is published using Kafka.
-

Product Service:

- Implemented **CRUD** operations using Reactive Mongo
- **Kafka** Integration implemented
- Implemented processing **ORDER_CREATED event**. Reception of event, and the result of processing is logged to DB (to "**inventory_requests**" collection). This info will be used to **prevent duplicate processing of the same order** if Kafka Broker or Product Service crashes before we ack the Kafka for the processed message (to ensure **idempotency for exactly once delivery**).
  Todos:
- This service is not tested yet
- Mongo is said to provide **Atomic updates** of two documents. I needed to use this mechanism to allocate the request quantities from **products** collection and to log the status of the request to **inventory_requests**, **atomically**. I need to check

documentation to learn how to do this. For now I've just marked these methods with @Transactional annotation.

Todos:

- **Product search** features are not implemented yet. **ElasticSearch** will be utilized for this.

Payment Service

- Not Implemented
- I will implement using SQL
- It will randomly return payment success and fail events for the incoming requests

Shipment Service

- Not Implemented
- I will implement using Mongo
- It will randomly return shipment success and fail events for the incoming requests

Unit and Integration Tests

- Implemented only for Order Service (I had a problem with Kafka TestContainer, I will fix it)

3. Running

- There is no front end. To run, you will need **Postman/Thunder** etc.
- Until I provide the docker-compose, you should download the codes and run each with: **mvn spring-boot:run** (requires **mvn** and **java 22**).
- I provide **Swagger API** for these services:
  - Order Service: http://localhost:8083/swagger-ui/index.html
  - Product Service: http://localhost:8082/swagger-ui/webjars/swagger-ui/index.html
- For API and Auth Services use the informatio in the Service Resources section below for now.

3.1. Service Endpoints:

- Api Gateway Service: http://localhost:8080
- Auth Service: http://localhost:8081

- Product Service: http://localhost:8082
- Order Service: http://localhost:8083
- Payment Service: http://localhost:8084
- MongoExpress: http://localhost:28017
- MongoDB Port: 27017
- MySQLPort: 3306
- Kafka Port: 9092

3.2. Service Resources

Api Gateway (8080) & Auth Service (8081):

- **Create A New User**
  http://localhost:8080/users
  http://localhost:8081/users
  POST {
  "username": "user",
  "password": "passwd",
  "role": "ADMIN",
  "email": "user@demirsoft.com",
  "address": {
  "state": "turkiye",
  "city": "istanbul",
  "street": "kosuyolu"
  }
  }
  Returns a **JWT**

- **Login User With JWT**
  http://localhost:8080/login
  http://localhost:8081/login

  POST: {
  "username": "user",
  "password": "passwd"
  }
  Add To Header
  key=Authorization value=Bearer <JWT copied from previous step>
  Returns a JWT

- **Update A User**

http://localhost:8080/users
http://localhost:8081/users
POST {
  "username": "user",
  "password": "passwd",
  "role": "USER",
  "email": "user@demirsoft.com",
  "address": {
    "state": "turkiye",
    "city": "istanbul",
    "street": "kosuyolu"
  }
}
Returns a JWT

- **Get Public Key to Parse JWTs**
  http://localhost:8081/public-key
  GET
  Returns Public Key

- **For Order And Product Services Use Swagger UIs:**
  - Order Service: http://localhost:8083/swagger-ui/index.html
  - Product Service: http://localhost:8082/swagger-ui/webjars/swagger-ui/index.html

4. Table Indexes

Auth Service (MySQL):
    User : id, **username**
Order Service (MongoDB):
    orders(id, **customerId**)
    carts(id, **customerId**)

Product Service (MongoDB):
    products(id)
    Shard key(id, **brand**)

5. Todos

- Product Search (ElasticSearch)
- Payment and Shipping Services
- Deployment to Kubernetes
- Fault Tolerance (Retry configs for REST/Kafka calls), Load Balancing Impl
- Unit/Integration/Contract Tests
- Refactoring (Get hardcoded values from config files etc.)
- Tracing (OpenTelemetry)
- Oauth2 Resource/Authorization Server Mechanism
- Kafka/MongoDB clusters
- Caching Before Product and User Services (Redis Cluster)