Bloq.it Challenge Report

Approach

After reading the challenge, I thought about some topics

- the role's requirements
- previous interviews being focused on scalability
- scalability and agility in a growth environment
- personal experience using smartlockers

I imagined myself as responsible for starting from scratch and performed these steps:

Defined what matters.

a. Making it clear I can provide real value as a team member.

b. Showing technical prowess.

c. Demonstrating how I would tackle this challenge in the real world, and just to look good in the challenge.

d. Having fun and understanding myself better.

Defined key requirements.

1. Creating a Rent
2. Dropping it off
3. Picking it up
4. Confirming delivery

Defined my strategy.

1. Leverage microservices for a decoupled, modularized and organized structure that wouldn't compromise application and team scalability.
2. Using docker to standardize and facilitate... everything.
3. Creating a closed network of services with a single gateway entrypoint - there will probably be many different clients interacting with the API, removing complexity removes hurdles.
4. Keeping the code modularized, but not over-engineered to promote the solution's plasticity over faultless idiomatic implementation.

Implementation

Entities, Moving Parts, Containerization

There are 3 core services, defined by the core entities - *Bloq, Locker, Rent*.

Each services has its own mongo instance.

Each service is containerized using Docker Compose.

There’s a single API entrypoint to the network.

API Gateway

I knew I wanted a single entrypoint, so I thought of using Ingress with Kubernetes, Nginx and Kong.

After wasting some time over analysing the pros and cons, I decided that I had to stay true to my plan.

I truly believe that over committing early in the development is a shot in the foot, so I made the decision to make this project’s philosophy to prioritize flexibility and leanness over perfect implementation.

I wanted to build a simple starting point, with all its initial moving parts, in an architecture that could adapt to any implementation.

More permanent choices will become aparent as development continues.

So I built a simple gateway in node that would route requests to the network.

Services

Each service has an *index.ts* file to boot up the server, a Dockerfile and its own environment variables.

The directory is divided into:

- **Controllers** - handling the route’s logic
- **Models** - defining schemas
- **Routes** - connects controller’s methods to routes
- **Types** - types and enum definitions

Communication

I wanted to use rabbitmq and built a completely decoupled communication service and protocol using events.

Considering the company uses MQTT, it wouldn't be dangerous to embrace it, but following the development philosophy and considering a great leap in testing complexity, I decided to keep it simple and route the logic to good old HTTP requests.

I also wanted to avoid bloating the project with massive, robust tools - Kubernetes was ditched because of this, the only exception is Docker.

Business Logic

I started by defining and implementing the core actions defined in the project scope. After drafting on paper and a few Lidl visits to check on their smartlockers, this is what I came up with.

Create a Rent

A client like Vinted or DHL uses its proprietary client to attempt to create a new Rent.

It requests the gateway to handle the data it provides to create a new Rent, and waits for a response.

1. Data is validated and the Rent is saved → let's see if we can drop it off
2. Bloq service, what is the nearest Bloq? → are there any available lockers?
    a. Bloq service receives an address and uses google maps API to convert to coordinates and calculates the nearest distance (planned but didn't implement)
    b. Depending on performance requirements, it would be wise to have a separate data structure holding this pre-processed coordinate data
    c. Returns the closest Bloq id
3. Locker service, can you provide me with an available Locker? → Mark it as ***occupied*** (I treated "occupied" === "allocated", not as "Locker has no Rent inside it"
    a. Fetch Lockers from db with the provided Bloq id

    b. Send it back to Rent service
4. Update Rent in the db as waiting drop off.
5. Notify the user and provide a code for authentication, sync data with a hypothetical tracking service. (planned but didn't implement)
6. Send the success response back!

Drop Off - Pick Up

A person arrives at the Bloq, scans/enters the code and drops off the Rent.

Behaviour for Pick Up replicates Drop Off. There is some redundant code that could be more reusable, but since it is expected that dropoff/pickup behaviour will diverge as development continues, I decided to keep them separate.

1. Fetch and validate the Rent as ready to drop off → Locker handles the situation
2. Estabilish a connection to the IoT device and pops open.
3. Awaits Rent to be delivered, succeeds if it's dropped off and user closes the door, fails if it times out.
4. Respond accordingly. If success, update data and notify accordingly.

Testing

Initially, I set up jest for automated tests, but as time went by I realized I had to choose between mediocre tests on jest or decent testing on Postman, so I kept using Postman, which I had been using along development. This wasn't due to simplicity, only time constraint.

I have provided a Postman file, aswell as easy mock data to test, based on what was provided in the challenge.

It's organized by endpoint, and has 3 extra methods for convenience:

- **Find All** - Shows every document in an entity's collection
- **Bulk Save** - saves mock data

- **Delete All** - Deletes every document in an entity’s collection

Documentation

Setup and Dependencies

You need Node and Docker installed on your machine.

While on the root directory, simply run

```
docker-compose up --build
```

API Endpoints

For your convenience, I left the individual services exposed in the Docker Compose file.

Functionality was planned so that all endpoints are handled by the gateway.

Bloq

```
// Find closest Bloq by address
GET http://localhost:3003/api/bloqs/findClosest/:address

// Utility
GET http://localhost:3003/api/bloqs/findAll // retrieve all reco
POST http://localhost:3003/api/bloqs/bulkSave // Save mock recor
DELETE http://localhost:3003/api/bloqs/deleteAll // delete all r
```

Locker

```
// Find available Locker given a Bloq id
GET http://localhost:3003/api/lockers/findAvailableByBloqId/:blo

// Handle drop off
```

```
GET http://localhost:3003/api/lockers/dropoff/:lockerId

// Handle pick up
GET http://localhost:3003/api/lockers/pickup/:lockerId

// Allocate a Locker
PATCH http://localhost:3003/api/lockers/allocateLockerById/:locl

// Free a Locker
PATCH http://localhost:3003/api/lockers/freeLockerById/:lockerId

// Utility
GET http://localhost:3003/api/lockers/findAll // retrieve all re
POST http://localhost:3003/api/lockers/bulkSave // Save mock rec
DELETE http://localhost:3003/api/lockers/deleteAll // delete all
```

Rent

```
// Create new Rent and prepare for drop off
POST http://localhost:3003/api/rents/create/:address

// Attempt Rent drop off
POST http://localhost:3003/api/rents/dropoff

// Attempt Rent pick up
POST http://localhost:3003/api/rents/pickup

// Utility
GET http://localhost:3003/api/rents/findAll // retrieve all reco
POST http://localhost:3003/api/rents/bulkSave // Save mock recor
DELETE http://localhost:3003/api/rents/deleteAll // delete all r
```

The Good, the Bad, the Future

The Good

I'm really happy for acting on the things that matter instead of rushing to build an amalgamation of tools.

It really went against my nature to compromise on super careful implementation in order to build something flexible, simple and embryonary.

In my first years I tended to over engineer and become paralyzed, but as my past roles forced me to deliver value, I realized that when I work this way, the end result is always better - especially in the resulting code cohesion.

The challenge itself was so much fun, especially because it brings together stuff I like to build and a domain I'm very interested by.

The Bad

There are some things I would've done differently.

- I should've used Typescript's advantages more, stricter typing and validations would be much better - there are some holes, especially in input data.
- I definetly should've started testing with Jest from the beginning, testing in Postman can't compare.
- Splitting controllers into controllers + services would be wise, and would make testing much better.
- I should've normalized error handling, so that errors could bubble up and give the gateway response more expressive messages.

The Future

I had so much fun planning and carrying this out that I'm already excited about the next branches to open. This is how I would make this 1000x better:

- Normalize errors
- Make great entities with TS.

- Commit to Kubernetes and Ingress, especially because of rate limiting and throttling.
- Extract more values to environment variables and interpolate configuration files to create dev and prod deployments.
- Close the network off and build authentication.
- Integrate RabbitMQ and make communication event-driven.
- Leverage AWS with services like API Gateway, SES, and SNS.

Final Thoughts

I have the profound belief that I will become an instrumental part of the team in the years to come.

Though it may be routinary to you, this challenge and this role mean the world to me.

Thank you for the opportunity to take part, I'm eagerly awaiting to hear back from you.

Until then,

José Machado