Software Development
Group A Lyngby
11/10/2024

Systems Integration

OLA 4

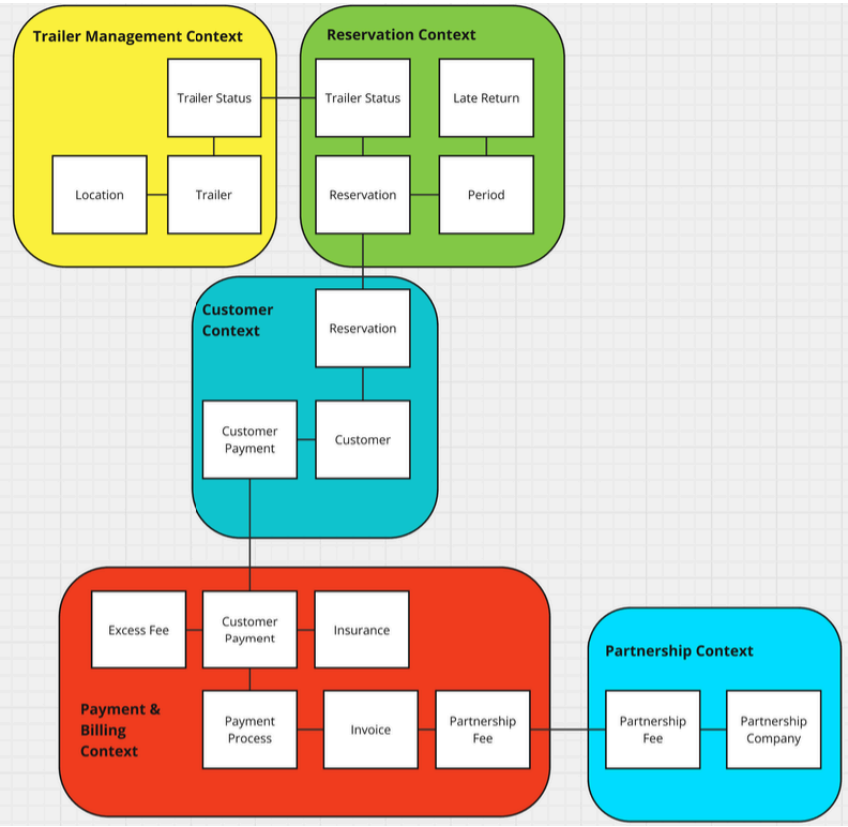
Name	Mail
Andreas Fritzbøger	cph-af167@cphbusiness.dk
Owais Dashti	cph-od42@cphbusiness.dk

cphbusiness
COPENHAGEN BUSINESS ACADEMY

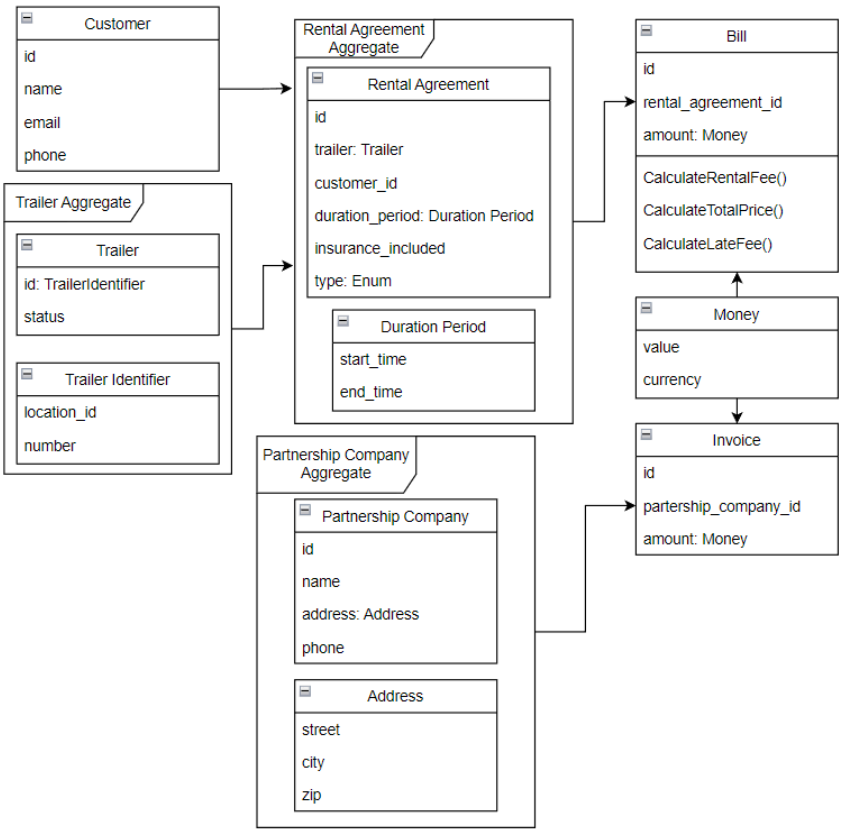
Contents

1. Introduction

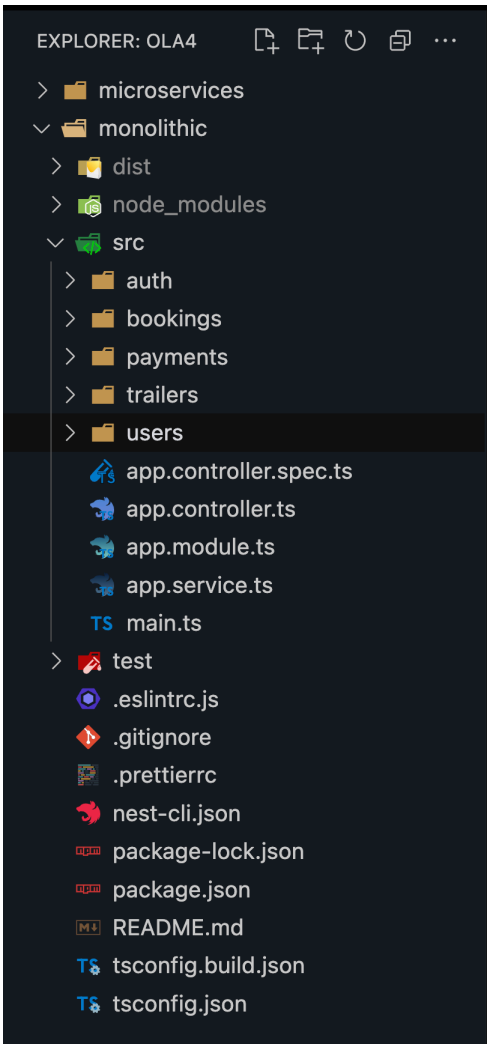
This report part 2 of the MyTrailer case study, where we in the previous part used Domain Driven Design resources and more to build models and diagrams representing the business logic. For the second part, we will now show, how we use the models and diagrams to build our application for the MyTrailer case study. We use bounded contexts from strategic design to divide domains up into contexts, we thought would fit best together.



With tactical design, we built a domain model to define entities, value objects, aggregates and relations between them in the project.



We have started building the project up based on the models and diagrams. However, not all parts in the models are used in the final product, because of changes we have made.
The most notable difference between our model and product is the entities, we have defined. In the the final product, we have four entities instead of six. The reason for this is that some of the entities have been combined, because we thought, two entities ended up being too similar. Customer and Partnership Company are both entities, which can make payments to MyTrailer, based on different circumstances. The entities have been combined to user sharing same attributes.
Bill and Invoice are two types of payment MyTrailer can receive, based on what type user the payment is assigned and from. Bill and Invoice have therefore been combined to payment.
Rental Agreement has its name changed to booking, because we agreed on the words having the same meaning in the case study, but booking is shorter and would fill less space in the code.
Below is a screenshot, which folders designated to the projects entities.



2. GitHub Repo

Link to GitHub Repository

3. Technology Stack

Current implementation uses:

- Programming language: **TypeScript**
  - A statically typed superset of JavaScript that enhances the development experience and reliability through type checking, reducing runtime errors and improving code quality.
- Framework: **NestJS** (a NodeJS framework)
  - A progressive Node.js framework designed for building efficient and scalable server-side applications using TypeScript. It promotes a modular architecture, facilitating maintainability and flexibility.
- Webserver: Underlying **ExpressJS** server
  - The underlying web server that provides a minimal and flexible Node.js framework, allowing for the development of robust APIs and web applications.

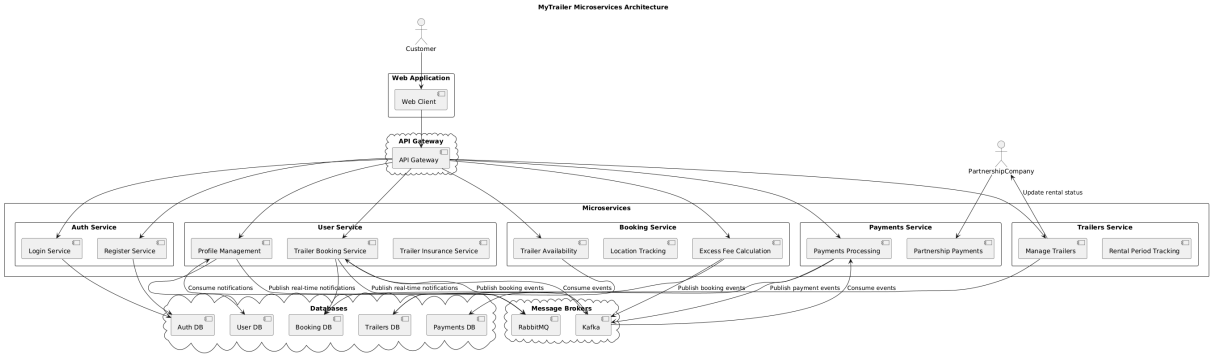
Additions in further development:

- Databases:
  - **PostgreSQL**: A powerful relational database system known for its robustness and extensibility, ideal for structured data storage.

- **Redis**: An in-memory key-value store that offers high performance for caching.
- **MongoDB**: A NoSQL database that allows for flexible data storage with its document-oriented architecture, suitable for handling unstructured data.

- Message Brokers
  - **Apache Kafka**: A distributed streaming platform that enables real-time data processing and is perfect for handling large volumes of data efficiently.
  - **RabbitMQ**: A widely-used message broker that facilitates communication between different parts of the application, ensuring reliable message delivery.
- Containerization
  - **Docker**: A platform for developing, shipping, and running applications in containers, promoting consistency across various environments.
- Orchestration
  - **Kubernetes**: An open-source container orchestration system for automating deployment, scaling, and management of containerized applications.
- API Documentation
  - **Swagger (OpenAPI)**: A powerful tool for documenting and designing APIs.
- Testing Framework
  - **Jest**: A comprehensive testing framework that provides a rich set of tools for unit and integration testing.
  - **Artillery**: A modern, powerful, and easy-to-use load testing tool. It helps simulate real-world traffic and assess application performance under various conditions.
- Logging and Monitoring
  - **Winston**: Logging library that provides a simple way to log messages and errors in various formats.
  - **Prometheus**: Monitoring system and time series database designed for reliability and scalability.
  - **Grafana**: Visualization tool that allows for the creation of interactive dashboards, providing insights into application performance.

4. System Architecture Diagram



Link to diagram

5. Patterns Implementation

5.1. Ports and Adapters Pattern (Hexagonal Architecture)

The Ports and Adapters pattern ensures that our application is decoupled from external services and frameworks. This approach allows us to easily swap out or modify external components (like databases or message brokers) without impacting the core application logic.

- **Ports**: define interfaces that represent the interactions with external systems, such as payment processing, user auth, db access.
- **Adapters**: are concrete implementations of these interfaces that handle the communication with the specific external services, such as PostgreSQL for data storage or RabbitMQ for messaging.

This pattern ensures that the core business logic remains isolated and focused on the MyTrailer domain, facilitating easier testing and integration.

5.2. Factory Pattern

The Factory Pattern used for creation of payment objects to manage different types of payments efficiently:

- **Payments**:
  - **Invoice**: used for partnership companies. When a partnership company rents a trailer, an invoice is generated to bill the company for the service provided.
  - **Bill**: used for individual customers. After a rental transaction, a bill is generated for the customer, detailing any fees incurred.

This separation makes handling of payment types quite flexible, making it easy to add new payment methods in the future without changing the core payment processing logic.

5.3. Decorator Pattern

The Decorator Pattern adds additional functionalities to rental agreements without modifying the core classes:

- **Insurance Decorator**: This decorator adds insurance options to the rental agreement. When a customer books a trailer, they can opt to add insurance to their rental agreement. This is achieved by wrapping the rental agreement object with an insurance decorator that modifies its behavior to include insurance charges and conditions.

This approach makes it quite flexible, as new decorators can be added for other features (such as additional equipment or services) without altering existing classes.

5.4. Strategy Pattern

The Strategy Pattern could also be beneficial in our system in managing different rental strategies based on user types or rental periods:

- **Rental Strategy**: Different strategies for short-term vs. long-term rentals can be implemented. For example, a strategy for handling overnight rentals (which may involve additional steps or conditions) can be defined separately, allowing the system to switch between rental strategies at runtime based on user selections.

5.5. Command Pattern

The Command Pattern could also be used for handling operations like booking a trailer, returning a trailer, or processing payments:

- **Command Objects**: Each operation can be encapsulated into command objects, which can be queued or executed based on user actions. This allows for better management of actions and can facilitate features like undoing an action or logging operations for audit purposes.

5.6. Observer Pattern

The Observer Pattern could be used for notifying users about changes in trailer availability or rental status:

- **Notifications**: When a trailer becomes available or a rental period ends, observers (such as user notifications) can be triggered to inform users via email or app notifications, enhancing user engagement.