Bachelor thesis

Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Computer Science

Cloud-Native and Microservice Application Development

Přemek Bělka

Supervisor: Ing. Martin Komárek
Field of study: Software Engineering and Technologies
May 2024

Acknowledgements

I would like to thank my thesis supervisor Martin Komárek from the Czech Technical University for his help with the writing of this thesis. I am also very thankful to my co-supervisor Chun-Wei Tsai from the National Sun Yat-sen University for his assistance with this thesis and for extending a very warm welcome to Taiwan.

Declaration

I hereby declare that I have prepared the submitted work independently and that I have indicated all information sources used in accordance with the Methodological Guidelines on ethical principles in the preparation of university theses.

Kao-Siung, 21st May 2024.

Abstract

The thesis is concerned with an extension of a demo application built using microservice and cloud-native architecture styles on the platform CodeNOW. It describes every phase of a typical software project. Starting with analysis, design proposal, implementation, testing, and deployment. Additionally, related theory is introduced and several microservice patterns are demonstrated.

Keywords: Microservices, Event Sourcing, Domain Driven Design, Event-Driven Architecture

Supervisor: Ing. Martin Komárek
E-431,
Karlovo Náměstí 13,
12000 Praha 2

Abstrakt

Práce se zabývá rozšířením ukázkové aplikace postavené pomocí mikroslužeb a cloud-native architektury na platformě CodeNOW. Popisuje všechny fáze typického softwarového projektu. Počínaje analýzou, návrhem designu, implementací, testováním a nasazením. Kromě toho je představena související teorie a demonstrováno několik mikroservisních vzorů.

Klíčová slova: Mikroservisy, Event Sourcing, Domain Driven Design, Event-Driven Architecture

Překlad názvu: Mikroservisní a cloud-native vývoj aplikací

Contents

Figures

Chapter 1

Introduction

1.1 Aim of the Work and Text Structure

This work aims to extend an already existing demo application focused on buying bus tickets with new functionality through the addition of new components. The objective of these components is to enable payment processing within the application and the generation of example tickets in PDF format, which can later be retrieved by the user. Secondly, the aim is to introduce a contemporary approach to application development, focusing on cloud-native and microservice architecture. This serves to lay a solid theoretical foundation for understanding how the extension will be executed

The text is organized in the following way. The rest of Chapter 1 briefly introduces the CodeNOW platform and the concept of iterative development.

Chapter 2 provides an analysis of the Ticket Reservation Application in its current state, determines the exact scope of the functionality, by which it is to be extended, and then proposes a solution.

Chapter 3 explains the necessary theoretical concepts to achieve a successful extension of the application. The focus is laid on cloud-native and microservice architecture.

Chapter 4 focuses on the implementation of the newly added microservices. It describes the technologies used and key points of the implementation. Furthermore, it explains the changes necessary on the part of the other, already existing microservices.

Chapter 5 describes the testing approaches to verify the correct behavior of the newly added functionality and the system.

Chapter 6 introduces the platform CodeNOW in more detail by describing the deployment process of the MinIO service and showcasing the application of some of the principles mentioned in the theoretical chapter.

The last chapter summarises the output of this work and evaluates the fulfillment of the goals set in the beginning. It also proposes the next steps for further development.

1.2 CodeNOW

The work on this thesis was done in cooperation with the company Stratox developing the Platform as a Service CodeNOW. According to the description on the official website, CodeNOW is a platform that integrates commonly used open-source technologies for cloud-native, DevOps, and microservices development. CodeNOW allows developers to easily manage, version, and deploy their applications without the need to delve into the details of the infrastructure on which the applications are deployed. CodeNOW can also be seen as an alternative to other major Platform As A Service providers, such as Azure or AWS. The main advantage of CodeNOW over these services is its much faster learning curve and absence of vendor lock-in. [5]

1.3 Iterative Development

The work on extending the Ticket Reservation Application was done iteratively instead of proceeding in a waterfall fashion. Iterative development is a methodology, in which the development cycle is conducted in repeating iterations. An iteration in this context can be defined as: "A self-contained mini-project, with a well-defined outcome: a stable, integrated, and tested release."[15, Chapter 1]

Such iteration usually encompasses all of the usual phases of a software engineering project as seen in Figure 1.1

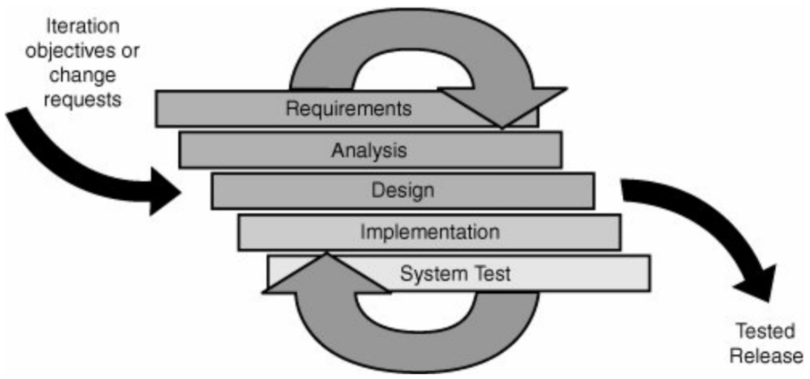


Figure 1.1: Iteration phases [15, Chapter 1]

Chapter 2
Analysis and Solution Design

The application being analyzed and developed in this text carries the name Ticket Reservation App. Its purpose is to provide an example to future users of the CodeNOW platform on how certain features of the platform can be utilized. Another goal is to demonstrate the use of several patterns and concepts, which can very often be found in cloud-native and microservice applications. This is shown in the process of searching for bus connections across cities in Europe and then creating a reservation for seats.

In the first part of this chapter, the emphasis is laid on analyzing the current state of the application. Namely the processes relevant to this work. The latter part formulates requirements for the extension of the application's functionality and then proposes a solution to fulfill them.

2.1 AS-IS State Analysis

The work on the application described in this thesis is a continuation of the bachelor's thesis "Cloud Native Application Development" by Robin Vávra. [13]. Chapter 3 of the thesis explains the transition of the application from a monolithic architecture to a microservice architecture. It provides a component diagram [13][Figure 3.18] of the new architecture. Unfortunately, this diagram does not represent reality anymore, as further undocumented development on the application has been done since the publication of the thesis. A more recent component diagram of the application's current state was provided during the writing of this thesis, which was used after several adjustments (Figure 2.1). Apart from this, no further documentation was provided, therefore, the analysis in this section is the result of reverse engineering the application's functioning from the code itself and the information available from the deployment on the CodeNOW platform.

2.1.1 Components

The application currently consists of 6 microservices divided into two applications - Reservation application and Notification application. This division is, however, only of significance for the deployment process on the CodeNOW platform and would not play any important role if the application was to be deployed on another platform. Despite that, the diagrams and the following text still considered it for completeness.

The list of microservices in the Reservation application, with a brief explanation of their functionality, is as follows.

Backend-logic microservice Contains most of the business logic regarding creating reservations. Owns the reservation entity.

FE microservice Provides a React UI to create reservations.

API microservice Is supposed to provide an entry point to the system for the FE microservice. However, it is not the only entry point, as the FE microservice also directly retrieves data from the Schedules microservice.

Schedules microservice Contains the business logic concerned with generating bus schedules and managing the number of available seats on given dates.

CJOB microservice Periodically executes a bash script that directly inserts new entities into the relevant database tables owned by the Schedules microservice, so that new seat reservations can be created in the future.

The Notification application consists of only a single microservice.

Notification microservice Enables the system to send emails without attachments.

The microservices employ an asynchronous communication style utilizing the messaging technology Kafka. The exception is the FE microservice that uses a synchronous REST API exposed by the API microservice and Schedules microservice.

The component diagram (Figure 2.1) omits the integration with old versions of the application in the backend-logic microservice, as shown in the diagram in the previous thesis [13][Figure 3.18]. This decision was made due to the lack of significance for the current thesis's scope.

2.1.2 Persistent Data Model

Only two of the microservices own persistent data (Figure 2.2). Throughout its lifecycle, the reservation entity goes through 4 states (Figure 2.3). The API microservice regularly replicates the last 50 reservations from the Backend-logic microservice.

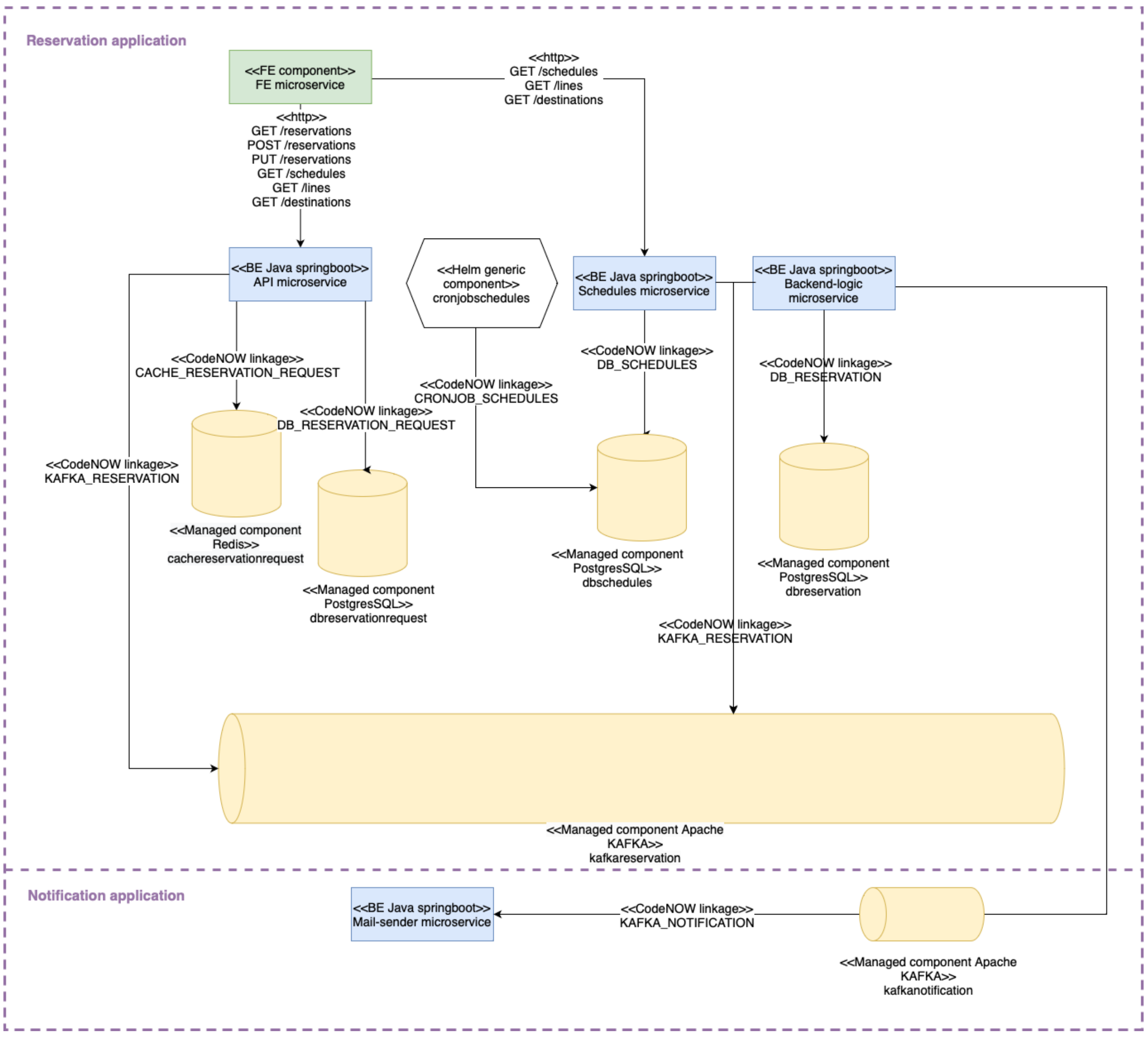

Reservation application
<<FE component>> FE microservice
<<http>> GET /schedules GET /lines GET /destinations
<<http>> GET /reservations POST /reservations PUT /reservations GET /schedules GET /lines GET /destinations
<<BE Java springboot>> API microservice
<<Helm generic component>> cronjobschedules
<<BE Java springboot>> Schedules microservice
<<BE Java springboot>> Backend-logic microservice
<<CodeNOW linkage>> CACHE_RESERVATION_REQUEST
<<CodeNOW linkage>> CRONJOB_SCHEDULES
<<CodeNOW linkage>> DB_SCHEDULES
<<CodeNOW linkage>> DB_RESERVATION
<<CodeNOW linkage>> DB_RESERVATION_REQUEST
<<CodeNOW linkage>> KAFKA_RESERVATION
<<Managed component Redis>> cachereservationrequest
<<Managed component PostgresSQL>> dbreservationrequest
<<Managed component PostgresSQL>> dbschedules
<<Managed component PostgresSQL>> dbreservation
<<CodeNOW linkage>> KAFKA_RESERVATION
<<Managed component Apache KAFKA>> kafkareservation
Notification application
<<BE Java springboot>> Mail-sender microservice
<<CodeNOW linkage>> KAFKA_NOTIFICATION
<<Managed component Apache KAFKA>> kafkanotification

Figure 2.1: Component diagram AS-IS

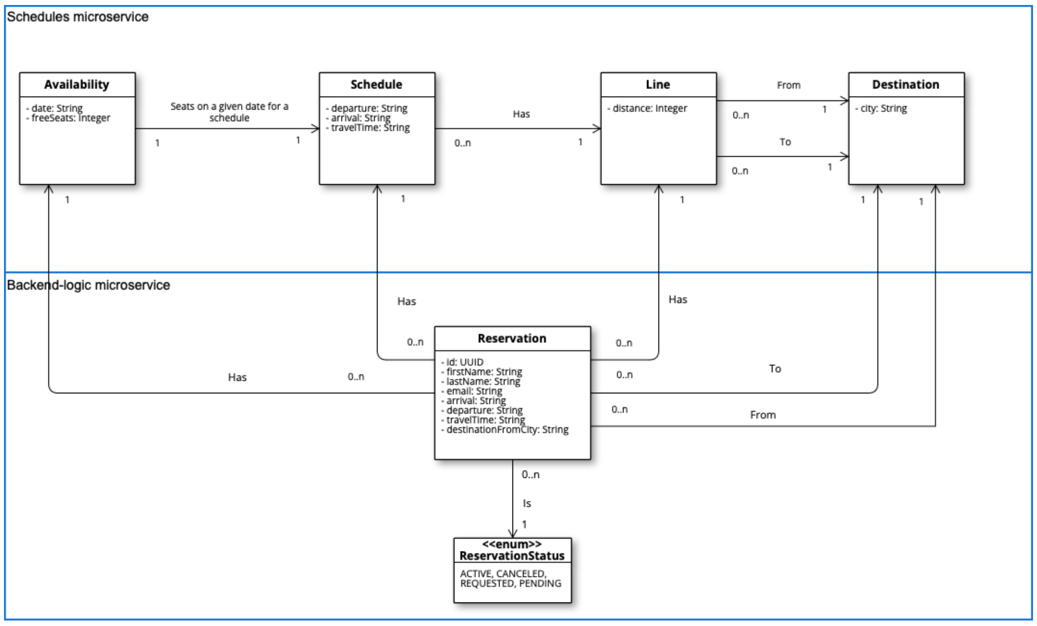


Figure 2.2: Data model AS-IS - Entity diagram

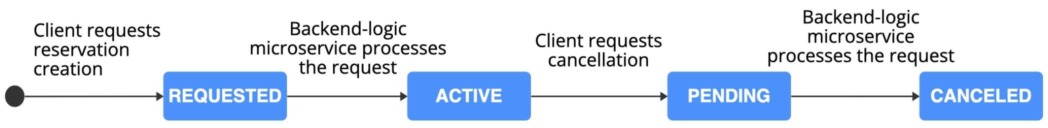

Client requests reservation creation
REQUESTED
Backend-logic microservice processes the request
ACTIVE
Client requests cancellation
PENDING
Backend-logic microservice processes the request
CANCELED

Figure 2.3: Data model AS-IS - Reservation state machine diagram

2.1.3 Create Reservation Process

The process of creating a reservation is the only process available to the user from the UI. The sequence diagram in Figure 2.4 describes the communication and operations in the system from the point of requesting a new reservation by the user.

2.1.4 Cancel Reservation Process

In the AS-IS state, the application does not expose a UI to the user to cancel their reservation, nonetheless, it contains code within the relevant services to do so.

The business logic also contains a bug where the number of available seats maintained by the schedules service is not changed upon a reservation cancellation.

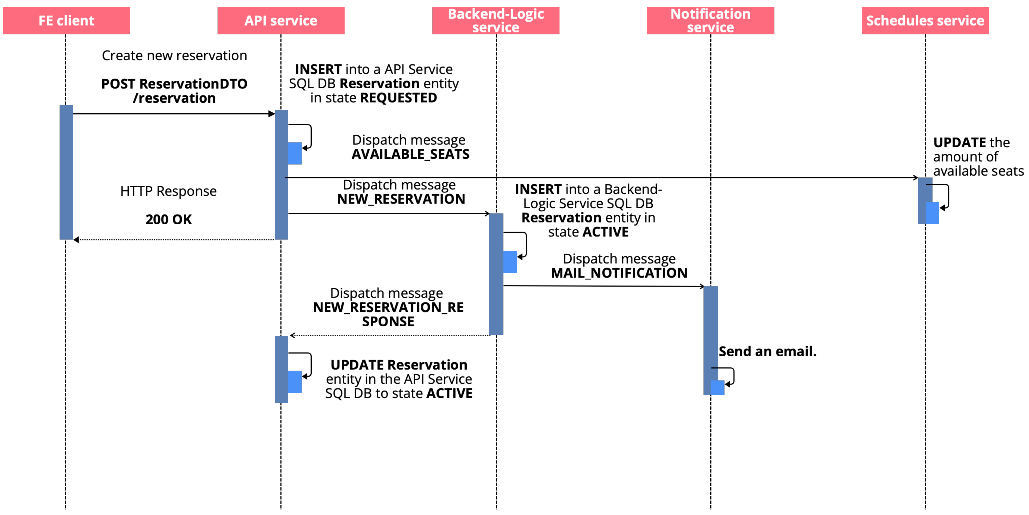

FE client
API service
Backend-Logic service
Notification service
Schedules service
Create new reservation
POST ReservationDTO /reservation
INSERT into a API Service SQL DB Reservation entity in state REQUESTED
Dispatch message AVAILABLE_SEATS
UPDATE the amount of available seats
HTTP Response
200 OK
Dispatch message NEW_RESERVATION
INSERT into a Backend-Logic Service SQL DB Reservation entity in state ACTIVE
Dispatch message MAIL_NOTIFICATION
Dispatch message NEW_RESERVATION_RESPONSE
Send an email.
UPDATE Reservation entity in the API Service SQL DB to state ACTIVE

Figure 2.4: Create Reservation AS-IS - Sequence diagram

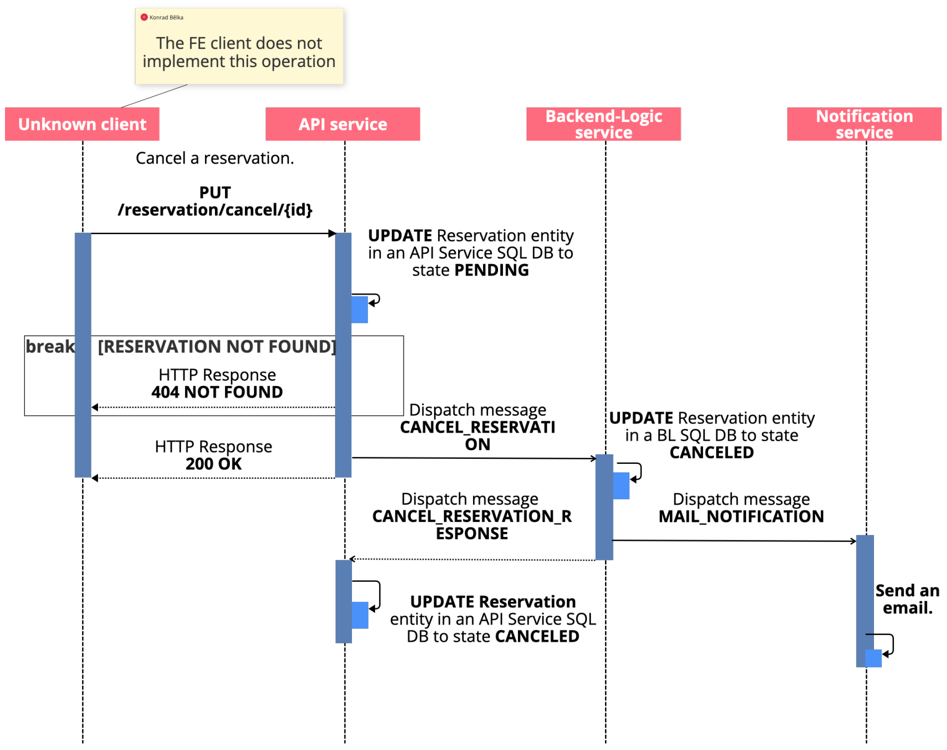

Konrad Bělka
The FE client does not implement this operation
Unknown client
API service
Backend-Logic service
Notification service
Cancel a reservation.
PUT /reservation/cancel/{id}
UPDATE Reservation entity in an API Service SQL DB to state PENDING
break [RESERVATION NOT FOUND]
HTTP Response 404 NOT FOUND
HTTP Response 200 OK
Dispatch message CANCEL_RESERVATION
UPDATE Reservation entity in a BL SQL DB to state CANCELED
Dispatch message CANCEL_RESERVATION_RESPONSE
Dispatch message MAIL_NOTIFICATION
UPDATE Reservation entity in an API Service SQL DB to state CANCELED
Send an email.

Figure 2.5: Cancel Reservation AS-IS - Sequence diagram

2.2 TO-BE State Design Proposal

The functionality specified by the thesis assignment was decomposed into two separate microservices - Credit microservice and PDF microservice.

Also, an additional integration of a Keycloak component into the system was requested as a follow-up to a previous thesis “Cloud Native Application Development” written by Alena Suvorova [18]. The thesis was concerned with the correct configuration of the Keycloak application for the Ticket Reservation Application.

This section gives an overview of the requirements for the new services and proposes a solution to achieve integration with the Ticket Reservation Application. The relevant diagrams also include the Keycloak service, however, the process of authentication and authorization is not described in detail as it was the subject of Alena Suvorova’s work and the code supporting this functionality was created by her. However, this code had to be modified, merged and manually tested by the author of this thesis.

The sequence diagrams in this section assume that an authorized user is making the requests.

2.2.1 New Requirements

Based on the thesis assignment and then further discussion with the thesis supervisor, the following functional and non-functional requirements were formulated during the iterative work on the application. The requirements determine the scope of the functionality to be added to the system.

FR01: Topping up the account balance The system must allow clients to top up their account balance using a credit card.

FR02: Money deduction from the account balance after a purchase The system must allow clients to deduct money from their account balance after a purchase.

FR03: Payment refund The system must allow clients to cancel the deduction of money from their account balance for a purchase.

FR04: Account balance retrieval The system must enable clients to retrieve their current account balance.

FR05: PDF boarding ticket generation The system must generate a PDF boarding ticket upon the successful creation of a new reservation. The PDF ticket must contain the departure date, departure time, departing destination, arrival time, arrival destination, amount of passengers, and the owner’s name. The PDF ticket must contain a QR code referring to the unique reservation ID.

FR06: PDF boarding ticket persistence The system must persist the boarding ticket after generation to a persistent data store.

FR07: PDF boarding ticket retrieval The system must allow clients to retrieve the PDF boarding ticket.

FR08: Notification component integration The system must be integrated with the notification component. It must notify the component to dispatch an email containing a PDF ticket to the email address contained in the reservation.

FR09: User registration The system must allow clients to create a new account.

FR10: User login The system must allow clients to authenticate.

NFR11: Stripe payment gateway The system must be integrated with the Stripe payment gateway.

NFR12: MinIO storage The system must use a MinIO storage as its persistent data store to save the PDF tickets.

NFR13: CodeNOW platform The system must be integrated and deployed on the CodeNOW platform. *(This requirement specifies that the technologies must be accessible and compatible with CodeNOW.)*

2.2.2 Components

Two new microservices will be added with their own persistent data stores and a Keycloak service. The separate SQL database for the Credit microservice was added to further maintain the database per service pattern, which promotes isolation between the services. This pattern is discussed in more detail in the Related Theory Chapter 3.2.3. Additionally, the Frontend microservice will not fetch data from the Schedules microservice directly; instead, the API microservice will serve as an intermediary point. This approach is used due to the API gateway pattern, which is discussed in Subsection 3.2.5.

The functionality determined in the functional requirements will be mainly implemented by the respective microservices as follows.

Credit microservice FR01 - FR04

PDF microservice FR05 - FR08

Keycloak microservice FR09 - FR10

2.2.3 Persistent Data Model

Both new microservices will have their own persistent data. The Credit microservice in the form of events. The events will represent an account balance change at a given time. Also, an optimization technique called Snapshot will be used. The `AccountBalance` class is a representation of such Snapshot. The PDF service will only persist the PDF ticket data. A further explanation can be found in the implementation chapter. (Chapter 4)

As far as the reservation entity is concerned, a new `FAILED` state will be added to represent a situation where payment for the reservation could not be successfully made due to an insufficient account balance.

The API microservice will also keep a data replica of all account balances. The replica is introduced to reduce synchronous communication between the Credit and API microservices. The replicated data is not shown in the data model diagram.

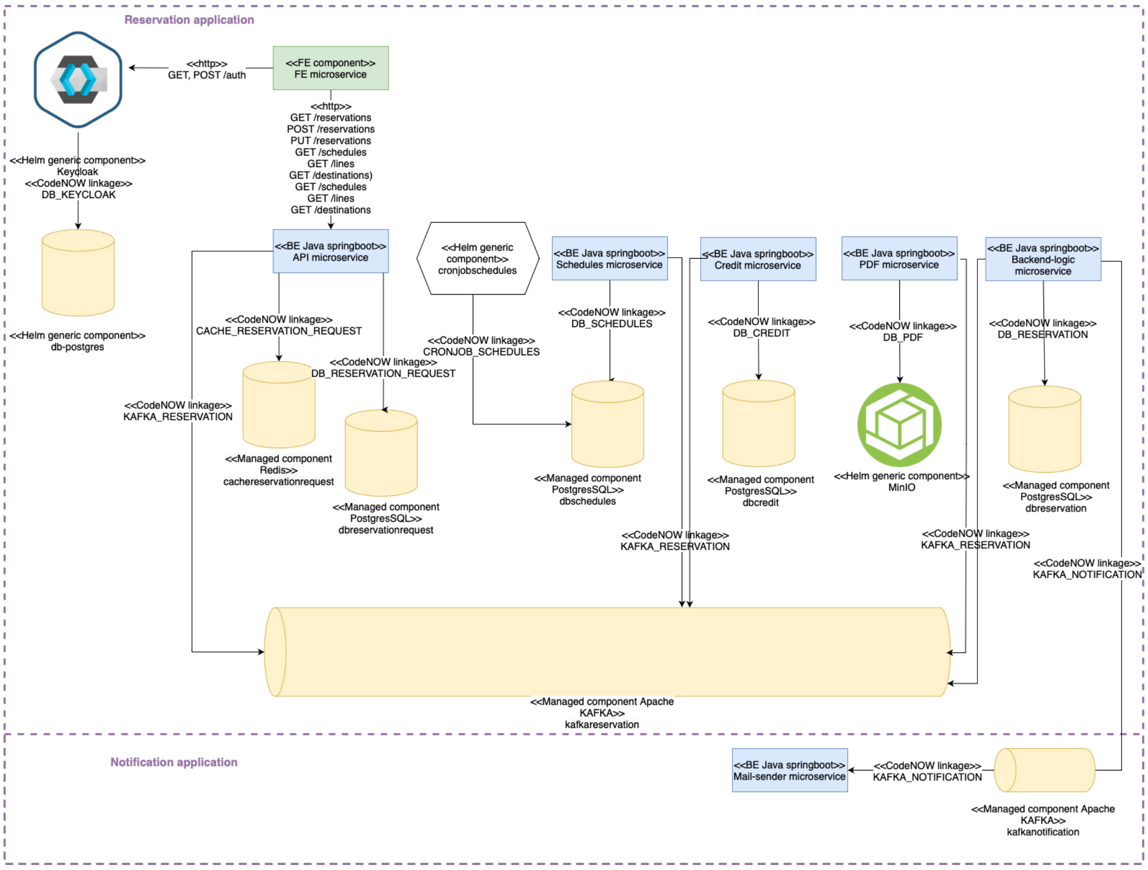

Reservation application
<<http>> GET, POST /auth
<<FE component>> FE microservice
<<http>> GET /reservations POST /reservations PUT /reservations GET /schedules GET /lines GET /destinations) GET /schedules GET /lines GET /destinations
<<Helm generic component>> Keycloak <<CodeNOW linkage>> DB_KEYCLOAK
<<Helm generic component>> db-postgres
<<BE Java springboot>> API microservice
<<CodeNOW linkage>> CACHE_RESERVATION_REQUEST
<<CodeNOW linkage>> DB_RESERVATION_REQUEST
<<CodeNOW linkage>> KAFKA_RESERVATION
<<Managed component Redis>> cachereservationrequest
<<Managed component PostgresSQL>> dbreservationrequest
<<Helm generic component>> cronjobschedules
<<CodeNOW linkage>> CRONJOB_SCHEDULES
<<BE Java springboot>> Schedules microservice
<<CodeNOW linkage>> DB_SCHEDULES
<<Managed component PostgresSQL>> dbschedules
<<BE Java springboot>> Credit microservice
<<CodeNOW linkage>> DB_CREDIT
<<Managed component PostgresSQL>> dbcredit
<<BE Java springboot>> PDF microservice
<<CodeNOW linkage>> DB_PDF
<<Helm generic component>> MinIO
<<BE Java springboot>> Backend-logic microservice
<<CodeNOW linkage>> DB_RESERVATION
<<Managed component PostgresSQL>> dbreservation
<<CodeNOW linkage>> KAFKA_RESERVATION
<<CodeNOW linkage>> KAFKA_RESERVATION
<<CodeNOW linkage>> KAFKA_NOTIFICATION
<<Managed component Apache KAFKA>> kafkareservation
Notification application
<<BE Java springboot>> Mail-sender microservice
<<CodeNOW linkage>> KAFKA_NOTIFICATION
<<Managed component Apache KAFKA>> kafkanotification

Figure 2.6: Component diagram TO-BE

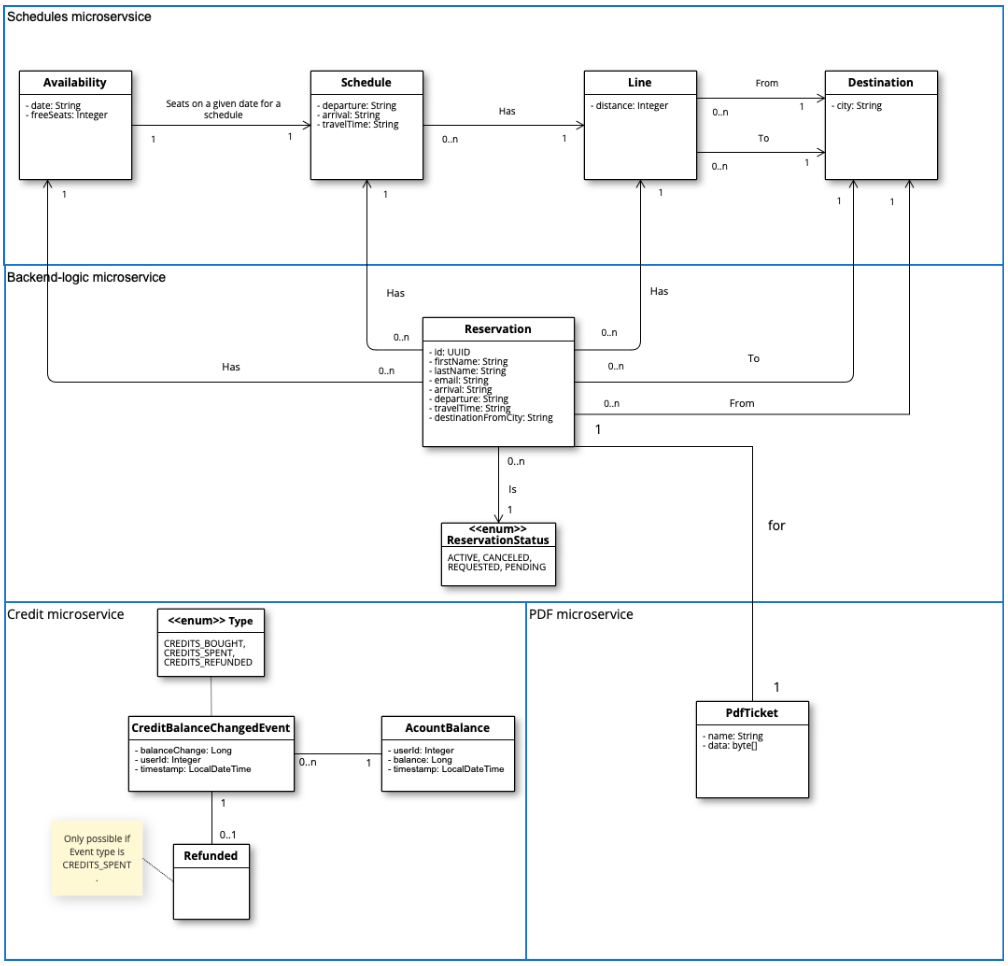


Figure 2.7: Data Model TO-BE - Entity diagram

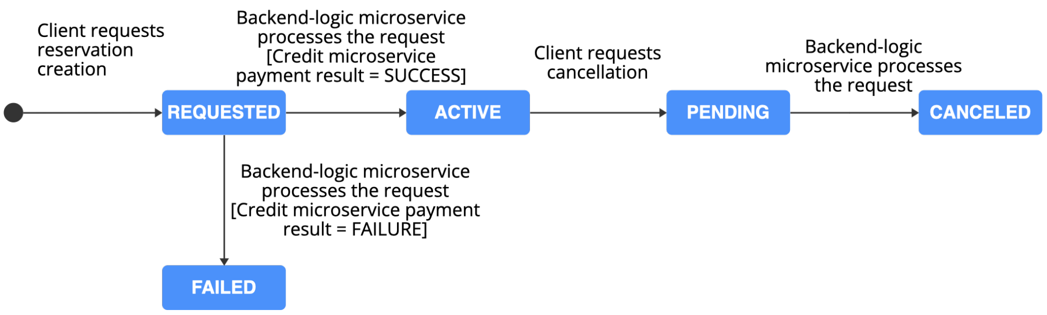

Client requests reservation creation
REQUESTED
Backend-logic microservice processes the request [Credit microservice payment result = SUCCESS]
ACTIVE
Client requests cancellation
PENDING
Backend-logic microservice processes the request
CANCELED
Backend-logic microservice processes the request [Credit microservice payment result = FAILURE]
FAILED

Figure 2.8: Data model TO-BE - Reservation state machine diagram

2.2.4 Top Up Account Balance Process

Based on the requirements, namely FR01 and NFR11, a new process to top up account balance must be added. The API microservice will first redirect the user to the Credit microservice, which will then redirect the user to the Stripe payment gateway. After successfully filling in the necessary payment information, the user will be redirected back to the Ticket Reservation Application. The account balance will get incremented once the Stripe gateway sends a confirmation about the payment being successful via a webhook API endpoint.

If the user cancels the payment, then they will also get redirected back to the Ticket Reservation Application, but to a screen showing a failure message.

2.2.5 Create Reservation Process

In the new modified reservation creation process, both new microservices need to be involved. The most important changes to the process are the following.

- The involvement of the Credit service before a reservation changes the state from `REQUESTED` to `ACTIVE`. This can only happen once the payment succeeds and the Credit service dispatches an appropriate event. To prevent a failure from happening due to an insufficient account balance, a front-end validation check will be performed. Were this validation to be bypassed, another validation would take place in the API service. This service's data can, however, be in an inconsistent state (more in Section 3.6) because it uses a data replica to perform the validation. If this validation falsely succeeds, then the reservation will change its state to `FAILED` after the payment fails directly in the Credit service.
- Creation, persistence, and email delivery of a PDF ticket after a successful reservation creation.
- The API microservice will update its local account balance replica.

2.2.6 Cancel Reservation Process

In the TO-BE version, the changes will be as follows.

- The Credit microservice will refund a matching payment.
- The Schedules microservice will now correctly adjust the amount of available seats.
- The API microservice will update its local account balance replica.

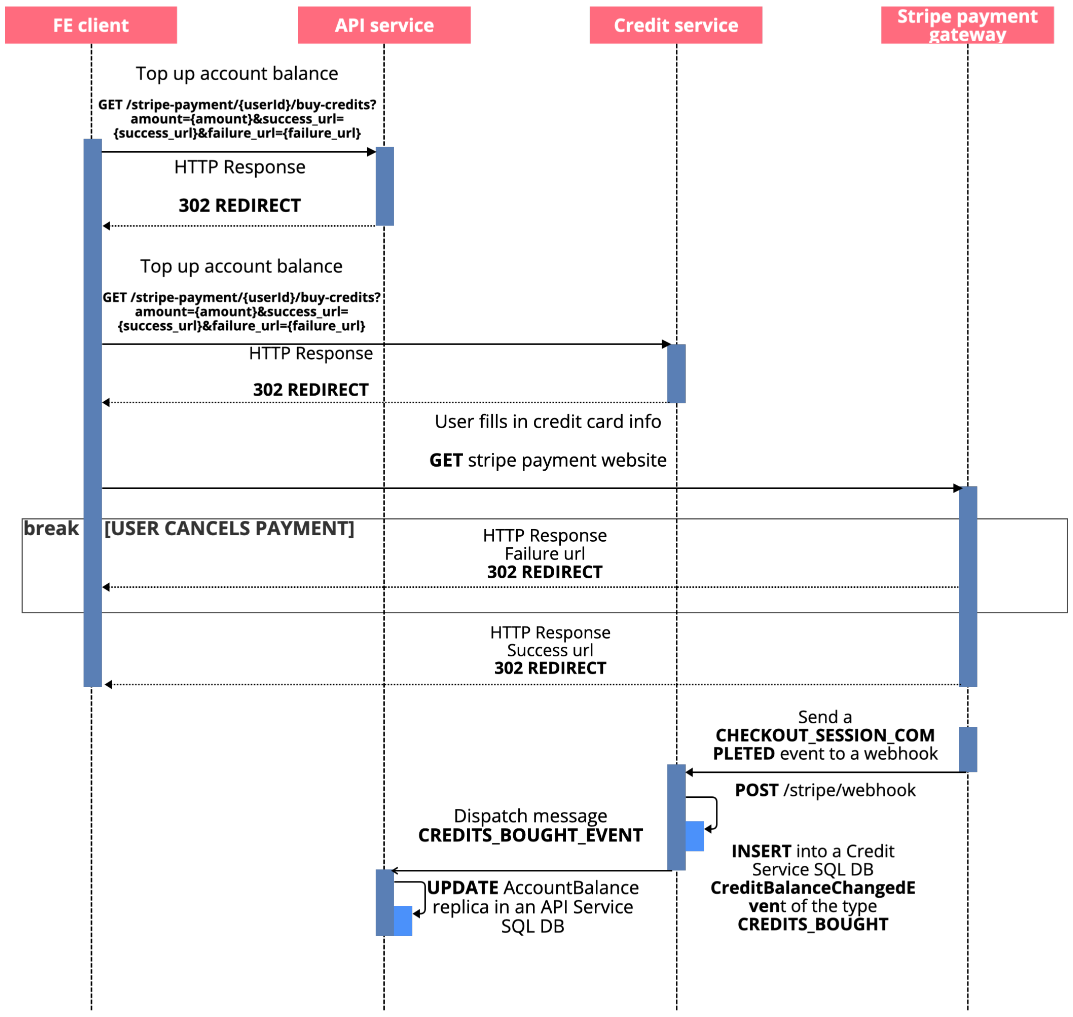

FE client
API service
Credit service
Stripe payment gateway
Top up account balance
GET /stripe-payment/{userId}/buy-credits?amount={amount}&success_url={success_url}&failure_url={failure_url}
HTTP Response
302 REDIRECT
Top up account balance
GET /stripe-payment/{userId}/buy-credits?amount={amount}&success_url={success_url}&failure_url={failure_url}
HTTP Response
302 REDIRECT
User fills in credit card info
GET stripe payment website
break
[USER CANCELS PAYMENT]
HTTP Response Failure url 302 REDIRECT
HTTP Response Success url 302 REDIRECT
Send a CHECKOUT_SESSION_COMPLETED event to a webhook
POST /stripe/webhook
Dispatch message CREDITS_BOUGHT_EVENT
INSERT into a Credit Service SQL DB CreditBalanceChangedEvent of the type CREDITS_BOUGHT
UPDATE AccountBalance replica in an API Service SQL DB

Figure 2.9: Top up account balance TO-BE - Sequence diagram

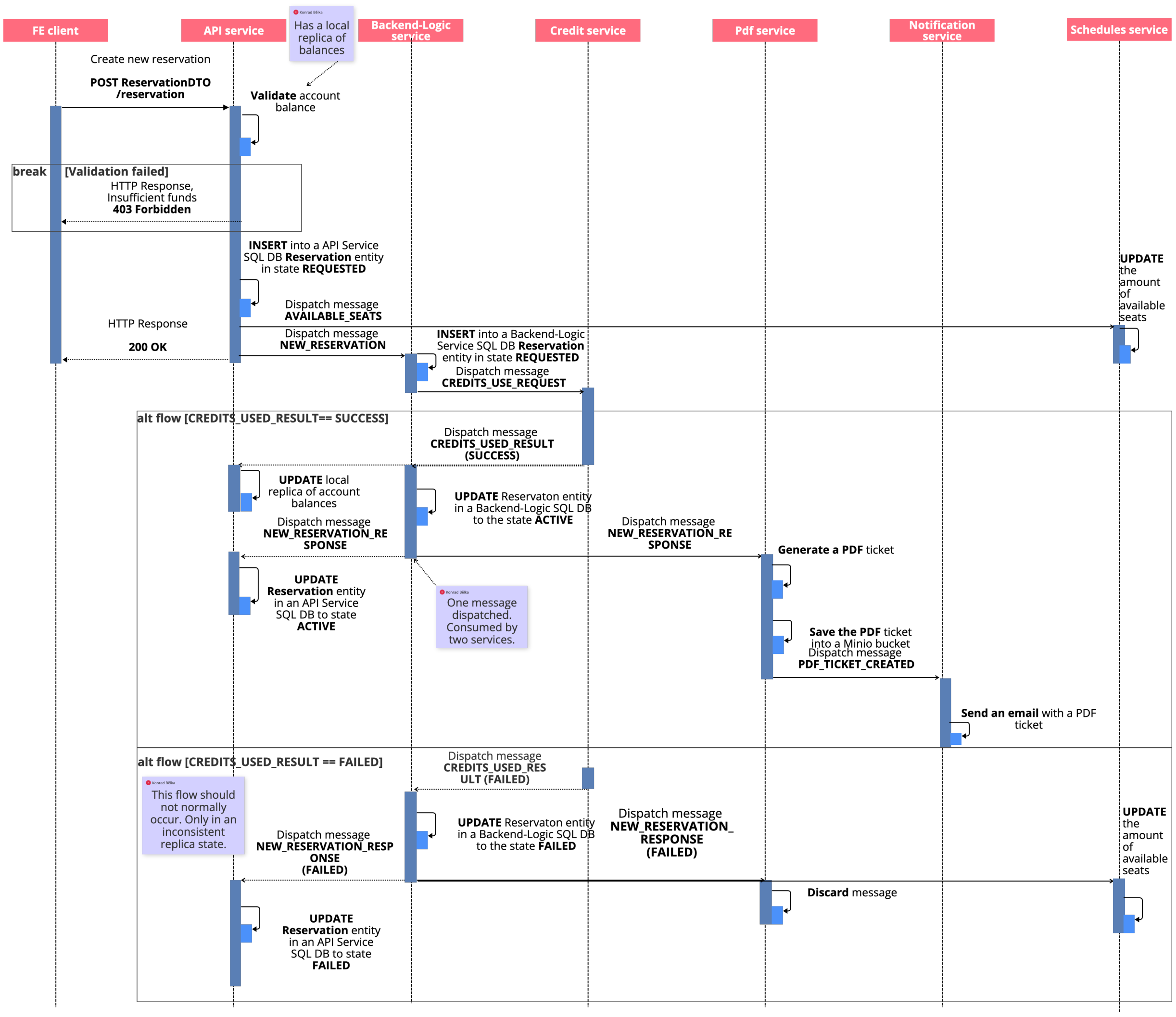

FE client
API service
Has a local replica of balances
Backend-Logic service
Credit service
Pdf service
Notification service
Schedules service
Create new reservation
POST ReservationDTO /reservation
Validate account balance
break [Validation failed]
HTTP Response, Insufficient funds 403 Forbidden
INSERT into a API Service SQL DB Reservation entity in state REQUESTED
Dispatch message AVAILABLE_SEATS
UPDATE the amount of available seats
HTTP Response
200 OK
Dispatch message NEW_RESERVATION
INSERT into a Backend-Logic Service SQL DB Reservation entity in state REQUESTED
Dispatch message CREDITS_USE_REQUEST
alt flow [CREDITS_USED_RESULT== SUCCESS]
Dispatch message CREDITS_USED_RESULT (SUCCESS)
UPDATE local replica of account balances
UPDATE Reservaton entity in a Backend-Logic SQL DB to the state ACTIVE
Dispatch message NEW_RESERVATION_RESPONSE
Dispatch message NEW_RESERVATION_RESPONSE
Generate a PDF ticket
UPDATE Reservation entity in an API Service SQL DB to state ACTIVE
One message dispatched. Consumed by two services.
Save the PDF ticket into a Minio bucket
Dispatch message PDF_TICKET_CREATED
Send an email with a PDF ticket
alt flow [CREDITS_USED_RESULT == FAILED]
Dispatch message CREDITS_USED_RESULT (FAILED)
This flow should not normally occur. Only in an inconsistent replica state.
UPDATE Reservaton entity in a Backend-Logic SQL DB to the state FAILED
Dispatch message NEW_RESERVATION_RESPONSE (FAILED)
Dispatch message NEW_RESERVATION_RESPONSE (FAILED)
UPDATE the amount of available seats
Discard message
UPDATE Reservation entity in an API Service SQL DB to state FAILED

Figure 2.10: Create reservation TO-BE - Sequence diagram

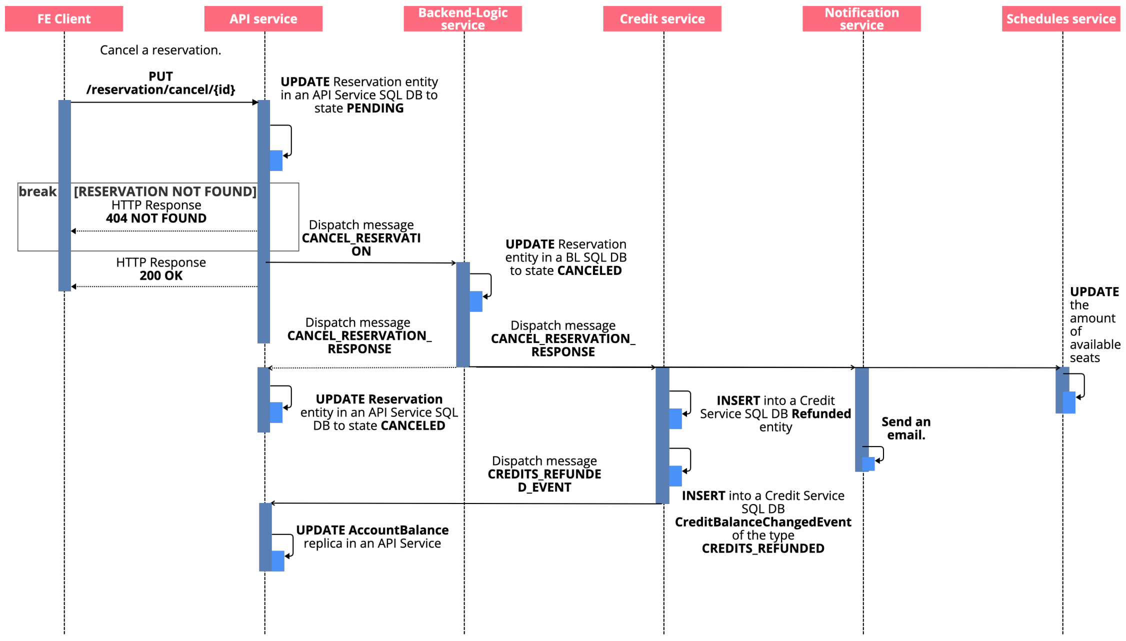
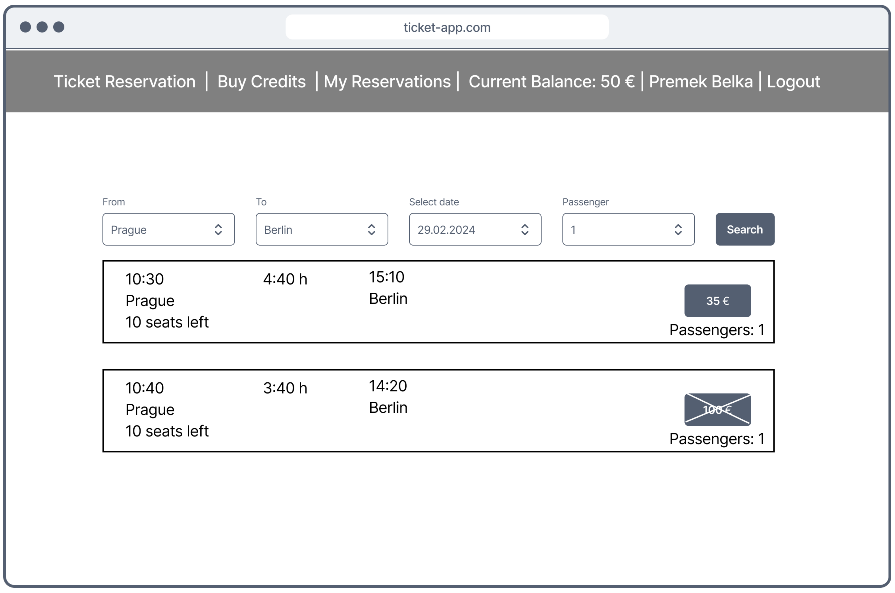

FE Client
API service
Backend-Logic service
Credit service
Notification service
Schedules service
Cancel a reservation.
PUT /reservation/cancel/{id}
UPDATE Reservation entity in an API Service SQL DB to state PENDING
break
[RESERVATION NOT FOUND]
HTTP Response 404 NOT FOUND
Dispatch message CANCEL_RESERVATION
HTTP Response 200 OK
UPDATE Reservation entity in a BL SQL DB to state CANCELED
UPDATE the amount of available seats
Dispatch message CANCEL_RESERVATION_RESPONSE
Dispatch message CANCEL_RESERVATION_RESPONSE
UPDATE Reservation entity in an API Service SQL DB to state CANCELED
INSERT into a Credit Service SQL DB Refunded entity
Send an email.
Dispatch message CREDITS_REFUNDED_EVENT
INSERT into a Credit Service SQL DB CreditBalanceChangedEvent of the type CREDITS_REFUNDED
UPDATE AccountBalance replica in an API Service

Figure 2.11: Cancel reservation TO-BE - Sequence diagram

2.2.7 User Interface

In total, five different new pages will be added, and changes will be made to the navigation bar to enable navigation between the new pages and to display the current account balance to the user. The page used to search for reservations will validate whether the user's account balance is sufficient to create a new reservation. To plan the changes, low fidelity prototypes were created (Figures 2.12 - 2.17).

The asynchronous nature of the system also needs to be considered in the UI design. Certain operations may not produce instant results to be shown to the user. For this reason, the intermediary pages (Figures 2.15, 2.16, 2.17) were added to inform the user that their request is being processed and may take some time to be completed. Under a light load, the operation (reservation creation, topping up the account balance) should produce a result before the user is automatically redirected or changes the page by themselves. A more complex approach to tackling this issue would involve keeping a local data model on the frontend part.


ticket-app.com

Ticket Reservation | Buy Credits | My Reservations | Current Balance: 50 € | Premek Belka | Logout

From: Prague
To: Berlin
Select date: 29.02.2024
Passenger: 1
Search

10:30 Prague 10 seats left | 4:40 h | 15:10 Berlin | 35 € | Passengers: 1

10:40 Prague 10 seats left | 3:40 h | 14:20 Berlin | ~~100 €~~ | Passengers: 1

Figure 2.12: Homepage

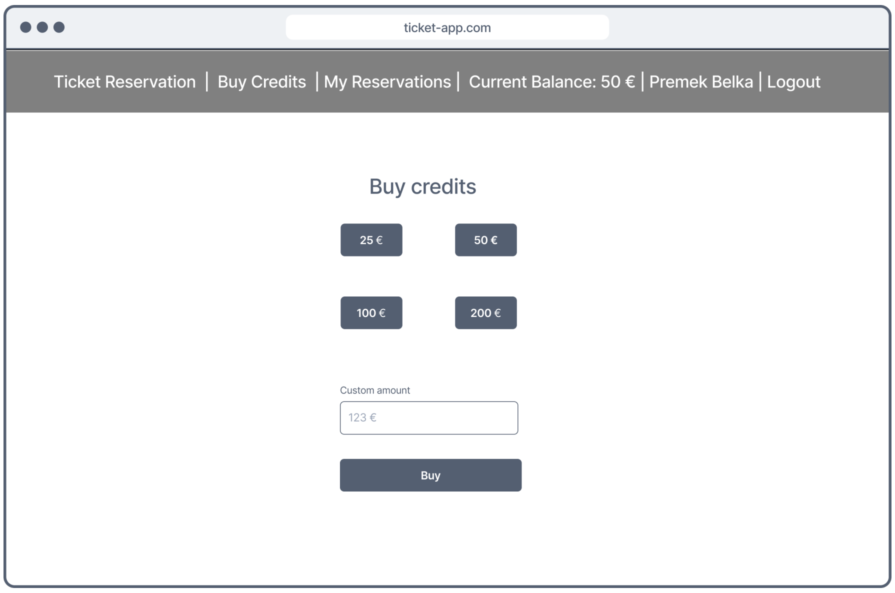

ticket-app.com
Ticket Reservation | Buy Credits | My Reservations | Current Balance: 50 € | Premek Belka | Logout
Buy credits
25 €
50 €
100 €
200 €
Custom amount
123 €
Buy

Figure 2.13: Buy credits page

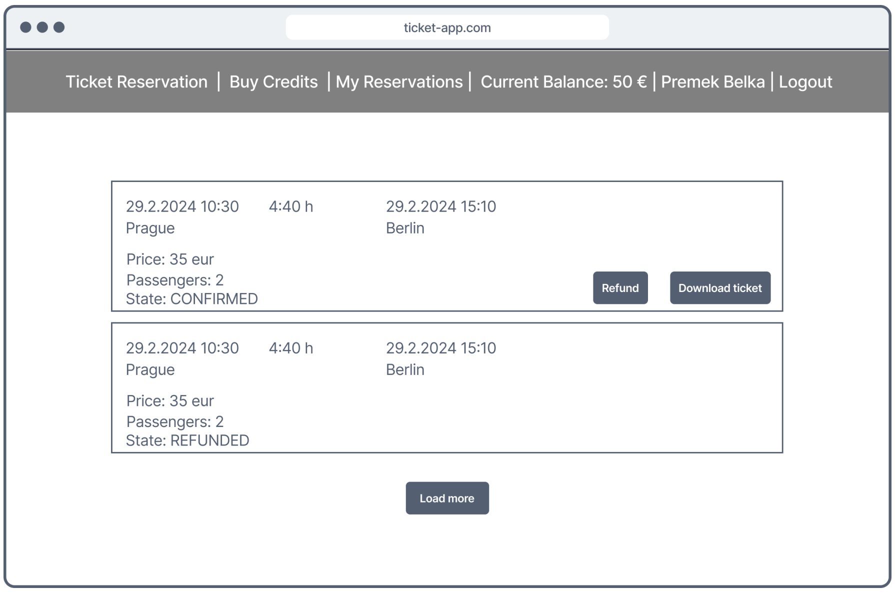
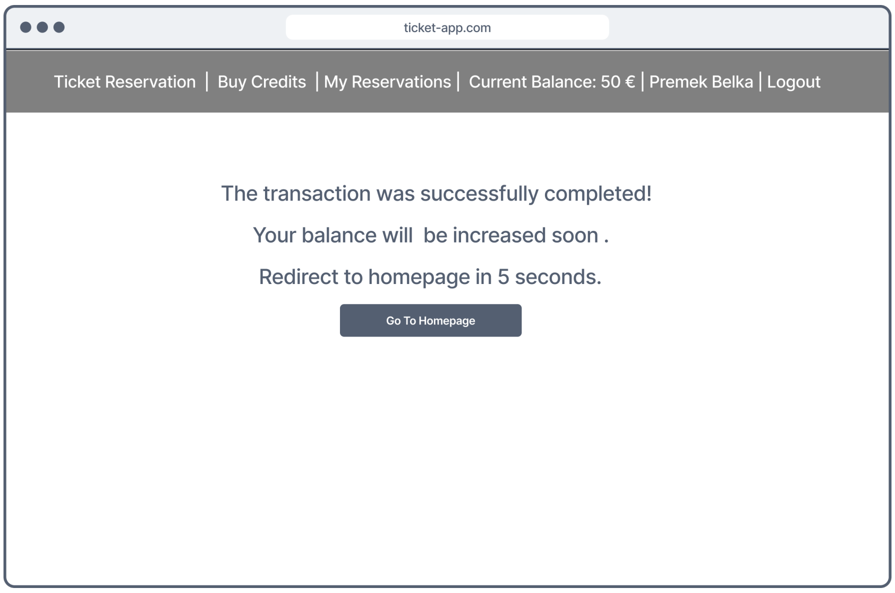


Figure 2.14: Reservation list page


ticket-app.com

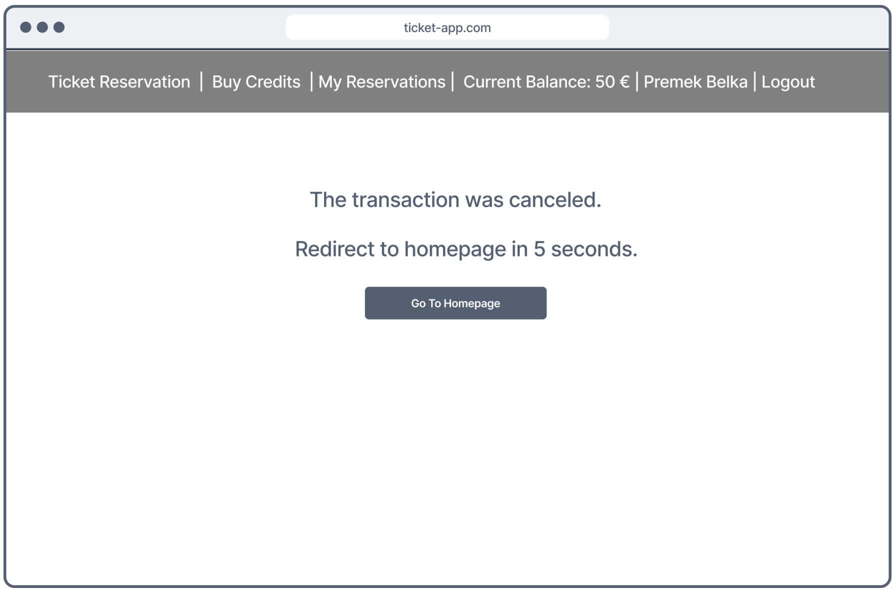
Ticket Reservation | Buy Credits | My Reservations | Current Balance: 50 € | Premek Belka | Logout

The transaction was successfully completed!

Your balance will be increased soon .

Redirect to homepage in 5 seconds.

Figure 2.15: Account topped up successfully page

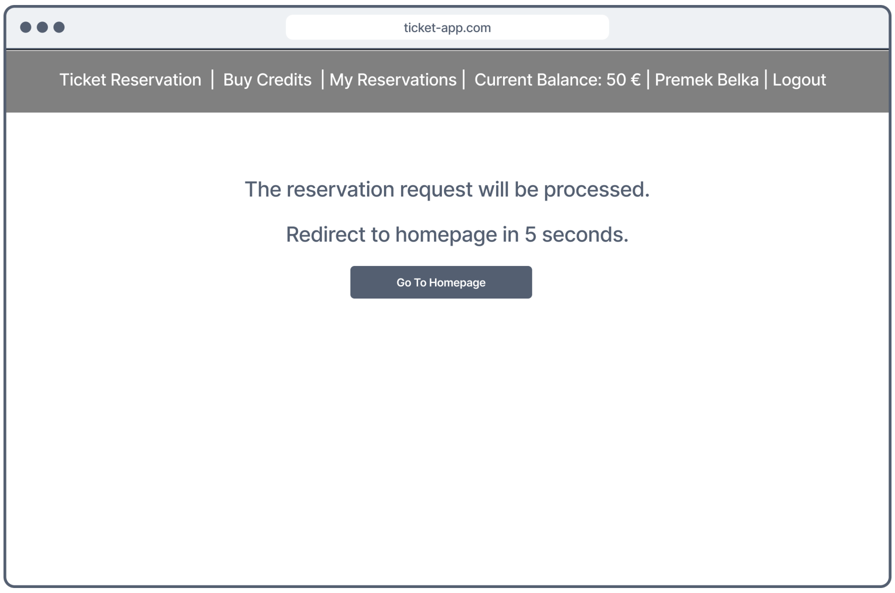

ticket-app.com

Ticket Reservation | Buy Credits | My Reservations | Current Balance: 50 € | Premek Belka | Logout

The transaction was canceled.

Redirect to homepage in 5 seconds.

Figure 2.16: Failed to top up the account page


ticket-app.com
Ticket Reservation | Buy Credits | My Reservations | Current Balance: 50 € | Premek Belka | Logout
Go To Homepage

The reservation request will be processed.

Redirect to homepage in 5 seconds.

Figure 2.17: Reservation completed page

Chapter 3
Related Theory

In order to effectively implement the solution outlined in the preceding chapter, it's imperative to grasp several key theoretical concepts. This chapter provides a concise overview of the most essential topics, offering a brief description of each.

3.1 Domain Driven Design

Domain Driven Design (DDD) is a technique of delivering software that mainly focuses on understanding the underlying business and its domain. It does so by advocating for the application of several key concepts.

Domain Model Model, which captures all the important entities and the relationships between them in a domain. [44]

Ubiquitous Language A common language in a domain used between developers, domain experts, and other stakeholders. It should reflect the domain as well as possible, therefore, making communication between all parties more effective and easier by striving to remove any ambiguity. [43]

Subdomain To manage a big domain model, it can be split into smaller subdomains, which target specific areas of expertise within a domain. [1][Chapter 2.2.3]

Bounded context A bounded context is a part of the software where particular terms, definitions, and rules apply consistently. [42]

These concepts can be used for specifying high-level architecture, as mentioned in the next section. However, DDD can also be applied to code design as described in an example in the Credit microservice implementation part of this work (Subsection 4.2.2). For this purpose, several more DDD concepts need to be introduced.

Entity Objects that have a distinct identity that runs through time and different representations. Such as a car or a person. [40]

Value object Objects that matter only as the combination of their attributes. Two value objects with the same values for all their attributes are considered equal. For example, a money object or a date object. [40]

Aggregate Within the domain, an aggregate is a collection of objects in the model that can be accessed as a single unit. [1][Chapter 5-2-2]

Domain event Is a class that captures the memory of something interesting happening that affects the domain. [41]

3.2 Microservice and Cloud-Native Architecture

Microservice architecture is an architectural style for developing applications. As opposed to a traditional monolithic architecture, it splits an application into a collection of independent microservices with their own responsibilities. In microservice architecture, an application request is processed by multiple small cooperating applications called microservices instead of just one bigger application. [34]

The world of microservices is very closely related to the concept of cloud-native architecture - a methodology that utilizes cloud services such as AWS, Azure, or CodeNOW to allow for applications to be developed in a more agile and dynamic way. These cloud services typically offer special support for creating microservice applications. [38]

The most relevant topics for this work are explored in this section.

3.2.1 Advantages and disadvantages

Applying microservice architecture has the following advantages:

- It makes continuous delivery possible.
- The services are smaller and easier to maintain.
- The services can be independently deployed. This allows the system to be scaled more easily.
- The services give a higher level of autonomy to teams. A whole service can be owned by one team.
- The services can be developed using various programming languages and technologies.

However, certain drawbacks also have to be considered. The most important ones are the following: [1][Chapter 1]

- It is difficult to correctly decompose a system into a set of services.
- A microservice system is a distributed system. Distributed systems are complex and come with their own set of challenges.

- The communication between services is more complicated than communication within a monolithic application.
- Transitioning a monolithic system to a microservice system is difficult and can be very costly.

3.2.2 Decomposition Using Domain Driven Design

One of the biggest challenges when designing a microservice system is the process of determining the services the system will consist of and their communication boundaries. This process is called decomposition.

One of the approaches is the decompose-by-subdomain technique. This technique works with two concepts taken from the DDD methodology: subdomains and bounded context. The whole application domain is split into subdomains, which all have their own domain models and terminology. The subdomains are specified by looking at the different areas of expertise within a business.

The models defined in each subdomain can represent bounded contexts for respective microservices. [39] An example of such separation determined by bounded contexts can, for example, be found in the TO-BE data model Figure 2.7.

3.2.3 Database Per Service

The database per service pattern advocates for each microservice to possess its own dedicated database, which is inaccessible to other microservices directly. Instead, data exchange occurs through APIs provided by the data-owning microservice. This architectural approach offers numerous advantages:

Scalability It is easier to scale one service based on the current needs along with its database.

Use of various technologies Each microservice can use a different kind of database most suitable to its data needs.

Reduced coupling The use of this pattern further promotes the reduction of coupling between the microservices. A change made to the database of one microservice does not affect the others.

The main drawback of this approach is the complicated implementation of queries, which need to join data owned by several microservices. Also, business transactions spanning data in various services become more difficult to implement. [48]

3.2.4 Communication

In traditional monolithic applications, operations between different parts of the system are typically executed by simply invoking methods on objects.

In microservices, this may not always be possible as different functionality resides in different microservices, which have to communicate with each other over a network.

When the communication takes place synchronously, the requesting side expects an answer to be returned before proceeding with its operations. This is very often done by exposing an API modeled with the use of the REST philosophy, which exposes URL endpoints designed as hierarchically organized resources. Operations on them are performed using corresponding HTTP methods. [12] One of the major downsides to this communication model is the reduction of the system's availability, due to the operation not being able to proceed until a response is returned. [1][Chapter 3.4.1]

The synchronous interactions can be replaced by using asynchronous communication instead. When a request is sent in an asynchronous communication, there's no anticipation of an immediate response, therefore none of the services are blocked, which results in higher system availability. The transition from synchronous interactions to asynchronous communication can be achieved by employing a message broker like Kafka to facilitate the exchange of messages between services. [1][Chapter 3.4.2]

3.2.5 API Gateway

In an application made out of many microservices, it would be very inconvenient for the clients to access their APIs directly. The API gateway pattern dictates that there should be a dedicated microservice that serves as a single entry point into the system for all external clients. This has the benefit of encapsulating the system's complexity. Also, in many cases, the microservices' APIs must not always be compatible with the clients. In such case, the API gateway can fulfill the function of an adapter. The API Gateway can also additionally be responsible for request routing, performing API composition, authentication and authorization, logging, caching, and load balancing. [1][Chapter 8]

3.2.6 Distributed Tracing

An important part of the development and maintenance of applications is the ability to understand their internal state and behavior from an outside perspective. In microservice applications, this can pose a significant challenge as a single request can pass through multiple microservices before a response is returned. To tackle this challenge, the pattern called distributed tracing can be applied.

The point of distributed tracing lies in instrumenting the code in such a way that at the start, each request is assigned a unique ID. The ID is propagated to each service responsible for handling the request. Important information, such as start time, end time, and operations invoked within a given service get saved into an external storage. Afterwards, this data can be queried to gain insights about the system's behavior, by being able to track a request's path as it got processed by various microservices. Additionally, each segment of processing within a service is encapsulated as a span. A span represents a unit of work within a trace, capturing timing information and metadata about that specific operation. [47]

3.2.7 Log Aggregation

Typically, each microservice generates large amounts of data in the form of logs. External storage is necessary to save and query this data for the whole application. Log aggregation is a pattern used to collect and store log messages from multiple sources in a centralized location. [51]

3.2.8 Contract Tests

A contract between microservices could be defined as a situation in which two services create an agreement on how to communicate with each other. Each contract has two sides: a providing side and a consuming side. Contract testing aims to test that both sides have the same understanding of what the contract is, or, in other words, that they have the same expectations about the form of the communication. [21]

In general, the contracts can be split into three categories. [21][Chapter 3.1.2.5]

Producer Contracts Producer contracts define what business functionalities a provider side offers externally, including message structures, interface operations, and service attributes like availability and security.

Consumer Contracts Consumer expectations of a Provider Contract are captured in a Consumer Contract. If the consumer and provider are compatible, the Consumer Contract is considered a subset of the Provider Contract, as only a portion of the available business functionality may be expected and used by the consumer. A provider may have multiple Consumer Contracts directed toward them, depending on the number of associated consumers.

Consumer-Driven Contracts Consumer-Driven Contracts can be seen as the union of all the expectations expressed by the Consumer Contracts for a provider. It is therefore a subset of Provider Contracts, as consumers do not have to take advantage of all the exposed functionality. There is always exactly one Consumer-Driven Contract per producer, which defines what functionality is being consumed and which is not.

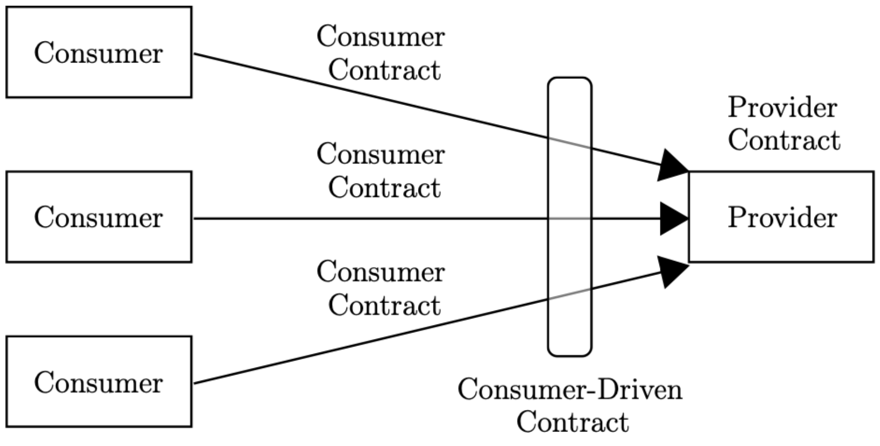


Figure 3.1: Types of contracts visualized [21][Chapter 3.1.2.5]

3.3 12-Factor Application

A 12-factor application is a document created by the founder of the Platform-as-a-Service Heroku. It contains recommendations and best practices to use for building modern web applications, sometimes also called Software-as-a-Service applications. The recommendations are split into 12 topics. The following list briefly summarizes the main point of each recommendation. [3]

Codebase A version control system should be used to develop an application. There should be only one codebase per application.

Dependencies An application should not rely on the implicit existence of pre-installed system libraries and dependencies.

Configuration Configuration should be stored in environment variables, not directly in code.

Backing services The application should not make a distinction between local and third-party services. Such as between a locally managed database and a mail server managed by a third party. Both should be attached resources accessible by a stored URL in the config.

Build, release, run The application should strictly separate the build, release, and run phases.

Processes The application processes should be stateless and should not share anything. Any persistent data should be saved to an external service.

Port binding The application should be completely self-contained and only expose its functionality as a service by binding to a port. For example, a web application can expose its service by binding to an HTTP port.

Concurrency The application should be able to scale horizontally by being able to easily add new processes.

Disposability The application processes should try to minimize their startup time and shut down gracefully upon receiving a SIGTERM signal. A graceful shutdown in the case of an HTTP service means not listening anymore on the exposed port and letting ongoing requests finish. The application should also be durable in case of sudden shutdowns.

Dev/prod parity Different application environments, such as the dev, staging, and prod environments should be kept as similar as possible.

Logs Logs should be treated as event streams. The application should not concern itself with their routing and storage. Instead, they should be written into the standard output stream and then handled specifically based on the environment.

Admin processes Admin tasks should be run as separate one-off processes in an identical environment as the main application processes. The execution's environment bundled tool should be used to run them if possible.

3.4 Containerized Applications

Modern-day applications typically require a wide range of dependencies to run, it could be different libraries, programming language runtimes, configurations, and more. Migrating applications across diverse environments—say, from one developer's local setup to another's—can pose considerable challenges. To make the deployment of applications easier containerization can be used. Containerization makes it possible to package applications into so-called containers that contain all the necessary dependencies and can be run in any environment. [33]

Containerization aligns closely with microservice architecture, primarily due to the necessity for fast and efficient deployment of services. By encapsulating each microservice within its own container, organizations can achieve greater agility and scalability. This decoupling enables independent development, testing, and deployment of microservices, facilitating quick iteration and updates without disrupting the entire application. Additionally, container orchestration platforms like Kubernetes further enhance the management and scalability of microservices by automating deployment, scaling, and service discovery across a cluster of containers. [34]

3.5 Event-Driven Architecture

In an Event-Driven architecture, components initiate actions that lead to the generation of events. These events are then transmitted through a message broker, enabling any component to subscribe to and react to them. A fundamental aspect of events is their immutable nature, as they represent occurrences that have already happened and cannot be changed or revoked. [35][Chapter 4]

A typical sign of Event-Driven systems is the use of an asynchronous style of communication. An important example of this found within the Ticket Reservation Application is the interaction with the Stripe payment gateway. Once a checkout is requested and the user is redirected to the gateway, then no immediate response is expected. Only if the user correctly fills in their data and confirms the payment, the Ticket Reservation application receives an asynchronous response containing the `CHECKOUT_SESSION_COMPLETED` event (Figure 2.9).

3.6 Eventual Consistency

According to the Encyclopedia of Database Systems, Eventual Consistency could be defined in the following way: "In a replicated database, the consistency level defines whether and how the values of the replicas of a logical object may diverge in the presence of updates. Eventual consistency is the weakest consistency level that guarantees useful properties. Informally, it requires that all replicas of an object will eventually reach the same, correct, final value, assuming that no new updates are submitted to the object." [36]

When working with replicated data, especially in distributed systems, the BASE (Basically Available, Soft state, Eventually consistent) model could be used for understanding the trade-offs involved in maintaining consistency and availability.

BASE transactions contrast with the ACID (Atomicity, Consistency, Isolation, Durability) transactions commonly associated with traditional databases. While ACID transactions prioritize strong consistency, BASE transactions prioritize availability and eventual consistency.

In the context of the definition provided by the Encyclopedia of Database Systems, eventual consistency, a key component of the BASE model, ensures that over time, all replicas of a data object will converge to the same correct value, assuming no new updates are introduced to the object. This aligns with the notion of a "soft state" in BASE, where the system tolerates temporary inconsistencies between replicas with the expectation that they will eventually reconcile.

BASE transactions reflect the reality of distributed systems, where achieving immediate consistency across replicas may be impractical due to factors like network latency, partition tolerance, and the need for high availability. By prioritizing availability and eventual consistency, BASE transactions enable systems to remain responsive even in the face of network failures, while still ensuring that data consistency is eventually achieved as updates propagate through the system. [37]

3.7 Object-Based Storage

Object-Based Storage systems offer a way to solve the problem of saving a large amount of unstructured data, such as images or audio files. Saving such data in a relational database or on the local filesystem becomes unsustainable with growing application traffic and increasing amounts of files. [16][Chapter 8]

In Object-Based Storage systems, the data is organized into buckets and objects. Buckets could be thought of as disks with an endless storage capacity that store the data. These buckets, can in reality, be replicated across several disks, ensuring high levels of availability and durability. Objects are files with unique names stored in buckets. [16][Chapter 8.1.1]

In contrast to relational databases, the ability to query data is greatly reduced. The Object-Based Storage systems typically offer a simple, almost key-value API for retrieving data. However, this is greatly compensated for by the scalability and performance benefits of Object-Based Storage. Since objects are stored with unique identifiers, retrieval is typically done through simple find-by-ID queries. This type of storage is especially well-suited for systems where availability and scalability play an important role. This, however, comes at the cost of sacrificing the ACID properties of a typical relational database. [16][Chapter 8.10]

Chapter 4

Implementation

This chapter focuses on the implementation of the new requirements and the proposed solution in Section 2.2. The first part introduces the used technological stack. Then the implementation of the newly added microservices is presented. The next part gives an overview of the work required to be done in the rest of the system. Lastly, distributed tracing and configuration handling are mentioned.

4.1 Used Technologies

The technologies for the PDF and Credit microservices were chosen based on the capabilities of the CodeNOW platform and the special support it provides for them — the most important one being predefined templates. The list of the most important technologies is the following.

Spring Boot Both the credit and PDF microservices were developed in Java using the open-source Spring Boot framework. This framework enables rapid development of web applications using the Spring framework. The main advantages that Spring Boot offers compared to Spring are autoconfiguration, easier dependency selection and management, and an automatically embedded Tomcat server. [7]

PostgreSQL The database used for the Credit microservice was PostgreSQL. It is an open-source, object-relational database management system that can be communicated with by using the SQL query language. [11]

MinIO MinIO is an open-source object storage server that is compatible with Amazon S3 cloud storage service. It is commonly used for storing large amounts of unstructured data such as images, videos, and log files. In this project, MinIO was utilized for storing and serving generated PDF documents. [57]

Kafka Kafka is an open-source distributed messaging platform commonly referred to as a messaging queue. Its main advantages include broad support through libraries for most commonly used programming languages, fast processing of published events, scalability, and high availability. [10]

In a microservices architecture, it can be utilized for implementing asynchronous communication between individual microservices. [1][Chapter 3-3]

Docker For simplifying local development and eliminating the need for local installation of individual external services, Docker was utilized. Docker is an open-source platform that enables developers to package software into so-called containers, which contain all the necessary dependencies for running the software. [6]

Stripe Stripe is an online platform offering a wide range of services for processing not only online financial transactions. [9] For the project purposes, the Java library provided by Stripe was used, facilitating the use of the Stripe payment gateway for processing one-time payments from payment cards.

Flyway Flyway is a library used for versioning the schema of a relational database. Versions are created using migration files, which contain SQL scripts for migrating the database to a specific version. [8]

4.2 Credit Microservice

The Credit microservice adds the ability to keep users' account balance and process payments and refunds within the system. In this section, the code structure, the application of Domain Driven Design, Event Sourcing and the Snapshot pattern are explained. Lastly, the integration with the Stripe payment gateway is shown.

4.2.1 Project and Code Structure

The list and the Figure 4.1 below describe the structure of the project and the code within it.

Codenow Contains a `.yaml` Spring configuration file.

Config Contains Java configuration classes.

Controller Contains REST controllers.

Dto Contains Data Transfer Objects used for sending and receiving data.

Exception Contains custom exceptions and exception handlers defining the controller response if a certain exception occurs.

Factory Contains `LineItemFactory`, which simplifies the creation of a Stripe specific object.

Fixtures Contains classes, which insert data upon application startup in a local development environment.

Kafka Contains Kafka consumers, producers, and events sent into the queue.

Model Contains domain entities, which contain most of the core business logic.

Repository Contains data access layer classes, which are used for communicating with the SQL database.

Scheduled Contains a class defining a CRON job, which periodically runs to persist an event snapshot into the database. More in the Snapshot Section 4.2.4.

Service Contains classes, which can be considered a sort of facade in the code. They contain methods, that perform important business functions, however, they only delegate to other classes and aggregate the functionality. More in the subsection Domain Driven Design 4.1.

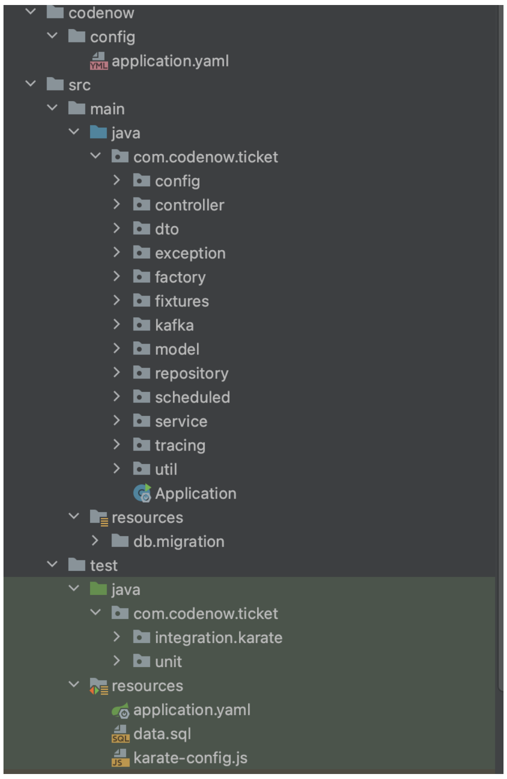
Tracing Contains distributed tracing instrumentation. More in Section 4.5.

Util Contains classes with helper static methods.

Resources Contains Flyway `.sql` migration scripts to version the database. In the case of test resources test configurations are included.

Integration Contains Karate integration tests.

Unit Contains unit tests.



```
codenow
  config
    application.yaml
src
  main
    java
      com.codenow.ticket
        config
        controller
        dto
        exception
        factory
        fixtures
        kafka
        model
        repository
        scheduled
        service
        tracing
        util
        Application
    resources
      db.migration
  test
    java
      com.codenow.ticket
        integration.karate
        unit
    resources
      application.yaml
      data.sql
      karate-config.js
```

Figure 4.1: Credit microservice project structure

4.2.2 Domain Driven Design

In designing the model and organizing business logic, some principles from the Domain Driven Design methodology were utilized. A crucial concept in designing the object model according to DDD is the notion of an aggregate. Within the domain, an aggregate is a collection of objects in the model that can be accessed as a single unit. [1][Chapter 5-2-2] In the model, there aren't many objects involved, so it cannot be said that this aggregate is a collection, as there are only two unrelated entities (`AccountBalance`, `Refunded`) and several types of events (`AccountBalanceChanged`) (Figure 4.2). Nevertheless, the entity `AccountBalance` will be referred to as an aggregate in the text, as other principles related to this methodology were used.

One way to organize business logic in an application is by using the transaction script design pattern. This pattern is based on placing all business logic into procedural scripts called services. Conversely, the model itself contains minimal business logic and typically consists only of classes with private attributes, getters, and setters. Such a model is sometimes referred to as anemic. [1][Chapter 5-1-1]

However, this approach is not utilized in this microservice. Instead, the domain model design pattern was employed. In this case, most of the business logic was placed within the domain classes. [1][Chapter 5-1-2] Still, some operations remain in service classes, such as publishing events or database storage interactions (Code Snippet 4.1). In the Code Snippets, the symbol of the three dots "..." is used to signify that part of the code was left out.

The factory design pattern was also applied to creating certain classes. It is implemented as a static method of a class and has the advantage of not allowing the creation of a class with business-invalid data. In DDD terminology - it makes sure all the object invariants are upheld (Code Snippet 4.2). [1][Chapter 5-1-3]

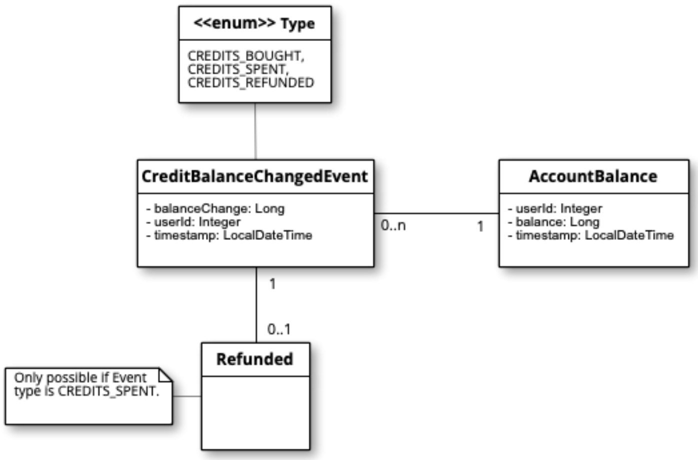


Figure 4.2: Credit microservice model

```
@Service
public class CreditService {
...
public void spendCredits(Long creditsToSpends, Integer userId) throws
     ExecutionException, InterruptedException {
       CreditBalanceChangedEvent event = CreditBalanceChangedEvent.createCreditsSpent
            (
              userId,
              creditsToSpends,
              creditBalanceChangeEventRepository,
              accountBalanceSnapshotRepository
       );

       creditBalanceChangeEventRepository.save(event);
       creditBalanceChangedProducer.publish(event);
    }
...
}
```

Code Snippet 4.1: A service method example

```
@Entity
@Table(name = "credit_balance_changed_event")
public class CreditBalanceChangedEvent {
...
   private CreditBalanceChangedEvent(Integer userId, Long balanceChange,
        CreditBalanceChangeType messageType) {
       this.id = UUID.randomUUID();
       this.userId = userId;
       this.balanceChange = balanceChange;
       this.messageType = messageType;
       timestamp = LocalDateTime.now();
   }

   public static CreditBalanceChangedEvent createCreditsRefunded(UUID
        paymentToRefund, CreditBalanceChangeEventRepository
        creditBalanceChangeEventRepository, RefundedRepository refundedRepository) {
   CreditBalanceChangedEvent event = creditBalanceChangeEventRepository.findById(
        paymentToRefund).get();
   if (event.messageType != CreditBalanceChangeType.CREDITS_SPENT) throw new
        NoSuchElementException("No such payment exists.");
   if (event.wasRefunded(refundedRepository)) throw new PaymentAlreadyRefunded("This
         payment has already been refunded.");

   return new CreditBalanceChangedEvent(event.getUserId(), -event.getBalanceChange(),
         CreditBalanceChangeType.CREDITS_REFUNDED);
   }
...
}
```

Code Snippet 4.2: An event factory method example

4.2.3 Event Sourcing

For tracking purchases, cancellations, and topping up the account balance, the Event Sourcing design pattern was employed. This design pattern involves not storing aggregates directly in the database; instead, they are stored as a series of events representing state changes. These events are also published in a messaging queue, where they can be consumed by other microservices. The aggregate is then not read directly from the database but is reconstructed by applying events backward, as demonstrated in Code Snippet 4.3.

In this specific implementation, events were modeled as an entity mapped to a single table in the database. The data it contains can be seen in Code Snippet 4.4 below. Upon relevant operations, the resulting event is stored in the database and then published to Kafka (Code Snippet 4.1).

This approach brings countless advantages compared to the traditional method, where aggregates are stored directly in the database and mutating operations are performed directly on them. The main advantage is that the entire history of operations is stored in the form of events, which can be used to reconstruct the aggregate's state at any given time. This may, for example, be needed for auditing purposes. Another important advantage, especially in microservice architecture, is that events can be published for other microservices, for which they may be relevant in some way. [1, Chapter-6.1] This has the advantage of potentially replacing necessary synchronous communication with asynchronous communication, which in turn results in higher system availability, as described in Subsection 3.2.4.

However, Event Sourcing also brings along with it a number of disadvantages. The most significant ones include the fact that it is an unfamiliar programming approach for many, increased memory requirements, and slower and more complicated read operations on the aggregate. [1, Chapter-6.1] The issue of slower reading has been mitigated using the Snapshot design pattern in the following chapter.

```java
public class AccountBalance {
...
    public static AccountBalance recreateFromEvents(
        Integer userId,
        CreditBalanceChangeEventRepository creditBalanceChangeEventRepository
        ) {
            AccountBalance accountBalance = new AccountBalance(userId);

            creditBalanceChangeEventRepository.findAllByUserId(userId)
                    .forEach(accountBalance::apply);

            accountBalance.setTimestampToNow();

            return accountBalance;
        }

    private void apply(CreditBalanceChangedEvent event) {
        balance += event.getBalanceChange();
    }
}
```

Code Snippet 4.3: Reacreating the AccountBalance agreggate

```
@Entity
@Table(name = "credit_balance_changed_event")
public class CreditBalanceChangedEvent {

    @Id
    @Getter
    private UUID id;

    @Getter
    private Long balanceChange;

    @Getter
    private Integer userId;

    @Enumerated(EnumType.STRING)
    private CreditBalanceChangeType messageType;

    @Column(name = "timestamp", columnDefinition = "TIMESTAMP")
    private LocalDateTime timestamp;

...
}
```

Code Snippet 4.4: Domain event structure

4.2.4 Snapshot pattern

Snapshot is a design pattern that allows for optimizing the speed of creating an aggregate from events. The principle lies in periodically calculating the state of the aggregate at some point in time, let's call it X, and storing it together with this time point X. During subsequent read operations, it is not necessary to recreate the aggregate from all stored events, but only from those whose publication time is greater than X. According to the attached image (Figure 4.3), this means that if we place point X between *Event N* and *Event N+1*, then during each subsequent read operation, the aggregate can be created from the Snapshot of version N and events greater than N. This way, the processing of N events can be saved. [1, Chapter-6.1.5]

In this specific implementation, a separate table is created in a relational database for Snapshots, which contains information about the time point X, the state of the credit account, and the user ID to which the account belongs. Snapshots are recalculated once daily at midnight using a CRON job.

In the example (Code Snippet 4.5), there is an optimized method for reading the `AccountBalance` aggregate that utilizes Snapshots.

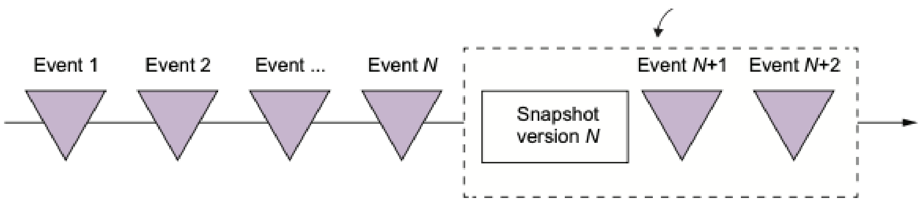


Figure 4.3: Snapshot [1, Chapter-6.1.5]

```java
public class AccountBalance {
...
    public static AccountBalance recreateFromEventsAndSnapshot(
    Integer userId,
    CreditBalanceChangeEventRepository creditBalanceChangeEventRepository,
    AccountBalanceSnapshotRepository accountBalanceSnapshotRepository
   ) {
    AccountBalance accountBalance = getSnapshot(userId,
         accountBalanceSnapshotRepository);

    if (Objects.isNull(accountBalance)) {
        return recreateFromEvents(userId, creditBalanceChangeEventRepository);
    }

    else {
        creditBalanceChangeEventRepository.findAllByUserIdAndTimestampAfter(userId,
             accountBalance.getTimestamp())
                .forEach(accountBalance::apply);
    }

    accountBalance.setTimestampToNow();

    return accountBalance;
    }
...
}
```

Code Snippet 4.5: Recreating the AccountBalance from a Snapshot

4.2.5 Integration with the Stripe Payment Gateway

An official Stripe Java client library was used for the integration.[1]

As the first step of the integration, an HTTP API endpoint was exposed, which starts the payment process and redirects the user to the Stripe gateway. The method validates the amount of money the client wants to top up. Then the provided redirect URLs are validated. At the end of the method, a Checkout Session object is created, and the user is redirected to the Stripe server to manually fill in their payment data. The payment is later linked to a specific user by parsing the user ID information from the Checkout Session metadata (Code Snippet 4.7).

The Checkout Session is an important concept when working with the Stripe gateway. It is an object that contains all the crucial information about the ongoing payment, including how the payment itself should proceed. It can be configured using a builder available in the provided Java library. [2]

If the payment is successful, then the Stripe service answers asynchronously by calling a Webhook HTTP endpoint. Although HTTP communication is synchronous, the answer is asynchronous relative to the completion of the payment, as the Ticket Reservation Application does not wait for the answer after the user fills in their payment data because the user is redirected immediately back to a page with information about his payment being processed. The webhook controller method validates the signature and deserializes the received data. If it contains information about a successful

[1]Stripe java client library, available at `https://github.com/stripe/stripe-java`

payment, then it is passed to the `StripeService` class for further processing by the application's business logic (Code Snippet 4.6).

```java
@PostMapping(path ="/webhook", consumes = "application/json")
public Mono<String> webHook(ServerHttpResponse response, ServerHttpRequest request,
     @RequestBody String payload) {
    String sigHeader = request.getHeaders().get("Stripe-Signature").get(0);
    Event event = null;

    try {
        event = Webhook.constructEvent(
                payload, sigHeader, endpointSecret
        );
    } catch (JsonSyntaxException e) {
        response.setStatusCode(HttpStatus.resolve(400));
        log.warn("Invalid Stripe webhook payload.");
        return Mono.just("");
    } catch (SignatureVerificationException e) {
        response.setStatusCode(HttpStatus.resolve(400));
        log.warn("Invalid Stripe webhook signature.");
        return Mono.just("Invalid signature.");
    }

    EventDataObjectDeserializer dataObjectDeserializer = event.
         getDataObjectDeserializer();
    StripeObject stripeObject = null;
    if (dataObjectDeserializer.getObject().isPresent()) {
        stripeObject = dataObjectDeserializer.getObject().get();
    } else {
        log.warn("Invalid nested object.");
        return Mono.just("");
    }

    switch (event.getType()) {
        case "checkout.session.completed": {
            stripeService.handleCheckoutSessionCompleted(event);
            break;
        }
        default:
            return Mono.just(("Unhandled event type: " + event.getType()));
    }

    response.setStatusCode(HttpStatus.resolve(200));
    return Mono.just("Invalid signature.");
}
```

Code Snippet 4.6: Stripe webhook controller method

```
@GetMapping("/{userId}/buy-credits")
public Mono<Void> buyCredits(ServerHttpResponse response,
                             ServerHttpRequest request,
                             @RequestParam("amount") Long amount,
                             @RequestParam(required = false, name = "success_url") String
                                  successUrl,
                             @RequestParam(required = false, name = "failure_url") String
                                  failureUrl,
                             @PathVariable String userId) throws StripeException,
                                  URISyntaxException {
    log.info("Received request for buying credits for user: " + userId + " amount: " +
           amount + " successUrl: " + successUrl + " failureUrl: " + failureUrl);

    if (amount < 1 || amount > 100_000) {
        throw new InvalidInputException("Minimal amount of credits to purchase is 1.
             Maximal amount is 100.000.");
    }

    if (successUrl != null && !allowedRedirectDomainName.contains(URLUtil.
         extractDomainName(successUrl)))
    {
        throw new NotAllowedRedirectUrl(successUrl + " Is not an allowed redirect URL.
              " +
                "Allowed hostname: " + allowedRedirectDomainName);
    }

    if (failureUrl != null && !allowedRedirectDomainName.contains(URLUtil.
         extractDomainName(failureUrl)))
    {
        throw new NotAllowedRedirectUrl(failureUrl + "Is not an allowed redirect URL.
             " +
                "Allowed hostname: " + allowedRedirectDomainName);
    }

    String defaultSuccessUrl = URLUtil.getBaseUrl(request.getURI()) + "/stripe-
         payment/buy-credits/success";
    String defaultFailureUrl = URLUtil.getBaseUrl(request.getURI()) + "/stripe-
         payment/buy-credits/cancel";

    SessionCreateParams params = SessionCreateParams.builder()
            .setMode(SessionCreateParams.Mode.PAYMENT)
            .setSuccessUrl(successUrl != null ? successUrl : defaultSuccessUrl)
            .setCancelUrl(failureUrl != null ? failureUrl : defaultFailureUrl)
            .addLineItem(lineItemFactory.createCreditsLineItem(amount))
            .putMetadata("userid", userId)
            .build();

    Session session = Session.create(params);

    response.setStatusCode(HttpStatus.PERMANENT_REDIRECT);
    response.getHeaders().setLocation(URI.create(session.getUrl()));
    response.getHeaders().setCacheControl(CacheControl.noCache().getHeaderValue());
    log.info("Redirecting " + userId + " to: " + session.getUrl());
    return response.setComplete();
}
```

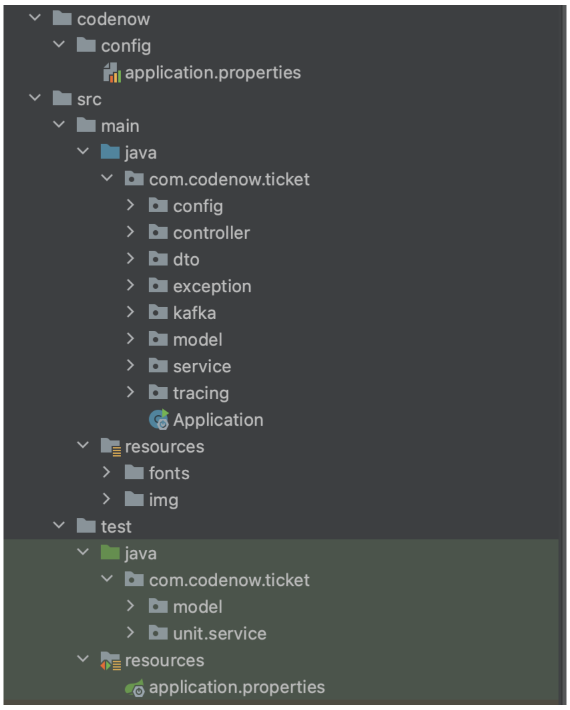
Code Snippet 4.7: Top up account controller endpoint

4.3 PDF Microservice

The PDF microservice is responsible for generating and managing PDF documents. This section explores the project structure, key libraries used for PDF ticket generation, and the integration with the storage MinIO.

4.3.1 Project and Code Structure

The project structure of this microservice is similar to the Credit microservice (Subsection 4.2.1), therefore, the package contents are not described in detail.



```
codenow
  config
    application.properties
src
  main
    java
      com.codenow.ticket
        config
        controller
        dto
        exception
        kafka
        model
        service
        tracing
        Application
    resources
      fonts
      img
  test
    java
      com.codenow.ticket
        model
        unit.service
    resources
      application.properties
```

Figure 4.4: PDF microservice project structure

4.3.2 PDF Ticket Generation

Two libraries were evaluated for generating tickets: Apache PDFBox[2] and iText[3]. After comparing their documentation, iText was selected due to its more high-level and straightforward interface for PDF document generation. This choice was deemed adequate for the task at hand, given the simple structure of the PDF ticket.

The Code Snippet 4.8 demonstrates the process of instantiating a PDF document in the iText library. In the beginning, an output stream in memory is instantiated and passed to a `PdfWriter` object. This object is afterwards used

[2]PDFBox, available at `https://pdfbox.apache.org`

[3]iText API documentation, available at `https://itextpdf.com/resources/api-documentation`

as a dependency for the `Document` object itself. After writing the ticket content, the document is closed. This flushes the actual data into the output stream, which can then be converted to a byte array. At the end of the method, the document is sent into Kafka as a part of a `PdfTicketCreatedEvent` object.

```java
public PdfTicket createPdfTicket(ReservationDTO reservationDTO) throws IOException,
     WriterException {
    ByteArrayOutputStream fos = new ByteArrayOutputStream();
    PdfWriter pdfWriter = new PdfWriter(fos);
    PdfDocument pdf = new PdfDocument(pdfWriter);
    Document document = new Document(pdf);

    writeTicketContent(document, reservationDTO);

    document.close();

    PdfTicket pdfTicket = new PdfTicket(reservationDTO.getId() + ".pdf",
         reservationDTO.getEmail(), fos.toByteArray());
    pdfTicketCreatedProducer.publish(new PdfTicketCreatedEvent(pdfTicket));

    return pdfTicket;
}
```

Code Snippet 4.8: PDF creation using the iText library

In the next Code Snippet 4.9, the actual content of the document is defined in the `writeTicketContent` method. The snippet demonstrates loading a custom font from the resources folder and then using it within the PDF document. After that, the CodeNOW logo is added. Finally, a paragraph with the user's name and the number of passengers the ticket is for is added. At the end, a line separating the paragraph is appended.

The ticket contains a QR code that encodes the reservation ID. In a hypothetical real-world scenario, it could be used to validate the ticket by a ticket inspector. To generate a QR code for the ticket, the library Google ZXing [4] was used. It was also chosen for the simplicity of its API. The Code Snippet 4.10 below shows the method used to generate the QR code. It instantiates an output stream and the QR code writer, which creates the QR code, writes it to the output stream and then returns the resulting byte array from the output stream. The resulting PDF ticket can be seen in the Figure 4.5.

[4]Google ZXing, available at `https://github.com/zxing/zxing`

```
private void writeTicketContent(Document document, ReservationDTO reservationDTO)
     throws IOException, WriterException {
    initFont(document);

    addCodenowLogo(document);

    Paragraph section1 = new Paragraph();
    addBoldTextToParagraph(section1, reservationDTO.getFirstName() + " " +
         reservationDTO.getLastName() + "\n");
    addBoldTextToParagraph(section1, "Amount of passengers: ");
    section1.add(String.valueOf(reservationDTO.getPassengers()));
    section1.setFontSize(17f);
    section1.setMarginBottom(0f);
    section1.setPaddingBottom(0f);
    document.add(section1);

    addLineSeparator(document);
    ...
    }

private void initFont(Document document) throws IOException {
    byte[] robotoMono = getClass().getResourceAsStream(ROBOTO_MONO_PATH).readAllBytes
         ();
    FontProgram fontProgram = FontProgramFactory.createFont(robotoMono);
    document.setFont(PdfFontFactory.createFont(fontProgram, PdfEncodings.IDENTITY_H));

}

private void addCodenowLogo(Document document) throws IOException {
    byte[] codenowLogo = getClass().getResourceAsStream(CODENOW_LOGO_PATH).
         readAllBytes();
    ImageData imageData = ImageDataFactory.create(codenowLogo);
    Image image = new Image(imageData)
            .setHeight(81)
            .setWidth(165)
            .setFixedPosition(1, 409, 745);
    document.add(image);
}
```

Code Snippet 4.9: Writing the PDF content using the iText library example

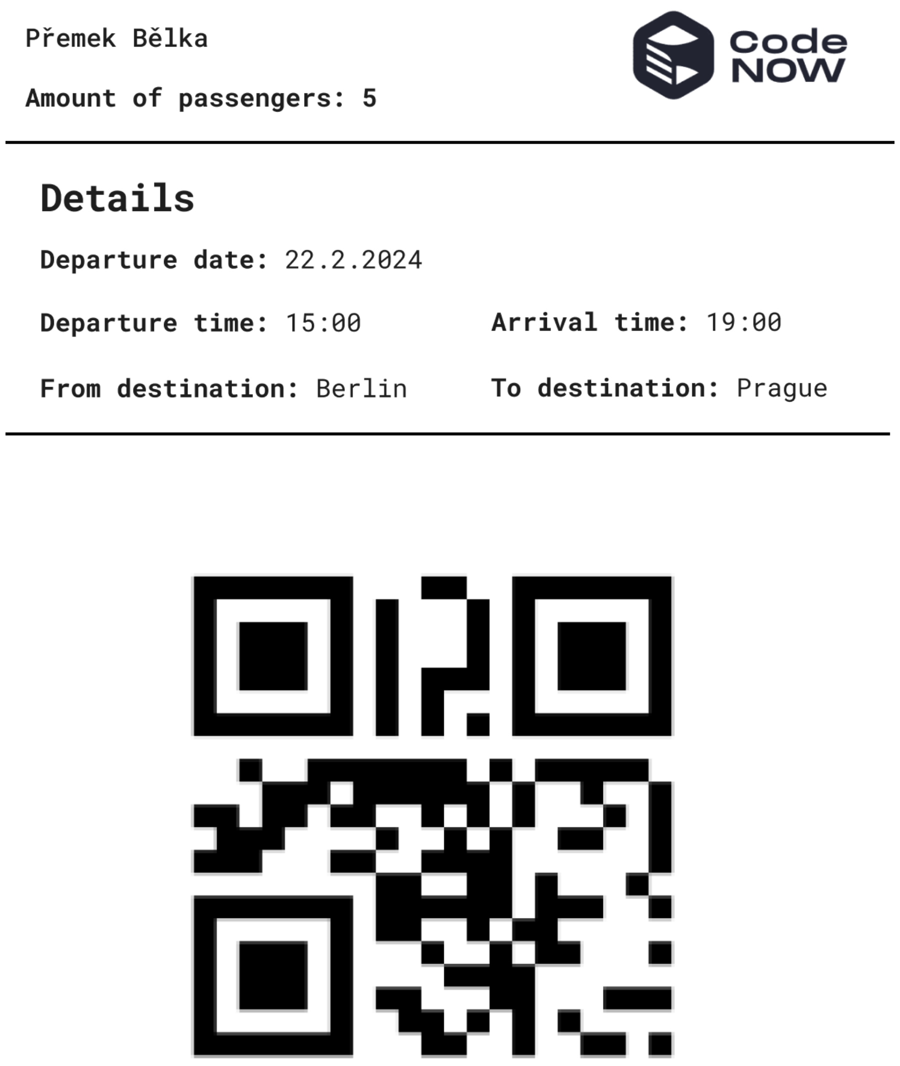
```
public byte[] createQrCode(String text) throws WriterException, IOException {
    ByteArrayOutputStream outputStream = new ByteArrayOutputStream();

    QRCodeWriter barcodeWriter = new QRCodeWriter();
    BitMatrix bitMatrix =
            barcodeWriter.encode(text, BarcodeFormat.QR_CODE, 400, 400);

    MatrixToImageWriter.writeToStream(bitMatrix, "PNG", outputStream);

    return outputStream.toByteArray();
}
```

Code Snippet 4.10: QRCode library ZXing code example



Přemek Bělka

Amount of passengers: 5

Details

Departure date: 22.2.2024

Departure time: 15:00 **Arrival time:** 19:00

From destination: Berlin **To destination:** Prague

Figure 4.5: PDF ticket

4.3.3 Interaction with MinIO

To interact with the MinIO object storage, the official Java client library was used. [30] The Code Snippet 4.11 shows the use of the library. The first method is concerned with saving a PDF ticket in a bucket. The method first checks if a bucket with a given name exists. If not, then a new bucket is created. After that, a ticket is persisted in the bucket. The second method retrieves ticket data from a bucket. The tickets in the bucket are uniquely identified by the reservation ID, as every file is named according to this rule: `filename = reservationID + ".pdf"`.

```java
public class MinioStorageServiceImpl implements StorageService {
...
    @Override
    public void savePdf(PdfTicket pdfTicket) throws FailedToSaveException {
        try {
            log.info("Saving pdf to Minio. Bucket name: " + bucketName);
            boolean bucketExists = client.bucketExists(
                    BucketExistsArgs.
                            builder().
                            bucket(bucketName).
                            build()
            );

            if (!bucketExists){
                client.makeBucket(
                        MakeBucketArgs
                                .builder()
                                .bucket(bucketName)
                                .build());
            }

            client.putObject(
                    PutObjectArgs.
                            builder().
                            bucket(bucketName).
                            object(pdfTicket.name).
                            stream(new ByteArrayInputStream(pdfTicket.data), pdfTicket.
                                 data.length, -1).
                            contentType(PDF_CONTENT_TYPE).
                            build()
            );
        }
        catch (Exception e) {
            throw new FailedToSaveException("Failed to save pdf to Minio. Bucket name:
                  " + bucketName + "\n"
                    + e.getMessage()
            );
        }
    }

    @Override
    public byte[] getTicketData(UUID reservationId) throws PdfNotFoundException {
        try {
            GetObjectResponse response = client.getObject(
                    GetObjectArgs.builder()
                            .bucket(bucketName)
                            .object(reservationId.toString() + ".pdf")
                            .build()
            );
            return response.readAllBytes();
        } catch (Exception e) {
            throw new PdfNotFoundException("Failed to retrieve ticket data from Minio
                 for reservationId: " + reservationId.toString() + "\n" + e.getMessage
                 ());
        }
    }
}
```

Code Snippet 4.11: MinIO interaction

4.4 Integration with the Rest of the System

To integrate the newly implemented microservices into the system and to support the redesigned processes, several changes had to be made on the part of other microservices. This section highlights the most important ones, beginning with the processing of new events, integration with the notification microservice, implementation of the API gateway and the data replication pattern, and lastly, the changes in the frontend microservice.

4.4.1 Messaging

As visualized in Section 2.2, all of the microservices in the system have to either consume or produce new messages. An example of such a producer and consumer class can be found below.

```java
@Service
public class PdfTicketCreatedProducer {
    private static final Logger log = LoggerFactory.getLogger(
         PdfTicketCreatedProducer.class);

    private final String topic;

    @Qualifier("mailkafkatemplate")
    private final KafkaTemplate<String, PdfTicketCreatedEvent> template;

    public PdfTicketCreatedProducer(@Qualifier("mailkafkatemplate") KafkaTemplate<
         String, PdfTicketCreatedEvent> template,
                                @Value("${spring.kafka.pdf.topic.pdf-ticket-created}"
                                     ) String topic) {
        this.template = template;
        this.topic = topic;
    }

    public void publish(PdfTicketCreatedEvent event) {
        template.send(topic, event);
        log.info("Published event to kafka: " + event.toString());
    }
}
```

Code Snippet 4.12: PdfTicketCreatedProducer class in the PDF microservice

```java
@Service
public class PdfTicketCreatedConsumer {

    private static final Logger log = LoggerFactory.getLogger(
         PdfTicketCreatedConsumer.class);

    private final MailServiceImpl mailService;

    public PdfTicketCreatedConsumer(MailServiceImpl mailService) {
        this.mailService = mailService;
    }

    // listen to new messages(mail requests) from kafka
    @KafkaListener(topics = "${spring.kafka.topic.pdf-ticket-created}",
         containerFactory = "pdfTicketCreatedKafkaListenerContainerFactory")
    public void mailListener(@Payload PdfTicketCreatedEvent pdfTicketCreatedEvent,
         Acknowledgment ack) throws InterruptedException {
        log.info("Pdf ticket created consumed.");
        log.info("Request: \n" + pdfTicketCreatedEvent);

        mailService.sendEmail(pdfTicketCreatedEvent);

        ack.acknowledge();
    }
}
```

Code Snippet 4.13: PdfTicketCreatedConsumer class in the Notification microservice

4.4.2 Notification Application

To fulfill the *FR8* (Section 2.2.1), the notification application had to be extended to allow adding attachments to emails being sent. This was done by adding a Kafka consumer for the `PdfTicketCreatedEvent`. Adding a new model `Attachment` class (Code Snippet 4.14) and then modifying the service's business logic to transform the `PdfTicketCreatedEvent` into the original `MailRequestDTO` object. The final modification to the business logic involves parsing an attachment and appending it to an email (Code Snippet 4.15).

```
public class Attachment {
    public byte[] data;

    public String mimeType;

    public String filename;

    public Attachment() {
    }

    public Attachment(byte[] data, String mimeType, String filename) {
        this.data = data;
        this.mimeType = mimeType;
        this.filename = filename;
    }

    public InputStreamSource getInputStreamSource() {
        return new ByteArrayResource(data);
    }
}
```

Code Snippet 4.14: New Attachment model class

```
public class MailServiceImpl implements MailService {
    @Override
    public void sendEmail(MailRequestDTO mail) {
    ...
        Attachment attachment = mail.getAttachment();
    if (attachment != null) {
        mimeMessageHelper.addAttachment(
                attachment.filename,
                attachment.getInputStreamSource(),
                attachment.mimeType
        );
    }

    mailSender.send(mimeMessageHelper.getMimeMessage());
    }

    @Override
    public void sendEmail(PdfTicketCreatedEvent pdfTicketCreatedEvent) {
    MailRequestDTO mailRequestDTO = new MailRequestDTO(
            pdfTicketCreatedEvent.getUserEmailAddress(),
            "",
            PDF_TICKET_EMAIL_SUBJECT,
            PDF_TICKET_EMAIL_BODY,
            new Attachment(
                    pdfTicketCreatedEvent.getData(),
                    PdfTicketCreatedEvent.MIME_TYPE,
                    "eticket.pdf"
            )
    );
    sendEmail(mailRequestDTO);
    }
}
```

Code Snippet 4.15: Modified business logic

4.4.3 API Gateway

The work implemented in this thesis is attempting to follow one of the very important philosophies contained in the API gateway pattern. In particular, it is the philosophy that application clients should not know about the structure of the system or the location of microservices themselves. Instead, all their requests should be processed by the API gateway (in this case, the API microservice) and routed to an appropriate place. Additionally, the API microservice in this instance also bears the responsibility for authenticating and authorizing users.

Code Snippet 4.16 shows a controller method in the API microservice used to download a PDF ticket. Firstly, authorization is performed in the `@PreAuthorize()` annotation. If it is successful, then the user gets redirected to the PDF microservice that retrieves the ticket from a MinIO storage and returns it in a response.

The implementation in this thesis serves just as a demonstration of the basic application of the pattern. In a real-life scenario, it would be preferable to use one of the many more sophisticated solutions, for example, the Spring Cloud Gateway. Which offers advanced routing support and several other features. [56]

```java
@RestController
@RequestMapping("/reservation")
public class ReservationControllerImpl {
...
    @Value("${custom.pdf-service.url}")
    private String PDF_SERVICE_URL;

    @GetMapping("/{id}/pdf-ticket")
    @PreAuthorize("hasRole('customer')")
    public Mono<ResponseEntity<Void>> getPdfTicket(@PathVariable UUID id) {
        String redirectUrl = PDF_SERVICE_URL + "/reservation/" + id + "/pdf-ticket";
        return Mono.just(ResponseEntity
                .status(HttpStatus.PERMANENT_REDIRECT)
                .header(HttpHeaders.LOCATION, redirectUrl)
                .build()
        );
    }
...
}
```

Code Snippet 4.16: API service authorization and redirect

4.4.4 Data Replication

The communication between the services in this thesis was designed to happen asynchronously, except for the front-end client. Although this brings a lot of advantages, it can also create a range of complicated situations. Such as when the front-end client requires a synchronous API to be able to show the account balance to the user. The API microservice can expose an HTTP endpoint, however, an HTTP endpoint would also have to be exposed at the side of the Credit microservice, to which the client would then be redirected. This approach is demonstrated in the previous section. Alternatively, the API microservice could send a GET request to the endpoint by itself and then just return the response to the client. Another solution to this problem, which does not involve synchronous communication with the Credit microservice is applying the data replication pattern.

Upon the API microservice deployment or every 12 hours, a snapshot of all user account balances is requested through Kafka from the Credit microservice. This data replica is persisted by the API microservice. The replica is then kept up-to-date by consuming events about all balance changes that occurred in the Credit microservice. This makes it possible to provide the data synchronously to the front-end client without introducing synchronous communication with the Credit microservice. However, this approach also brings several disadvantages, primarily because certain code, including the business logic for processing events, had to be duplicated in the API microservice. Also, according to the Eventual Consistency definition, the data may be in an inconsistent state at times (Section 3.6). Therefore, a reservation could be created for a user who does not have enough funds in their account. To prevent this situation from happening, a secondary validation is taking place in the Credit microservice, which owns the data, during the processing of this payment. If this validation fails, an event in a `FAILED` state is emitted, and the alternative flow of the topping up account balance process is executed (Figure 2.10). [36]

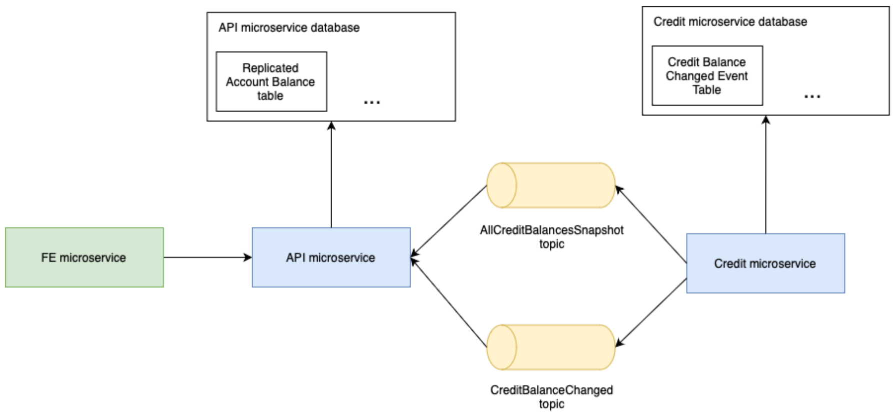

API microservice database
Replicated Account Balance table
...
Credit microservice database
Credit Balance Changed Event Table
...
FE microservice
API microservice
AllCreditBalancesSnapshot topic
Credit microservice
CreditBalanceChanged topic

Figure 4.6: Data replication visualization

4.4.5 Front-End

The last step of integrating the new changes into the system involved extending and modifying the React front-end part of the application. This is visualized on the wireframes in the solution design Subsection 2.2.7.

The described example is concerned with the implementation of a React component showing information about an existing reservation (Figure 2.14).

As an input, the component receives a `ReservationComponentProps` object, which contains all necessary information about the reservation to be displayed. The input is passed as a prop. Props are an important concept in the React framework, which is used to pass data between components. Props could be briefly described as component input arguments passed in a unidirectional manner. An important aspect of working with props is that they are treated as immutable by the component that receives them. [31]

```
type ReservationComponentProps = {
    reservation: ReservationResponse;
};

export const ReservationComponent = ({reservation}: ReservationComponentProps) => {
...
}
```

Code Snippet 4.17: Reservation component declaration

The component defines two methods (Code Snippet 4.18). The method `cancelMutation` is used to send a request to refund a reservation. After sending the request, the user is either informed that the request is being processed or that it failed. The second method, `onDownload`, handles the process of downloading a PDF ticket. It first requests the data, if the request is successful and the data is received, then a `BLOB` object is created, and a browser download is triggered by clicking an unrendered helper link element. If the download button links directly to the API service URL, then the user would get redirected. Therefore, this approach is used to prevent the redirect.

```
export const ReservationComponent = ({reservation}: ReservationComponentProps) => {
const cancelMutation = useCancelReservation(reservation.id, {
    onSuccess: () => {
        enqueueSnackbar("Reservation is being cancelled", {variant: "success"});
    }, onError: () => {
        enqueueSnackbar("Error cancelling reservation", {variant: "error"});
    }
});

const onDownload = (reservationId: string) => {
    // @ts-ignore
    const baseUrl = window.env.API_BACKEND_URL;
    const url = `${baseUrl}/reservation/${reservationId}/pdf-ticket`;
    HttpService().cacheAxios.get(url, { responseType: 'blob' })
        .then(response => {
            if (response.status === 200) { // File exists
                const blob = new Blob([response.data], { type: 'application/pdf' });
                const url = window.URL.createObjectURL(blob);
                const link = document.createElement('a');
                link.href = url;
                link.download = 'eticket.pdf';
                link.click();
            } else {
                enqueueSnackbar("Error downloading ticket", {variant: "error"});
            }
        })
        .catch(() => {
            enqueueSnackbar("Error downloading ticket", {variant: "error"});
        })
};
```

Code Snippet 4.18: Reservation component methods

The last part of the component (Code Snippet 4.19) defines the structure of the rendered component in the JSX markup language. JSX is a syntax extension for JavaScript that makes it possible to write HTML-like code within JavaScript files. [32] The provided snippet renders the refund and download button within the grid of the reservation element. Upon clicking the buttons the respective methods described in the previous paragraph are called.

```
return (
...
<Grid item xs={4} sx={{ display: 'flex', flexDirection: 'column', justifyContent: '
    flex-end' }}>
    <Stack spacing={0.5} direction="row" alignItems="flex-end">
        <Button
            variant="contained"
            onClick={() => cancelMutation.mutate(null)}
            disabled={reservation.status != "ACTIVE"}
            sx={{minWidth: 75}}
        >
            <Typography fontWeight="bold" color="white">
                Refund
            </Typography>
        </Button>
        <Button
            variant="contained"
            disabled={reservation.status != "ACTIVE"}
            onClick={() => onDownload(reservation.id)}
            sx={{minWidth: 75}}
        >
            <Typography fontWeight="bold" color="white">
                Download ticket
            </Typography>
        </Button>
    </Stack>
</Grid>
...
)
```

Code Snippet 4.19: Reservation component JSX

4.5 Distributed Tracing Instrumentation

To instrument the code for distributed tracing, the library Spring Cloud Sleuth included in the CodeNOW templates was used. One of its very useful features is its auto-configuration capability. This feature makes it possible to integrate distributed tracing into Spring Boot applications without the need for extensive manual setup. [52]

The only manual setup that needed to be done was the following:

- The URL of an external storage for collecting the traces had to be specified. More about the Jaeger project can be found in the Deployment Chapter 6.6.
- The propagation of the trace ID had to be enabled for Kafka messages. It is achieved by appending the trace ID to a B3 header for each message. The B3 header is a widely spread propagation standard for messaging systems. [53]
- An aspect class was added to create spans for each interaction with a Spring repository (Code Snippet 4.21). The aspect intercepts calls to

methods within Spring Data repositories, starts a tracing span before the method execution, and ends the span after the method execution.

```yaml
zipkin:
  enabled: true
  baseUrl: http://tracing-jaeger-collector.tracing-system:9411
sleuth:
  propagation:
    type: B3
    tag:
      enabled: true
  messaging:
    kafka:
      enabled: true
```

Code Snippet 4.20: application.yaml Sleuth

```java
@Aspect
@Component
public class RepositoryTracingAspect {
    private static final String TRACING_TYPE = "repository";
    private TracingHelper tracingHelper;

    @Autowired
    public RepositoryTracingAspect(final TracingHelper tracingHelper) {
        this.tracingHelper = tracingHelper;
    }

    @Around("within(org.springframework.data.repository.CrudRepository+)")
    public Object traceRepositoryCalls(ProceedingJoinPoint joinPoint) throws
         Throwable {
        String className = joinPoint.getSignature().getDeclaringTypeName();
        String targetMethod = joinPoint.getSignature().getName();

        Span span = tracingHelper.createClientSpan(targetMethod, TRACING_TYPE,
             className);

        Object proceed = joinPoint.proceed();

        span.end();

        return proceed;
    }
}
```

Code Snippet 4.21: RepositoryTracingAspect

4.6 Configuration

During the development of any application, it is crucial to manage configuration variables correctly for different environments. For example, data for connecting to external services or the configuration of the application server. The configuration management was based on the 12-Factor Application recommendation that advises using environment variables (Section 3.3).

The example showcases a part of the `application.yaml` configuration file (Code Snippet 4.22) that Spring Boot utilizes for configuring various aspects of the application. Three values regarding a Stripe service connection are injected into the file from environment variables. In the next step, these values are injected into the actual Java code, as can be seen in the next snippet of the `StripePaymentController` class (Code Snippet 4.23).

```yaml
custom:
  stripe:
    webhook: ${STRIPE_CREDIT_WEBHOOK_SECRET}
    secret-key: ${STRIPE_CREDIT_SECRET_KEY}
    allowed-redirect-domain-name: ${STRIPE_CREDIT_ALLOWED_REDIRECT_DOMAIN_NAME}
```

Code Snippet 4.22: application.yaml ENV variables

```java
@RestController
@RequestMapping("/stripe-payment")
public class StripePaymentController {

    @Value("${custom.stripe.secret-key}")
    private String stripeSecretKey;

    @Value("${custom.stripe.allowed-redirect-domain-name}")
    private String allowedRedirectDomainName;

...
```

Code Snippet 4.23: StripePaymentController with injected configuration values

Chapter 5

Tests

5.1 Unit Tests

Unit test is a type of test that tests a single unit of code in isolation. The unit tested in the case of this project is a single public method. In total, 22 unit tests were developed in the PDF and Credit microservices to cover the business logic in domain entities and service classes.

The JUnit5 testing framework in combination with Mockito library was used as both of these tools are included and recommended in the CodeNOW Spring Boot microservice template.

The example (Code Snippet 5.1) shows a unit test that controls whether an event with correct data would be published to Kafka when the method `spendCredits` in a `CreditService` is called. Because unit tests require isolation from other components, the dependency on Kafka and other external services is removed using the so-called mock objects, and only the core logic of the method is tested. A mock object is an object that imitates the behavior of an object with which the class under test has an association. They are used to assist with unit testing. [17] The Mockito library makes it possible to easily create such objects and also define their behavior. The example (Code Snippet 5.1) follows an ARRANGE-ACT-ASSERT method of structuring unit tests, which divides the test into three parts to allow for faster reading and easier future maintenance. The respective parts perform these tasks:

ARRANGE A setup method is called before executing a test case to construct the `CreditService` class using mocked dependencies. Then, the behavior of these mocked dependencies is defined. Additionally, on the 3rd and 4th lines of the test, setup is performed to intercept the event created within the method, ensuring later that its data are correct.

ACT The tested method `spendCredits` is called, and its output is saved to a local variable.

ASSERT The intercepted event is retrieved, the correctness of its data is asserted, and that the publish method was called with it.

```java
@TestPropertySource(locations = "classpath:application.yaml")
public class CreditServiceImplTest {
...
    @BeforeEach
    public void setup() {
        creditBalanceChangeEventRepositoryMock = Mockito.mock(
             CreditBalanceChangeEventRepository.class);
        refundedRepositoryMock = Mockito.mock(RefundedRepository.class);
        accountBalanceSnapshotRepositoryMock = Mockito.mock(
             AccountBalanceSnapshotRepository.class);
        creditsSpentProducerMock = Mockito.mock(CreditsSpentProducer.class);
        creditsRefundedProducerMock = Mockito.mock(CreditsRefundedProducer.class);
        creditsBoughtMock = Mockito.mock(CreditBalanceChangedEvent.class);
        creditService = new CreditServiceImpl(
                creditBalanceChangeEventRepositoryMock,
                refundedRepositoryMock,
                accountBalanceSnapshotRepositoryMock,
                creditsSpentProducerMock,
                creditsRefundedProducerMock
        );
    }

    @Test
    public void
         spendCredits_spendPositiveCreditsForCorrectUser_creditsSpentEventPersisted()
         throws ExecutionException, InterruptedException {
        Mockito.when(creditsBoughtMock.getBalanceChange()).thenReturn(100L);
        Mockito.when(creditBalanceChangeEventRepositoryMock.findAllByUserId("user1")).
             thenReturn(List.of(creditsBoughtMock));
        ArgumentCaptor<CreditBalanceChangedEvent> creditChangedEventCaptor =
             ArgumentCaptor.forClass(CreditBalanceChangedEvent.class);
        doAnswer(invocation -> {
            CreditBalanceChangedEvent event = invocation.getArgument(0);
            return event;
        }).when(creditBalanceChangeEventRepositoryMock).save(creditChangedEventCaptor.
             capture());

        UUID returnedEventId = creditService.spendCredits(100L, "user1");

        CreditBalanceChangedEvent capturedCreditChangedEvent =
             creditChangedEventCaptor.getValue();
        Mockito.verify(creditBalanceChangeEventRepositoryMock, Mockito.times(1)).save(
             capturedCreditChangedEvent);
        Assert.assertEquals(-100L, (long) capturedCreditChangedEvent.getBalanceChange
             ());
        Assert.assertEquals("user1", capturedCreditChangedEvent.getUserId());
        Assert.assertEquals(returnedEventId, capturedCreditChangedEvent.getId());
    }
}
```

Code Snippet 5.1: Unit test example

5.2 Integration Tests

Integration tests are types of tests that focus on testing the interactions between components in system. [19]

5.2.1 Karate Tests

In the early stages of the work on the thesis, a REST API was implemented for the Credit microservice. The Karate testing framework was used to implement integration tests for the REST interface. Later, it was decided that all communication with the service would asynchronously take place using a messaging queue. The tests were still included in this version since the application is supposed to mainly serve as a reference and the tests still verify the core business logic nonetheless.

Karate is an open-source framework for writing automated API and UI tests. One of the big advantages of the framework is that it uses the Gherkin language to write tests, which allows tests to be written in an almost natural language and requires almost no prior programming knowledge (Code Snippet 5.2). These technologies were chosen based on the successful demonstration of implementing UI tests in the Ticket Reservation Application in the previous thesis by Robin Vávra [13]. However, as an alternative to Karate, the framework JBehave, which offers similar functionality, could also be potentially used. [20]

To make the tests run in any environment and not depend on external services, it was necessary to fulfill the dependency on the database and messaging queue. For this purpose, the H2DB and Spring Kafka Test libraries were used, which allow the services to be temporarily turned on locally during testing, replacing the connection to the actual instances of these services.

```
Feature: PaymentController

  Scenario: 100 credits successfully spent for user id: 1.

    Given url baseUrl + '/payment'
    And request {userId: 1, amount: 100}
    When method POST
    Then status 201

    ...
```

Code Snippet 5.2: Sample test written in Gherkin

```java
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
@EmbeddedKafka(partitions = 1, brokerProperties = { "listeners=PLAINTEXT://localhost
    :9092", "port=9092" })
@TestPropertySource(locations = "classpath:application.yaml")
public class PaymentControllerTest {

    @LocalServerPort
    private String localServerPort;

    @Karate.Test
    Karate feature() {
        return Karate
                .run("PaymentController")
                .relativeTo(getClass())
                .systemProperty("karate.port", localServerPort);
    }
}
```

Code Snippet 5.3: Karate Java testing class

5.2.2 Contract Tests

For lack of time, no contract tests were implemented as a part of this work. However, it is still described how such a test could be implemented, as contract testing poses an important concept for testing microservice applications. It is also theoretically introduced in the Subsection 3.2.8.

An example in the case of this application could be a PDF microservice being the producing side of the `PdftTicketCreatedEvent`. And the Notification microservice being the consuming side of the event. A Consumer-Driven Contract could be specified by the Notification microservice, which would define what data it needs to be able to deliver an email message. In this case, it would be the PDF ticket data, an email address, and the name of the file.

A tool that could be used to implement a contract test for the PDF and Notification microservices is the Pact framework. It is an open-source tool that makes it possible to develop Consumer-Driven Contract tests for HTTP APIs as well as for asynchronous APIs using messaging queues. [22]

Pact utilizes a broker, which contains all of the Consumer Contracts defined by the consumer sides. At first, these contracts are published to the broker and used for the execution of tests by the consumer side, where a message created by a mocked producer is sent to the consuming service to validate correct processing. In the next stage, the contracts are fetched by the targeted producer sides and it is validated that the messages they send fulfill the expected requirements imposed by the contracts. The broker also contains information on whether the contract was successfully used in tests on both the consuming and producing sides, since the test results are sent back to the broker after the tests are run. [23][Chapter 2.4.2]

5.3 End-to-End Tests

End-to-End tests (E2E tests) are types of tests, in which business processes are tested from start to finish under production-like circumstances. [24]

To test the application, one E2E test was implemented using the Selenium framework. Selenium is a popular testing framework used for automating browsers. The framework makes it possible to write automatic E2E tests by specifying the interactions a browser should execute on a given web page. [26] Another very popular alternative is the Cypress framework, which offers similar functionality in regards to automating web browsers. [27]

The test focuses on testing the process of creating and then refunding a reservation. It executes the following steps. The Code Snippet 5.4 is shortened and does not contain the setup and login parts.

1. Open the index page, navigate to the login page, fill in the login information, and click the login button.
2. Navigate to the buy credits page, click the button to pay 200 euros, and fill in the testing stripe payment information.
3. Wait up to 30 seconds until the payment is processed and the account balance shows the according number.
4. Search for a seat from Prague to Berlin and create a reservation.
5. Wait up to 30 seconds until the reservation is processed and the account balance shows the according number.
6. Navigate to the reservations page and click the button to refund the last reservation.
7. Wait up to 30 seconds until the refund is processed and the account balance shows the according number.
8. Wait for 15 seconds, then fetch the last two unread emails from a testing mailbox belonging to the test user. Verify their subject and the fact that the reservation creation email also has a file attached (the ticket). Flag the emails as read.

Also, certain preconditions have to be met before executing the test.

- The whole application (all microservices) is running, including all of its external dependencies.
- A testing user is present in the system.
- The price of a trip from Prague to Berlin is 200 euros or less, and there is at least one empty seat.

```java
@Test
public void createReservationE2E() throws InterruptedException, MessagingException,
     IOException {
    IndexPage indexPage = new IndexPage(driver);
    Integer initialAccountBalance = indexPage.getAccountBalance();
    indexPage.goToBuyCredits();

    BuyCreditsPage buyCreditsPage = new BuyCreditsPage(driver);
    buyCreditsPage.clickBuyTwoHundredCredits();

    StripePage stripePage = new StripePage(driver);
    stripePage.payTwoHundredEuro();
    Integer expectedAccountBalanceAfterTopUp = initialAccountBalance + 200;

    indexPage = new IndexPage(driver);
    WebDriverWait wait = new WebDriverWait(driver, Duration.ofSeconds(30));
    wait.until(ExpectedConditions.textToBePresentInElement(
           indexPage.getAccountBalanceSpan(), "Account Balance : " +
                expectedAccountBalanceAfterTopUp + " euro")
    );
    indexPage.search("Prague", "Berlin");

    ReservationsSearchedPage reservationsSearchedPage = new ReservationsSearchedPage(
         driver);
    reservationsSearchedPage.clickFirstReservationButton();

    ReservationCreatePage reservationCreatePage = new ReservationCreatePage(driver);
    reservationCreatePage.clickCreateReservationButton();
    Integer reservationPrice = reservationCreatePage.getReservationPrice();
    Integer expectedAmountLeft = expectedAccountBalanceAfterTopUp - reservationPrice;
    wait.until(ExpectedConditions.textToBePresentInElement(
           indexPage.getAccountBalanceSpan(), "Account Balance : " +
                expectedAmountLeft + " euro")
    );

    indexPage = new IndexPage(driver);
    indexPage.goToReservations();

    ReservationsPage reservationsPage = new ReservationsPage(driver);
    reservationsPage.clickRefundLastReservation();
    wait.until(ExpectedConditions.textToBePresentInElement(
           indexPage.getAccountBalanceSpan(), "Account Balance : " +
                expectedAccountBalanceAfterTopUp + " euro")
    );

    Thread.sleep(15000);
    List<MailMessage> mails = MailReader.getLastTwoUnreadEmails();
    Assertions.assertEquals("Reservation canceled", mails.get(0).subject);
    Assertions.assertEquals("Ticket for your reservation.", mails.get(1).subject);
    Assertions.assertTrue(mails.get(1).hasAttachments);
}
```

Code Snippet 5.4: Create reservation E2E test

The Page Object Model (POM) pattern is applied in the test. The pattern is used to model single UI pages as separate classes. These classes then expose an interface, which allows the client to interact with elements on the page. The main benefits of the pattern are avoiding code duplication and improving code reusability. [25][Chapter 7] The code example shows the POM for a Keycloak login page. The login input elements are found using the `@FindById` annotation, and then they are initialized in the constructor. Furthermore, it contains a login method that fills in the fields and clicks the login button.

```java
public class LoginPage {
    private WebDriver driver;
    @FindBy(id = "kc-login")
    private WebElement loginButton;

    @FindBy(id = "username")
    private WebElement usernameInput;

    @FindBy(id = "password")
    private WebElement passwordInput;

    private String username;

    private String password;

    public LoginPage(WebDriver driver, String username, String password) {
        this.driver = driver;
        this.username = username;
        this.password = password;

        PageFactory.initElements(driver, this);
    }

    public void login() {
        usernameInput.sendKeys(username);
        passwordInput.sendKeys(password);

        loginButton.click();
    }
}
```

Code Snippet 5.5: Login page POM

To retrieve the emails from the mailbox, a JavaMail API reference implementation was used. The library provides a simple way for Java applications to communicate with mail servers. [28] The provided example shows how, by using the library, a connection is created, and then the last two unread emails are fetched and marked, as seen in Code Snippet 5.6.

```java
public static List<MailMessage> getLastTwoUnreadEmails() throws MessagingException,
     IOException {
    Properties props = new Properties();
    props.setProperty("mail.store.protocol", PROTOCOL);
    props.setProperty("mail.imaps.host", HOST);
    props.setProperty("mail.imaps.port", PORT);
    props.setProperty("mail.imaps.starttls.enable", STARTTLS);
    props.setProperty("mail.imaps.auth", AUTH);

    Session session = Session.getDefaultInstance(props, null);
    Store store = session.getStore(PROTOCOL);
    store.connect(EMAIL_ADDRESS, PASSWORD);

    Folder inbox = store.getFolder("INBOX");
    inbox.open(Folder.READ_WRITE);

    FlagTerm ft = new FlagTerm(new javax.mail.Flags(javax.mail.Flags.Flag.SEEN),
         false);
    Message[] messages = inbox.search(ft);

    // Fetch only the last two unread messages
    int count = messages.length >= 2 ? 2 : messages.length;
    List<MailMessage> mails = new ArrayList<>();
    for (int i = messages.length - 1; i >= messages.length - count; i--) {
        mails.add(new MailMessage(messages[i].getSubject(), hasAttachments(messages[i
             ])));
        messages[i].setFlag(Flags.Flag.SEEN, true);
    }

    inbox.close(false);
    store.close();

    return mails;
}

private static boolean hasAttachments(Message message) throws MessagingException,
     IOException {
    return message.getContent() instanceof Multipart;
}
```

Code Snippet 5.6: JavaMail API

5.4 Usability Tests

Usability tests aim to test the functionality of an application by observing real users interacting with it as they are trying to complete assigned tasks. [29]

The usability testing was performed with three independent testers. They were asked to complete the following task.

Berlin to Prague trip You need to travel to Prague from Berlin on 25.4.2024 with your friend. Book a ticket for two passengers and download it from your email mailbox.

After completing this scenario, they were presented with an additional task.

Trip refund Your friend cannot travel with you anymore. Refund the ticket and book the same trip, but only for you. This time, download the ticket directly from the website.

After collecting their feedback, several problems were identified.

Tester 1 Could not top up the account, because they overlooked the testing Stripe card information on the buy credits page.

Tester 2 Did not encounter any problems.

Tester 3 Noticed the emailed PDF ticket contained wrong information. The arrival and departure stations were the same.

In response to the feedback, the testing Stripe card information was made more visible and the PDF ticket information was corrected. After retests, no further defects and problems were discovered.

5.5 Testing Environment

The unit and Karate tests are implemented in such a way that they do not depend on the environment in which they are run. Thus, they can be run both on CodeNOW and locally during development. On CodeNOW, running them and getting a successful result is a prerequisite for creating a `.jar` file that can then be deployed. They are triggered using the Maven tool, which integrates them into the build process.

The E2E tests were not integrated into CodeNOW's deployment process. In the current state, they were only run locally. To do so, it was necessary to manually start all of the microservices and their external dependencies using a docker-compose file.

The usability tests were performed on the deployed version of the application on CodeNOW.

Chapter 6
Deployment

This chapter introduces the necessary terminology used on the CodeNOW platform. The basic process of containerizing an application using Docker is explained. Then the intricacies of deploying a MinIO service are described. The rest of the chapter is focused on selected features that implement patterns mentioned in the related theory part of this work.

6.1 Terminology

To be able to use CodeNOW, certain terminology used by the platform needs to be understood. [46]

Component A single microservice in an application. Can be created using a predefined template that contains preconfigured libraries and frameworks to be able to quickly start the development. Such as the template Java 17 and Spring Boot 3.1.0 with Maven 3.9.1.

Application A collection of components.

Package A collection of versioned artifacts built from components that can be deployed to an environment together with a deployment configuration.

Deployment configuration A set of configuration files containing environment variables. A deployment configuration is always associated with an environment and a package.

Environment A Kubernetes namespace. On a conceptual level, it is a place for an application to be deployed. Such as a production environment or a development environment.

Managed service An instance of a pre-configured service, such as Apache Kafka that can be requested and created automatically. The components can be then easily connected to such a service.

6.2 Containerization

Before deploying a microservice to a Kubernetes cluster, it must first be containerized. This can be achieved by encapsulating it within a Docker container. Below are the steps to containerize either the Credit or PDF microservice using Docker.

Building an artifact First, it is necessary to build a `.jar` file for the Spring Boot application using Maven. This creates a packaged version of the application that can be run.

Defining a Dockerfile A Dockerfile is a text document that contains all the commands and Docker-specific instructions, which are necessary to call to create a Docker image. It encompasses elements such as the necessary dependencies to include in the container, the ports to expose and their mapping in the container, shell commands to run, the definition of environment variables, and much more. [58]

A Docker image After the Dockerfile is defined, it can be built using the `docker build` command. Additionally, it can be published to a remote repository, from which it can later be retrieved. The image can be used to create and run a container by using the `docker run` command.

The whole process is automatically handled by the CodeNOW platform when a component release is being made. This, however, only applies to components created using the standard predefined templates. As seen in the next section about MinIO a custom Dockerfile definition had to be provided.

6.3 MinIO

Formerly, MinIO used to be provided at CodeNOW as a managed service. Therefore, deploying and using a MinIO instance in an application required minimal effort. Unfortunately, at the start of writing of this thesis, MinIO was no longer provided as a preconfigured managed service, therefore, a custom deployment had to be made using a generic Docker component option. The deployment of the MinIO database on CodeNOW posed a significant challenge as it required an understanding of certain parts of the Kubernetes container orchestrating technology.

As a first step, a Dockerfile based on the MinIO image had to be defined.

```
# Inherit all the files, configurations, and environment settings present in the
    MinIO base image.
FROM minio/minio

# Copy the codenow config file
COPY codenow/config/config.yaml /

# Expose ports
EXPOSE 9000 9001

# Set the data directory as the volume
VOLUME /data

# Command to start MinIO server and the admin console
CMD ["server", "--console-address", ":9001", "/data"]
```

Code Snippet 6.1: MinIO Dockerfile

By creating the generic Docker component, a repository with default template files gets generated. The files contain default values to deploy the containerized application specified in the Dockerfile to the Kubernetes cluster. However, several adjustments had to be made to achieve a successful deployment. The live and ready probe paths had to be specified accordingly. Also, it had to be specified that the MinIO database will be exposed to the other pods internally at port 9000 using the `ClusterIP Service`. This port is also used for the `Istio Virtual Service` resource, which exposes the service to the outside of the Kubernetes cluster.

The configuration values are specified in a `Values.yaml` file and then get injected into the actual definition files with the use of the Helm templating engine. Only the values shown in the `Values.yaml` snippet had to be changed; the rest was left as pre-generated.

```yaml
...
liveProbePath: /minio/health/live
readyProbePath: /minio/health/ready

service:
  type: ClusterIP
  port: 9000
  externalEndpointEnabled: true
...
```

Code Snippet 6.2: Values.yaml snippet

```yaml
apiVersion: v1
kind: Service
metadata:
  name: {{ .Values.codenow.componentRuntimeName }}
  namespace: {{ .Release.Namespace }}
  labels:
    app.kubernetes.io/name: {{ .Values.codenow.componentRuntimeName }}
spec:
  type: {{ .Values.service.type }}
  ports:
    - port: {{ .Values.service.port }}
      protocol: TCP
      name: minio
  selector:
    app.kubernetes.io/name: {{ .Values.codenow.componentRuntimeName }}
{{- if .Values.service.externalEndpointEnabled }}
apiVersion: networking.istio.io/v1beta1
kind: VirtualService
metadata:
  name: {{ .Values.codenow.componentRuntimeName }}-service
  namespace: {{ .Release.Namespace }}
spec:
  hosts:
  - {{ .Values.codenow.componentRuntimeName }}-{{ .Release.Namespace }}.{{ .Values.
       codenow.domainName }}
  gateways:
  - istio-system/public-gateway
  http:
  - match:
    - uri:
       prefix: /
    route:
    - destination:
        host: {{ .Values.codenow.componentRuntimeName }}.{{ .Release.Namespace }}.svc.
             cluster.local
        port:
          number: {{ .Values.service.port }}
{{- end }}
```

Code Snippet 6.3: Service.yaml

Additionally, for the data to be persistent in case of a redeployment, a new cluster resource of the type `PersistentVolumeClaim` had to be defined. It represents a request for storage to be provisioned dynamically by the Kubernetes cluster. [45]

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
  storageClassName: standard
```

Code Snippet 6.4: PersistentVolumeClaim definition

6.4 Credit and PDF Microservice

Together with the MinIO service, two more components in the existing application had to be created. The Credit and PDF microservice components were created using a pre-configured Java and Spring Boot template. Their configuration and deployment followed a standard step-by-step procedure described in the CodeNOW manual. Therefore it won't be further detailed here. [1]

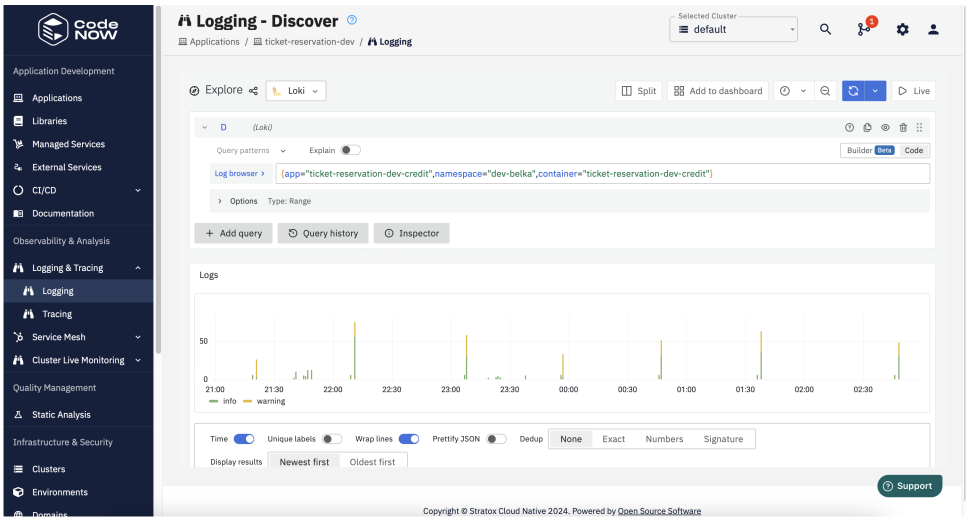
6.5 Log Aggregation

To be able to easily obtain insights into the operations of a deployed application, the tool Loki is integrated into the CodeNOW platform. Loki is a log aggregation system that makes it possible to store and query logs from applications. [49]

Loki makes it easy to query and see logs based on several factors, such as the source microservice, time, severity level, and custom labels, facilitating efficient troubleshooting and analysis.

Since the 12-Factor application recommendation about logging (Section 3.3) was followed, it is possible to store the logs into Loki when the application is deployed on CodeNOW. While also ensuring that the logs can still be output to a terminal during local development.

[1]Step-by-step CodeNOW deployment, available at `https://docs.codenow.com/admin-manuals/deploy-application`



```
{app="ticket-reservation-dev-credit",namespace="dev-belka",container="ticket-reservation-dev-credit"}
```

Type: Range

Time ✓ · Unique labels · Wrap lines ✓ · Prettify JSON · Dedup: [✓ None] Exact Numbers Signature

Display results: [✓ Newest first] Oldest first

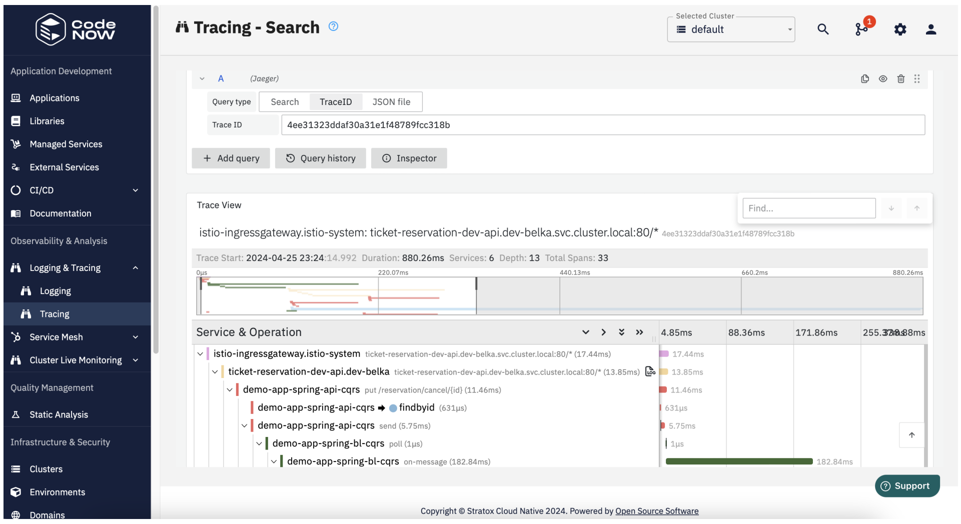
Copyright © Stratox Cloud Native 2024. Powered by Open Source Software

Figure 6.1: Query for retrieving logs from the Credit microservice using Loki

6.6 Distributed Tracing

Another tool integrated by CodeNOW to facilitate better observability is Jaeger. It is an open-source distributed tracing system that gathers data about the flow of requests through a distributed system, providing insights into performance bottlenecks, latency issues, and dependencies between microservices. [50]

Since the application has been instrumented accordingly (Section 4.5), the request flow can be inspected by providing a trace ID, which can be found in the logs or by viewing stored messages in Kafka.



Query type: Search [✓ TraceID] JSON file

Trace ID: 4ee31323ddaf30a31e1f48789fcc318b

istio-ingressgateway.istio-system: ticket-reservation-dev-api.dev-belka.svc.cluster.local:80/* 4ee31323ddaf30a31e1f48789fcc318b

Trace Start: 2024-04-25 23:24:14.992 Duration: 880.26ms Services: 6 Depth: 13 Total Spans: 33

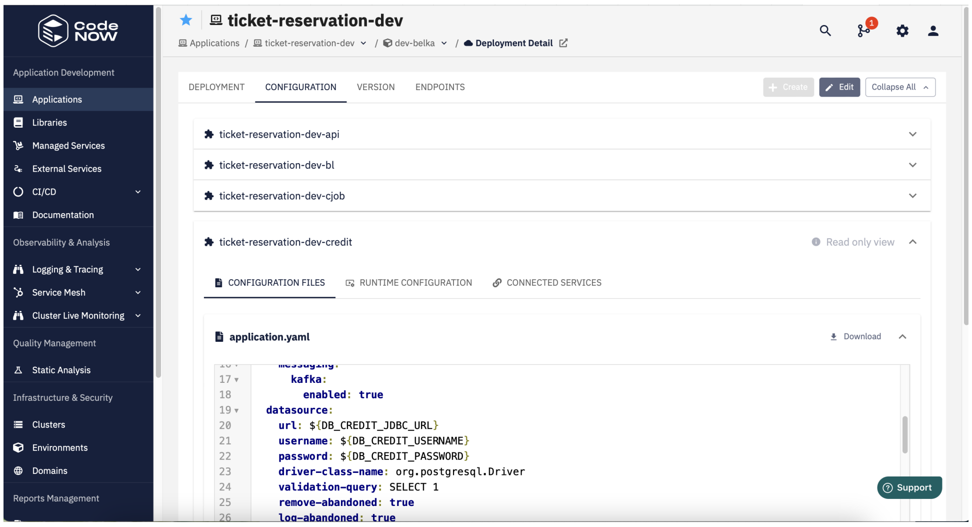
```
Service & Operation
istio-ingressgateway.istio-system  ticket-reservation-dev-api.dev-belka.svc.cluster.local:80/* (17.44ms)   17.44ms
  ticket-reservation-dev-api.dev-belka  ticket-reservation-dev-api.dev-belka.svc.cluster.local:80/* (13.85ms)   13.85ms
    demo-app-spring-api-cqrs  put /reservation/cancel/{id} (11.46ms)   11.46ms
      demo-app-spring-api-cqrs ➜ findbyid (631µs)   631µs
      demo-app-spring-api-cqrs  send (5.75ms)   5.75ms
        demo-app-spring-bl-cqrs  poll (1µs)   1µs
          demo-app-spring-bl-cqrs  on-message (182.84ms)   182.84ms
```

Copyright © Stratox Cloud Native 2024. Powered by Open Source Software

Figure 6.2: Jaeger visualization of a refund reservation request

6.7 Configuration Management

As microservice applications grow, it may be difficult to keep track of the configuration of each service in different versions. For this reason, CodeNOW associates a deployment configuration with each package for each environment. This feature makes it possible to easily rollback to an older version of a package because the deployment configuration for different environments is versioned and saved with it. The deployment configuration is created each time a new package version is deployed to an environment it has previously not been deployed to. In the event that an older version of the package is deployed, the new deployment configuration is automatically based on it. CodeNOW allows users to modify the configuration files directly from the browser UI. The platform also allows users to inspect what exact configuration files were used with which service after it was deployed.

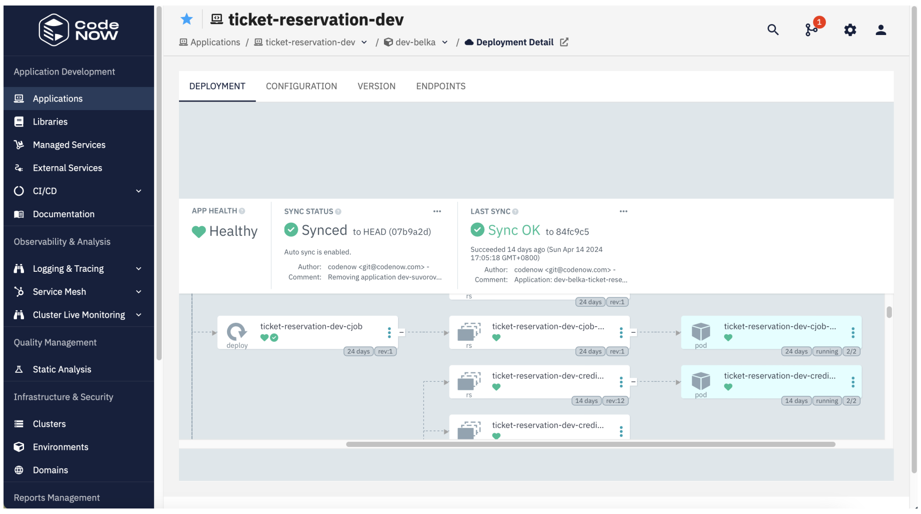


```yaml
    messaging:
      kafka:
        enabled: true
  datasource:
    url: ${DB_CREDIT_JDBC_URL}
    username: ${DB_CREDIT_USERNAME}
    password: ${DB_CREDIT_PASSWORD}
    driver-class-name: org.postgresql.Driver
    validation-query: SELECT 1
    remove-abandoned: true
    log-abandoned: true
```

Figure 6.3: Deployment configuration example

6.8 Cluster Monitoring

To oversee the status of Kubernetes resources associated with a deployed application, the tool ArgoCD is available on CodeNOW. ArgoCD is a tool used to manage and automate the deployment of applications on Kubernetes clusters. It follows the GitOps philosophy that the desired application state and infrastructure should be declaratively defined in a Git repository. ArgoCD actively monitors the deployed application's state in the Kubernetes cluster and compares it to the definition in the git repository. This information is presented to the user in a user-friendly dashboard interface, allowing for easy tracking of any discrepancies and facilitating efficient management of application deployments. [54]



APP HEALTH
Healthy

SYNC STATUS
Synced to HEAD (07b9a2d)
Auto sync is enabled.
Author: codenow <git@codenow.com> -
Comment: Removing application dev-suvorov...

LAST SYNC
Sync OK to 84fc9c5
Succeeded 14 days ago (Sun Apr 14 2024 17:05:18 GMT+0800)
Author: codenow <git@codenow.com> -
Comment: Application: dev-belka-ticket-rese...

Figure 6.4: ArgoCD dashboard

Chapter 7
Conclusion

The main goal of this work was to iteratively extend an already existing demo cloud-native application by adding an additional functionality, thus demonstrating selected microservice patterns. The most important tasks that led to the addition of the required functionality are summarized in the following points below. A more detailed description can be found in the Subsection New Requirements 2.2.1.

- Addition of a Credit microservice with the use of a Domain Driven Design philosophy for payment processing.
- Integration of the Credit microservice with the Stripe payment gateway to process topping up the user's account balance.
- Addition of a PDF microservice to generate PDF tickets with reservation information.
- Integration of the PDF microservice with MinIO object storage to save the generated tickets.
- Extension of the Notification microservice to allow sending emails with a PDF attachment.
- Refactoring of all the other original microservices to support the newly added functionality E2E.
- Deployment of the whole application to the CodeNOW platform.

Also, additional work was undertaken beyond the initial thesis assignment and requirements.

- Integration of a Keycloak component was introduced to the system. This made it possible to add authentication and authorization to the application. The integration was a follow-up of a previous thesis written by Alena Suvorova. [18]
- At the start of writing this thesis, MinIO was no longer available at CodeNOW as a standard managed component. Therefore, a custom Kubernetes Helm configuration files had to be created, to achieve its deployment and consequent use in this thesis.

Lastly, the following architectural patterns and methodologies typical for microservice applications were applied and demonstrated.

- Domain Driven Design
- Event-Driven Architecture and Messaging
- Event Sourcing and Snapshot
- Data replication
- Distributed tracing
- Log aggregation
- API Gateway
- Database per service

The result of this work is a demo application deployed on the CodeNOW platform that is available to future developers as a reference point. It is also this thesis with an explanation of the necessary theoretical foundations and a description of the E2E process of delivering the required solution.

7.1 Future Work

In its current state, the application does not support the creation of reservations for users, who are not logged in. In real-life scenarios, this feature is essential for enhancing user accessibility and convenience. Therefore, I believe that implementing this functionality should be the next step in the development of the application. Another possible improvement could also include using a more sophisticated approach to the implementation of the API Gateway pattern as discussed in the API Gateway implementation Section 4.4.3.

Bibliography

[1] Richardson, Chris. Microservices Patterns: With examples in Java. Simon and Schuster, 2018.

[2] Stripe Checkout Session API. https://stripe.com/docs/api/checkout/sessions. Accessed 10 January 2024

[3] 12 Factor App. https://12factor.net. Accessed 10 January 2024

[4] CodeNOW configuration. https://docs.codenow.com/admin-manuals/deployment-configurations. Accessed 20 January 2024

[5] What is CodeNOW. https://docs.codenow.com/what-is-codenow. Accessed 28 April 2024

[6] Docker. https://aws.amazon.com/docker/. Accessed 22 January 2024

[7] Spring Boot. https://www.ibm.com/topics/java-spring-boot. Accessed 22 January 2024

[8] Flyway. https://flywaydb.org. Accessed 22 January 2024

[9] Stripe. https://stripe.com/gb/payments/checkout. Accessed 10 January 2024

[10] Kafka. https://kafka.apache.org. Accessed 10 February 2024

[11] PostgreSQL. https://www.postgresql.org/docs/current/intro-whatis.html. Accessed 25 January 2024

[12] REST. https://www.codecademy.com/article/what-is-rest. Accessed 25 January 2024

[13] Vávra, Robin. Cloud Native Application Development. 2022, dspace.cvut.cz/handle/10467/101026. Accessed 26 Jan. 2024.

[14] Decompose by subdomain. https://microservices.io/patterns/decomposition/decompose-by-subdomain.html. Accessed 25 March 2024

[15] Bittner, Kurt, and Ian Spence. Managing Iterative Software Development Projects. Addison-Wesley Professional, 2006.

[16] Wolohan, J. T. Object Storage across the Cloud. Manning, 2020.

[17] Mock Object Models for Test Driven Development. https://ieeexplore.ieee.org/document/1691384. Accessed 10 2024

[18] Suvorová, Alena. Cloud-Native Application Development. 2024, dspace.cvut.cz/handle/10467/113414. Accessed 28 Feb. 2024.

[19] Integration testing. https://glossary.istqb.org/en_US/term/integration-testing-3-2. Accessed 10 April 2024

[20] What is JBehave. https://jbehave.org. Accessed 10 April 2024

[21] Quast, Felix. Testing Microservice Integration with Consumer-Driven Contract Tests. 2022, oss.cs.fau.de/wp-content/uploads/2022/01/quast_2022.pdf. Accessed 1 Apr. 2024.

[22] What is Pact. https://docs.pact.io. Accessed 15 April 2024

[23] Maanonen, Tuomas. Consumer-Driven Contract Testing for Microservices. 2024, aaltodoc.aalto.fi/server/api/core/bitstreams/e035e9e7-b7a8-43c2-8c37-8020ae36dfee/content. Accessed 1 Apr. 2024.

[24] What is E2E testing. https://istqb-glossary.page/e2e-testing/. Accessed 15 April 2024

[25] Garcia, Boni. Hands-on Selenium WebDriver with Java: A Deep Dive into the Development of End-To-End Tests 1st Edition. O'Reilly, 2022.

[26] Selenium. https://www.selenium.dev. Accessed 15 April 2024

[27] Cypress. https://www.cypress.io. Accessed 15 April 2024

[28] JavaMail. https://javaee.github.io/javamail/. Accessed 15 April 2024

[29] Usability testing. https://www.hotjar.com/usability-testing/. Accessed 20 April 2024

[30] Minio Java Client Library. https://min.io/docs/minio/linux/developers/java/API.html. Accessed 22 April 2024

[31] React props. https://react.dev/learn/passing-props-to-a-component. Accessed 25 March 2024

[32] JSX. https://legacy.reactjs.org/docs/introducing-jsx.html. Accessed 25 March 2024

[33] Containerised applications. https://cloud.google.com/discover/what-are-containerized-applications. Accessed 28 March 2024

[34] Microservice architecture and containerisation. https://cloud.google.com/learn/what-is-microservices-architecture, Accessed 28 March 2024

[35] Moises Macero, Learn Microservices with Spring Boot: A Practical Approach to RESTful Services using RabbitMQ, Eureka, Ribbon, Zuul and Cucumber, 2017

[36] Liu, Ling, et al. Encyclopedia of Database Systems. Springer New York, 2019, doi.org/10.1007/978-0-387-39940-9_1366. Accessed 28 Mar. 2024.

[37] BASE and ACID transactions. https://aws.amazon.com/compare/the-difference-between-acid-and-base-database/. Accessed 30 March 2024

[38] What is cloud native architecture. https://www.appdynamics.com/topics/what-is-cloud-native-architecture. Accessed 30 March 2024

[39] Bounded Context. https://martinfowler.com/bliki/BoundedContext.html. Accessed 5 April 2024

[40] DDD definitions. https://martinfowler.com/bliki/EvansClassification.html#: :text=Entity

[41] Domain Event. https://martinfowler.com/eaaDev/DomainEvent.html. Accessed 5 April 2024

[42] Bounded context. https://www.infoq.com/news/2019/06/bounded-context-eric-evans/. Accessed 5 April 2024

[43] Ubiquitous language. https://martinfowler.com/bliki/UbiquitousLanguage.html. Accessed 5 April 2024

[44] Domain Model. https://martinfowler.com/eaaCatalog/domainModel.html. Accessed 6 April 2024

[45] Persistent Volume Claim. https://kubernetes.io/docs/concepts/storage/persistent-volumes/. Accessed 25 April 2024

[46] Basic concepts CodeNOW. https://docs.codenow.com/glossary#deployment-environment. Accessed 25 April 2024

[47] Distributed tracing. https://microservices.io/patterns/observability/distributed-tracing.html. Accessed 27 April 2024

[48] Database Per Service. https://microservices.io/patterns/data/database-per-service.html. Accessed 27 April 2024

[49] Loki. https://grafana.com/oss/loki/. Accessed 27 April 2024

[50] Jaeger. https://www.jaegertracing.io. Accessed 27 April 2024

[51] Log aggregation. https://microservices.io/patterns/observability/application-logging.html. Accessed 28 April 2024

[52] Spring cloud sleuth. https://spring.io/projects/spring-cloud-sleuth. Accessed 28 April 2024

[53] B3. https://github.com/openzipkin/b3-propagation#. Accessed 28 April 2024

[54] ArgoCD, https://argo-cd.readthedocs.io, Accessed 28 April 2024

[55] Step-by-step CodeNOW deployment. https://docs.codenow.com/admin-manuals/deploy-application. Accessed 28 April 2024

[56] Spring Cloud Gateway. https://spring.io/projects/spring-cloud-gateway. Accessed 3 May 2024

[57] MinIO, https://min.io, Accessed 7 May 2024

[58] Dockerfile, https://docs.docker.com/reference/dockerfile/, Acessed 7 May 2024

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Bělka** Jméno: **Přemek** Osobní číslo: **499007**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Softwarové inženýrství a technologie**

II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

Rozšíření mikroservisní a cloud-native aplikace

Název bakalářské práce anglicky:

Cloud-Native and Microservice Application Development

Pokyny pro vypracování:

Rozšiřte o nové funkce stávající ukázkovou cloud-native aplikaci “Rezervace lístků“[1] a implementaci několika vzorů/technologií typických pro architekturu mikroslužeb a cloud-native architektur[2].
1. Integrujte platební bránu Stripe do aplikace. Využijte vzory/technologie: Event-Driven Architecture (Kafka), Event sourcing + snapshot, Domain Driven Design, Containerization (Docker),
1.1 Umožněte zpracovávání plateb při nakupování jízdenek.
1.2 Udržujte stav kreditového konta uživatele a poskytujte asynchronní API pro ostatní mikroslužby. 2. Přidejte podporu pro generování, odesílání a ukládání jízdenek v PDF formátu.
2.1 Integrujte se na stávající notifikační komponentou.
2.2 PDF dokument uložte do MinIO úložiště.
Využijte vzory/technologie: MinIO, Event-Driven Architecture (Kafka), Containerization (Docker). Při vývoji používejte relevantní mikroservisní vzory a postupujte iterativně. Vše průběžné testujte, nasazujte na Value Stream Delivery Platformu CodeNOW[3] a dokumentujte. Získejte zpětnou minimálně od tří nezávislých uživatelů.

Seznam doporučené literatury:

[1] Vávra Robin. Vývoj cloud native aplikací v praxi. B.S. thesis, České vysoké učení technické v Praze. Výpočetní a informační centrum., 2022. Dostupné z: http://hdl.handle.net/10467/101026.
[2] Chris Richardson. Microservices patterns: with examples in Java. Manning Publications, 2018. Dostupné z: https://learning.oreilly.com/library/view/microservices-patterns/9781617294549/.
[3] CodeNOW. CodeNOW Documentation, 2022. Dostupné z: https://docs.codenow.com/.

Jméno a pracoviště vedoucí(ho) bakalářské práce:

Ing. Martin Komárek kabinet výuky informatiky FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) bakalářské práce:

Datum zadání bakalářské práce: **25.01.2024** Termín odevzdání bakalářské práce: **24.05.2024**

Platnost zadání bakalářské práce: **21.09.2025**

Ing. Martin Komárek podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Mgr. Petr Páta, Ph.D. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Student bere na vědomí, že je povinen vypracovat bakalářskou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací.
Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v bakalářské práci.

.	
Datum převzetí zadání	Podpis studenta