SOA - Project Milestone 2
TrainCatcher - Train Booking Service

Victor Alecu (s2675889)
Vlad Gosa (s2755572)

1 Project Description

TrainCatcher is a Service-Oriented Architecture (SOA) system designed to streamline train trip planning and booking. It aggregates real-time schedule and availability data from multiple operators, enabling dynamic itinerary construction. Moreover, it allows users to book and pay for multi-operator journeys in one go. By consolidating various data streams into a single platform, TrainCatcher communicates with each operator's service to confirm seat availability and secure bookings, resulting in a unified digital ticket for the entire trip. This integrated approach simplifies the reservation process and enhances the overall travel experience. Additionally, through a dedicated notification service the application is able to provide passengers not only with tickets for their booked journeys, but also updates regarding delays and upcoming trips/transfers.

To better illustrate the benefits of TrainCatcher, let's consider a trip from Enschede to Amsterdam. Traditionally, passengers must consult the websites of three separate operators—traveling from Enschede to Zwolle with Blauwnet, from Zwolle to Utrecht with Arriva, and from Utrecht to Amsterdam with NS—resulting in multiple bookings and monitoring efforts. TrainCatcher streamlines this process by allowing users to simply enter their departure and arrival stations along with their preferred departure time. The system then queries all available carriers, checks seat availability, and constructs optimal itineraries. Once a trip is selected, it handles the consolidated payment and aggregates ticket data into a single code, while continuously monitoring the journey and providing timely notifications for any delays or transfer reminders.

2 Services Involved

2.1 Train Operator Service

Description

One goal of our application is to allow train operators to dynamically link to our system. This implies that, in order to mimic such functionality, we need to define the interface between our system and an external train company system. The train operator service therefore represents a microservice which stores all the data on behalf of a train company, essentially acting as the company's API endpoint. For the purpose of this project, the data stored by each company has been intentionally simplified to focus on core components: trains and the journeys they operate. Each train record includes basic details such as the model name and number of seats. A journey is defined solely by a departure and an arrival station—similar to a direct plane flight—along with the scheduled times, the specific train executing the journey (including the count of occupied seats), and the ticket price. This deliberate simplification, which omits intermediate stops, enables the search service to easily aggregate journeys from multiple operators to construct more complex routes (i.e., journeys are only direct train trips while routes may have stops). Technically, our system does support multi-stop journeys; however, they are represented as distinct journey entities. For example, a journey from Enschede to Amsterdam on train X can be decomposed into three separate journeys: Enschede to Hengelo, Hengelo to Zwolle, and Zwolle to Amsterdam. In a real-life scenario, journeys would be more complex and require advanced routing algorithms with various heuristics, but for our project, this streamlined model meets our service-oriented architecture goals. Additionally, each train operator can be dynamically added to the train search service, with dedicated endpoints for booking and payment. This design allows the

booking service to query available journeys, reserve a seat on each segment of a desired route, and subsequently process payments for each operator accordingly.

Communication with other services

All interactions with the train operator service are performed synchronously via RESTful calls. This synchronous model is favored because booking and payment operations require immediate, verifiable responses to ensure transactional consistency and prompt error handling. RESTful APIs offer a well-defined, standardized interface that simplifies integration, making it easier to manage direct request-response cycles. Additionally, since train operators also need to provide journey data to the search service, RESTful communication ensures that the search service receives up-to-date and directly available information without the delays or complexities associated with ensuring consistency in an asynchronous setting, like message queues or websockets. Given that the booking process involves sequential steps—where the booking service must confirm seat reservations and process payments in real time—the immediacy and simplicity of synchronous RESTful communication are better suited to our operational requirements.

Communication with other services

All interactions with the train operator service are performed synchronously via RESTful calls. This synchronous model is preferred because booking and payment operations require immediate, verifiable responses to ensure transactional consistency and prompt error handling. Moreover, the search service queries each train operator service independently through RESTful endpoints to retrieve current journey data, which it then aggregates locally to construct complete routes. RESTful communication ensures that the search service receives up-to-date and directly available information without the delays or complexities associated with asynchronous message queues or websockets. This approach simplifies integration, testing, and debugging, while also avoiding the additional overhead of managing eventual consistency, making it a more efficient and reliable option for both transactional operations and route aggregation.

Implementation Progress

The main functionality for managing trains and journeys (CRUD operations) has been implemented, along with the integration of operator services with the search service to facilitate journey aggregation. Additionally, the compensation activities for processing payments have been developed; which handle the transfer of funds to each operator for the journeys provided in a trip. The remaining task is to implement the compensating activities for journey booking, marking each journey as pending one booked seat which is subsequently marked as occupied when the associated payment is processed.

2.2 Train Search Service

Description

The train search service serves as the gateway into the system and initiates the main journey booking saga. It allows train operators to register themselves via their URL, thereby validating them as approved sources of train journey data. On the user side, passengers can search for journeys by specifying a departure station, an arrival station, a departure date and time, and a maximum number of allowed train changes. The service queries all registered operators to retrieve available journeys and aggregates these into potential routes that match the user's criteria. When a user selects a route, the complete route details—comprising all individual journeys—are forwarded to the booking service to continue the saga. Additionally, as the primary service serving the frontend, the train search service provides RESTful proxies for login, sign-in, and logout operations by forwarding authentication requests to the user service.

Communication with other services

The train search service relies on synchronous RESTful communication for its core endpoints, including operator registration, journey search, and the authentication proxies (login, sign-in, and logout).

RESTful APIs offer a direct, standardized interface that returns immediate responses, which is essential for real-time operations. This approach ensures that operator registrations are confirmed instantly and that search queries yield up-to-date journey data without the complexity and latency associated with asynchronous systems. Additionally, using synchronous calls for authentication processes guarantees that users receive prompt validation, and it seems to be common practice for such use-cases. Overall, the simplicity and low-latency nature of RESTful communication make it the most suitable choice for these critical, time-sensitive features.

Implementation Progress

Functional requirements-wise, the service has been implemented completely. Both the train operator registration and journey search endpoints are fully operational, and a basic algorithm for aggregating journeys into more complex routes is in place. While a more advanced route aggregation algorithm was considered, it was deemed out of scope for this SOA project. The remaining task is to configure an Nginx API gateway that will expose only the login proxy, search, and operator registration endpoints, while keeping all other service endpoints hidden.

2.3 Booking Service

Description

Once a user selects a journey via the train search service, the train booking service takes over to secure the booking. It notifies each relevant train operator about the specific journey component that requires a seat to be reserved. The operators then mark the seat as booked until payment for the journey is processed; afterward, the seat status is updated to occupied. Once all operators confirm that the journey components can be reserved, the booking service forwards a ticketing request to the ticket service through a dedicated API endpoint.

Communication with other services

All interactions within the booking process are mainly conducted synchronously. The service receives booking requests from the search service, then queries each train operator service to reserve seats, and finally issues a ticketing request transmitted by the orchestrator. Synchronous communication is preferable in this context because it guarantees a controlled, step-by-step execution, crucial for coordinating multi-component bookings. This deterministic sequence simplifies error detection and handling during the booking process, ensuring that any failure in reserving a seat can be immediately addressed with compensation actions. Such tight coordination would be more challenging with asynchronous messaging, which could introduce delays and require additional mechanisms to maintain transactional consistency.

Implementation Progress

All relevant API endpoints have been implemented, along with integration into the main saga via Camunda. An H2 in-memory database has been set up to store booking data. Remaining tasks include finalizing the communication with the train operator services—primarily handled on the operator side—and implementing user session verification using JWT where necessary.

2.4 Ticketing Service

Description

Once a route's journeys have been confirmed as bookable, the ticketing service aggregates all relevant journey information and passenger details into a single ticket. The service then notifies the payment service that the ticket is ready to be issued. Upon successful payment—where funds are transferred for all relevant journey legs—the ticket is sent to the user. To deliver the ticket, the ticketing service publishes it onto a message queue for the notification service, which subsequently sends it via email to the address specified on the ticket.

Communication with other services

The ticketing service employs a mix of synchronous and asynchronous communication. Synchronous RESTful API calls are used when interacting with the booking and payment services (via the orchestrator), ensuring immediate confirmation and consistent transactional state during the critical steps of ticket issuance and payment processing. This direct, real-time interaction is essential for maintaining control over the sequential operations. In contrast, asynchronous communication is utilized with the notification service through a message queue, allowing the email dispatch process to be handled independently. This separation ensures that the delivery of the ticket does not impede the overall workflow, while still providing reliable and scalable notification processing.

Implementation Progress

All necessary functionality for the ticketing service has been implemented. The service is fully capable of aggregating ticket data, communicating with both booking and payment services, and publishing tickets to the notification service's message queue. Future improvements may include additional JWT verification for user sessions to further enhance security and provide finer-grained access control.

2.5 Payment Service

Description

After confirming that all journeys can be booked and the ticket can be printed, the main saga prompts the payment service to transfer funds from the user's account balance to the corresponding train operators - as depicted in the mini-saga of Figure 1. For each operator, the service dispatches payment instructions corresponding to the ticket price of the journeys they provide. Once payments for all journey segments are successfully processed, the payment service signals the ticket service to publish the already created ticket onto the message queue for the notification service, which then delivers it to the user.

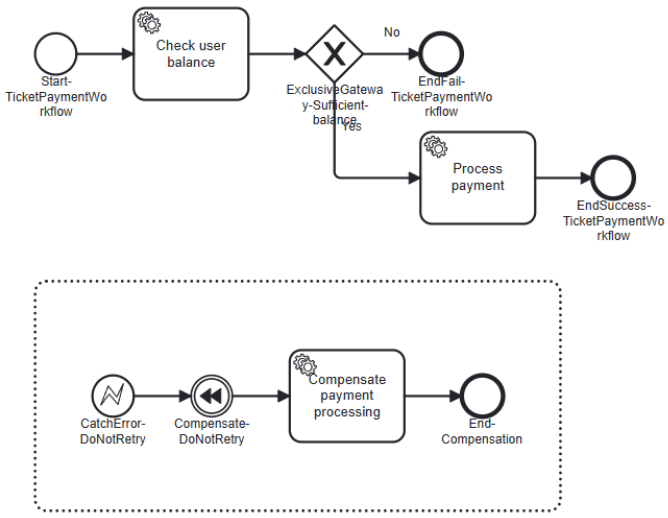

Figure 1: Camunda workflow of payment process

Communication with other services

The payment service uses synchronous RESTful communication for all interactions. Each transaction is initiated directly by the orchestrator, with the payment service performing immediate, blocking calls to the train operator APIs to execute the payments. This approach ensures that every payment is fully validated and confirmed in real time, which is crucial for financial operations where accuracy and consistency are paramount. Synchronous communication eliminates the potential delays and

complexity of managing eventual consistency that come with asynchronous methods. It also reduces the risk of duplicate transactions or lost messages, as each call provides a clear and immediate response that can be used to enforce atomicity and trigger compensation actions if necessary.

Implementation Progress

The entire functionality of the payment service has been implemented, covering its role in the main ticket booking saga as well as the internal mini-saga for issuing payments to the various train operators. All compensation activities for handling payment failures are in place. As a result, the payment service is fully operational and integrated within the broader orchestration, ensuring that all financial transactions are reliably managed.

2.6 User Service

Description

The user service stores all data for registered users and functions as the main authentication and access control broker. It manages user registration, profile management, and authentication via email and password. Upon successful login, the service generates a JWT token that authorizes users for subsequent operations, particularly within the ticket booking saga, ensuring secure interactions throughout the system.

Communication with other services

All interactions with the user service are conducted synchronously via RESTful calls. This synchronous approach is favored because authentication requires immediate validation and token issuance, which are essential for secure access control. Real-time responses ensure that only authenticated users can proceed with actions like booking and account management. In contrast, asynchronous communication would introduce delays and complicate error handling, potentially compromising security and user experience. RESTful APIs offer a standardized, easy-to-integrate interface that meets the low-latency and high-reliability demands of authentication processes.

Implementation Progress

The core functionality of the user service has been implemented, including full CRUD operations for user data, authentication processes, and JWT token generation. This service is already integrated into the main ticket booking saga to ensure that only authenticated users can search for and book trips. Future work involves further integrating JWT-based authentication across additional system components, such as account balance top-ups and payment processing, to maintain secure and consistent access control throughout the platform. One step towards that is that our security features are defined in a local library, shared among all the microservices. Using aspect-based programming, we defined annotations to automatically verify whether users are authenticated, which can easily be added to any REST method. Additionally, features regarding profile management (e.g., update credentials or user info) could be implemented; however, they are of little consequence to the main flows of the system.

2.7 Notification Service

Description

The notification service is designed to transmit relevant data to passengers via email, including tickets, reminders of upcoming trips, and updates about delays or journey modifications.

Communication with other services

The service utilizes an asynchronous publisher-subscriber model by subscribing to multiple message queues based on the type of notification. When a message is received, the service processes it and dispatches an email to the intended passenger. The choice of asynchronous communication over synchronous methods is motivated by the need to decouple processing tasks, enabling parallel execution and incorporating retry mechanisms without stalling the entire workflow. This architecture facilitates

load balancing and fault tolerance by isolating issues within individual message handlers. In addition, using message queues supports horizontal scaling, as additional workers can be deployed to manage increased loads. This approach results in improved throughput, enhanced error isolation, and a more resilient system under high-demand conditions. Conversely, a synchronous model would not really be feasible for this use case, as each notification request would block the process until a response is received, which increases overall latency and risks system-wide delays if a downstream component—such as the mail server—experiences high response times or failures.

Implementation Progress

To date, the service has been implemented as a standalone component capable of listening to two message queues—one for ticket information and another for general journey notifications—using ActiveMQ. A fake mail server (MailHog) is employed for email testing purposes. Additionally, a failsafe mechanism has been integrated to monitor the mail server connection, automatically disabling queue listeners if the connection is lost to avoid processing messages that cannot be sent. Future work includes integrating a saga for journey notifications (complementing the existing ticket saga) and configuring a Dead Letter Queue (DLQ) for handling emails that fail to be delivered.

3 Orchestration

Business Flow Orchestration

Our application leverages Camunda to orchestrate one main business flow, along with a complementary flow (payment process, see Figure 1). In the ticket booking flow (depicted in Figure 2), a logged-in user searches for a route and selects one to book. The booking service then sends booking requests to each relevant train operator, providing details on the journey components that require seat reservations. Upon confirmation that all segments are available, the operators mark the journeys as pending an additional booking, and the booking service issues a ticket creation request to the ticketing service. The ticketing service aggregates all journey and passenger information into a single ticket and then calls on the payment service—our pivot process—to process the payments for each journey leg as depicted in Figure 1. Once all payments are confirmed, the payment service notifies the ticketing service, which subsequently publishes the ticket on a message queue for the notification service to deliver via email.

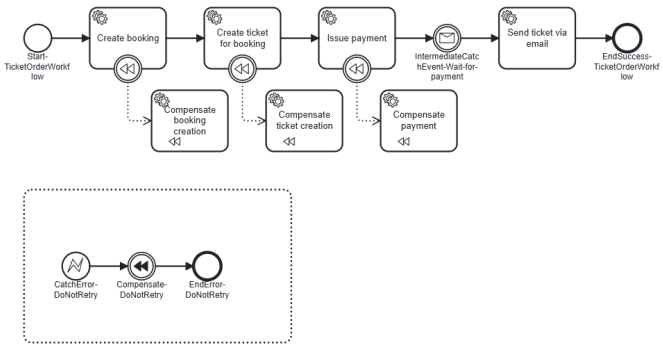

Figure 2: Camunda workflow of main process

Rationale for Using Camunda

We selected Camunda for orchestration instead of embedding the process entirely in Java due to its robust workflow management and monitoring capabilities. Camunda's BPMN engine provides a clear, visual representation of the sequential and conditional steps involved in the ticket booking saga, facilitating easier adjustments and error handling. While a pure Java implementation might offer tighter control over each component, it would require extensive custom development to manage compensating

transactions, retries, and state monitoring. Camunda simplifies these aspects through built-in features, reducing development overhead. For the scope of this project—where the focus lies in demonstrating a service-oriented architecture—Camunda offers an optimal balance between flexibility and operational transparency.

Instead of defining BPMN models using Camunda Modeler, we have built a Java-based model builder helper class which allows us to programmatically define BPMN workflows and deploy them to the embedded Camunda engine. The main advantage of this approach is that we can now more easily debug workflows both visually through Camunda Cockpit, as well as programmatically through actually debugging live code at runtime. Moreover, each deployed workflow is versioned in our git repository, making this approach highly flexible compared to modifying XML models.

Pivot Process Design

In designing the orchestration for ticket issuance, we evaluated two primary approaches. In Option 1, the flow follows: booking, ticketing, then payment (serving as the pivot), and finally sending the ticket. In this sequence, the ticket is prepared after booking, but before payment is processed. Here, payment acts as the final go/no-go point, ensuring that funds are only withdrawn if every preceding operation is successful. This approach follows the natural business flow, ensuring that the complete ticket information is confirmed before any financial transactions occur, which minimizes the risk of having to refund payments if later steps fail. Conversely, Option 2 follows: booking, payment, then ticketing (as the pivot) before sending the ticket. This alternative, which aligns more with the process/logical flow, offers the advantage of reducing wait times by processing payment earlier, allowing the ticket to be created and dispatched immediately upon confirmation. However, processing payment too early may lead to situations where, if subsequent operations fail, funds must be returned—adding complexity and potential user dissatisfaction. Ultimately, we chose Option 1 because it best meets our hypothetical business requirements by ensuring transactional consistency and a more robust, user-friendly experience.