Assignment #7 - TicketBoss

Software Systems Architecture

Alexandre Ferreira Nunes
André Correia da Costa
André Filipe Garcez Moreira de Sousa
Daniel José Mendes Rodrigues
Gonçalo da Costa Sequeira Pinto

U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

April 2024

Contents

1 Introduction

This report presents the architectural design of TicketBoss, a ticketing system tailored to handle ticket sales for various entertainment events. In response to the ever-growing demand for seamless and efficient ticketing solutions, TicketBoss has been meticulously crafted to provide users with a streamlined experience from event and seat selection to purchase confirmation.

Throughout this report, we will delve into TicketBoss's architecture, examining its logical, process, and use-case views. We will elucidate the underlying components, interactions, and use cases, shedding light on the system's design principles and key architectural decisions.

2 TicketBoss's Architecture

In the development of TicketBoss, attention has been devoted to crafting an architecture with a main focus on the demands of scalability, availability, and reliability, but also ensuring optimal performance and maintainability. Our architectural framework is rooted in the principles of microservices, supplemented by the employment of load balancing and database partitioning.

As shown in **Figure 1**, we consider five different services, **User Interface (UI)** only interacts with **Event Manager** and **Seat Manager**. Finally, **Seat Manager** handles two different external entities: **Payment Service** and **Email Service**.

This modular approach offers several advantages. Firstly, it facilitates the development process by breaking down the system into smaller, more manageable components, enabling parallel development, testing, and easier debugging. Additionally, microservices promote agility, allowing for swift updates and modifications to individual modules without disrupting the entire system. Moreover, they enhance fault isolation, as issues within one module are less likely to cascade throughout the system, thus reinforcing overall system resilience.

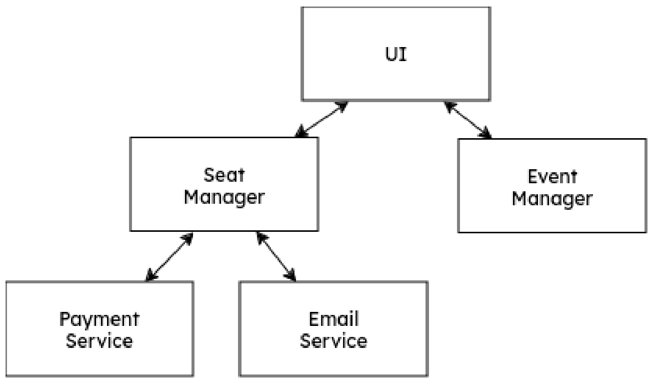

Figure 1: TicketBoss Microservices

TicketBoss leverages load-balancing mechanisms, as depicted in **Figure 2**. By distributing incoming traffic across multiple instances of the application, load balancers ensure optimal utilization of resources and mitigate the risk of overwhelming any single server infrastructure. This not only improves response times and throughput but also enhances fault tolerance by redirecting traffic away from unhealthy or overloaded instances.

In addition to using the load balancer, our architecture foresees the use of a "Leaky Bucke t" strategy for peak times, such as when Taylor Swift announces ticket sales and there is a large number of users trying to access the site. In this approach, all requests are collected and stored in a "bucket" that functions as a queue. Requests are removed from this queue at a rate compatible with the servers' capacities, ensuring that although users may have to wait a bit, our server infrastructure never becomes overloaded to the point of failure.

Additionally, our architectural design incorporates database sharding to augment efficiency and manageability. There would be a read-only general database with Event information. Subsequently, a broker would be introduced to distribute the event seats so that every database would have roughly the same number of accesses. Note that the Event database is read-only from the user's point of view, it can be modified by the TicketBoss's Administrators. By segmenting the database into distinct partitions, each dedicated to a specific subset of data, we avoid service outages and improve access times. This segmentation enhances performance and facilitates scalability by enabling independent scaling of each partition to accommodate varying workloads.

Finally, we have implemented a database backup system to ensure data integrity and prevent loss. For each database, we've set up regular automated backups, ensuring that crucial information is securely stored. These backups are scheduled to run at intervals optimized for our operational needs, daily during non-peak hours, with redundancy built in for added security.

In the event of unforeseen incidents or failures, these backups provide a safety net, allowing us to swiftly restore data and minimize any potential disruptions. This proactive approach to data backup significantly enhances the resilience and dependability of our system, safeguarding against data loss or corruption.

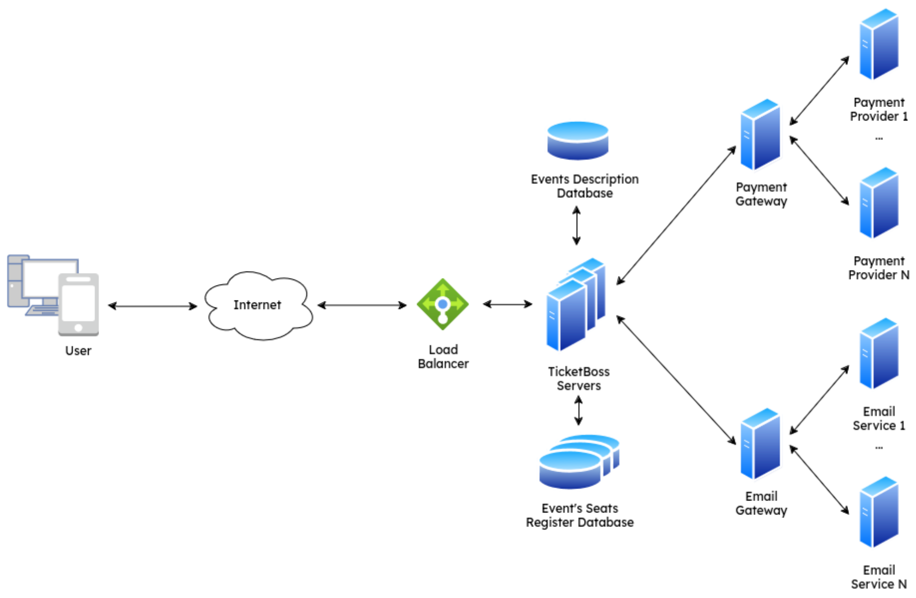

Figure 2: Physical View

2.1 Logical view

As shown in **Figure 3**, the logical view of the architecture of TicketBoss, a software system designed to manage ticket sales for various entertainment events, adheres to the separation of concerns principle, where each component has a well-defined role and interacts with others to achieve the system's overall objectives. In the upcoming subsection, we will further describe the different components and their interactions.

2.1.1 Components and Interactions

This software system will be built using multiple components that are grouped into four main packages:

1. **User Interface:** The user interface serves as the entry point for user interaction. It acts as a separate layer that orchestrates user interactions and presentations without being entangled with the system's internal workings. This package contains five different components:

    (a) **Catalog Display:** This component is responsible for receiving the information about all events in the platform from the backend and displaying it to the user.

(b) **Seat Selection:** allows users to interactively choose the seats they want to reserve.

(c) **Checkout:** gathers user information such as billing address and credit card details necessary for payment processing. It securely transmits this information to the Payment Service for authorization and transaction processing. Upon successful payment, Checkout interacts with the Seat Manager to finalize the purchase by marking the seats as sold. It also triggers the Email Service to send a confirmation email to the user.

(d) **Email Confirmation:** This component is responsible for informing the user that he/she received the tickets in the email address provided and that the purchase process was complete.

(e) **Admin Page:** This component allows the administrators of TicketBoss to apply CRUD operations to the event catalog, using a Graphical User Interface (GUI).

2. **Functional Components:** This package is the middleman between the GUI and the Data Storage package. It includes all the business logic of the platform and it is responsible for making sure that the business rules are not violated. With this goal, it is built with different components:

   (a) **Event Catalog:** This component acts as the system's doorman of the repository for event data. It serves as the Model (of the **Model-View-Controller** (MVC) pattern), encapsulating event data, and exposing an API for the "Catalog Display" component to access this data.

   (b) **Seat Manager:** This component is responsible for interacting with the Data Storage package to remain up-to-date with all the seat reservations in all events of the catalog. This component allows the scheduling of seat reservations and enforces the 10-minute rule, starting a timeout event that deletes the reservation when the time expires. When the user performs the checkout, then this component is notified to cancel the timeout, confirming the reservation.

   (c) **Seat Guard:** This is the component that prevents conflicts in reserving seats: when a new request arrives, it uses Seat Manager to check if the reservation is valid and it informs the Seat Selection component accordingly. It is worth noting that a reservation is only valid if the seats are available and the total number of seats allocated for that user is not greater than ten.

   (d) **Seat Selection:** This component plays a pivotal role in managing user interactions related to seat selection. It makes use of the Seat Guard component to check the availability of the selected seats for a chosen event from the Event Catalog. If the seats are available, then it interacts with the Seat Manager component to make the reservation and start the aforementioned timeout event to make sure the 10-minute rule is enforced. If they are not available or if the user has reached its maximum number of allowed seats (ten), then the system informs the user accordingly.

   (e) **Event Manager:** This component is used by the administrators of the platform to add, remove, and edit the events present in the Event Catalog.

3. **Data Storage:** This package encompasses the functionality for accessing and interacting with the database so that all events scheduling and seat reservations are persisted in stable storage. With this goal in mind, we created a component that is responsible for this interaction: **Database Management**.

4. **External Systems:** this package includes all the services that are not native to the application but are required for it to function properly:

   (a) **Payment Service:** This service securely processes credit card transactions during checkout. It interacts with the user's financial institution to verify payment information and authorize the transaction.

   (b) **Email Service:** This service sends email confirmations to users after a successful checkout.

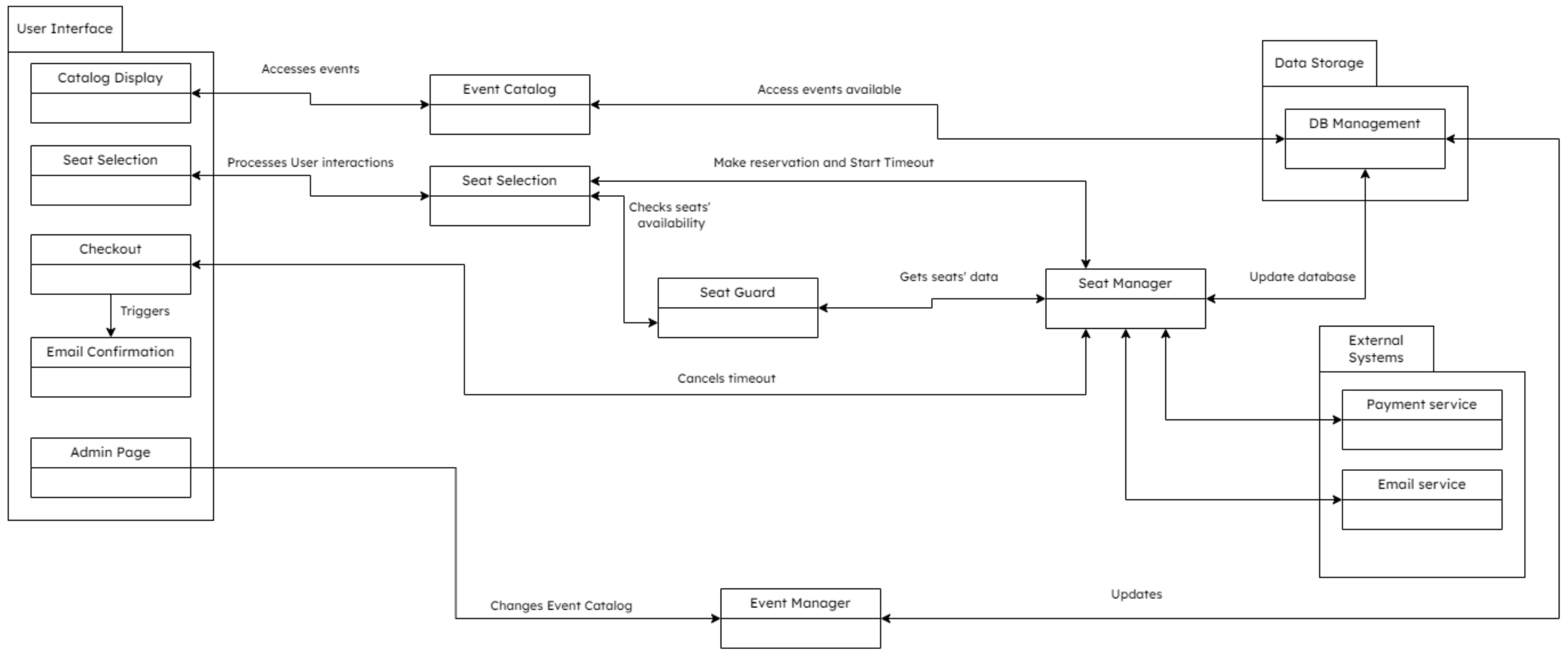

Figure 3: Logical view

2.2 Process view

The process view depicted in **Figures 4 and 5** shows the interactions between different parts of the TicketBoss system. The process involved interactions between **7** distinct participants: **User**, **UI**, **EventManager**, **SeatManager**, **Database**, **External Payment System** and **External Email Service**.

The user communicates directly with the User Interface, that redirects the request to one of the backend services. The EventManager handles event searches by communicating with the Database to get the most recent information. The SeatManager handles seat search, selection, and buying. It interacts with the Database to retrieve initial seat data. As users select seats, it caches this information for efficiency, minimizing the amount of accesses to the database, and sets a 10-minute timeout. Upon successful checkout (confirmed by the external payment entity), the Database is updated with the latest information and a confirmation email is sent to the user. If the timeout expires, seats are released and removed from the cache.

It is worth noting that in both searches, managers only access the database once every 10 minutes to update the events/seat information. Every request in between is made to their cache, allowing a more responsive and efficient system.

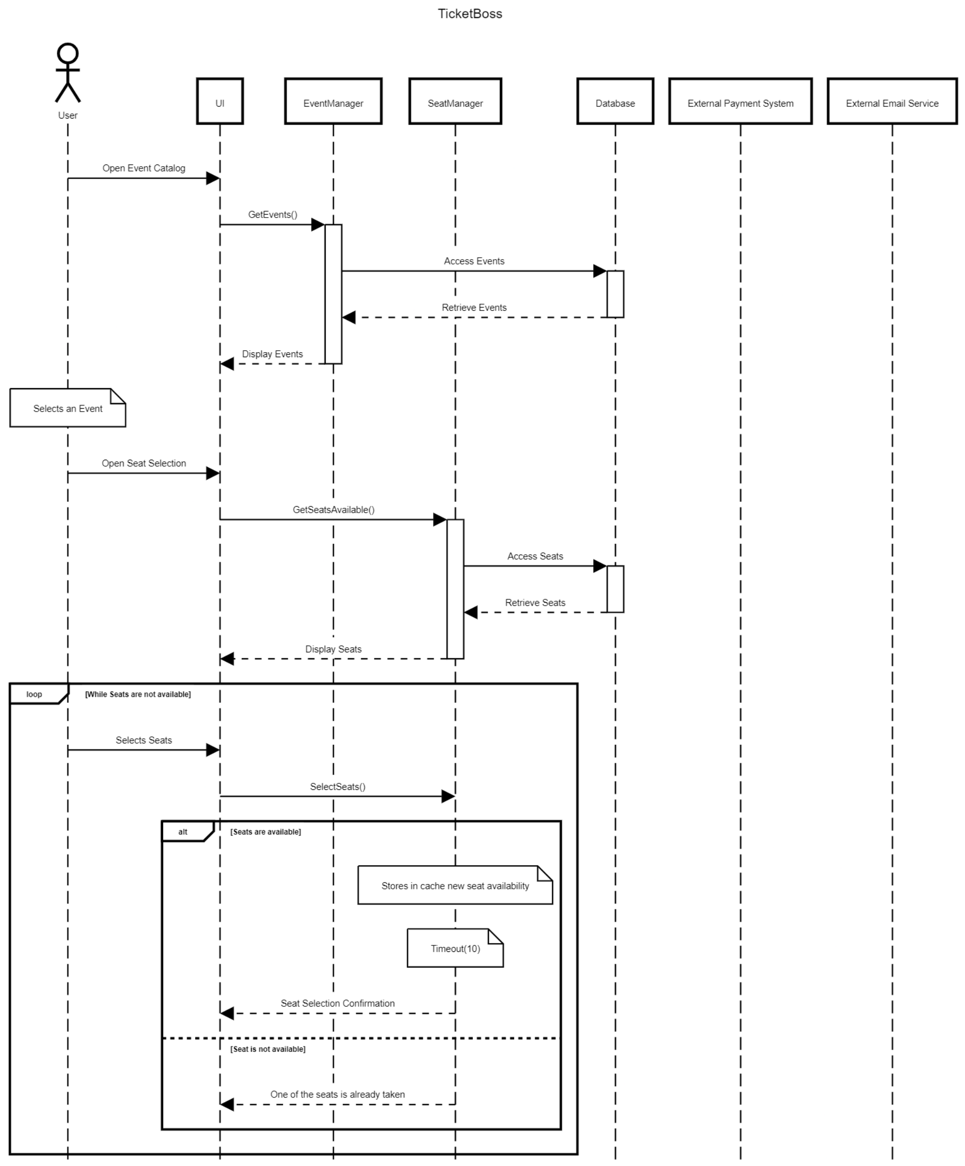

Figure 4: Process View (Part 1)

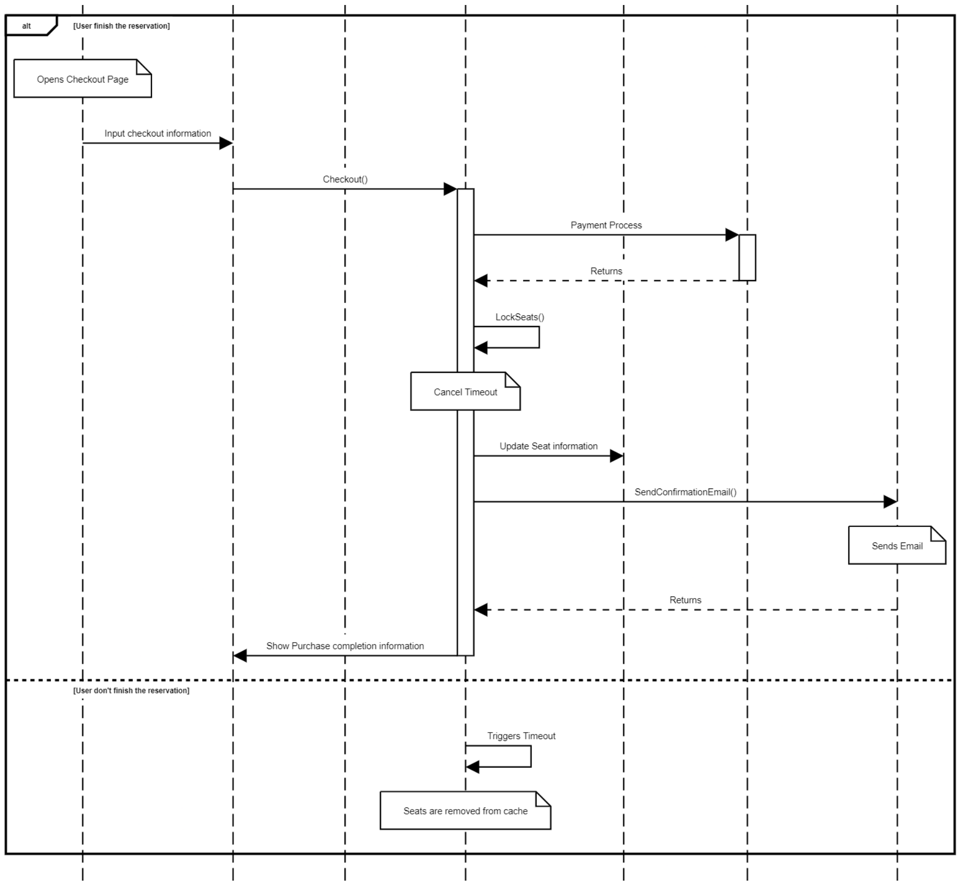

Figure 5: Process View (Part 2)

2.3 Use Cases view

In the Use Cases view, the various scenarios depicting user interactions with the TicketBoss system are outlined.

2.3.1 User selects seats and buys them

- **Actors:** User, External Payment Service
- **Preconditions:** There are events and seats available
- **Postconditions:** User receives an email with his ticket or the process is aborted

- **Description:** As seen in **Figure 6** in order to select and purchase seats, several sub-use cases must be completed, such as viewing events, browsing available seats, reserving desired seats, and completing the payment process. As previously mentioned, the checkout will be handled by the external Payment Service.
- **Exception Path:** If the user takes more than 10 minutes to complete the checkout after selecting seats, the process is canceled.
- **Priority:** High, since it is one of the most common paths that the users will perform.

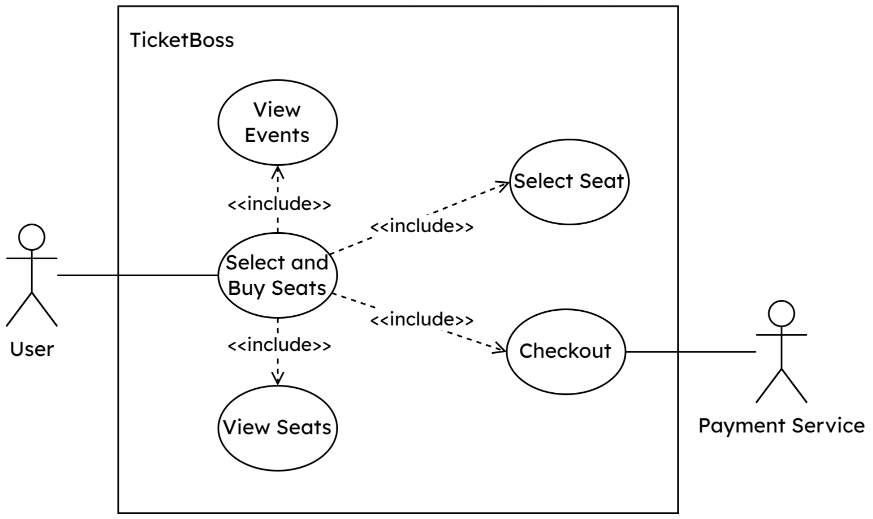

Figure 6: Use Case 1

2.3.2 Two or more people select seats for the same event at the same time (within ten minutes of each other.) There is no conflict.

- **Actors:** Two different Users
- **Preconditions:** There are events and seats available
- **Postconditions:** Both users receive tickets for different seats or the process is aborted
- **Description:** As seen in **Figure 7**, two distinct users concurrently select seats for the same event, each within a 10-minute window of the other. There are represented two optional paths in the diagram: "Reserve Seats

[1-3]" and "Reserve Seats [4-5]", however, each user will be linked to a different path, indicating their selection of distinct seats. To fulfil the use case requirements, both users will choose the same event, denoted as "Choose Event X", which encompasses the viewing of available events. Finally, the reservation process depends on users browsing the available seats.

- **Exception Path:** If any user takes more than 10 minutes to complete the checkout after selecting seats, or if they choose any common seat, the process is cancelled.
- **Priority:** High, it is important to ensure that any seat is sold to only one person.

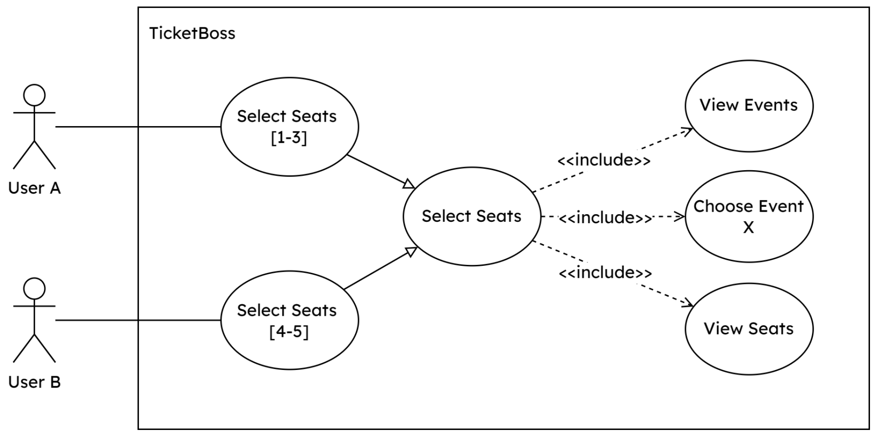

Figure 7: Use Case 4

3 Advantages of Our Architecture

In this section, we explain some choices about the architecture design process, that had impact in some quality attributes and that the team considered essential for the successful implementation of the system. Among them, the following stand out:

- **Separation of Concerns:** The architecture adheres to the principle of separation of concerns, where each component has a well-defined role and responsibility. This makes the system easier to understand, maintain, and modify. Changes to one component are less likely to impact others.

- **Maintainability:** By isolating functionalities within distinct components, the architecture promotes maintainability. Developers can focus on modifying specific components without needing to delve into the intricacies of the entire system. This reduces the risk of introducing bugs during maintenance activities.

- **Scalability**: the architecture exhibits scalability. As the number of users, events, and transactions increases, components like the Data Storage can be scaled independently to handle the growing load. This ensures the system can accommodate future growth without significant architectural changes.

- **Data Integrity:** Utilizing a centralized Data Storage managed via a Data Access Object (DAO) layer aids in preserving data integrity. This layer guarantees uniform data access and manipulation throughout all components, thereby minimizing the potential for data discrepancies.

- **Security:** The design improves security by using outside services, such as a secure Payment Service for financial transactions. This reduces security risks by not requiring TicketBoss to handle sensitive payment data directly.

- **Testability:** The separation of concerns often leads to better testability. Individual components can be unit-tested in isolation, simplifying the testing process and improving overall application quality.

- **Interoperability:** The microservices architecture requires special attention to data sharing among its various components, necessitating the creation of multiple APIs. Certain functionalities of these APIs, such as the dissemination of upcoming events, occupancy information, etc., can be made public, enabling third-party software to use this data.

- **Flexibility:** As the platform grows, the demand for new features inevitably arises to stay ahead in the market. Adopting a microservices-based architecture streamlines the integration of these new features, necessitating only the creation of new components without modifying existing code, thus ensuring a seamless and efficient process.

In section 5, we explain in more detail each attribute and their importance to our system.

4 Architectural Patterns

In the architecture designed for TicketBoss, several software architectural patterns are employed to enhance its robustness, scalability, and maintainability.

4.1 Publish-Subscribe Pattern

The Publish-Subscribe pattern is implemented in the Event Catalog component. When an administrator makes changes to an event in the catalog, all users who have booked seats for that event receive notifications about the changes. This decoupled communication ensures that users stay updated on relevant event modifications without the need for direct polling.

4.2 Object-Oriented Programming (OOP)

Object-Oriented Programming principles are fundamental to the architecture of TicketBoss. The system is structured around objects representing real-world entities such as events, seats, and users. Encapsulation, inheritance, and polymorphism are leveraged to model behaviors and relationships, promoting modularity, reusability, and maintainability.

4.3 Client-Server Architecture

TicketBoss follows a Client-Server architecture where the client-side UI interacts with server-side components like the Event Manager, Seat Manager, and External Services. This separation facilitates scalability, as multiple clients can connect to the centralized server infrastructure, and promotes a clear division of responsibilities between the presentation layer and the backend services.

4.4 Event-Based Architecture

While TicketBoss's server actions are synchronous, an event-based architecture is still utilized for communication and workflow orchestration. Events such as seat reservations, purchases, and updates trigger corresponding actions across different components. This pattern enhances responsiveness and modularity by allowing components to react to changes asynchronously.

4.5 Broker Pattern

The Broker pattern is employed to route requests to different server instances in TicketBoss. A centralized broker component receives incoming requests from clients and forwards them to appropriate server instances based on load balancing or other routing strategies. This pattern enhances scalability, fault tolerance, and flexibility in resource allocation.

4.6 Shared Repository Pattern

TicketBoss employs the Shared Repository pattern to provide a centralized data storage mechanism for events, seats, and user information. The Data Storage package serves as a shared repository accessed by various components, ensuring data consistency and integrity. This pattern enhances data access efficiency, modifiability, and maintainability.

4.7 Model-View-Controller (MVC) Pattern

TicketBoss's architecture embodies the MVC pattern, particularly in the separation of concerns between the presentation layer, application logic, and data access layers. The User Interface acts as the View, interacting with Controller components such as Seat Selection and Checkout, which manipulate the Model represented by the Event Catalog and Seat Manager. This separation enhances modularity, testability, and maintainability.

4.8 Presentation Layer Pattern

TicketBoss adheres to the Presentation Layer pattern, where User Interface components focus solely on presenting information to users and capturing user interactions. The Presentation Layer orchestrates user interactions and delegates business logic to other layers, promoting separation of concerns, modifiability, and reusability of UI components.

4.9 Microservices Architecture

TicketBoss embraces a Microservices architecture, with different functionalities encapsulated within independent services such as Event Manager, Seat Manager, Payment Service, and Email Service. This modular approach enables scalability, fault isolation, and independent deployment of services, enhancing flexibility, and facilitating continuous integration and delivery practices.

4.10 Service-Oriented Architecture (SOA)

TicketBoss exhibits characteristics of a Service-Oriented Architecture (SOA) by encapsulating distinct business functionalities into services. Each service, such as Event Manager and Payment Service, provides well-defined interfaces and encapsulates specific business logic. SOA promotes loose coupling between services, enabling easier integration, maintenance, and scalability while fostering interoperability and reusability.

5 Quality Attributes

This section presents the Non-Functional Attributes of this system, ordered by the Importance parameter and accompanied by its meaning to the system and an Acceptance Level.

5.1 Scalability

The ability to adjust the system's capacity to accommodate fluctuations in workloads.

5.1.1 Acceptance Level

Ability to add more events and components to handle the new volume of requests, such as hardware resources, without affecting other quality attributes and without stopping the entire system.

5.1.2 Importance

Very High, since the scalability of the system would affect multiple other quality attributes. Adding more components can be critical to various attributes, so it's crucial that the system eases this scalability.

5.2 Availability

The system's capacity to be operational and accessible when required for use.

5.2.1 Acceptance Level

Due to the system modularity, even in the presence of faults or Server latency, the frontend can/may be 100% available in the presence of a network connection.

5.2.2 Importance

Very High. This attribute can be very tricky to achieve due to the high dependency on how reliable our system may be. The availability has a direct impact on our clients' opinion of the system.

5.3 Reliability

The system's capacity to perform specific operations under some conditions for a period of time.

5.3.1 Acceptance Level

The large amount of requests can bring faults to the system. With the Leaky Bucket implementation, it can handle this high demand. Additionally, it should have fault-tolerance mechanisms to keep the database updated and the Server side alive.

5.3.2 Importance

High. If the system is not fault-tolerant, it may crash, affect other quality attributes, and negatively impact the customers' happiness with the system.

5.4 Security

The degree to which the system defends itself from malicious users and protects its customers' data.

5.4.1 Acceptance Level

The system must guarantee the security of users' confidential information, such as credit card numbers, and allow each user to only modify their own data.

5.4.2 Importance

High. A fault in this attribute would create a lack of confidence in the system.

5.5 Performance

The ability of the system to perform its functions within a specified time and efficiently use its resources under certain conditions.

5.5.1 Acceptance Level

Requests to the system must not exceed 2 minutes, not including the user interaction. The high demand on a specific event can bring latency to server responses, but it must not exceed 2 minutes of waiting.

5.5.2 Importance

Medium. Mainly on events with bigger dimensions, people already expect high demand and latency on the system part. Although it is important to take this attribute into account, it is not the most important.

5.6 Maintainability

The system's effectiveness and efficiency in being modified, corrected, or adapted to meet the specified system requirements.

5.6.1 Acceptance Level

This micro-services architecture brings modularity to the system, preventing high dependency between components. Each component must be tested individually. System updates should be launched every 3 months, and an update to a component should not affect the entire system.

5.6.2 Importance

Medium. Ensuring maintainability facilitates agility in responding to evolving requirements and enhances the system's long-term sustainability.

5.7 Usability

The degree to which a system brings satisfaction to a user. It is measured in factors such as intuitiveness, efficiency, learnability, and user satisfaction.

5.7.1 Acceptance Level

The mean total time spent by the user to fill in data and reach the final checkout should be less than 3 minutes.

5.7.2 Importance

Medium. Usability enhances user experience, leading to increased engagement and satisfaction, ultimately contributing to the success of the system.

5.8 Extensibility

The ability of the system to easily accommodate the addition of new features or functionality without requiring significant modification to its existing structure.

5.8.1 Acceptance Level

With the modularity already mentioned, the system should be able to incorporate new functionalities or components without affecting the entire architecture.

5.8.2 Importance

Low. In this project scope, we do not envision the frequent addition of new features.

5.9 Portability

The ease with which a software system can be transferred or adapted to different hardware or software environments.

5.9.1 Acceptance Level

The web application must be accessible through any device, and its frontend must be responsive to ensure ease of use across different devices.

5.9.2 Importance

Low. The web-based nature of the application inherently provides a level of portability across devices. Therefore, immediate emphasis on this attribute may be lower compared to other critical quality aspects.