ASSO - *TicketBoss*

Homework 7

Team 33
Guilherme de Matos Ferreira de Almeida
João Pedro Carvalho Moreira
Jorge Daniel de Almeida Sousa
Lia da Silva Linhares Vieira
Nuno Afonso Anjos Pereira

U. PORTO

Masters in Informatics and Computing Engineering

17/04/2024

Contents

1 Introduction

TicketBoss is a system designed to handle ticket sales for various entertainment events. It provides its users with the possibility to list various events that are happening in the surrounding area and to purchase tickets to attend those events. Each ticket corresponds to a given seat and any given user can hold at most 10 tickets at any given time. Each ticket is held for a maximum of 10 minutes without being purchased at which point it is released and made available for other users.

Upon purchasing a ticket, the user receives via e-mail a confirmation of the purchase they made, along with other relevant information such as invoices and the tickets themselves. Payment details and processing are handled by third parties.

This document describes our group's proposed design of this system, including several architectural views of the system, alongside descriptions of 2 important use cases.

2 View-Model Architecture

2.1 Logical view

The logical view of a system focuses on components and how their interactions contribute to achieving the system requirements. We decided to represent TicketBoss' logical view with a class diagram, which is presented below.

2.1.1 Class Diagram

The following diagram shows the various logical entities present in the system and the relationships between them. It provides insights into how data might be modeled in this system.

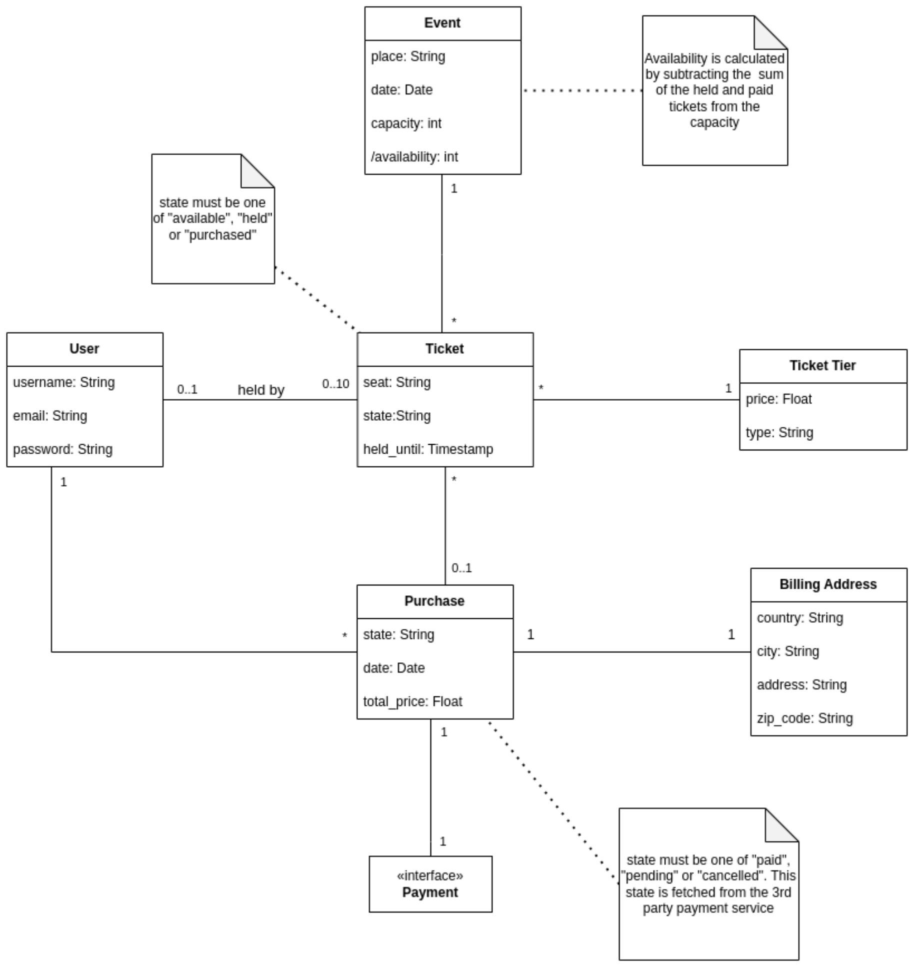

Figure 1: TicketBoss Class Diagram

2.1.2 Event

Represents an entertainment event hosted by TicketBoss. It contains information such as the event location, date, total capacity, and the current availability of seats, which is calculated by subtracting the sum of held and paid tickets from the total capacity.

2.1.3 Ticket

Represents a ticket for an event. Each ticket is associated with a specific seat and has a state indicating whether it's *available*, *held*, or *purchased.* If a ticket is in the *held* state, it also includes the timestamp until which it is held.

2.1.4 Ticket Tier

Represents a tier of tickets available for purchase for an event. It includes the price of the ticket tier and a type indicating the category of tickets.

2.1.5 Purchase

Represents a purchase transaction made by a user for tickets. It includes information such as the current state of the purchase, which can be *paid*, *pending*, or *canceled*, the date, and the total price of the purchase. The purchase state is fetched from the third-party payment service.

2.1.6 Billing Address

Represents the billing address associated with a user's purchase. It includes details such as the country, city, street address, and ZIP or postal code.

2.1.7 User

Represents a user of the TicketBoss system. It includes basic user information such as the username, email address, and password.

2.1.8 Payment (Interface)

Represents the payment interface utilized for processing payments. This interface allows TicketBoss to integrate with various third-party payment service providers. Implementation of this interface handles payment authorization, transaction processing, and communication with external payment gateways, providing flexibility and security in payment processing operations.

The main entity in this diagram is the Ticket class since this is the entity that the rest of the system will revolve around.

2.2 Development/Implementation view

At the implementation level, following from the Class Diagram presented in Picture 1, the main component of the system will be the **Ticket Management** module, which will handle most of the business logic required of the system. There are separate modules, like the **Ticket Timer**, which is responsible for managing seat-holding timers for a given ticket.

2.2.1 Component Diagram

The following Component Diagram serves as a more formal description of the **Implementation View**:

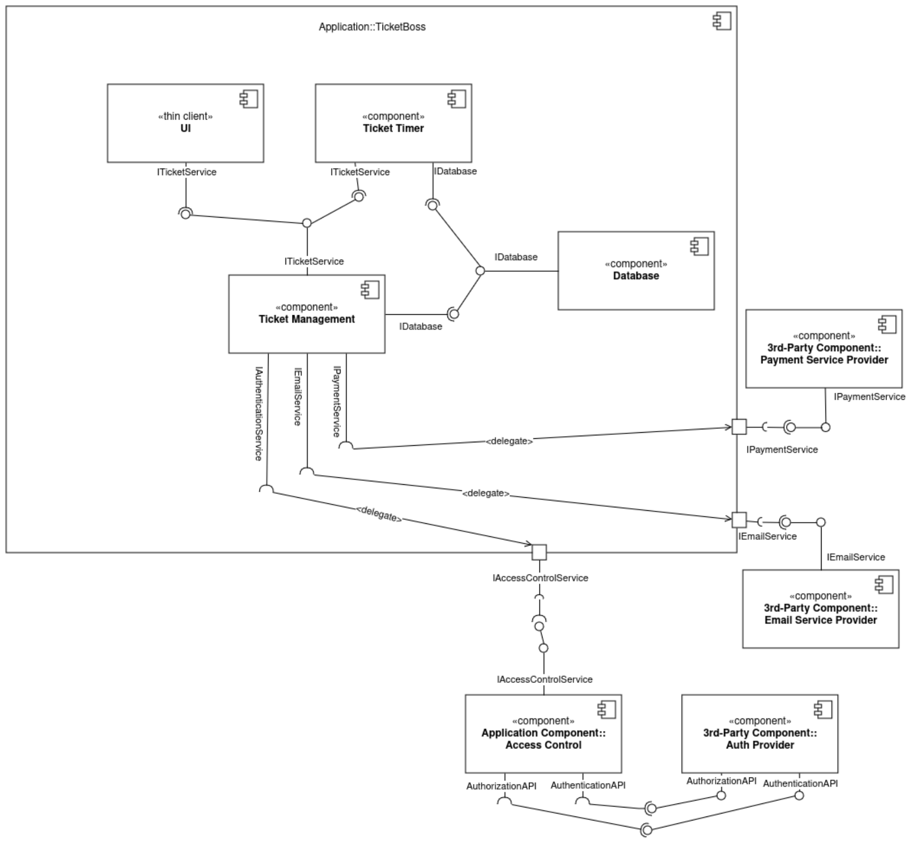

Figure 2: TicketBoss Component Diagram

Here we can highlight the central role of the **Ticket Management** component. Payment processing and email dispatching are all handled by 3rd-party providers.

We decided to have authentication and authorization handled by a system component that is external to the **TicketBoss** system itself. This can be implemented as a 1$^{\text{st}}$-party service or as a dependency on a 3$^{\text{rd}}$-party auth-provider:

- The former provides more control over what data is stored but implies that we have to comply with local and government regulations regarding data storage standards and security practices;
- The latter makes it easier to provide external authentication methods at the expense of having less control over that part of our system;

Both approaches offer the advantage that the *auth* solution is not tied to the system itself, allowing for easy replacement of any vulnerable component of the authentication/authorization process.

2.3 Process view

2.3.1 Sequence Diagram

The basic sequence of actions, as described in the requirements for the system, for a typical usage scenario of **TicketBoss** is as follows:

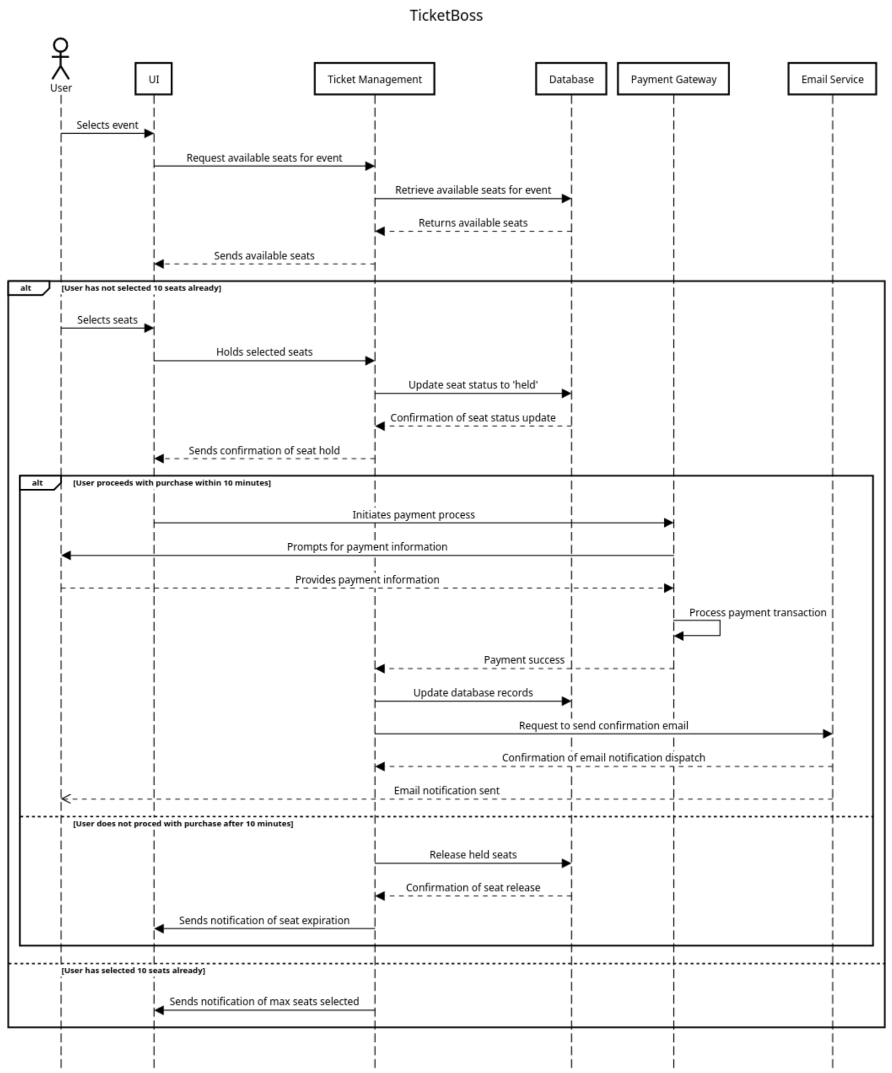
TicketBoss
User
UI
Ticket Management
Database
Payment Gateway
Email Service
Selects event
Request available seats for event
Retrieve available seats for event
Returns available seats
Sends available seats
alt
[User has not selected 10 seats already]
Selects seats
Holds selected seats
Update seat status to 'held'
Confirmation of seat status update
Sends confirmation of seat hold
alt
[User proceeds with purchase within 10 minutes]
Initiates payment process
Prompts for payment information
Provides payment information
Process payment transaction
Payment success
Update database records
Request to send confirmation email
Confirmation of email notification dispatch
Email notification sent
[User does not proced with purchase after 10 minutes]
Release held seats
Confirmation of seat release
Sends notification of seat expiration
[User has selected 10 seats already]
Sends notification of max seats selected

Figure 3: TicketBoss Sequence Diagram

The ticket purchase sequence involves the following steps:

1. **User Interaction**:

- The user interacts with the UI to select an event.
- The UI communicates with the Ticket Management module to request available seats for the selected event.

2. **Seat Selection**:
   - The Ticket Management module retrieves available seats for the event from the Database and sends them to the UI.
   - The user selects seats through the UI, and the UI informs the Ticket Management module to hold the selected seats.

3. **Seat Holding**:
   - The Ticket Management module updates the status of the held seats in the Database.
   - The UI receives confirmation of the seat hold from the Ticket Management module.

4. **Payment Process**:
   - If the user completes the purchase within 10 minutes, the UI initiates the payment process with the Payment Gateway.
   - The Payment Gateway prompts the user for payment information and processes the payment transaction with the Database.
   - Upon successful payment, the Payment Gateway sends a confirmation to the Database, which updates the transaction status.
   - The Payment Gateway notifies the Email Service to send a confirmation email to the user.

5. **Seat Expiration**:
   - If the user does not complete the purchase within 10 minutes, the held seats expire.
   - The Ticket Management module releases the held seats by updating their status in the Database.
   - The Ticket Management module notifies the UI to inform the user about the expiration of the held seats.

This process view provides a detailed understanding of the interactions and steps involved in the ticket purchase sequence within the TicketBoss system. Each component plays a crucial role in ensuring the smooth execution of the ticket purchase process.

The following sequence diagram illustrates the actions involved in the login process:

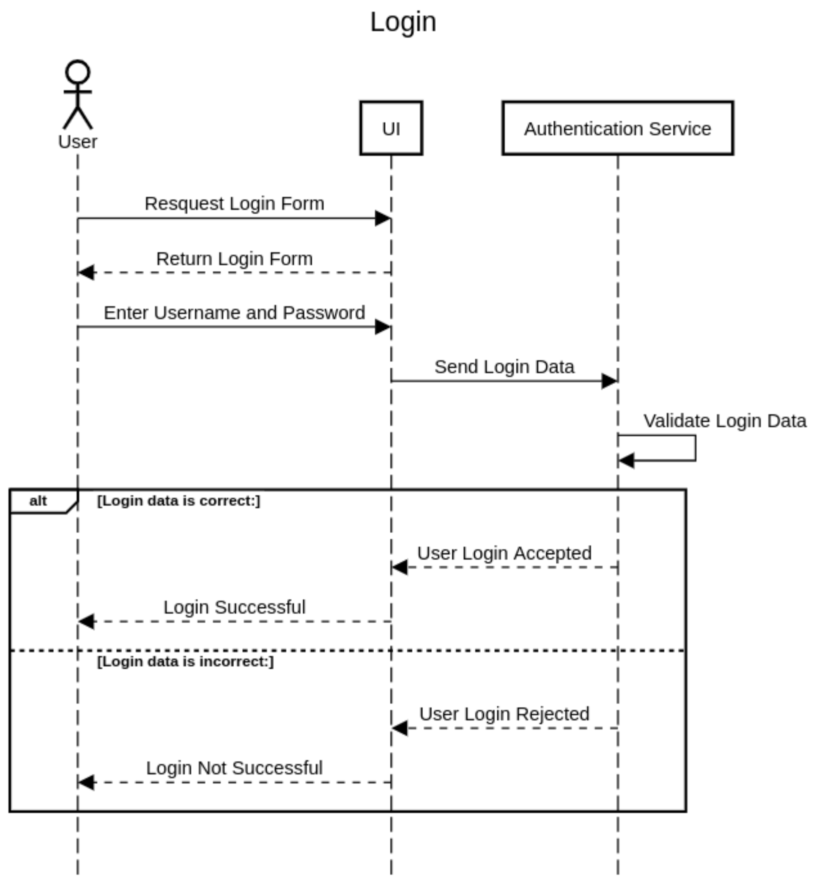

Figure 4: Login Sequence Diagram

2.4 Physical View

There are 2 main architectural approaches to implementing this system in terms of components that can be deployed into any existing physical infrastructure:

- Client-Server
- Middleware/Broker (+ Micro-services)

2.4.1 Client-Server

Using a **Client-Server** architectural approach, the system would have many clients, corresponding to the various users of the platform, and a server (or set of servers, in case redundancy measures are taken), all of which might be connected through various other network-related components (i.e. proxies or load-balancers). This pattern presents itself as the simpler approach of the 2, requiring that the developers potentially implement the system as a big monolith that handles all business logic.

2.4.2 Middleware/Broker

Using a **Middleware/Broker** architectural style, the system would employ a central component (the broker, which might not be restricted to a single component for redundancy purposes, as is the case in, for example, a distributed Kafka cluster) that would be responsible for accepting and routing all messages within the system. This enables loose coupling of the components, allowing them to scale according to their usage needs. For example, during a peak in traffic to the ticket management system due to increased demand for tickets for a given event. This approach also invites using a **Micro-services** architectural pattern as a framework for dividing and separating system concerns from one another.

2.4.3 Conclusion

While the **Client-Server** architecture has been proven to work pretty decently[1],[2] the techniques needed to implement it, for our given use case, are not trivial nor standard practice. Although it would require more initial effort to build, adopting a **Middleware/Broker** pattern not only allows for independent teams to work on their part of the system, potentially speeding up development but also enables seamless extensions of the system in the future, without needing to change the code-base entirely. As such, this is the architectural pattern we will be adopting, as can be seen in the following:

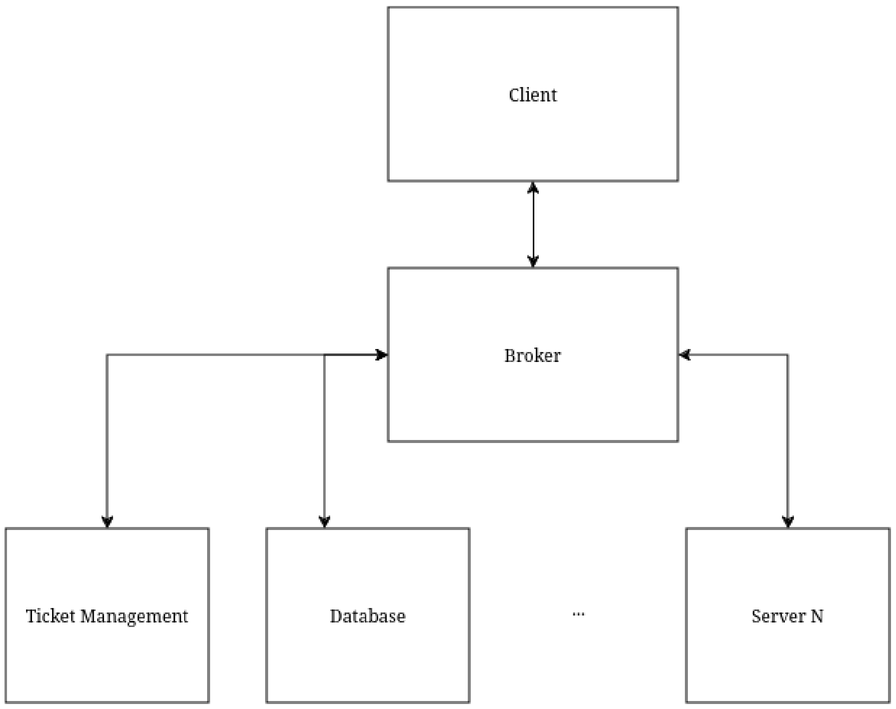

Figure 5: TicketBoss Physical View

2.5 Use-case view

While there were many use cases that we had to consider while designing the system, many of them overlapped in their flows. For this reason, we have written a use-case diagram and chosen 2 use cases that best depict the system's usage.

2.5.1 Use Case 1 - User selects seats and buys them

This use case describes the process where a user selects seats for an event and completes the purchase. The main flow goes as follows:

1. The system displays the available seats for the selected event.
2. The user selects one or more seats.
3. The system holds the selected seats for ten minutes.
4. The user proceeds to checkout.
5. The system prompts the user to enter payment information, including credit card details and billing address.
6. The user confirms the purchase.

7. The system processes the payment and updates the booking status for the selected seats.
8. The system sends a confirmation email to the user.
9. The user receives the confirmation email with the tickets.

Alternate Flow:

In step 4: If the user does not complete the purchase within ten minutes:

5. The system releases the held seats.
6. The user is notified that the seats have expired.

2.5.2 Use Case 4 - Two or more people select seats for the same event at the same time

This use case describes the process where two or more users select different seats for the same event at the same time. The actors are User1 and User2. The main flow goes as follows:

1. User1 and User2 navigate to the same event page.
2. User1 selects one or more seats for the event.
3. User2 also selects seats for the same event but does not overlap with User1's selection.
4. Both User1 and User2 proceed to checkout.
5. The system handles each user's checkout process independently, prompting both users to enter payment information, including credit card details and billing address. There are no conflicts since the users have selected different seats.
6. The users confirm the purchase.
7. The system processes the payment and updates the booking status for the selected seats.
8. The system sends a confirmation email to the users.
9. The users receive the confirmation email with the tickets.

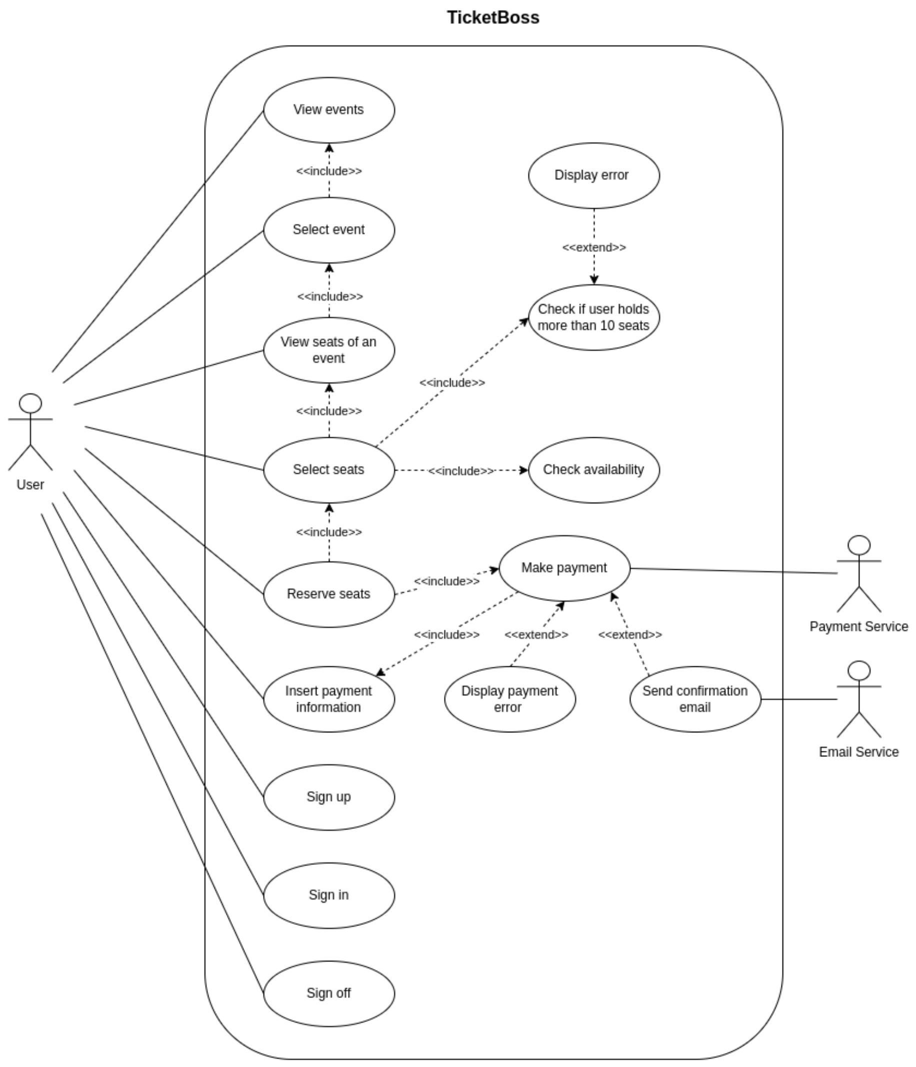

Figure 6: Use Case Diagram

3 Quality Attribute Requirements

Keeping **TicketBoss** in mind, we have identified and prioritized key quality attributes for consideration. The selected attributes are:

3.1 Functional Correctness

This quality attribute describes the system's ability to perform its intended tasks, such as accurately displaying the available events, seats, and prices and facilitating the purchase process without errors. A very high degree of functional correctness, ideally 100%, is required. Customers expect the system to work flawlessly, therefore any mistakes or inaccuracies could cause them to become unhappy and abandon purchases or financial losses for both the company and the customer. Making sure everything works correctly contributes to user trust and satisfaction.

In the specific case of **TicketBoss**, this is handled at the application level, using strong-consistency-based techniques to ensure the entire system has a uniform view of its state at any given moment.

3.2 Security

This quality attribute refers to the protection of the system, user data, and transactions from unauthorized access. This involves protecting user payment information, and personal details, and ensuring transaction integrity. Any system that handles user data and financial transactions must prioritize security. Failure to maintain adequate security measures can lead to repercussions, such as monetary losses, legal troubles, and harm to **TicketBoss**' reputation.

As mentioned previously, we are implementing both authorization/authentication and payment processing as external services, with the most likely option being opting for a 3$^{\text{rd}}$-party solution. As stated, this has the benefit of us not having to deal with the intricacies of complying with industry standards for data security, enabling us to shift our focus to other development tasks.

3.3 Availability

This quality attribute ensures that users can access the platform, browse events, select seats, and complete purchases without encountering downtime or unavailability. The system should aim for near 100% availability, with minimal downtime for maintenance or unexpected issues. To minimize the impact on user experience, users should be informed in advance of any planned downtime. Unavailability can result in lost sales opportunities and frustrated users. **TicketBoss** may guarantee continuous service delivery and satisfy user expectations by prioritizing availability and implementing measures such as redundancy and fail-over mechanisms.

3.4 Scalability

Scalability guarantees that **TicketBoss** can adapt to changing demands, user growth, and future expansion plans while maintaining high performance, responsiveness, and reliability. It's a critical quality attribute that directly impacts the system's ability to meet the needs of its users and stakeholders effectively.

The system's architecture ensures that scaling the system is as easy as instantiating more replicas of each component, providing an easy way to ensure the high degree of scalability needed for this type of system.

3.5 Maintainability

Maintainability is essential for ensuring that **TicketBoss** remains adaptable, reliable, and cost-effective throughout its lifecycle. By prioritizing maintainability, the development team can mitigate risks, respond to changes quickly, and deliver value to users consistently.

This is especially important in our case since we are adopting a more "distributed" architectural approach and as such knowledge of each sub-system is more spread-out and possibly harder to track. Ensuring the maintainability of the system not only helps current developers as well as future team members who might be introduced to the project.

4 Architectural Patterns

As mentioned previously, mainly in the section 2.4, the proposed solution would follow a **Middleware/Broker** architectural style. This enables high-decoupling of system components and allows different parts of the system to scale according to their traffic: in a system like **TicketBoss**, this is especially important when, for example, a famous singer is holding a concert or there is some other famous show happening. Another key benefit is that it promotes concurrent development since each sub-system is not constrained by the others concerning its deployment schedule and maintenance.

This description resembles the **Micro-Services** pattern a lot, which could be used on top of the proposed style. As seen in the **Implementation View** section, there has been an effort to divide systems into smaller components that each do their part. Since no component is dependent on the others, the system's nature becomes inherently asynchronous. While other architectural patterns might be useful for this, the **Broker** pattern excels in these cases, serving as the foundation for others, more specific styles, such as **Event-Driven** patterns.

One could argue that a simpler approach could be more beneficial. However, we think that the long-term benefits of employing a "more complicated" architectural pattern from the very beginning outweigh the short-term problems that might be faced.

5 Conclusion

In conclusion, the architecture of **TicketBoss** is designed to provide a reliable, secure, and user-friendly platform for handling ticket sales. By prioritizing key quality attributes such as functional correctness, security, and availability, we ensure a positive user experience and maintain customer trust. The chosen architecture patterns and design decisions aim to support the system's requirements while allowing for future growth and evolution.

References

[1] Navjot Bansal. Case study: How stackoverflow's monolith beats microservice performance. https://www.linkedin.com/pulse/case-study-how-stackoverflows-monolith-beats-navjot-bansal/.

[2] Theresa. How will you design the stack overflow website? https://blog.bytebytego.com/p/ep27-stack-overflow-architecture