SA
Documentation

# RATTE

Semester: Autumn 2025

Version: 00.01
Date: 2025-12-19 15:34:58Z


**Project Team:** Doriano Di Pierro
Silvan Lendi
Fabio Zahner

**Project Advisor:** Prof Dr. Frieder Loch




School of Computer Science
OST Eastern Switzerland University of Applied Sciences

# Abstract

The Rattenfest, a local festival in Rapperswil, faced the challenge of replacing its generic ticketing provider with a solution that offers greater control, lower costs, and improved usability for its volunteer committee. This project documents the design and implementation of RATTE, a software suite tailored to these specific needs.

The solution contains a robust microservice backend and three frontend applications: a web-based Ticket shop for guests, an administrative Dashboard for organizers, and a mobile Scanner application for entrance staff. Utilizing a modern technology stack including React, React Native, Golang and Kubernetes, the system was engineered for reliability, scalability, and ease of maintenance.

The outcome is a usage ready ticketing platform that successfully meets the functional and non-functional requirements of the festival. It empowers the organization with full data ownership and full control, proving that custom software can effectively replace commercial alternatives for smaller, community-driven events.

# Management Summary

The volunteer-run Rattenfest in Rapperswil required a tailored, cost-effective replacement for their ticketing system. The primary challenge was to develop a solution that met the festivals specific needs while still being a good foundation to build upon for future work and maintenance. Another challenge was the initial planning and architecture of the system to ensure a scalable and maintainable solution.

## The Solution: RATTE

To address these needs, we developed the RATTE system, a comprehensive suite of four applications. The Ticket shop enables guests to purchase tickets using TWINT or Credit Card, redeem resident vouchers, and recover lost tickets. The Dashboard provides organizers with real-time sales statistics, revenue tracking, and management tools. For event staff, the Ticket Scanner mobile app utilizes device cameras for entrance validation and includes a "Box Office" mode for on-site sales. Supporting these is the Central Backend, a robust microservice architecture ensuring data consistency, security, and high availability.

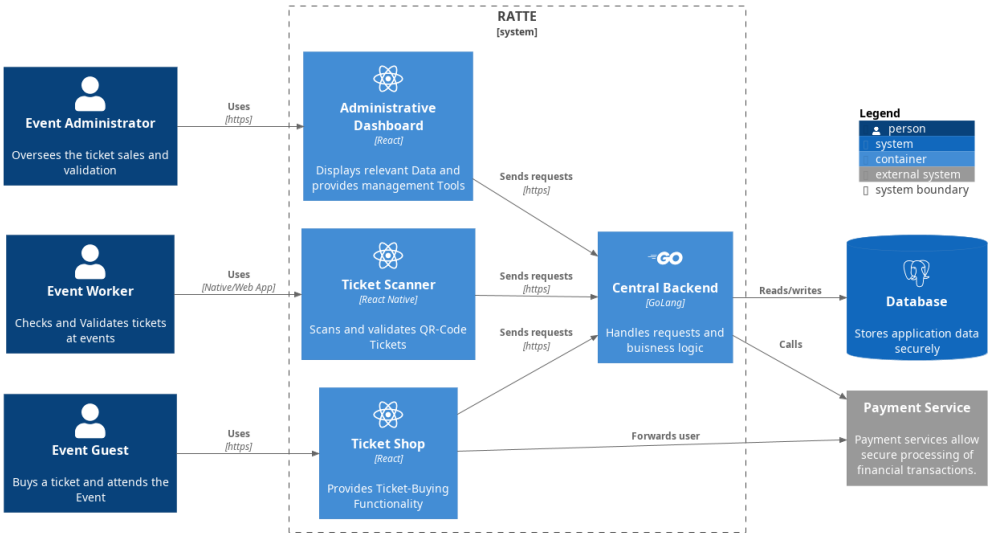



Figure 1: Container diagram of the RATTE system

## Key Achievements

The project successfully delivered a fully functional system meeting almost all functional requirements. Microservice architecture and Kubernetes ensure scalability, with load tests confirming stability under high traffic. The use of a modern technology stack, React web applications, and React Native with Expo for mobile, guarantees future-proof applications.

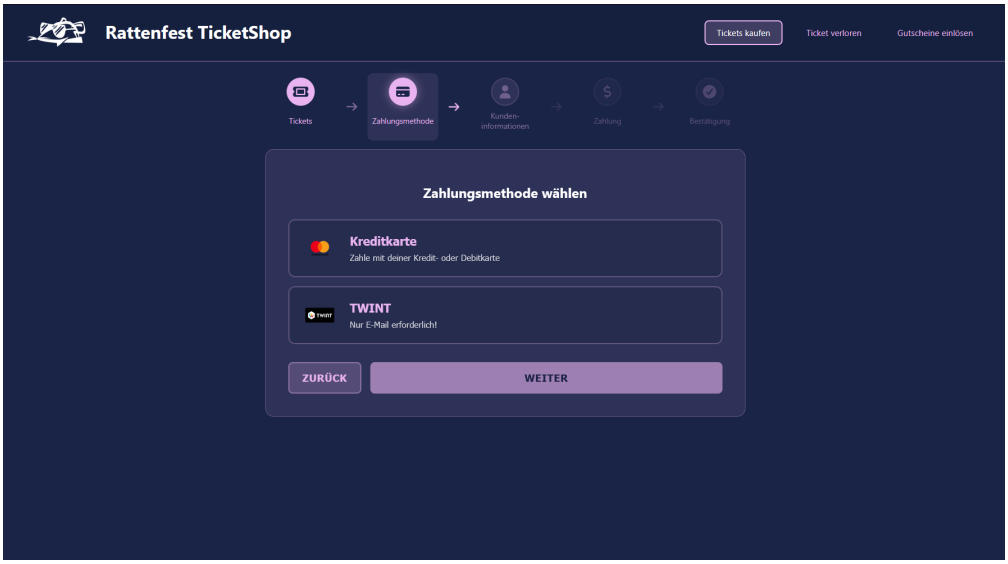



Figure 2: Ticket shop depicting payment selection

## Conclusion

The RATTE system represents a significant upgrade, providing the Rattenfest organizers with a powerful, open-source tool ready for deployment. The project met its technical goals and established a solid foundation for future enhancements, such as database redundancy and further customization options.

# Contents

# Chapter 1

# Problem Description

## 1.1 Goal of the Project

The Rattenfest is a small, volunteer-run local festival in Rapperswil attended by a few thousand people each year. Currently, ticketing is handled with ticketcorner light, which is a generic solution for ticketing. This project aims to deliver a ticketing application that is tailored to the specific needs of the Rattenfest, which we will get into later in the documentation, and can be operated intuitively. Ease of use and documentation are important as the organizational committee changes every year, and there is no guarantee that members with IT experience will be part of the committee in future years.

Thus, the goal of our project is to build an application which allows the sale and management of tickets for an event or festival. The application should be able to provide a solution for selling and validating tickets, as well as providing the necessary data to event organizers. We will focus on ease of installation and usability for event organizers as well as event guests. Additionally, we want to ensure that the resulting application can efficiently handle a high volume of requests while maintaining performance, even with limited resources.

## 1.2 Functional Requirements

### 1.2.1 Sources of User Stories and Functional Requirements

The functional requirements were created by asking members of the organizational committee of the Rattenfest, students who have attended the festival multiple times in the past and people who have helped out with ticket scanning at the entrance of the festival. By asking them about their experiences, figuring out what went well and where there are areas of improvement, we were able to derive user stories which are relevant for this project and from which we could define the functional requirements.

### 1.2.2 Actors

We start off by defining key actors and their goals. These actors will help in creating user stories which will then be used to define the functional requirements.

- **Festival attendee:** A person who wants to attend a festival.
- **Festival organizer:** A person who wants to organize a festival.
- **Box office worker:** A person who works at the entrance of a festival.

### 1.2.3 User Stories to Functional Requirements

The functional requirements are intentionally written in a granular way, so that they can be easily tested and verified. This also removes the need for acceptance criteria for each user story, as the functional requirements are already specific enough to be tested directly.

---

**User Story 1:** As a festival attendee, I want to buy a ticket for me and/or my friends online and pay using my preferred payment method, so that I can attend the festival.
**Functional Requirements:**

- **FR-01:** The system must allow a person to identify themselves using an email address.
- **FR-02:** The system must not require more personal information than an email address for payments with TWINT.
- **FR-03:** The system must allow a person to buy a ticket using a credit card.
- **FR-04:** The system must allow a person to buy a ticket using TWINT.
- **FR-05:** The system must send the ticket to the person's email address.
- **FR-06:** The system must create a unique QR code for each ticket.
- **FR-07:** The system must allow a person to buy multiple tickets in one transaction.
- **FR-08:** The system must prevent the purchase of tickets when the maximum number of tickets has been sold.

---

**User Story 2:** As a festival attendee, I want to recover my lost tickets to my email, so that I can still attend the festival even though I deleted the email containing the tickets.
**Functional Requirements:**

- **FR-09:** The system must allow a person to recover a lost ticket using their email address.

---

**User Story 3:** As a festival organizer, I want to sell a certain amount of tickets online for a set price, so that I can control the maximum number of attendees for the festival and make sure I generate enough revenue to cover the costs.
**Functional Requirements:**

- **FR-10:** The system must allow an organizer to set the price of a ticket.
- **FR-11:** The system must allow an organizer to set the maximum number of tickets available for sale.

---

**User Story 4:** As a festival organizer I want to make sure that only people with valid tickets can enter the festival.
**Functional Requirements:**

- **FR-12:** The system must allow the box office workers to scan a QR code using their mobile device camera.
- **FR-13:** The system must provide immediate feedback on whether the ticket is valid or not.
- **FR-14:** The system must mark a ticket as used after it has been validated to ensure it cannot be used again.

---

**User Story 5:** As a festival organizer, I want to allow nearby residents to attend the festival for free, so that I can give them compensation for the noise.
**Functional Requirements:**

- **FR-15:** The system must allow an organizer to create QR code vouchers that can be used on the website to generate free tickets.

---

**User Story 6:** As a festival organizer, I want to view statistics on ticket sales, so that I can monitor the success of the Rattenfest and make informed decisions for the next year.
**Functional Requirements:**

- **FR-16:** The system must show the number of tickets sold.
- **FR-17:** The system must show the revenue generated from ticket sales.
- **FR-18:** The system must allow the organizer to filter statistics by year.
- **FR-19:** The system must show a chart of the ticket sales per day in a selected week.

- **FR-20:** The system must show a chart of the ticket sales per hour in a selected day.
- **FR-21:** The system must show a chart on how many tickets of each type have been sold.

---

**User Story 7:** As a festival organizer, I want to view statistics on ticket scans, so that I can monitor when most people attend the festival to optimize staffing and know how many tickets were not used.
**Functional Requirements:**

- **FR-22:** The system must show a chart of the ticket scans per hour in a selected day.
- **FR-23:** The system must provide a graphical comparison of how many tickets have been scanned and how many were left unused.

---

**User Story 8:** As a festival organizer, I want the tickets to have the sponsors logos on them, so that I can achieve better sponsor deals.
**Functional Requirements:**

- **FR-24:** The tickets should include logos of the sponsors on them.

---

**User Story 9:** As a festival organizer, I want to see past ticket sales and revenue and compare them to current sales and revenue in graphs, so that I can see trends.
**Functional Requirements:**

- **FR-25:** The system must provide historical data on ticket sales and revenue for the past years.
- **FR-26:** The system must provide visualizations of historical data in the form of graphs and charts.

---

**User Story 10:** As a festival organizer, I want to be able to send free tickets to helpers, artists and the committee, so that they can attend without charge.
**Functional Requirements:**

- **FR-27:** The system must allow an organizer to generate and send free tickets via email to specified recipients.

---

**User Story 11:** As a festival organizer, I want the dashboard to update periodically without me having to refresh the page, so that I always have the latest information during the festival without having to manually refresh.
**Functional Requirements:**

- **FR-28:** The system must allow the dashboard to update periodically without manual refresh.
- **FR-29:** The system must allow the user to set the update interval and if desired pause the automatic updates.

---

**User Story 12:** As a festival organizer, I want to be able to see which percentage of vouchers have been used, so that I can evaluate if they are needed in the future.
**Functional Requirements:**

- **FR-30:** The system must provide statistics on the number and percentage of vouchers that have been redeemed for tickets.

---

**User Story 13:** As a festival organizer, I want to be able to delete tickets, so that I can simplify a refund process if needed and also correct mistakes that can happen during in-house sales.
**Functional Requirements:**

- **FR-31:** The system must have a view that shows all sold tickets with an option to delete them.
- **FR-32:** The system must still show the deleted tickets in the view, but mark them as deleted.
- **FR-33:** The system must ensure that deleted tickets can no longer be used for entry.
- **FR-34:** The system must allow the user to search for specific tickets by email.

---

**User Story 14:** As a festival organizer, I want to allow the box office workers to report tickets sold in the box office, so that I can have a complete overview of ticket sales and how many people are attending the festival.
**Functional Requirements:**

- **FR-35:** The system must allow the box office worker to report tickets sold in box office.
- **FR-36:** The system must mark tickets sold in box office as scanned, as they will not be scanned at the entrance.

---

**User Story 15:** As a technically versed festival organizer, I want to be able to understand the underlying application and be able to make minor modifications, so that I can tailor the system to new requirements that might come up.
**Functional Requirements:**

- **FR-37:** The system must be open source and documented well enough for a technically versed person that has the necessary know how of the used technologies to be able to understand the application and make minor modifications.

---

**User Story 16:** As a festival organizer, I want to be able to download the raw data of ticket sales and vouchers as a XLSX file, so that I can create custom reports and charts.
**Functional Requirements:**

- **FR-38:** The system must allow the user to download the raw data of ticket sales and vouchers as a XLSX file.

---

### 1.2.4 Classification of Functional Requirements

The functional requirements are divided in three categories in Table 1.1 based on the MoSCoW method [Pronda].

| | | |
|---|---|---|
| | Must have | Non-negotiable product needs that are mandatory for the system to function correctly and meet its core objectives. |
| | Should have | Requirements that are important for usability, efficiency, or reliability, but not strictly essential. The system can function without them, but with reduced quality or convenience. |
| | Could have | Requirements that are desirable enhancements or additional features. |

Table 1.1: Classification of Functional Requirements

| **Requirement** | **Classification** |
|---|---|
| FR-01: Email identification | |
| FR-02: Minimal personal info | |
| FR-03: Credit card payment | |
| FR-04: Twint payment | |
| FR-05: Email ticket delivery | |
| FR-06: Unique QR code per ticket | |
| FR-07: Multiple tickets per transaction | |
| FR-08: Limit ticket sales | |
| FR-09: Ticket recovery via email | |
| FR-10: Organizer sets ticket price | |
| FR-11: Organizer sets ticket limit | |
| FR-12: QR code scanning | |
| FR-13: Immediate ticket validation | |
| FR-14: Mark ticket as used | |
| FR-15: Ticket vouchers for residents | |
| FR-16: Sale statistics | |
| FR-17: Revenue statistics | |
| FR-18: Filter statistics by year | |
| FR-19: Sales per day | |
| FR-20: Sales per hour | |
| FR-21: Tickets per type chart | |
| FR-22: Scans per hour | |
| FR-23: Amount of tickets scanned | |
| FR-24: Sponsor logos on tickets | |
| FR-25: Historical data visualization | |
| FR-26: Graphical comparison of yearly data | |
| FR-27: Free tickets sending | |
| FR-28: Auto-updating dashboard | |
| FR-29: Configurable update intervals | |
| FR-30: Voucher usage statistics | |
| FR-31: View sold tickets with delete option | |
| FR-32: Show deleted tickets | |
| FR-33: Make deleted tickets unusable | |
| FR-34: Search tickets by email | |
| FR-35: Box office reporting | |
| FR-36: Box office tickets scanned | |
| FR-37: Open source and documentation | |
| FR-38: Download raw data as XLSX file | |

## 1.3 Non-Functional Requirements

The non-functional requirements define the general properties and constraints of the system. They should apply to all components of the system.

- **NFR-01 (Uptime):** The attendee system must be available 99% of the time during the ticket sales period (march-may) and a 99.99% uptime during the 5 days around the festival (29. April - 3. Mai). Uptime will be measured for the Ticket Sale website as well as the Dashboard, so both server- and application outages will be included.
- **NFR-02 (Stability under Load):** The system must support at least 50 concurrent users while keeping loading times for all webpages below 1 second in performance, as this is perceived as seamless for users [Nie10].
- **NFR-03 (Cross-Browser support):** The system must ensure all functionality on all major browsers (Chrome, Firefox, Edge, Safari).
- **NFR-04 (Website Design):** The system's user interface must match the visuals of the rattenfest website. This is evaluated by the developers.
- **NFR-05 (Web Accessibility):** The system must be accessible according to WCAG standards. This is evaluated using Web Accessibility Evaluation Tools.
- **NFR-06 (Configurability):** The system must provide means to configure basic settings (e.g. ticket price, ticket limit) without requiring application code changes. This is evaluated by the developers.
- **NFR-07 (Recoverability):** The system must recover from a partial or complete server crash within 5 minutes, minimizing downtime. This is evaluated through simulated crash tests.
- **NFR-08 (Reliability):** The system must provide a way to monitor server health and logs in real-time. This is evaluated by the developers.
- **NFR-09 (Ease of deployment):** The system must be deployable by event staff with basic technical knowledge using provided documentation. This is evaluated by checking if all steps in the deployment guide can be followed without additional help.

# Chapter 2

# Market Analysis

This chapter contains the market analysis of existing ticketing solutions. It shows the evaluation of selected commercial and open-source applications in regard to the functional requirements established in section 1.2 to determine whether to use an existing application, build on an existing open-source project or to develop a custom ticketing system. The chapter is organized into a description of the existing ticketing services, a feature comparison table and a decision section that interprets the comparison and explains our choice.

## 2.1 Existing Ticketing Services

This section describes four ticketing services that were evaluated for this project: Ticketcorner Light, Eventfrog, Eventbrite and hi.events. For each provider we summarize its positioning, business model and typical use-cases.

### Ticketcorner Light

Ticketcorner is a major Swiss ticketing provider with 1.4 million website visitors per month and over 900 event organizers [Ticnd]. The "Light" naming refers to a version of their platform that is aimed at smaller events and organizers that need a simple sales interface without the full enterprise feature set. Ticketcorner Light is a commercial, closed-source platform that integrates with common payment providers and provides standard ticketing features.

### Eventfrog

Eventfrog is a ticketing and event platform that was founded in Switzerland and targets small and medium-sized events. It has 80'000 event organizers and over 1 million app downloads [Evendb]. Eventfrog is commercial and closed-source, but offers better pricing compared to other solutions.

### Eventbrite

Eventbrite is a global, widely used, closed-source ticketing marketplace and platform. It has around 89 million monthly users and hosts 4.7 million events in nearly 180 countries [Evenda]. Eventbrite the largest of the ticketing systems we evaluated and offers events in many categories.

### hi.events

hi.events is an open-source ticketing project that emphasizes flexibility and self-hosting. It also offers a commercial hosting service. hi.events promises the lowest fees and mentions being used by 1000s of organizers [Hi nd]. It has a scarce documentation for out of the box use and a subpar documentation for technical development and extension.

## 2.2 Comparison with Existing Ticketing Services

The tables 2.1 and 2.2 show a comparison of the four ticketing services Ticketcorner Light (abbreviated as T-Light in the table), Eventfrog, Eventbrite and hi.events. We compared them based on our functional requirements defined in the previous section to figure out which solution fits our needs best or whether we need to implement our own solution.

Table 2.1: Comparison of Ticket Solutions based on Functional Requirements

| **Functional Requirement** | **T-Light** | **Eventfrog** | **Eventbrite** | **hi.events** |
|---|---|---|---|---|
| FR-01: Email identification | ✓ | ✓ | ✓ | ✓ |
| FR-02: Minimal personal info | ✗ | ✗ | ✗ | ✗ |
| FR-03: Credit card payment | ✓ | ✓ | ✓ | ✓ |
| FR-04: Twint payment | ✓ | ✓ | ✓ | ✓ |
| FR-05: Email ticket delivery | ✓ | ✓ | ✓ | ✓ |
| FR-06: Unique QR code per ticket | ✓ | ✓ | ✓ | ✓ |
| FR-07: Multiple tickets per transaction | ✓ | ✓ | ✓ | ✓ |
| FR-08: Limit ticket sales | ✓ | ✓ | ✓ | ✓ |
| FR-09: Ticket recovery via email | ✓ | ✓ | ✓ | ✓ |
| FR-10: Organizer sets ticket price | ✓ | ✓ | ✓ | ✓ |
| FR-11: Organizer sets ticket limit | ✓ | ✓ | ✓ | ✓ |
| FR-12: QR code scanning | ✓ | ✓ | ✓ | ✓ |
| FR-13: Immediate ticket validation | ✓ | ✓ | ✓ | ✓ |
| FR-14: Mark ticket as used | ✓ | ✓ | ✓ | ✓ |
| FR-15: Ticket vouchers for residents | ✗ | ✗ | ✗ | |
| FR-16: Sale statistics | ✓ | ✓ | ✓ | ✓ |
| FR-17: Revenue statistics | ✓ | ✓ | ✓ | ✓ |
| FR-18: Filter statistics by year | ✓ | ✓ | ✓ | ✓ |
| FR-19: Sales per day | ✓ | ✓ | ✓ | ✓ |
| FR-20: Sales per hour | ✓ | ✓ | ✓ | ✓ |

Table 2.2: Comparison of Ticket Solutions based on Functional Requirements

| **Functional Requirement** | **T-Light** | **Eventfrog** | **Eventbrite** | **hi.events** |
|---|---|---|---|---|
| FR-21: Tickets per type chart | ✓ | ✓ | ✓ | ✓ |
| FR-22: Scans per hour | ✓ | ✓ | ✓ | ✓ |
| FR-23: Amount of tickets scanned | ✓ | ✓ | ✓ | ✓ |
| FR-24: Sponsor logos on tickets | ✗ | ✗ | ✗ | ✗ |
| FR-25: Historical data visualization | ✓ | ✓ | ✓ | ✓ |
| FR-26: Graphical comparison of yearly data | ✗ | ✗ | ✗ | ✓ |
| FR-27: Free tickets sending | ✗ | ✗ | ✗ | ✗ |
| FR-28: Auto-updating dashboard | ✗ | ✗ | ✗ | ✗ |
| FR-29: Configurable update intervals | ✗ | ✗ | ✗ | ✗ |
| FR-30: Voucher usage statistics | ✗ | ✗ | ✗ | ✗ |
| FR-31: View sold tickets with delete option | ✓ | ✓ | ✓ | ✓ |
| FR-32: Show deleted tickets | ✓ | ✓ | ✓ | ✓ |
| FR-33: Make deleted tickets unusable | ✓ | ✓ | ✓ | ✓ |
| FR-34: Search tickets by email | ✓ | ✓ | ✓ | ✓ |
| FR-35: Box office reporting | ✓ | ✓ | ✓ | ✓ |
| FR-36: Box office tickets scanned | ✓ | ✓ | ✓ | ✓ |
| FR-37: Open source and documentation | ✗ | ✗ | ✗ | |
| FR-38: Download raw data as XLSX file | ✓ | ✓ | ✓ | ✓ |

## 2.3 Decision

Based on the tables 2.1 and 2.2 hi.events supports the most functional requirements out of the four compared solutions. The vouchers for residents (FR-15) is highlighted in orange, as hi.events has coupons but does not allow a bulk creation of them, so it would require a lot of manual work to create the required amount of 200 vouchers. Hi.events is the only open-source solution in the comparison, which is a big advantage, however the documentation is lacking and it would require a significant amount of effort and work to understand how it works internally. Furthermore, hi.events has a non-transparent pricing model. One can either host the solution themselves, where only the payment processor fees of Stripe apply, or use the hosted service. For the hosted version, they claim it costs $0 for the organizer as shown in Figures C.27 and C.28, but when going to the actual pricing they make you choose to either increase the ticket price by 0.40 and 0.75% making the customer finance the fees or pay them yourself as shown in Figure C.28. This only makes the service free for the organizer if they make the customer pay for it, which is misleading. On top of this, the processing fees of Stripe still apply, which we will compare in the Figure 4.2.
Due to the missing bulk voucher creation, the lack of technical documentation, and the misleading pricing model, we decided to develop a custom solution instead of contributing to hi.events. This allows us to cover all functional requirements and optimize the workflow for the specific needs of the Rattenfest, additionally it allows for us to evaluate which payment processor is the most cost-effective and suitable for our needs.

# Chapter 3

# Project Management

This chapter describes how we managed the project and its product throughout its duration.

## 3.1 Risk management

In the following section we will describe the risks which could influence our project and how we will try to mitigate them.

### 3.1.1 Continuous Risk Management

Risk management is not a one-time task but an ongoing process that requires regular attention throughout the project. By continuously monitoring risks, we can ensure the project's success. We take the following steps to ensure correct risk management:

- **Risk Identification:** Identify new risks during meetings or as they emerge during the project lifecycle.
- **Risk Assessment:** Continuously assess/reassess risks determining if their likelihood or impact has changed.
- **Risk Mitigation:** Update risk mitigation strategies based on new assessments.
- **Risk Monitoring:** Continuously monitor risks and their mitigation measures.

### 3.1.2 Risk Matrix

The following Risk Matrix was used to assess the risks found:

| Likelihood | Impact | | | | |
|---|---|---|---|---|---|
| | **Insignificant** | **Minor** | **Significant** | **Major** | **Severe** |
| **Probable** | Moderate | High | Critical | Critical | Critical |
| **Likely** | Low | Moderate | High | Critical | Critical |
| **Moderate** | Negligible | Low | Moderate | High | Critical |
| **Unlikely** | Negligible | Negligible | Low | Moderate | High |
| **Rare** | Negligible | Negligible | Negligible | Low | Moderate |

### 3.1.3 Identified Risks

| **Risk** | **Mitigation** | **Risk Assessment** |
|---|---|---|
| **Payment Integration:** Payment processes prove difficult or impossible to integrate due to limitations in budget, time or technical abilities. | Multiple Payment providers are evaluated. This ensures that a backup provider is ready to be integrated if major difficulties are encountered with the "first priority" payment provider. | Likelihood: Moderate<br>Impact: Significant |
| **Insecure Software:** With insufficient resources invested into maintaining application security, attack points can be used to exploit the application, resulting in financial loss or loss of data (sold tickets or similar) | *As this has been identified as a critical risk, we have defined a more detailed mitigation strategy in Section 3.1.4.* | Likelihood: Likely<br>Impact: Major |
| **Scalability:** During periods of high load on the application (e.g. on the day of an Event), the application begins to slow to the point of being noticeable, or crashes completely. | The software being used (primarily the backend) are designed with performance and redundancy in mind. Additionally, Kubernetes will be used to allow for load-balancing important components. | Likelihood: Unlikely<br>Impact: Significant |
| **System Outage:** Unavailability of critical services or infrastructure due to factors outside our control, especially nearing the event, result in financial losses due to customers not being able to purchase a ticket. | Kubernetes will be used to allow for load-balancing in case of the failure of a single service. | Likelihood: Rare<br>Impact: Severe |
| **Scanner App Troubles:** The technologies that are required to allow the scanner app to function cross-platform (at least iOS & Android) do not allow implement an application which is adequate for the scanning functionality. | Web technologies will be used to ensure cross-platform for all relevant devices. | Likelihood: Moderate<br>Impact: Significant |

### 3.1.4 Mitigation of High and critical Risks

For high and critical risks, we have established more detailed mitigation strategies:

- **Insecure Software:**
  - **Assign a security officer:** A team member is designated. He will continuously monitor the security of the application to the best of his abilities and will create new tickets which will improve system security.
  - **Follow established Security Frameworks:** Security frameworks will be used to provide guidance and checklists for the various components of the system. Examples of frameworks might be
    * OWASP Top 10: List of Web application security risks
    * CIS Controls: Best practices in cybersecurity
    * CIS K8s Benchmarks: Kubernetes specific security guidelines and benchmarks.
    * Security Guidelines of Payment Providers: Ensure that customer payments are secure
    * Automated Scanning Tools: Fully- or Semi Automated Tools can highlight potential system weak-points.
  - **Communicate with a Specialist:** Once the application has reached our definition of a *Minimal Viable Product*, we will consult a Security expert which can help highlight both problems in our application and in our methodologies regarding Application Securities, or give us pointers on how to measure and improve the security of our cluster.

## 3.2 Project Timeline

We have planned the project in multiple milestones, each with its own goals and deliverables. A detailed timeline can be found in Appendix C.5.

# Chapter 4

# System Design and Development

This chapter outlines the steps we have taken and decisions we have made to plan, design and develop the RATTE system.

## 4.1 Technology Decision

During Market Analysis, Chapter 2, we have decided that we will create our own System to better fulfill the needs of a festival like the Rattenfest. For this, we need to choose a technology stack which will allow us to create this System. We have defined the following main criteria for all technology decisions:

- **Performance:** In our non-functional requirements, Section 1.3, we have also defined standards for speed and performance. Because the events we are catering towards will have limited resources to fulfill these NFRs, we have set resource-efficiency and speed as a focus making our decision.
- **Public Awareness:** The technologies we choose should be well known and established in the industry. Because the application is intended to be used by event organizers with limited technical experience, we want to ensure that there is a wide range of documentation, tutorials, and community support available.

In addition to these criteria, we also define context-specific criteria for each technology decision.

### 4.1.1 Backend Technology

The table 4.1 compares Go (Golang), Python and PHP. We have selected these languages for comparison because they are popular for use in API Web servers (see Criteria "Public Awareness"). Additionally, we have defined the following context-specific criteria:

- **Ease of Use:** How easy is it to learn and use the language?
- **Performance:** How well does the language perform in terms of speed and resource usage?

- **Scalability:** How well can scaling be handled in the language?
- **Use Case:** For which use case is the language best suited?

Table 4.1: Backend Language Comparison

| Feature | Go | Python | PHP |
|---|---|---|---|
| **Ease of Use** | Simple syntax with a large standard library. [vK17] | Simple, good readability extensive ecosystem of libraries [Pland] | Simple syntax [OYO23] |
| **Performance** | Exceptional performance with native compilation and efficient memory usage. [Blond] | Slower than go but still quite fast. [Dai21] | Poor performance, especially for concurrent operations without non-blocking I/O workaround. [Blond] |
| **Scalability** | Excellent built-in concurrency with Goroutine ([Blond]), perfect for microservices. [P23] | Limited by GIL for CPU-bound tasks, requires workarounds. [Pytnd] | Difficult to scale, poor concurrency model. [Zen23] |
| **Use Case** | Ideal for microservices and performance sensitive applications. [P23] | Good for data processing, scripting, and prototyping. | Mainly suited for traditional web applications. [OYO23] |

#### 4.1.1.1 Selection

After the analysis, we have decided to use **Golang** as our backend technology, because it fulfills our set criteria the best.

### 4.1.2 Frontend Technology

For the frontend technology, we have compared React, Angular and Vue.js, as they are the three leading frontend frameworks [Emm23] (see the criteria "Public Awareness"). Additionally to the main criteria, we have defined the following criteria for our comparison:

- **Ecosystem:** How large and active is the community around the framework, and how many third-party libraries and tools are available?
- **Flexibility:** How easily can the framework adapt to different project requirements?

Table 4.2: Frontend Framework Comparison based on [KV18, Ver22]

| Criteria | React | Angular | Vue.js |
|---|---|---|---|
| **Performance**[Ver22] | Highest Performance (25 Pts.) | Slightly slower than React (20 Pts.) | Significantly lower than both other frameworks (10 Pts.) |
| **Ecosystem** | Very large and active community with extensive third-party libraries and tools available. | Comprehensive built-in ecosystem with official packages for most use cases. | Growing community with good library selection, though smaller than React. |
| **Flexibility** | Supports both web and mobile development via React Native. Offers multiple state management solutions. | Opinionated and structured; excellent for large projects but less flexible for diverse use cases. | Flexible and versatile; performs well in multiple contexts with good state management options. |

#### 4.1.2.1 Selection

After the analysis, using the defined criteria (see Figure 4.2), we have decided to use **React** as our frontend technology.
React was chosen for our project because it offers the greatest flexibility: React Native allows us to build the scanner application for mobile platforms using the same language and code, reducing development time and effort. At the same time, React is comparable to Vue.js and Angular in terms of ecosystem size, making it the best choice for our needs.

### 4.1.3 End-to-end and load testing technology

For end-to-end and load testing, we have compared the three leading end-to-end testing frameworks: Cypress, Selenium and Playwright (Criteria "Public Awareness").

Table 4.3: End-to-end and load testing technology comparison based on [Andnd]

| Criteria | Cypress | Selenium | Playwright |
|---|---|---|---|
| **Initial setup** | Easy | Requires effort to set up. | Easy |
| **Ease of use** | User-friendly interface with minimal configuration required. | Requires more setup and has a steeper learning curve. | User-friendly interface with minimal configuration required. |
| **DOM handling** | Easy | Moderate | Easy |
| **Parallel execution** | Supports parallel execution using CI/CD tools. | Supports parallel execution. | Supports parallel execution. |
| **Load testing support** | Can be done using google lighthouse. [Brond] | Can be done using selenium grid. [Loand] | Extensive support with artillery [Artnd] |

#### 4.1.3.1 Selection

After the analysis of the article [Andnd] with some important criteria summarized in the table 4.3, we have decided to use Playwright as our end-to-end testing technology. Even though the article outlined that Cypress is on par with Playwright in most categories, Playwright offers a superior load testing baseline with artillery, which is a great advantage for us since we want to ensure that our system can handle high loads during ticket sales (NFR-02).

### 4.1.4 Payment Processor

Festival attendees have the ability to buy tickets in the ticket shop. In order to complete transactions online, we need an interface that allows attendees to select a payment method like credit card or twint, add their personal details and complete the purchase.

The EU and card issuers like Mastercard have strict regulations on how you are allowed to handle payment information and process transactions because of fraud protection, security standards for encryption, and consumer protection. This leads us having to outsource the process of customer payment to other companies which have licenses and meet the regulatory requirements.

The government of Switzerland offers a list of most used and reputable providers on their website[1]. The website also describes the role of the so called payment-service-provider (PSP) and financial institutions, the acquirers, that settle the transactions.

[1] https://www.kmu.admin.ch/kmu/en/home/concrete-know-how/sme-management/e-commerce/creating-own-website/online-payment-methods/payment-service-providers.html

#### 4.1.4.1 Roles in the payment process

We describe the roles of the PSP and the acquirers in more detail here for our use case: The PSP will provide a widget or redirect page to their own website where the user enters their payment details. This means we do not handle sensitive card information and the responsibility of secure handling is passed to the PSP. The acquirer has the role of checking whether the credit card is valid and has not reached its limit before it initiates the money transfer. Some PSP offer both services at once while others allow or require you to sign a contract with an acquirer.

Figure 4.1 shows the steps and roles that are involved in the process. The figure shows the customer, making an order (*Order*) via the payment processor. The payment processor forwards the request to the acquirer (*Settle*) where if the card is valid, and it has not reached its limit, it makes the money transfer to the merchant (*Transfer*) in this case the bank account of the rattenfest organization. The customer gets a response (*Response*) that the payment was valid and has been processed successfully.

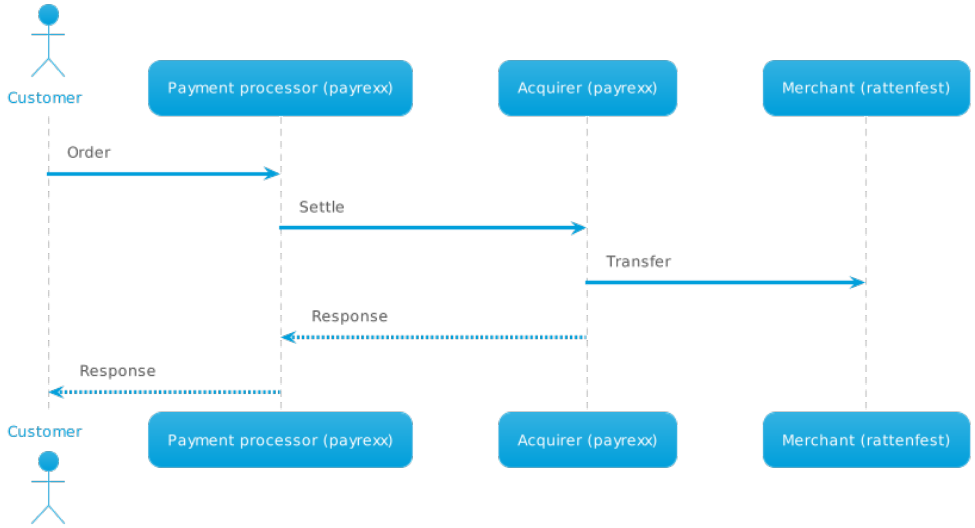



Figure 4.1: Payment process showing interactions between, customer, PSP, acquirer, and merchant

#### 4.1.4.2 Comparison

Table 4.4 shows a comparison between different payment service providers on criteria relevant to us. The selection of PSP orients itself on a blog article [Mic24]. It provides an overview of processors suitable for smaller businesses or use cases. This table is a mix of both publicly available data and offers that sales departments of the listed companies made to us. Data that differs from public available data is marked with an asterisk (*).

This combination of data makes the comparisons between the providers difficult because if you do contact sales, they might offer you better terms than what you would expect looking at their website. Some payment service providers have different fees for Twint and credit cards respectively. In the table you can see Twint labeled as (T) and credit cards as (C).

Here we show a brief description of the chosen PSP:

- **DataTrans**
  DataTrans is based in Switzerland and provides payment processing services. It supports credit cards, e-wallets, and direct debit payments. Its customers include medium to large enterprises operating in Switzerland and Europe.
- **Postfinance**
  Postfinance is a Swiss financial institution that provides banking and payment services. It processes domestic and international payments, including credit cards and electronic transfers. Its services are used by small, medium, and large businesses within Switzerland.
- **Payrexx**
  Payrexx is a Swiss cloud-based payment platform offering multi-currency payment processing and e-commerce integration. It supports credit cards, PayPal, and other online payment methods. Its customers are primarily small and medium-sized enterprises in Switzerland and Europe.
- **Stripe**
  Stripe is a US-based payment processor operating internationally. It supports credit cards, digital wallets, and local payment methods in multiple currencies. Its clients include startups, medium enterprises, and large corporations worldwide.

#### 4.1.4.3 Evaluation

In table 4.4 we are assuming that 2500 transactions will commence, since this is how many tickets are sold on average each year. Additionally, we are assuming that about 70% of transactions are done with Twint and the rest with credit cards. Since this amount can vary and the respective cost vary with it, we have plotted the total yearly cost in reference to the share of Twint transactions in figure 4.2. We are expecting that over half of transactions will happen with Twint since our customers are younger people, with some underage where they have access to twint but not to credit cards.

We have neither data from previous years on what payment methods were used nor is the main actor in Switzerland (Swiss Payment Monitor) studying this issue able to differentiate the used payment method in online shopping. [Swi25] To evaluate on whether our expectation on Twint usage is correct, we have concluded a survey of 10 people that either have visited the Rattenfest before or are the age of our target group. In the

Table 4.4: PSP Comparison at 2500 transactions
(T) stands for Twint, (C) stands for credit card

| **PSP** | **DataTrans** | **Postfinance** | **Payrexx** | **Stripe** |
|---|---|---|---|---|
| **Acquirer included** | YES* | YES | YES | YES |
| **Upfront cost** | YES* | YES | NO | NO |
| **Monthly cost [CHF]** | 0.00* | 14.90 | 15.00 | 0.00 |
| **Cost per Ticket sale** | 0.50 CHF + 0.50% (C) + 1.70% (T)* | 2.5% | 0.18 CHF + 1.65% (C) + 1.25% (T) | 0.30 CHF + 2.90% (C) + 1.90% (T) |
| **Total estimated Cost [CHF]**** | 2'237 | 1'941 | 1'486 | 2'125 |
| **Sources** | [dat25] | [pos25] | [pay25] | [str25] |

* Values come from sales departments of respective company.
** Assuming that 70% of total 2500 transactions are made with twint

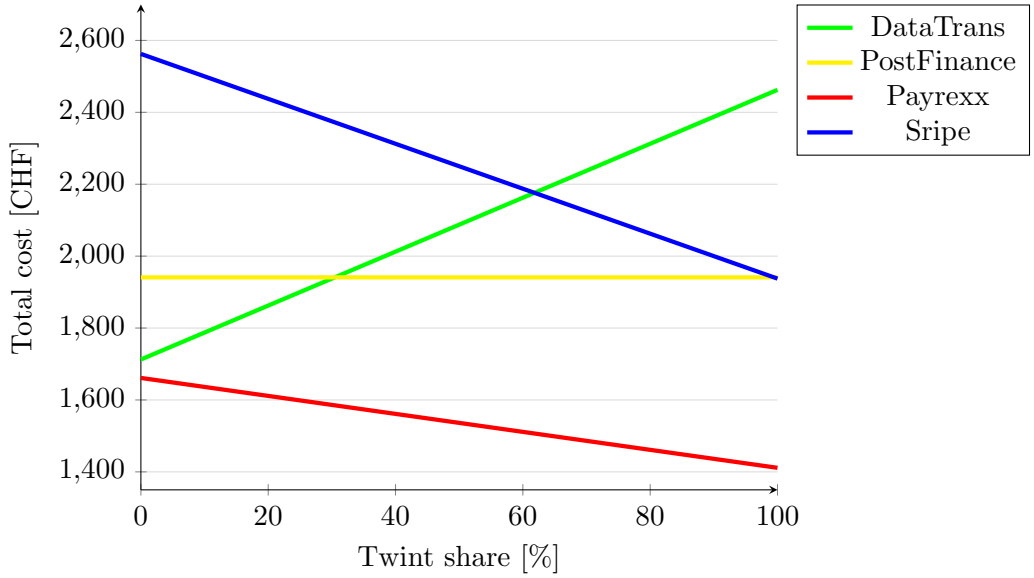



Figure 4.2: Total cost in relation to Twint transaction share at 2500 transaction

survey, 9 people answered that they would rather use Twint over credit card while one person prefers credit card over Twint.

This is supporting of our assumption that 70% of payments are done with Twint. Finally, we are concluding that Payrexx is the cheapest option to process our payments for our use case.

### 4.1.5 Integration and Deployment

As stated in Section 1.1, the project aims to provide basic configurability (NFR-06) while ensuring the system is secure, reliable, and redundant (NFR-03, NFR-07, NFR-08). Ticketing systems handle sensitive data, and downtime can cause financial losses. These goals conflict: simplicity often reduces robustness, and robustness can complicate management.

#### 4.1.5.1 Required Components

To deploy the RATTE system, we will assume that any organizations deploying the system has the following:

- Physical or virtual servers for hosting.
- A public domain for external access.
- An SMTP server for email notifications.
- Staff familiar with documented IT processes.

#### 4.1.5.2 Evaluation

Three of the most widely used [MD22] container orchestration platforms—Kubernetes, Docker Swarm, and OpenShift—were evaluated in Table 4.5 based on scalability, fault tolerance, ecosystem support, and automation capabilities. These factors directly impact the final product's reliability, maintainability, and performance.

| Metric | Kubernetes | Docker Swarm | OpenShift |
|---|---|---|---|
| Scalability | Medium–Large | Small–Medium | Small–Medium |
| Self-Healing | Yes | Limited | Yes |
| Maturity | Very mature | Mature but evolving | Evolving |
| Multi-Cloud Support | Native | Limited | Supported |
| GitOps Support | ArgoCD | No | Integrated |
| Cost | Free | Free | Paid License |

Table 4.5: Comparison of Orchestration Platforms (adapted from [MD22])

**Kubernetes** supports large-scale deployments with built-in automation and GitOps integration.
**Docker Swarm** is simpler but lacks advanced fault tolerance and GitOps features.
**OpenShift** builds upon Kubernetes, offering enterprise-grade security, monitoring, and integrated GitOps tools.

#### 4.1.5.3 Selection

Kubernetes was selected as it offers a better balance of scalability, automation, and ecosystem maturity, while maintaining a free and open source license.

### 4.1.6 Ticket Information Storage

Because our system will require storing ticket information, we will need a persistent storage system. Since the nature of the data which will be stored is private, for example customer information or ticket identification numbers, the storage needs to be able to guarantee security and redundancy.

#### 4.1.6.1 Selection

We have decided **to not include a persistent storage system in our final Product** due to the following reasons:

1. Implementing and providing a secure, redundant database in such a way that it can be used by event organizers with limited technical experience would in our estimation exceed the scope of this SA.

2. Failure to do so correctly will not only result in temporary downtime of the service, but in the permanent loss of transactions and tickets. Such a scenario would pose an existential risk to any event.

Because of this, users of the *RATTE* system are expected to host or provision such a storage solution themselves.
This means that event organizers will need to provide a database connection for the system to work. This could be a self-hosted database or a managed database service.

## 4.2 Application Architecture

This section describes the architecture and implementation of our solution. It also documents important design decisions we have made during the development process and the reasoning behind them.
To plan the architecture of our solution, we used the C4 Model[2]. By visualizing our software on different levels of hierarchical abstractions, the C4 model allows us to:

- Review and evaluate our approach
- Discuss and refine our ideas within the developer team
- Communicate our ideas with people outside the developer team (e.g. SA Advisor)

We have also structured our documentation according to the C4 model, first explaining the high-level architecture, then explaining the connections between the different components and finally documenting the implementation of each component.

[2]A lean graphical notation technique for modeling the architecture of software systems: http://c4model.com

### 4.2.1 System Context

The system context diagram (Figure 4.3) provides the big picture of the application and the environment it interacts with. In our case, we are expecting three different kinds of users to interact with the System:

- **The Event Organizer** Sets up and manages the *RATTE* System. They ensure the functionality of the platform and monitors ticket sales and other stats before, during and after the event. They may generate tickets for special guests.
- **The Event Worker:** Validates tickets of arriving Guests and triggers boxoffice sales.
- **The Event Guest:** Selects and buys the desired ticket, which they then uses to gain entry to the event.

To process the Payments, a Payment Provider is used. This is further described in Section 4.1.4. Also shown is the database, which is not part of the system as described in Section 4.1.6.

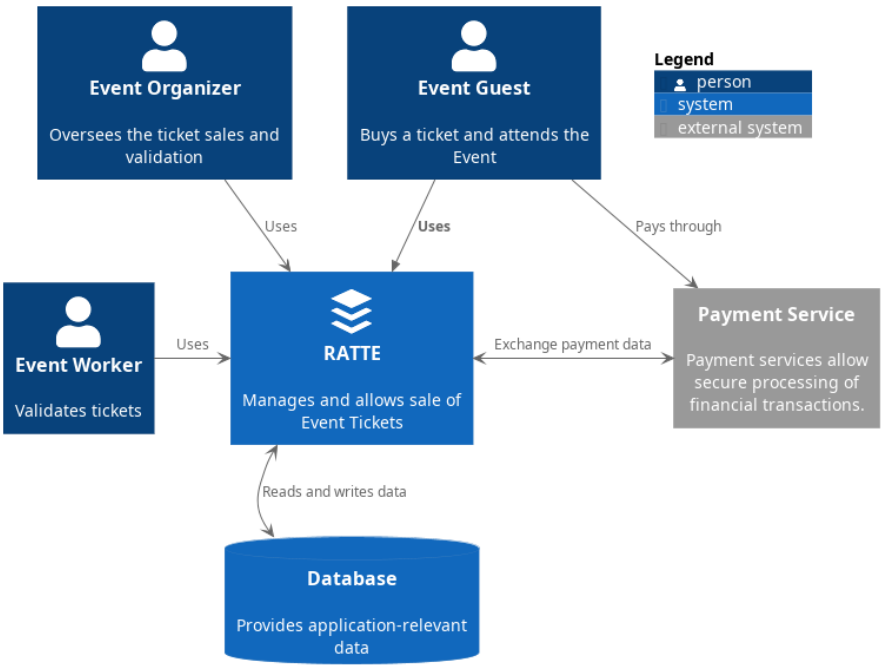



Figure 4.3: C4 System Context Diagram

### 4.2.2 Container Level

The container diagram 4.4 shows the general software architecture and major technology choices. In our case, it particularly highlights the four separate applications, which help

fulfill the needs for different types of users.
It also shows the communication between these applications: Ticket and User information is stored in a separate database, outside the *RATTE* system, managed by the event administrators.

#### 4.2.2.1 Administrative Dashboard

Provides event organizers with tools to manage tickets and monitor event statistics. The dashboard web application covers the functional requirements FR-15 to FR-34, except FR-26. It also covers the non-functional requirements NFR-02, NFR-03, NFR-04 and NFR-06.

#### 4.2.2.2 Ticket Scanner

Enables event workers to validate tickets at the entrance and manage box office sales. The ticket scanner applicatoin covers the functional requirements FR-12 to FR-14, FR-22, FR-23, FR-35 and FR-36. It also fulfills the non-functional requirements NFR-02, NFR-03, NFR-04 and NFR-06.

#### 4.2.2.3 Ticket Shop

Allows users to purchase tickets, recover lost tickets and redeem vouchers. The ticket shop web application covers the functional requirements FR-01 to FR-05, FR-07, FR-09 and FR-15. It also fulfills the non-functional requirements NFR-01 to NFR-06.

#### 4.2.2.4 Central Backend

Serves as the backbone of the system, providing RESTful APIs for all frontend applications and handling business logic. It is involved in providing functional requirements FR-01-17 and FR-24-37. It also covers NFR-01, NFR-02 and NFR-06-09 .

### 4.2.3 Component Level

The following sections each zoom in on one of the containers inside the RATTE system shown in the container diagram 4.4 and describe their internal architecture and some implementation details.

#### 4.2.3.1 Ticket shop

The Ticket shop is the public-facing interface for event guests, designed as a Single Page Application using React. As illustrated in Figure 4.5, the architecture separates the presentation logic (Pages) from the data access logic (Services).

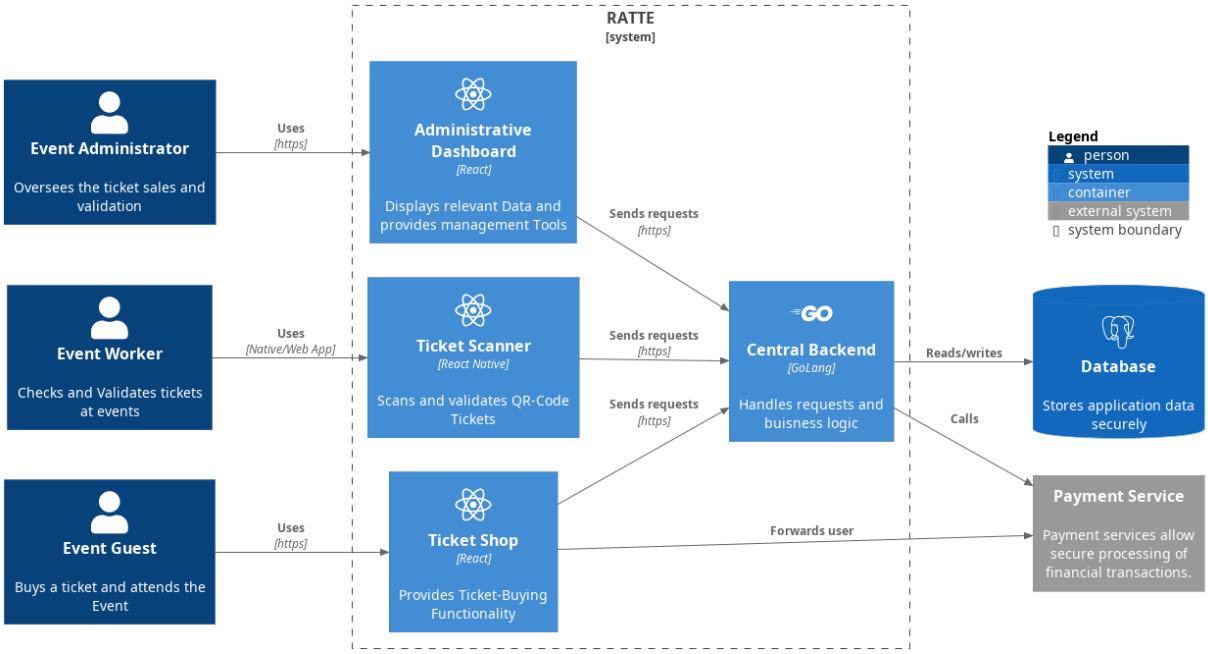



Figure 4.4: C4 Container Diagram

**Presentation Components** The user interface is structured around a set of page components, each responsible for a set of functional requirements:

- **Base Page:** This component acts as the base layout of the application. It implements the main layout structure, including the navigation header and the responsive container for content. It utilizes React Router's `Outlet` mechanism to dynamically render the appropriate child page based on the current URL, ensuring a seamless navigation experience without full page reloads.
- **Ticket Page:** This is the most complex component in the application, encapsulating the entire ticket purchasing workflow. It implements a multi-step pattern that guides the user through:
  1. **Selection:** Choosing the quantity of tickets (FR-07). This step is illustrated in Figure C.6.
  2. **Payment Method:** Selecting between Credit Card or TWINT (FR-03, FR-04). This step is illustrated in Figure C.7.
  3. **Personal Data:** Entering contact information with real-time validation (FR-01, FR-02, FR-05). This step is shown in Figures C.8 and C.9.
  4. **Submission:** Aggregating the state and initiating the purchase transaction.

  This component manages its own state and delegates the final transaction submission to the API Service described in the paragraph 4.2.3.1.

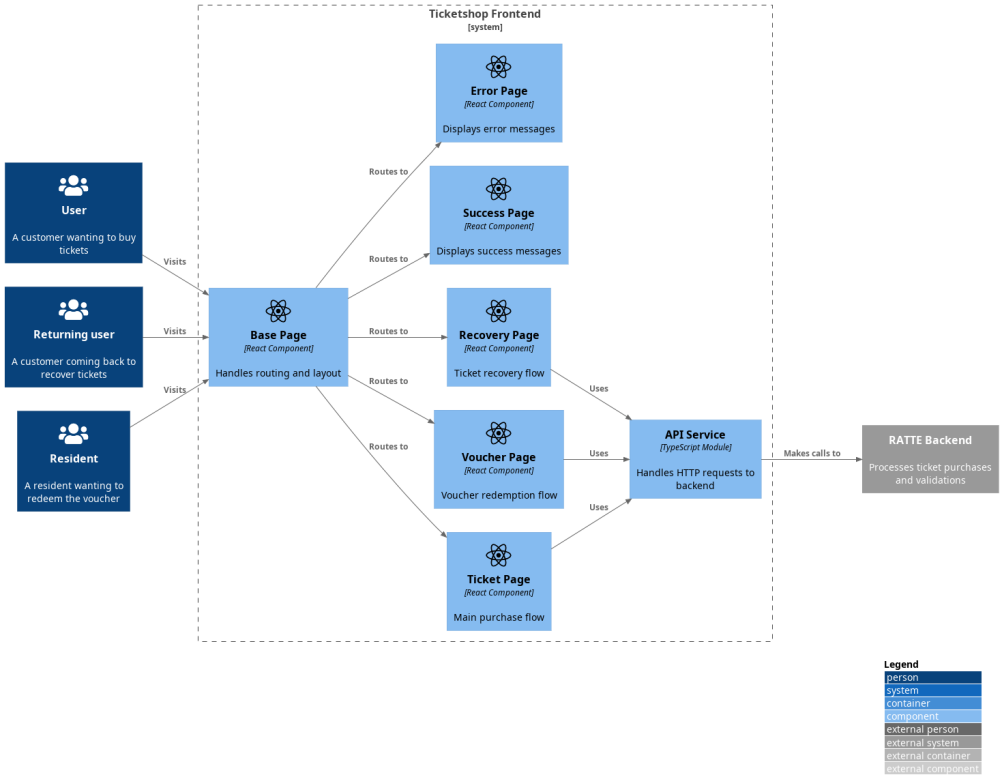



Figure 4.5: Ticket shop component diagram

- **Voucher Page:** This component handles the redemption of residential vouchers (FR-15). It parses the voucher UUID from the URL parameters to pre-fill the redemption form, providing a streamlined experience for users with valid codes. It interacts with the API Service to validate and redeem the code. It's illustrated in Figure C.12.
- **Recovery Page:** This component provides the functionality for users to recover lost tickets (FR-09). It presents a simple interface for users to input their email address (FR-02) and triggers the recovery process via the API Service. It's illustrated in Figure C.13
- **Feedback Components (Success/Error):** These are stateless components designed to display the final outcome of the payment process. They are the landing targets for the payment provider's redirects, parsing URL parameters to display the email. They are shown in Figures C.10 and C.11.

**Data Access Component** To decouple the UI from the backend implementation details, all network communication is encapsulated within a dedicated service component called API Service.

#### 4.2.3.2 Dashboard

The Dashboard is the administrative control center for event organizers. As an access restricted Single Page Application, it provides tools for monitoring sales and managing tickets. The figure 4.6 illustrates the architecture, which divides the application into presentation components and data access components.

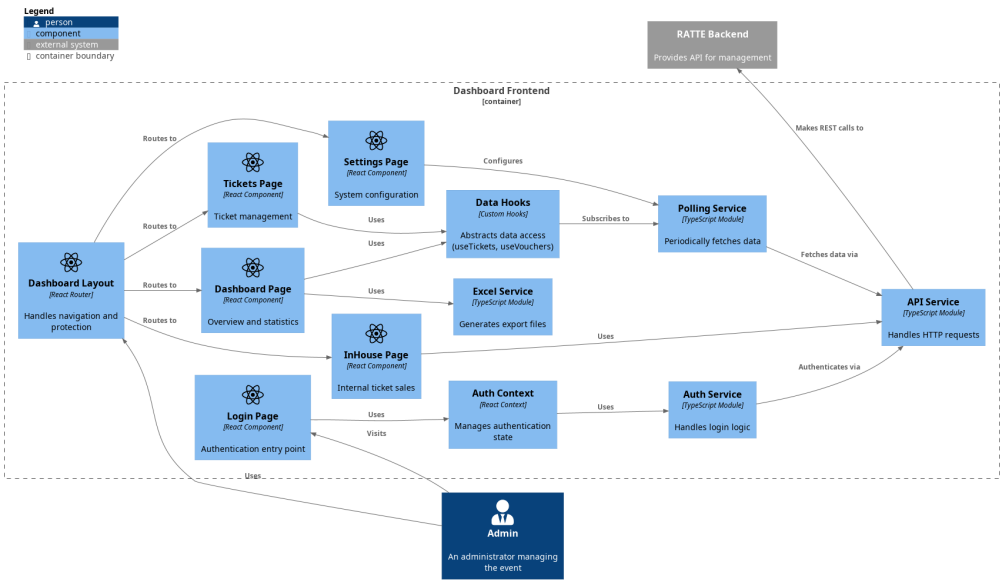



Figure 4.6: Dashboard component diagram

**Presentation Components** The user interface is built upon a secure layout structure that manages navigation and access control:

- **Dashboard Layout:** This component serves as the secure container for the application. It integrates the `Sidebar` for navigation and the `Header` for user actions. Most importantly, it wraps all protected routes in a `ProtectedRoute` component, which checks the `AuthContext` state and redirects unauthenticated users to the login page, ensuring that sensitive administrative functions are inaccessible to unauthorized users.

- **Login Page:** This component handles user authentication. It collects credentials and interacts with the Auth Service to establish a secure session. Upon successful login, it updates the global authentication state and redirects the user to the main dashboard. It is shown in Figure C.14.

- **Dashboard Page:** The landing page for authenticated admins. It aggregates data from the `useTickets` and `useVouchers` hooks to render interactive charts. It provides an overview of the sales (FR-16, FR-17, FR-19, FR-20), voucher usages

(FR-30) and ticket scans (FR-22, FR-23). This overview can be filtered by year (FR:18, FR:25).The dashboard also integrates the `Excel Service` to allow data export for external processing (FR-38). This is shown in Figure C.15.

- **Tickets Page:** A list view of all sold tickets. It allows admins to filter, search (FR-34), and delete tickets (FR-31, FR-32, FR-33). This is shown in Figure C.17.
- **InHouse Page:** Designed for back office operations. It provides functionality to generate and download bulk vouchers (FR-15) or send ticket emails (FR-27) to specific recipients. As shown in Figure C.16.
- **Settings Page:** Allows the user to configure the polling interval (FR-29) for data updates or enable and disable polling (FR-28). As shown in Figure C.18.

**Data Access Components** The dashboard employs a robust service layer to handle authentication and data synchronization:

- **Auth Context & Service:** The `AuthContext` provides a global state for the current user's session. It delegates the actual login logic and token management to the `AuthService`, ensuring that authentication concerns are isolated from UI components.
- **Polling Service & Data Hooks:** To provide near real-time updates without the overhead of WebSockets, the application uses a `PollingService`. This singleton service fetches the latest data at configurable intervals. Custom hooks (`useTickets`, `useVouchers`) subscribe to this service, abstracting the synchronization logic away from the components. This allows pages to simply receive the latest data state (FR-28).
- **API Service:** Similar to the Ticket shop, this module handles all HTTP communication. It includes interceptors to automatically attach the authentication token to every request, ensuring secure communication with the backend.
- **Excel Service:** A utility component responsible for transforming the JSON data sets of the tickets and vouchers into a downloadable Excel file, facilitating reporting and external data analysis (FR-38).

#### 4.2.3.3 Scanner

The Ticket Scanner is a mobile first application designed for event staff to validate tickets and manage on-site sales. Built with React Native and Expo, it runs natively on iOS and Android devices while maintaining a web-compatible fallback. Figure 4.7 depicts its architecture, which bridges React components with native device capabilities.

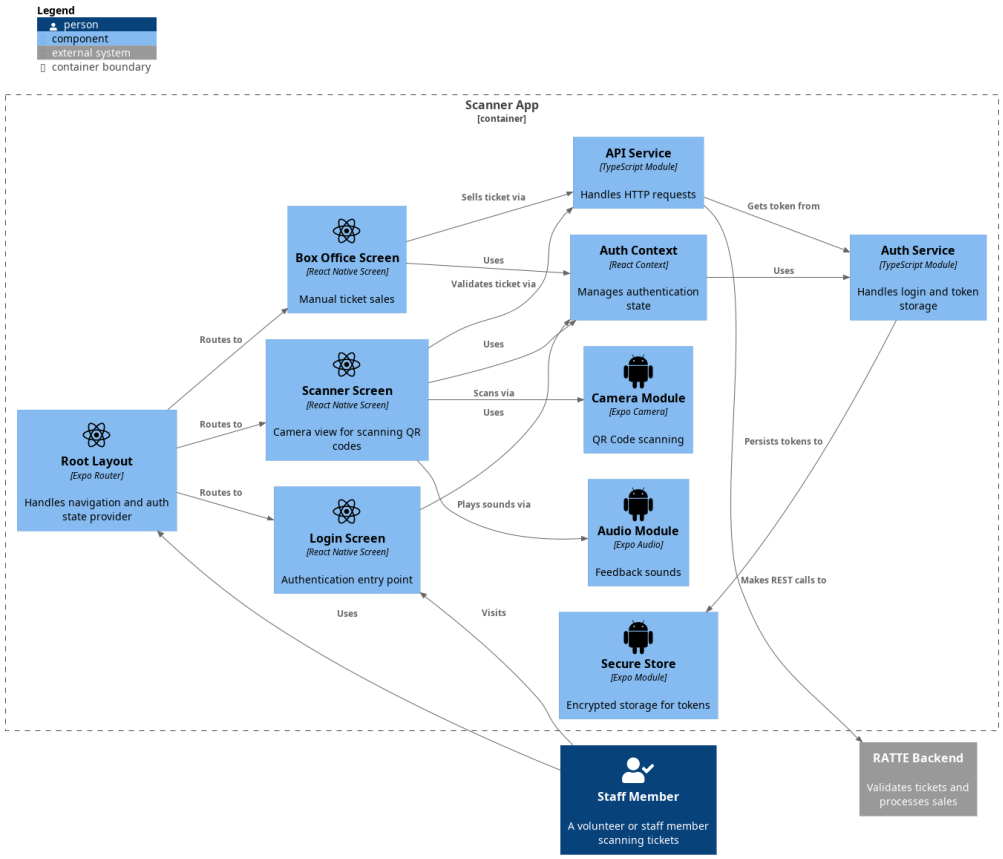



Figure 4.7: Scanner component diagram

**Presentation Components** The user interface is optimized for mobile usage, featuring large touch targets and built-in camera and flashlight controls.

- **Root Layout:** Utilizing Expo Router, this component defines the navigation. It acts as the application entry point, wrapping all screens in the `AuthContext` provider to ensure authentication state is globally available. It manages the transition between the public login screen and the protected application routes.
- **Login Screen:** The login screen handles credential input and communicates with the Auth Service to establish a secure session, which is needed for the rest of the pages. It is shown in Figure C.19.
- **Scanner Screen:** The scanner screen integrates the device's camera hardware via `expo-camera` to scan for QR codes. It implements a pausing logic to prevent duplicate scans while validating with the backend. It also controls the device's flashlight and triggers visual and audio feedback upon scan results (FR-12, FR-13). This is shown in Figure C.20. The various scan outcomes are handled as follows:

  - **Successful Scan:** On a valid ticket, the application provides positive feedback through a chime and the modal shown in Figure C.24.
  - **Failed Scan:** On an unrecognized QR code, the application provides negative feedback through a buzz and the modal shown in Figure C.22.
  - **Already Used Ticket:** If a ticket has already been scanned, the application provides warning feedback through a different buzz and the modal shown in Figure C.23.

- **Box Office Screen:** A dedicated interface for selling tickets at the door. It allows the staff to report exactly one ticket sale at a time with a confirmation modal to prevent accidental sales (FR-35). This is shown in Figure C.21.

**Infrastructure and Data Access** The application relies on the following services to handle security and backend communication:

- **Auth Service:** Security on mobile devices requires persistent storage beyond browser cookies. This service implements a hybrid strategy: it uses `expo-secure-store` to encrypt and store JWT tokens in the device's native Keychain/Keystore on iOS/Android, falling back to secure cookies when running in a web browser.
- **API Service:** A singleton wrapper for network requests. It implements an automatic token refresh mechanism. If a request fails with a 401 Unauthorized error, the service intercepts the response, uses the refresh token to obtain a new access token, and retries the original request transparently. This ensures that workers are not logged out in the middle of a scanning session.

#### 4.2.3.4 API Service

The React applications depend on an API endpoint that supplies data and allows the execution of various actions. This is achieved by using multiple HTTP REST endpoints provided by the backend. Because our focus is on resiliency (high uptime and low response times), we are using a **microservice architecture**, where tasks provided by the backend are split into multiple microservices. This is beneficial in multiple ways:

1. More granular control over horizontal scaling. Services with resource intensive jobs can be replicated in the cluster, distributing the load.
2. Contains failures to individual services, preventing cascading failures across the entire system.
3. Enables independent development and deployment of services, allowing more frequent updates.

In practice, we split our backend into five services:

- **Ticket Service:** Provides external endpoints to manage the creation of tickets.

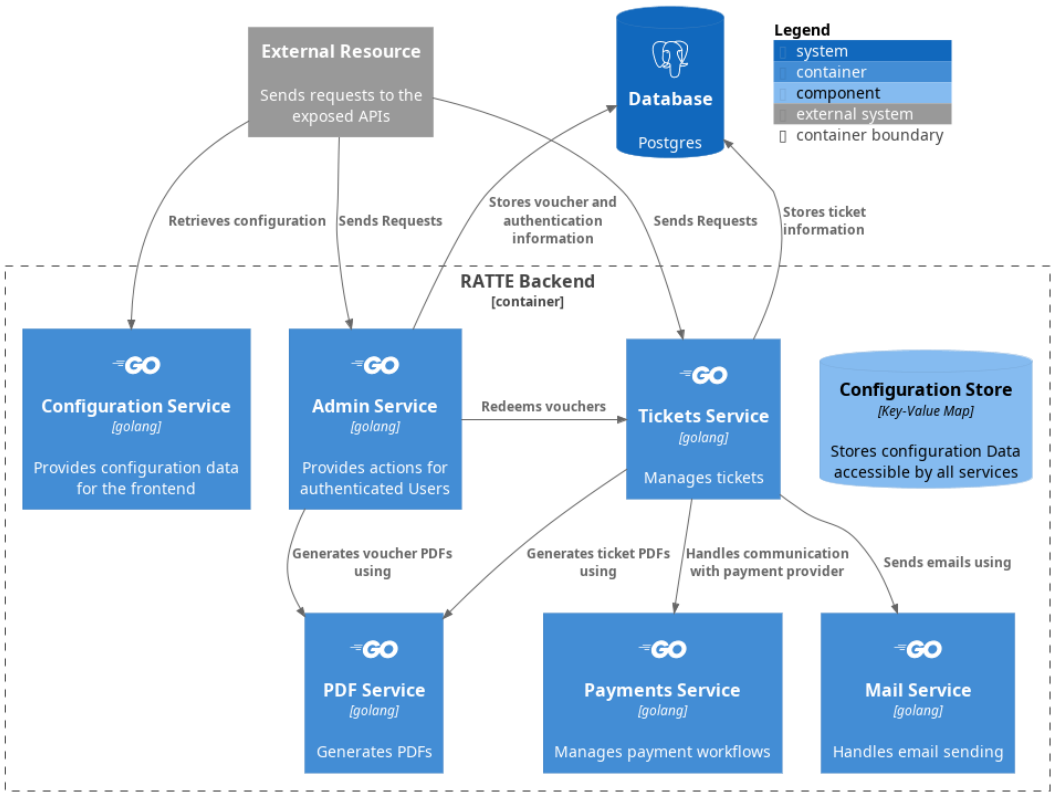



Figure 4.8: Backend Component Diagram

- **Admin Service:** Provides external endpoints for administrative actions which require authentication.
- **Payment Service:** Initializes payment sessions with the payment provider and returns the payment link.
- **PDF Service:** Creates PDF tickets and vouchers with QR Codes and other information.
- **Mail Service:** Sends emails with or without attachments using a provided SMTP server.
- **Config Service:** Provides configuration variables for frontend applications.

Communication within the cluster (between the microservices) is carried out by HTTP REST API calls.

**Ticket Service** The Tickets Service manages the creation and maintenance of tickets and vouchers. Every event has a hard limit on how many guests can attend from a

regulatory and safety standpoint (FR-11). The service ensures that the total amount of tickets issued does not exceed that limit (FR-08). This is the core business logic of the tickets service and is tracked via an entry in the `ticket_trackers` table, see Figure 4.11.
Additionally, it provides a ticket recovery mechanism and determination of the current ticket price.
Whenever a ticket is created, it receives a randomly generated UUID.

**Ticket maintenance** There are two main ways a customer can receive a ticket:

- The user redeems a voucher they received from the organizers
- The user buys a ticket in the ticket shop

Redeeming a voucher contains few steps. Assuming the user scans a valid QR-code of a voucher, the ticket shop will make a request to the tickets service and the service will then read how many tickets the voucher grants from the database. The service tries to create the amount of tickets when decrementing the available tickets in the `ticket_trackers` table. If the decrement fails, the event is sold out or does not hold as many available tickets as the voucher is valid for and a `soldOut` message is returned to the ticket shop. If successful, the tickets service calls the PDF service 4.2.3.4 to create the PDF file for each ticket and then calls the mailer service 4.2.3.4 to mail the files as an attachment to the email address of the person that redeemed the voucher. Finally, the voucher is updated as used in the database in the `vouchers` table as can be seen in figure 4.11.
Buying a ticket from the ticket shop, is a more complex workflow. You can see an overview of the process in Figure 4.9:

1. The ticket shop calls the tickets service via the `/buy` endpoint.
2. The tickets service creates a new transaction and adds a unique `reference_id` string as can be seen in the database model in figure 4.11. This helps identify the tickets with a payment the customer makes. The service tries to create the requested amount of tickets. If unsuccessful or the available tickets is smaller than the requested amount of tickets, then:
   (a) The transaction gets deleted.
   (b) The service returns a `soldOut` value to the ticket shop
   (c) The process is finished and no further action commences

   If successful and the available tickets is greater or equal the requested amount of tickets, then:
   (a) The Tickets service calls the payments service 4.2.3.4 to retrieve a redirect URL.
   (b) Returns the redirect URL to the ticket shop to which the user gets forwarded to proceed with the payment.

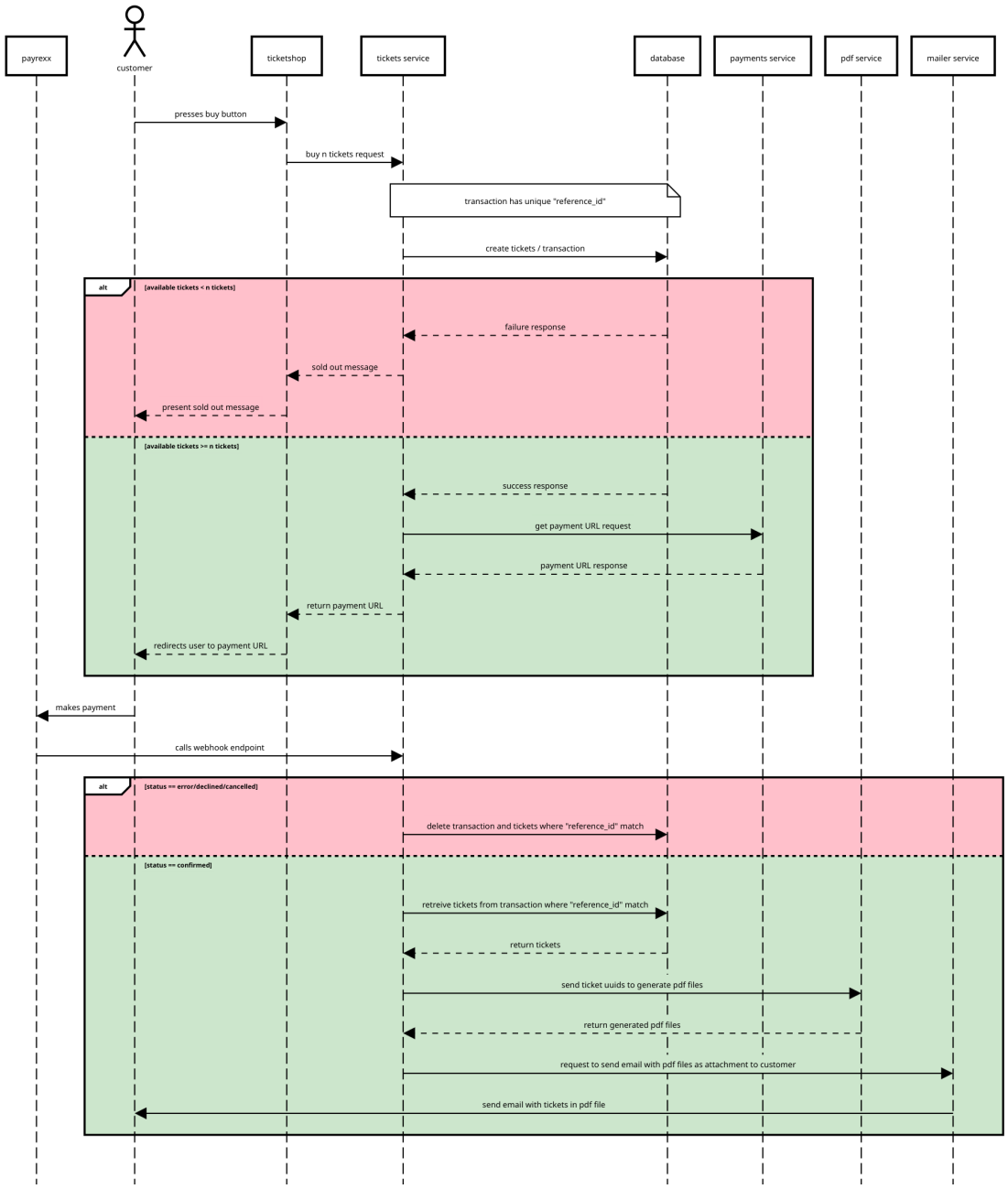



Figure 4.9: Sequence diagram workflow when the customer buys a ticket with the ticket shop

3. The service now waits for payrexx to call the webhook endpoint `/webhook/payment`. This endpoint is called by payrexx whenever a transaction by the user finished. When called, the payload of the request to the endpoint contains the `reference_id`

and the status of the payment. If the status is `confirmed`:

(a) The service retrieves the transaction where the `reference_id` matches from the database.

(b) The tickets, where the foreign key association to a transaction match that of the retrieved transaction, are retrieved from the database.

(c) The service now calls the PDF service 4.2.3.4 to generate a PDF file for each ticket.

(d) The generated PDF files are sent to the mailer service 4.2.3.4 to send the ticket PDFs to the customer via mail.

If the status is either `error`, `declined` or `cancelled`, the payment did not succeed, so the customer should not receive their tickets. The tickets must be deleted to make space for purchases by other customers:

(a) The service retrieves the transaction where the `reference_id` matches from the database.

(b) The tickets, where the foreign key association to a transaction match that of the retrieved transaction, are retrieved from the database.

(c) Both the tickets and the transaction are deleted from the database.

It is evident, that the tickets are created before the customer has made the payment. This is because if we were to create the tickets after customer **A** has paid money, it would be possible that another customer **B** buys the last remaining tickets while customer **A** is in the process of payment. The result would be that customer **A** has paid money while not receiving tickets since the event is sold out. Figure 4.10 depicts the process in a sequence diagram. To avoid this happening, the tickets are created before the customer reaches the payment site and are either issued to the customer on successful payment or deleted on unsuccessful payment. This doubles as a form of ticket reservation.

It is also important to note that the decrement in the remaining tickets field (table `ticket_trackers` in figure 4.11) are incremented and decremented atomically. This is necessary since the field is shared among multiple services and used in parallel, thus making sure no data races take place.

There is a possibility a user presses the buy button on the ticket shop, the tickets get created, but the user either never gets redirected to the website or takes too long to make the payment. In this case, tickets would be created, but they would never be sent to the customer. Neither would they be deleted because our webhook endpoint is never called. To solve this problem, the tickets service runs a “sweeper” in the background every 10 minutes. The sweeper retrieves all tickets and transactions that are older than 10 minutes and have not yet been confirmed. The retrieved tickets and transactions are then deleted from the database, making space for new ticket sales.

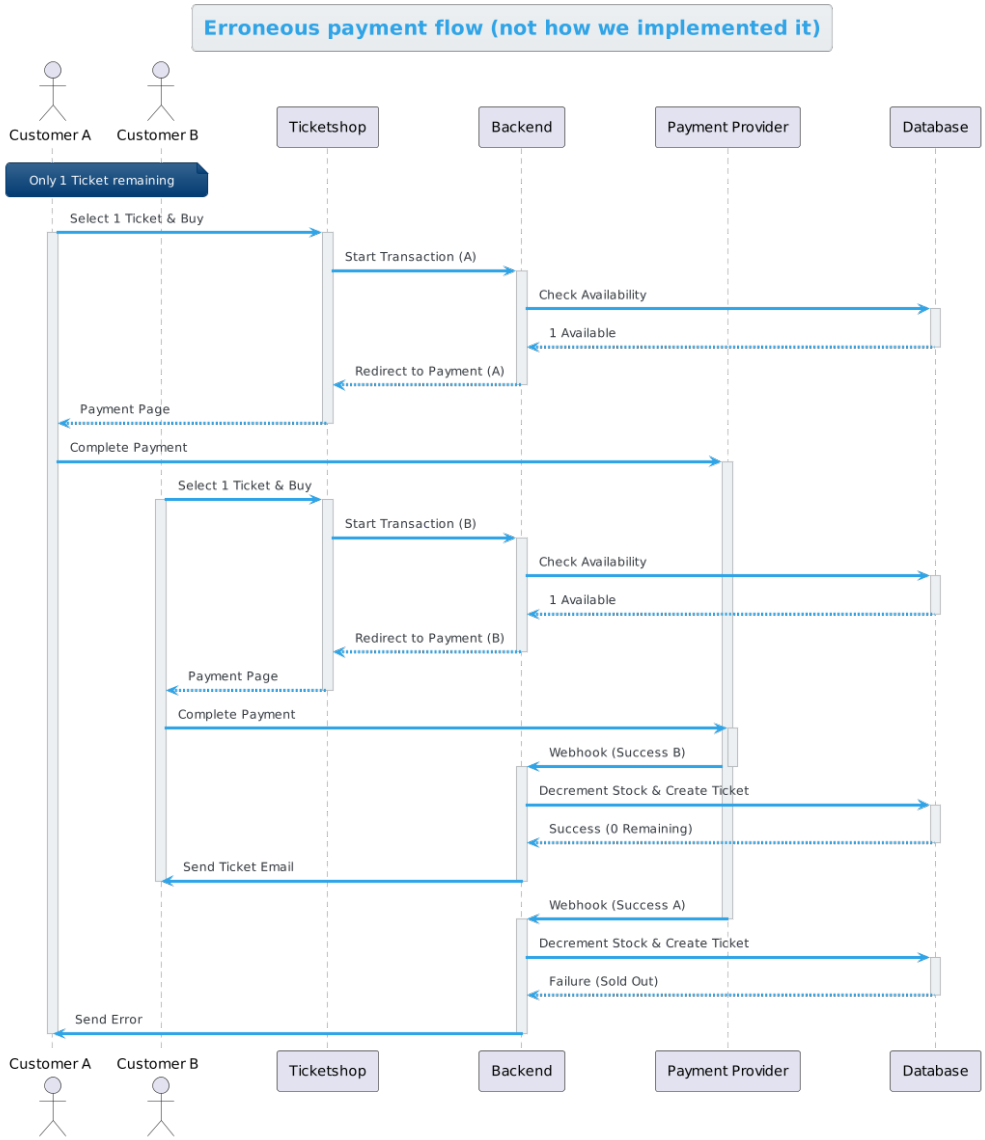



Figure 4.10: Sequence diagram workflow of a payment race condition

**Ticket recovery** When a customer loses the confirmation email that contains their tickets, they essentially lost their access to the tickets and therefore the event. To facilitate a recovery mechanism of the purchased tickets, the user can enter their email address in the ticket shop. The ticket shop then sends a request with the email as a parameter to the tickets service on the `/recovery` endpoint. In order to prevent abuse on the recovery mechanism such as sending hundreds of mails in few minutes, there is a `ticket_recoveries` table, as can be seen in figure 4.11, that tracks attempted recoveries. If a customer requests a ticket recovery, the service fetches the last recovery that matches the email address and checks whether a minute has passed or not. If a minute passed, the ticket PDFs are regenerated via the PDF service4.2.3.4 and then sent to the customers email address via the mailer service 4.2.3.4 (FR-09). When a recovery commenced to quickly, the message is returned to the ticket shop.

**Ticket price** The `/price` endpoint serves the ticket shop in fetching the current price of a ticket. The event organizer can set three ticket prices in cent (1/100 of a Swiss franc) value (FR-10):

- Early Bird
- Regular
- Boxoffice

Additionally, the event organizer sets two timestamps where the price switches from early bird to regular and from regular to boxoffice respectively. The service returns the appropriate ticket price based on which pricing period the current time falls into.

**Payments Service** The purpose of the payments service is to call the payrexx api and retrieve the redirect URL where the customer gets redirected. As you can see in figure 4.8, the payments service is not exposed to external services but is instead called by the tickets service 4.2.3.4. This decoupling gives us an additional safety barrier since the service holds the API-key to our payrexx account. When the `/creategateway` endpoint is called, the service makes a http request to `https://api.payrexx.com/v1.11/` with the necessary payloads like the `reference_id`, as mentioned in the tickets service 4.2.3.4, the payment amount, currency and payment methods. If successful, payrexx returns a json object containing the redirect URL. This url is then returned as a response.

**PDF Service** The PDF documents which are created by the RATTE application are either tickets or vouchers, each fulfilling different purposes. The service uses the *"go-pdf"* library, which provides methods to design and render text, images, and other graphical elements onto a PDF.

**PDF Tickets** The contents of the tickets follows the template of last year's Rattenfest tickets, which are standard "ticketcorner" tickets. Each ticket includes placeholders for two 4:3 aspect ratio sponsor images and a prominently displayed QR code

containing the ticket's UUID. An example ticket can be found in the Appendix, Figure C.25.

**PDF Vouchers** The vouchers feature a simplified design with explanatory text, a unique code, and a QR code. This QR code contains the full URL path, including the UUID parameter, to pre-fill the voucher code field on the ticket shop website, streamlining the redemption process. The designs for tickets and vouchers are intentionally distinct to prevent confusion, ensuring that event workers can easily differentiate between the two. While most content in the PDFs can be modified by directly altering the code, certain adjustable settings, such as URLs, are externalized in configuration files. This separation facilitates domain migration and other environment-specific adjustments. An example voucher can be found in the Appendix, Figure C.26.

**Mail Service** The Mail Service is responsible for delivering tickets to customers via email following a successful payment (NFR-05) or after they received a free ticket (NFR-27). These emails include one or more attachments: Typically the purchased tickets or received vouchers in PDF form. It accepts mail requests that specify the recipient's email address, subject line, email body, and any optional attachments. The service utilizes SMTP for email transmission. SMTP server credentials, such as login details and server configuration, are securely loaded from external configuration files. Once a mail request is received, the Mail Service establishes a connection to the SMTP server, transmits the email, and returns a success or failure response to the calling microservice, ensuring reliable and tracable communication.

**Admin Service** The Admin Service exposes endpoints for administrative operations such as:

- Querying ticket and voucher data (NFR-16-23)
- Deleting tickets (NFR-31-33)
- Generating new vouchers and complimentary tickets (NFR-15)
- Validating tickets during the event (NFR-12-14)

All endpoints are protected behind JWT token-based authentication. The JWT tokens are valid for a limited time period and must be included in the Authorization header of each request. After a token expires, a new one should be obtained automatically by using the refresh token. After the refresh token expires, the user must re-authenticate to obtain a new token.
The system enforces role-based access control through two user types. Admin users have complete access to all endpoints, while guest users are restricted basic actions like ticket validation or box office sale. This separation of concerns allows event organizers to delegate scanning responsibilities to staff without exposing sensitive administrative functions.

The service depends on other microservices and maintains its own database connection to provide the required functionality. As seen in the Backend Component Diagram in Section 4.8:

- It communicates with the Ticket Service to retrieve ticket and voucher data for reporting and monitoring.
- It interacts with the PDF Service to generate voucher PDFs.
- It relies on the Mail Service to send vouchers and tickets via email.

#### 4.2.3.5 Config Service

The Config service enables our frontend applications to be configured dynamically. When called at the `/` endpoint, it provides configuration variables such as the hosts base-url or logging levels.

#### 4.2.3.6 Database

The backend relies on the database for data persistence and consistency. An overview of the Tables and its relations can be seen in Figure 4.11.

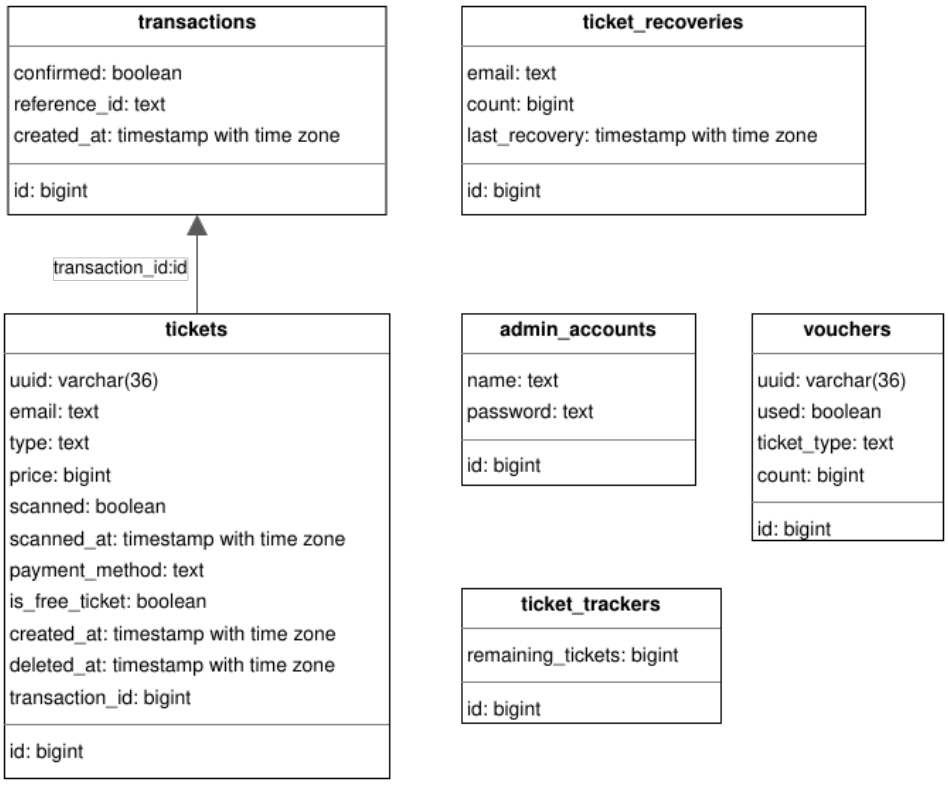



Figure 4.11: Database Diagram

The only relationship in the database is between the ticket and the transaction. The transaction represents the request when a customer buys a ticket from the ticket shop. The user enters the amount of tickets they wants to buy in one go and proceeds to the payment with payrexx, our payment provider. When the user finishes the payment, payrexx will call our webhook endpoint with the status or outcome of the payment together with a unique identifier that we passed when getting the payment-url from payrexx. The unique identifier allows us to associate a payment the user made with the tickets and the transaction that were generated before the payment finished. This unique identifier can be seen as the `reference_id` in the transactions table in figure 4.11.

We chose 3NF normal form for the relationship between tickets and transaction. This gives us less data redundancy and eliminates data manipulation anomalies while keeping data models simple relative to higher normal forms. [SKS20]
The `recoveries` table is used to store entries of attempted ticket recoveries, which resends all tickets the customer bought with their respective email-address. By checking when an actor last recovered their tickets, we can prevent abuse where a bad actor attempts to resend their tickets multiple times in a short time span, trying to overload the system.

The `admin_accounts` table stores the required accounts for accessing the admin service. It stores an admin user and a guest user with different permission levels respectively.

The `vouchers` table stores all vouchers that have been created and whether they have already been used or not.

Finally, the `ticket_trackers` table stores one entry, which is used to track the remaining tickets.

### 4.2.4 Code Level

The code level documentation C.6 dives in deeper into the architecture and implementation of each frontend application. Additionally it contains code diagrams illustrating the main components and their interactions within each application.

## 4.3 Deployment Architecture

The *RATTE* system does not only consist of the front- and backend-apps, but aims to be a complete package which can also deploy itself. The system is deployed at events with real-world ticket transactions and variable traffic over multiple months, which makes for a high-stakes environment. Because of this, focus was placed on making the deployment as reliable, scalable and user-friendly as possible.

Non-Functional Requirements NFR-06-09 specify requirements regarding ease of deployment, scalability and maintainability. To fulfill these requirements, we designed the deployment architecture with the goal of being easy to use for event staff with limited experience in deploying complex systems.
Because of this, we opted to use the Infrastructure-as-Code (IaC) approach: All system components and their relationships are defined in configuration files in human-readable format. These files are version-controlled in a GitLab repository[3], which can be forked and customized for different events. This approach provides versioned infrastructure history and provides a better overview for inexperienced maintainers.

### 4.3.1 Kubernetes & ArgoCD

Because we are working with Kubernetes (decided in Section 4.1.5.2), we are declaring the infrastructure in *YAML* files. We are using ArgoCD, which is a CI/CD tool for GitOps. ArgoCD continuously monitors the deployment repository and tries to match the state of the Kubernetes cluster to the desired state on the repository.
In the Kubernetes cluster, we require the following components to realize our requirements:

- Frontend and Backend images of the RATTE application. They are contained within a namespace, which is visualized in Figure C.1.
- *MetalLB* load balancer: Designed to load-balance (Layer 2) on bare-metal servers. This is required for self-hosting as it allows us to assign external IP addresses to services in the cluster.
- *nginx* ingress: Proxy server which handles incoming and outgoing traffic, provides rate-limiting for requests and load-balancing on the Application Layer (Layer 7).
- *Certificate Manager*: Manages certificates for HTTPS connections. Certificates are obtained using LetsEncrypt certificates with the HTTP-01 ACME challenge type.
- *Kubeseal* secrets manager: Handles encryption and decryption of secrets to ensure they can be checked into gitlab repository without leaking secret values.
- *Reloader*: Watches for changes in ConfigMaps and Secrets and restarts pods when changes are detected. This will automatically apply changes in the configuration.

### 4.3.2 Secret Management

In order to provide applications in the *RATTE* system with their required secret variables, we are utilizing Kubernetes Secrets. However, because we are managing our infrastructure through a potentially public repository, we cannot commit plaintext secret values. This is achieved using Kubeseal, which allows anyone with access to the cluster

[3] Repository hosted on OST GitLab, but can be forked to other software forges

to encrypt their secrets locally and then upload encrypted secrets to the repository as seen in Figure 4.12. Because this process requires some understanding of Kubernetes and Kubeseal, we have also developed a shell script[4], which allows users to upload their own secret values by simply inputting an .env file. This process is documented in the deployment *README.md*.

```yaml
---
apiVersion: bitnami.com/v1alpha1
kind: SealedSecret
metadata:
  name: backend-secret
  namespace: ratte
spec:
  encryptedData:
    ADMIN_USER_NAME: "example"
    ADMIN_USER_PASSWORD: "demo_password"
    PAYREXX_API_KEY: "some-api-key"
    REFRESH_TOKEN_SECRET: "some-secret"
  template:
    metadata:
      name: backend-secret
      namespace: ratte
```

(a) Unsealed secrets on a local dev machine

```yaml
---
apiVersion: bitnami.com/v1alpha1
kind: SealedSecret
metadata:
  name: backend-secret
  namespace: ratte
spec:
  encryptedData:
    ADMIN_USER_NAME: /L2tBgKMXgsriTqLSL5cJtM
    ADMIN_USER_PASSWORD: AgAdsnXmiDFjCMA83sI
    PAYREXX_API_KEY: AgCCsurR7pFcrLILp7bVXo+
    REFRESH_TOKEN_SECRET: AgAUlDi1cUo6+dU173
  template:
    metadata:
      name: backend-secret
      namespace: ratte
```

(b) Sealed secrets, ready to be committed to a repository

Figure 4.12: Secrets in a *YAML* file before and after encoding.

### 4.3.3 Compatibility of Deployment

To avoid any potential events having to restructure their existing IT infrastructure, we did neither want our solution to require a main domain (e.g. https://example.ch), nor did we want IT staff having to check for URL path conflicts with existing paths when integrating our system (e.g. https://example.ch/tickets). We therefore resorted to using only subdomains for our location. We are using the following five subdomains:

- *tickets*.example.ch for ticket buying.
- *scanner*.example.ch for the scanner web-app.
- *dashboard*.example.ch for admin dashboards.
- *api*.example.ch for the API.
- *argocd*.example.ch for the ArgoCD dashboard.

Additionally, we relay all calls to /config on each subdomain to the configuration API, so that frontend applications can easily retrieve configuration data such as API endpoints without hard-coding them. This has been visualized in 4.13.

[4] https://gitlab.ost.ch/ratte/deployment/-/tree/main/scripts

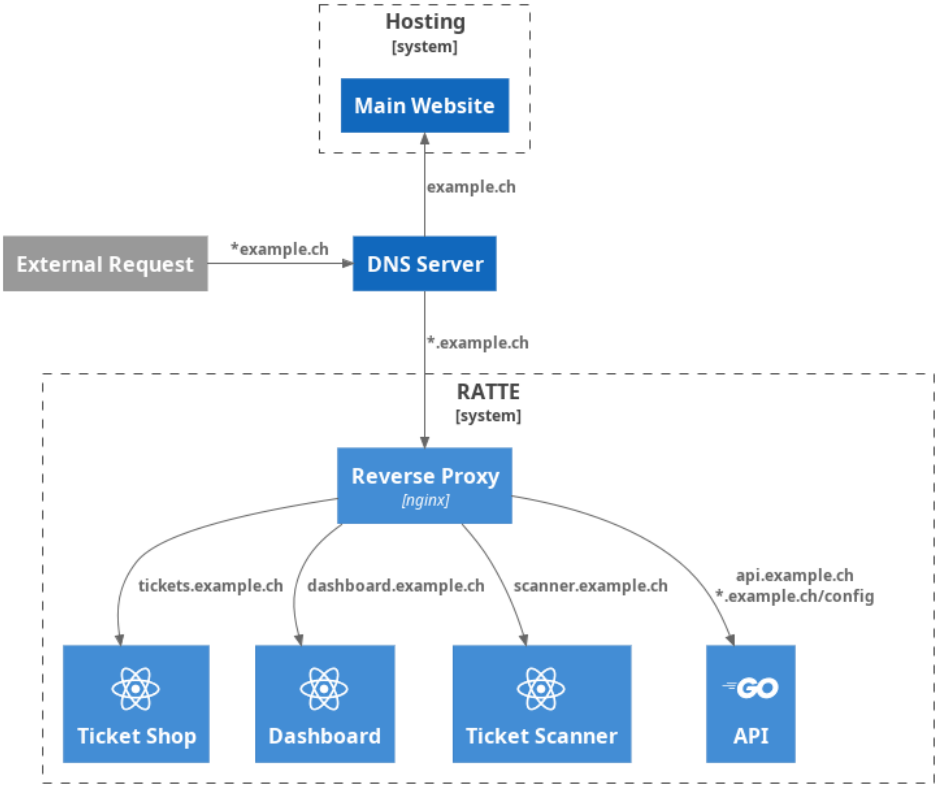



Figure 4.13: Visualization of DNS routing

### 4.3.4 Testing and Deployment Process

To ensure high code quality and prevent breaking changes, the development process employs separate staging and main pipelines. The staging pipeline includes unit test builds, deployment to a staging environment, and automated tests, whereas the main pipeline focuses solely on test builds, with deployments performed manually to ensure version stability.
The staging environment provides a full replica of the production deployment, accessible under subdomains such as *dashboard.staging.example.ch*, mirroring the real environment (e.g., *dashboard.example.ch*).
The development workflow follows these steps:

1. Ensure that the staging and main branch are in the same state.
2. Create a feature branch derived from the staging branch for implementing changes.
3. Develop and test locally using the provided *docker-compose* file to simulate the system.

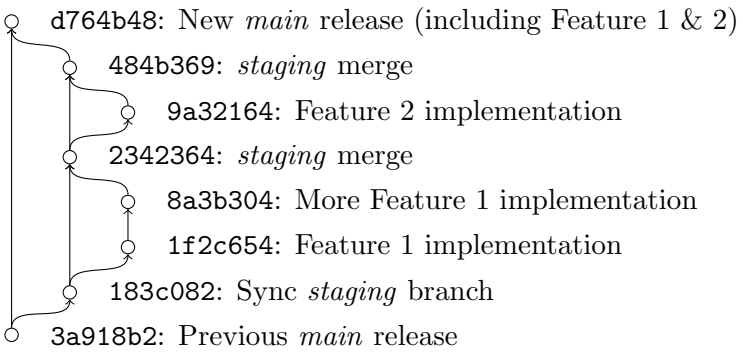



Figure 4.14: Example workflow of a new *main* release

4. Merge the feature into the staging branch. Automated and manual tests are conducted in the staging pipeline as described in Section 4.4.

5. Once validated, merge into the main branch. Images are rebuilt and version-tagged for deployment.

6. Use the version tag to update the production deployment. In production, the *:latest* should not be used due to poor auditability. Rather, a specific image tag should be specified.

An example workflow for a development of two features and their release is shown in Figure 4.14.

### 4.3.5 Setup & Bootstrapping

As defined in (FR-09), our goal was to make deployment on a bare-metal server as easy and well-documented as possible for event staff.

1. Installing *kubelet, kubectl* and other required software to run Kubernetes on one or more servers.

2. Running the necessary kubernetes resources to set up the *RATTE* system.

3. Making necessary configuration changes to the deployment repository.

The first two steps are documented step-by-step in the deployment repository README files. For running the kubernetes resources, we use bootstrapping, where the initial setup of the cluster is done using *kubectl apply* commands, before ArgoCD takes over and automatically deploys the required resources.

To change application configuration details such as API endpoints, credentials or ticket prices, the event staff only has to change configuration file values in the deployment

repository and push them to the git server. ArgoCD will then automatically detect and apply the changes to the cluster. For more complicated actions such as encrypting secrets or rotating passwords, we also provide scripts to make this process easier for users with limited kubernetes experience.

## 4.4 System Testing and Validation

To ensure the quality and security of our application and deployment, we have planned and implemented a variety of testing strategies. This chapter outlines them in detail.

### 4.4.1 Integration Testing

The backend microservices are written in Go, and we utilize the API testing framework *httpexpect* for integration tests. These tests validate the services by making HTTP requests to the API endpoints and verifying the responses as well as the state of the database and mail traffic. The goal is to test edge cases such as malformed inputs and transaction rollbacks to ensure the services behave correctly in all scenarios. The tests run in a containerized environment on the pipeline to ensure consistency and isolation.
The integration tests are executed automatically on every push to the staging branch of the repository using our CI/CD pipeline.

### 4.4.2 End-to-end Testing

We use Playwright for end-to-end testing to verify the system from a user's perspective. These tests run against the fully deployed application stack on the staging environment and simulate real user stories. Happy as well as unhappy paths are tested to ensure the system behaves correctly in all scenarios.
The tests get run automatically on every merge to the staging branch using our CI/CD pipeline and will wipe the database before running to ensure a consistent state.

### 4.4.3 Load Testing

To guarantee the system can handle high traffic, we use Artillery. We use it to run our existing Playwright scenarios at scale. This allows us to simulate hundreds of concurrent users performing complex actions like buying tickets, rather than just hitting API endpoints. By doing this we simulate real traffic during which we will manually test the application to see if it is still responsive and working as expected.
The load tests are executed manually before merging into the main branch to ensure the system can handle expected loads in production.

### 4.4.4 Security Testing

To improve the security posture of our system, we employ different security testing tools and practices.

#### 4.4.4.1 Vulnerability Scanning

Nuclei is a modern, fast, and customizable vulnerability scanner. Unlike traditional scanners which can be slow and closed-source, Nuclei is open-source and community-driven. It leverages simple YAML-based templates to define detection rules, allowing the global security community to collaborate on tackling trending attack vectors and newly released CVEs. [Prondb] We run Nuclei against our production environments to detect common vulnerabilities, misconfigurations, and exposed sensitive information. This helps us catch security issues early.
This is not a replacement for an actual pentester, but it allows us to create a baseline level of security.
The security scans are performed manually every week.

#### 4.4.4.2 Cluster Security

To improve cluster security, we use *Kubescape*, an open-source Kubernetes security scanner. It scans our Kubernetes clusters against various security frameworks. We specifically use the NSA framework, which scans for different best-practices within the cluster to ensure a secure configuration. A test result of such a test can be seen in the Appendix A.1.

### 4.4.5 Availability Monitoring

For continuous health checking, we utilize Uptime Robot. This external monitoring service pings our public endpoints (Ticket shop, Dashboard, Scanner, API) at regular intervals. It provides immediate alerts in case of downtime, ensuring we can respond quickly to any service interruptions which will be beneficial for the long-term maintenance of the system.

### 4.4.6 User Tests

#### 4.4.6.1 User testing for design

User testing was conducted during the early and middle development phaseses to gather feedback on usability and functionality. Participants from the target audience were invited to use the system and provide feedback on their experience. This feedback was valuable for the design process and helped us identify some features and requirements we had not considered before.

#### 4.4.6.2 User acceptance testing

In the final stage of development, we conducted user acceptance testing with the president of the Rattenfest committee to ensure the system met the specified requirements and was ready for deployment.

#### 4.4.6.3 Population

The table 4.6 shows the amount of users per test round and application.

Table 4.6: Amount of users per test round

| Test Round | Ticket shop | Scanner | Dashboard |
|---|---|---|---|
| 1 | 5 People, all of which have attended the Rattenfest before | 3 People, who have scanned tickets at the entry of the Rattenfest in the past | 2 People, one of which is the president of the Rattenfest and the other one is the cashier. |
| 2 | 4 People, all of which have attended the Rattenfest before | | The president of the rattenfest. |
| 3 | 1 Person, who has attended the Rattenfest before | | |

#### 4.4.6.4 Goals of the Tests

The table 4.7 shows the implied goals per round of test and application. These were told to the participants as mentioned in the testing routine.

#### 4.4.6.5 Testing routine

- Pre-test questionnaire to gather background information
- Explanation of the think-aloud concept
- Execution of the test
- Post-task questionnaire to gather feedback
- Final session to discuss the overall experience

#### 4.4.6.6 Ticket shop

**First Ticket shop test** During the first Ticket shop user test, participants were confused by the workflow and would have appreciated a progress bar or breadcrumbs. They were happy that they only needed to enter their email address when paying with TWINT and did not have to create an account. Some users suggested adding "-" and "+" buttons to select ticket amounts, as they have seen on Ticketcorner. Users liked the dynamically updating total price. They also liked the simple, straightforward visual

Table 4.7: Goals per round of test

| Test Round | Ticket shop | Scanner | Dashboard |
|---|---|---|---|
| 1 | Buy 3 Tickets for you and your friends using the payment method you prefer. | Decide whether these tickets are valid or not. | Log in and tell me stats about the ticketing sales. |
| 2 | Buy 3 Tickets for you and your friends using the payment method you prefer. | Boxoffice simulation where they had to report someone buying a ticket from boxoffice. For this they had to log in and navigate to the boxoffice subsection | Log in and print a voucher for 2 free tickets for someone living close by. |
| 3 | Get your tickets using a voucher you received in your mail | | Log in, take a look at the statistics, sort the tickets by scanned date, search for a ticket bought by doriano and delete it. Also enable automatic polling and set an interval. Send a ticket to themselves with a custom text. |

design. However, users pointed out the lack of animations on the website. Some users were confused about which fields were mandatory. From the first round of testing, the following changes were made:

- Added a progress bar to indicate the current step in the process.
- Added "-" and "+" buttons to select ticket amounts.
- Made mandatory fields more visually distinct.
- Added animations to enhance user experience.

**Second Ticket shop test** During the second round of user testing for the Ticket shop, participants suggested implementing a logical limit for selecting ticket amounts and setting one Ticket as the minimum value. One user was pleased that they could complete the form without having to use a mouse. Another user thought it was unclear when a step had been completed. Some users disliked the word wrapping in the progress bar. One user wished the back button was bigger on desktop. Some users wanted a logo to be present on the page. One user suggested adding a subtitle that says, "Get your tickets here exclusively." One user also noticed that the plus sign was not fully aligned inside the circle. From the second round of testing, the following changes were made:

- Implemented a logical limit for selecting ticket amounts and set one Ticket as the minimum value.
- Made the back button larger on desktop.
- Improved the alignment of the plus sign inside the circle.
- Added a logo to the page.
- Added a subtitle that says, "Get your tickets here exclusively."
- Improved the word wrapping in the progress bar.

**Third Ticket shop test** During the third round of testing for the Ticket shop, the goal was to test the voucher system. Since it starts by scanning a QR code, the Ticket shop will only be used on a mobile device. The participant appreciated that the voucher was prefilled in the form and that he only had to enter his email address. In contrast to previous feedback, the title and logo were criticized for taking up too much space of the screen.

- Create a header for more compact use on mobile devices. This shifts the focus from the logo and header text to the relevant content.

#### 4.4.6.7 Scanner App

**First scanner test** During the first user test of the scanner app, participants noted that the "Flash" button was not working on the web version. After further discussion, they agreed to remove it entirely. They appreciated the color scheme of the success and error messages. One user with a broken rear-facing camera suggested adding a button to switch between the front and rear cameras. Another user mentioned the "ribbit" sound on the Eventfrog app when a scan is successful and suggested adding something similar. Feedback also highlighted the need for a more visually appealing font, better use of white space, and a logo. From the first round of testing, the following changes were implemented:

- Removed the "Flash" button entirely from the web version.

- Added a button to switch between the front and rear cameras.
- Improved the font, white space, and added a logo.
- Added a sound effect for successful scans.

**Second scanner test** During the second round of testing for the scanner app, the participants were asked to complete a boxoffice simulation. They had to report someone buying a ticket at the box office. For this they used the new tab in the scanner app. They liked the design but would have liked the navigation to be visible even if the camera permission is not given. This request made sense as some people might not have to scan tickets at all if the rattenfest organizes their entry in different lanes, where one would be dedicated to boxoffice only. The icon to get to the boxoffice page was intuitive, but they were hesitant to click it at first. They appreciated the confirmation modal to avoid accidentally reporting a ticket sale.

- Write a text below the icon to make it more obvious.
- Make the navigation visible even if camera permission is not given.

#### 4.4.6.8 Admin Dashboard

**First dashboard test** During the first user test for the admin dashboard, participants noted that it looked like a trading dashboard due to the many stats. They were able to find all necessary information. One suggestion was to add a light mode for the application and some other customization options. They also found the background color to be too bright. Additionally, the line of the header was not fully aligned to the line of the sidebar. From the first round of testing, the following changes were made:

- Adjusted the background color to be less bright.
- Aligned the header line with the sidebar line.
- Added customization options for the application.

**Second dashboard test** In the second round of testing the Dashboard the goal was to print vouchers for free tickets. The president of the Rattenfest found the process to be straightforward and easy to follow. The only concern was the naming convention for the tab, "Tickets" was not immediately clear to him, so he suggested changing it to "InHouse Sales". He also suggested putting a function on that page to directly send tickets to an email address, this will be used for helpers of the rattenfest and for musical artists. He suggested removing an upper logical limit for the amount of vouchers that can be downloaded. He also asked if it would be possible to add a tab where you can see all the sold tickets in a list view, where you can also refund tickets if necessary and do some sorting. He also noticed a double scroll bar on the right side of the page on a certain resolution. Additionally, he suggested moving the + and - buttons for selecting vouchers and tickets per vouchers amounts to the right side of the input field.

- Renamed the "Tickets" tab to "InHouse Sales" for better clarity.
- Added a function to send tickets directly to an email address.
- Removed the upper logical limit for the amount of vouchers that can be downloaded.
- Added tab to view all tickets in a list and functionalities to sort, search and refund them.
- Fixed the double scroll bar issue on certain resolutions.

**Third dashboard test** In the third round of testing the Dashboard, the president and two members of the committee tested the functionality to email tickets directly. They also took a look at the different statistics that are shown on the dashboard. Additionally, they looked at the newly implemented ticket list, where they also refunded a ticket that was sent to doriano's email, so they had to search for the ticket first. When emailing the tickets they suggested being able to select the type of ticket that it is going to be instead of it being a separate type. They suggested to only allow refund / deletion of tickets that are not scanned yet, but to show the tickets even though they are deleted. They suggested doing that using a strike-through on the ticket entry. They liked the stats a lot but suggested using a different color palette for the pie charts, as the type chart especially has around 8 colors. They also tested the automatic polling and if they were able to intuitively enable and disable it. They suggested it should be automatically enabled but set to a reasonable interval like 30 seconds.

- Added the ability to select the type of ticket when emailing tickets.
- Implemented a strike-through feature for refunded or deleted tickets.
- Removed the option to delete tickets that were already scanned.
- Updated the color palette for pie charts to improve clarity.
- Set automatic polling to be enabled by default with a reasonable interval of 30 seconds.

# Chapter 5

# Evaluation and Results

This chapter evaluates the results of the project by comparing the implemented system against the defined functional and non-functional requirements.

## 5.1 Evaluation

### 5.1.1 Functional Requirements

Overall, we were able to fulfill almost all functional requirements we set out to achieve, defined in Section 1.2. The only unfulfilled functional requirement is FR-26 (Graphical comparison of yearly data). We set higher priority on other requirements, which is why this one was not implemented in the end. The comparison is still possible, but it requires manual comparison of the charts for each year. The actual comparison graph, meaning each graph showing multiple lines for different years all in one is not implemented. To further combat this, we implemented the requirement FR-38 (Download raw data as XLSX file) to be able to download the raw data as XLSX file, so the user can create any desired comparison graph using the spreadsheet data.

### 5.1.2 Non-Functional Requirements

To evaluate the non-functional requirements defined in Section 1.3, we performed the following tests:

- **NFR-01 (Uptime):** We have set up monitoring using UptimeRobot[1] to track the system's availability over longer periods. Because the NFRs specify target uptimes for the time period leading up and during the event, we cannot yet mark this as fulfilled. However, even during development, where breaking changes were frequent, we have been able to reach an uptime of 99.76%, so we are confident that we will be able to reach our set goal.

[1] https://uptimerobot.com/

- **NFR-02 (Stability under load):** We used artillery to create load on the server with up to 50 concurrent users performing buy requests on the ticket shop. While the test was running, we performed a manual buy request and measured the time the request needed for processing in a browser. This was performed 5 times both with load and without load on the server, to compare stability and performance. A comparison of load times can be seen in figure A.7. It is evident that loading the application with 50 concurrent users does not impact performance and stability in a measurable way.

- **NFR-03 (Cross-Browser Support):** The ticket sale and dashboard application work identically on all major browsers, as verified by end-to-end tests. The scanner application however does not work on Firefox due an incompatibility between the used camera library and the Firefox web engine. We have deemed this acceptable, as this scanner app is only used by internal staff, which will be informed of the limitation and will then use another browser.

- **NFR-04 (Website design):** We have determined that we have successfully matched the design of the rattenfest website in such a way that the RATTE applications can be associated directly with the rattenfest. This includes a matching colorscheme, logo, and button styling.

- **NFR-05 (Web Accessibility):** We evaluated the web accessibility of our applications using the WAVE tool[2]. All three applications did not contain any Errors, as seen in Figure A.6.

- **NFR-06 (Configurability):** We have determined that the system can be configured using environment variables and secrets. This allows for easy adaptation to different environments and requirements.

- **NFR-07 (Recoverability):** To simulate system failures, we simulated different outages by creating a kernel panic in Linux for different nodes and deleting pods in the kubernetes cluster. In all cases, the system was able to recover within 10 minutes. The logs of the test can be seen in Figure A.5

- **NFR-08 (Reliability):** We have evaluated that server health can be monitored using ArgoCD, which provides overviews of the components of the system, their status and the logs of applications, ingresses and other services.

- **NFR-09 (Ease of deployment):** We have determined that the system can be deployed by only following the steps outlined in the documentation. We have tested this by doing the deployment step-by-step on two virtual machines, following the documentation exactly.

[2]Web Accessibility Evaluation Tools (https://wave.webaim.org/)

## 5.2 Conclusion

Through user testing, detailed analysis, and evaluation of both functional and non-functional requirements, we observe that *RATTE* system the fulfills its intended purpose, while demonstrating a high degree of reliability and robustness. We therefore conclude that the system fulfills almost all the expected criteria and is equipped to handle the demands of the Rattenfest or any similar event effectively.

# Chapter 6

# Conclusion

This chapter provides a retrospective analysis of the project and results, along with recommendations for future work and potential improvements to the RATTE ticketing system.

## 6.1 Retrospective

Towards the end of this project, we have all compiled a list of our personal reflections on the project. The following is a summary of the insights which could be useful for future projects. A more complete and specific summary can be found in Appendix B.

From a **technical perspective**, investing time in comprehensive architecture and infrastructure planning early in the project proved highly beneficial. Specifically, implementing **CI/CD pipelines** for automated testing and deployment, adopting a **microservice architecture**, containerization with Kubernetes, and **external configuration management** through environment variables resulted in reduced technical debt and better long-term maintainability, which paid off at the later stages of the project.
However, a **more detailed application planning phase** focused on application architecture and data model design would have reduced the amount of refactoring required during development. Early investment in architecture and infrastructure planning, while requiring more upfront time, ultimately saved considerable effort during the implementation phase.

In general, we found that with increasing project complexity, the benefits of investing more time in the initial architecture and development processes become more apparent.

## 6.2 Recommendations and Future Works

In this section we outline recommendations for future work and potential improvements to the RATTE ticketing system.

The RATTE ticketing system is ready for deployment and use for the next Rattenfest. It has also laid a solid foundation for future enhancements and improvements that can be made by interested developers. However, to ensure a long-term usage and maintainability of the system, which does not require deep technical knowledge, we recommend creating a separate configuration website that allows event organizers to customize the application in an intuitive and user-friendly way without having to modify the source code directly. Areas for configuration and customization could include:

- Theming (colors, logos, etc.)
- Texts and labels in the applications
- Ticket types and pricing models
- Ticket design
- Email templates

Additionally, some features and improvements to the system in various areas that could be considered for future work are:

- **Database Redundancy:** Implementing a safe and redundant database setup would enhance data integrity and availability. This could involve setting up database replication or clustering to ensure that data is not lost in case of hardware failures.
- **Security Enhancements:** Hiring a professional penetration tester to conduct a thorough security assessment of the system would help identify and mitigate potential vulnerabilities we might have overlooked.
- **Multi-language Support:** Adding multi-language support could better accommodate the multilingual nature of Switzerland. For this, a language selection feature could be implemented in the user interfaces.
- **Time-boxed Scanner Logins:** Implementing time-boxed logins for the scanner application could enhance security by limiting access duration for temporary staff.
- **Role-based Account Management:** Implementing user account management with more granular roles would enhance security and facilitate other features like time-boxed scanner logins.

# Glossary

**bare-metal** A physical computer server, as opposed to a virtual server, like a Virtual Machine or Cloud Instance. 43, 45

**bootstrapping** Initial setup process of a system where a simple system is used to to automatically set up a more complex system. 46

**CI/CD** Software engineering approach in which code changes are automatically prepared for a release to production. 42

**CVE** Common Vulnerabilities and Exposures, a list of publicly disclosed information security vulnerabilities and exposures. 47

**GitOps** Operational model that uses Git repositories as the single source of truth for declarative infrastructure and applications. 43

**JWT** JSON Web Token: A compact way of securely transmitting information between parties as a JSON object. 40

**SMTP** Simple Mail Transfer Protocol is an internet standard for email transmission across IP networks. It defines how email messages are sent from a client to a mail server or between mail servers. 40

**UUID** Universally Unique IDentifier is a 128-bit value used to uniquely identify information across systems without requiring a central authority. The probability of generating a duplicate UUID is so low, it is considred negligible. 34

# Appendix A

# Testing Attachments

## A.1 Security

### A.1.1 Kubescape Tests

Framework scanned: NSA

| | |
|---|---|
| Controls | 25 |
| Passed | 14 |
| Failed | 6 |
| Action Required | 5 |

Failed resources by severity:

| | |
|---|---|
| Critical | 0 |
| High | 0 |
| Medium | 24 |
| Low | 7 |

| Severity | Control name | Failed resources | All Resources | Compliance score |
|---|---|---|---|---|
| Critical | API server insecure port is enabled | 0 | 0 | 100% |
| Critical | Disable anonymous access to Kubelet service | 0 | 0 | Action Required ** |
| Critical | Enforce Kubelet client TLS authentication | 0 | 0 | Action Required ** |
| High | Applications credentials in configuration files | 0 | 24 | 100% |
| High | Host PID/IPC privileges | 0 | 19 | 100% |
| High | HostNetwork access | 0 | 19 | 100% |
| High | Insecure capabilities | 0 | 19 | 100% |
| High | Privileged container | 0 | 19 | 100% |
| High | CVE-2021-25742-nginx-ingress-snippet-annotation-vu... | 0 | 0 | 100% |
| High | Ensure CPU limits are set | 0 | 19 | 100% |
| High | Ensure memory limits are set | 0 | 19 | 100% |
| Medium | Prevent containers from allowing command execution | 0 | 0 | 100% |
| Medium | Non-root containers | 7 | 19 | 63% |
| Medium | Allow privilege escalation | 7 | 19 | 63% |
| Medium | Ingress and Egress blocked | 1 | 37 | 97% |
| Medium | Automatic mapping of service account | 2 | 21 | 90% |
| Medium | Administrative Roles | 0 | 0 | 100% |
| Medium | Container hostPort | 0 | 19 | 100% |
| Medium | Cluster internal networking | 0 | 0 | 100% |
| Medium | Linux hardening | 7 | 19 | 63% |
| Medium | CVE-2021-25741 - Using symlink for arbitrary host ... | 0 | 0 | 100% |
| Medium | Secret/etcd encryption enabled | 0 | 0 | Action Required * |
| Medium | Audit logs enabled | 0 | 0 | Action Required * |
| Low | Immutable container filesystem | 7 | 19 | 63% |
| Low | PSP enabled | 0 | 0 | Action Required * |
| | Resource Summary | 9 | 44 | 73.62% |

* failed to get cloud provider, cluster: kubernetes-admin-kubernetes
** This control is scanned exclusively by the Kubescape operator, not the Kubescape CLI. Install the Kubescape operator: https://kubescape.io/docs/install-operator/.

Figure A.1: Kubescape Scan with the NSA framework

### A.1.2 HTTP Security Headers

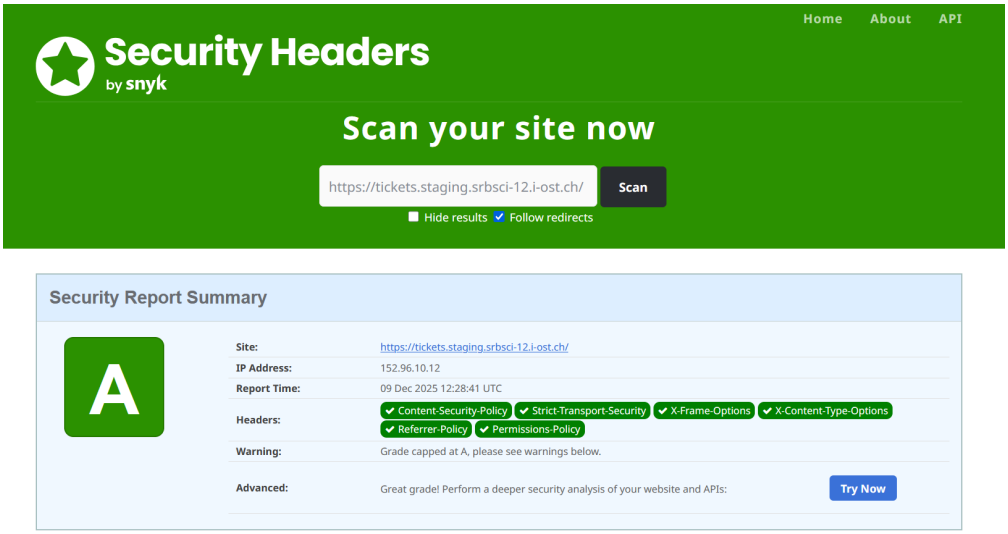



Figure A.2: securityheaders.com scan of the ticket shop application (staging environment)

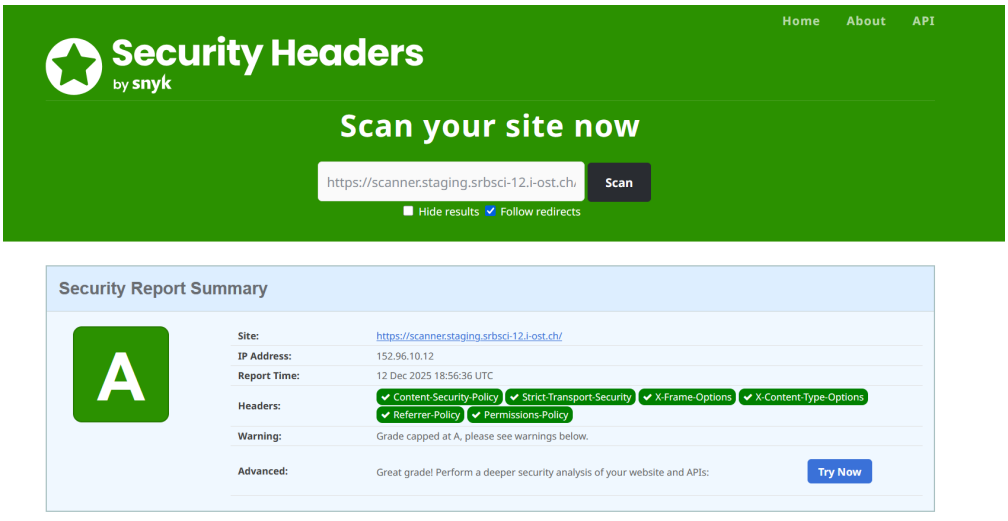



Figure A.3: securityheaders.com scan of the scanner application (staging environment)

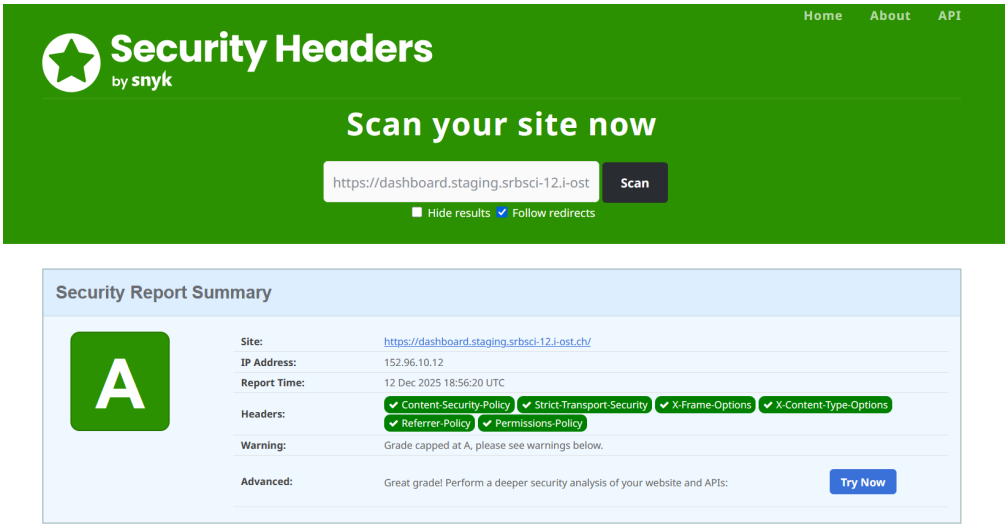


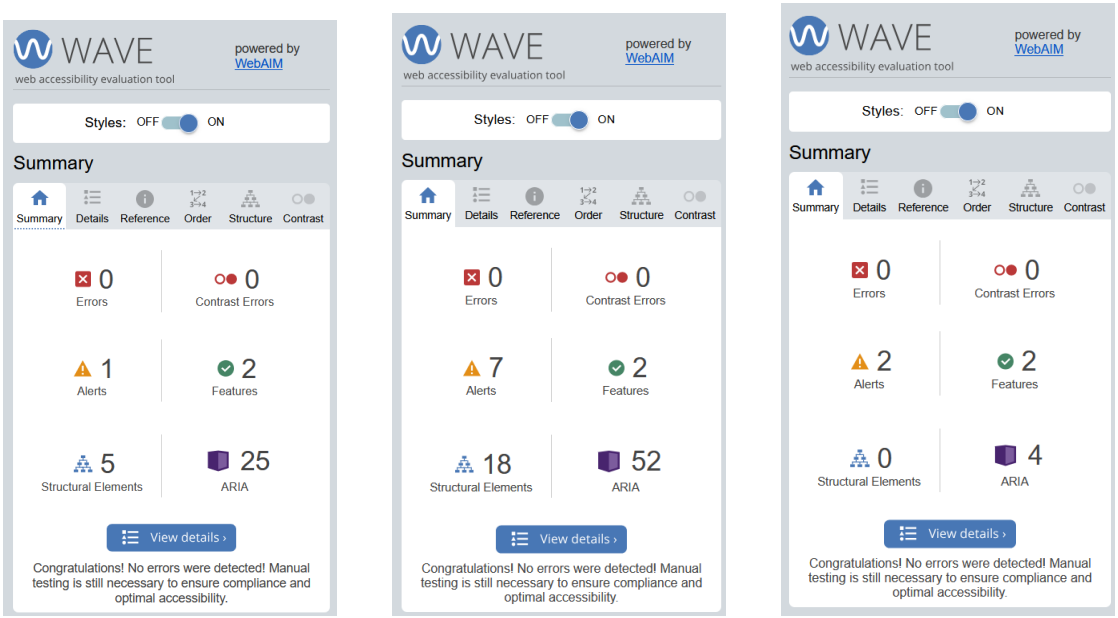

Figure A.4: securityheaders.com scan of the ticket shop application (staging environment)

### A.1.3 Crash Tests

```
fabio-arch@SpectreX360 ~/O/O/S/security [SIGINT]> ./measure-downtime.sh https://tickets.srbsci-12.i-ost.ch/ 1
=== Website Downtime Monitor ===
Monitoring: https://tickets.srbsci-12.i-ost.ch/
Check interval: 1 seconds
Timeout: 10 seconds

Waiting for website to be up before starting monitoring...
✓ Website is up. Starting downtime monitoring...

× Website is DOWN - monitoring downtime...
✓ Website is back up!
Downtime: 3m 39s

^C
fabio-arch@SpectreX360 ~/O/O/S/security [SIGINT]> ./measure-downtime.sh https://tickets.srbsci-12.i-ost.ch/ 1
=== Website Downtime Monitor ===
Monitoring: https://tickets.srbsci-12.i-ost.ch/
Check interval: 1 seconds
Timeout: 10 seconds

Waiting for website to be up before starting monitoring...
✓ Website is up. Starting downtime monitoring...

× Website is DOWN - monitoring downtime...
✓ Website is back up!
Downtime: 18s

× Website is DOWN - monitoring downtime...
✓ Website is back up!
Downtime: 2m 7s

× Website is DOWN - monitoring downtime...
✓ Website is back up!
Downtime: 1m 36s
```

Figure A.5: Server Recovery of times for (1) worker node and (2) control node outages

## A.2 Accessibility




(a) Ticket Shop (b) Dashboard (c) Scanner

Figure A.6: WAVE Reports of the three web-applications

## A.3 Performance

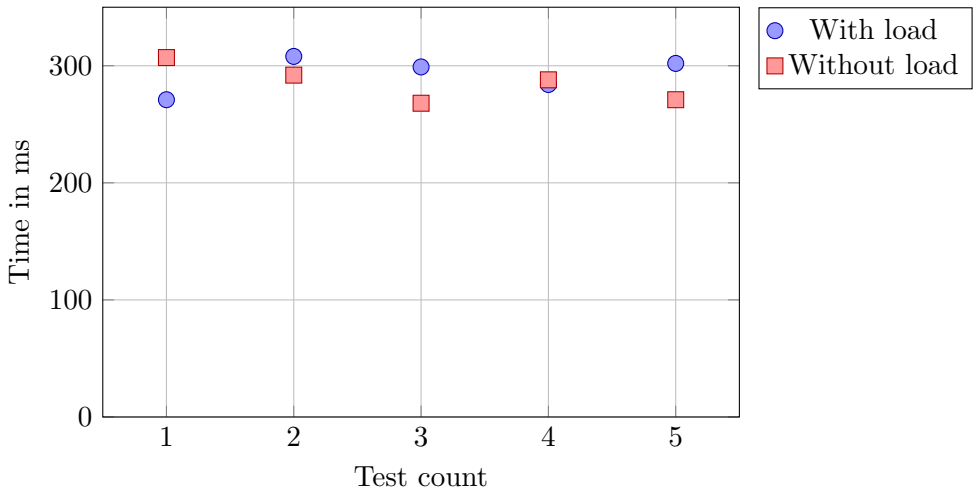



Figure A.7: Comparison of load times on buy requests with and without user load

# Appendix B

# Personal Retrospective

This is a summary of our personal reflections on the project.

## B.1 Team Dynamics

The team dynamics were excellent throughout the project. As we have been friends for a long time and have collaborated on numerous school assignments in the past, the forming, storming and norming phases of team development as described by Bruce Tuckman have all already occurred prior to this project. This allowed us to focus entirely on the performing phase of team development from the start of the project, with only some small refinements having to be made along the way. Because of this, we also benefited from established communication channels. This allowed for frequent, informal exchanges and rapid problem-solving without the friction often found in newly formed teams. We also set up regular meetings to discuss progress, these were usually not so important since we had constant communication, but they provided a good opportunity to align on goals and next steps.

The only downside to this close collaboration was the blurring of lines between work and leisure. Since we interact frequently in our free time, conversations often circled back to the project. While this showed our dedication, it sometimes made it difficult to mentally disconnect from the work, as the project became a dominant topic even outside of scheduled working hours.

## B.2 Technical Perspective

At the beginning of the project, while discussing the technology stack and architecture to use for this project, we were trying to find the right balance between planning ahead and avoiding an unnecessarily complex end-product. At the same time, we were aware that there was a real possibility that this project will end up being used with real financial transactions, which is why we would rather invest in comprehensive architecture and

security measures than risk building with insufficient reliability and security measures. During our previous comparable project, we suffered from a confusing and unorganized codebase as well as technical debt, which became most apparent during the final stages of the project. This time around, we decided to invest more time in the initial architecture and design phase, which, while taking up more time at the start, paid off in the long run.
Particularly, we have invested time in the following:

- CI/CD pipelines for automated testing and deployment
- A microservice architecture
- Containerization and orchestration with Kubernetes
- External configuration management with Environment Variables

We believe that getting these points right early in the project helped save time later on, and we recommend them for future projects of similar scope.
By using a technology stack of modern, but well established technologies (React, Go, PostgreSQL, Kubernetes), we were able to benefit from the latest features, while still having a large ecosystem to depend on and ensuring that future maintainers will be able to find help and documentation online. In our opinion, this increases the future maintainability of the project significantly.

## B.3 Planning and Time Management

Early in the project, we split up responsibilities for the components of the project, which meant that each group member was able to gain more in-depth knowledge of his field. This allowed us to work more independently, while always knowing who to approach for specific questions or need. This division of responsibilities worked well for us, and we would do it again in future projects.
Our approach to planning and time management prioritized early investment in architecture and infrastructure over rapid initial development. What we did not invest enough time into, however, was planning of the application details (e.g. Data Models, API Endpoints, User Flows): Throughout the development phase, we often reached the conclusion that refactoring was required to implement required features. This created resulted in wasted hours or suboptimal implementations. While some of this was unavoidable, we believe that a more detailed planning phase, including working out data models and endpoint names, would have reduced the amount of refactoring required later on and resulted in a better coordinated.

# Appendix C

# Additional Artifacts

## C.1 Links

- RATTE Application Repository: https://gitlab.ost.ch/ratte/application
- RATTE Deployment Repository: https://gitlab.ost.ch/ratte/deployment
- RATTE Documentation Repository: https://gitlab.ost.ch/ratte/documentation

## C.2 List of aids

Table C.1: Tools used during the project (Hilfsmittelverzeichnis)

| **Aufgabenbereich** | **Tools** |
|---|---|
| Literatur-Recherche und Verwaltung | Google, Google Scholar, IEEE Library, Springer |
| Datenanalyse und Visualisierung | ChatGPT, Latex |
| Ideengenerierung | ChatGPT, Claude |
| Übersetzung | DeepL, Leo |
| Coding | Visual Studio Code, Neovim, ChatGPT, Copilot, React, React Native, JetBrains GoLand, Insomnium |
| Texterstellung, Textoptimierung, Rechtschreibe- und Grammatikprüfung | cspell, Gemini, LTex |
| Zusammenarbeit und Projektmanagement | Outlook, Teams, GitLab, YouTrack, Discord, Threema |
| DevOps | Docker, GitLab, Kubernetes, ArgoCD |

## C.3 User test questions

### C.3.1 General questions

These questions were asked to all users in each user test, regardless of the application they were testing.

- How often do you attend festivals / events that require tickets per year?
- How was your experience with the ticketing process at the rattenfest.
- Which application do you like best for buying tickets online?
- Which payment method do you prefer when buying tickets online?

And post task questions

- What was clear?
- What was unclear?
- What did you like?
- What did you dislike?
- How would you improve the application / process?
- Did you encounter any unexpected behavior?

### C.3.2 Ticket shop questions

These questions were specifically asked to users testing the Ticket shop application, split into the test rounds.
Round 1:

- Was the workflow clear?
- What did you like about the visual design?
- What did you dislike about the visual design?
- Would you have liked to create an account?

Round 2:

- Was it clear which fields were mandatory?
- Did you like the animations?
- Was the progress bar helpful?
- Did you always know which step you were in and how many were left?
- Would you have preferred to select ticket amount with an input field?

### C.3.3 Scanner questions

These questions were specifically asked to users testing the Scanner application, split into the test rounds.
Round 1:

- Was it clear how to use the application?

Round 2:

- Where would you place the camera feed on the screen?
- Do you think the title is necessary?

## C.4 Architecture

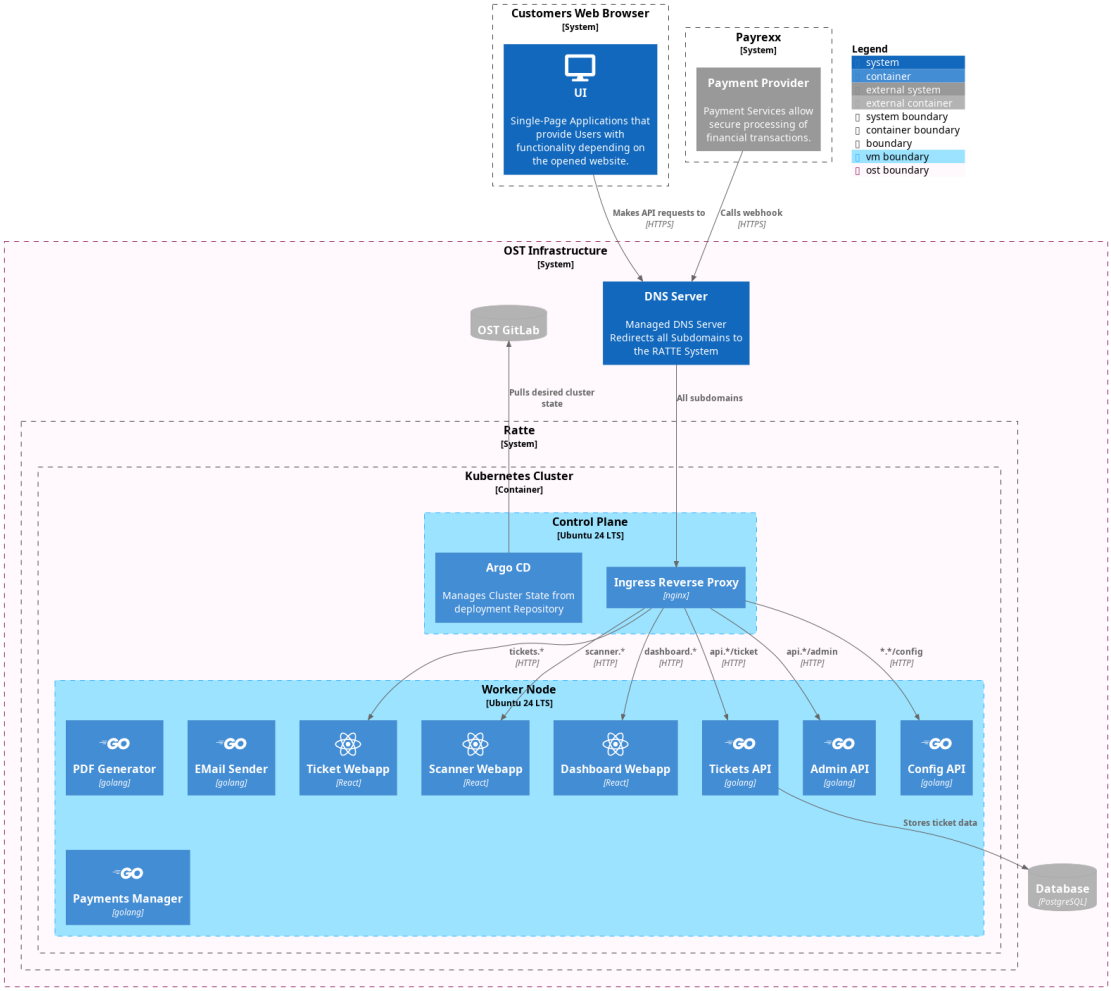



Figure C.1: C4 Deployment Diagram

## C.5 Long term project plan

We split the planning into the 5 predefined Milestones:

1. **M1: Project Setup and Requirements:** 30.09.2025

   Tooling is set up with automated pipelines for project building. Initial requirements, functional and non-functional, are set that define the project in an overview. Risk analysis covers all major risk factors

2. **M2: Domain model and Cluster:** 14.10.2025

   Refinement of requirements. Domain model represents the problem domain and is syntactically correct. Wireframes are done. A bare-bone kubernetes cluster is running.

3. **M3: First running Prototype Ticket Shop:** 04.11.2025

   The ticket sale shop is in functional with MVP requirements implemented.

4. **M4: First running Prototype Scanning App:** 18.11.2025

   The ticket scanning app is functional with all MVP requirements implemented.

5. **M5: Final Architecture and Documentation:** 21.12.2025

   The ticket shop, scanning app and organizer app have all MVP and Should-Have requirements met. The shop is running in a kubernetes cluster with loadbalancing.

Figure C.2 shows the plan in a gantt-chart.

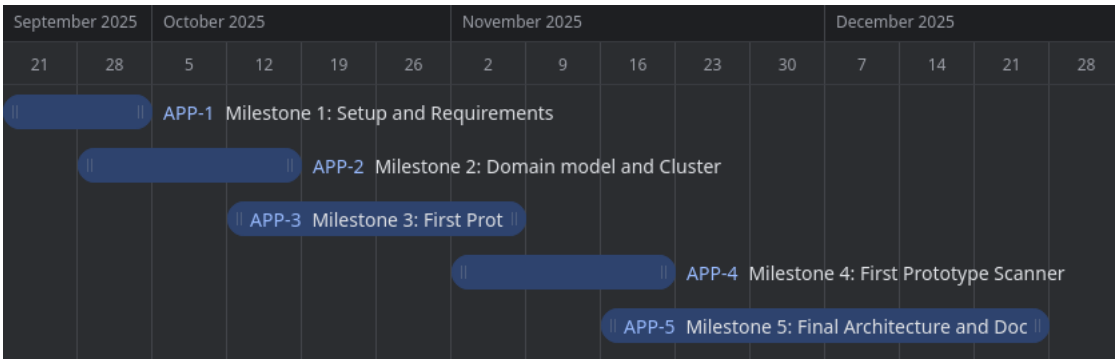



Figure C.2: Initial Gantt-Chart

## C.6 Code Level Documentation

### C.6.1 Ticket shop

The following sub section describes the architecture and implementation of the Ticket shop application which is illustrated in the corresponding Figure C.3. The Ticket shop is the public facing application of the system, designed to be the primary entry point for event guests. It is built as a Single Page Application using React, bundled with Vite for optimal performance (FR-01, FR-03). How the routing and the services are structured is explained in the following paragraph C.6.1.1.

#### C.6.1.1 Pages and routing

Starting off by explaining the pages and its routing structure, as seen in the blue package of the figure C.3. The application is structured around a central layout component, BasePage, which provides a header with navigation and the logo. The sub-pages are rendered inside the BasePage using the Outlet component from React Router. This enables smooth client side refreshing without full page reloads, the URL updates accordingly to reflect the current page. The main routes and corresponding pages are:

- /: The main ticket purchase page, the details of this as well as the illustrations are further explained in paragraph C.6.1.3.
- **/voucher**: A dedicated page for redeeming voucher codes. The voucher code is passed as a URL parameter and pre-fills the redemption form as seen in Figure C.12.
- **/recovery**: A page for lost ticket recovery as illustrated in Figure C.13.
- **/success**: Feedback page for successful transactions where payrexx redirects to. Takes in email as a URL parameter to display confirmation message as seen in Figure C.10.
- **/error**: Feedback page for failed transactions where payrexx redirects to as shown in Figure C.11.

These pages all interact with the backend API to perform their respective functions, which is why we created a dedicated API service as described in the next paragraph.

#### C.6.1.2 Services

The only service we implemented in this application is the aforementioned API service, illustrated in the orange package of the figure C.3. This service is a singleton, meaning it is instantiated once and shared across the entire application to ensure consistent state and configuration. It uses Axios as the HTTP client for making requests to the backend API. We chose Axios over the native Fetch API for several reasons outlined in [Log25], a small summary of the key benefits we found useful for our application are:

- **Automatic JSON Transformation:** Axios automatically stringifies request bodies and parses JSON responses, reducing boilerplate code.
- **Better Error Handling:** Unlike Fetch, Axios automatically rejects the promise for HTTP error statuses (e.g., 4xx, 5xx), simplifying error catching in our UI components.
- **Type Safety:** Axios supports generic types for responses, allowing us to strictly type the return values of our API calls to predefined models.

The models to ensure the mentioned type safety are the following, these are also illustrated in the grey package of the figure C.3:

- **PurchaseData**: Defines the payload for buying tickets, including `ticket_count`, `email`, and payment method flags.
- **VoucherRequest**: Used for voucher redemption, requiring an `email` and the voucher `uuid`.
- **GetTicketPriceResponse**: Allows the frontend to dynamically fetch pricing from the backend.

#### C.6.1.3 Ticketpage Implementation

The core functionality lies within the TicketPage. It implements a multi-step process to guide users through the purchase process intuitively:

1. **Ticket amount selection**: Users choose the quantity of tickets. The price is fetched dynamically from the backend on mount to display the correct pricing and total. This component is shown in Figure C.6.
2. **Payment method selection**: Users select their preferred payment method (Credit Card or TWINT). This step is needed before entering personal information to avoid unnecessary data entry (FR-02), in case of TWINT payment we only require the email, for the credit card we also need the first and last name as well as the phone number. This component is shown in Figure C.7.
3. **Personal Information**: Users enter their contact details. This includes a confirmation email field, we intentionally make the user enter the email twice, as the system is not account based and ticket recovery will happen over email only (FR-01). We implemented custom validation logic here, including a check to ensure the email and confirmation email match. The check provides delayed feedback after end of input for a wrong email but immediate positive feedback on correct entry to ensure the best user experience. This component is shown in Figures C.8 and C.9.
4. **Payment**: This step redirects the user to the payment provider payrexx to complete the transaction.

5. **Confirmation**: This step gets shown in the breadcrumbs but is never actually displayed. It helps the user as a visual guide to know that the purchase process will be completed after the payment.

State management within this component handles the transition between steps and aggregates the data required for the final API call called buyTickets in figure C.3 to the backend using the apiService. As you can see in the figure, the model PurchaseData is used to strictly type the payload of the request. The other single purpose pages are implemented straightforwardly.

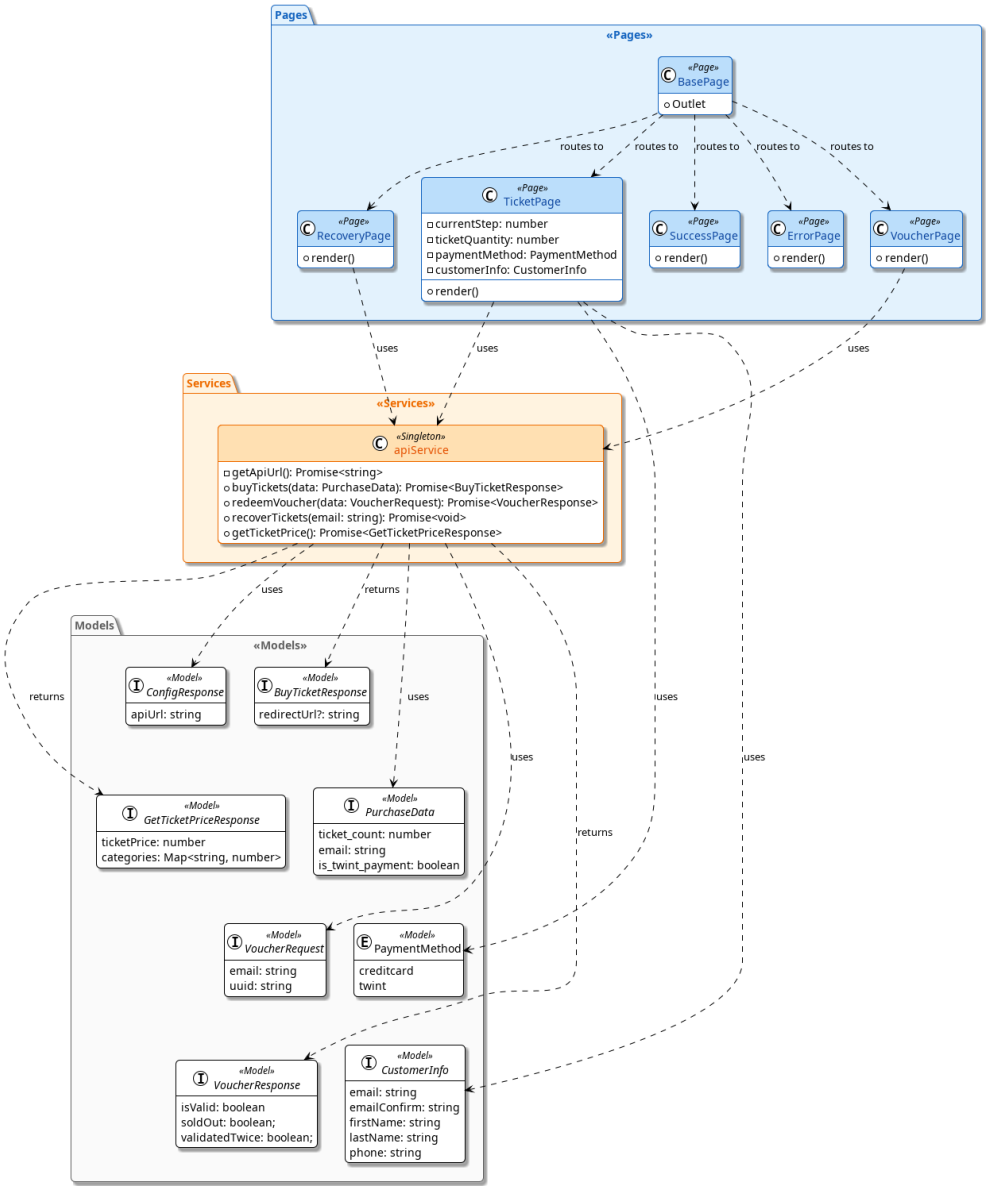



Figure C.3: Ticket shop code diagram

### C.6.2 Dashboard

The following section describes the architecture and implementation of the Dashboard application which is illustrated in the corresponding Figure C.4. The Dashboard is a restricted application for event organizers to manage ticket sales and monitor statistics. It is also built as a Single Page Application using React and bundled with Vite with performance in mind (FR-03). How the routing and the pages are structured is explained in the paragraph C.6.2.1.

#### C.6.2.1 Architecture and Routing

The pages and the routing structure are depicted in the blue package of the figure C.4. The dashboard is protected by a `ProtectedRoute` component that checks for a valid authentication token before rendering any child routes. The application layout is defined in DashboardLayout, which includes a persistent Sidebar for navigation and a Header. The main routes are:

- **/login**: The public login page.
- **/dashboard**: The main overview page displaying charts and statistics.
- **/tickets**: A detailed list view of all sold tickets.
- **/inHouse**: A page for downloading vouchers or sending tickets via mail.

#### C.6.2.2 Services

The dashboard relies on several specialized services to handle data fetching and authentication:

- **API Service:** Configures a global Axios instance. It uses request interceptors to dynamically resolve the API base URL from a local /config endpoint (similar to the Ticket shop) and injects the JWT authentication token from cookies into every request.
- **Auth Service:** Manages the user session. It handles login, token refreshing, and logout. Tokens are securely stored in the cookies.
- **Polling Service:** To provide near real-time updates without the complexity of WebSockets, we implemented a polling service. It uses the Observer pattern, allowing multiple components to subscribe to data updates. It fetches the latest Ticket and Voucher data at a set interval and notifies all subscribers.

#### C.6.2.3 State Management and Custom Hooks

To connect React components with the pollingService, we created custom hooks:

- **useTickets**: Subscribes to ticket updates from the polling service. It manages the local state for the ticket list, loading status, and errors. It automatically starts the polling process when the first component mounts.
- **useVouchers**: Similar to useTickets, but for voucher data.

This abstraction allows any component to access real-time data with a single line of code (`const { tickets } = useTickets();`) without worrying about the underlying fetching logic.

#### C.6.2.4 Component Implementation

The dashboard visualizes data using react-chartjs-2 and chart.js. Components like TicketPieChart and TicketSalesPerDay render interactive graphs to give organizers a quick overview of sales performance. Data models such as Ticket and Voucher are strictly typed in TypeScript interfaces, mirroring the backend structures to ensure consistency.

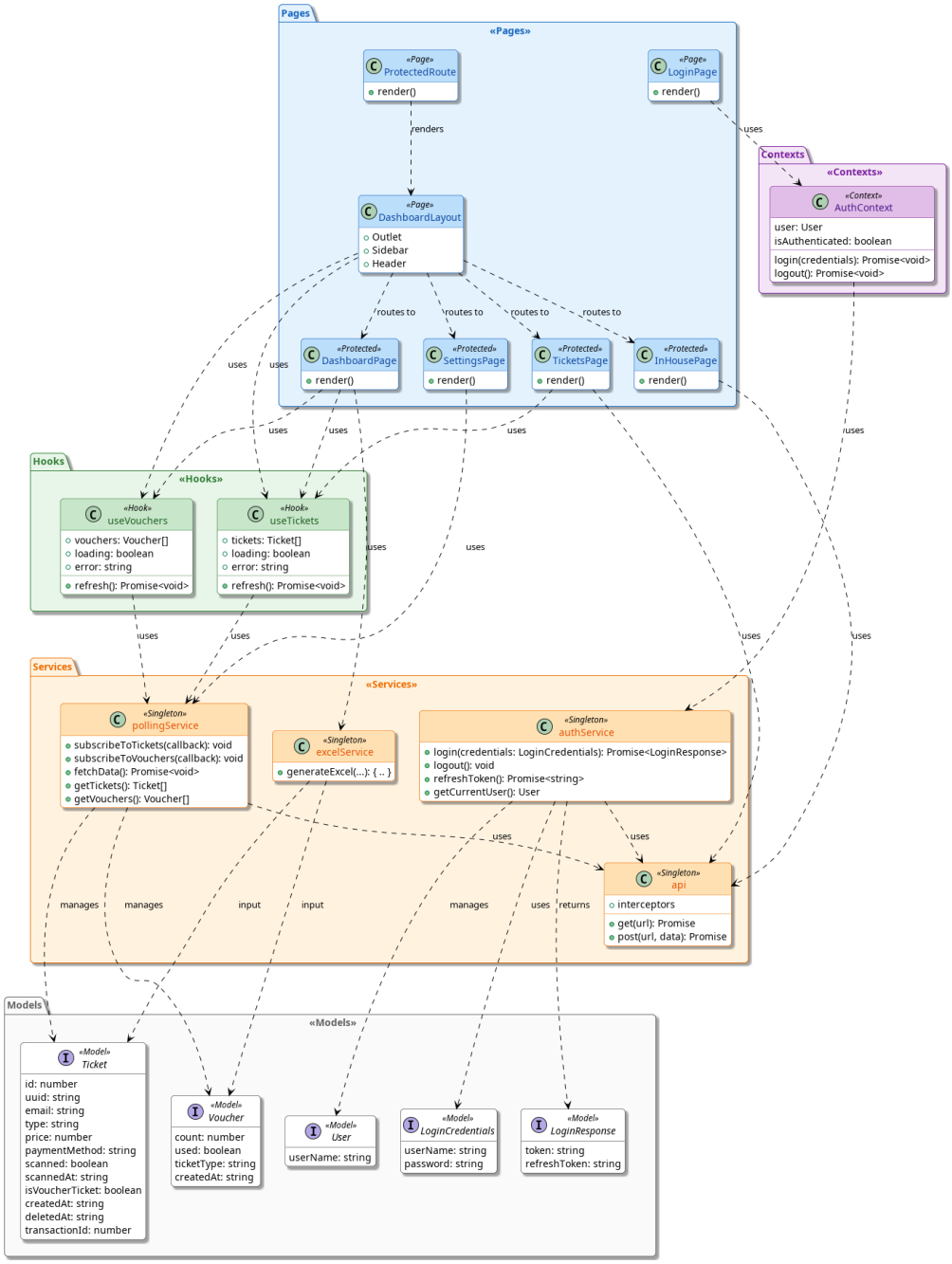



Figure C.4: Dashboard code diagram

### C.6.3 Scanner

The following section describes the architecture and implementation of the Scanner application which is illustrated in the corresponding Figure C.5.
The Scanner application is a tool designed for event workers to validate tickets at the entrance. The main idea is to use it on mobile devices, thus providing portability and easy of use for the staff. With this in mind it is built using React Native with Expo, allowing it to run natively on iOS and Android devices while also maintaining a web-compatible version which serves as a fallback in case we don't publish native builds.

#### C.6.3.1 Architecture and Routing

The application uses Expo Router for file based routing, similar to how Next.js does it. The navigation structure is defined in a dedicated file, which wraps the entire application in an AuthProvider to manage global authentication state. The main routes are:

- **/login**: The entry point for unauthenticated users.
- **/scanner**: The core functionality screen. It activates the device's camera to scan QR codes and provides immediate visual and audio feedback.
- **/boxoffice**: A secondary protected route that allows staff to sell tickets directly at the door.

#### C.6.3.2 Authentication and Security

Unlike the web-based Ticket shop and Dashboard which use HTTP-only cookies, the Scanner app must persist credentials securely on the device. We implemented a hybrid storage strategy in authService.ts:

- **Native (iOS/Android):** Uses expo-secure-store to encrypt and store JWT tokens (access and refresh tokens) in the device's secure keychain/keystore.
- **Web Fallback:** Uses secure cookies when running in a browser environment.

The AuthContext exposes the authentication state and methods (login, logout) to the rest of the app, ensuring that protected screens like /scanner and /boxoffice automatically redirect unauthenticated users to the login screen.

#### C.6.3.3 Services and API Integration

The apiService.ts acts as a singleton wrapper for all network requests:

- **Authenticated Fetch:** A wrapper around the native fetch API that automatically injects the Bearer token from secure storage into the Authorization header.

- **Automatic Token Refresh:** If an API call returns a 401 Unauthorized status, the service intercepts the response, attempts to use the stored refresh token to get a new access token, and seamlessly retries the original request. This ensures workers are not logged out in the middle of scanning.
- **Dynamic Configuration:** Similar to the other frontends, it attempts to resolve the API base URL dynamically.

#### C.6.3.4 Native Features and Implementation Details

The Scanner app uses specific native capabilities for an optimal user experience:

- **Camera Integration:** Uses expo-camera to scan QR codes. The implementation in scanner.tsx includes a "pause" mechanism to prevent multiple scans of the same code while the API request is processing.
- **Audio Feedback:** Uses expo-audio to play distinct sounds for valid (success.mp3) and invalid (error.mp3) tickets.
- **Flashlight Control:** Includes a toggle for the device's flashlight in case its dark.

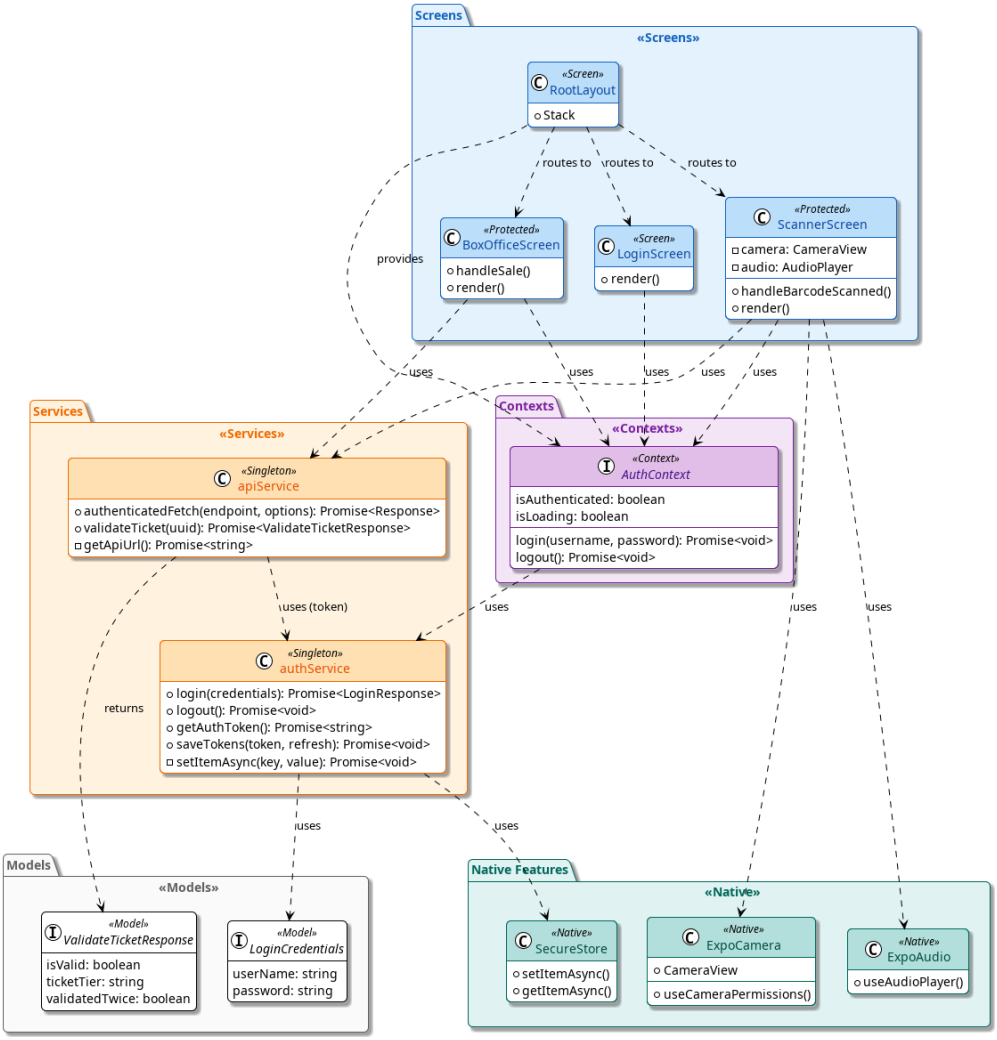



Figure C.5: Scanner code diagram

## C.7 Frontend Pictures

### C.7.1 Ticket shop

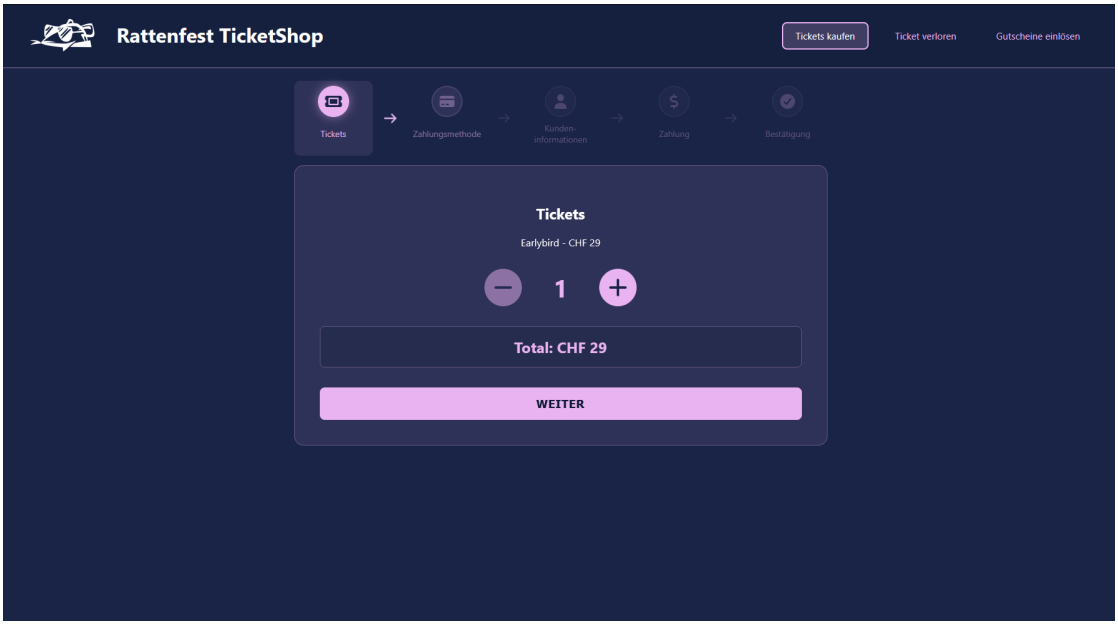



Figure C.6: Ticket shop - Ticket Amount Selection

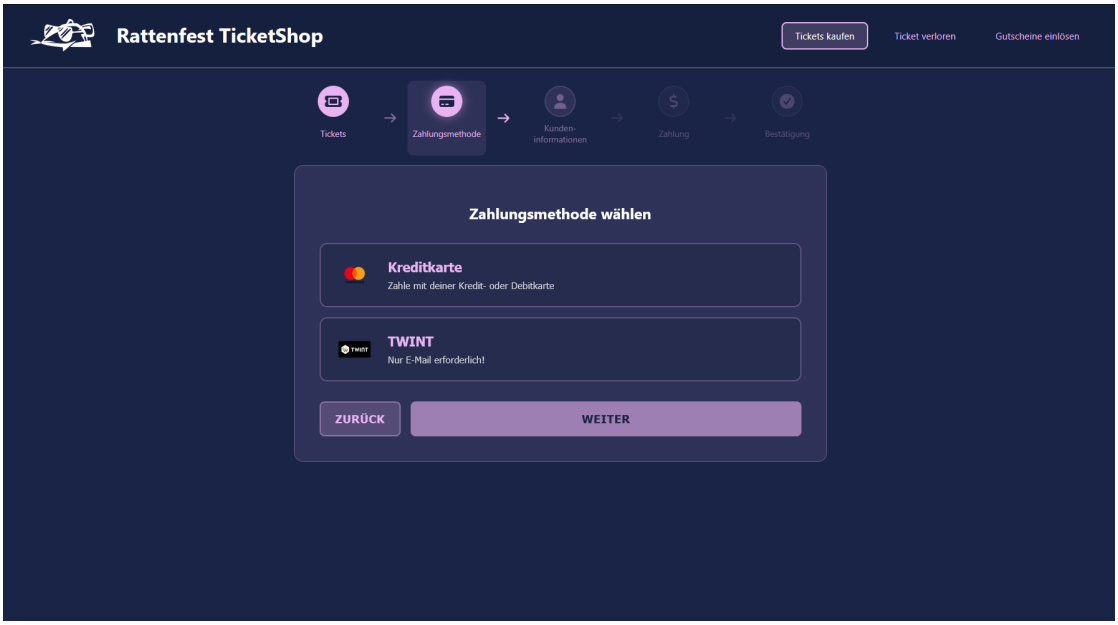



Figure C.7: Ticket shop - Payment Method Selection

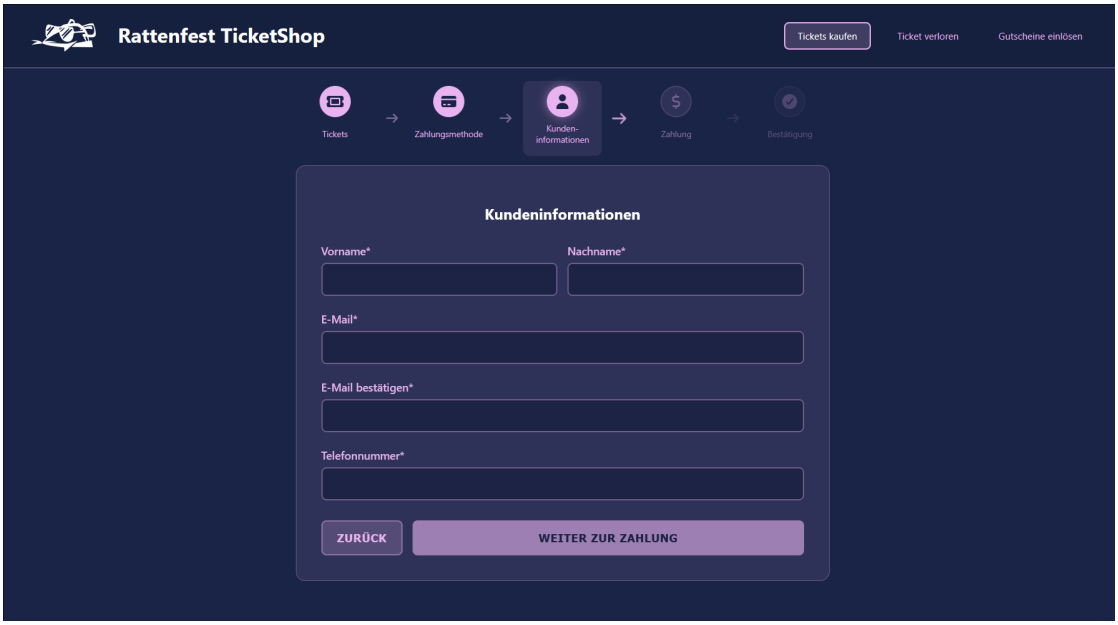



Figure C.8: Ticket shop - Personal Information (Credit Card)

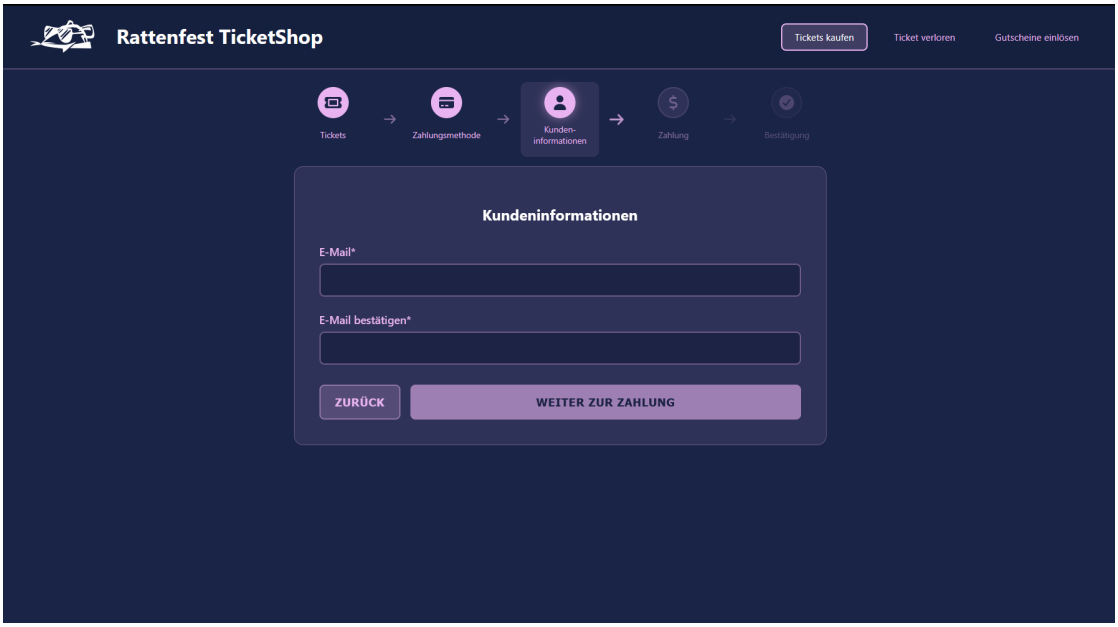



Figure C.9: Ticket shop - Personal Information (Twint)

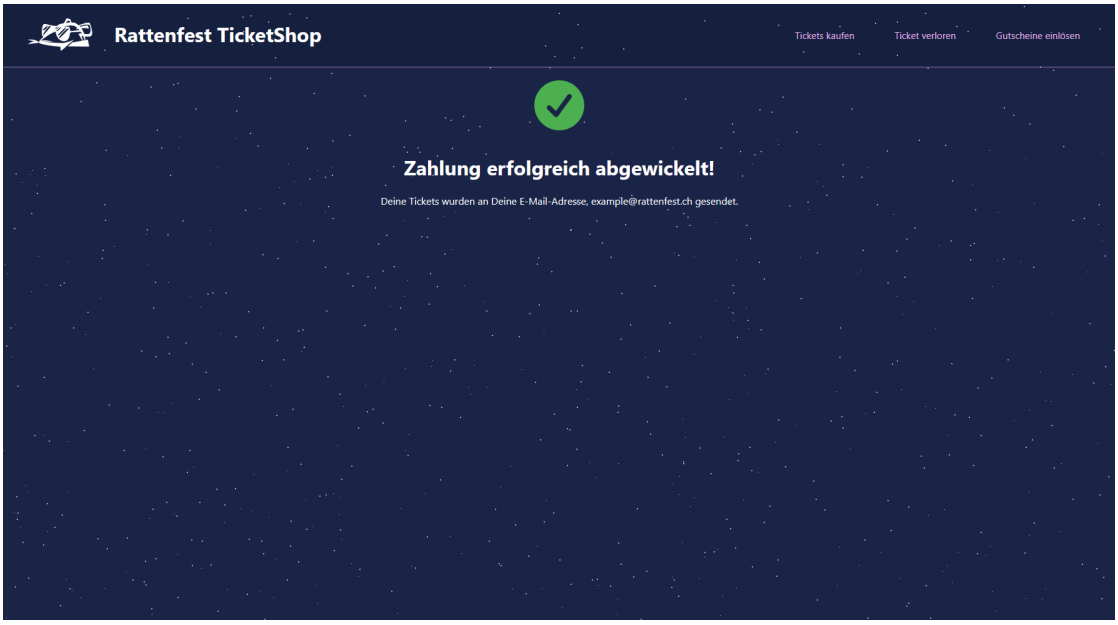



Figure C.10: Ticket shop - Success

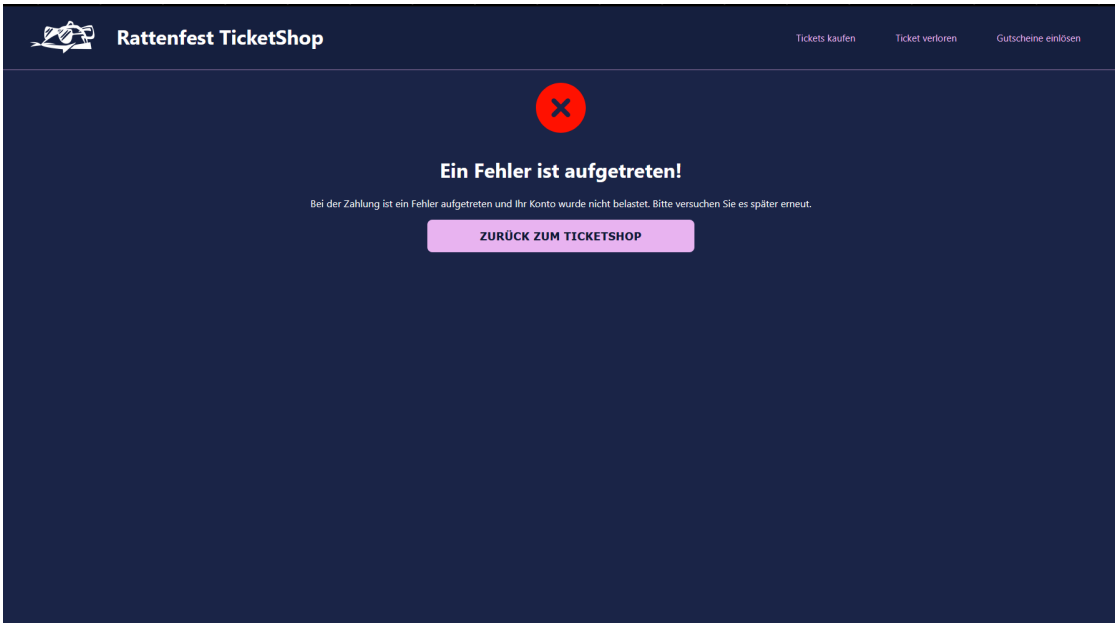



Figure C.11: Ticket shop - Error

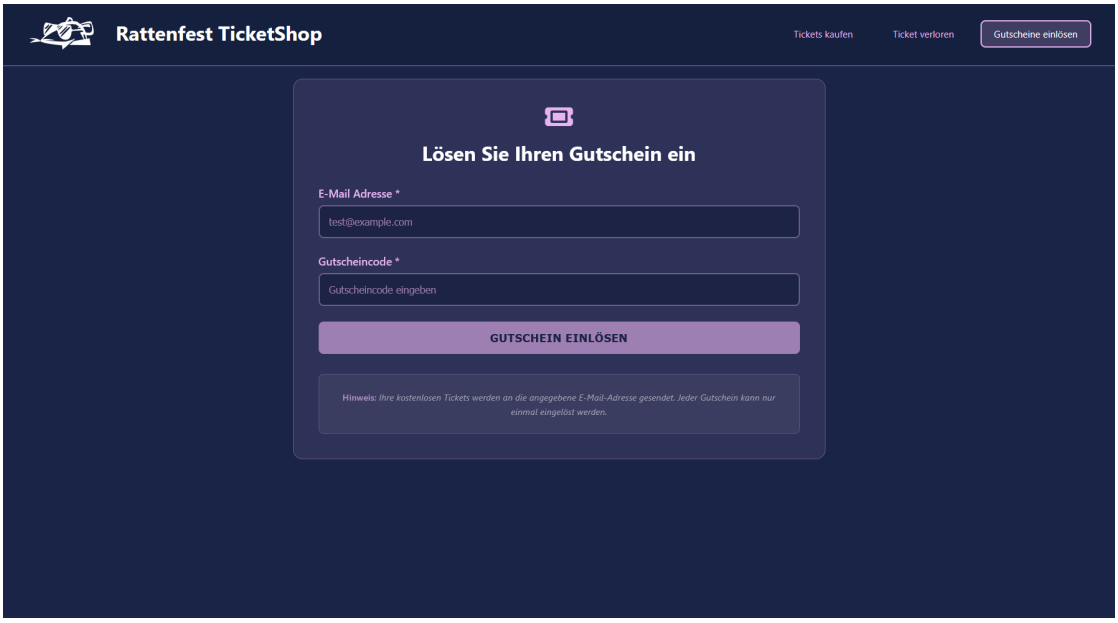



Figure C.12: Ticket shop - Voucher

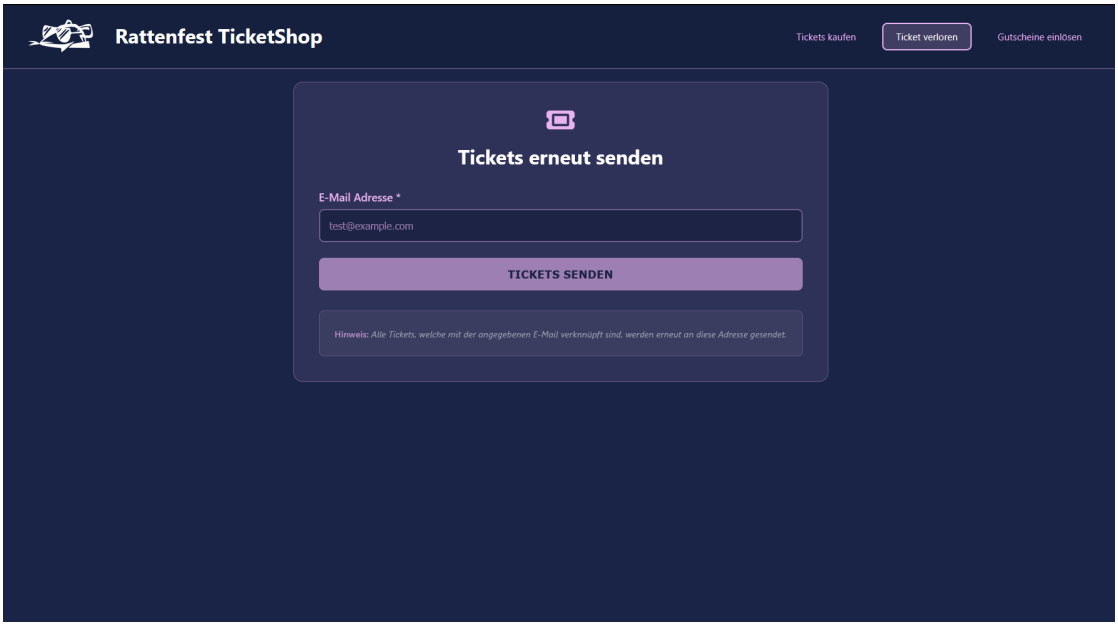



Figure C.13: Ticket shop - Recovery

### C.7.2 Dashboard

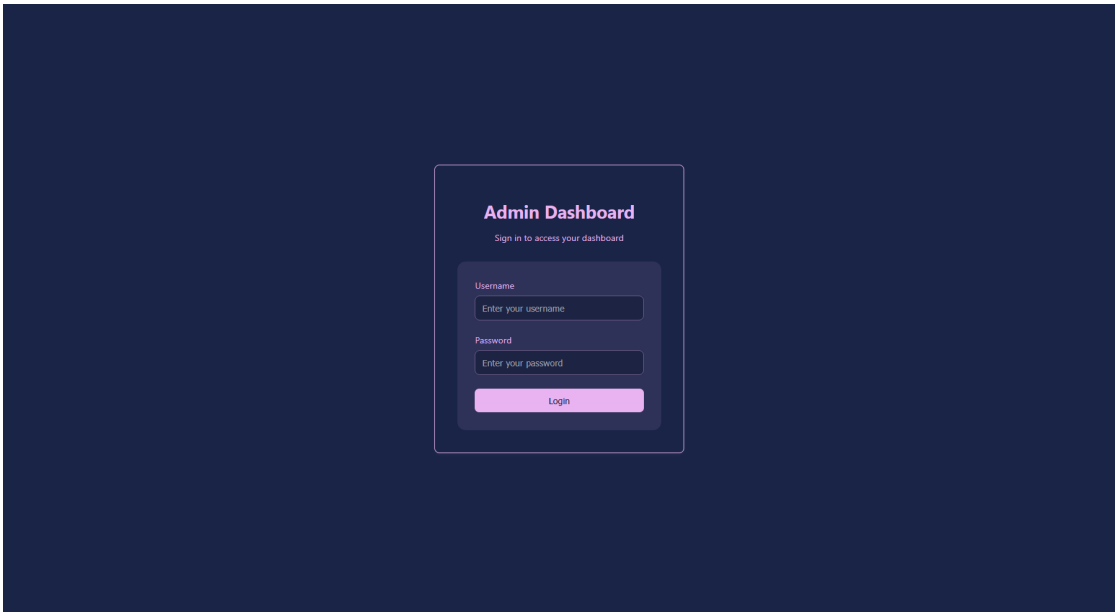



Figure C.14: Dashboard - Login


Figure C.15: Dashboard - Main Dashboard

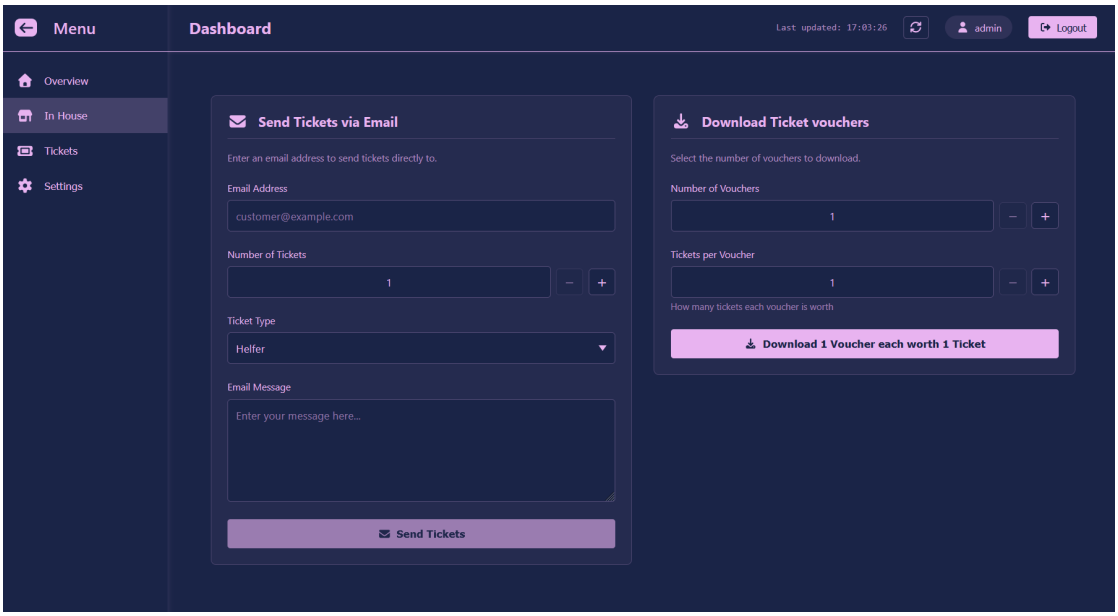



Figure C.16: Dashboard - InHouse

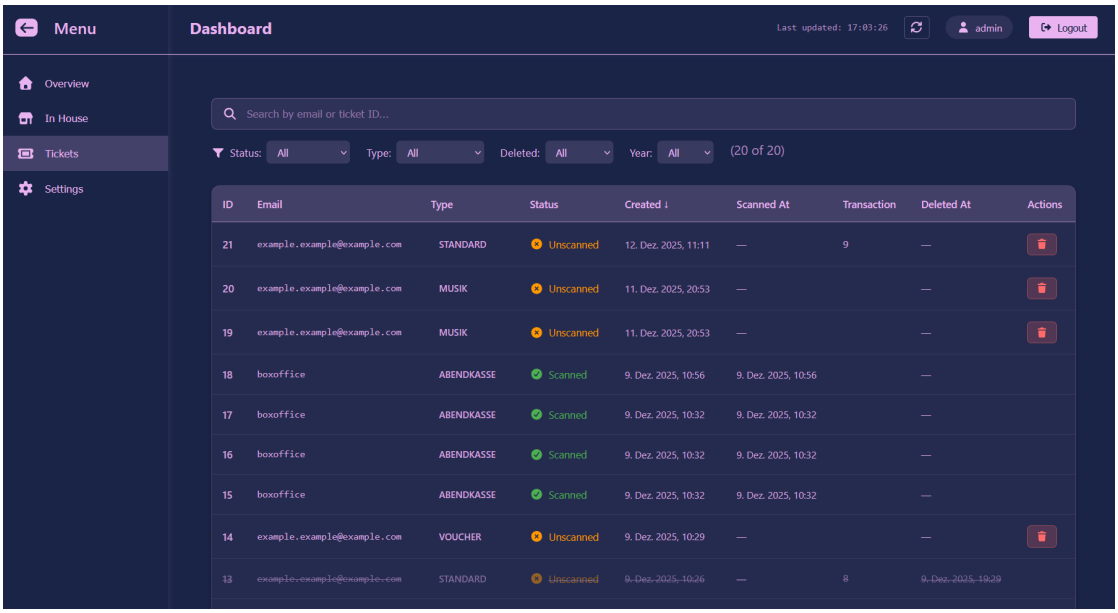



Figure C.17: Dashboard - Tickets

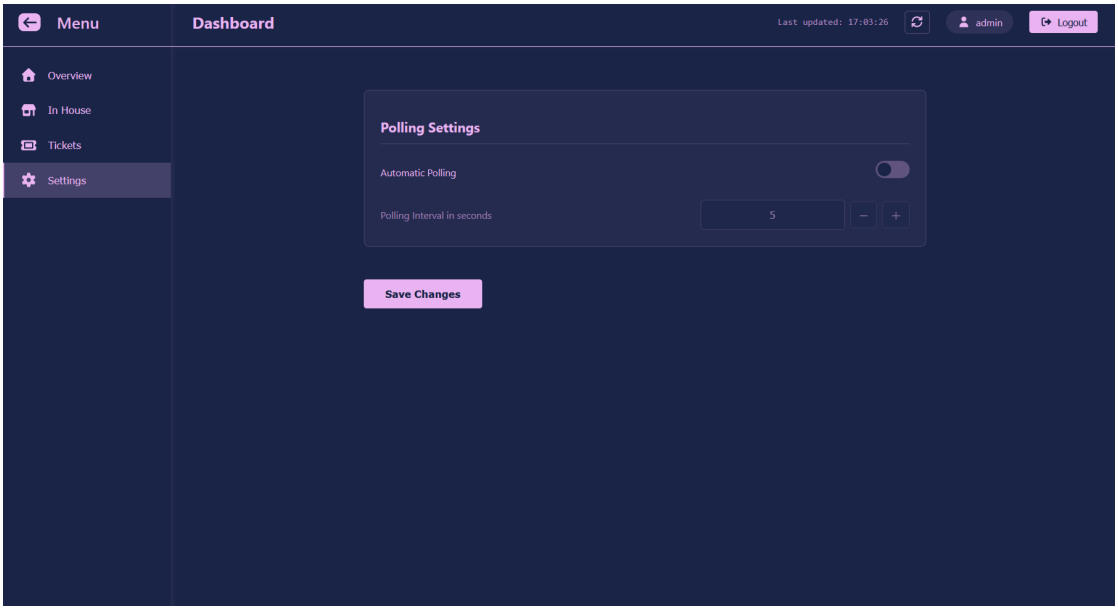



Figure C.18: Dashboard - Settings

### C.7.3 Scanner

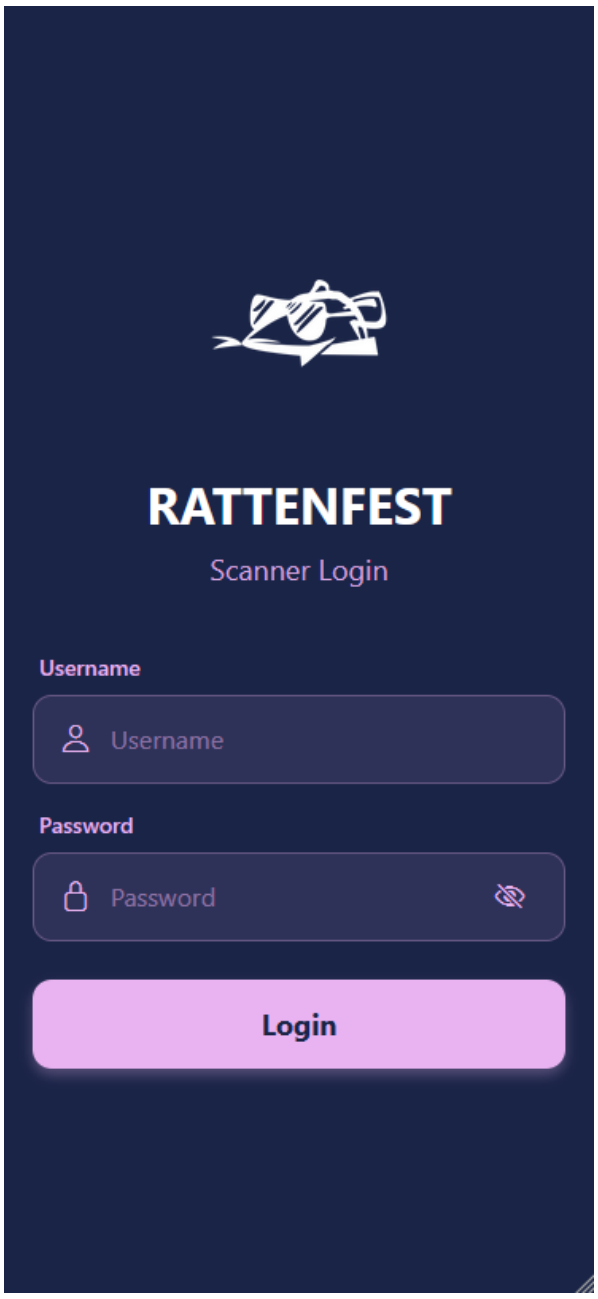



Figure C.19: Scanner - Login

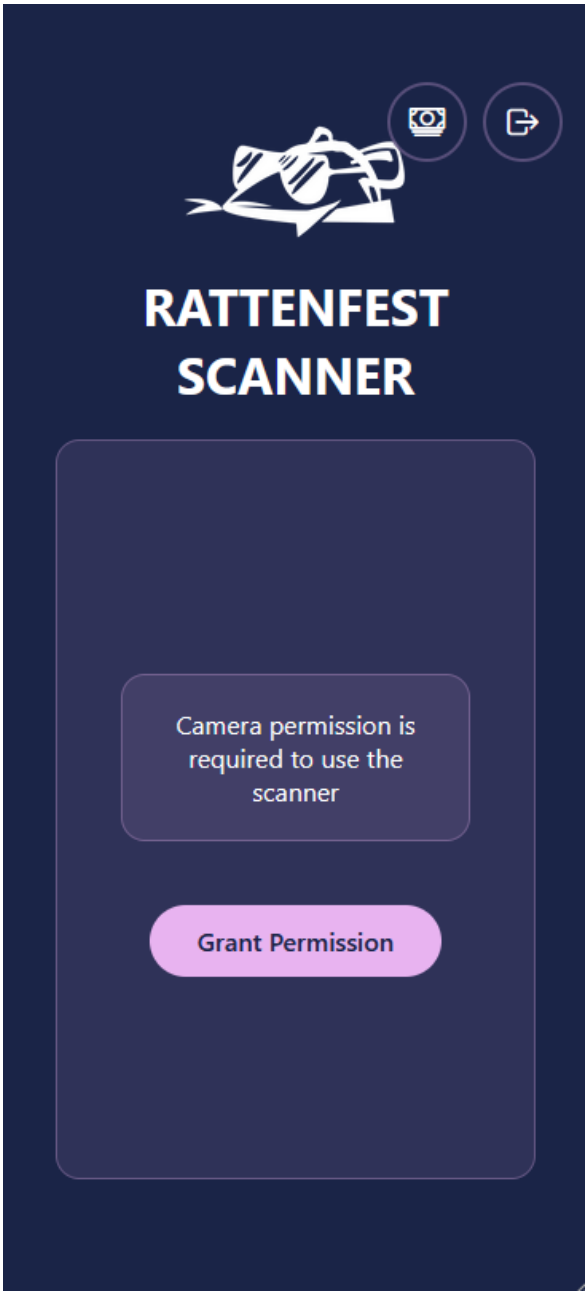



Figure C.20: Scanner - Scanner

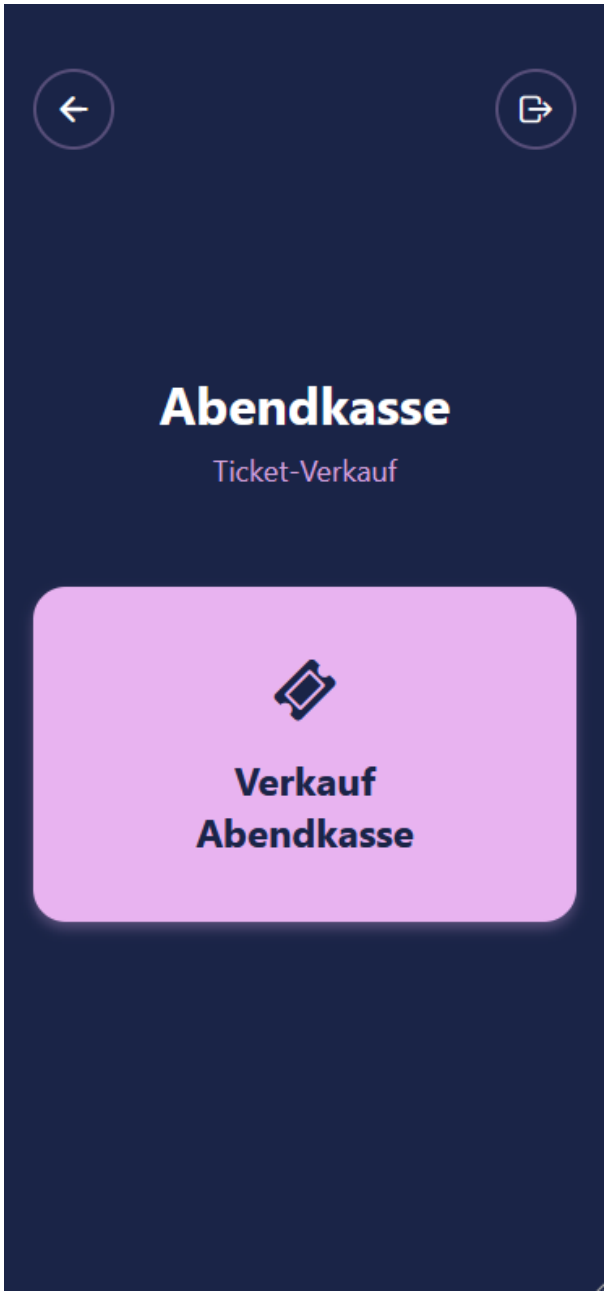



Figure C.21: Scanner - Box Office

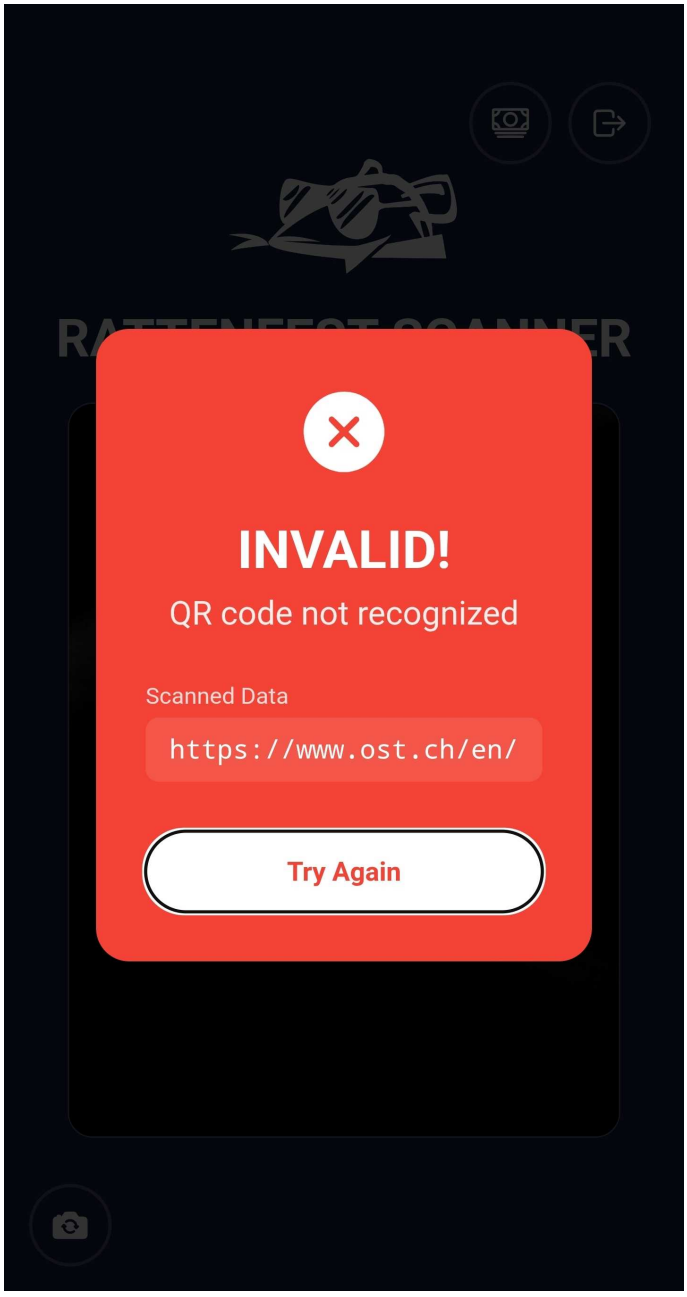



Figure C.22: Scanner - Unknown

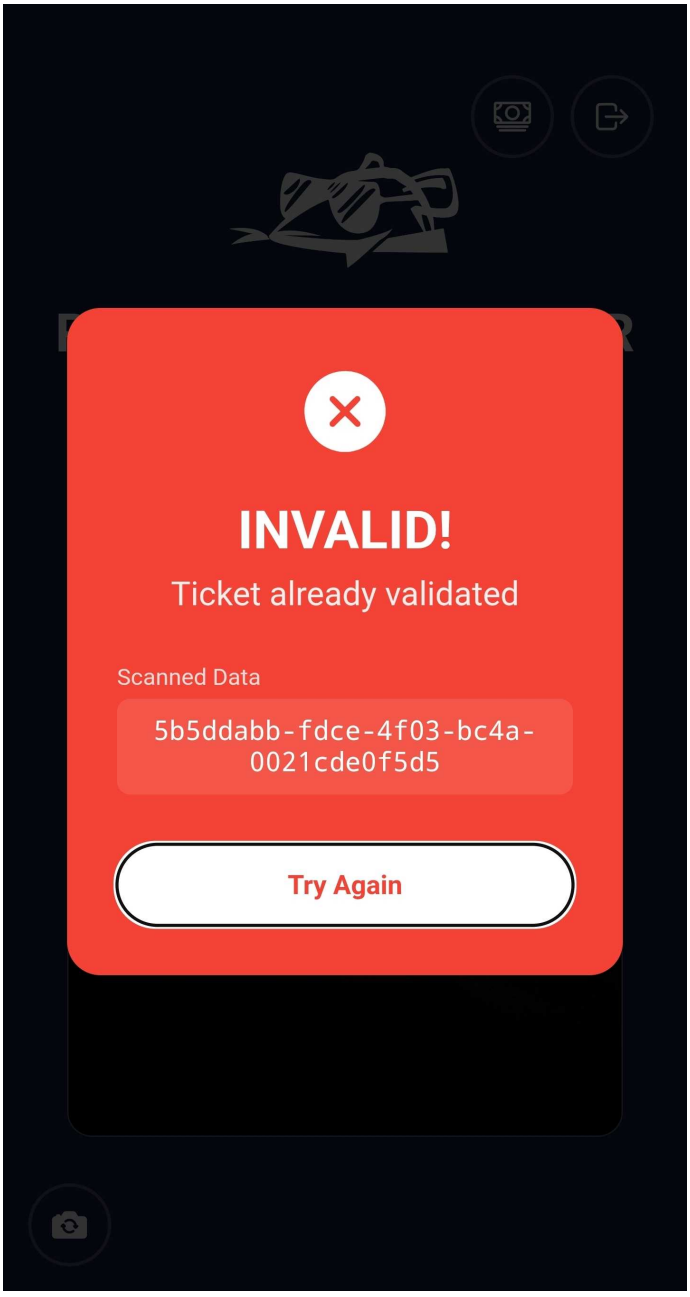



Figure C.23: Scanner - Already Validated

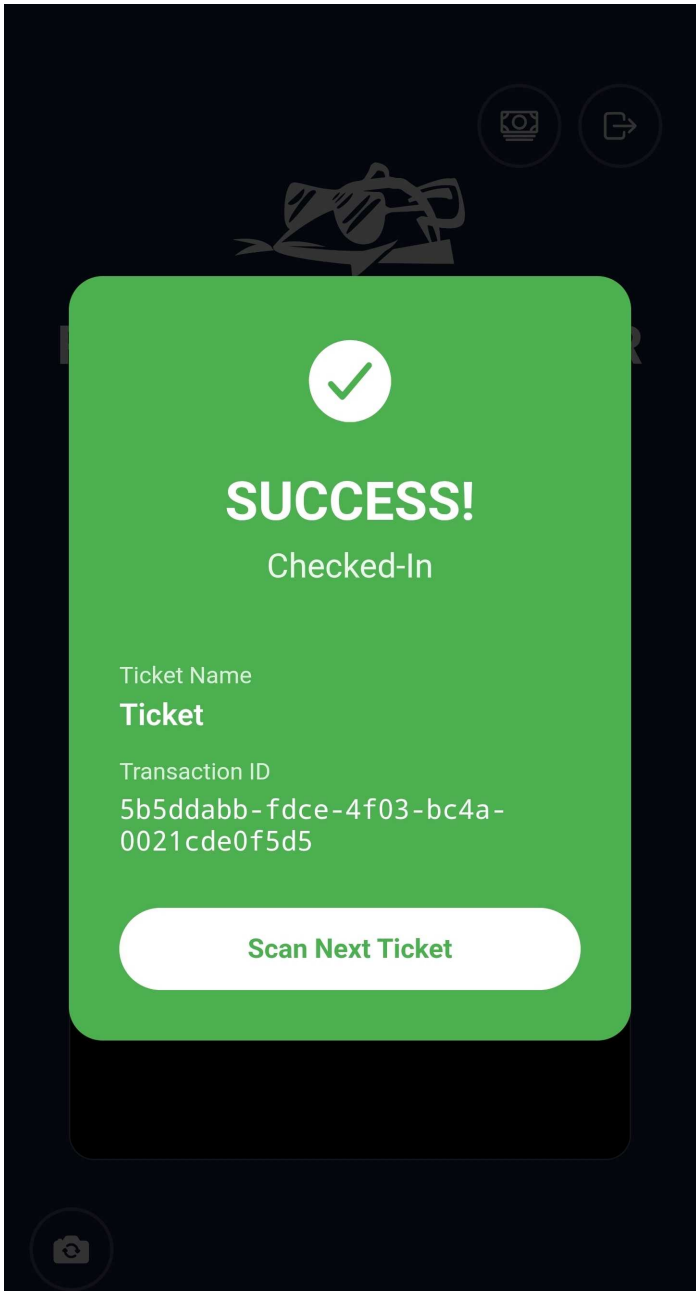



Figure C.24: Scanner - Success

## C.8 Tickets & Vouchers

### C.8.1 PDF Ticket


Figure C.25: PDF Ticket with Testing Note

### C.8.2 PDF Voucher


Figure C.26: PDF Voucher with Testing Note

## C.9 hi.events

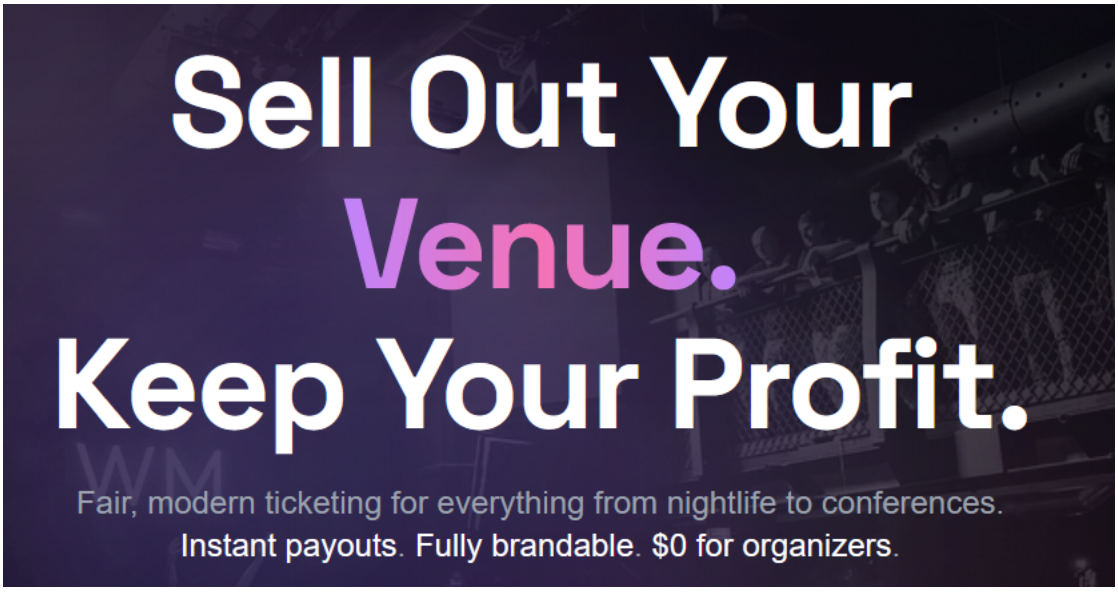



Figure C.27: Hi.events start page

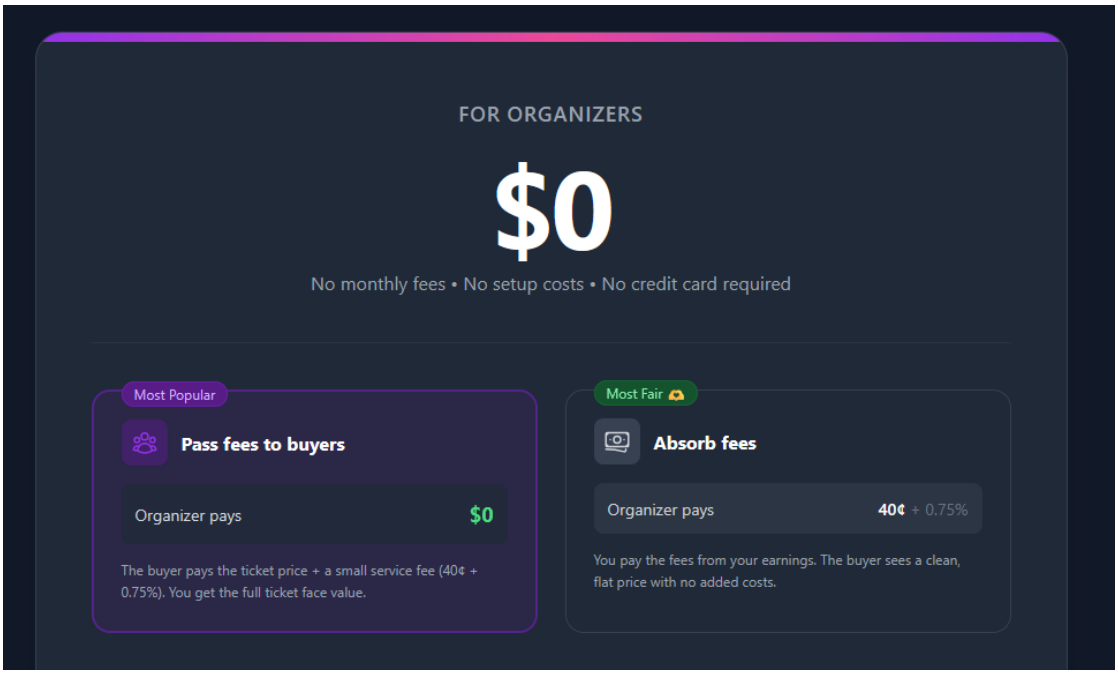



Figure C.28: Hi.events start page

# Appendix D

# Declaration of Independence

**Erklärung**
Ich erkläre hiermit,

- dass ich die vorliegende Arbeit selbst und ohne fremde Hilfe durchgeführt habe, ausser derjenigen, welche explizit in der Aufgabenstellung erwähnt ist oder mit der Betreuerin dem Betreuer schriftlich vereinbart wurde,
- dass ich sämtliche verwendeten Quellen erwähnt und gemäss gängigen wissenschaftlichen Zitierregeln korrekt angegeben habe,
- dass ich keine durch Copyright geschützten Materialien (z.B. Bilder) in dieser Arbeit in unerlaubter Weise genutzt habe.

**Ort, Datum:** 19.12.2025, Rapperswil

Doriano Di Pierro

Silvan Lendi

Fabio Zahner

# Appendix E

# Meeting Minutes

**Kick-Off Meeting**
**19.09.25**

| **Thema** | **Notizen** |
|---|---|
| Diskussion | |
| Todo's | Die folgenden Punkte werden bis zum nächsten Meeting erledigt:<br>• Marktanalyse<br>• Anforderungen<br>• Zeitplan<br>• Meilensteine<br>• Projektrisiken |

---

**Meeting 1**
**23.09.25**

| **Thema** | **Notizen** |
|---|---|
| Diskussion | |
| Todo's | Die folgenden Punkte werden bis zum nächsten Meeting erledigt:<br>• Functinoal requirements erweitern um uns von bestehenden Produkten abzuheben<br>• Nonfunctional requirements ergänzen<br>• Quellen für requirements aufzeigen<br>• "goal of the project" ausführen<br>• "parts" im dokument weglassen<br>• additional artifacts und meeting minutes in anhang<br>• conclusion chapter am schluss<br>• Functional requirements klassifizieren nach wichtigkeit |

---

**Meeting 2**
**30.09.25**

| Thema | Notizen |
|---|---|
| Diskussion | Ein erstes Mock-Up der Ticket-Verkauf Plattform wurde gezeigt. Verschiedene Aspekte des Designs wurden besprochen. |
| Todo's | Die folgenden Punkte werden bis zum nächsten Meeting erledigt:<br>• Änderungen seit letzter Woche wurden besprochen<br>• Feedback zu folgenden Punkten wurde gegeben: Generell zu verbessernde Wissenschaftliche Schreibweise, mehr Quellen für Begründungen (NFRs, Technologieauswahl), Dokumentation der Payment Technologie |

---

**Meeting 3**
**07.10.25**

| Thema | Notizen |
|---|---|
| Todo's | Die folgenden Punkte werden bis zum nächsten Meeting erledigt:<br>• Farben in Tabellen überarbeiten<br>• user stories fragenkatalog ausführen<br>• zahlen bei technology decisions verwenden<br>• kontextdiagramm für payment process aufzeigen<br>• text überarbeiten so dass wissenschaftlich klingt |

---

**Meeting 4**
**14.10.25**

| Thema | Notizen |
|---|---|
| Todo's | Die folgenden Punkte werden bis zum nächsten Meeting erledigt:<br>• Farben in Tabellen überarbeiten<br>• begriffe im bild von payment process aufzeigen<br>• abkürzungen in tabellen in überschrift tun<br>• begründen warum twint eher benutzt wird als kreditkarten |

---

**Meeting 5**
**21.10.25**

| Thema | Notizen |
|---|---|
| Todo's | Die folgenden Punkte werden bis zum nächsten Meeting erledigt:<br>• Ausbauen der Scanner App<br>• Rules für User Tests definieren<br>• Am Backend weiterarbeiten |

**Meeting 6**
**28.11.25**

| Thema | Notizen |
|---|---|
| Diskussion | Möglichkeit einer Bachelorarbeit zur weiterführung des Projekts. |

**Meeting 7**
**5.11.25**

| Thema | Notizen |
|---|---|
| Diskussion | Sind verben in endpoints in Ordnung? |
| Todo's | Testing webseite, healthchecks, endpoints anpassen. |

**Meeting 8**
**11.11.25**

| Thema | Notizen |
|---|---|
| Diskussion | Das Team demonstriert einen Kompletten Ablauf eines "Voucher zu Ticket zu Validierung" Workflows. |
| Todo's | Das Team fokussiert sich auf die Fertigstellung der "Should Have" und "Nice to Have" Items. |

**Meeting 9**
**19.11.25**

| Thema | Notizen |
|---|---|
| Diskussion | Das Team bespricht das erhaltene Feedback für die Dokumentation. |

**Meeting 10**
**25.11.25**

| Thema | Notizen |
|---|---|
| Demo | Das Team zeigt einen kompletten Ablauf der verschiedenen Tätigkeiten, welche mit der Applikation möglich sind. |
| Feedback | F. Loch gibt Feedback in einem kleinen User Test. |

**Meeting 11**
**02.12.25**

| Thema | Notizen |
|---|---|
| Testing | Das Team zeigt das Testing-Konzept und diskutiert weitere testbare Bereiche der Applikation. |
| Security | F. Loch bringt Vorschläge, durch welche weitere Schritte das System sicherer gemacht werden kann. |

**Meeting 12**
**09.12.25**

| Thema | Notizen |
|---|---|
| Dokumentation | Das Team bespricht das erhaltene Feedback zur Dokumentation. |
| Frontend | Kleinere Anpassungen, welche noch benötigt werden, werden besprochen. |

**Meeting 13**
**16.12.25**

| Thema | Notizen |
|---|---|
| Dokumentation | Finale Anpassungen zur Dokumentation werden besprochen. |
| Abgabe | Details zur Abgabe werden besprochen. |

# Bibliography


[Andnd] AndradeTC86. End-to-end testing technology (gist), n.d. Accessed: 04.12.2025.

[Artnd] Artillery. Playwright integration, n.d. Accessed: 04.12.2025.

[Blond] Toptal Engineering Blog. Server-side i/o performance, n.d. Accessed: 25.09.2025.

[Brond] BrowserStack. Performance testing with cypress, n.d. Accessed: 04.12.2025.

[Dai21] Software Engineering Daily. Why we switched from python to go, 2021. Accessed: 26.09.2025.

[dat25] Datatrans: Pricing, October 2025. [Online; accessed 1. Oct. 2025].

[Emm23] Phani Sekhar Emmanni. Comparative analysis of angular, react, and vue.js in single page application development. *International Journal of Science and Research (IJSR) 12(6)*, 06 2023.

[Evenda] Eventbrite. Investor overview, n.d. Accessed: 18.11.2025.

[Evendb] Eventfrog. References, n.d. Accessed: 18.11.2025.

[Hi nd] Hi Events. Hi events, n.d. Accessed: 18.11.2025.

[KV18] Marin Kaluža and Bernard Vukelic. Comparison of front-end frameworks for web applications development. *Zbornik Veleučilišta u Rijeci*, 6:261–282, 01 2018.

[Loand] LoadView. Selenium load testing, n.d. Accessed: 04.12.2025.

[Log25] LogRocket. Axios vs fetch: Best http requests, 2025. Accessed: 04.12.2025.

[MD22] Anshita Malviya and Rajendra Kumar Dwivedi. A comparative analysis of container orchestration tools in cloud computing. In *2022 9th International Conference on Computing for Sustainable Global Development (INDIACom)*, pages 698–703, 2022.

[Mic24] Brütsch Michael. Payment Service Provider Vergleich: Kreditkartenzahlungen im Online Shop annehmen - Webdesign, SEO & Online Marketing | Schwyzer Webagentur 80/20 Webdesign, February 2024. [Online; accessed 1. Oct. 2025].

[Nie10] Jakob Nielsen. Website response times, 2010. Accessed: 29.09.2025.

[OYO23] OYOVA. Is php still relevant?, 2023. Accessed: 26.09.2025.

[P23] Mukhil P. Building scalable microservices with golang: Lessons from real-world implementations, 2023. Accessed: 26.09.2025.

[pay25] Payrexx, October 2025. [Online; accessed 1. Oct. 2025].

[Pland] Planeks. Benefits of python for rapid prototyping, n.d. Accessed: 26.09.2025.

[pos25] PostFinance, October 2025. [Online; accessed 1. Oct. 2025].

[Pronda] ProductPlan. Moscow prioritization, n.d. Accessed: 18.11.2025.

[Prondb] Project Discovery. Nuclei - fast and customizable vulnerability scanner, n.d. Accessed: 07.12.2025.

[Pytnd] Real Python. An intro to parallel processing in python, n.d. Accessed: 26.09.2025.

[SKS20] A. Silberschatz, H.F. Korth, and S. Sudarshan. *Database System Concepts*. McGraw-Hill Education, 2020.

[str25] Stripe, October 2025. [Online; accessed 1. Oct. 2025].

[Swi25] Aktuelle Ergebnisse | Swiss Payment Monitor | ZHAW & HSG, October 2025. [Online; accessed 10. Oct. 2025].

[Ticnd] Ticketcorner. About us, n.d. Accessed: 18.11.2025.

[Ver22] Dhruv Verma. A comparison of web framework efficiency–performance and network analysis of modern web frameworks, 2022. Accessed: 04.12.2025.

[vK17] Danny van Kooten. Laravel to golang, 2017. Accessed: 25.09.2025.

[Zen23] Zend. Scaling php: What makes scaling php difficult?, 2023. Accessed: 26.09.2025.