Event Ticket Platform

Published: June 13, 2025

Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License

This work is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License.

You are free to:

- Share — copy and redistribute the material in any medium or format
- Adapt — remix, transform, and build upon the material

Under the following terms:

- Attribution — You must give appropriate credit to Devtiro, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.
- NonCommercial — You may not use the material for commercial purposes.
- ShareAlike — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.
- No additional restrictions — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.

Notices:

- You do not have to comply with the license for elements of the material in the public domain or where your use is permitted by an applicable exception or limitation.
- No warranties are given. The license may not give you all of the permissions necessary for your intended use. For example, other rights such as publicity, privacy, or moral rights may limit how you use the material.

To view a copy of this license, visit https://creativecommons.org/licenses/by-nc-sa/4.0/

For attribution, please include the following:

"Event Ticket Platform" by Devtiro is licensed under CC BY-NC-SA 4.0. Original content available at https://www.youtube.com/@devtiro and https://www.devtiro.com.

Event Ticket Platform

Getting Started

Welcome

Have you ever looked at some code and wondered *what on earth were they thinking?*

Why did you code it in *that way*?

Why not use library x? Why not use pattern y?

I think I've found a fix for that...

Over the next 8-12 weeks we're going to build an "Event Ticket Platform". This app let's organizers create events and sell tickets, event goers buy those tickets and the system handles the whole ticket QR code scanning at the event.

This is usually where I'd show you clips of the fanciest bits the finished app, but here's the thing -- the app's not built yet.

We've got a project brief, you can grab that PDF from Discord too, but that's it -- but fear not, that's what we want!

This build is going to be a bit more raw than my previous builds, as I'll be taking you through the build, as I build it.

No safety net, we're all in.

Without the benefit of foresight, We'll likely have to rework some stuff as we go, but as a result you'll get to see my real, as-it-happens, build thought process -- the good, the bad and the ugly.

Build Plan

Here's the plan for this build. It's likely to change as we learn more about the domain, but here's today's understanding:

1. System Design - Analyze the domain, user personas, UI Design and domain modelling That's this video.
2. Design REST API and the application's architecture
3. Project Setup, including Security
4. Domain Implementation
5. Event Creation & Management
6. Ticket Sale & Purchase
7. Ticket Validation
8. Sales Reporting

Prerequisites

That's the plan, we'll see how we get on, so let's cover the prereqs for this build.

This is going to be in the intermediate build territory, and as we're on a time limit we'll need to focus on the build and can't make too many diversions cover theory in-depth.

As a result, you should already be comfortable with:

- Java
- Spring Boot
- Spring Security

Have a basic understanding of:

- OAuth2 and OpenID connect
- React + npm

Source Code

This build will be focused on the Spring Boot backend and I'll be uploading the source code to Discord.

As an app isn't complete without a UI, I will also be building a React frontend for this build and uploading the source code to Discord too, so you can download it.

I'll not include how to code the react app in this series, but if you're interested in that, I'll be posting some details to the community on how to access it soon.

Getting Support

If you have questions or get stuck at any point during this build, there's a whole community of Developers, and dedicated help channels in Discord.

Right, with that covered, let's jump into system design.

Project Brief

In this lesson, we'll explore the project brief for our event ticketing platform, aiming to understand the requirements of the system we are to build.

Project Overview

Let's break down the main components of our event ticketing system.

Our system needs to handle event creation, ticket sales, sales monitoring, and ticket validation -- the complete event management lifecycle.

The platform will serve three types of users:

1. Event *organizers*
2. Event *attendees*
3. Event *staff*

Each user type has their own needs, and way of using the system.

User Story Analysis

Event Creation Requirements

Create Event

As an event organizer

I want to create and configure a new event with details like date, venue, and ticket types

So that I can start selling tickets to attendees

Acceptance Criteria

- Organizer can input event name, date, time, and venue
- Organizer can set multiple ticket types with different prices
- Organizer can specify total available tickets per type
- Event appears on the platform after creation

The event creation story focuses on organizers setting up new events.

We'll need to design a robust data model to store event details including:

- Name, date, time, and venue
- Multiple ticket types with varying prices
- Ticket quantity limits per type

The system must maintain data integrity to prevent issues like duplicate events or invalid ticket configurations.

Ticket Purchase Flow

Purchase Event Ticket

As an event goer

I want to purchase the correct ticket for an event

So that I can attend and experience the event

Acceptance Criteria

- Event goer can search for events
- Event goer can browse and select different ticket types available each even
- Event goer can purchase their chosen ticket type

The ticket purchase story requires a user-friendly search and selection process.
We'll need to implement:

- A search mechanism for events
- A clear display of available ticket types
- A secure payment processing system
- Real-time inventory management to prevent overselling

Sales Management Features

Manage Ticket Sales

As an event organizer

I want to monitor and manage ticket sales

So that I can track revenue and attendance

Acceptance Criteria

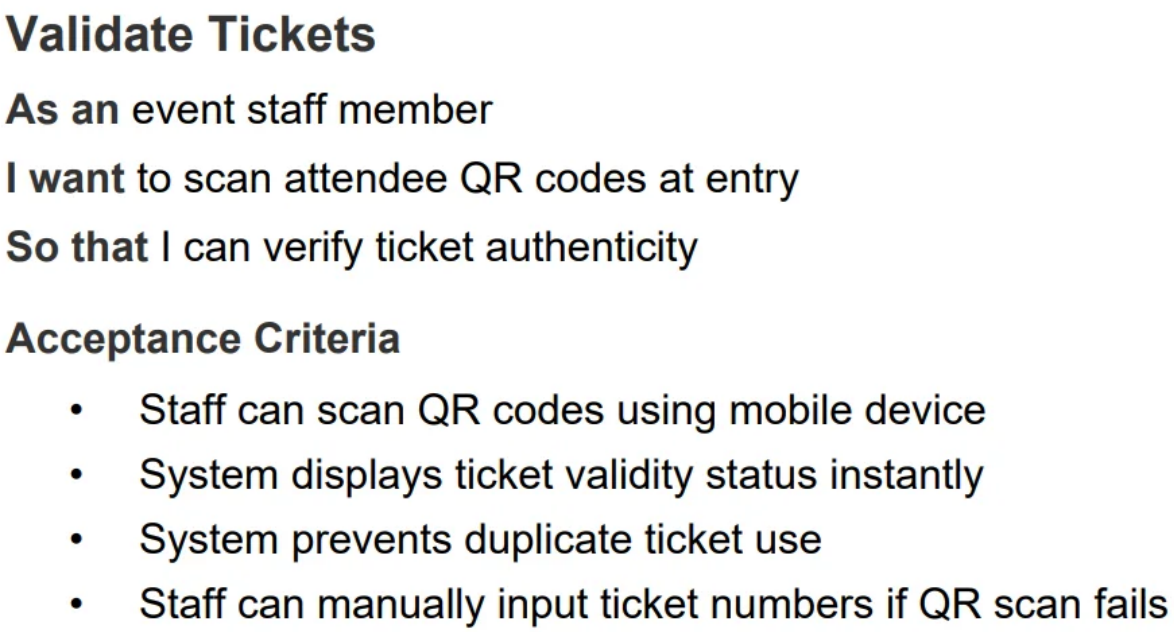
- Dashboard displays sales metrics
- Organizer can view purchaser details
- System prevents overselling of tickets
- Sales automatically stop at specified end date

The sales management story requires comprehensive tracking capabilities.
This involves creating:

- A dashboard for sales metrics
- Secure storage of purchaser information

- Automated sales control based on quantity and date rules

Ticket Validation System



The validation story focuses on entry management at events.
Key technical considerations include:

- QR code generation and scanning functionality
- Ticket status verification
- Prevention of duplicate ticket use
- Fallback manual entry system

Technical Implementation Notes

We'll need to create RESTful endpoints for each major function:

- Event management APIs
- Ticket purchase APIs
- Sales monitoring APIs
- Ticket validation APIs

Summary

- Platform manages complete event lifecycle from creation through validation
- System serves three user types: organizers, attendees, and staff members
- Features include event setup, ticket sales, monitoring, and entry validation

User Interface Design

Module Overview

In this module, we'll create an initial design for our event ticket platform.

We'll do this by looking to understanding our users, their needs, and how they'll interact with our system.

Module Structure

1. Explore the user personas
2. Explore the user journeys
3. Design the user interface

Learning Objectives

1. Describe the application's user personas
2. Describe the application's user journeys
3. Produce the first iteration of a user interface design

User Personas Summary

Let's summarize the user personas which model users of our system.

Please note that the following user personas were generated by an LLM based on our project brief, so we can be confident that these people are fictional, and their descriptions un-biased as possible.

Organizers

Corporate Event Manager

Working Name: Corporate Event Manager

Primary Goal: To represent company brand values through professional events while tracking attendee analytics for business development.

Key Challenge: Needs comprehensive analytics and reporting features to justify event ROI to executives while ensuring seamless attendee experience.

Distinguishing Characteristics:

- Data-driven decision maker who values detailed analytics
- Brand-conscious and needs customization options
- Security and privacy focused due to handling customer data

Usage Context: Uses the platform on both desktop and tablet devices, often preparing reports in the office but needing mobile access during events to monitor real-time metrics and address issues.

Event Planning Professional

Working Name: Event Planning Professional

Primary Goal: To efficiently create and manage multiple events with different ticket tiers while maximizing attendance and revenue.

Key Challenge: Juggling numerous events simultaneously while ensuring accurate ticket inventory and preventing overselling.

Distinguishing Characteristics:

- Detail-oriented with strong organizational skills
- Tech-savvy but values intuitive interfaces
- Time-conscious and appreciates automation

Usage Context: Uses the platform primarily on a laptop while at the office or remotely, often needs to make quick updates between meetings with clients and venue representatives.

Part-Time Event Organizer

Working Name: Part-Time Event Organizer

Primary Goal: To create occasional events with minimal effort while maintaining a professional appearance to attendees.

Key Challenge: Limited technical expertise and time to dedicate to learning complex systems while needing to appear professionally competent.

Distinguishing Characteristics:

- Prefers simple, guided interfaces with templates
- Values mobile accessibility for on-the-go management
- Prioritizes customer-facing aesthetics over backend complexity

Usage Context: Uses the platform primarily on mobile devices during free moments between other responsibilities, often creating and managing events during evenings and weekends.

Attendees

Busy Parent Event Attendee

Working Name: Busy Parent Event Attendee

Primary Goal: To reliably secure tickets for family-friendly events well in advance with clear information about venue facilities and accessibility.

Key Challenge: Coordinating tickets for multiple family members while ensuring all necessary information is available for planning the family outing.

Distinguishing Characteristics:

- Careful planner who books events weeks or months in advance
- Values clear information about venue accessibility and facilities
- Appreciates email confirmations and reminders

Usage Context: Primarily uses tablet or desktop devices during evening hours after children are in bed, often comparing multiple events before making a purchase decision, and may print physical tickets as backup.

Young Event-Goer

Working Name: Young Event-Goer

Primary Goal: To easily discover, purchase, and access tickets for trendy events without complications or hidden fees.

Key Challenge: Finding affordable tickets to popular events and ensuring ticket validation works smoothly to avoid entry issues.

Distinguishing Characteristics:

- Digital native who expects intuitive mobile experiences
- Price-sensitive but willing to pay for unique experiences
- Often purchases tickets at the last minute

Usage Context: Uses exclusively mobile devices to purchase tickets, often while commuting or during short breaks, and prefers digital tickets stored in mobile wallet apps for easy access.

Corporate Networking Attendee

Working Name: Corporate Networking Attendee

Primary Goal: To efficiently register for professional events and conferences with the ability to expense tickets and receive proper documentation.

Key Challenge: Needs detailed receipts and event information for expense reporting and calendar integration for busy schedule management.

Distinguishing Characteristics:

- Values streamlined checkout processes with minimal steps
- Requires detailed receipts for corporate expense reporting
- Appreciates calendar integration and professional reminders

Usage Context: Uses both desktop (during office hours) and mobile devices (while traveling) to purchase tickets, often needs to buy multiple tickets for colleagues, and requires seamless integration with professional tools like calendar and expense apps.

Staff

Event Staff Coordinator

Working Name: Event Staff Coordinator

Primary Goal: To efficiently manage entry validation processes for large events with multiple entry points and staff members.

Key Challenge: Ensuring consistent and accurate ticket validation across different entry points while handling high-volume entry during peak times.

Distinguishing Characteristics:

- Process-oriented with focus on security and accuracy
- Comfortable with technology but needs reliable, simple tools for team
- Values speed and reliability over complex features

Usage Context: Primarily uses the admin interface before events for setup and staff training, then uses mobile validation tools during events while moving between entry points to supervise staff.

Entry-Level Event Staff

Working Name: Entry-Level Event Staff

Primary Goal: To quickly and accurately validate tickets at event entry points without causing delays or errors.

Key Challenge: Must handle high-pressure entry situations with minimal training while maintaining positive customer interactions despite potential technical issues.

Distinguishing Characteristics:

- Limited technical expertise requires intuitive interfaces
- May be working first job or temporary position

- Values clear instructions and error messages

Usage Context: Uses primarily the mobile ticket scanning application during event hours, often in challenging environments (poor lighting, noisy, crowded) requiring quick and clear validation feedback.

Summary

- Explored user personas representing organizers, attendees and staff
- These user personas will influence how the system is designed

User Journey

Let's explore how these user personas may interact with our application.

I've used some imagination here to fill in the gaps, however we should still be able to learn a thing or two from these user journeys!

Organizers

Corporate Event Manager

```
journey
   title Corporate Event Manager - Platform User Journey
   section Event Setup
       Access platform: 5: Sarah Lee
       Configure event details: 4
       Set ticket types & pricing: 5
       Customize branding: 3
   section Pre-Event Management
       Monitor ticket sales: 5
       Review analytics dashboard: 5
       Export attendee data: 4
       Configure staff access: 3
   section Event Day Operations
       Brief staff on scanning process: 4
       Monitor entry validation: 5
       Handle scanning issues: 3
       Track real-time attendance: 4
   section Post-Event
       Export final attendance data: 5
       Generate sales report: 5
       Review validation metrics: 4
```

Event Planning Professional

```
journey
   title Event Planning Professional - User Journey
   section Event Setup
       Login to platform: 5: Priya
       Create new event: 4
       Configure ticket types: 3
       Set pricing tiers: 4
       Review event details: 5
       Publish event: 4
   section Sales Management
       Monitor real-time dashboard: 5
       Track ticket inventory: 4
       Export sales reports: 3
       Adjust ticket pricing: 4
   section Event Preparation
       Generate QR codes: 5
       Brief staff on check-in process: 4
       Test scanning equipment: 5
```

```
   section Event Day
       Initialize check-in system: 4
       Monitor entry analytics: 5
       Handle ticket exceptions: 3
       Generate final reports: 4
```

Part-Time Event Organizer

```
journey
   title Part-Time Event Organizer - User Journey
   section Event Creation
       Access platform during lunch break: 4: Marcus
       Create new event template: 3
       Input event details: 4
       Configure ticket types/prices: 5
       Preview event listing: 4
   section Pre-Event Management
       Monitor ticket sales on mobile: 5
       Check metrics during breaks: 4
       Update event details as needed: 3
       Share on social media: 5
   section Event Day Setup
       Generate QR scanner link: 4
       Brief venue staff on scanning: 3
       Test scanner functionality: 5
   section Event Execution
       Monitor entry scanning: 4
       Track real-time attendance: 5
       Handle scanning issues: 3
   section Post-Event
       Export attendance data: 4
       Review sales metrics: 5
       Archive event details: 4
```

Attendees

Busy Parent Event Attendee

```
journey
   title Family Event Planner - Sarah Mitchell's Journey
   section Research & Discovery
       Browse family events: 4: Sarah
       Read venue details & reviews: 5
       Check family calendar: 5
       Coordinate with spouse: 3
   section Planning
       Compare ticket options: 4
       Check venue accessibility: 5
       Review child age policies: 4
   section Purchase
       Select family tickets: 5
       Enter payment details: 4
```

```
      Receive confirmation: 5
   section Pre-Event Prep
      Add to family calendar: 5
      Save digital tickets: 4
      Print backup tickets: 5
      Pack family essentials: 4
   section Event Day
      Travel to venue: 3
      Park & navigate to entrance: 4
      Present tickets: 5
      Enter venue: 5
```

Young Event-Goer

```
journey
   title Young Event Goer - Event Ticket Platform Journey
   section Discovery
      Browse events on morning commute: 5: Alex
      Check ticket prices: 4
      Share event with friends: 5
      Coordinate group attendance: 3
   section Purchase
      Select ticket quantity: 5
      Review fees and total: 2
      Complete mobile payment: 4
      Receive digital tickets: 5
   section Pre-Event
      Store tickets in digital wallet: 4
      Share tickets with friends: 3
      Get event reminder: 5
   section Attendance
      Arrive at venue: 4
      Quick QR code scan: 5
      Enter event: 5
   section Post-Event
      Share experience on social: 4
      Follow venue for future events: 5
```

Corporate Networking Attendee

```
journey
   title Corporate Event Attendee - User Journey
   section Discovery
      Search business events: 4: Karim
      Review event details: 5
      Check calendar availability: 4
   section Registration
      Select ticket quantity: 5
      Input team member details: 3
      Corporate payment/billing: 4
      Receive confirmation email: 5
   section Pre-Event
      Add to Outlook calendar: 5
```

```
       Download mobile tickets: 4
       Save event details: 3
   section Event Day
       Access mobile tickets: 5
       Present QR code: 4
       Enter venue: 5
   section Post-Event
       Download receipt: 4
       Submit expense report: 3
```

Event Staff Coordinator

```
journey
   title Event Staff Coordinator - User Journey
   section Pre-Event Setup
       Access event details: 5: Jordan
       Review ticket types/rules: 4
       Configure scanning devices: 3
       Brief staff on procedures: 5
   section Entry Point Setup
       Test scanning equipment: 4
       Position staff members: 5
       Verify radio communications: 4
       Set up backup validation: 3
   section Event Operations
       Monitor entry flow: 5
       Scan attendee tickets: 4
       Handle validation issues: 3
       Coordinate between gates: 4
   section Post-Event
       Generate entry reports: 4
       Debrief with staff: 5
       Document issues: 4
       Submit final counts: 5
```

Entry-Level Event Staff

```
journey
   title Entry-Level Event Staff - Ticket Validation Journey
   section Pre-Event Setup
       Arrive at venue: 4: Event Staff
       Attend briefing: 3
       Test scanning equipment: 4
       Set up entry lanes: 5
   section Entry Rush Preparation
       Review event details: 4
       Position at assigned station: 5
       Open scanning app: 4
       Test scan sample ticket: 3
   section Main Entry Period
       Greet attendee: 5
       Request ticket: 4
```

```
      Scan QR code: 3
      Verify ticket status: 4
      Direct to entrance: 5
  section Issue Resolution
      Identify scan problems: 2
      Try manual entry: 3
      Escalate to supervisor: 2
      Document issues: 3
  section End of Shift
      Record final statistics: 4
      Report technical issues: 3
      Clean up station: 5
      Complete shift log: 4
```

Summary

- Explored user journeys for organizers, attendees and staff
- The user journeys offer insight into how different users may interact with the system
- The user journeys will influence how the system is designed and implemented

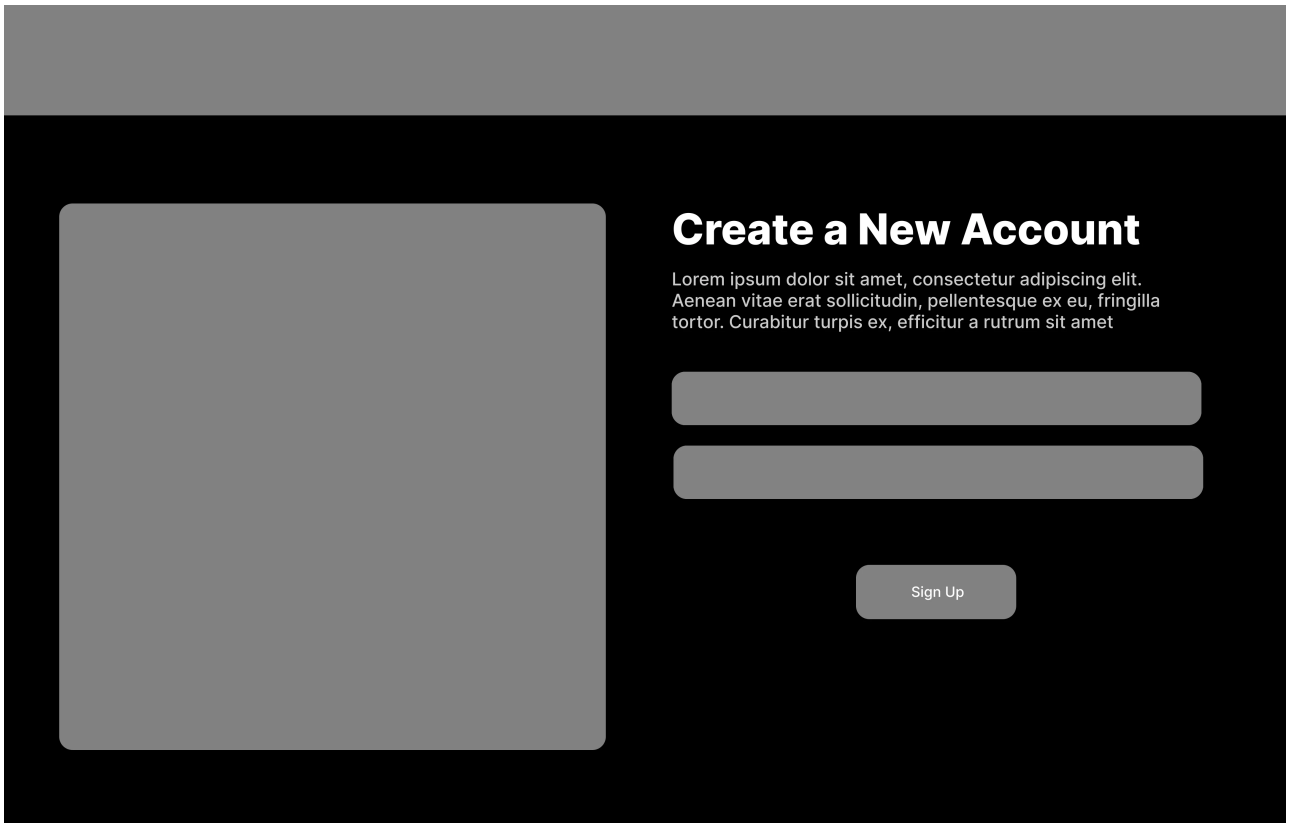
UI Design

Let's explore the initial design of the user interface.

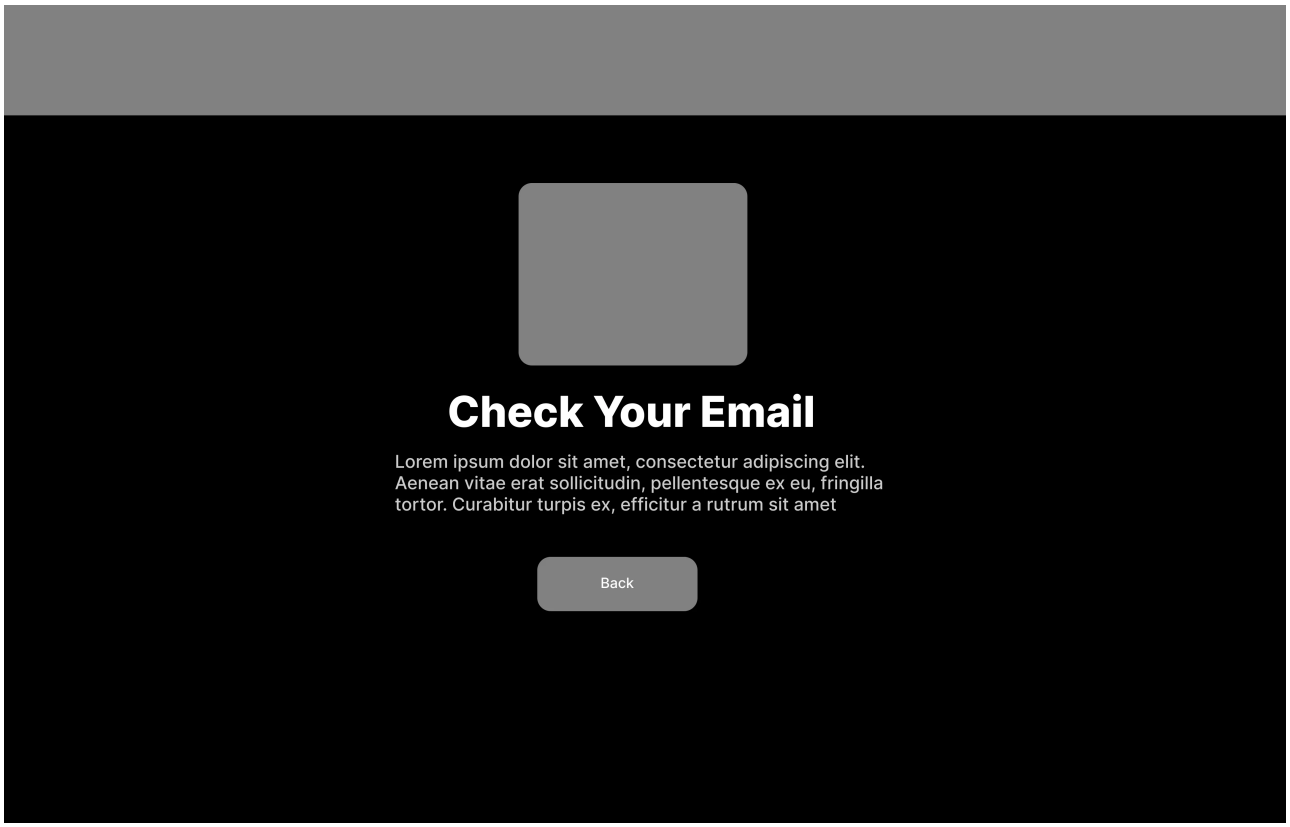
Sign up and Logging

All users will need to sign up and log in, although we may not go as far as implementing a custom sign up and log in page, I've included them for completeness.

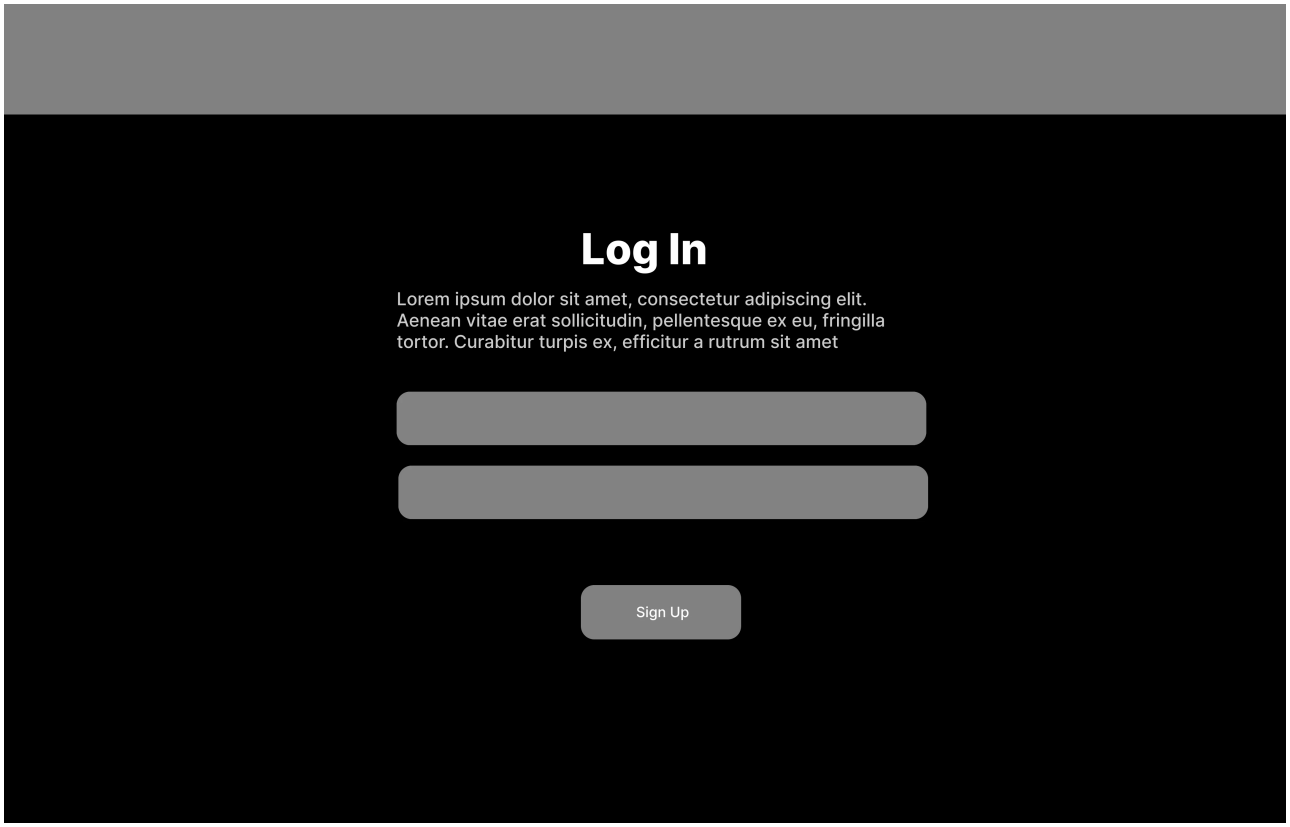
Sign Up Page



Sign Up Confirmation Page



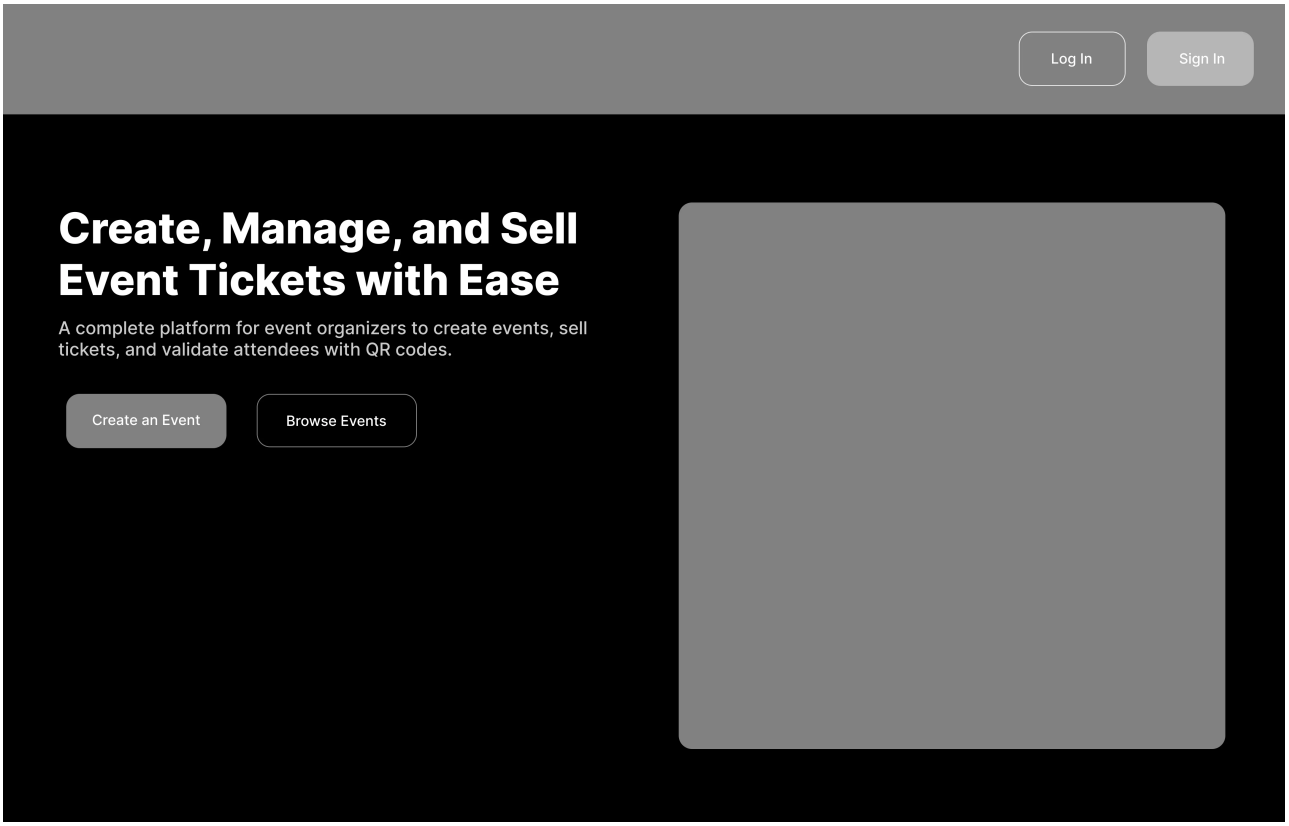
Log In Page



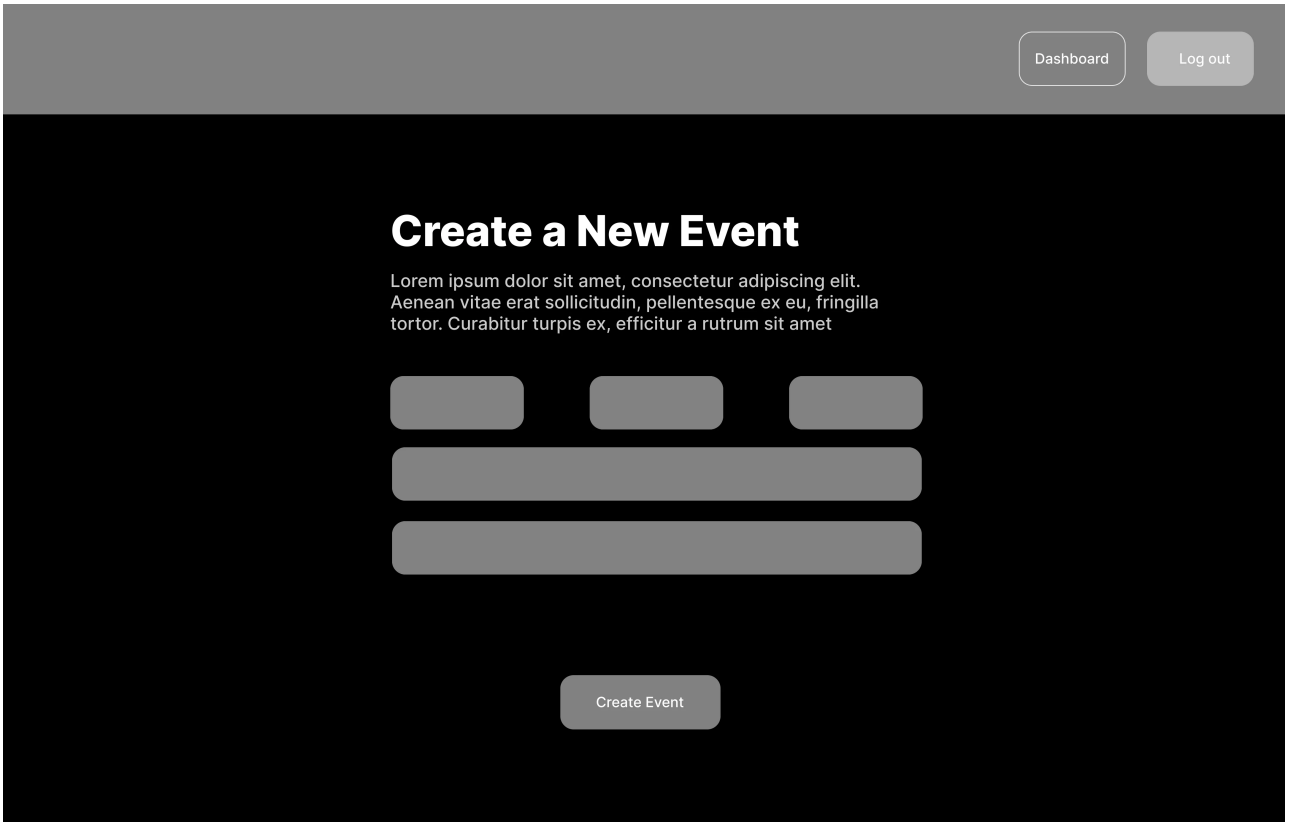
Organizer Flow

Here's how I imagined an organizer user type to interact with our application.

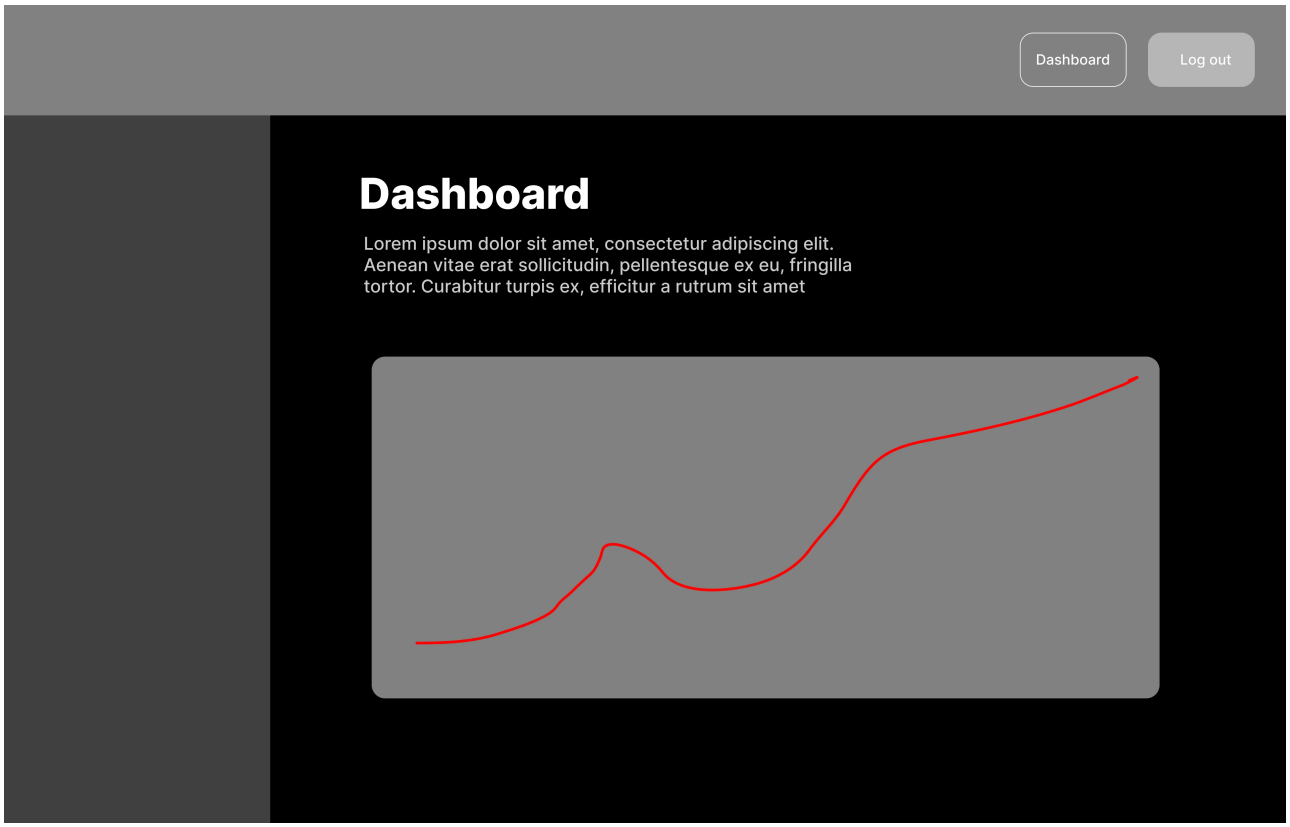
Organizer Landing Page



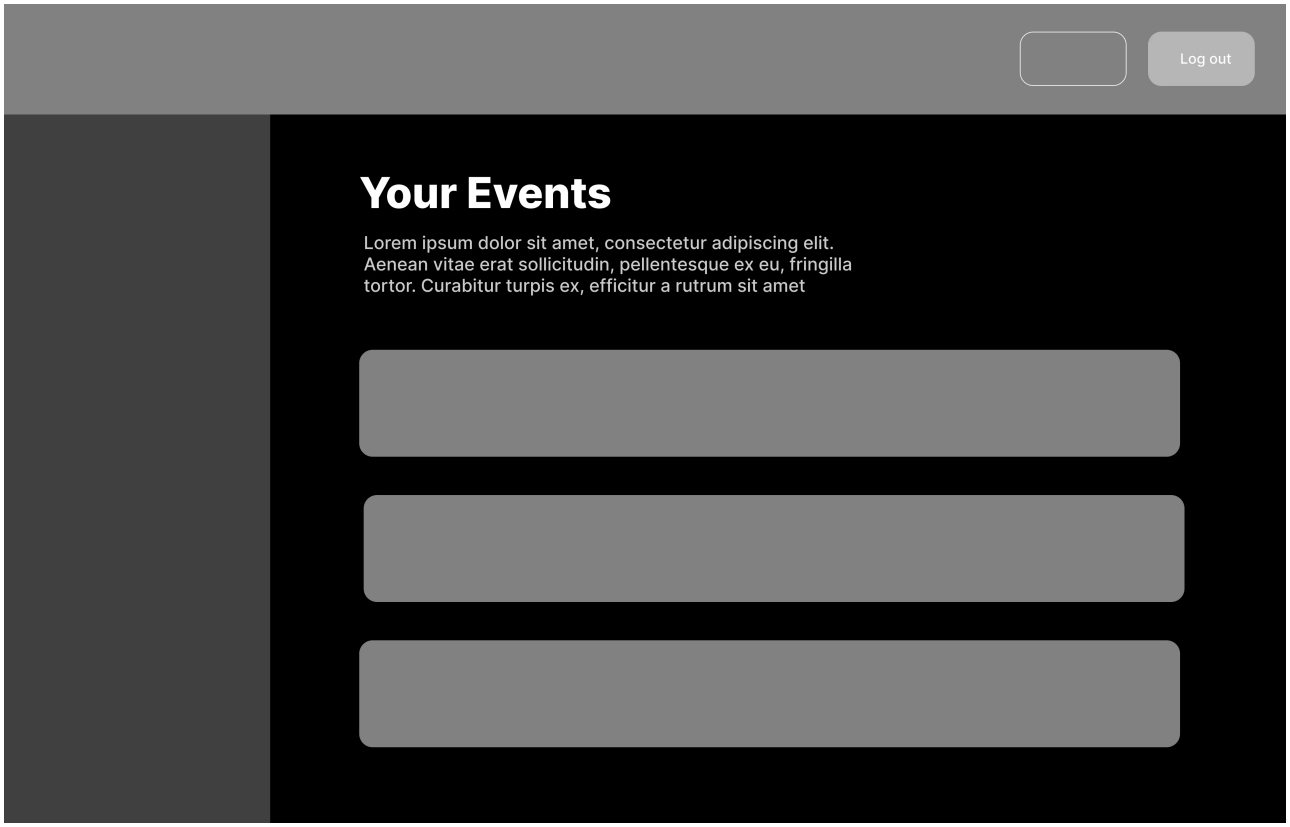
Create & Edit Event Page



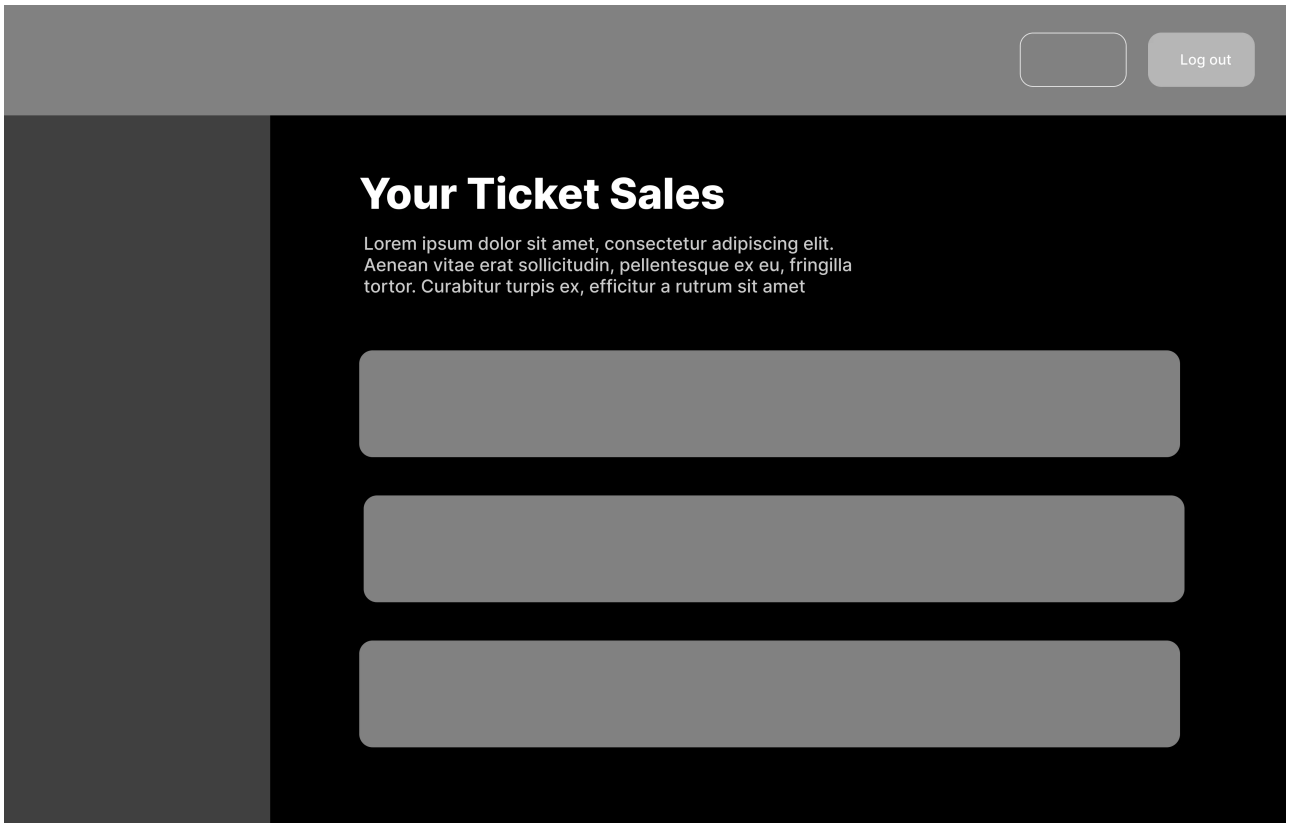
Dashboard Landing Page



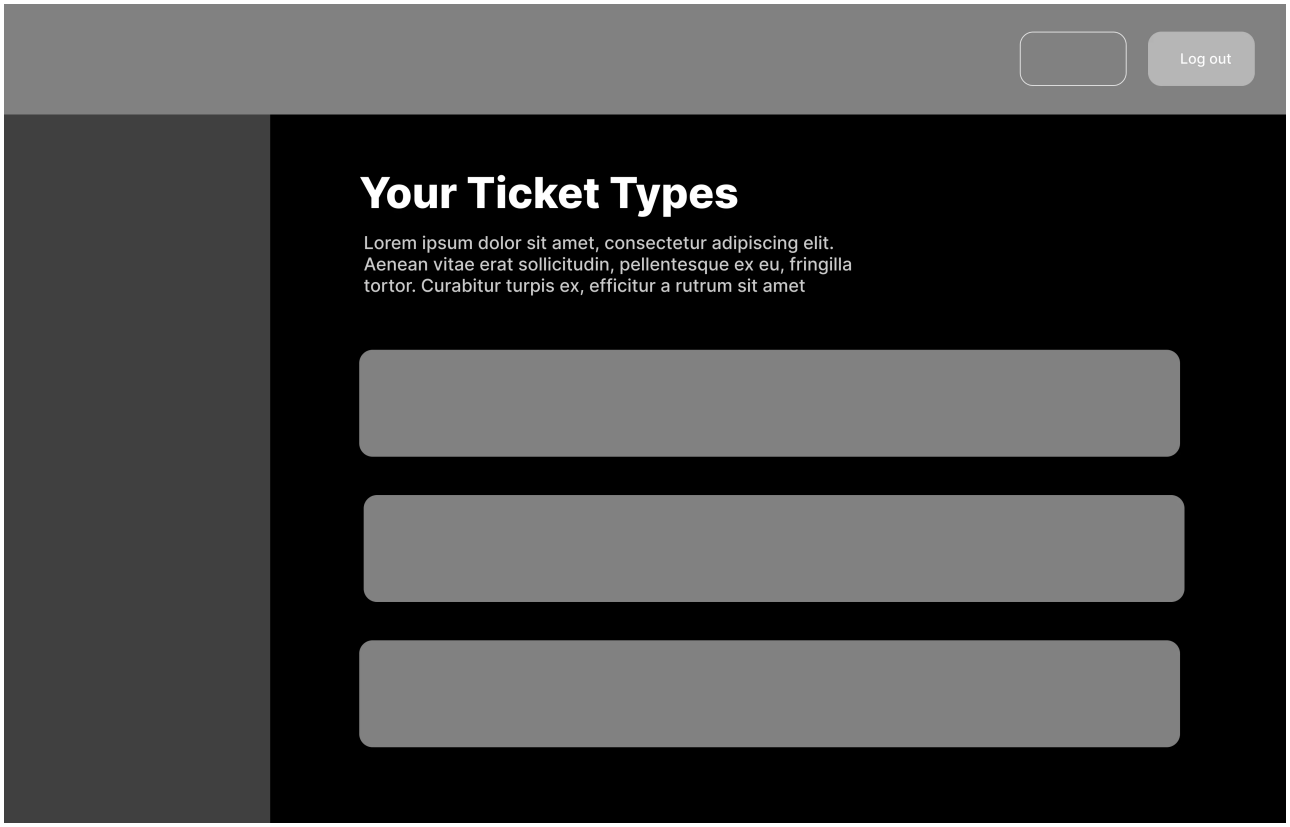
Dashboard Events Page



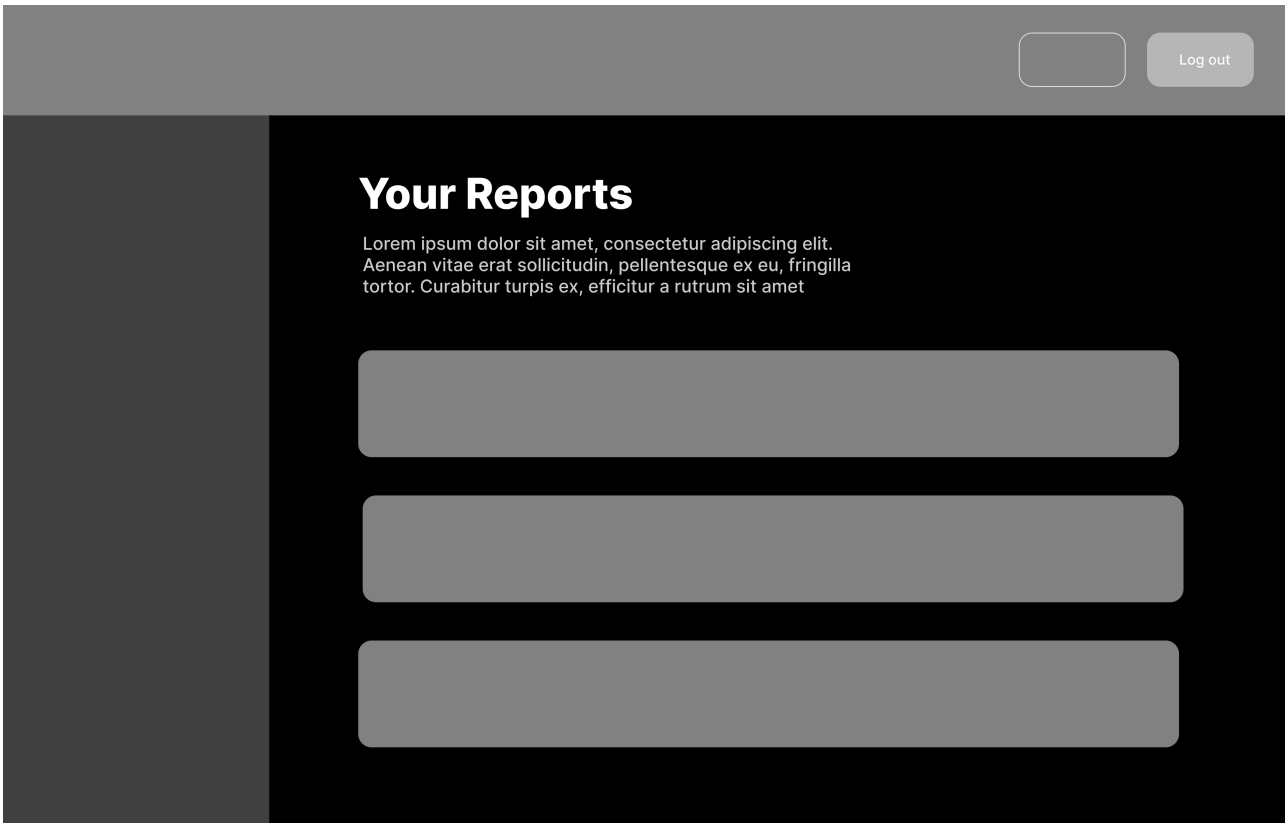
Dashboard Ticket Sales Page



Dashboard Ticket Types Page



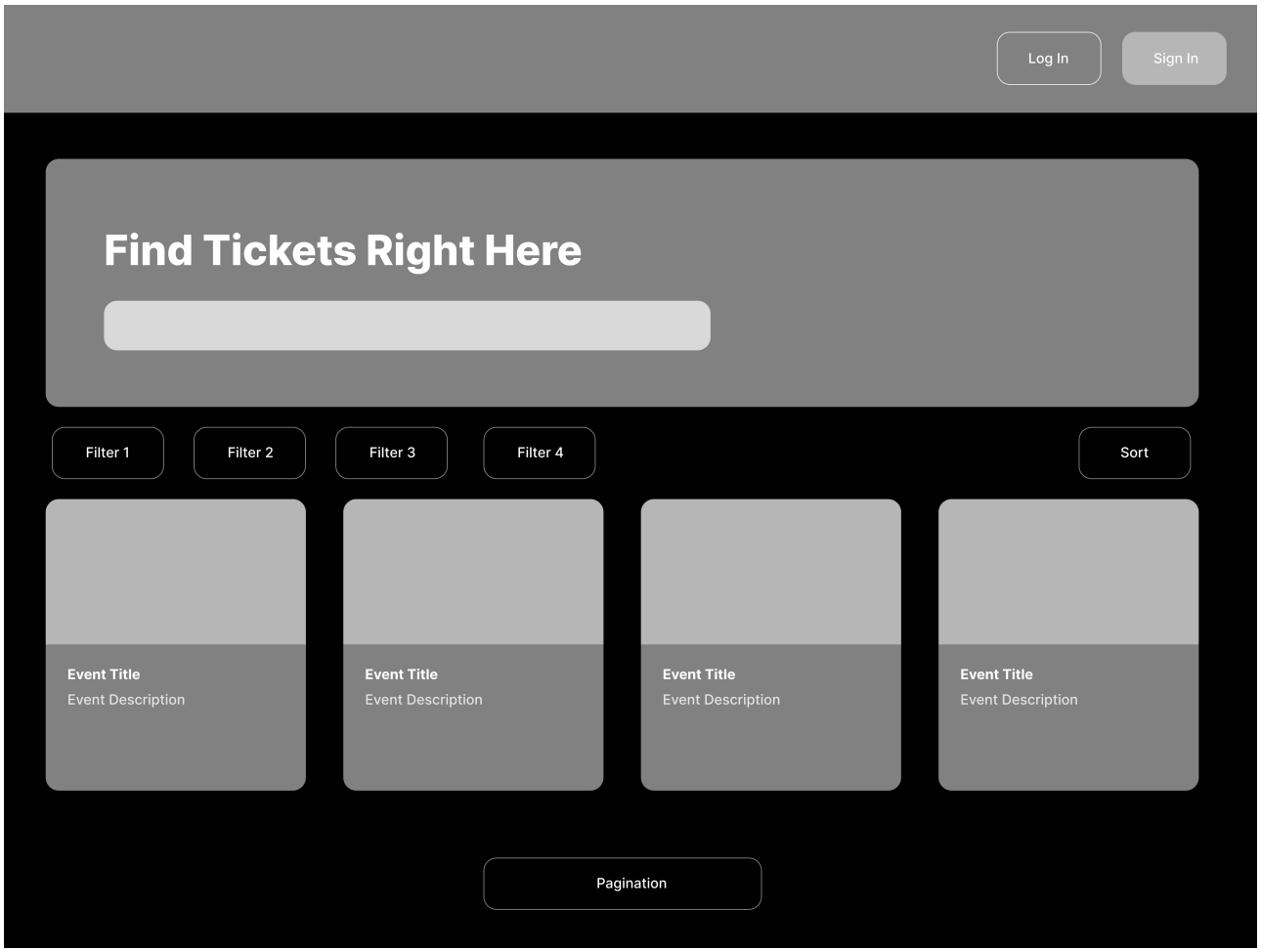
Dashboard Reports Page



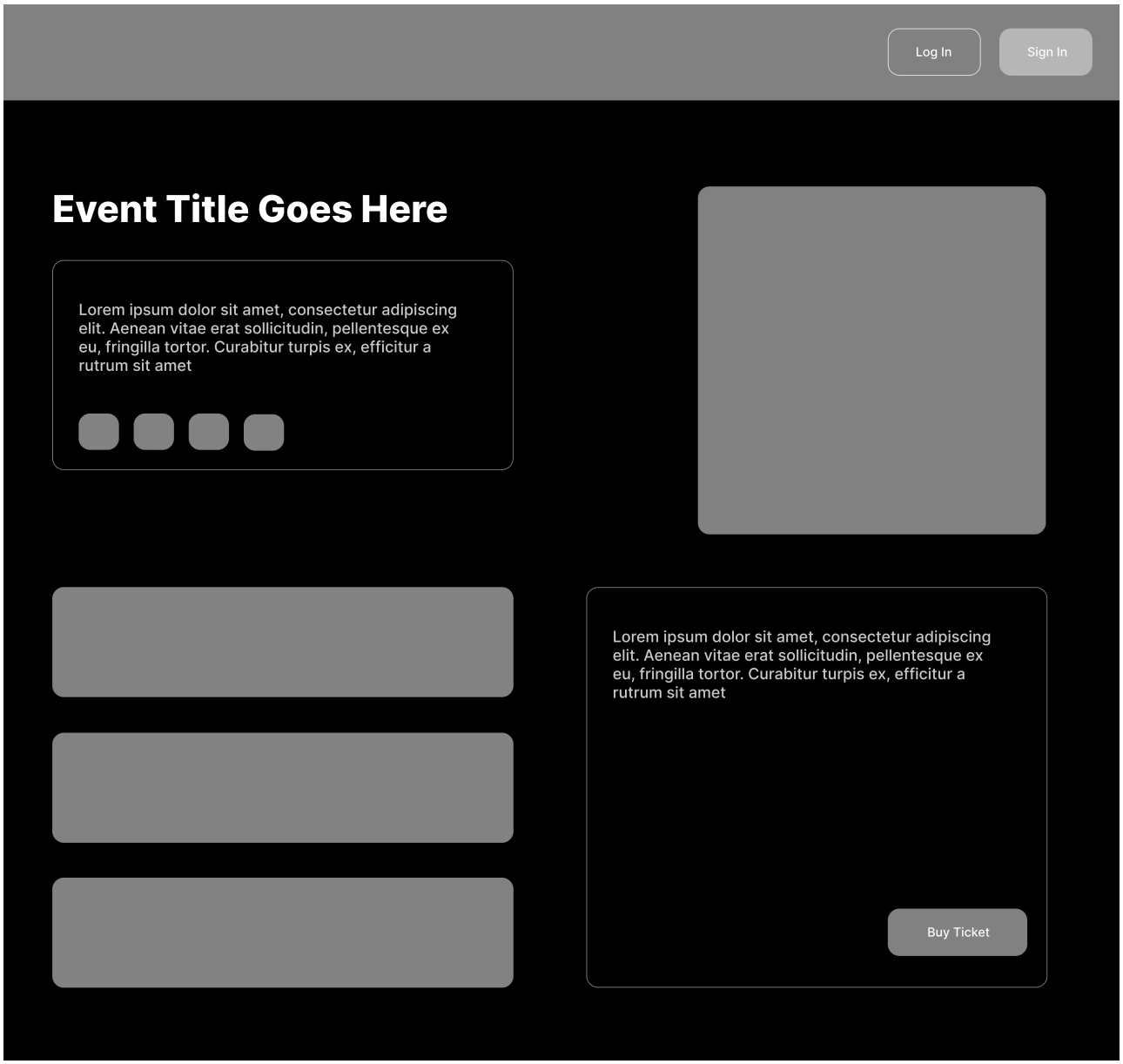
Attendee Flow

Here's how I imagine attendees interacting with our system:

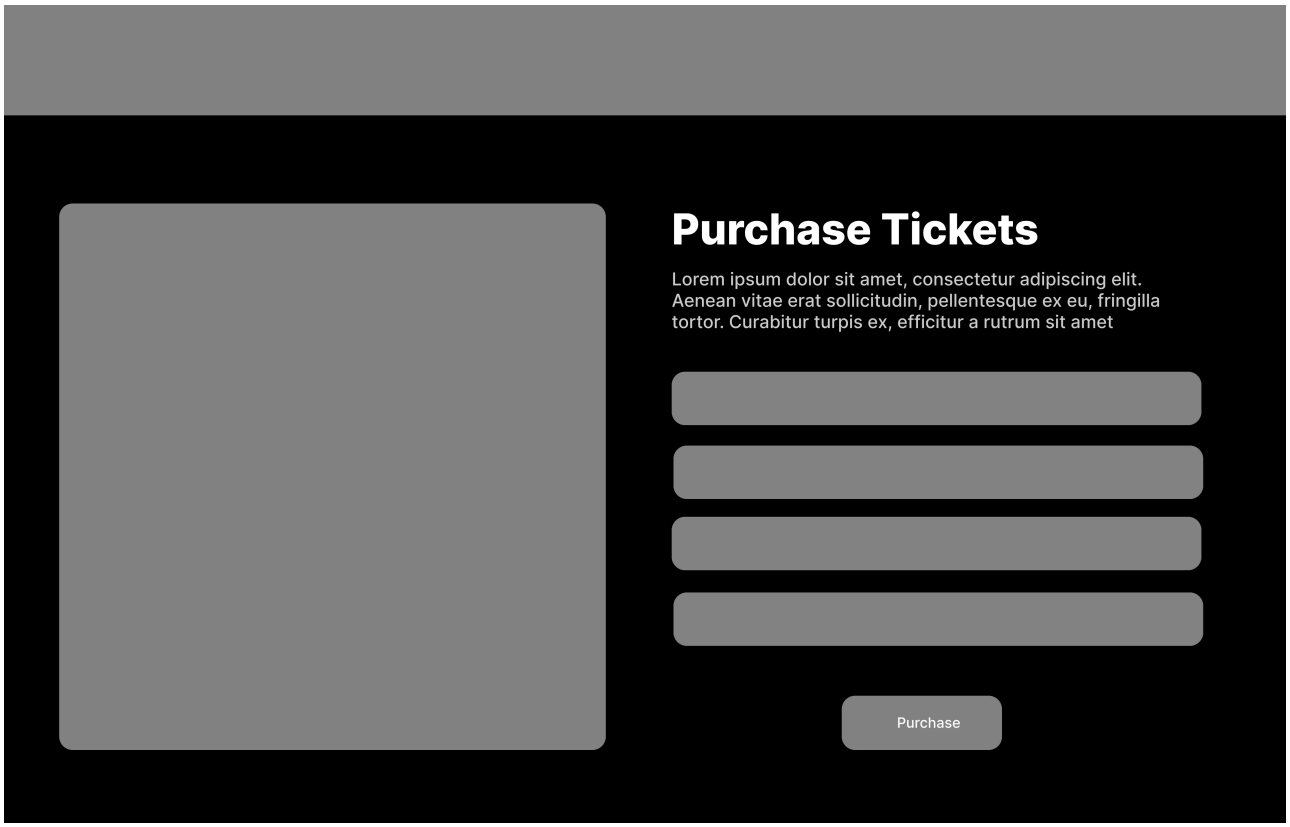
Attendee Landing Page



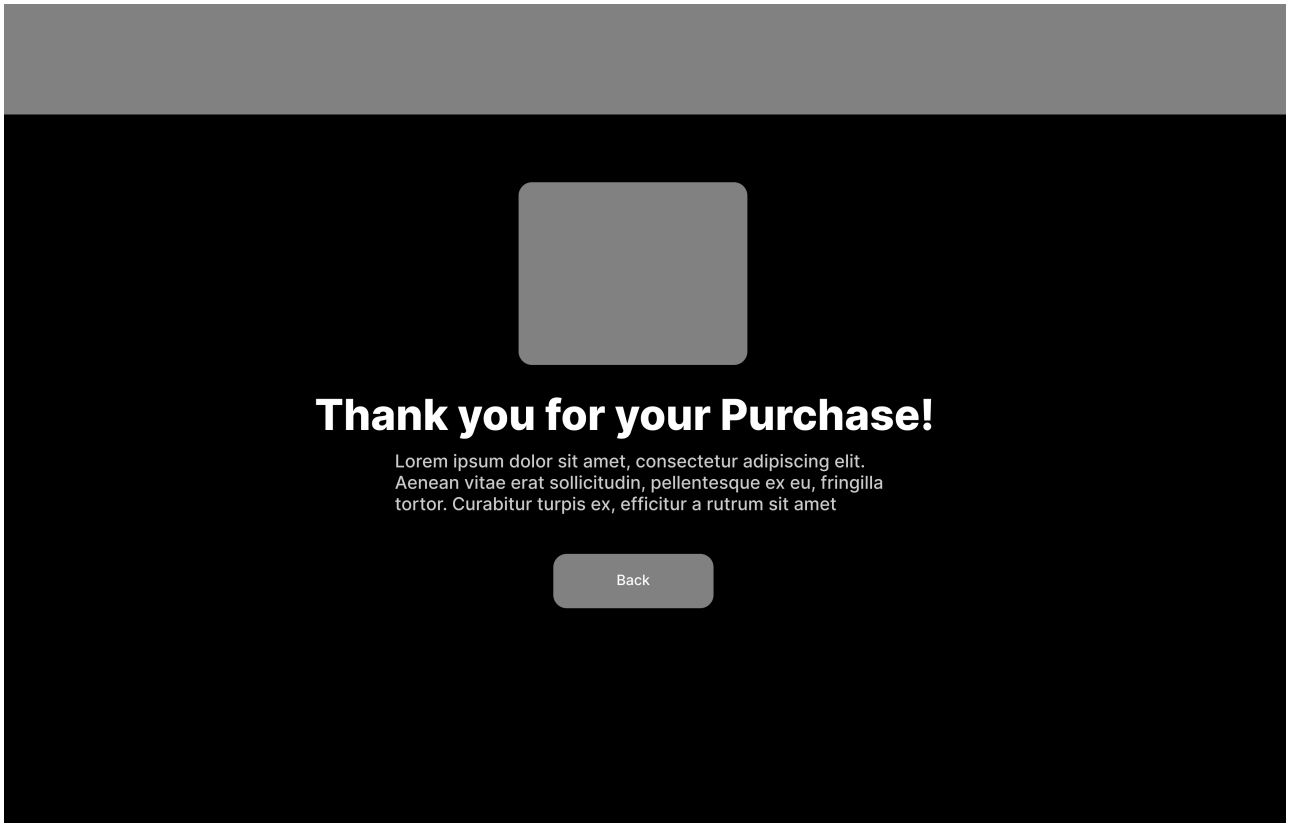
Event Details Page



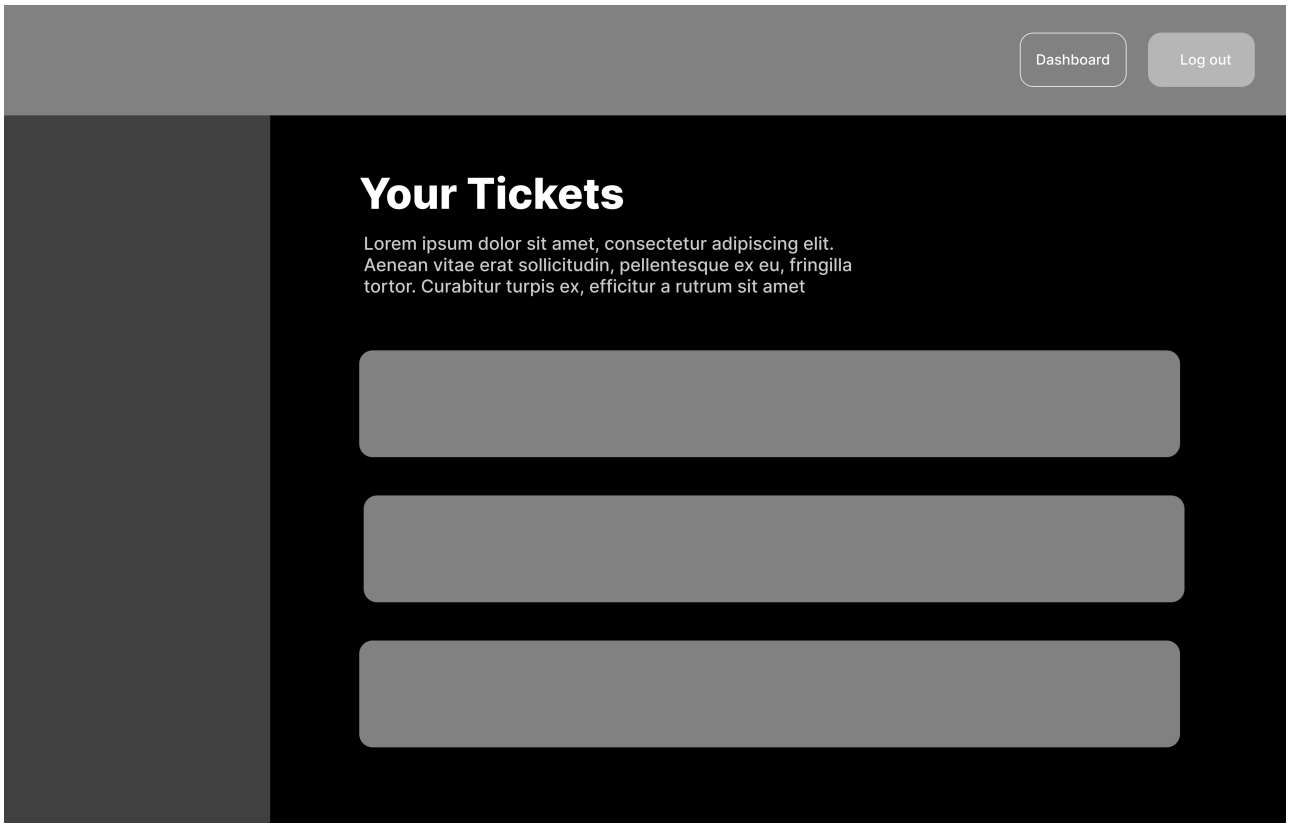
Purchase Tickets Page



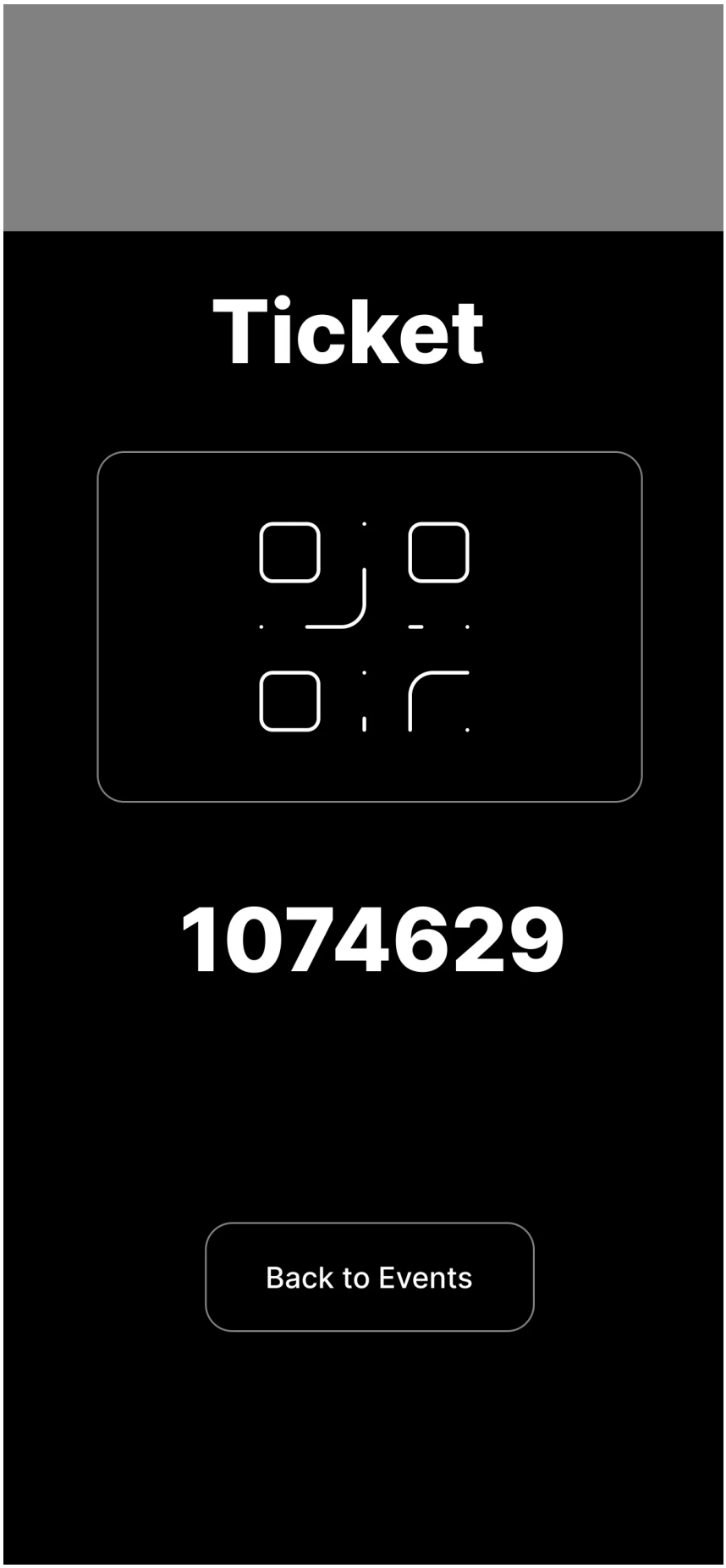
Purchase Tickets Confirmation Page



Dashboard Purchased Tickets



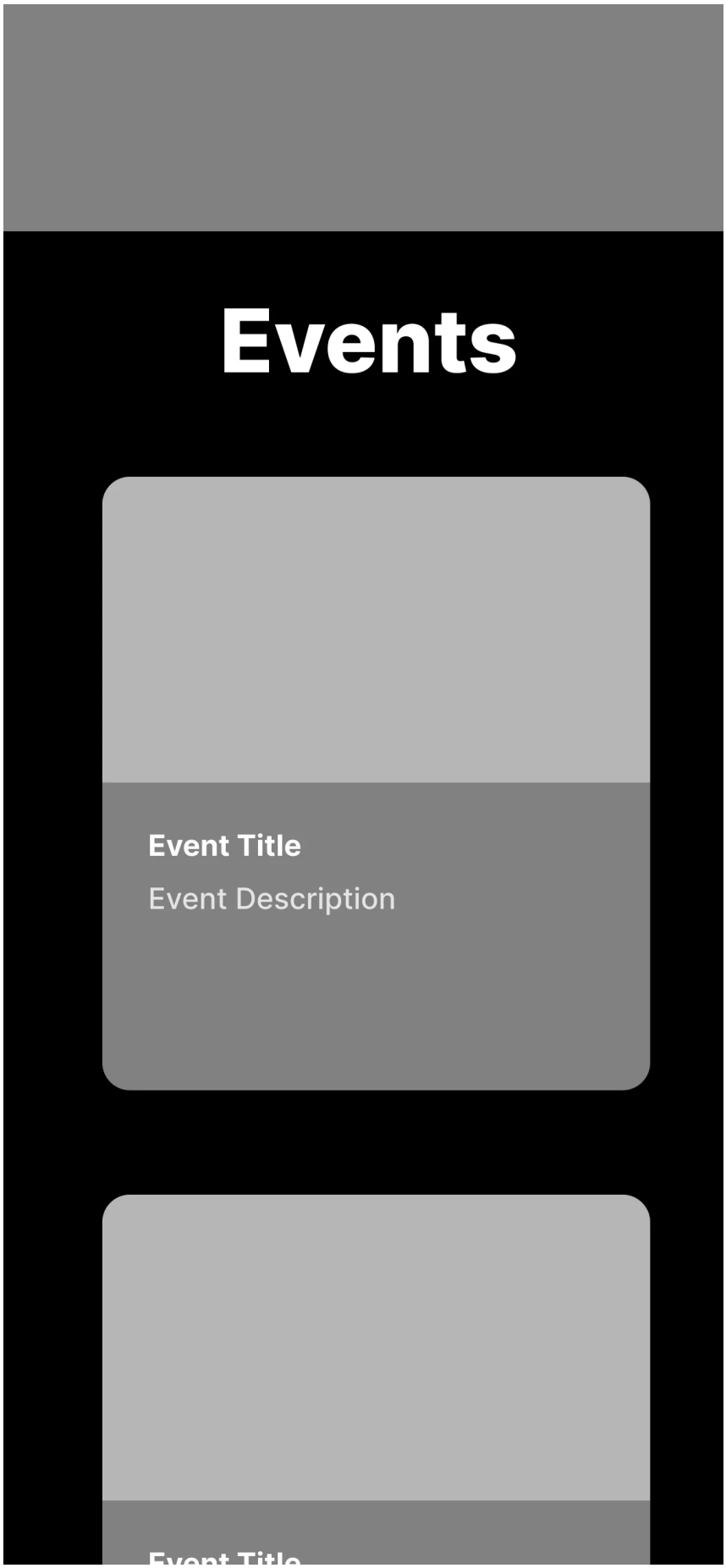
Ticket QR Code Page



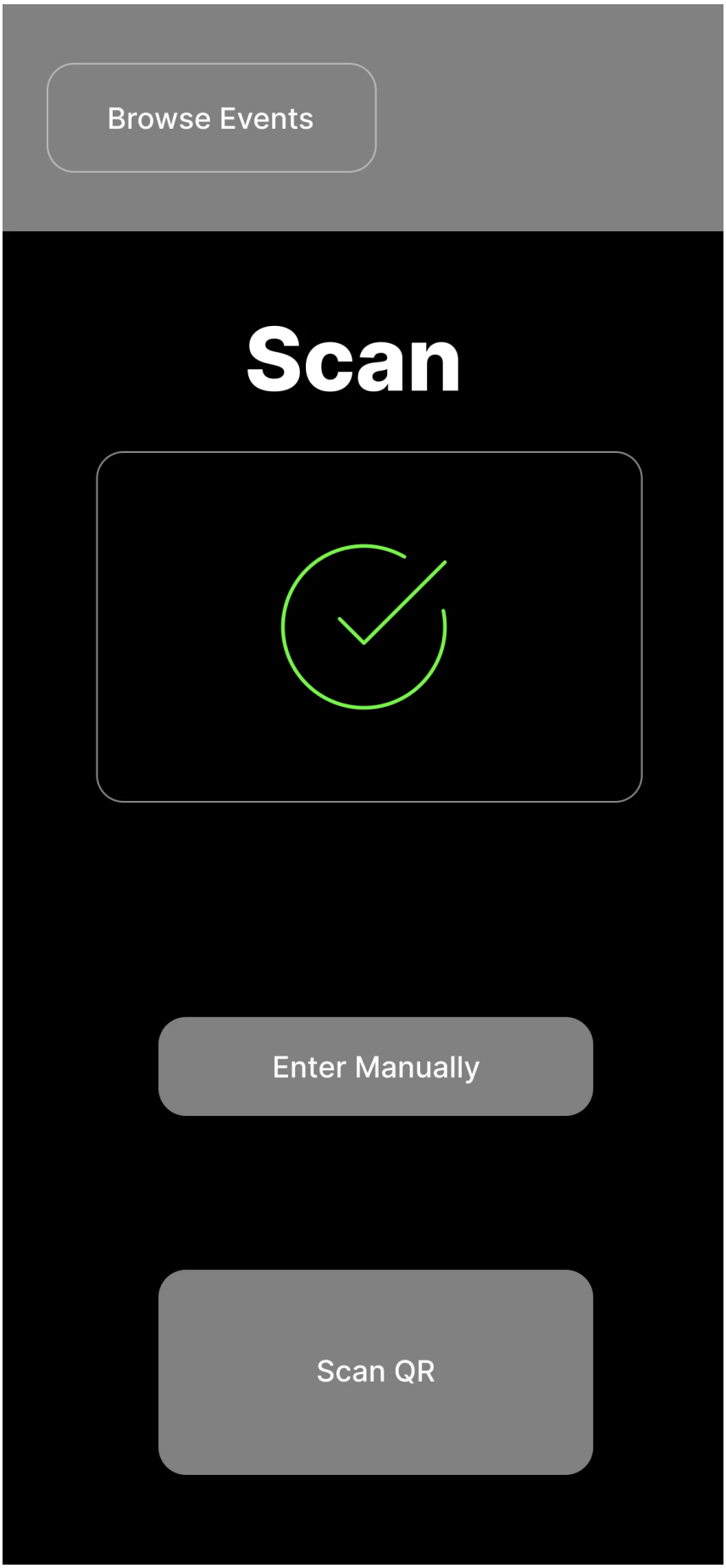
Staff Flow

Finally, these are the pages I imagine the staff using:

Select Event Page



Ticket Scanning Page



Summary

- Designed the first iteration of the user interface
- Designs have been influenced by user stories, personas, and journeys

Module Summary

Key Concepts Covered

User Personas Summary

- Explored user personas representing organizers, attendees and staff
- These user personas will influence how the system is designed

User Journey

- Explored user journeys for organizers, attendees and staff
- The user journeys offer insight into how different users may interact with the system
- The user journeys will influence how the system is designed and implemented

UI Design

- Designed the first iteration of the user interface
- Designs have been influenced by user stories, personas, and journeys

Domain Modelling

Module Overview

In this module, we'll create a class diagram that represents the ticket platform's domain, giving us the information we need to design our application's REST API.

Module Structure

1. Establish the domain model
2. Add cardinality information to the class diagram
3. Add data types to the class diagram
4. Define each variable as required / optional on the class diagram

Learning Objectives

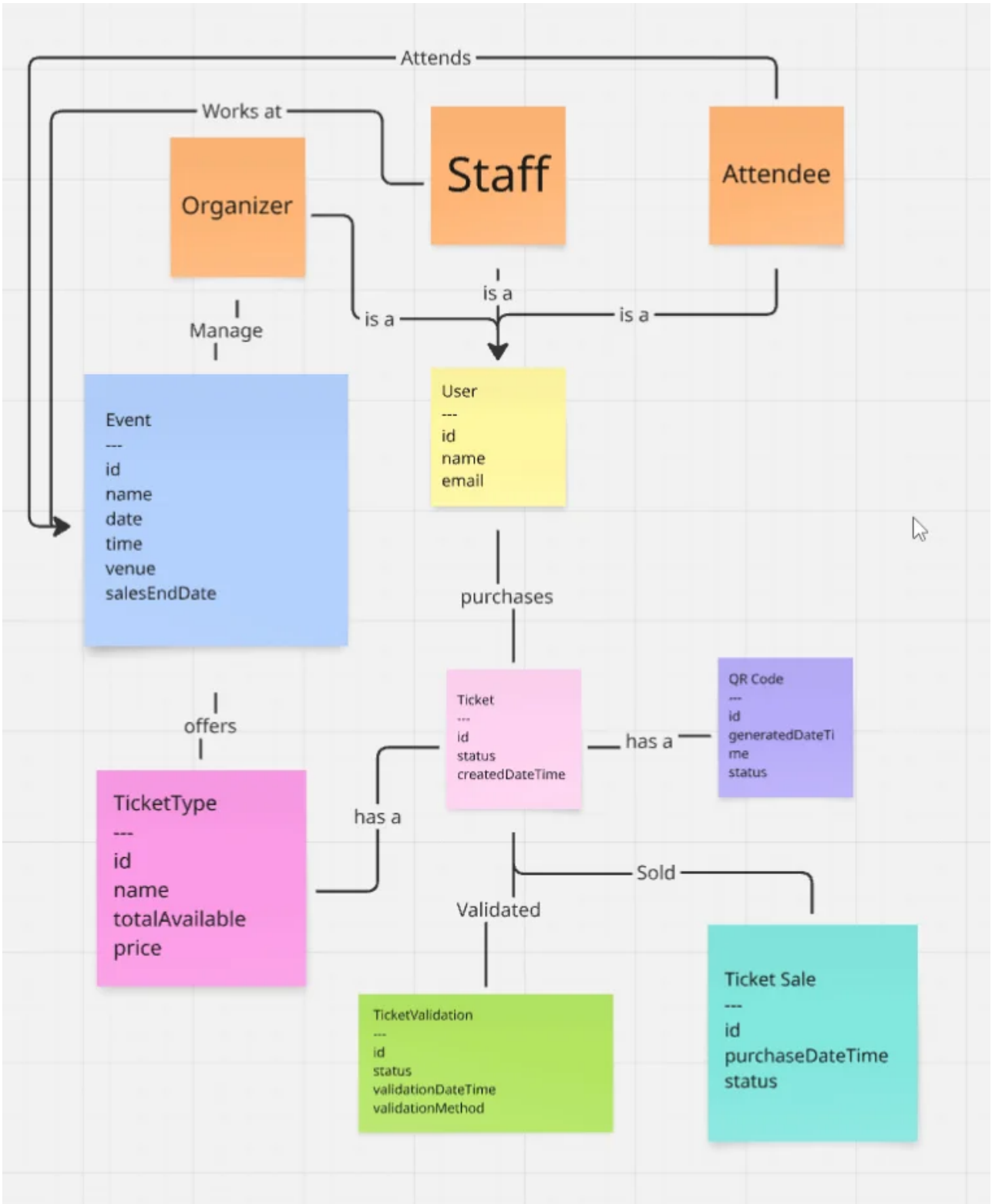
1. Describe the domain represented by the application's domain diagram
2. Add cardinality information to the application's class diagram
3. Add data type information to the application's class diagram
4. Add required and optional information to the application's class diagram

Domain Model Summary

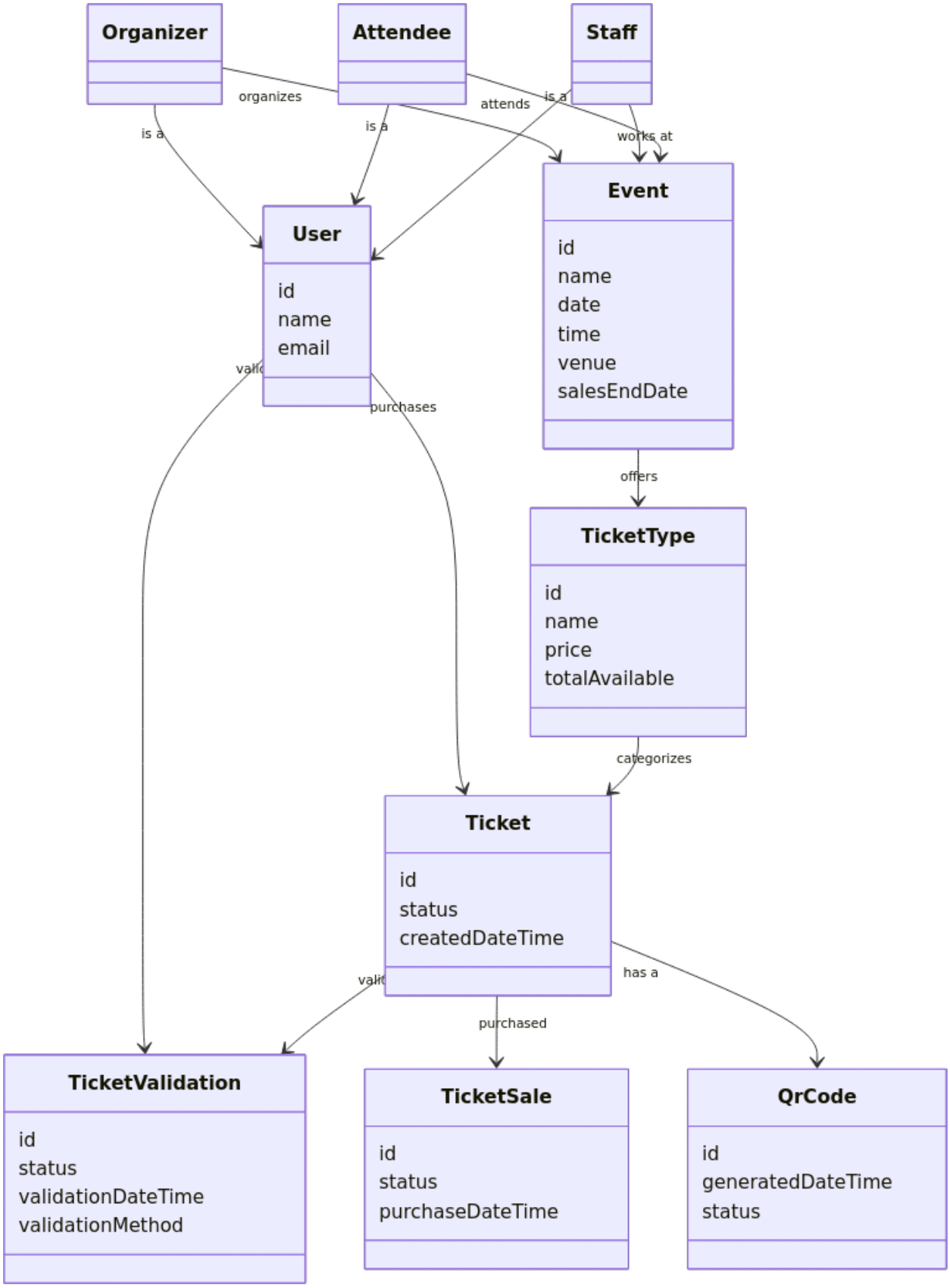
Let's explore the initial domain model described in the project brief.

Review the Domain Diagram

After spending a bit of time analyzing the project brief, using a variation on the "noun-verb analysis" technique, I produced the following domain diagram:



Or if you prefer the MermaidJs version:



Explore the Initial Domain

From the domain model, I've identified the following domain objects.

Note that this is just our initial understanding, which we will refine over time.

Event

The `Event` class represents a planned gathering with properties like:

- `id` - A unique identifier
- `name` - The event's title
- `date` and `time` - When the event occurs
- `venue` - Where the event takes place
- `salesEndDate` - When ticket sales stop

Ticket Type

The `TicketType` class defines different categories of tickets available for an event:

- `id` - A unique identifier
- `name` - The type of ticket (e.g., "VIP", "Standard")
- `price` - Cost of the ticket
- `totalAvailable` - Maximum number that can be sold

Ticket

The `Ticket` represents an individual purchase:

- `id` - A unique identifier
- `status` - The status of the ticket, perhaps it's been cancelled?
- `createdDateTime` - The date and time the ticket was created

QR Code

The `QrCode` represents the code used to represent the ticket's information, present on each ticket:

- `id` - A unique identifier
- `generatedDateTime` - The time and date the QR code was generated
- `status` - The status of the QR code -- is it still valid?

User

The `User` class represents people interacting with our system, where `Organizer`, `Attendee` and `Staff` are different types of `User`:

- `id` - A unique identifier
- `name` - Person's name
- `email` - Contact information

Ticket Validation

Finally, `TicketValidation` is for event entry management:

- `id` - A unique identifier
- `validationTime` - When validation occurred
- `validationMethod` - How it was validated
- `status` - Result of validation

We'll evolve this domain diagram further as our design progresses.

Summary

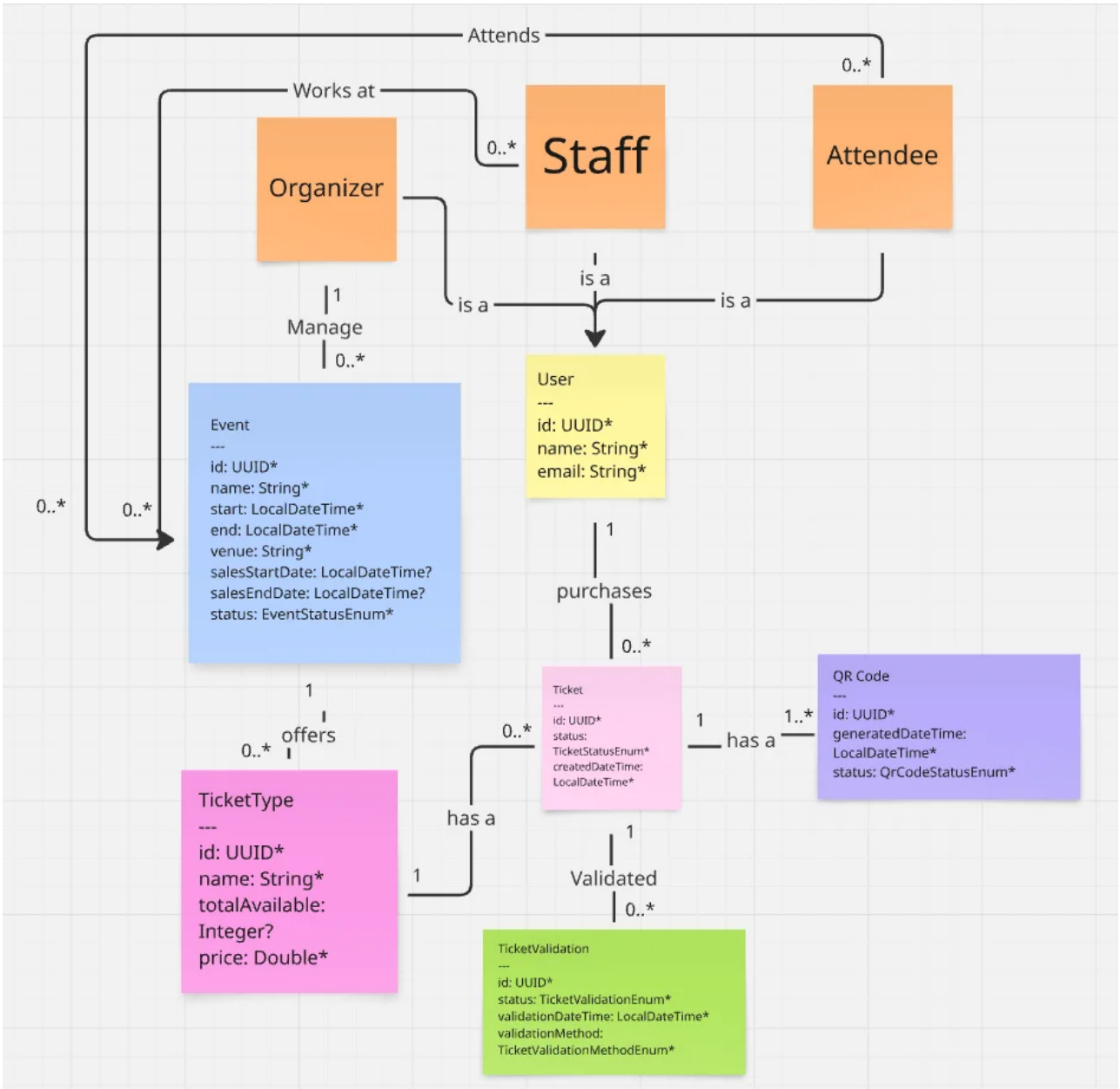
- Events can have multiple ticket types, each with their own pricing and availability
- Users can act as organizers, event goers, and staff
- Tickets include QR codes for validation at event entry
- Ticket validation ensures proper event access control

Domain Modelling - Cardinality

Let's now add cardinality information to the class diagram.

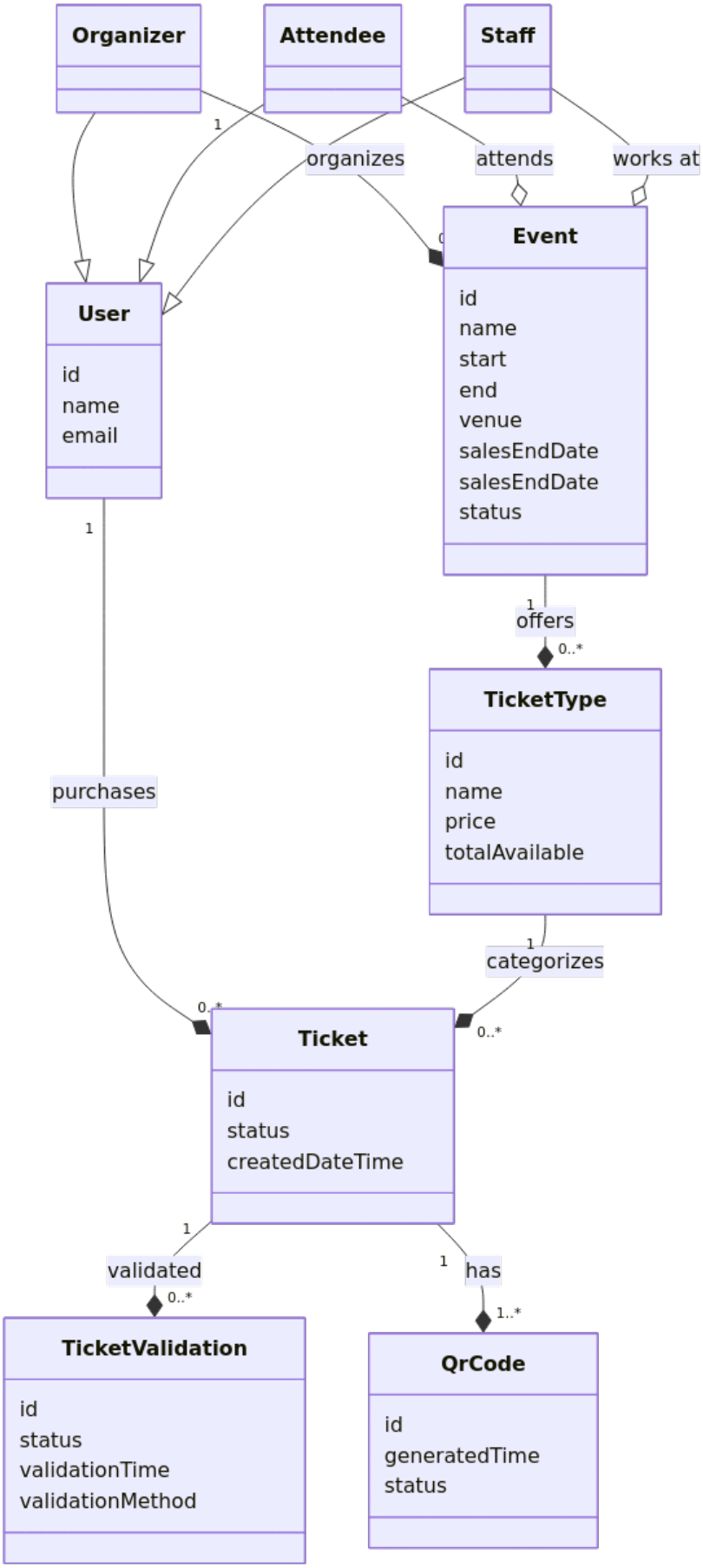
Add Cardinality to the Class Diagram

Here's what I came to in Miro:



Formalize the Class Diagram

Here's the class diagram formalized as MermaidJs:



Summary

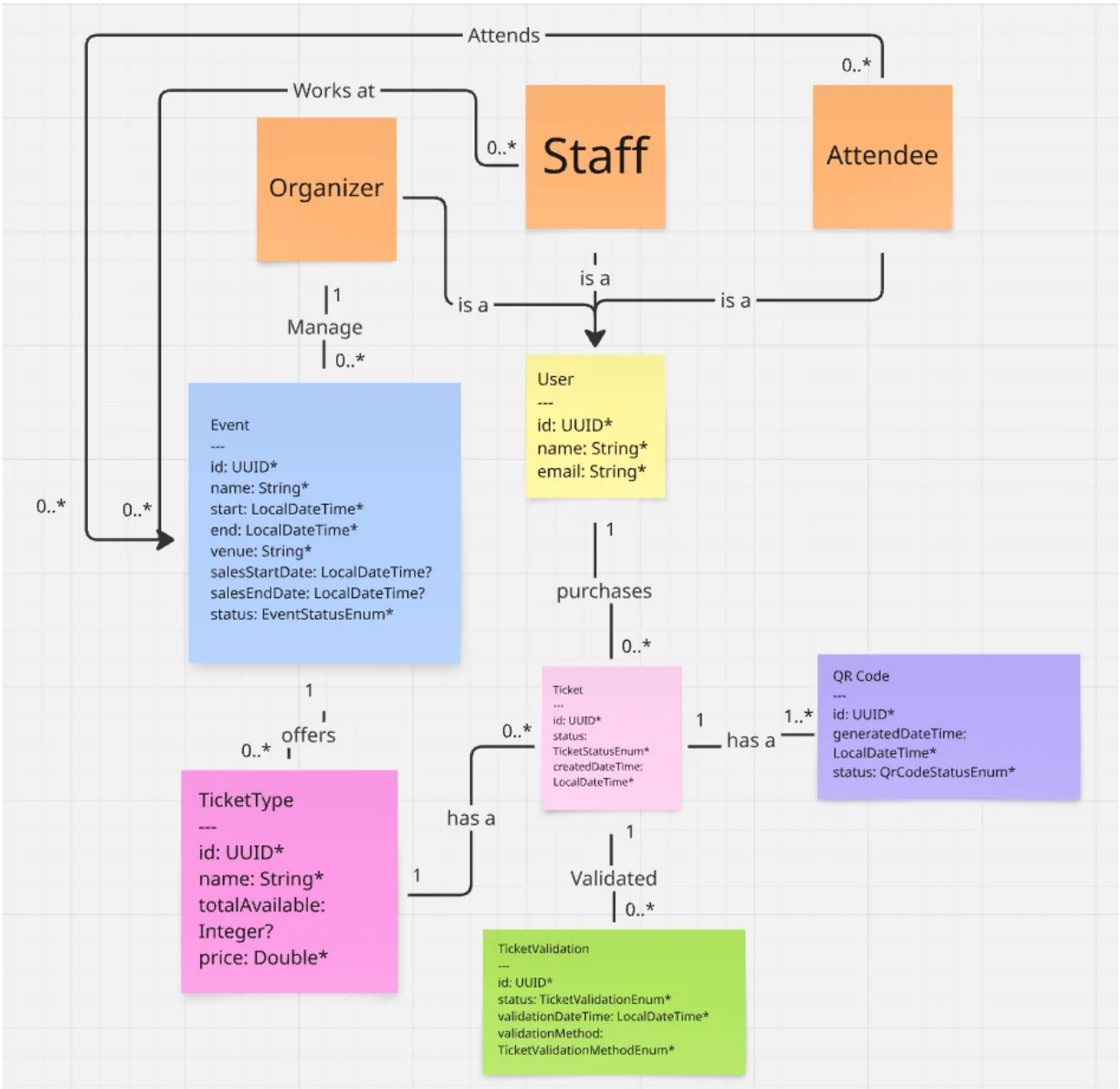
- Added cardinality information to the application's class diagram

Data Modelling - Data Types

Let's now add type information to our class diagram.

Add Type Information to the Class Diagram

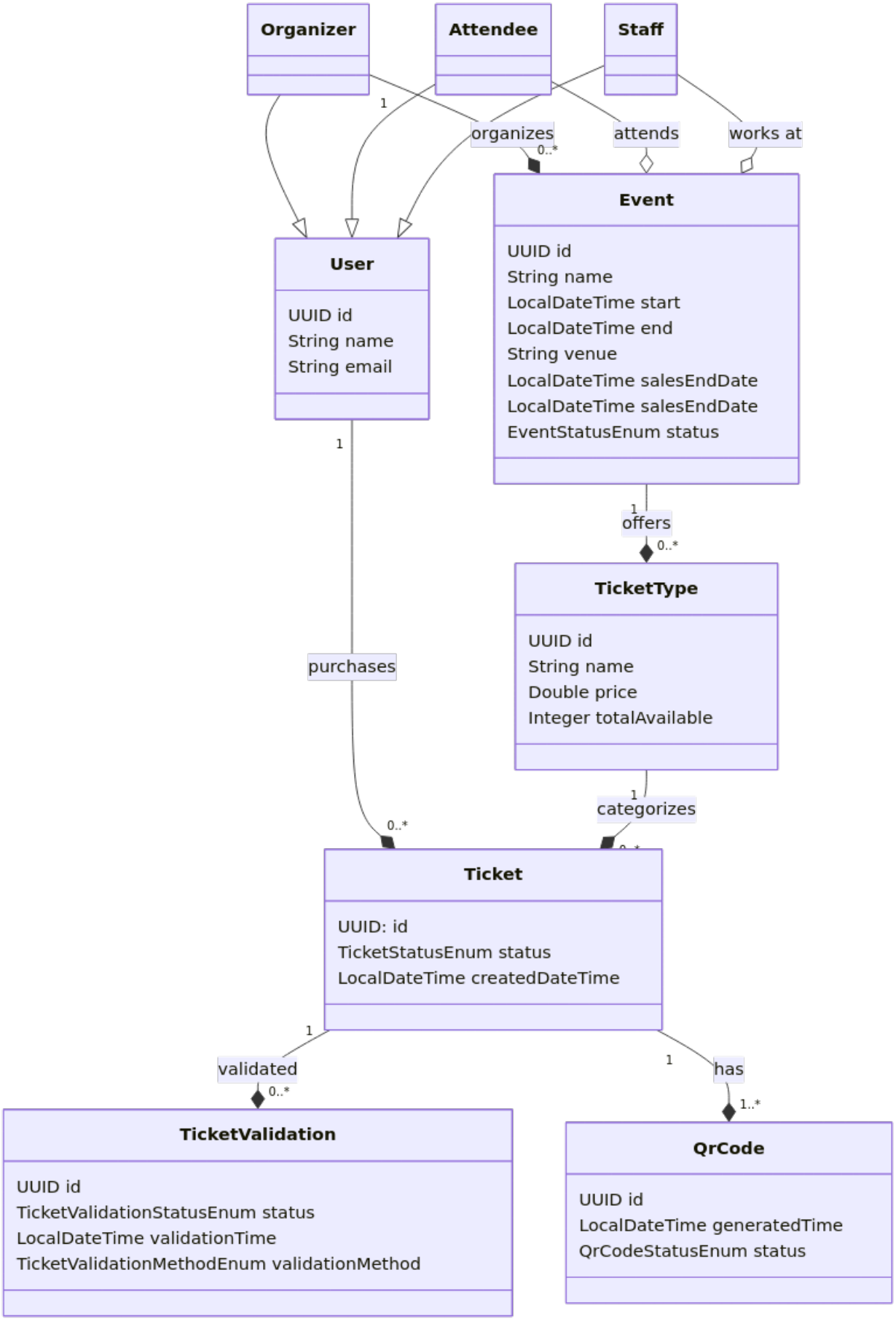
Here's what I came to in Miro:



Formalize the Class Diagram

Here's the class diagram formalized as MermaidJs.

I've went with the Java-style way of declaring types in the formalized version:



Summary

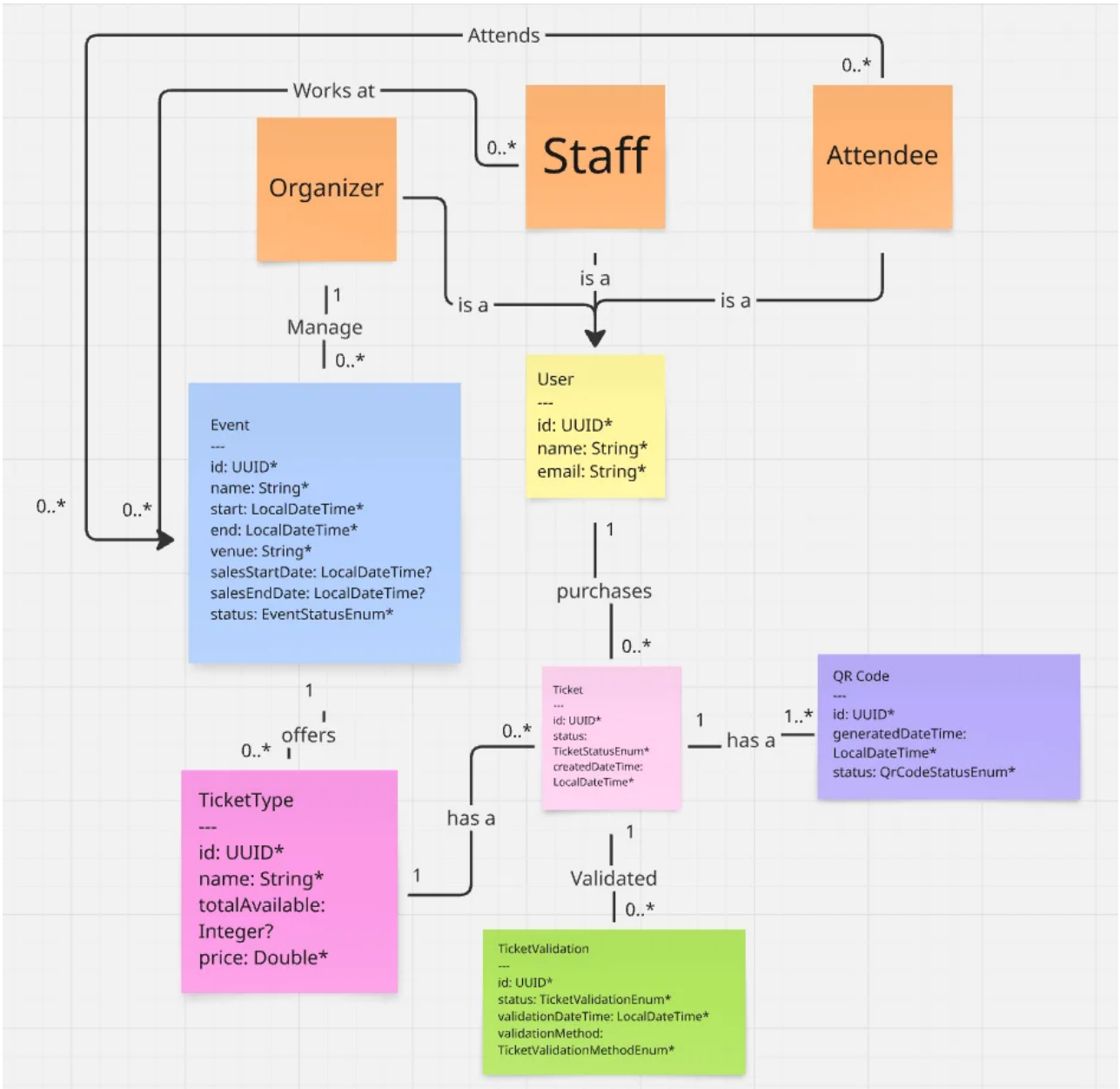
- Added data type information to the application's class diagram

Data Modelling - Required & Optional

Let's now define each property as either required or optional.

Add Required/Optional Information to the Class Diagram

Here's what I came to in Miro:

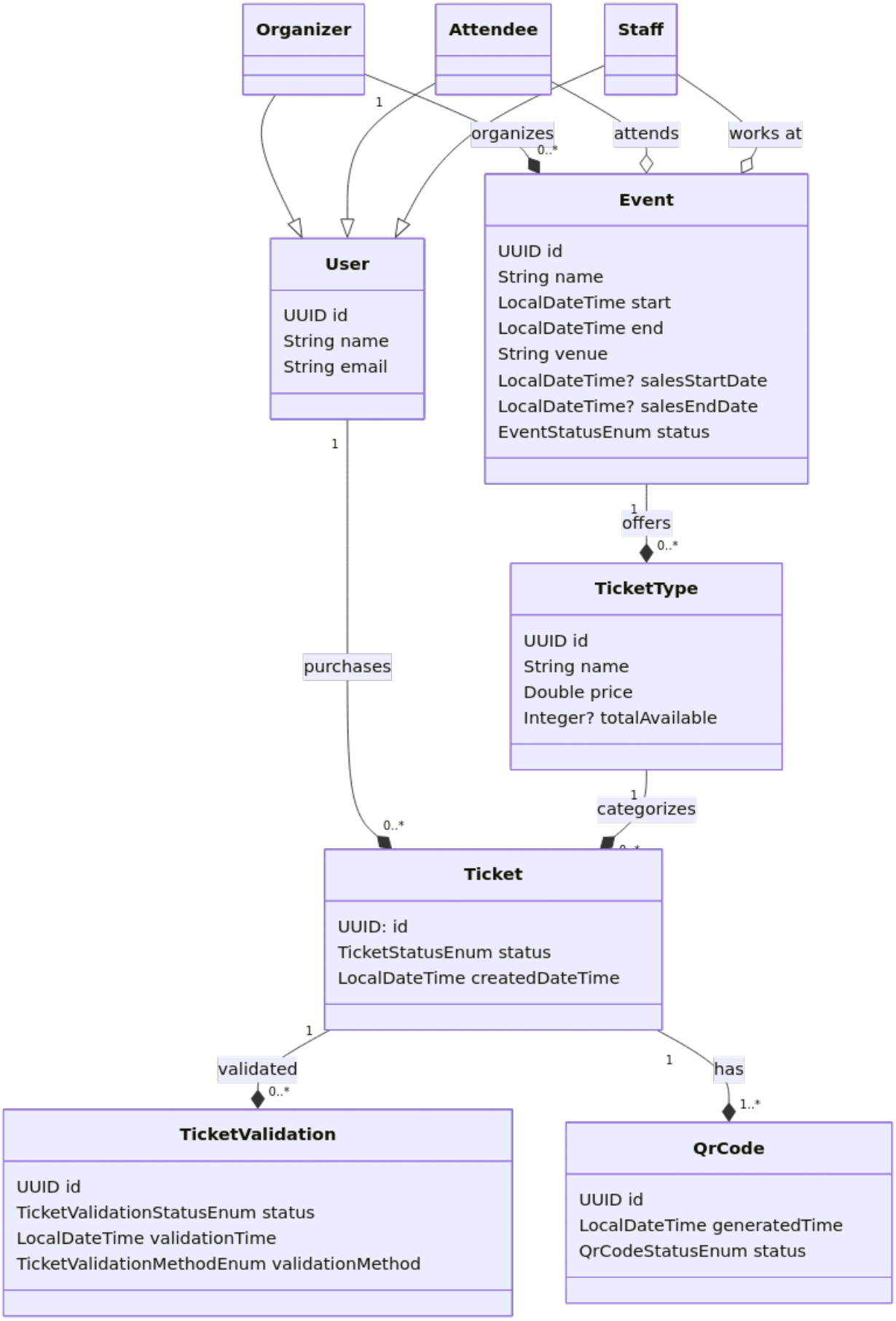


Formalize the Class Diagram

Here's the class diagram formalized as MermaidJs.

Here I've assumed that all properties are required, unless they have a question mark `?` suffix. e.g. `String` is required, where's `String?` is optional.

This saves us writing a whole bunch of asterisks in our class diagram.



Summary

- Added optional / required information to the application's class diagram

Module Summary

Key Concepts Covered

Domain Model Summary

- Events can have multiple ticket types, each with their own pricing and availability
- Users can act as organizers, event goers, and staff
- Tickets include QR codes for validation at event entry
- Ticket validation ensures proper event access control

Domain Modelling - Cardinality

- Added cardinality information to the application's class diagram

Data Modelling - Data Types

- Added data type information to the application's class diagram

Data Modelling - Required & Optional

- Added optional / required information to the application's class diagram

System Design

Module Overview

In this module we'll design the REST API and architecture for our event ticket management platform.

This design work will serve as our blueprint for implementation.

Module Structure

1. Design the event organizer's REST API endpoints
2. Design the event attendee's REST API endpoints
3. Design the event staff's REST API endpoints

4) Design the system architecture

Learning Objectives

1. Design REST API endpoints for event organizers to create and manage events
2. Design REST API endpoints for attendees to browse events and purchase tickets
3. Design REST API endpoints for staff to browse events and validate tickets
4. Design the initial event ticket platform system architecture

REST API Design - Organizer Flow

Let's analyze the organizer user interface flow in order to identify the REST API endpoints.

Analyze the Organizer Flow

Here's the REST API endpoints I've identified from the first round of analysis:

```
## Create Event
POST /api/v1/events
Request Body: Event

## List Events
GET /api/v1/events

## Retrieve Event
GET /api/v1/events/{event_id}

## Update Event
PUT /api/v1/events/{event_id}
Request Body: Event

## Delete Event
DELETE /api/v1/events/{event_id}

## List Ticket Sales
GET /api/v1/events/{event_id}/tickets

## Retrieve Ticket Sale
GET /api/v1/events/{event_id}/tickets/tickets/{ticket_id}

## Partial Update
PATCH /api/v1/events/{event_id}/tickets
Request Body: Partial Event

## List Ticket Type
GET /api/v1/events/{event_id}/ticket-types

## Retrieve Ticket Type
GET /api/v1/events/{event_id}/ticket-types/{ticket_type_id}

## Delete Ticket Type
DELETE /api/v1/events/{event_id}/ticket-types/{ticket_type_id}

## Partial Update Ticket Type
PATCH GET /api/v1/events/{event_id}/ticket-types/{ticket_type_id}
Request Body: Partial Ticket Type

## TODO: Dedicated endpoint for report data
```

Summary

- Identified multiple REST API endpoints from the organizer UI Flow

REST API Design - Attendee Flow

Let's analyze the attendee UI flow, further refining our REST API design.

Analyze the Attendee Flow

```
## Create Event
POST /api/v1/events
Request Body: Event

## List Events
GET /api/v1/events

## Retrieve Event
GET /api/v1/events/{event_id}

## Update Event
PUT /api/v1/events/{event_id}
Request Body: Event

## Delete Event
DELETE /api/v1/events/{event_id}

## List Ticket Sales
GET /api/v1/events/{event_id}/tickets

## Retrieve Ticket Sale
GET /api/v1/events/{event_id}/tickets/tickets/{ticket_id}

## Partial Update Ticket
PATCH /api/v1/events/{event_id}/tickets
Request Body: Partial Ticket

## List Ticket Type
GET /api/v1/events/{event_id}/ticket-types

## Retrieve Ticket Type
GET /api/v1/events/{event_id}/ticket-types/{ticket_type_id}

## Delete Ticket Type
DELETE /api/v1/events/{event_id}/ticket-types/{ticket_type_id}

## Partial Update Ticket Type
PATCH GET /api/v1/events/{event_id}/ticket-types/{ticket_type_id}
Request Body: Partial Ticket Type

## Search Published Events
GET /api/v1/published-events

## Retrieve Published Event
GET /api/v1/published-event/{published_event_id}

## Purchase Ticket
```

```
POST /api/v1/published-event/{published_event_id}/ticket-types/{ticket_types_id}

## List Tickets (for user)
GET /api/v1/tickets

## Retrieve Ticket (for user)
GET /api/v1/tickets/{ticket_id}

## Retrieve Ticket QR Code
GET /api/v1/tickets/{ticket_id}/qr-codes

## TODO: Dedicated endpoint for report data
```

Summary

- Identified several more REST API endpoints from the attendee UI Flow

REST API Design - Staff Flow

Let's analyze the staff UI flow, attempting to refine our understanding of the REST API we are to build.

Analyze the Staff Flow

Here's what I've come up with:

```
## Create Event
POST /api/v1/events
Request Body: Event

## List Events
GET /api/v1/events

## Retrieve Event
GET /api/v1/events/{event_id}

## Update Event
PUT /api/v1/events/{event_id}
Request Body: Event

## Delete Event
DELETE /api/v1/events/{event_id}

## Validate Ticket
POST /api/v1/events/{event_id}/ticket-validations

## List Ticket Validations
GET /api/v1/events/{event_id}/ticket-validations

## List Ticket Sales
GET /api/v1/events/{event_id}/tickets

## Retrieve Ticket Sale
GET /api/v1/events/{event_id}/tickets/tickets/{ticket_id}

## Partial Update Ticket
PATCH /api/v1/events/{event_id}/tickets
Request Body: Partial Ticket

## List Ticket Type
GET /api/v1/events/{event_id}/ticket-types

## Retrieve Ticket Type
GET /api/v1/events/{event_id}/ticket-types/{ticket_type_id}

## Delete Ticket Type
DELETE /api/v1/events/{event_id}/ticket-types/{ticket_type_id}

## Partial Update Ticket Type
PATCH GET /api/v1/events/{event_id}/ticket-types/{ticket_type_id}
```

```
Request Body: Partial Ticket Type

## Search Published Events
GET /api/v1/published-events

## Retrieve Published Event
GET /api/v1/published-event/{published_event_id}

## Purchase Ticket
POST /api/v1/published-event/{published_event_id}/ticket-types/{ticket_types_id}

## List Tickets (for user)
GET /api/v1/tickets

## Retrieve Ticket (for user)
GET /api/v1/tickets/{ticket_id}

## Retrieve Ticket QR Code
GET /api/v1/tickets/{ticket_id}/qr-codes

## TODO: Dedicated endpoint for report data
```

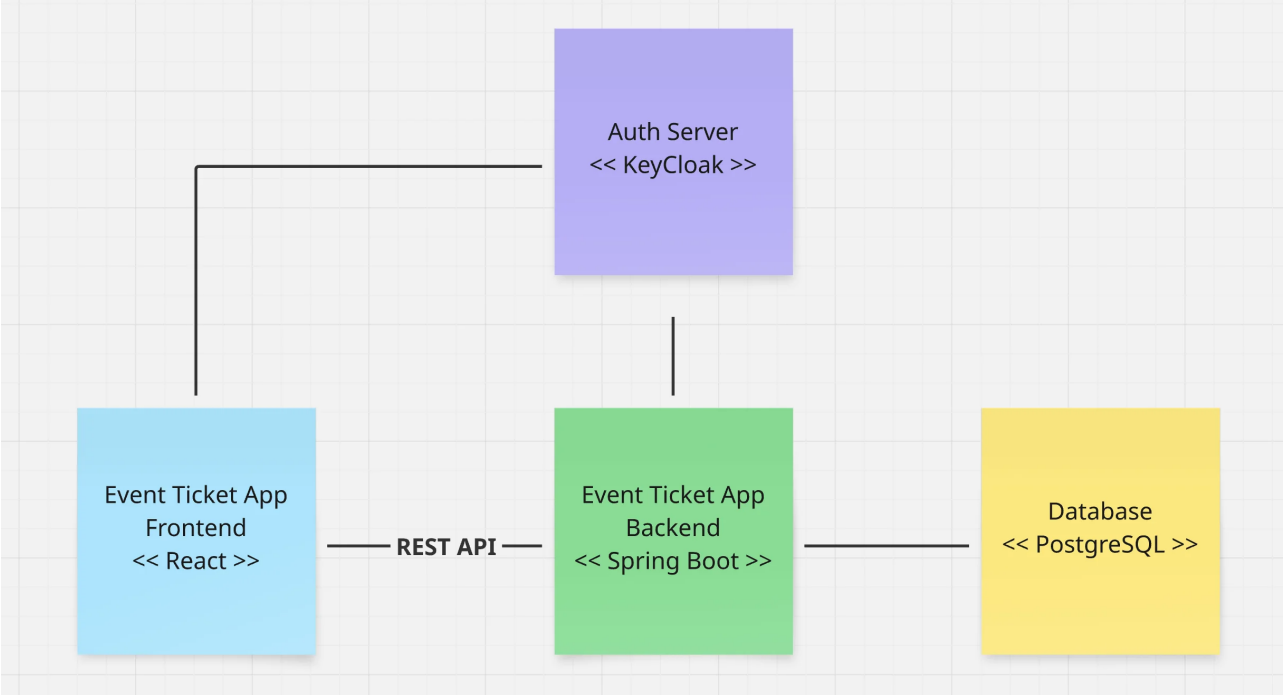
Summary

- Identified several more REST API endpoints from the staff UI Flow
- Completed the initial REST API Design

Architecture Design

We've learned a lot about the application we are to build, let's use this knowledge to design our application's architecture.

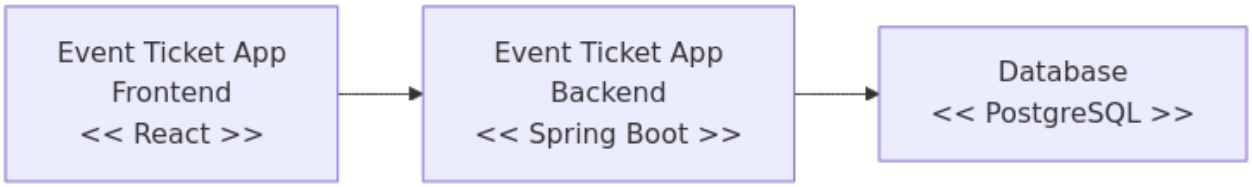
Design the Architecture



Based on the functionality we've captured, we'll need a few components:

- Spring Boot app -- The backend of our application, exposing a REST API
- React App -- The frontend of our application, which calls the REST API
- Keycloak -- Our auth server, handling authentication and authorization

Here's the mermaid diagram:



Summary

- Our architecture includes a Spring Boot backend, React frontend, PostgreSQL database, and a Keycloak server

Module Summary

Key Concepts Covered

REST API Design - Organizer Flow

- Identified multiple REST API endpoints from the organizer UI Flow

REST API Design - Attendee Flow

- Identified several more REST API endpoints from the attendee UI Flow

REST API Design - Staff Flow

- Identified several more REST API endpoints from the staff UI Flow
- Completed the initial REST API Design

Architecture Design

- Our architecture includes a Spring Boot backend, React frontend, PostgreSQL database, and a Keycloak server

Project Setup

Module Overview

In this module we'll set up a new Spring Boot project and configure it with all the tools we need to build an event ticket platform.

This includes creating the project, connecting it to a database, setting up authentication, and configuring helpful development tools.

Module Structure

1. Create a new Spring Boot project
2. Explore the project structure
3. Set up and connect to PostgreSQL
4. Set up and connect to Keycloak
5. Configure MapStruct for object mapping

Learning Objectives

1. Create a new Spring Boot project using the Spring Initalzr
2. Describe the structure and content of your new Spring Boot project
3. Run PostgreSQL and connect your app to it
4. Run Keycloak and connect your app to it
5. Configure MapStruct in the application

New Project

Let's create a new Spring Boot project that will serve as the foundation for our event ticket platform. We'll use the Spring Initializer to set up our project with all the necessary dependencies and configurations we'll need.

Set Up the Project

The Spring Initializer (start.spring.io) helps us create a new Spring Boot project with our chosen configuration.

Let's select these project settings:

- Build Tool: Apache Maven
- Language: Java
- Spring Boot Version: 3.4.4 (Latest stable release)
- Java Version: 21 (Latest LTS)
- Packaging: JAR

For our project metadata:

- Group: `com.devtiro`
- Artifact: `tickets`
- Description: An Event Ticket Platform
- Package Name: `com.devtiro.tickets`

Add Dependencies

Our application needs several key dependencies to function:

Web Dependencies

For building our REST API endpoints we'll use *Spring Web*. We selected this over *WebFlux* for a simpler development experience, and it provides good performance for our needs.

Security Dependencies

We'll need *Spring Security* for securing our application and *OAuth2 Resource Server* for integration with Keycloak.

Database Dependencies

We choose *Spring Data JPA* for database interactions using Java objects. We'll want the *PostgreSQL Driver* for our production database and *H2 Database* for running isolated tests.

Development Tools:

Let's also select *Lombok* as it reduces boilerplate code through annotations.

Summary

- Created a new Spring Boot project using Spring Initializer
- Set up core dependencies for web, security, and data persistence
- Added development tools like Lombok to improve productivity

Explore the Project

Let's explore the structure and content of your Spring Boot project for building an event ticket platform.

Project Structure

The project follows the standard Maven project structure with separate directories for source code, tests, and resources.

In the main source directory (`src/main/java`), we have:

- The main application class `TicketsApplication.java` under the `com.devtiro.tickets` package
- The `@SpringBootApplication` annotation marks this as our application's entry point
- The `main` method uses `SpringApplication.run()` to start our application

The resources directory (`src/main/resources`) contains:

- `application.properties` file for configuration settings
- We've removed the unused `static` and `templates` directories since we'll be using React for our frontend

Dependencies

Our `pom.xml` file includes key dependencies:

- Spring Boot starter dependencies for web, JPA, security, and OAuth2
- PostgreSQL for our production database
- H2 database for testing
- Lombok for reducing boilerplate code
- Testing dependencies including Spring Security Test

Configuration

The `application.properties` file contains settings for our application, currently only the application's name.

```
spring.application.name=tickets
```

Testing Setup

The test directory (`src/test/java`) mirrors the main source structure and includes:

- A basic test class that verifies our Spring context loads correctly
- H2 database configuration for testing, preventing the need for PostgreSQL during tests

Summary

- Project uses standard Maven structure with Spring Boot configuration

- Main application class serves as the entry point with Spring Boot annotations
- Key dependencies include Spring Web, JPA, Security, and PostgreSQL
- H2 database configured for testing environment

Running PostgreSQL

In this lesson, we'll set up PostgreSQL using Docker and configure our Spring Boot application to connect to it.

Set Up PostgreSQL with Docker

Docker makes running PostgreSQL simple and consistent across different development environments. Let's create a `docker-compose.yml` file to define our database setup:

```
services:
 db:
   image: postgres:latest
   ports:
     - "5432:5432"
   restart: always
   environment:
     POSTGRES_PASSWORD: changemeinprod!

 adminer:
   image: adminer:latest
   restart: always
   ports:
     - 8888:8080
```

This configuration sets up two services:

- A PostgreSQL database running on port 5432
- Adminer, a web-based database management tool, running on port 8888

To start these services, run:

```
docker-compose up
```

Configure Spring Boot Database Connection

Our application needs to know how to connect to PostgreSQL. We'll configure this in `application.properties`:

```
# Database Connection
spring.datasource.url=jdbc:postgresql://localhost:5432/postgres
spring.datasource.username=postgres
spring.datasource.password=changemeinprod!

# JPA Configuration
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.PostgreSQLDialect
```

The database connection properties tell Spring Boot:

- Where to find the database (`localhost:5432`)

- Which database to use (`postgres`)
- The login credentials

The JPA configuration enables:

- Automatic table creation and updates based on our entity classes
- SQL logging for development debugging
- PostgreSQL-specific SQL dialect for optimal database interaction

Development Best Practices

When working with databases in development:

- Never store real passwords in configuration files
- Use environment variables or secure configuration management in production
- Consider using database migration tools like Flyway for production deployments
- Keep the SQL logging enabled only in development for debugging

Summary

- PostgreSQL runs in Docker for consistent local development
- Adminer provides a web interface for database management
- Spring Boot connects to PostgreSQL using configuration properties
- JPA automatically manages database schema changes

Running Keycloak

In this lesson, we'll set up Keycloak, a powerful identity and access management solution, to handle authentication and authorization for our event ticket platform. We'll configure Keycloak using Docker Compose and create the initial realm, client, and user settings needed for our application.

Set Up Keycloak with Docker Compose

Docker Compose makes it easy to run Keycloak alongside our other services. Let's add the Keycloak service to our existing `docker-compose.yml`:

```
  keycloak:
   image: quay.io/keycloak/keycloak:latest
   ports:
     - "9090:8080"
   environment:
     KEYCLOAK_ADMIN: admin
     KEYCLOAK_ADMIN_PASSWORD: admin
   volumes:
     - keycloak-data:/opt/keycloak/data
   command:
     - start-dev
     - --db=dev-file

volumes:
 keycloak-data:
   driver: local
```

The configuration maps Keycloak's default port `8080` to `9090` on our host machine to avoid conflicts with our Spring Boot application.

We're using volumes to persist Keycloak's data between container restarts, unlike our PostgreSQL setup where we prefer a fresh start each time.

Configuring Keycloak

Once Keycloak is running, we need to set up three main components:

1. Create a realm named `event-ticket-platform`
2. Set up a client for our frontend application
3. Create a test user to represent an organizer

For the client configuration:

- Client ID: `event-ticket-platform-app`
- Client authentication: Off (for public access)
- Valid redirect URIs: `http://localhost:5173`
- Post logout redirect URIs: `http://localhost:5173`

Connecting Spring Boot to Keycloak

To connect our Spring Boot application to Keycloak, we add this property to `application.properties`:

```
spring.security.oauth2.resourceserver.jwt.issuer-uri=http://localhost:9090/realms/event-ticket-plat
```

This tells Spring Security to validate JWTs against our Keycloak instance.

Summary

- Set up Keycloak using Docker Compose with data persistence
- Created realm, client, and test user in Keycloak
- Connected Spring Boot application to Keycloak for JWT validation

Configuring MapStruct

In this lesson, we'll integrate Mapstruct with our Event Ticket Platform to efficiently handle object mapping between different layers of our application, while ensuring it works smoothly with Project Lombok.

Understanding Mapstruct and Lombok Integration

Mapstruct is a code generator that helps us create type-safe bean mappings between Java bean classes.

When using Mapstruct alongside Lombok, we need special configuration to ensure both tools work together properly during the compilation process.

The order of annotation processing is important - Lombok must process its annotations before Mapstruct can generate its mapper implementations.

Configuring Dependencies

Let's start by adding the required version properties and dependencies to our `pom.xml` file.

First, we'll add the version properties:

```
<properties>
   <org.mapstruct.version>1.6.3</org.mapstruct.version>
   <lombok.version>1.18.36</lombok.version>
</properties>
```

We'll need to pin the Lombok version to ensure it works with Mapstruct:

```
<dependency>
   <groupId>org.projectlombok</groupId>
   <artifactId>lombok</artifactId>
   <version>${lombok.version}</version>
   <optional>true</optional>
</dependency>
```

Next, we'll add the Mapstruct dependency:

```
<dependency>
   <groupId>org.mapstruct</groupId>
   <artifactId>mapstruct</artifactId>
   <version>${org.mapstruct.version}</version>
</dependency>
```

Setting up the Maven Compiler Plugin

The Maven Compiler Plugin needs specific configuration to handle both Lombok and Mapstruct annotation processing:

```
<plugin>
   <groupId>org.apache.maven.plugins</groupId>
   <artifactId>maven-compiler-plugin</artifactId>
```

```
   <configuration>
       <annotationProcessorPaths>
           <!-- Lombok must come first -->
           <path>
               <groupId>org.projectlombok</groupId>
               <artifactId>lombok</artifactId>
               <version>${lombok.version}</version>
           </path>
           <!-- Mapstruct processor -->
           <path>
               <groupId>org.mapstruct</groupId>
               <artifactId>mapstruct-processor</artifactId>
               <version>${org.mapstruct.version}</version>
           </path>
           <!-- Lombok-Mapstruct binding -->
           <path>
               <groupId>org.projectlombok</groupId>
               <artifactId>lombok-mapstruct-binding</artifactId>
               <version>0.2.0</version>
           </path>
       </annotationProcessorPaths>
   </configuration>
</plugin>
```

Understanding the Configuration

The `annotationProcessorPaths` section defines the order of annotation processing:

1. Lombok processes its annotations first
2. Mapstruct processes its annotations second
3. The Lombok-Mapstruct binding ensures compatibility between both tools

There's some conflicting information about if the order is required or not, but I found that it is needed.

Summary

- Added Mapstruct dependency and version property to the project
- Configured Maven Compiler Plugin for annotation processing
- Set up Lombok-Mapstruct binding for compatibility

Module Summary

Key Concepts Covered

New Project

- Created a new Spring Boot project using Spring Initializer
- Set up core dependencies for web, security, and data persistence
- Added development tools like Lombok to improve productivity
- Created a project structure ready for building our ticket platform

Explore the Project

- Project uses standard Maven structure with Spring Boot configuration
- Main application class serves as the entry point with Spring Boot annotations
- Key dependencies include Spring Web, JPA, Security, and PostgreSQL
- H2 database configured for testing environment

Running PostgreSQL

- PostgreSQL runs in Docker for consistent local development
- Adminer provides a web interface for database management
- Spring Boot connects to PostgreSQL using configuration properties
- JPA automatically manages database schema changes
- Development settings help with debugging database interactions

Running Keycloak

- Set up Keycloak using Docker Compose with data persistence
- Created realm, client, and test user in Keycloak
- Connected Spring Boot application to Keycloak for JWT validation

Configuring MapStruct

- Added Mapstruct dependency and version property to the project
- Configured Maven Compiler Plugin for annotation processing
- Set up Lombok-Mapstruct binding for compatibility

Domain

Module Overview

This module focuses on creating the domain classes that will form the backbone of our ticketing platform.

Module Structure

2. Create enum classes
3. Create the user entity class
4. Create the event entity class
5. Create the ticket type entity class
6. Create the ticket entity class
7. Create the ticket validation entity class
8. Create the QR code entity class
9. Generate `equals` and `hashCode` methods

Learning Objectives

1. Create the enum classes used in the domain
2. Create the user domain entity
3. Create the event domain entity
4. Create the ticket type domain entity
5. Create the ticket domain entity
6. Create the ticket validation entity
7. Create the QR code entity
8. Generate the `equals` and `hashCode` method for each entity

Create Enums

In this lesson we'll create the enumerations we'll need when we start implementing our entities.

We'll implement the enums first as they're the simplest to implement and will allow us to build our domain form the bottom up.

Create our Domain's Enums

Enums in Java provide a way to define a fixed set of constants.

In our ticket platform, enums help us maintain data integrity by restricting certain fields to specific, predefined values.

Let's explore each enum and its purpose:

Event Status

The `EventStatusEnum` represents the different states an event can be in throughout its lifecycle:

```
public enum EventStatusEnum {
   DRAFT,      // Initial state when creating an event
   PUBLISHED,  // Event is live and tickets can be purchased
   CANCELLED,  // Event will not take place
   COMPLETED   // Event has finished
}
```

Ticket Status

The `TicketStatusEnum` tracks the state of purchased tickets:

```
public enum TicketStatusEnum {
   PURCHASED,  // Default state when ticket is bought
   CANCELLED   // Ticket has been cancelled
}
```

Ticket Validation

We use two enums for ticket validation:

```
public enum TicketValidationMethod {
   QR_SCAN,  // Ticket validated via QR code scan
   MANUAL    // Ticket validated via manual entry
}

public enum TicketValidationStatusEnum {
   VALID,    // Ticket is valid for entry
   INVALID,  // Ticket is not valid
   EXPIRED   // Ticket has expired
}
```

QR Code Status

The `QrCodeStatusEnum` manages the state of QR codes:

```
public enum QrCodeStatusEnum {
   ACTIVE,   // QR code can be used
   EXPIRED   // QR code has been invalidated
}
```

Summary

- The `EventStatusEnum` represents the states an `Event` could be in
- The `TicketStatusEnum` represents the states a `Ticket` could be in
- The `TicketValidationMethodEnum` represents the different way a ticket can be validated
- The `TicketValidationStatusEnum` represents the the state a `TicketValidation` could be in
- The `QrCodeStatusEnum` represents the different states a `QrCode` could be in

Create User Entity

In this lesson, we'll create a user entity that represents users in our ticketing platform.

Decide the Implementation Approach

While we have three distinct types of users (organizers, staff, and attendees), we'll use a single `User` entity rather than creating separate classes for each type.

This approach simplifies our domain model while still maintaining the flexibility to handle different user roles through Keycloak.

Create the User Entity

Let's create our `User` entity with the necessary fields and JPA annotations:

```
@Entity
@Table(name = "users")
@Getter
@Setter
@NoArgsConstructor
@AllArgsConstructor
@Builder
public class User {
   @Id
   @Column(name = "id", updatable = false, nullable = false)
   private UUID id;

   @Column(name = "name", nullable = false)
   private String name;

   @Column(name = "email", nullable = false)
   private String email;

   // Relationships to be implemented later
   // TODO: Organized events
   // TODO: Attending events
   // TODO: Staffing events

   @CreatedDate
   @Column(name = "created_at", updatable = false, nullable = false)
   private LocalDateTime createdAt;

   @LastModifiedDate
   @Column(name = "updated_at", nullable = false)
   private LocalDateTime updatedAt;
}
```

Summary

- Created a `User` class to represent a user of the system
- We'll use the same `User` class for attendees, staff and organizers
- We'll model the user's permissions using roles which we can assign in Keycloak

- Added `createdAt` and `updatedAt` audit fields

Create Event Entity

In a ticketing platform, events are central to everything - they're what people buy tickets for and attend. Let's create the event entity which will store all the important information about events in our system.

Create the Event Entity

The event entity needs to store basic event details and maintain relationships with users who organize, attend, or staff the event.

```
@Entity
@Table(name = "events")
@Getter
@Setter
@NoArgsConstructor
@AllArgsConstructor
@Builder
public class Event {

   @Id
   @Column(name = "id", updatable = false, nullable = false)
   @GeneratedValue(strategy = GenerationType.UUID)
   private UUID id;

   @Column(name = "name", nullable = false)
   private String name;

   @Column(name = "start")
   private LocalDateTime start;

   @Column(name = "end")
   private LocalDateTime end;

   @Column(name = "venue", nullable = false)
   private String venue;

   @Column(name = "sales_start")
   private LocalDateTime salesStart;

   @Column(name = "sales_end")
   private LocalDateTime salesEnd;

   @Column(name = "status", nullable = false)
   @Enumerated(EnumType.STRING)
   private EventStatusEnum status;

   @ManyToOne(fetch = FetchType.LAZY)
   @JoinColumn(name = "organizer_id")
   private User organizer;

   @ManyToMany(mappedBy = "attendingEvents")
   private List<User> attendees = new ArrayList<>();
```

```
   @ManyToMany(mappedBy = "staffingEvents")
   private List<User> staff = new ArrayList<>();

   @CreatedDate
   @Column(name = "created_at", updatable = false, nullable = false)
   private LocalDateTime createdAt;

   @LastModifiedDate
   @Column(name = "updated_at", nullable = false)
   private LocalDateTime updatedAt;

}
```

Update the User Entity

The `User` class needs corresponding relationships to events:

```
@Entity
@Table(name = "users")
@Getter
@Setter
@NoArgsConstructor
@AllArgsConstructor
@Builder
public class User {

   @Id
   @Column(name = "id", updatable = false, nullable = false)
   private UUID id;

   @Column(name = "name", nullable = false)
   private String name;

   @Column(name = "email", nullable = false)
   private String email;

   @OneToMany(mappedBy = "organizer", cascade = CascadeType.ALL)
   private List<Event> organizedEvents = new ArrayList<>();

   @ManyToMany
   @JoinTable(
           name = "user_attending_events",
           joinColumns = @JoinColumn(name = "user_id"),
           inverseJoinColumns = @JoinColumn(name = "event_id")
   )
   private List<Event> attendingEvents = new ArrayList<>();

   @ManyToMany
   @JoinTable(
           name = "user_staffing_events",
           joinColumns = @JoinColumn(name = "user_id"),
           inverseJoinColumns = @JoinColumn(name = "event_id")
```

```
    )
    private List<Event> staffingEvents = new ArrayList<>();

    @CreatedDate
    @Column(name = "created_at", updatable = false, nullable = false)
    private LocalDateTime createdAt;

    @LastModifiedDate
    @Column(name = "updated_at", nullable = false)
    private LocalDateTime updatedAt;
}
```

Summary

- Created an initial `Event` class
- Added `Event` references to the `User` class

Create Ticket Type Entity

In this lesson, we'll create the `TicketType` entity which represents different categories of tickets available for events in our ticket platform.

Create the Ticket Type Entity

The ticket type entity requires specific attributes to effectively model different ticket categories.

Let's create a new entity class with its required fields and relationships:

```
@Entity
@Table(name = "ticket_types")
@Getter
@Setter
@NoArgsConstructor
@AllArgsConstructor
@Builder
public class TicketType {

   @Id
   @Column(name = "id", nullable = false, updatable = false)
   @GeneratedValue(strategy = GenerationType.UUID)
   private UUID id;

   @Column(name = "name", nullable = false)
   private String name;

   @Column(name = "price", nullable = false)
   private Double price;

   @Column(name = "total_available")
   private Integer totalAvailable;

   @ManyToOne(fetch = FetchType.LAZY)
   @JoinColumn(name = "event_id")
   private Event event;

   // TODO: Tickets

   @CreatedDate
   @Column(name = "created_at", updatable = false, nullable = false)
   private LocalDateTime createdAt;

   @LastModifiedDate
   @Column(name = "updated_at", nullable = false)
   private LocalDateTime updatedAt;
}
```

Event Class Update

To complete the relationship between events and ticket types, we need to update the `Event` class:

```
@Entity
@Table(name = "events")
@Getter
@Setter
@NoArgsConstructor
@AllArgsConstructor
@Builder
public class Event {

   @Id
   @Column(name = "id", updatable = false, nullable = false)
   @GeneratedValue(strategy = GenerationType.UUID)
   private UUID id;

   @Column(name = "name", nullable = false)
   private String name;

   @Column(name = "start")
   private LocalDateTime start;

   @Column(name = "end")
   private LocalDateTime end;

   @Column(name = "venue", nullable = false)
   private String venue;

   @Column(name = "sales_start")
   private LocalDateTime salesStart;

   @Column(name = "sales_end")
   private LocalDateTime salesEnd;

   @Column(name = "status", nullable = false)
   @Enumerated(EnumType.STRING)
   private EventStatusEnum status;

   @ManyToOne(fetch = FetchType.LAZY)
   @JoinColumn(name = "organizer_id")
   private User organizer;

   @ManyToMany(mappedBy = "attendingEvents")
   private List<User> attendees = new ArrayList<>();

   @ManyToMany(mappedBy = "staffingEvents")
   private List<User> staff = new ArrayList<>();

   @OneToMany(mappedBy = "event", cascade = CascadeType.ALL)
   private List<TicketType> ticketTypes = new ArrayList<>();

   @CreatedDate
   @Column(name = "created_at", updatable = false, nullable = false)
   private LocalDateTime createdAt;

   @LastModifiedDate
```

```
   @Column(name = "updated_at", nullable = false)
   private LocalDateTime updatedAt;

}
```

Summary

- Created `TicketType` class to model the different types of ticket available for events
- Added `ticketTypes` reference in the `Event` class

Create Ticket Entity

In this lesson, we'll build the `Ticket` entity, which represents a purchased ticket for an event.

Create the Ticket Entity

A ticket is a record of a purchase that gives someone access to an event. Each ticket needs to track its status, who bought it, and what type of ticket it is.

Let's create our `Ticket` class:

```
@Entity
@Table(name = "tickets")
@Getter
@Setter
@NoArgsConstructor
@AllArgsConstructor
@Builder
public class Ticket {

   @Id
   @Column(name = "id", nullable = false, updatable = false)
   @GeneratedValue(strategy = GenerationType.UUID)
   private UUID id;

   @Column(name = "status", nullable = false)
   @Enumerated(EnumType.STRING)
   private TicketStatusEnum status;

   @ManyToOne(fetch = FetchType.LAZY)
   @JoinColumn(name = "ticket_type_id")
   private TicketType ticketType;

   @ManyToOne(fetch = FetchType.LAZY)
   @JoinColumn(name = "purchaser_id")
   private User purchaser;

   // TODO: Validation

   // TODO: QrCode

   @CreatedDate
   @Column(name = "created_at", updatable = false, nullable = false)
   private LocalDateTime createdAt;

   @LastModifiedDate
   @Column(name = "updated_at", nullable = false)
   private LocalDateTime updatedAt;

}
```

Update the Ticket Type Entity

We also need to update our `TicketType` class to include the inverse relationship:

```
@Entity
@Table(name = "ticket_types")
@Getter
@Setter
@NoArgsConstructor
@AllArgsConstructor
@Builder
public class TicketType {

   @Id
   @Column(name = "id", nullable = false, updatable = false)
   @GeneratedValue(strategy = GenerationType.UUID)
   private UUID id;

   @Column(name = "name", nullable = false)
   private String name;

   @Column(name = "price", nullable = false)
   private Double price;

   @Column(name = "total_available")
   private Integer totalAvailable;

   @ManyToOne(fetch = FetchType.LAZY)
   @JoinColumn(name = "event_id")
   private Event event;

   @OneToMany(mappedBy = "ticketType", cascade = CascadeType.ALL)
   private List<Ticket> tickets = new ArrayList<>();

   @CreatedDate
   @Column(name = "created_at", updatable = false, nullable = false)
   private LocalDateTime createdAt;

   @LastModifiedDate
   @Column(name = "updated_at", nullable = false)
   private LocalDateTime updatedAt;
}
```

Summary

- Created `Ticket` class to model a ticket to an event
- Added `tickets` reference to the `TicketType` class

Create Ticket Validation Entity

In this lesson, we'll create the ticket validation entity that records the validation of a ticket when attendees enter an event.

This entity tracks important details like the validation method used (QR code scan or manual) and the validation status, forming a key part of our ticket management system.

Creating the Ticket Validation Entity

The `TicketValidation` entity represents a single validation attempt of a ticket:

```
@Entity
@Table(name = "ticket_validations")
@Getter
@Setter
@AllArgsConstructor
@NoArgsConstructor
@Builder
public class TicketValidation {

   @Id
   @Column(name = "id", nullable = false, updatable = false)
   @GeneratedValue(strategy = GenerationType.UUID)
   private UUID id;

   @Column(name = "status", nullable = false)
   @Enumerated(EnumType.STRING)
   private TicketValidationStatusEnum status;

   @Column(name = "validation_method", nullable = false)
   @Enumerated(EnumType.STRING)
   private TicketValidationMethod validationMethod;

   @ManyToOne(fetch = FetchType.LAZY)
   @JoinColumn(name = "ticket_id")
   private Ticket ticket;

   @CreatedDate
   @Column(name = "created_at", updatable = false, nullable = false)
   private LocalDateTime createdAt;

   @LastModifiedDate
   @Column(name = "updated_at", nullable = false)
   private LocalDateTime updatedAt;
}
```

Establishing the Relationship with Ticket

We need to update the `Ticket` class to maintain the relationship with its validations:

```
@Entity
@Table(name = "tickets")
```

```
@Getter
@Setter
@NoArgsConstructor
@AllArgsConstructor
@Builder
public class Ticket {

   @Id
   @Column(name = "id", nullable = false, updatable = false)
   @GeneratedValue(strategy = GenerationType.UUID)
   private UUID id;

   @Column(name = "status", nullable = false)
   @Enumerated(EnumType.STRING)
   private TicketStatusEnum status;

   @ManyToOne(fetch = FetchType.LAZY)
   @JoinColumn(name = "ticket_type_id")
   private TicketType ticketType;

   @ManyToOne(fetch = FetchType.LAZY)
   @JoinColumn(name = "purchaser_id")
   private User purchaser;

   @OneToMany(mappedBy = "ticket", cascade = CascadeType.ALL)
   private List<TicketValidation> validations = new ArrayList<>();

   // TODO: QrCode

   @CreatedDate
   @Column(name = "created_at", updatable = false, nullable = false)
   private LocalDateTime createdAt;

   @LastModifiedDate
   @Column(name = "updated_at", nullable = false)
   private LocalDateTime updatedAt;

}
```

Summary

- Created `TicketValidation` class to model a ticket being validated at the event
- Added `validations` reference to the `Ticket` class

Create QR Code Entity

In this lesson, we'll create the QR code entity which represents a unique QR code associated with a ticket.

QR codes are a key part of our ticketing system, allowing for quick and reliable ticket validation at events.

Create the QR Code Entity

Here's the complete implementation of the `QrCode` class:

```
@Entity
@Table(name = "qr_codes")
@Getter
@Setter
@AllArgsConstructor
@NoArgsConstructor
@Builder
public class QrCode {

   @Id
   @Column(name = "id", nullable = false, updatable = false)
   @GeneratedValue(strategy = GenerationType.UUID)
   private UUID id;

   @Column(name = "status", nullable = false)
   @Enumerated(EnumType.STRING)
   private QrCodeStatusEnum status;

   @Column(name = "value", nullable = false)
   private String value;

   @ManyToOne(fetch = FetchType.LAZY)
   @JoinColumn(name = "ticket_id")
   private Ticket ticket;

   @CreatedDate
   @Column(name = "created_at", updatable = false, nullable = false)
   private LocalDateTime createdAt;

   @LastModifiedDate
   @Column(name = "updated_at", nullable = false)
   private LocalDateTime updatedAt;
}
```

Updating the Ticket Entity

To complete the relationship between tickets and QR codes, we need to update the `Ticket` entity to include a reference to its QR codes:

```
@Entity
@Table(name = "tickets")
```

```
@Getter
@Setter
@NoArgsConstructor
@AllArgsConstructor
@Builder
public class Ticket {

   @Id
   @Column(name = "id", nullable = false, updatable = false)
   @GeneratedValue(strategy = GenerationType.UUID)
   private UUID id;

   @Column(name = "status", nullable = false)
   @Enumerated(EnumType.STRING)
   private TicketStatusEnum status;

   @ManyToOne(fetch = FetchType.LAZY)
   @JoinColumn(name = "ticket_type_id")
   private TicketType ticketType;

   @ManyToOne(fetch = FetchType.LAZY)
   @JoinColumn(name = "purchaser_id")
   private User purchaser;

   @OneToMany(mappedBy = "ticket", cascade = CascadeType.ALL)
   private List<TicketValidation> validations = new ArrayList<>();

   @OneToMany(mappedBy = "ticket", cascade = CascadeType.ALL)
   private List<QrCode> qrCodes = new ArrayList<>();

   @CreatedDate
   @Column(name = "created_at", updatable = false, nullable = false)
   private LocalDateTime createdAt;

   @LastModifiedDate
   @Column(name = "updated_at", nullable = false)
   private LocalDateTime updatedAt;

}
```

Summary

- Created `QrCode` class to model a ticket's QrCode
- Added `qrCodes` reference to the `Ticket` class

Equals & Hashcode

When working with Java entities in a Spring Boot application, implementing `equals()` and `hashCode()` methods correctly is necessary to prevent potential issues like infinite recursion in bidirectional relationships and to ensure proper object comparison behavior.

Equals and HashCode in JPA Entities

The `equals()` and `hashCode()` methods are fundamental Java methods used for object comparison and hash-based collections.

When dealing with JPA entities that have relationships with other entities, we need to be careful about which fields we include in these methods to avoid stack overflow errors caused by circular references.

Implementing Equals and HashCode

For our ticket platform entities, we'll generate these methods using our IDE, following these guidelines:

- Include primitive fields and basic types
- Include the entity's ID
- Exclude references to other entities
- Include audit fields (`createdAt` and `updatedAt`)

Here's an example for our `Event` entity:

```
@Override
public boolean equals(Object o) {
   if (o == null || getClass() != o.getClass()) return false;
   Event event = (Event) o;
   return Objects.equals(id, event.id) &&
          Objects.equals(name, event.name) &&
          Objects.equals(start, event.start) &&
          Objects.equals(end, event.end) &&
          Objects.equals(venue, event.venue) &&
          Objects.equals(salesStart, event.salesStart) &&
          Objects.equals(salesEnd, event.salesEnd) &&
          status == event.status &&
          Objects.equals(createdAt, event.createdAt) &&
          Objects.equals(updatedAt, event.updatedAt);
}

@Override
public int hashCode() {
   return Objects.hash(id, name, start, end, venue,
                       salesStart, salesEnd, status,
                       createdAt, updatedAt);
}
```

Notice how we've excluded the `organizer`, `attendees`, `staff`, and `ticketTypes` fields to prevent recursive calls.

Summary

- Used our IDE to generate `equals` and `hashCode` methods for each entity class

Module Summary

Key Concepts Covered

Create Enums

- The `EventStatusEnum` represents the states an `Event` could be in
- The `TicketStatusEnum` represents the states a `Ticket` could be in
- The `TicketValidationMethodEnum` represents the different way a ticket can be validated
- The `TicketValidationStatusEnum` represents the the state a `TicketValidation` could be in
- The `QrCodeStatusEnum` represents the different states a `QrCode` could be in

Create User Entity

- Created a `User` class to represent a user of the system
- We'll use the same `User` class for attendees, staff and organizers
- We'll model the user's permissions using roles which we can assign in Keycloak
- Added `createdAt` and `updatedAt` audit fields

Create Event Entity

- Created an initial `Event` class
- Added `Event` references to the `User` class

Create Ticket Type Entity

- Created `TicketType` class to model the different types of ticket available for events
- Added `ticketTypes` reference in the `Event` class

Create Ticket Entity

- Created `Ticket` class to model a ticket to an event
- Added `tickets` reference to the `TicketType` class

Create Ticket Validation Entity

- Created `TicketValidation` class to model a ticket being validated at the event
- Added `validations` reference to the `Ticket` class

Create QR Code Entity

- Created `QrCode` class to model a ticket's QrCode
- Added `qrCodes` reference to the `Ticket` class

Equals & Hashcode

- Used our IDE to generate `equals` and `hashCode` methods for each entity class

User Provisioning

Module Overview

Module Purpose

In this module, we'll implement automatic user creation when users first log in to our ticketing platform.

This functionality ensures that user data is properly stored and managed in our database, which is essential for tracking ticket purchases and user activity.

Module Structure

Create a custom filter for user provisioning

Configure Spring to use the filter

Enable JPA audit field annotations

Learning Objectives

1. Implement a filter to create new users in the database
2. Configure Spring to use the user provisioning filter
3. Enable the created and last updated annotations used in the entity classes

User Provisioning filter

In this lesson, we'll implement a filter that creates new users in our database when they first log in, ensuring every authenticated user has a corresponding `User` in the database.

Understanding User Provisioning

A user provisioning filter intercepts incoming requests after authentication to check if a user exists in our database and creates them if they don't.

This filter is valuable because it automatically creates user records in our database when users first authenticate through Keycloak, without requiring additional API endpoints or manual intervention.

Implementing the User Repository

First, we need a repository to interact with our user database:

```
@Repository
public interface UserRepository extends JpaRepository<User, UUID> {
}
```

By extending `JpaRepository`, we get built-in methods for:

- Creating users
- Checking if users exist
- Finding users by ID
- And many other common database operations

Creating the User Provisioning Filter

The filter needs to:

1. Extract user information from the JWT token
2. Check if the user exists in our database
3. Create the user if they don't exist

Here's the implementation:

```
@Component
@RequiredArgsConstructor
public class UserProvisioningFilter extends OncePerRequestFilter {

   private final UserRepository userRepository;

   @Override
   protected void doFilterInternal(
           HttpServletRequest request,
           HttpServletResponse response,
           FilterChain filterChain) throws ServletException, IOException {

       // Get the authentication object from the security context
```

```
        Authentication authentication = SecurityContextHolder.getContext().getAuthentication();

        if (authentication != null
                && authentication.isAuthenticated()
                && authentication.getPrincipal() instanceof Jwt jwt) {

            // Extract the user ID from the JWT subject
            UUID keycloakId = UUID.fromString(jwt.getSubject());

            if (!userRepository.existsById(keycloakId)) {
                User user = new User();
                user.setId(keycloakId);
                user.setName(jwt.getClaimAsString("preferred_username"));
                user.setEmail(jwt.getClaimAsString("email"));

                userRepository.save(user);
            }
        }

        filterChain.doFilter(request, response);
    }
}
```

The filter extends `OncePerRequestFilter` to ensure it only runs once per request.

We use `@RequiredArgsConstructor` to inject our `UserRepository` through constructor injection.

The filter checks if we have an authenticated user with a JWT token, then extracts the user ID and creates a new user record if one doesn't exist.

Summary

- Created the `UserRepository` interface
- Implemented the `UserProvisioningFilter`

Spring Security Configuration

In this lesson, we'll configure Spring Security to work with our user provisioning filter.

Create Spring Security Configuration

Spring Security configuration is central to managing authentication and authorization in our application.

Let's create a new configuration class:

```
@Configuration
public class SecurityConfig {

   @Bean
   public SecurityFilterChain filterChain(
           HttpSecurity http,
           UserProvisioningFilter userProvisioningFilter) throws Exception {
       http
               .authorizeHttpRequests(authorize ->
                       // All requests must be authenticated
                       authorize.anyRequest().authenticated())
               .csrf(csrf -> csrf.disable())
               .sessionManagement(session ->
                       session.sessionCreationPolicy(SessionCreationPolicy.STATELESS))
               .oauth2ResourceServer(oauth2 ->
                       oauth2.jwt(
                               Customizer.withDefaults()
                       ))
               .addFilterAfter(userProvisioningFilter, BearerTokenAuthenticationFilter.class);

       return http.build();
   }
}
```

Configuration Components

The configuration consists of several key parts that work together:

The `authorizeHttpRequests()` method sets up request authorization, requiring authentication for all incoming requests.

We disable CSRF protection as it's not typically needed for REST APIs.

The `sessionManagement()` configuration sets our application to be stateless, which is standard for REST APIs.

The `oauth2ResourceServer()` configuration sets up JWT token validation using default settings.

Finally, we add our user provisioning filter after the `BearerTokenAuthenticationFilter`, ensuring authentication happens before user provisioning.

Summary

- Created a basic Spring Security configuration file

- Disabled the CSRF mechanism on our REST API
- Configured Spring Security to be stateless
- Configured Spring Security to use the `UserProvisioningFilter`

Enable Spring JPA Audit

In order to track when our entities are created and updated, we need to enable Spring JPA's auditing functionality to automatically maintain audit fields like `@CreatedDate` and `@LastModifiedDate`.

Enable JPA Auditing

Spring JPA auditing allows us to automatically track when entities are created and modified, but it needs to be explicitly enabled.

We'll enable it globally for our application with two key components:

1. A configuration class with the `@EnableJpaAuditing` annotation
2. An `orm.xml` configuration file that registers the auditing entity listener

Here's how we implement the configuration class:

```
@Configuration
@EnableJpaAuditing
public class JpaConfiguration {
}
```

Next, we need to create the `orm.xml` file in a specific location:

```
<?xml version="1.0" encoding="UTF-8"?>
<entity-mappings xmlns="http://java.sun.com/xml/ns/persistence/orm"
                 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                 xsi:schemaLocation="http://java.sun.com/xml/ns/persistence/orm http://java.sun.com/
                 version="2.0">
   <persistence-unit-metadata>
       <persistence-unit-defaults>
           <entity-listeners>
               <entity-listener class="org.springframework.data.jpa.domain.support.AuditingEntityLi
           </entity-listeners>
       </persistence-unit-defaults>
   </persistence-unit-metadata>
</entity-mappings>
```

The `orm.xml` file needs to be placed in the `src/main/resources/META-INF` directory. When the application is built, this file will be included in the final package and made available to Hibernate.

Why Global Configuration?

While we could add the `@EntityListeners` annotation to each entity class individually, using global configuration through `orm.xml` is a better approach because:

- It reduces code duplication
- It makes the auditing behavior consistent across all entities
- It's easier to maintain and modify in one central location

Summary

- Created `JpaConfiguration` class with the `@EnableJpaAuditing` annotation
- Added `orm.xml` configuration file enabling the audit fields

Module Summary

Key Concepts Covered

User Provisioning filter

- Created the `UserRepository` interface
- Implemented the `UserProvisioningFilter`

Spring Security Configuration

- Created a basic Spring Security configuration file
- Disabled the CSRF mechanism on our REST API
- Configured Spring Security to be stateless
- Configured Spring Security to use the `UserProvisioningFilter`

Enable Spring JPA Audit

- Created `JpaConfiguration` class with the `@EnableJpaAuditing` annotation
- Added `orm.xml` configuration file enabling the audit fields

Create Event

Module Overview

In this module we'll build the functionality to allow event organizers to create new events.

Module Structure

1. Design the implementation
2. Add custom exceptions
3. Implement the service layer
4. Implement DTOs & Mappers
5. Implement the REST API endpoint
6. Add error handling
7. Test the new feature

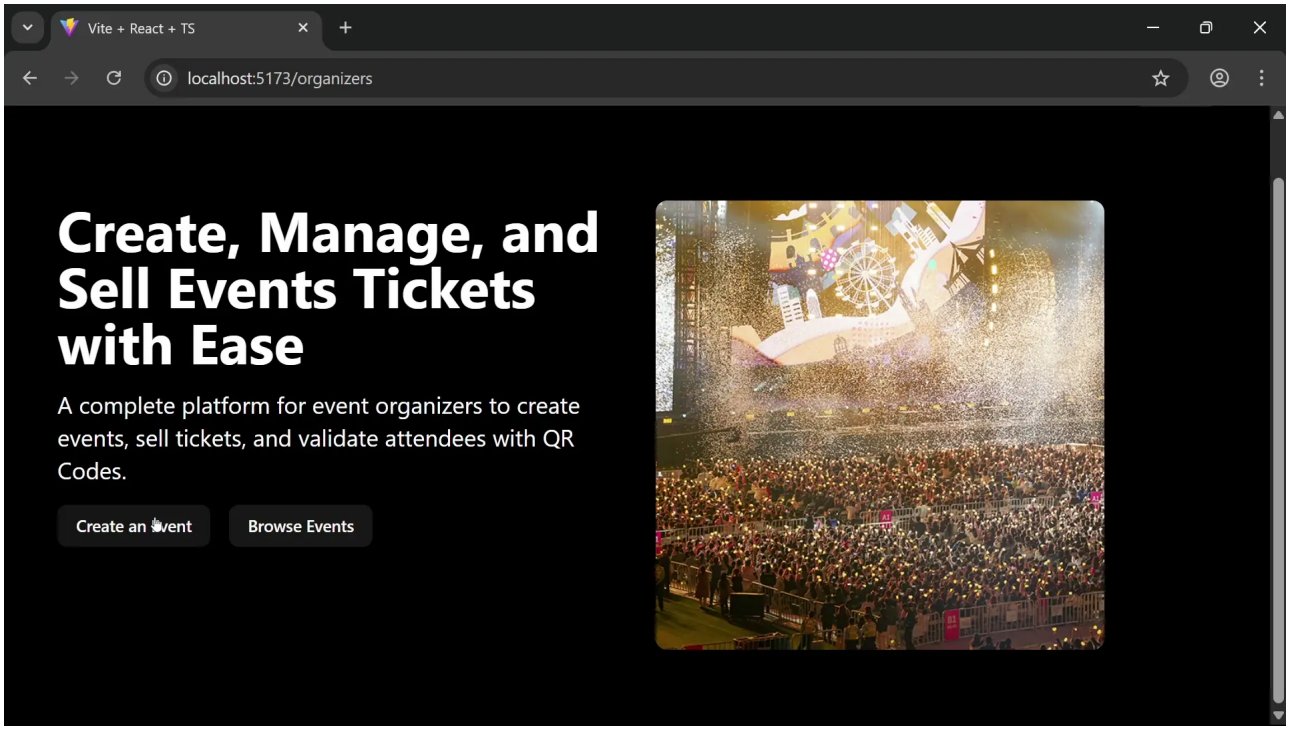
Learning Objectives

1. Summarize the user interface, and any additional requirements it introduces to the backend
2. Design the create event service layer implementation
3. Implement an exception thrown when expected users do no exist
4. Implement the create event business logic in the backend
5. Implement the DTOs and Mappers required to create event in the backend
6. Implement the create event REST API endpoint
7. Appropriately return errors to the callers of the REST API
8. Test if the create event functionality is working by using the user interface

UI Overview

In this lesson, we'll explore the user interface for creating events and identify any new requirements it introduces for our backend implementation.

Frontend Architecture



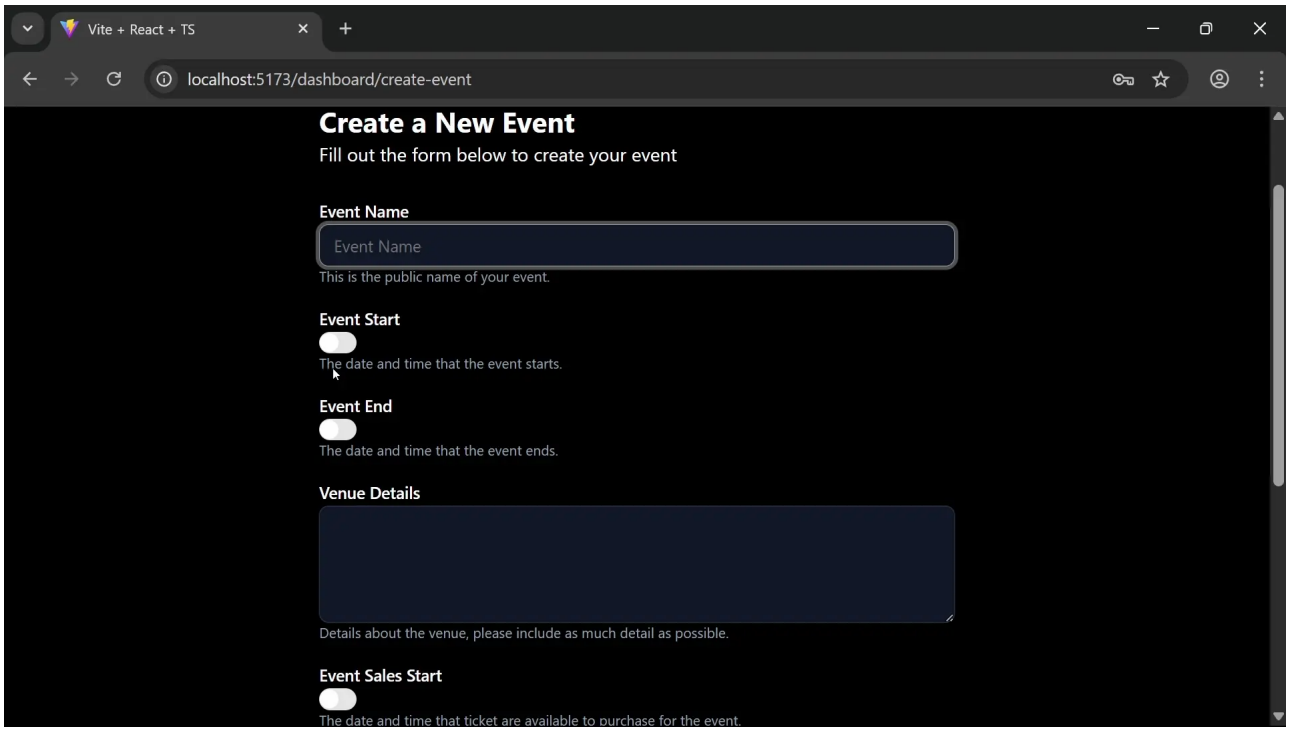
The frontend application is built using React and Vite, with authentication handled through Keycloak. To run it locally:

```
# Install dependencies
npm install --force  # Force flag needed due to shadcn dependency conflicts

# Start development server
npm run dev
```

The application will be available at `localhost:5173`.

Create Event Form



The create event form allows organizers to input:

- Event name (required)
- Event dates (start/end, optional)
- Venue details (required)
- Sales dates (start/end, optional)
- Ticket types (optional)
- Event status (draft/published)

Each ticket type has:

- Name
- Price
- Total available tickets
- Description (new field)

New Backend Requirements

The UI introduces a change to the requirements we need to consider. The `description` field for ticket types needs to be added to our backend model.

Summary

- The user interface can be used to create an event
- A field named `description` has been added to the ticket type object

Create Event Design

In this lesson, we'll design the service layer for creating events in our ticket platform.

Domain Objects Design

The first step is creating data transfer objects (DTOs) that will carry the event creation information between layers.

We'll create two DTOs:

```
@Data
@AllArgsConstructor
@NoArgsConstructor
public class CreateTicketTypeRequest {
   private String name;
   private Double price;
   private String description;
   private Integer totalAvailable;
}
```

```
@Data
@AllArgsConstructor
@NoArgsConstructor
public class CreateEventRequest {
   private String name;
   private LocalDateTime start;
   private LocalDateTime end;
   private String venue;
   private LocalDateTime salesStart;
   private LocalDateTime salesEnd;
   private EventStatusEnum status;
   private List<CreateTicketTypeRequest> ticketTypes = new ArrayList<>();
}
```

The `CreateTicketTypeRequest` contains only the fields needed to create a ticket type, omitting relationships and audit fields. This makes the object focused and easier to validate.

The `CreateEventRequest` follows the same pattern, including only the data needed to create an event. Notice we're using a list of `CreateTicketTypeRequest` objects rather than `TicketType` entities, maintaining the separation between our domain and persistence layers.

Service Interface Design

The service interface defines the contract for creating events:

```
public interface EventService {
   Event createEvent(UUID organizerId, CreateEventRequest event);
}
```

The `createEvent` method takes two parameters:

- `organizerId`: The UUID of the user creating the event
- `event`: The event creation request containing all event details

This design separates the concerns of user identification from event data, making the code more flexible and easier to test.

Summary

- Implemented the `CreateTicketTypeRequest` class
- Implemented the `CreateEventRequest` class
- Created the `EventService` interface with `createEvent` method

User Not Found Exception

In this lesson, we'll implement an exception that is thrown when a user is not found in our system.

This will help us handle error cases in a clean and organized way.

Understanding Custom Exceptions

Custom exceptions help us handle application-specific error cases in a way that makes sense for our domain.

When creating custom exceptions, it's helpful to have a base exception class that all other exceptions extend.

Let's create our base exception class:

```
public class EventTicketException extends RuntimeException {
   public EventTicketException() {
   }

   public EventTicketException(String message) {
       super(message);
   }

   public EventTicketException(String message, Throwable cause) {
       super(message, cause);
   }

   public EventTicketException(Throwable cause) {
       super(cause);
   }

   public EventTicketException(String message, Throwable cause,
            boolean enableSuppression, boolean writableStackTrace) {
       super(message, cause, enableSuppression, writableStackTrace);
   }
}
```

Creating the User Not Found Exception

Now that we have our base exception, we can create our specific exception for when a user is not found:

```
public class UserNotFoundException extends EventTicketException {
   public UserNotFoundException() {
   }

   public UserNotFoundException(String message) {
       super(message);
   }

   public UserNotFoundException(String message, Throwable cause) {
       super(message, cause);
```

```
    }

    public UserNotFoundException(Throwable cause) {
        super(cause);
    }

    public UserNotFoundException(String message, Throwable cause,
            boolean enableSuppression, boolean writableStackTrace) {
        super(message, cause, enableSuppression, writableStackTrace);
    }
}
```

Using Runtime Exceptions

We extend `RuntimeException` in our base exception class rather than `Exception`.

This choice means we don't need to declare throws clauses on methods that might throw our exceptions.

This approach, recommended by Robert C. Martin in "Clean Code", helps maintain the Open-Closed Principle by preventing changes to method signatures when new exceptions are added.

Summary

- Created a `EventTicketException` parent custom exception
- Created the `UserNotFoundException` to represent the invalid state when a specified user does not exist

Create Event Implementation

In this lesson, we'll implement the service layer logic to create events in our event tick platform.

Implementation Details

The event creation service needs two pieces of information: the organizer's ID and the event details.

```
@Service
@RequiredArgsConstructor
public class EventServiceImpl implements EventService {
   private final UserRepository userRepository;
   private final EventRepository eventRepository;

   @Override
   public Event createEvent(UUID organizerId, CreateEventRequest event) {
       // Find the organizer or throw an exception if not found
       User organizer = userRepository.findById(organizerId)
               .orElseThrow(() -> new UserNotFoundException(
                       String.format("User with ID '%s' not found", organizerId))
               );

       // Create ticket types
       List<TicketType> ticketTypesToCreate = event.getTicketTypes().stream().map(
               ticketType -> {
                   TicketType ticketTypeToCreate = new TicketType();
                   ticketTypeToCreate.setName(ticketType.getName());
                   ticketTypeToCreate.setPrice(ticketType.getPrice());
                   ticketTypeToCreate.setDescription(ticketType.getDescription());
                   ticketTypeToCreate.setTotalAvailable(ticketType.getTotalAvailable());
                   return ticketTypeToCreate;
               }).toList();

       // Create and populate the event
       Event eventToCreate = new Event();
       eventToCreate.setName(event.getName());
       eventToCreate.setStart(event.getStart());
       eventToCreate.setEnd(event.getEnd());
       eventToCreate.setVenue(event.getVenue());
       eventToCreate.setSalesStart(event.getSalesStart());
       eventToCreate.setSalesEnd(event.getSalesEnd());
       eventToCreate.setStatus(event.getStatus());
       eventToCreate.setOrganizer(organizer);
       eventToCreate.setTicketTypes(ticketTypesToCreate);

       return eventRepository.save(eventToCreate);
   }
}
```

Error Handling

The service includes error handling for cases where the organizer doesn't exist:

- We use `userRepository.findById()` to look up the organizer
- If not found, we throw a `UserNotFoundException` with a clear message
- The exception includes the ID that wasn't found to help with troubleshooting

Summary

- Created the `EventRepository` interface
- Created the `EventServiceImpl` class
- Implemented the `createEvent` method

DTOs & Mappers

In this lesson, we'll implement the DTOs and Mappers needed to handle event creation in our REST API.

These components help us maintain a clear separation between our API contract and internal domain model, while ensuring data validation at the API boundary.

Understanding DTOs and Their Purpose

Data Transfer Objects (DTOs) serve as specialized objects for transferring data between our API and clients. Unlike our domain models, DTOs:

- Are tailored specifically for API communication
- Include validation rules for request data
- Can be modified without affecting our domain models
- Help prevent exposing internal implementation details

Create Request DTOs

Let's create our request DTOs with proper validation:

```
@Data
@AllArgsConstructor
@NoArgsConstructor
public class CreateEventRequestDto {
   @NotBlank(message = "Event name is required")
   private String name;

   private LocalDateTime start;
   private LocalDateTime end;

   @NotBlank(message = "Venue information is required")
   private String venue;

   private LocalDateTime salesStart;
   private LocalDateTime salesEnd;

   @NotNull(message = "Event status must be provided")
   private EventStatusEnum status;

   @NotEmpty(message = "At least one ticket type is required")
   @Valid
   private List<CreateTicketTypeRequestDto> ticketTypes;
}
```

For ticket types, we create a separate DTO with its own validations:

```
@Data
@AllArgsConstructor
@NoArgsConstructor
public class CreateTicketTypeRequestDto {
```

```
   @NotBlank(message = "Ticket type name is required")
   private String name;

   @NotNull(message = "Price is required")
   @PositiveOrZero(message = "Price must be zero or greater")
   private Double price;

   private String description;
   private Integer totalAvailable;
}
```

Creating Response DTOs

Response DTOs represent the data we send back after creating an event:

```
@Data
@AllArgsConstructor
@NoArgsConstructor
public class CreateEventResponseDto {
   private UUID id;
   private String name;
   private LocalDateTime start;
   private LocalDateTime end;
   private String venue;
   private LocalDateTime salesStart;
   private LocalDateTime salesEnd;
   private EventStatusEnum status;
   private List<CreateTicketTypeResponseDto> ticketTypes;
   private LocalDateTime createdAt;
   private LocalDateTime updatedAt;
}
```

Implementing the Mapper

MapStruct helps us convert between DTOs and domain objects:

```
@Mapper(componentModel = "spring", unmappedTargetPolicy = ReportingPolicy.IGNORE)
public interface EventMapper {
   CreateTicketTypeRequest fromDto(CreateTicketTypeRequestDto dto);
   CreateEventRequest fromDto(CreateEventRequestDto dto);
   CreateEventResponseDto toDto(Event event);
}
```

Summary

- Created the `CreateEventRequestDto`
- Created the `CreateEventResponseDto`
- Created the `EventMapper` interface
- The DTOs make use of validation annotations to ensure data consistency

Event Controller

In this lesson, we'll implement the REST API endpoint for creating events.

Create the Event Controller

The event controller is responsible for handling HTTP requests related to events. Let's create a new controller class with Spring's `@RestController` annotation:

```
@RestController
@RequestMapping("/api/v1/events")
@RequiredArgsConstructor
public class EventController {
   private final EventMapper eventMapper;
   private final EventService eventService;

   @PostMapping
   public ResponseEntity<CreateEventResponseDto> createEvent(
           @AuthenticationPrincipal Jwt jwt,
           @Valid @RequestBody CreateEventRequestDto createEventRequestDto) {
       // Convert DTO to domain object
       CreateEventRequest createEventRequest = eventMapper.fromDto(createEventRequestDto);

       // Extract user ID from JWT
       UUID userId = UUID.fromString(jwt.getSubject());

       // Create the event
       Event createdEvent = eventService.createEvent(userId, createEventRequest);

       // Convert response to DTO
       CreateEventResponseDto createEventResponseDto = eventMapper.toDto(createdEvent);

       return new ResponseEntity<>(createEventResponseDto, HttpStatus.CREATED);
   }
}
```

Summary

- Created the `EventController` class
- Implemented create event endpoint

Global Exception Handler

In REST APIs, handling errors consistently and providing meaningful feedback to clients is a key part of creating a good developer experience.

In this lesson, we'll implement a global exception handler to manage errors across our application.

Custom Error Response Format

To maintain consistency, we'll create a simple DTO class for our error responses:

```
@Data
@AllArgsConstructor
@NoArgsConstructor
public class ErrorDto {
   private String error;
}
```

Understanding Exception Handlers

Spring's exception handling mechanism allows us to centralize our error handling logic in one place. This means we can define how different types of exceptions should be handled and what response the client should receive.

Here's how we'll create our global exception handler:

```
@RestControllerAdvice
@Slf4j
public class GlobalExceptionHandler {

   @ExceptionHandler(Exception.class)
   public ResponseEntity<ErrorDto> handleException(Exception ex) {
       log.error("Caught exception", ex);
       ErrorDto errorDto = new ErrorDto();
       errorDto.setError("An unknown error occurred");
       return new ResponseEntity<>(errorDto, HttpStatus.INTERNAL_SERVER_ERROR);
   }
}
```

Handling Specific Exceptions

For validation errors, we need to handle two specific types of exceptions: `ConstraintViolationException` and `MethodArgumentNotValidException`. These occur when request validation fails:

```
@ExceptionHandler(ConstraintViolationException.class)
public ResponseEntity<ErrorDto> handleConstraintViolation(ConstraintViolationException ex) {
   log.error("Caught ConstraintViolationException", ex);

   String errorMessage = ex.getConstraintViolations()
       .stream()
       .findFirst()
```

```
        .map(violation -> violation.getPropertyPath() + ": " + violation.getMessage())
        .orElse("A constraint violation occurred");

    ErrorDto errorDto = new ErrorDto();
    errorDto.setError(errorMessage);
    return new ResponseEntity<>(errorDto, HttpStatus.BAD_REQUEST);
}
```

```
  @ExceptionHandler(MethodArgumentNotValidException.class)
    public ResponseEntity<ErrorDto> handleMethodArgumentNotValidException(
            MethodArgumentNotValidException ex
    ) {
        log.error("Caught MethodArgumentNotValidException", ex);
        ErrorDto errorDto = new ErrorDto();

        BindingResult bindingResult = ex.getBindingResult();
        List<FieldError> fieldErrors = bindingResult.getFieldErrors();
        String errorMessage = fieldErrors.stream()
                .findFirst()
                .map(fieldError -> fieldError.getField() + ": " + fieldError.getDefaultMessage())
                .orElse("Validation error occurred");

        errorDto.setError(errorMessage);
        return new ResponseEntity<>(errorDto, HttpStatus.BAD_REQUEST);
    }
```

```
@ExceptionHandler(UserNotFoundException.class)
public ResponseEntity<ErrorDto> handleUserNotFoundException(UserNotFoundException ex) {
    log.error("Caught UserNotFoundException", ex);
    ErrorDto errorDto = new ErrorDto();
    errorDto.setError("User not found");
    return new ResponseEntity<>(errorDto, HttpStatus.BAD_REQUEST);
}
```

Summary

- Created the `ErrorDto` class
- Created the `GlobalExceptionHandler` class
- All errors and now returned in the expected format

Try it Out

In this lesson, we'll test the newly implemented event creation functionality through the user interface.

Set Up the Environment

Before we can test our event creation functionality, we need to start our development environment:

1. Start the Docker services by running:

```
docker compose up
```

2. Run Maven clean and compile to ensure all generated code is up to date:

```
mvn clean compile
```

3. Start the Spring Boot application

Fixing the Database Column Names

When starting the application, you might encounter an error related to reserved keywords in PostgreSQL.

To fix this, update the `Event` entity's column names:

```
@Column(name = "event_start")
private LocalDateTime start;

@Column(name = "event_end")
private LocalDateTime end;
```

This change avoids using PostgreSQL's reserved keywords `start` and `end` as column names.

Creating Your First Event

Navigate to '/organizers' in your browser and click "Create an Event".

You'll need to:

- Log in with your organizer credentials
- Fill in the event details:
    - Name (required)
    - Venue (required)
    - At least one ticket type with:
        - Name
        - Price
        - Total available tickets

        - Description (optional)
- Choose whether to publish the event or keep it as a draft

Verifying Event Creation

After creating an event, you can verify its creation in several ways:

- Check the HTTP response status (should be 201 Created)
- Review the response JSON to ensure all fields are correct
- Use Adminer (available at port 8888) to check the database tables:
    - events table - contains the event details
    - ticket_types table - contains the ticket configuration
    - users table - contains the organizer information

Link Event and TicketType

When checking adminer we see that the ticket types's `event_id` field is null in the database.

We fix this by ensuring we set the event object on the `TicketType` class when we create them in the event service:

```
@Override
public Event createEvent(UUID organizerId, CreateEventRequest event) {
   User organizer = userRepository.findById(organizerId)
           .orElseThrow(() -> new UserNotFoundException(
                   String.format("User with ID '%s' not found", organizerId))
           );

   // eventToCreate needs to be moved up here
   Event eventToCreate = new Event();

   List<TicketType> ticketTypesToCreate = event.getTicketTypes().stream().map(
           ticketType -> {
               TicketType ticketTypeToCreate = new TicketType();
               ticketTypeToCreate.setName(ticketType.getName());
               ticketTypeToCreate.setPrice(ticketType.getPrice());
               ticketTypeToCreate.setDescription(ticketType.getDescription());
               ticketTypeToCreate.setTotalAvailable(ticketType.getTotalAvailable());
               // 2. The next line needs to be added
               ticketTypeToCreate.setEvent(eventToCreate);
               return ticketTypeToCreate;
           }).toList();

   eventToCreate.setName(event.getName());
   eventToCreate.setStart(event.getStart());
   eventToCreate.setEnd(event.getEnd());
   eventToCreate.setVenue(event.getVenue());
   eventToCreate.setSalesStart(event.getSalesStart());
   eventToCreate.setSalesEnd(event.getSalesEnd());
   eventToCreate.setStatus(event.getStatus());
```

```
    eventToCreate.setOrganizer(organizer);
    eventToCreate.setTicketTypes(ticketTypesToCreate);

    return eventRepository.save(eventToCreate);
}
```

Summary

- We experienced an error when attempting to save an event in the database
- `start` and `end` are reserved keywords in PostgreSQL
- We updated the event object to no longer use the reserve keywords
- We have been able to successfully create an event

Module Summary

Key Concepts Covered

UI Overview

- The user interface can be used to create an event
- A field named `description` has been added to the ticket type object

Create Event Design

- Implemented the `CreateTicketTypeRequest` class
- Implemented the `CreateEventRequest` class
- Created the `EventService` interface with `createEvent` method

User Not Found Exception

- Created a `EventTicketException` parent custom exception
- Created the `UserNotFoundException` to represent the invalid state when a specified user does not exist

Create Event Implementation

- Created the `EventRepository` interface
- Created the `EventServiceImpl` class
- Implemented the `createEvent` method

DTOs & Mappers

- Created the `CreateEventRequestDto`
- Created the `CreateEventResponseDto`
- Created the `EventMapper` interface
- The DTOs make use of validation annotations to ensure data consistency

Event Controller

- Created the `EventController` class
- Implemented create event endpoint

Global Exception Handler

- Created the `ErrorDto` class
- Created the `GlobalExceptionHandler` class
- All errors and now returned in the expected format

UI Testing

- We experienced an error when attempting to save an event in the database
- `start` and `end` are reserved keywords in PostgreSQL

- We updated the event object to no longer use the reserve keywords
- We have been able to successfully create an event

List Events

Module Overview

In this module, we'll build the functionality that allows organizer to list their events.

Module Structure

1. Implement the service layer for event listing
2. Create DTOs and mappers for event data
3. Build the events controller endpoint
4. Test the functionality through the UI

Learning Objectives

1. Implement list event functionality in the service layer
2. Implement the DTOs and mappers required to list events in the presentation layer
3. Implement the list events functionality in events controller
4. Test that list events works in the user interface

List Event Service

In this lesson, we'll implement the list events functionality in the service layer, allowing organizers to view their events through pagination. This feature enables efficient data retrieval and better performance when dealing with large numbers of events.

Understanding Pagination

Spring Data JPA's pagination feature helps manage large datasets by breaking them into smaller, manageable chunks. Instead of fetching all records at once, pagination returns a specific "page" of results, which includes both the data and metadata about the total results.

Repository Method

Spring Data JPA's method naming conventions allow us to create custom finder methods without writing SQL queries. The repository interface defines the method signature, and Spring generates the implementation.

```
// In EventRepository interface
public interface EventRepository extends JpaRepository<Event, UUID> {
   // Spring Data JPA will generate the implementation based on the method name
   Page<Event> findByOrganizerId(UUID organizerId, Pageable pageable);
}
```

The method name `findByOrganizerId` tells Spring Data JPA to:

- Find events (`find`)
- Filter by the organizer's ID (`ByOrganizerId`)
- Return results in pages (`Page<Event>`)
- Accept pagination parameters (`Pageable pageable`)

Implement the Service Layer

```
// Method in EventService interface
Page<Event> listEventsForOrganizer(UUID organizerId, Pageable pageable);

// Implementation in EventServiceImpl
@Override
public Page<Event> listEventsForOrganizer(UUID organizerId, Pageable pageable) {
   // Use the repository to find events by organizer ID with pagination
   return eventRepository.findByOrganizerId(organizerId, pageable);
}
```

Summary

- Added the `listEventsForOrganizer` method to the `EventService` interface
- Added the `findByOrganizerId` method to the `EventRepository` interface
- Implemented `listEventsForOrganizer` method in the `EventServiceImpl` class

DTOs and Mappers

In this lesson we implement the DTOs and mappers required to implement the list event functionality.

Understanding DTOs for Event Listing

Data Transfer Objects (DTOs) help us move data between layers while controlling exactly what information we share. When listing events, we need to consider what information is necessary for display and what should be kept private.

Let's create two DTOs:

```
@Data
@AllArgsConstructor
@NoArgsConstructor
public class ListEventResponseDto {
   private UUID id;
   private String name;
   private LocalDateTime start;
   private LocalDateTime end;
   private String venue;
   private LocalDateTime salesStart;
   private LocalDateTime salesEnd;
   private EventStatusEnum status;
   private List<ListEventTicketTypeResponseDto> ticketTypes = new ArrayList<>();
}
```

```
@Data
@AllArgsConstructor
@NoArgsConstructor
public class ListEventTicketTypeResponseDto {
   private UUID id;
   private String name;
   private Double price;
   private String description;
   private Integer totalAvailable;
}
```

Mapping Strategy

We'll update our `EventMapper` interface to include methods for converting our entities to these new DTOs:

```
@Mapper(componentModel = "spring", unmappedTargetPolicy = ReportingPolicy.IGNORE)
public interface EventMapper {
   // Existing methods...

   ListEventTicketTypeResponseDto toDto(TicketType ticketType);

   ListEventResponseDto toListEventResponseDto(Event event);
}
```

Data Transfer Considerations

When designing DTOs for listing events, we've made specific choices about what data to include:

- Included basic event details needed for display (ID, name, dates, venue)
- Included ticket types because they're shown in event listings
- Excluded sensitive or unnecessary data (organizer details, attendees, staff)
- Excluded audit fields (createdAt, updatedAt) as they're not displayed
- Initialized collections to empty lists to prevent null pointer issues

Summary

- Created the `ListEventResponseDto`
- Create the `ListEventTicketTypeResponseDto`
- Updated the `EventMapper`to map to these DTO classes

Event Controller

In this lesson we will implement the list events functionality in the events controller.

Implementing the List Events Endpoint

Let's add a new endpoint to our `EventController` that will handle GET requests to list events.

```
@GetMapping
public ResponseEntity<Page<ListEventResponseDto>> listEvents(
       @AuthenticationPrincipal Jwt jwt, Pageable pageable
) {
   UUID userId = parseUserId(jwt);
   Page<Event> events = eventService.listEventsForOrganizer(userId, pageable);
   return ResponseEntity.ok(
           events.map(eventMapper::toListEventResponseDto)
   );
}
```

The endpoint takes two parameters:

- The `@AuthenticationPrincipal Jwt jwt` which contains the authenticated user's information
- A `Pageable` object that Spring automatically creates from query parameters

Understanding the Implementation

The endpoint follows a clear flow:

1. Extract the user ID from the JWT token using our helper method `parseUserId`
2. Call the service layer to retrieve a page of events for the organizer
3. Map the page of events to DTOs using our mapper
4. Return the mapped page with a 200 OK status

The `Pageable` parameter deserves special attention.

Spring will automatically create this object from standard query parameters:

- `page` - The page number (0-based)
- `size` - The number of items per page
- `sort` - The sorting criteria

For example: `/api/v1/events?page=0&size=10&sort=name,desc`

Authentication vs Authorization

At this stage, our implementation handles authentication but not full authorization.

We verify that users are who they claim to be through JWT validation, and we filter events by organizer.

However, we haven't yet implemented role-based access control to restrict endpoints to specific user types (organizers, staff, attendees).

This distinction is important:

- Authentication confirms identity (Who are you?)
- Authorization controls access (What are you allowed to do?)

Summary

- Added the list event endpoint to the `EventsController`

UI Testing

Testing the event listing functionality in the user interface allows us to verify that our Spring Boot backend is correctly integrated with our frontend application and that users can view their events with pagination support.

Testing the List Events Endpoint

Let's verify that our list events functionality works correctly by testing it through the user interface:

Step 1: Compile and Run

First, we need to ensure our application builds correctly and is running:

- Run `mvn clean compile` to build the application
- Start the Spring Boot application
- Navigate to the organizer's landing page in your browser
- Login using your Keycloak credentials

Step 2: Inspecting the Network Requests

To verify our backend is working correctly, we can use the browser's developer tools:

1. Open the Network tab in your browser's developer tools
2. Clear any existing network requests
3. Refresh the page

The browser will make a request to `/api/v1/events` with query parameters:

```
page=0&size=2
```

Step 3: Analyzing the Response

The response from our endpoint contains:

- A `content` array containing the event data
- Pagination metadata including:
    - `pageNumber`: Current page (starting from 0)
    - `pageSize`: Number of records per page
    - `totalPages`: Total number of pages
    - `totalElements`: Total number of events

Step 4: Testing Pagination

The user interface implements pagination controls:

- Events are displayed in pages of 2 items
- Navigate between pages using the pagination controls
- The UI updates to show the next set of events when clicking "Next"
- Event details displayed include:

- Start and end dates
- Sales period
- Description
- Venue information
- Ticket types

Summary

- We tested the list events functionality using the user interface

Module Summary

Key Concepts Covered

List Event Service

- Added the `listEventsForOrganizer` method to the `EventService` interface
- Added the `findByOrganizerId` method to the `EventRepository` interface
- Implemented `listEventsForOrganizer` method in the `EventServiceImpl` class

DTOs and Mappers

- Created the `ListEventResponseDto`
- Create the `ListEventTicketTypeResponseDto`
- Updated the `EventMapper`to map to these DTO classes

Event Controller

- Added the list event endpoint to the `EventsController`

UI Testing

- We tested the list events functionality using the user interface

Get Event Endpoint

Module Overview

Module Purpose

In this module we'll build an endpoint that allows users to retrieve detailed information about individual events.

Module Structure

1. Implement the get event service layer
2. Create DTOs and mappers methods
3. Implement the get event endpoint
4. Test the endpoint using the user interface

Learning Objectives

1. Implement the service method that retrieves event details
2. Implement the DTO classes and mapper methods needed for the get event endpoint
3. Implement the get event endpoint
4. Test that the get event endpoint works in the user interface

Implement Service Method

In this lesson, we'll implement the get event service layer functionality.

Add Method to the Repository

Let's add a new method to our `EventRepository` interface:

```
@Repository
public interface EventRepository extends JpaRepository<Event, UUID> {
   // Existing methods...

   Optional<Event> findByIdAndOrganizerId(UUID id, UUID organizerId);
}
```

This repository method combines two search criteria:

- The event's `id`
- The `organizerId` to ensure users can only access their own events

Update the Service Layer

Next, let's add the corresponding method to our `EventService` interface:

```
public interface EventService {
   // Existing methods...

   Optional<Event> getEventForOrganizer(UUID organizerId, UUID id);
}
```

Finally, let's implement the method in our `EventServiceImpl` class:

```
@Service
@RequiredArgsConstructor
public class EventServiceImpl implements EventService {
   private final EventRepository eventRepository;

   // Existing methods...

   @Override
   public Optional<Event> getEventForOrganizer(UUID organizerId, UUID id) {
       // Pass the parameters to the repository method
       return eventRepository.findByIdAndOrganizerId(id, organizerId);
   }
}
```

Summary

- Added repository method to find events by ID and organizer ID
- Created service interface method for retrieving events
- Implemented service method to fetch event details using repository

Create DTO Classes

In this lesson, we'll implement the Data Transfer Objects (DTOs) needed to implement our get event endpoint.

Create the DTOs

We'll create two DTOs - one for the event details and another for its ticket types.

Let's start with the ticket type DTO:

```
@Data
@AllArgsConstructor
@NoArgsConstructor
public class GetEventDetailsTicketTypesResponseDto {
   private UUID id;
   private String name;
   private Double price;
   private String description;
   private Integer totalAvailable;
   private LocalDateTime createdAt;
   private LocalDateTime updatedAt;
}
```

Next, let's create the event details DTO:

```
@Data
@AllArgsConstructor
@NoArgsConstructor
public class GetEventDetailsResponseDto {
   private UUID id;
   private String name;
   private LocalDateTime start;
   private LocalDateTime end;
   private String venue;
   private LocalDateTime salesStart;
   private LocalDateTime salesEnd;
   private EventStatusEnum status;
   private List<GetEventDetailsTicketTypesResponseDto> ticketTypes = new ArrayList<>();
   private LocalDateTime createdAt;
   private LocalDateTime updatedAt;
}
```

Adding Mapper Methods

We need to add methods to our `EventMapper` interface to convert our entities to these DTOs:

```
@Mapper(componentModel = "spring", unmappedTargetPolicy = ReportingPolicy.IGNORE)
public interface EventMapper {
   // ... existing methods ...

   GetEventDetailsTicketTypesResponseDto toGetEventDetailsTicketTypesResponseDto(TicketType ticketT
```

```
   GetEventDetailsResponseDto toGetEventDetailsResponseDto(Event event);
}
```

Summary

- Created `GetEventDetailsResponseDto` to represent complete event information
- Created `GetEventDetailsTicketTypeResponseDto` to represent ticket type details
- Added mapper methods to convert entities to DTOs

Implement Controller Endpoint

In this lesson, we'll implement a controller endpoint that lets event organizers retrieve the details of a specific event.

Implement the Controller Endpoint

The get event endpoint needs to handle both successful and unsuccessful requests. Let's implement this in our `EventController`:

```
@GetMapping(path = "/{eventId}")
public ResponseEntity<GetEventDetailsResponseDto> getEvent(
       @AuthenticationPrincipal Jwt jwt,
       @PathVariable UUID eventId
) {
   // Get the user's ID from the JWT token
   UUID userId = parseUserId(jwt);

   // Call the service layer and transform the response
   return eventService.getEventForOrganizer(userId, eventId)
           .map(eventMapper::toGetEventDetailsResponseDto)
           .map(ResponseEntity::ok)
           .orElse(ResponseEntity.notFound().build());
}
```

Let's break down what's happening:

- We use `@GetMapping` with a path parameter `{eventId}` to capture the event identifier
- The `@PathVariable` annotation maps the URL path parameter to our method parameter
- We extract the user ID from the JWT token using our helper method
- We use the service layer to find the event, which returns an `Optional`
- We chain `map` operations to convert the event to a DTO and wrap it in a response
- If no event is found, we return a 404 Not Found response

Error Handling

Our endpoint handles two main cases:

- When the event exists and belongs to the organizer, it returns HTTP 200 with the event details
- When the event doesn't exist or doesn't belong to the organizer, it returns HTTP 404

This approach follows REST best practices by using standard HTTP status codes to communicate the outcome of the request.

Summary

- Implemented a GET endpoint to retrieve event details by ID
- Used Optional to handle cases where events aren't found

UI Testing

In this lesson, we'll verify that our get event endpoint functions correctly through the user interface.

By testing through the UI, we can ensure our endpoint not only returns data but also that the data is correctly displayed to users.

Testing the API Response

First, let's prepare our development environment:

1. Clean and compile the project to ensure no issues with Lombok or MapStruct:

```
./mvnw clean compile
```

You may notice some warnings about the use of `builder`, but since we're not using that feature, we can proceed.

Start the application and navigate to the organizer's landing page.

After logging in, go to the "Create an Event" page where you'll see your existing events.

Examining the Network Traffic

When clicking the "Edit" button for an event, we can observe the API request:

Open your browser's Developer Tools (F12) and select the Network tab

Click the "Edit" button for Test Event 2

Observe the GET request:

- URL pattern: `api/v1/events/{uid}`
- Response status: HTTP 200
- Response body contains:

```
{
 "id": "...",
 "name": "Test Event 2",
 "start": "...",
 "end": "...",
 "venue": {...},
 "published": false,
 "ticketTypes": [...],
 "createdAt": "...",
 "updatedAt": "..."
}
```

Verifying UI Display

The UI should correctly display all event details:

- Event name

- Event dates
- Venue information
- Sales period dates
- Ticket types (including capacity)
- Publication status

The fact that all this information displays correctly confirms that:

- The API endpoint is working
- The response format is correct
- The UI can properly parse and display the data

Summary

- Tested the get event endpoints works in the user interface

Module Summary

Key Concepts Covered

Implement Service Method

- Added repository method to find events by ID and organizer ID
- Created service interface method for retrieving events
- Implemented service method to fetch event details using repository

Create DTO Classes

- Created `GetEventDetailsResponseDto` to represent complete event information
- Created `GetEventDetailsTicketTypeResponseDto` to represent ticket type details
- Added mapper methods to convert entities to DTOs

Implement Controller Endpoint

- Implemented a GET endpoint to retrieve event details by ID
- Used Optional to handle cases where events aren't found

UI Testing

- Tested the get event endpoints works in the user interface

Update Event

Module Overview

In this module, we'll build the functionality that allows event organizers to update their events after they've been created.

Module Structure

1. Design the update event feature and create necessary interfaces
2. Create custom exceptions for error handling
3. Implement the service layer functionality
4. Create DTOs and mappers
5. Implement the controller endpoint
6. Test the feature through the user interface

Learning Objectives

1. Create the objects and interface declaration required to implement the update event endpoint
2. Implement the exceptions we will need to implement the update event functionality
3. Implement the update event functionality in the service layer
4. Implement the DTOs and mappers required to implement the update event endpoint in the presentation layer
5. Implement the update event functionality in the events controller
6. Test that update event endpoint works in the user interface

Update Event Design

In this lesson, we'll create the objects and interface declaration needed for implementing the update event endpoint, building on our existing event management functionality.

Design Overview

Let's start by examining what we need for our update functionality.

When updating an event, we want to replace all the event data with new data, except for system-managed fields like `id`, `createdAt`, and `updatedAt`.

Here's how we'll structure our update objects:

```
@Data
@AllArgsConstructor
@NoArgsConstructor
public class UpdateEventRequest {
   private UUID id;
   private String name;
   private LocalDateTime start;
   private LocalDateTime end;
   private String venue;
   private LocalDateTime salesStart;
   private LocalDateTime salesEnd;
   private EventStatusEnum status;
   private List<UpdateTicketTypeRequest> ticketTypes = new ArrayList<>();
}
```

```
@Data
@AllArgsConstructor
@NoArgsConstructor
public class UpdateTicketTypeRequest {
   private UUID id;
   private String name;
   private Double price;
   private String description;
   private Integer totalAvailable;
}
```

Service Interface Declaration

We need to declare our update method in the `EventService` interface.

The method needs to:

- Take the organizer's ID to verify ownership
- Take the event ID to identify which event to update
- Accept the update request containing the new data
- Return the updated event

Here's the interface declaration:

```
public interface EventService {
   // ... existing methods ...

   Event updateEventForOrganizer(UUID organizerId, UUID id, UpdateEventRequest event);
}
```

Summary

- Added the `UpdateTicketTypeRequest` class
- Added the `updateEventForOrganizer` method to the `EventService` interface

Exceptions

In this lesson, we'll implement the exceptions needed for updating events in our ticket platform. We'll create custom exceptions to handle various error scenarios that could occur during event updates, making our application more robust and user-friendly.

Custom Exceptions

Let's create three custom exceptions that extend our base `EventTicketException` class:

```
public class EventNotFoundException extends EventTicketException {
   public EventNotFoundException() {
   }

   public EventNotFoundException(String message) {
       super(message);
   }

   public EventNotFoundException(String message, Throwable cause) {
       super(message, cause);
   }

   public EventNotFoundException(Throwable cause) {
       super(cause);
   }

   public EventNotFoundException(String message, Throwable cause, boolean enableSuppression, boolea
       super(message, cause, enableSuppression, writableStackTrace);
   }
}
```

```
public class TicketTypeNotFoundException extends EventTicketException {
   public TicketTypeNotFoundException() {
   }

   public TicketTypeNotFoundException(String message) {
       super(message);
   }

   public TicketTypeNotFoundException(String message, Throwable cause) {
       super(message, cause);
   }

   public TicketTypeNotFoundException(Throwable cause) {
       super(cause);
   }

   public TicketTypeNotFoundException(String message, Throwable cause, boolean enableSuppression, b
       super(message, cause, enableSuppression, writableStackTrace);
   }
}
```

```
public class EventUpdateException extends EventTicketException {
   public EventUpdateException() {
   }

   public EventUpdateException(String message) {
       super(message);
   }

   public EventUpdateException(String message, Throwable cause) {
       super(message, cause);
   }

   public EventUpdateException(Throwable cause) {
       super(cause);
   }

   public EventUpdateException(String message, Throwable cause, boolean enableSuppression, boolean
       super(message, cause, enableSuppression, writableStackTrace);
   }
}
```

Each exception serves a specific purpose:

- `EventNotFoundException` - When a requested event doesn't exist
- `TicketTypeNotFoundException` - When a referenced ticket type can't be found
- `EventUpdateException` - For general update-related errors

Exception Handler

We'll add these exceptions to our global exception handler to ensure consistent error responses:

```
@RestControllerAdvice
@Slf4j
public class GlobalExceptionHandler {

   @ExceptionHandler(EventUpdateException.class)
   public ResponseEntity<ErrorDto> handleEventUpdateException(EventUpdateException ex) {
       log.error("Caught EventUpdateException", ex);
       ErrorDto errorDto = new ErrorDto();
       errorDto.setError("Unable to update event");
       return new ResponseEntity<>(errorDto, HttpStatus.BAD_REQUEST);
   }

   @ExceptionHandler(TicketTypeNotFoundException.class)
   public ResponseEntity<ErrorDto> handleTicketTypeNotFoundException(TicketTypeNotFoundException ex
       log.error("Caught TicketTypeNotFoundException", ex);
       ErrorDto errorDto = new ErrorDto();
       errorDto.setError("Ticket type not found");
       return new ResponseEntity<>(errorDto, HttpStatus.BAD_REQUEST);
   }

   @ExceptionHandler(EventNotFoundException.class)
```

```
    public ResponseEntity<ErrorDto> handleEventNotFoundException(EventNotFoundException ex) {
        log.error("Caught EventNotFoundException", ex);
        ErrorDto errorDto = new ErrorDto();
        errorDto.setError("Event not found");
        return new ResponseEntity<>(errorDto, HttpStatus.BAD_REQUEST);
    }

    // Other handler methods
}
```

All these exceptions return HTTP 400 Bad Request responses, as they represent client-side errors.

Summary

- Added the `EventNotFoundException` exception
- Added the `EventUpdateException` exception
- Added the `TicketTypeNotFoundException` exception
- Updated the `GlobalExceptionHandler` to handle the new exceptions

Update Event Service

In this lesson, we'll implement the update event functionality in the service layer, which allows event organizers to modify existing events and their associated ticket types.

Service Layer Implementation

The update functionality needs to handle both the event details and its ticket types, ensuring data consistency and proper validation.

Let's implement the `updateEventForOrganizer` method in our service:

```
@Override
@Transactional
 public Event updateEventForOrganizer(UUID organizerId, UUID id, UpdateEventRequest event) {
    if(null == event.getId()) {
        throw new EventUpdateException("Event ID cannot be null");
    }

    if(!id.equals(event.getId())) {
        throw new EventUpdateException("Cannot update the ID of an event");
    }

    Event existingEvent = eventRepository
            .findByIdAndOrganizerId(id, organizerId)
            .orElseThrow(() -> new EventNotFoundException(
                    String.format("Event with ID '%s' does not exist", id))
            );

    existingEvent.setName(event.getName());
    existingEvent.setStart(event.getStart());
    existingEvent.setEnd(event.getEnd());
    existingEvent.setVenue(event.getVenue());
    existingEvent.setSalesStart(event.getSalesStart());
    existingEvent.setSalesEnd(event.getSalesEnd());
    existingEvent.setStatus(event.getStatus());

    Set<UUID> requestTicketTypeIds = event.getTicketTypes()
            .stream()
            .map(UpdateTicketTypeRequest::getId)
            .filter(Objects::nonNull)
            .collect(Collectors.toSet());

    existingEvent.getTicketTypes().removeIf(existingTicketType ->
            !requestTicketTypeIds.contains(existingTicketType.getId())
    );

    Map<UUID, TicketType> existingTicketTypesIndex = existingEvent.getTicketTypes().stream()
            .collect(Collectors.toMap(TicketType::getId, Function.identity()));

    for(UpdateTicketTypeRequest ticketType : event.getTicketTypes()) {
        if(null == ticketType.getId()) {
            // Create
            TicketType ticketTypeToCreate = new TicketType();
```

```
            ticketTypeToCreate.setName(ticketType.getName());
            ticketTypeToCreate.setPrice(ticketType.getPrice());
            ticketTypeToCreate.setDescription(ticketType.getDescription());
            ticketTypeToCreate.setTotalAvailable(ticketType.getTotalAvailable());
            ticketTypeToCreate.setEvent(existingEvent);
            existingEvent.getTicketTypes().add(ticketTypeToCreate);

        } else if(existingTicketTypesIndex.containsKey(ticketType.getId())) {
            // Update
            TicketType existingTicketType = existingTicketTypesIndex.get(ticketType.getId());
            existingTicketType.setName(ticketType.getName());
            existingTicketType.setPrice(ticketType.getPrice());
            existingTicketType.setDescription(ticketType.getDescription());
            existingTicketType.setTotalAvailable(ticketType.getTotalAvailable());
        } else {
            throw new TicketTypeNotFoundException(String.format(
                    "Ticket type with ID '%s' does not exist", ticketType.getId()
            ));
        }
    }

    return eventRepository.save(existingEvent);
}
```

Handle Orphaned Types

We'll also need to update our `Event` entity to handle orphaned types:

```
@Entity
@Table(name = "events")
@Getter
@Setter
@NoArgsConstructor
@AllArgsConstructor
@Builder
public class Event {

   // ...

   @OneToMany(mappedBy = "event", cascade = CascadeType.ALL, orphanRemoval = true)
   private List<TicketType> ticketTypes = new ArrayList<>();

   //...
}
```

Summary

- Implemented the `updateEventForOrganizer` method in the `EventServiceImpl` class
- Updated the `Event` class to handle orphaned `TicketTypes`

DTOs & Mappers

In this lesson, we'll implement the DTOs and mappers needed for updating events through our presentation layer, following the same pattern we used for creating events.

Data Transfer Objects

Let's create dedicated DTOs for updating events while maintaining separation from our create event DTOs.

First, let's create the ticket type update DTOs:

```
@Data
@AllArgsConstructor
@NoArgsConstructor
public class UpdateTicketTypeRequestDto {

   private UUID id;

   @NotBlank(message = "Ticket type name is required")
   private String name;

   @NotNull(message = "Price is required")
   @PositiveOrZero(message = "Price must be zero or greater")
   private Double price;

   private String description;

   private Integer totalAvailable;
}
```

Now for the event update DTOs:

```
@Data
@AllArgsConstructor
@NoArgsConstructor
public class UpdateEventRequestDto {

   @NotNull(message = "Event ID must be provided")
   private UUID id;

   @NotBlank(message = "Event name is required")
   private String name;

   private LocalDateTime start;

   private LocalDateTime end;

   @NotBlank(message = "Venue information is required")
   private String venue;

   private LocalDateTime salesStart;

   private LocalDateTime salesEnd;
```

```
    @NotNull(message = "Event status must be provided")
    private EventStatusEnum status;

    @NotEmpty(message = "At least one ticket type is required")
    @Valid
    private List<UpdateTicketTypeRequestDto> ticketTypes;
}
```

Mapper Updates

We need to extend our `EventMapper` interface to handle the new update DTOs:

```
@Mapper(componentModel = "spring", unmappedTargetPolicy = ReportingPolicy.IGNORE)
public interface EventMapper {
    // Existing mappings...

    UpdateTicketTypeRequest fromDto(UpdateTicketTypeRequestDto dto);

    UpdateEventRequest fromDto(UpdateEventRequestDto dto);

    UpdateTicketTypeResponseDto toUpdateTicketTypeResponseDto(TicketType ticketType);

    UpdateEventResponseDto toUpdateEventResponseDto(Event event);
}
```

Summary

- Created DTOs required for event updating
- Updated the `EventMapper` to handle the new event update classes

Event Controller

In this lesson, we'll implement a REST endpoint to update events.

Understanding the Update Event Endpoint

The update event endpoint uses HTTP PUT to support full updates of event resources. Let's look at how we implement this endpoint in our event controller:

```
@PutMapping(path = "/{eventId}")
public ResponseEntity<UpdateEventResponseDto> updateEvent(
       @AuthenticationPrincipal Jwt jwt,
       @PathVariable UUID eventId,
       @Valid @RequestBody UpdateEventRequestDto updateEventRequestDto){
   UpdateEventRequest updateEventRequest = eventMapper.fromDto(updateEventRequestDto);
   UUID userId = parseUserId(jwt);

   Event updatedEvent = eventService.updateEventForOrganizer(
           userId, eventId, updateEventRequest
   );

   UpdateEventResponseDto updateEventResponseDto = eventMapper.toUpdateEventResponseDto(updatedEven

   return ResponseEntity.ok(updateEventResponseDto);
}
```

Summary

- Implemented the update event endpoint on the `EventController`

UI Testing

In this lesson, we'll test the user interface for updating events in our ticketing platform.

We'll use the browser's development tools to inspect the HTTP requests and responses as we modify an existing event, ensuring our update functionality works correctly.

Testing the Update Event UI

Let's walk through the process of updating an event through the user interface and verify that our endpoint is working correctly.

First, we need to launch our application and navigate to the UI:

- Run `mvn clean compile` to ensure no issues with MapStruct or Lombok
- Start the Spring Boot application
- Navigate to the organizer's landing page
- Log in with organizer credentials
- Click through to the list events page

When viewing an existing event with complete information (dates, venue, ticket types), clicking the edit button shows us the edit event page populated with current event data via the `GET /api/v1/events/{id}` endpoint.

Making Updates

Let's modify various aspects of the event to test our update functionality:

- Update the event name by adding "updated" suffix
- Adjust event dates and times
- Modify venue information
- Change ticket sales period
- Update existing ticket type details
- Add a new ticket type

Verifying the Update

After the update, we can verify the changes by returning to the edit page and examining the fresh GET request, confirming all our modifications were saved correctly.

Summary

- Tested the event update functionality in the user interface

Module Summary

Key Concepts Covered

Update Event Design

- Added the `UpdateTicketTypeRequest` class
- Added the `updateEventForOrganizer` method to the `EventService` interface

Exceptions

- Added the `EventNotFoundException` exception
- Added the `EventUpdateException` exception
- Added the `TicketTypeNotFoundException` exception
- Updated the `GlobalExceptionHandler` to handle the new exceptions

Update Event Service

- Implemented the `updateEventForOrganizer` method in the `EventServiceImpl` class
- Updated the `Event` class to handle orphaned `TicketTypes`

DTOs & Mappers

- Created DTOs required for event updating
- Updated the `EventMapper` to handle the new event update classes

Event Controller

- Implemented the update event endpoint on the `EventController`

UI Testing

- Tested the event update functionality in the user interface

Delete Event

Module Overview

In this module, we'll build the delete event functionality for our ticketing platform.

Module Structure

1. Implement the delete event service layer
2. Implement the delete event controller endpoint
3. Test the delete functionality in the UI

Learning Objectives

1. Implement the delete events service layer
2. Implement the delete event endpoint in the presentation layer
3. Test the delete endpoint in the user interface

Delete Event Service

In this lesson, we'll implement the `deleteEventForOrganizer` method in our service layer, allowing event organizers to delete their events from the system.

Service Implementation

Let's break down the implementation of the delete event functionality:

```
  void deleteEventForOrganizer(UUID organizerId, UUID eventId);
```

```
@Override
@Transactional
public void deleteEventForOrganizer(UUID organizerId, UUID id) {
   // Get the event and delete it if found
   getEventForOrganizer(organizerId, id).ifPresent(eventRepository::delete);
}
```

This implementation:

- Uses the `@Transactional` annotation to ensure database operations are atomic
- Leverages the existing `getEventForOrganizer` method to verify ownership
- Only deletes the event if it exists and belongs to the specified organizer
- Returns void, silently handling cases where the event doesn't exist

The code follows a simple but effective pattern:

1. Reuses the existing `getEventForOrganizer` method which checks both existence and ownership
2. Uses the `ifPresent` method to only execute the delete operation if an event is found
3. Passes a method reference to `eventRepository::delete` for clean, functional programming style

Security Considerations

The delete operation is secure because:

- It verifies the organizer owns the event before deletion
- Uses the same authorization check as other event operations
- Operates within a transaction to maintain data consistency
- Prevents unauthorized users from deleting events they don't own

Error Handling

The current implementation takes a silent failure approach when:

- The event doesn't exist
- The organizer doesn't own the event
- The event ID is invalid

This approach might need to be revisited if explicit error feedback becomes a requirement.

Summary

- Added the `deleteEventForOrganizer` method to the service layer

Delete Event Endpoint

In this lesson, we'll build the delete event endpoint in our REST controller.

Implementation

The delete endpoint follows REST conventions by using the HTTP DELETE method and accepting the event ID as a path variable. Let's add the endpoint to our `EventController`:

```
@DeleteMapping(path = "/{eventId}")
public ResponseEntity<Void> deleteEvent(
   @AuthenticationPrincipal Jwt jwt,
   @PathVariable UUID eventId
) {
   UUID userId = parseUserId(jwt);
   eventService.deleteEventForOrganizer(userId, eventId);
   return ResponseEntity.noContent().build();
}
```

Summary

- Implemented the delete event endpoint

UI Testing

In this lesson, we'll test the delete event functionality through the user interface.

Testing the Delete Operation

Before testing the delete functionality in the browser, we should ensure our application compiles correctly:

```
# Run these commands in your terminal
./mvnw clean compile
```

User Interface Elements

The delete functionality appears as a delete button in the bottom right corner of each event card.

When clicked, it triggers a confirmation dialog to prevent accidental deletions:

- The dialog displays the event name and asks for confirmation
- Users can choose to cancel (which closes the dialog) or continue with the deletion

Network Communication

To observe the backend communication, open your browser's developer tools and select the Network tab.

When deleting an event:

1. The frontend sends a DELETE request to `/api/v1/events/{eventId}`
2. The server responds with HTTP status code 204 (No Content) on success
3. The UI automatically refreshes to show the updated list of events

Summary

- Tested the delete event functionality works in the frontend

Module Summary

Key Concepts Covered

Delete Event Service

- Added the `deleteEventForOrganizer` method to the service layer

Delete Event Endpoint

- Implemented the delete event endpoint

UI Testing

- Tested the delete event functionality works in the frontend

List Published Events

Module Overview

In this module, we'll build a feature that lets users see all the published events in our ticketing system.

We'll use this endpoint to populate the attendee landing page.

Module Structure

1. Implement the list published events service layer
2. Implement DTOs and mappers
3. Implement the REST endpoint
4. Test the endpoint using the user interface

Learning Objectives

1. Implement list published events service layer
2. Implement the DTOs and mappers required for the list published events endpoint
3. Implement the list published event endpoint in the presentation layer
4. Test the list published event endpoint in the user interface

List Published Events Service

In this lesson, we'll implement the list published events service layer.

Service Layer Implementation

Let's add the `listPublishedEvents` method to our `EventService` interface:

```
public interface EventService {
   // ... existing methods ...

   Page<Event> listPublishedEvents(Pageable pageable);
}
```

Next, we'll implement the method in our `EventServiceImpl` class:

```
@Override
public Page<Event> listPublishedEvents(Pageable pageable) {
   // Use the repository to find events with PUBLISHED status
   return eventRepository.findByStatus(EventStatusEnum.PUBLISHED, pageable);
}
```

Repository Layer Implementation

To support our service layer, we need to add a method to our `EventRepository` interface that can find events by their status:

```
@Repository
public interface EventRepository extends JpaRepository<Event, UUID> {
   // ... existing methods ...

   // Find events by their status (e.g., PUBLISHED)
   Page<Event> findByStatus(EventStatusEnum status, Pageable pageable);
}
```

The `findByStatus` method follows Spring Data JPA's method naming convention, which automatically generates the correct query based on the method name.

Summary

- Added the `findByStatus` method to the `EventRepository` interface
- Added the `listPublishedEvents` method to the `EventService` interface
- Implemented the `listPublishedEvents` method in the `EventServiceImpl` class

DTOs & Mappers

In this lesson, we'll implement the DTOs and mappers needed for displaying published events on the attendee landing page.

Implement the DTO

Let's create the `ListPublishedEventResponseDto` class:

```
@Data
@AllArgsConstructor
@NoArgsConstructor
public class ListPublishedEventResponseDto {
   private UUID id;
   private String name;
   private LocalDateTime start;
   private LocalDateTime end;
   private String venue;
}
```

This DTO includes only the fields needed for the event cards on the landing page: the event's ID, name, start and end times, and venue.

Adding the Mapper Method

Now we'll add a method to our `EventMapper` interface to convert `Event` entities to our new DTO:

```
@Mapper(componentModel = "spring", unmappedTargetPolicy = ReportingPolicy.IGNORE)
public interface EventMapper {
   // ... existing methods ...

   ListPublishedEventResponseDto toListPublishedEventResponseDto(Event event);
}
```

The MapStruct library will automatically generate the implementation because our DTO field names match the `Event` entity's field names.

Summary

- Created the `ListPublishedEventResponseDto` class
- Added the `toListPublishedEventResponseDto` method to the `EventMapper` interface

List Published Event Endpoint

In this lesson, we'll create an endpoint that allows anyone to view published events, enabling potential attendees to browse available events without needing to log in first.

Creating the Published Events Controller

Let's create a dedicated controller for published events, keeping it separate from our existing event management endpoints.

```
@RestController
@RequestMapping(path = "/api/v1/published-events")
@RequiredArgsConstructor
public class PublishedEventController {

   private final EventService eventService;
   private final EventMapper eventMapper;

   @GetMapping
   public ResponseEntity<Page<ListPublishedEventResponseDto>> listPublishedEvents(Pageable pageable
       // Map the events to DTOs and return them in the response
       return ResponseEntity.ok(eventService.listPublishedEvents(pageable)
           .map(eventMapper::toListPublishedEventResponseDto));
   }
}
```

Configuring Public Access

To make the endpoint accessible without authentication, we need to update our security configuration.

```
@Configuration
public class SecurityConfig {
   @Bean
   public SecurityFilterChain filterChain(
           HttpSecurity http,
           UserProvisioningFilter userProvisioningFilter) throws Exception {
       http
           .authorizeHttpRequests(authorize ->
               authorize
                   // Allow public access to published events
                   .requestMatchers(HttpMethod.GET, "/api/v1/published-events").permitAll()
                   // All other endpoints require authentication
                   .anyRequest().authenticated())
           // ... rest of the configuration
       return http.build();
   }
}
```

Summary

- Created the `PublishedEventController` class

- Implemented the list published events endpoint
- Made the list published events endpoint public

UI Testing

Now that we've built our list published events endpoint, let's validate it works correctly by testing it through our user interface, where we'll see our events displayed on the landing page.

Setting Up for Testing

Before we begin testing in the browser, we need to ensure our application is running correctly:

1. Clean and compile the project to ensure everything is in order
2. Start the application
3. Navigate to `http://localhost:5173` in your browser

Testing the User Interface

The landing page has been updated to showcase published events to attendees.

The page now features:

- A hero image at the top
- A login button
- Event cards displaying published events

Each event card shows:

- A placeholder random image
- The event name
- Venue details
- Event dates

An important aspect to test is that these events are visible without requiring authentication.

Verifying the API Call

We can confirm our endpoint is working correctly by examining the network traffic:

1. Open your browser's developer tools
2. Navigate to the Network tab
3. Filter to show API calls
4. Refresh the page

You should see a successful HTTP 200 response from `api/v1/published-events`.

The response includes events marked as published, with our test demonstrating that both "Test Event 3" and "Test Event 4" are visible.

To verify the authentication requirement is working as intended:

1. Log in as an organizer
2. Confirm you can see the same events in the organizer view
3. Log out

4. Verify the events remain visible on the landing page

Summary

- Tested the list published events endpoint works by using the user interface

Module Summary

Key Concepts Covered

List Published Events Service

- Added the `findByStatus` method to the `EventRepository` interface
- Added the `listPublishedEvents` method to the `EventService` interface
- Implemented the `listPublishedEvents` method in the `EventServiceImpl` class

DTOs & Mappers

- Created the `ListPublishedEventResponseDto` class
- Added the `toListPublishedEventResponseDto` method to the `EventMapper` interface

List Published Event Endpoint

- Created the `PublishedEventController` class
- Implemented the list published events endpoint
- Made the list published events endpoint public

UI Testing

- Tested the list published events endpoint works by using the user interface

Published Events Search

Module Overview

In this module, we'll add published event search functionality to our event ticket platform. This will allow users to find events they're interested in attending more easily.

Module Structure

1. Implement search functionality in the service layer
2. Update the controller to handle search requests
3. Test the search functionality through the user interface

Learning Objectives

1. Implement published event search functionality in the service layer
2. Update the published event controller to support optional searching
3. Test the search published event functionality by using the user interface

Search Service Layer

In this lesson, we'll implement search functionality for published events using PostgreSQL's text search capabilities in the service layer. This will allow users to find events by searching terms that match event names and venues.

PostgreSQL Text Search Query

Let's look at the SQL query that powers our search functionality:

```
@Query(value = "SELECT * FROM events WHERE " +
   "status = 'PUBLISHED' AND " +
   "to_tsvector('english', COALESCE(name, '') || ' ' || COALESCE(venue, '')) " +
   "@@ plainto_tsquery('english', :searchTerm)",
   countQuery = "SELECT count(*) FROM events WHERE " +
       "status = 'PUBLISHED' AND " +
       "to_tsvector('english', COALESCE(name, '') || ' ' || COALESCE(venue, '')) " +
       "@@ plainto_tsquery('english', :searchTerm)",
   nativeQuery = true)
Page<Event> searchEvents(@Param("searchTerm") String searchTerm, Pageable pageable);
```

This query:

- Uses PostgreSQL's `to_tsvector` to create a searchable text vector from the name and venue fields
- Applies `plainto_tsquery` to convert the search term into a format PostgreSQL can use
- Only returns events with PUBLISHED status
- Supports pagination through Spring Data JPA's `Pageable` parameter

Service Layer Implementation

The service layer implementation connects the repository query to our application:

```
@Override
public Page<Event> searchPublishedEvents(String query, Pageable pageable) {
   return eventRepository.searchEvents(query, pageable);
}
```

Summary

- Added the `searchEvents` custom query to the `EventRepository` interface
- Added the `searchPublishedEvents` method to the `EventService` interface
- Implemented the `searchPublishedEvents` method in the `EventServiceImpl` class

Search Controller Updates

In this lesson, we'll enhance the published events controller to support optional search functionality, allowing users to find events by searching through event names and venues.

Understanding Optional Search Parameters

The `@RequestParam` annotation in Spring Boot lets us add optional parameters to our endpoints.

By setting `required = false`, we tell Spring that the parameter is optional, meaning the endpoint will work both with and without the search parameter.

Here's how we update our controller:

```
@GetMapping
public ResponseEntity<Page<ListPublishedEventResponseDto>> listPublishedEvents(
   @RequestParam(required = false) String q,
   Pageable pageable) {

 Page<Event> events;
 if(null != q && !q.trim().isEmpty()) {
   events = eventService.searchPublishedEvents(q, pageable);
 } else {
   events = eventService.listPublishedEvents(pageable);
 }

 return ResponseEntity.ok(
     events.map(eventMapper::toListPublishedEventResponseDto)
 );
}
```

Search Flow Implementation

The controller now handles two different scenarios:

- When a search query is provided, it calls `searchPublishedEvents` with the query
- When no search query is provided, it calls `listPublishedEvents` to show all published events

The code checks if the query parameter `q` exists and isn't empty after trimming whitespace.

We use the query parameter name `q` as it's a common convention in search APIs and keeps our URLs clean and readable.

Summary

- Added the search published events endpoint the `PublishedEventsController`

UI Testing

In this lesson, we'll explore how to test the published event search functionality through the user interface.

Testing through the UI provides a practical way to validate that our search feature works as expected from the user's perspective.

Manual UI Testing

The search functionality allows users to find published events by matching text in the event details.

Let's walk through the testing process:

1. First, ensure your application is running by executing a clean compile:

```
./mvnw clean compile
```

   Navigate to your frontend application where you should see the published events displayed (in this case, "Test Event Three" and "Test Event Four").

   To properly test and monitor the search functionality, open your browser's developer tools (F12 in most browsers) and select the Network tab.

Testing Search Scenarios

Let's test different search scenarios to verify the functionality:

1. Basic Search Test:

- Enter "test" in the search field
- Click search
- Verify both events are returned
- Check the network tab shows a 200 status code

2. Specific Event Search:

- Enter "test event three"
- Verify only "Test Event Three" is displayed
- Enter "test event four"
- Verify only "Test Event Four" is displayed

3. Venue Details Search:

- Enter "details" in the search field
- Verify only events containing "details" in their venue information are displayed

4. Empty Search:

- Clear the search field
- Click search
- Verify all published events are displayed

When testing through the UI, pay attention to:

- The immediate response of the interface
- The network requests being made
- The accuracy of the returned results
- The handling of different search terms

Summary

- Tested the search event functionality using the user interface

Get Published Event

Module Overview

In this module, we'll implement the get published event details endpoint, which we will use to populate data on the published event page.

Module Structure

1. Implement the get published event functionality in the service layer
2. Implement the DTOs and mappers
3. Implement the get published event details endpoint
4. Test the functionality through the user interface

Learning Objectives

1. Implement the service layer functionality to get a published event
2. Implement the DTOs & mappers needed to implement the get published event endpoint in the presentation layer
3. Implement the get published event endpoint in the presentation layer
4. Test the get published event functionality by using the user interface

Service Layer

In this lesson, we'll implement the get published event functionality in our service layer.

Service Layer Implementation

Let's add the `getPublishedEvent` method to our event service interface:

```
public interface EventService {
   // Other methods...
   Optional<Event> getPublishedEvent(UUID id);
}
```

In the implementation class, we'll use our repository to find events that match both the ID and published status:

```
@Override
public Optional<Event> getPublishedEvent(UUID id) {
   // Only return events that are both published and match the ID
   return eventRepository.findByIdAndStatus(id, EventStatusEnum.PUBLISHED);
}
```

Repository Extension

To support this functionality, we need to add a custom query method to our repository:

```
public interface EventRepository extends JpaRepository<Event, UUID> {
   // Other methods...
   Optional<Event> findByIdAndStatus(UUID id, EventStatusEnum status);
}
```

This method follows Spring Data JPA's method naming conventions, creating a query that filters by both ID and status.

Summary

- Added the `findByIdAndStatus` method to the `EventRepository`
- Added the `getPublishedEvent` method to the `EventService` interface
- Implemented the `getPublishedEvent` method in the `EventServiceImpl` class

DTOs & Mappers

In this lesson, we'll implement the DTOs and mappers needed for the get published event endpoint in the presentation layer.

Implement the DTOs

Let's look at what we need to include in our DTOs:

```
@Data
@AllArgsConstructor
@NoArgsConstructor
public class GetPublishedEventDetailsResponseDto {
   private UUID id;
   private String name;
   private LocalDateTime start;
   private LocalDateTime end;
   private String venue;
   private List<GetPublishedEventDetailsTicketTypesResponseDto> ticketTypes = new ArrayList<>();
}
```

For ticket types, we'll create a separate DTO with only the necessary fields:

```
@Data
@AllArgsConstructor
@NoArgsConstructor
public class GetPublishedEventDetailsTicketTypesResponseDto {
   private UUID id;
   private String name;
   private Double price;
   private String description;
}
```

Implementing the Mappers

We need to add methods to our `EventMapper` interface to convert between our domain objects and DTOs:

```
@Mapper(componentModel = "spring", unmappedTargetPolicy = ReportingPolicy.IGNORE)
public interface EventMapper {
   // ... existing methods ...

   GetPublishedEventDetailsTicketTypesResponseDto toGetPublishedEventDetailsTicketTypesResponseDto(

   GetPublishedEventDetailsResponseDto toGetPublishedEventDetailsResponseDto(Event event);
}
```

The `@Mapper` annotation tells MapStruct to generate the implementation of these methods.

Summary

- Added the `GetPublishedEventDetailsResponseDto` and `GetPublishedEventDetailsTicketTypesResponseDto` DTO classes
- Added the `toGetPublishedEventDetailsResponseDto` and `toGetPublishedEventDetailsTicketTypesResponseDto` mapper methods

Get Published Event Endpoint

In this lesson, we'll implement the get published event details endpoint.

Implementing the Get Published Event Endpoint

We'll add a new endpoint to our `PublishedEventController` class that retrieves the details of a specific published event.

```
@GetMapping(path = "/{eventId}")
public ResponseEntity<GetPublishedEventDetailsResponseDto> getPublishedEventDetails(
   @PathVariable UUID eventId
) {
   return eventService.getPublishedEvent(eventId)
       .map(eventMapper::toGetPublishedEventDetailsResponseDto)
       .map(ResponseEntity::ok)
       .orElse(ResponseEntity.notFound().build());
}
```

Let's break down what this code does:

- The `@GetMapping` annotation with a path variable defines the URL pattern for this endpoint
- The method takes an `eventId` parameter that is extracted from the URL path
- It calls the `eventService.getPublishedEvent()` method to fetch the event details
- The result is mapped to our DTO using the `eventMapper`
- If an event is found, it returns a 200 OK response with the event details
- If no event is found, it returns a 404 Not Found response

Configuring Security

We need to ensure this endpoint is publicly accessible. Let's update the security configuration:

```
.requestMatchers(HttpMethod.GET, "/api/v1/published-events/**").permitAll()
```

This configuration uses a wildcard (`**`) to match any path after `/api/v1/published-events/`, including our new endpoint that includes the event ID.

Summary

- Implemented the get published event endpoint
- Updated `SecurityConfig` to make calls to make the get published event endpoint public

UI Testing

In this lesson, we'll test the get published event functionality using the user interface, ensuring our endpoint works correctly and delivers the expected event information to users.

Testing Through the User Interface

First, we need to compile and start our backend application:

```
# Clean and compile the application
mvn clean compile

# Start the Spring Boot application
mvn spring-boot:run
```

Once the application is running, we can navigate to the attendee landing page where we should see our test events listed.

When clicking on an event (like "Test Event Three"), the application makes a network request to our `/api/v1/published-events/{id}` endpoint.

Let's examine what happens in this request:

- The request returns a HTTP 200 OK status
- No authorization header is present, confirming the endpoint is public
- The response includes complete event details and ticket types

The UI displays several key pieces of information from the response:

- Event name
- Venue details
- Event dates
- Event image
- Ticket types with their respective prices
- Purchase options (though not yet implemented)

Summary

- Tested the get published event endpoint through the user interface

Module Summary

Key Concepts Covered

Service Layer

- Added the `findByIdAndStatus` method to the `EventRepository`
- Added the `getPublishedEvent` method to the `EventService` interface
- Implemented the `getPublishedEvent` method in the `EventServiceImpl` class

DTOs & Mappers

- Added the `GetPublishedEventDetailsResponseDto` and `GetPublishedEventDetailsTicketTypesResponseDto` DTO classes
- Added the `toGetPublishedEventDetailsResponseDto` and `toGetPublishedEventDetailsTicketTypesResponseDto` mapper methods

Get Published Event Endpoint

- Implemented the get published event endpoint
- Updated `SecurityConfig` to make calls to make the get published event endpoint public

UI Testing

- Tested the get published event endpoint through the user interface

New Users & Roles

Module Overview

In this module, we'll add users and roles to Keycloak to support both attendees who will purchase tickets and staff who will manage events.

This foundation will enable us to control what different users can do in our system.

Module Structure

1. Configure the attendee user and role
2. Configure the organizer role
3. Configure the staff user and role

Learning Objectives

1. Add the organizer role, attendee user and role to Keycloak
2. Add the staff user and role to Keycloak

Attendee User

In this lesson, we'll expand our Keycloak configuration by adding an attendee user and corresponding roles.

Adding the Attendee User

Creating a new user in Keycloak involves setting up basic user information and credentials.

Let's create a new attendee user with these steps:

1. Navigate to the Keycloak Admin Console at `localhost:1990`
2. Select the "event-ticket-platform" realm
3. Go to Users and click "Add User"
4. Fill in the following details:
    - Username: `attendee`
    - Email: `attendee@yourdomain.com`
    - First Name: `attendee`
    - Last Name: `user`

After creating the user, we need to set up their password:

1. Go to the Credentials tab
2. Set the password as "password" (Note: This is for development only)
3. Disable the "Temporary" option to prevent password reset requirements

Creating and Assigning Roles

Roles in Keycloak help us manage user permissions effectively.

We'll create two roles:

```
ROLE_ATTENDEE    // For regular event attendees
ROLE_ORGANIZER   // For event organizers
```

To create these roles:

1. Navigate to Realm Roles
2. Click "Create Role"
3. Enter `ROLE_ATTENDEE` for the first role
4. Repeat the process with `ROLE_ORGANIZER`

Next, we'll assign these roles:

For the attendee user:

- Go to Users → attendee → Role Mapping
- Click "Assign Role"
- Select `ROLE_ATTENDEE`

For the organizer user:

- Go to Users → organizer → Role Mapping
- Click "Assign Role"
- Select `ROLE_ORGANIZER`

Summary

- Added the attendee user to Keycloak
- Added the `ROLE_ATTENDEE` role to Keycloak
- Added the `ROLE_ORGANIZER` role to Keycloak

Staff User

In this lesson, we'll expand our user management system by adding a staff user and role to Keycloak.

Adding the Staff User

Creating a new staff user in Keycloak follows the same pattern we used for our other users, but with staff-specific details:

1. Navigate to the Users page in the Keycloak admin console
2. Click "Add User" and provide these details:
    - Username: `staff`
    - Email: `staff@example.com`
    - First Name: `staff`
    - Last Name: `user`

Creating the Staff Role

The staff role will be used to control access to staff-specific features in our application:

1. Navigate to "Realm Roles" in the sidebar
2. Click "Create Role"
3. Set the role name as `ROLE_STAFF`
4. Save the role

Assigning the Role

To connect our new user with their role:

1. Go back to Users and select the staff user
2. Click on "Role Mapping"
3. Choose "Assign Role"
4. Filter by realm roles
5. Select `ROLE_STAFF`
6. Confirm the assignment

Summary

- Added the staff user to Keycloak
- Added the `ROLE_STAFF` role to Keycloak

Module Summary

Key Concepts Covered

Attendee User

- Added the attendee user to Keycloak
- Added the `ROLE_ATTENDEE` role to Keycloak
- Added the `ROLE_ORGANIZER` role to Keycloak

Staff User

- Added the staff user to Keycloak
- Added the `ROLE_STAFF` role to Keycloak

Purchase Ticket

Qr Code Generation Design

In this lesson, we'll design the QR code generation functionality that will be used to create scannable tickets for event attendees.

Service Design Overview

The QR code generation process needs to be flexible and maintainable. Let's examine the key components we've established:

- The `QrCode` entity has been updated to store QR codes as text in the database
- The `id` field is now manually set rather than auto-generated
- A `QrCodeRepository` has been created for database operations
- A `QrCodeService` interface defines the core functionality
- A `QrCodeGenerationException` handles error cases

Database Storage Considerations

We're storing QR codes in the database as base64 encoded strings using a `TEXT` column type. Here's why this approach works for our current needs:

- The `TEXT` type allows for variable-length storage without the 255 character limit of `VARCHAR`
- Base64 encoding lets us store binary image data as text
- This approach requires minimal additional infrastructure

However, it's worth noting that this solution may need to be revised as the system scales, since storing images in the database can impact performance.

Error Handling

We've implemented a dedicated exception type for QR code generation failures:

```
public class QrCodeGenerationException extends EventTicketException {

 public QrCodeGenerationException() {
   super();
 }

 public QrCodeGenerationException(String message) {
   super(message);
 }

 public QrCodeGenerationException(String message, Throwable cause) {
   super(message, cause);
 }

 public QrCodeGenerationException(Throwable cause) {
   super(cause);
 }
```

```
 public QrCodeGenerationException(String message, Throwable cause, boolean enableSuppression,
     boolean writableStackTrace) {
   super(message, cause, enableSuppression, writableStackTrace);
 }
}
```

```
@ExceptionHandler(QrCodeGenerationException.class)
public ResponseEntity<ErrorDto> handleQrCodeGenerationException(QrCodeGenerationException ex) {
   log.error("Caught QrCodeGenerationException", ex);
   ErrorDto errorDto = new ErrorDto();
   errorDto.setError("Unable to generate QR Code");
   return new ResponseEntity<>(errorDto, HttpStatus.INTERNAL_SERVER_ERROR);
}
```

This provides clear error messages to clients while using the appropriate 500 status code, since QR code generation failures are server-side issues.

Service Interface

The `QrCodeService` interface is intentionally simple:

```
public interface QrCodeService {
   QrCode generateQrCode(Ticket ticket);
}
```

Summary

- Updated types in the `QrCode` entity
- Added `QrCodeGenerationException`
- Added `QrCodeService`
- Added `QrCodeRepository`

Qr Code Generation Service Layer

In this lesson, we'll implement the QR code generation functionality in our ticket platform's service layer using the ZXing library.

Setting Up QR Code Dependencies

To generate QR codes, we need to add the ZXing library dependencies to our project:

```
<dependency>
 <groupId>com.google.zxing</groupId>
 <artifactId>core</artifactId>
 <version>3.5.1</version>
</dependency>
<dependency>
 <groupId>com.google.zxing</groupId>
 <artifactId>javase</artifactId>
 <version>3.5.1</version>
</dependency>
```

Creating the QR Code Writer Bean

Let's create a configuration class to provide a `QRCodeWriter` bean:

```
@Configuration
public class QrCodeConfig {
   @Bean
   public QRCodeWriter qrCodeWriter() {
       return new QRCodeWriter();
   }
}
```

Implementing the QR Code Service

The QR code service implementation handles generating and storing QR codes:

```
@Service
@RequiredArgsConstructor
public class QrCodeServiceImpl implements QrCodeService {
   private static final int QR_HEIGHT = 300;
   private static final int QR_WIDTH = 300;

   private final QRCodeWriter qrCodeWriter;
   private final QrCodeRepository qrCodeRepository;

   @Override
   public QrCode generateQrCode(Ticket ticket) {
       try {
           // Generate a unique ID for the QR code
           UUID uniqueId = UUID.randomUUID();
           String qrCodeImage = generateQrCodeImage(uniqueId);
```

```
            // Create and save the QR code entity
            QrCode qrCode = new QrCode();
            qrCode.setId(uniqueId);
            qrCode.setStatus(QrCodeStatusEnum.ACTIVE);
            qrCode.setValue(qrCodeImage);
            qrCode.setTicket(ticket);

            return qrCodeRepository.saveAndFlush(qrCode);
        } catch(IOException | WriterException ex) {
            throw new QrCodeGenerationException("Failed to generate QR Code", ex);
        }
    }

    private String generateQrCodeImage(UUID uniqueId) throws WriterException, IOException {
        // Create a bit matrix for the QR code
        BitMatrix bitMatrix = qrCodeWriter.encode(
            uniqueId.toString(),
            BarcodeFormat.QR_CODE,
            QR_WIDTH,
            QR_HEIGHT
        );

        // Convert to BufferedImage
        BufferedImage qrCodeImage = MatrixToImageWriter.toBufferedImage(bitMatrix);

        // Convert to base64 string
        try(ByteArrayOutputStream baos = new ByteArrayOutputStream()) {
            ImageIO.write(qrCodeImage, "PNG", baos);
            byte[] imageBytes = baos.toByteArray();
            return Base64.getEncoder().encodeToString(imageBytes);
        }
    }
}
```

Summary

- Added the `zxing` dependency
- Implemented QRCode generation

Ticket Purchase Service Layer

In this lesson, we'll build the service layer functionality for purchasing tickets, implementing concurrent access handling and QR code generation to create a robust ticket purchasing system.

Understanding the Purchase Flow

The ticket purchase process involves several key steps:

1. Finding the user and ticket type in the database
2. Checking if tickets are still available
3. Creating a new ticket record
4. Generating a QR code for the ticket

Let's implement this in our `TicketTypeServiceImpl` class:

```
@Service
@RequiredArgsConstructor
public class TicketTypeServiceImpl implements TicketTypeService {
   private final UserRepository userRepository;
   private final TicketTypeRepository ticketTypeRepository;
   private final TicketRepository ticketRepository;
   private final QrCodeService qrCodeService;

   @Override
   @Transactional
   public Ticket purchaseTicket(UUID userId, UUID ticketTypeId) {
       // Look up the user
       User user = userRepository.findById(userId)
           .orElseThrow(() -> new UserNotFoundException(
               String.format("User with ID %s was not found", userId)
           ));

       // Get ticket type with pessimistic lock
       TicketType ticketType = ticketTypeRepository.findByIdWithLock(ticketTypeId)
           .orElseThrow(() -> new TicketTypeNotFoundException(
               String.format("Ticket type with ID %s was not found", ticketTypeId)
           ));

       // Check ticket availability
       int purchasedTickets = ticketRepository.countByTicketTypeId(ticketType.getId());
       Integer totalAvailable = ticketType.getTotalAvailable();

       if(purchasedTickets + 1 > totalAvailable) {
           throw new TicketsSoldOutException();
       }

       // Create new ticket
       Ticket ticket = new Ticket();
       ticket.setStatus(TicketStatusEnum.PURCHASED);
       ticket.setTicketType(ticketType);
       ticket.setPurchaser(user);
```

```
        // Save and generate QR code
        Ticket savedTicket = ticketRepository.save(ticket);
        qrCodeService.generateQrCode(savedTicket);

        return ticketRepository.save(savedTicket);
    }
}
```

Handling Concurrent Access

To prevent overselling tickets when multiple users try to purchase at the same time, we use a pessimistic lock:

```
@Repository
public interface TicketTypeRepository extends JpaRepository<TicketType, UUID> {
    @Query("SELECT tt FROM TicketType tt WHERE tt.id = :id")
    @Lock(LockModeType.PESSIMISTIC_WRITE)
    Optional<TicketType> findByIdWithLock(@Param("id") UUID id);
}
```

This ensures that when one user is purchasing a ticket, other users must wait until the transaction completes before they can access the same ticket type.

Error Handling

We handle several error cases:

- User not found
- Ticket type not found
- Tickets sold out
- QR code generation failure

These are caught and handled by our global exception handler to provide clear error messages to users.

Summary

- Implemented the initial purchase ticket functionality

Ticket Purchase Endpoint

In this lesson, we'll implement the ticket purchase endpoint.

Creating the Controller

Let's create a `TicketTypeController` that will handle ticket purchase requests:

```
@RestController
@RequiredArgsConstructor
@RequestMapping(path = "/api/v1/events/{eventId}/ticket-types")
public class TicketTypeController {

 private final TicketTypeService ticketTypeService;

 @PostMapping(path = "/{ticketTypeId}/tickets")
 public ResponseEntity<Void> purchaseTicket(
     @AuthenticationPrincipal Jwt jwt,
     @PathVariable UUID ticketTypeId
 ) {
   // Purchase the ticket using the service
   ticketTypeService.purchaseTicket(parseUserId(jwt), ticketTypeId);
   return new ResponseEntity<>(HttpStatus.NO_CONTENT);
 }
}
```

Understanding the Implementation

The purchase ticket endpoint follows RESTful conventions and includes several important components:

The endpoint URL structure follows the pattern `/api/v1/events/{eventId}/ticket-types/{ticketTypeId}/tickets`

We use `@PostMapping` since we're creating a new ticket resource

The controller accepts a JWT token to identify the purchaser and a ticket type ID to specify which type of ticket to purchase

We return HTTP 204 (No Content) on success since we don't need to send any information back to the client

Summary

- Implemented the purchase ticket endpoint

UI Testing

In this lesson, we'll test our ticket purchasing functionality through the user interface to verify the purchase flow works correctly and validate both the frontend and backend components work together properly.

Testing Through the UI

The first step in testing our ticket purchase functionality is to start our backend application.

We'll begin by cleaning and compiling our project using Maven, then starting the Spring Boot application.

Once the application is running, we can access the attendee landing page in our browser.

Here's the sequence of steps to test the ticket purchase:

- Select an event from the landing page
- Choose a ticket type (e.g. standard entry)
- Open browser dev tools to monitor network requests
- Click "Purchase Ticket" button
- Login with attendee credentials
- Enter mock payment information
- Complete purchase

Troubleshooting Issues

When testing the purchase flow, we may encounter errors that need debugging.

If you receive an HTTP 500 error after attempting to purchase, check the server logs.

In our case, we discovered a null pointer exception in the `QrCodeService` due to a missing `final` keyword on the repository field.

After fixing the code and restarting the server, the purchase flow should complete successfully with an HTTP 204 response.

Verifying the Purchase

To confirm the ticket purchase worked correctly:

```
-- Check the tickets table
SELECT * FROM tickets;

-- Check the QR codes table
SELECT * FROM qr_codes;
```

When examining the database records, verify:

- The purchaser ID matches the logged in user
- The ticket type ID matches the selected ticket
- A QR code was generated and stored
- The timestamps are correct

Summary

- Fixed NPE bug
- Created a ticket in the database
- Created a QRCode in the database

Module Summary

Key Concepts Covered

Qr Code Generation Design

- Updated types in the `QrCode` entity
- Added `QrCodeGenerationException`
- Added `QrCodeService`
- Added `QrCodeRepository`

Qr Code Generation Service Layer

- Added the `zxing` dependency
- Implemented QRCode generation

Ticket Purchase Service Layer

- Implemented the initial purchase ticket functionality

Ticket Purchase Endpoint

- Implemented the purchase ticket endpoint

UI Testing

- Fixed NPE bug
- Created a ticket in the database
- Created a QRCode in the database

Role Based Access

Module Overview

In this module, we'll add role-based security to our application.

This allows us to control which users can access specific parts of our application based on their assigned roles.

Module Structure

1. Extracting roles from JWTs
2. Lock down the events controller endpoints

Learning Objectives

1. Implement role extraction from the JWT
2. Lock down access to endpoints to certain roles

Extract roles

In this lesson we're going to extract the roles from the user's access token.

Understanding JWT Claims

The JWT used for authentication contains useful information about the user called claims.

Among these claims is the `realm_access` claim which contains the roles assigned to the user in Keycloak.

When we decode a JWT at jwt.io, we can see claims like `ROLE_ORGANIZER`, `ROLE_ATTENDEE`, and `ROLE_STAFF` under the `realm_access.roles` section.

Implementing Role Extraction

To extract roles from the JWT, we need to create a custom converter that transforms the JWT into Spring Security's internal representation.

Here's how we implement the `JwtAuthenticationConverter`:

```
@Component
public class JwtAuthenticationConverter implements Converter<Jwt, JwtAuthenticationToken> {

 @Override
 public JwtAuthenticationToken convert(Jwt jwt) {
   Collection<GrantedAuthority> authorities = extractAuthorities(jwt);
   return new JwtAuthenticationToken(jwt, authorities);
 }

 private Collection<GrantedAuthority> extractAuthorities(Jwt jwt) {
   Map<String, Object> realmAccess = jwt.getClaim("realm_access");

   if(null == realmAccess || !realmAccess.containsKey("roles")) {
     return Collections.emptyList();
   }

   @SuppressWarnings("unchecked")
   List<String> roles = (List<String>)realmAccess.get("roles");

   return roles.stream()
       .filter(role -> role.startsWith("ROLE_"))
       .map(SimpleGrantedAuthority::new)
       .collect(Collectors.toList());
 }
}
```

The converter does the following:

1. Gets the `realm_access` claim from the JWT
2. Checks if the claim exists and contains roles
3. Extracts the roles and converts them to Spring Security's `SimpleGrantedAuthority` objects

Configuring Security to Use the Converter

We need to update our security configuration to use our custom converter:

```
@Bean
public SecurityFilterChain filterChain(
   HttpSecurity http,
   UserProvisioningFilter userProvisioningFilter,
   JwtAuthenticationConverter jwtAuthenticationConverter) throws Exception {
 http
     .authorizeHttpRequests(authorize ->
         authorize
             .requestMatchers(HttpMethod.GET, "/api/v1/published-events/**").permitAll()
             // Catch all rule
             .anyRequest().authenticated())
     .csrf(csrf -> csrf.disable())
     .sessionManagement(session ->
         session.sessionCreationPolicy(SessionCreationPolicy.STATELESS))
     .oauth2ResourceServer(oauth2 ->
         oauth2.jwt(jwt ->
             jwt.jwtAuthenticationConverter(jwtAuthenticationConverter) // Add this
         ))
     .addFilterAfter(userProvisioningFilter, BearerTokenAuthenticationFilter.class);

 return http.build();
}
```

The key change is adding the `jwtAuthenticationConverter` to the JWT configuration.

Summary

- Added a custom JWT converter to extract a user's roles
- Updated `SecurityConfig` to use the JWT converter

Lock Down Endpoints

In this lesson, we're going to lock down certain endpoints to only certain roles.

Configuring Role-Based Access

Spring Security allows us to restrict access to API endpoints based on user roles.

In our application, we want to ensure that only users with the organizer role can access the events controller endpoints.

Here's how we implement this in the `SecurityConfig` class:

```
.authorizeHttpRequests(authorize ->
   authorize
       .requestMatchers(HttpMethod.GET, "/api/v1/published-events/**").permitAll()
       .requestMatchers("/api/v1/events").hasRole("ORGANIZER")
       // Catch all rule
       .anyRequest().authenticated())
```

The `.hasRole("ORGANIZER")` method is used to restrict access to users with the organizer role.

When using `hasRole()`, Spring Security automatically adds the `ROLE_` prefix to the role name, so we don't need to include it in our configuration.

Testing Role-Based Access

To verify our role-based access is working correctly, we can test with different user roles:

When logged in as an organizer user (with the `ROLE_ORGANIZER` role), requests to the events endpoint return HTTP 200 OK.

When logged in as an attendee user (without the `ROLE_ORGANIZER` role), requests to the events endpoint return HTTP 403 Forbidden.

Summary

- Locked down the events endpoints to only organizer users

Module Summary

Key Concepts Covered

Extract roles

- Added a custom JWT converter to extract a user's roles
- Updated `SecurityConfig` to use the JWT converter

Lock Down Endpoints

- Locked down the events endpoints to only organizer users

List Ticket

Module Overview

In this module, we'll build the functionality that allows users list their purchased tickets.

Module Structure

1. Implement the list ticket service layer functionality
2. Create DTOs and mappers
3. Implement the list ticket endpoint
4. Test through the user interface

Learning Objectives

1. Implement the service layer to list a ticket
2. Implement the DTOs and mappers to list a ticket
3. Implement the list ticket endpoint
4. Test the list ticket endpoint by using the user interface

List Ticket Service Layer

In this lesson, we're going to implement the list ticket service layer.

Repository Method Implementation

The first step is to add a method to the `TicketRepository` interface that will retrieve tickets for a specific user.

We'll add a method called `findByPurchaserId` that takes two parameters:

```
Page<Ticket> findByPurchaserId(UUID purchaserId, Pageable pageable);
```

This method will return a page of tickets, which allows for pagination of results.

Spring Data JPA will automatically implement this method based on the method name, as it understands the relationship between a `Ticket` and its purchaser.

Service Interface Creation

Next, we'll define the contract for our ticket service by creating the `TicketService` interface:

```
public interface TicketService {
 Page<Ticket> listTicketsForUser(UUID userId, Pageable pageable);
}
```

This interface declares a single method that will retrieve a page of tickets for a given user ID.

Service Implementation

Finally, we'll create the implementation of our `TicketService` interface:

```
@Service
@RequiredArgsConstructor
public class TicketServiceImpl implements TicketService {

 private final TicketRepository ticketRepository;

 @Override
 public Page<Ticket> listTicketsForUser(UUID userId, Pageable pageable) {
   return ticketRepository.findByPurchaserId(userId, pageable);
 }
}
```

The implementation is straightforward - it simply delegates to the repository method we created earlier.

Summary

- Added the `findByPurchaserId` method to `TicketRepository`
- Implemented the `TicketService` with `listTicketsForUser` method

List Ticket DTO & Mapper

In this lesson we'll implement the DTOs and mappers that we need for the list tickets endpoint.

Creating the List Ticket Response DTOs

We'll start by creating two Data Transfer Objects (DTOs) to represent ticket information when listing tickets.

The first DTO, `ListTicketResponseDto`, will contain the main ticket information:

```
@Data
@AllArgsConstructor
@NoArgsConstructor
public class ListTicketResponseDto {
 private UUID id;
 private TicketStatusEnum status;
 private ListTicketTicketTypeResponseDto ticketType;
}
```

The second DTO, `ListTicketTicketTypeResponseDto`, will contain information about the ticket type:

```
@Data
@AllArgsConstructor
@NoArgsConstructor
public class ListTicketTicketTypeResponseDto {
 private UUID id;
 private String name;
 private Double price;
}
```

Implementing the Ticket Mapper

Now we'll create a mapper to convert between our entities and DTOs.

The `TicketMapper` interface uses MapStruct to handle the conversion:

```
@Mapper(componentModel = "spring", unmappedTargetPolicy = ReportingPolicy.IGNORE)
public interface TicketMapper {

 ListTicketTicketTypeResponseDto toListTicketTicketTypeResponseDto(TicketType ticketType);

 ListTicketResponseDto toListTicketResponseDto(Ticket ticket);

}
```

The mapper includes two methods:

- `toListTicketTicketTypeResponseDto` converts a `TicketType` entity to its DTO representation
- `toListTicketResponseDto` converts a `Ticket` entity to its DTO representation

Summary

- Created `ListTicketResponseDto` and `ListTicketTicketTypeResponseDto` classes
- Created `TicketMapper` with mapper methods

List Ticket Endpoint

In this lesson we'll implement the list ticket endpoint.

Creating the Ticket Controller

We'll start by creating a new controller class to handle ticket-related operations.

Let's create a new `TicketController` class with the `@RestController` annotation and set up the base path for our API:

```
@RestController
@RequestMapping(path = "/api/v1/tickets")
@RequiredArgsConstructor
public class TicketController {

 private final TicketService ticketService;
 private final TicketMapper ticketMapper;

 @GetMapping
 public Page<ListTicketResponseDto> listTickets(
     @AuthenticationPrincipal Jwt jwt,
     Pageable pageable
 ) {
   return ticketService.listTicketsForUser(
       parseUserId(jwt),
       pageable
   ).map(ticketMapper::toListTicketResponseDto);
 }

}
```

Summary

- Implemented the list ticket endpoint

UI Testing

In this lesson, we'll test our list ticket endpoint through the user interface.

Setting Up the Environment

Before testing through the UI, we need to ensure our application is running correctly:

1. Start by building the backend with Maven:

```
mvn clean compile
```

2. Run the Spring Boot application.

Testing Through the Browser

Let's walk through the testing process in the browser:

Open your application's UI in your browser.

Log in using an attendee account that has previously purchased tickets.

Navigate to the dashboard, which should automatically redirect to the list tickets page.

Inspecting the Network Calls

Using the browser's developer tools, we can examine the API calls being made:

Open the Network tab in your browser's developer tools.

Refresh the page to see the network requests.

Look for calls to `/api/v1/tickets` which should include:

- Page information in the request
- Response containing ticket `id`, `status`, and `ticketType`

Summary

- Tested the list ticket endpoint using the user interface

Module Summary

Key Concepts Covered

List Ticket Service Layer

- Added the `findByPurchaserId` method to `TicketRepository`
- Implemented the `TicketService` with `listTicketsForUser` method

List Ticket DTO & Mapper

- Created `ListTicketResponseDto` and `ListTicketTicketTypeResponseDto` classes
- Created `TicketMapper` with mapper methods

List Ticket Endpoint

- Implemented the list ticket endpoint

UI Testing

- Tested the list ticket endpoint using the user interface

Get Ticket

Module Overview

In this module, we'll build the functionality that allows users to retrieve their ticket information and associated QR code.

Module Structure

1. Implement ticket retrieval in the service layer
2. Create DTOs and mappers
3. Implement the get ticket endpoint
4. Implement QR code retrieval in the service layer
5. Implement the QR code retrieval endpoint
6. User interface testing

Learning Objectives

1. Implement the service layer to get a ticket
2. Implement the DTOs and mappers to get a ticket
3. Implement the get ticket endpoint
4. Implement get QR Code functionality in the service layer
5. Implement the get ticket QR code image
6. Test the get ticket endpoint by using the user interface

Get Ticket Service Layer

In this lesson we'll implement the get ticket functionality in the service layer.

Adding a New Repository Method

Let's start by adding a new method to the `TicketRepository` interface.

The method will help us find a ticket by both its ID and the purchaser's ID.

This ensures tickets can only be retrieved by their rightful owners.

```
Optional<Ticket> findByIdAndPurchaserId(UUID id, UUID purchaserId);
```

Implementing the Service Layer

Now we'll create a method in the `TicketService` that uses our new repository method.

This method will act as a pass-through to the repository, maintaining the same return type and validation logic.

```
@Override
public Optional<Ticket> getTicketForUser(UUID userId, UUID ticketId) {
 return ticketRepository.findByIdAndPurchaserId(ticketId, userId);
}
```

Summary

- Added the `findByIdAndPurchaserId` method to `TicketRepository`
- Implemented the `getTicketForUser` method in `TicketService`

Get Ticket DTO & Mapper

In this lesson, we'll implement the DTOs and mappers that we need to implement the get ticket endpoint.

Creating the Get Ticket Response DTO

The `GetTicketResponseDto` will combine information from the ticket, ticket type, and event entities.

Instead of nesting this information in separate objects, we'll flatten it into a single DTO for simplicity:

```
@Data
@AllArgsConstructor
@NoArgsConstructor
public class GetTicketResponseDto {
 private UUID id;
 private TicketStatusEnum status;
 private Double price;
 private String description;
 private String eventName;
 private String eventVenue;
 private LocalDateTime eventStart;
 private LocalDateTime eventEnd;
}
```

Implementing the Ticket Mapper

The ticket mapper is responsible for converting between our entities and DTOs.

We'll add a new method to map a ticket entity to our response DTO:

```
@Mapper(componentModel = "spring", unmappedTargetPolicy = ReportingPolicy.IGNORE)
public interface TicketMapper {
 @Mapping(target = "price", source = "ticket.ticketType.price")
 @Mapping(target = "description", source = "ticket.ticketType.description")
 @Mapping(target = "eventName", source = "ticket.ticketType.event.name")
 @Mapping(target = "eventVenue", source = "ticket.ticketType.event.venue")
 @Mapping(target = "eventStart", source = "ticket.ticketType.event.start")
 @Mapping(target = "eventEnd", source = "ticket.ticketType.event.end")
 GetTicketResponseDto toGetTicketResponseDto(Ticket ticket);
}
```

The `@Mapping` annotations tell MapStruct how to navigate the relationships between our entities to get the right information.

We can verify our mapper is working by running a clean and compile, which will generate the actual implementation code.

Summary

- Created the `GetTicketResponseDto` class
- Added the `toGetTicketResponseDto` method to `TicketMapper`

Get Ticket Endpoint

In this lesson we'll implement the get ticket endpoint.

Implementing the Get Ticket Endpoint

The get ticket endpoint allows users to retrieve detailed information about a specific ticket they have purchased.

Let's add this new endpoint to our `TicketController` class:

```
@GetMapping(path = "/{ticketId}")
public ResponseEntity<GetTicketResponseDto> getTicket(
   @AuthenticationPrincipal Jwt jwt,
   @PathVariable UUID ticketId
) {
   return ticketService
       .getTicketForUser(parseUserId(jwt), ticketId)
       .map(ticketMapper::toGetTicketResponseDto)
       .map(ResponseEntity::ok)
       .orElse(ResponseEntity.notFound().build());
}
```

The endpoint follows these steps:

1. Gets the authenticated user's ID from the JWT token
2. Uses the `TicketService` to find the ticket for that user
3. Maps the ticket to a DTO if found
4. Returns HTTP 200 with the ticket data if found, or HTTP 404 if not found

Summary

- Implemented the get ticket endpoint

Ticket Get QR Code Service Layer

In this lesson, we'll implement the service layer logic to get the QR code image.

QR Code Repository Method

Let's begin by adding a new method to our QR code repository to help us find QR codes by both ticket ID and ticket purchaser ID.

```
Optional<QrCode> findByTicketIdAndTicketPurchaserId(UUID ticketId, UUID ticketPurchaseId);
```

This method allows us to look up QR codes using both the ticket ID and the ID of the purchaser, ensuring we only return QR codes to their rightful owners.

Service Layer Implementation

Next, we'll implement the method in our QR code service to retrieve the QR code image.

```
@Override
public byte[] getQrCodeImageForUserAndTicket(UUID userId, UUID ticketId) {
   QrCode qrCode = qrCodeRepository.findByTicketIdAndTicketPurchaserId(ticketId, userId)
       .orElseThrow(QrCodeNotFoundException::new);

   try {
     return Base64.getDecoder().decode(qrCode.getValue());
    catch(IllegalArgumentException ex) {
     log.error("Invalid base64 QR Code for ticket ID: {}", ticketId, ex);
     throw new QrCodeNotFoundException();
   }
}
```

The method performs two main tasks:

1. It retrieves the QR code from the database using both the user ID and ticket ID.
2. It decodes the Base64-encoded QR code back into a byte array.

If anything goes wrong during the process - either the QR code isn't found or can't be decoded - we throw appropriate exceptions and log the error.

Summary

- Added `findByTicketIdAndTicketPurchaserId` to `QrCodeRepository`
- Implemented `getQrCodeImageForUserAndTicket` on `QrCodeService`

Get QR Code Endpoint

In this lesson, we implement the endpoint to get the QR code image.

Creating the QR Code Endpoint

We'll add a new endpoint to the `TicketController` that returns a QR code image associated with a specific ticket.

The endpoint will be an extension of the existing get ticket functionality:

```
@GetMapping(path = "/{ticketId}/qr-codes")
public ResponseEntity<byte[]> getTicketQrCode(
   @AuthenticationPrincipal Jwt jwt,
   @PathVariable UUID ticketId
) {
   byte[] qrCodeImage = qrCodeService.getQrCodeImageForUserAndTicket(
       parseUserId(jwt),
       ticketId
   );

   HttpHeaders headers = new HttpHeaders();
   headers.setContentType(MediaType.IMAGE_PNG);
   headers.setContentLength(qrCodeImage.length);

   return ResponseEntity.ok()
       .headers(headers)
       .body(qrCodeImage);
}
```

HTTP Headers for Image Response

To properly serve the QR code image, we need to set specific HTTP headers:

1. We set the `Content-Type` header to `image/png` since we're returning a PNG image
2. We set the `Content-Length` header to match the size of our image data in bytes

These headers help the client browser understand how to handle and display the received data correctly.

Summary

- Implemented the get QR Code endpoint

UI Testing

In this lesson, we'll test the get ticket and QR code functionality through the user interface, building on our previous implementation of these endpoints.

Testing the Get Ticket Functionality

Before we begin testing, we need to ensure our environment is properly set up:

- Run `clean` and `compile` commands to build the application
- Start the backend application
- Log in as an attendee user through the UI

Once logged in, navigate to the dashboard where you'll see your purchased tickets.

Each ticket entry displays basic information and clicking on a ticket reveals the full details including:

- Event name
- Venue
- Start and end times
- Ticket ID
- QR code representation

Summary

- Tested the get ticket functionality using the user interface

Module Summary

Key Concepts Covered

Get Ticket Service Layer

- Added the `findByIdAndPurchaserId` method to `TicketRepository`
- Implemented the `getTicketForUser` method in `TicketService`

Get Ticket DTO & Mapper

- Created the `GetTicketResponseDto` class
- Added the `toGetTicketResponseDto` method to `TicketMapper`

Get Ticket Endpoint

- Implemented the get ticket endpoint

Ticket Get QR Code Service Layer

- Added `findByTicketIdAndTicketPurchaserId` to `QrCodeRepository`
- Implemented `getQrCodeImageForUserAndTicket` on `QrCodeService`

Get QR Code Endpoint

- Implemented the get QR Code endpoint

UI Testing

- Tested the get ticket functionality using the user interface

Validate Ticket

Module Overview

In this module we'll implement the ticket validation functionality in the service layer.

Module Structure

1. Implement ticket validation service layer functionality
2. Create DTOs and mappers
3. Implement the validate ticket endpoint
4. Test through the UI

Learning Objectives

1. Implement the service layer to validate a ticket
2. Implement the DTOs and mappers to validate a ticket
3. Implement the validate ticket endpoint
4. Test the validate ticket endpoint by using the user interface

Validate Ticket Service Layer

In this lesson, we'll implement the validate ticket service layer.

Repository Setup

First, we need to create a repository for ticket validations. This forms the foundation for our database interactions:

```
@Repository
public interface TicketValidationRepository extends JpaRepository<TicketValidation, UUID> {

}
```

Service Interface

The service interface defines two methods for validating tickets - one using a QR code and another for manual validation:

```
public interface TicketValidationService {
 TicketValidation validateTicketByQrCode(UUID qrCodeId);
 TicketValidation validateTicketManually(UUID ticketId);
}
```

Service Implementation

The implementation handles the business logic for ticket validation:

```
@Service
@RequiredArgsConstructor
@Transactional
public class TicketValidationServiceImpl implements TicketValidationService {

 private final QrCodeRepository qrCodeRepository;
 private final TicketValidationRepository ticketValidationRepository;
 private final TicketRepository ticketRepository;

 @Override
 public TicketValidation validateTicketByQrCode(UUID qrCodeId) {
   QrCode qrCode = qrCodeRepository.findByIdAndStatus(qrCodeId, QrCodeStatusEnum.ACTIVE)
       .orElseThrow(() -> new QrCodeNotFoundException(
           String.format(
               "QR Code with ID %s was not found", qrCodeId
           )
       ));

   Ticket ticket = qrCode.getTicket();

   return validateTicket(ticket);
 }

 private TicketValidation validateTicket(Ticket ticket) {
```

```
    TicketValidation ticketValidation = new TicketValidation();
    ticketValidation.setTicket(ticket);
    ticketValidation.setValidationMethod(TicketValidationMethod.QR_SCAN);

    TicketValidationStatusEnum ticketValidationStatus = ticket.getValidations().stream()
        .filter(v -> TicketValidationStatusEnum.VALID.equals(v.getStatus()))
        .findFirst()
        .map(v -> TicketValidationStatusEnum.INVALID)
        .orElse(TicketValidationStatusEnum.VALID);

    ticketValidation.setStatus(ticketValidationStatus);

    return ticketValidationRepository.save(ticketValidation);
  }

  @Override
  public TicketValidation validateTicketManually(UUID ticketId) {
    Ticket ticket = ticketRepository.findById(ticketId)
        .orElseThrow(TicketNotFoundException::new);
    return validateTicket(ticket);
  }
}
```

The implementation includes these key features:

The `validateTicketByQrCode` method looks up an active QR code and validates the associated ticket.

The `validateTicketManually` method looks up a ticket directly by ID and validates it.

The private `validateTicket` method contains the shared validation logic.

Tickets can only be validated once - subsequent validations will return invalid status.

Summary

- Implemented the ticket validation service layer functionality

Validate Ticket DTO & Mapper

In this lesson, we're going to create the DTOs and mappers that we need in order to implement our validate ticket endpoint.

Creating the Request DTO

The `TicketValidationRequestDto` is a simple data class that carries validation request information.

```
@Data
@AllArgsConstructor
@NoArgsConstructor
public class TicketValidationRequestDto {
 private UUID id;
 private TicketValidationMethod method;
}
```

The class has two fields:

- An `id` field of type `UUID` which can represent either a QR code ID or a ticket ID
- A `method` field of type `TicketValidationMethod` enum to specify the validation method

Creating the Response DTO

The `TicketValidationResponseDto` represents the result of a ticket validation attempt.

```
@Data
@AllArgsConstructor
@NoArgsConstructor
public class TicketValidationResponseDto {
 private UUID ticketId;
 private TicketValidationStatusEnum status;
}
```

This class contains:

- A `ticketId` field to identify the validated ticket
- A `status` field using `TicketValidationStatusEnum` to indicate the validation result

Implementing the Mapper

The `TicketValidationMapper` interface handles the conversion between domain objects and DTOs.

```
@Mapper(componentModel = "spring", unmappedTargetPolicy = ReportingPolicy.IGNORE)
public interface TicketValidationMapper {

 @Mapping(target = "ticketId", source = "ticket.id")
 TicketValidationResponseDto toTicketValidationResponseDto(TicketValidation ticketValidation);

}
```

The mapper has a single method that:

- Takes a `TicketValidation` domain object as input
- Maps it to a `TicketValidationResponseDto`
- Uses `@Mapping` to specify that the `ticketId` field should come from `ticket.id`

Summary

- Added `TicketValidationRequestDto` and `TicketValidationResponseDto` DTO classes
- Added the mapping method `toTicketValidationResponseDto` to `TicketValidationMapper`

Validate Ticket Endpoint

In this lesson we're going to implement the validate ticket endpoint.

Creating the Controller

Let's create a new controller to handle ticket validation requests. The controller will be responsible for validating tickets through two methods - QR code scanning and manual validation.

First, we'll create a new class called `TicketValidationController` with the required annotations:

```
@RestController
@RequestMapping(path = "/api/v1/ticket-validations")
@RequiredArgsConstructor
public class TicketValidationController {

 private final TicketValidationService ticketValidationService;
 private final TicketValidationMapper ticketValidationMapper;

 @PostMapping
 public ResponseEntity<TicketValidationResponseDto> validateTicket(
     @RequestBody TicketValidationRequestDto ticketValidationRequestDto
 ){
   TicketValidationMethod method = ticketValidationRequestDto.getMethod();
   TicketValidation ticketValidation;
   if(TicketValidationMethod.MANUAL.equals(method)) {
     ticketValidation = ticketValidationService.validateTicketManually(
         ticketValidationRequestDto.getId());
   } else {
     ticketValidation = ticketValidationService.validateTicketByQrCode(
         ticketValidationRequestDto.getId()
     );
   }
   return ResponseEntity.ok(
       ticketValidationMapper.toTicketValidationResponseDto(ticketValidation)
   );
 }
}
```

Securing the Endpoint

To ensure only staff members can validate tickets, we need to secure the endpoint with the appropriate role:

```
http
   .authorizeHttpRequests(authorize ->
       authorize
           .requestMatchers("/api/v1/ticket-validations").hasRole("STAFF")
           // Catch all rule
           .anyRequest().authenticated())
```

Summary

- Implemented the validate ticket endpoint

UI Testing

In this lesson, we'll test the ticket validation functionality through the user interface.

Testing QR Code Validation

QR code validation allows staff members to quickly scan and validate attendee tickets using their device's camera.

Let's start by testing the QR code scanning functionality:

- Log in as an attendee in one browser window to display the ticket
- Log in as staff in another browser window to access the validation page
- Use the network panel in the browser's developer tools to monitor the API calls

When scanning the same QR code twice, we should observe:

- First scan: Returns "valid" status with a green checkmark
- Second scan: Returns "invalid" status with a red cross (as tickets can only be validated once)

Testing Manual Validation

Manual validation provides a fallback method when QR code scanning isn't possible or practical.

To test manual validation:

- Copy the ticket ID from the attendee's ticket
- Navigate to the manual input section on the validation page
- Enter the ticket ID and submit
- Verify the response in both the UI and network panel

Summary

- Tested the ticket validation functionality through the UI

Module Summary

Key Concepts Covered

Validate Ticket Service Layer

- Implemented the ticket validation service layer functionality

Validate Ticket DTO & Mapper

- Added `TicketValidationRequestDto` and `TicketValidationResponseDto` DTO classes
- Added the mapping method `toTicketValidationResponseDto` to `TicketValidationMapper`

Validate Ticket Endpoint

- Implemented the validate ticket endpoint

UI Testing

- Tested the ticket validation functionality through the UI