SeatGen - The Seating Plan Generation Tool For Stadiums

DIPLOMA THESIS

submitted for the

Reife- und Diplomprüfung

at the

Department of IT-Medientechnik

Submitted by:
Michael Ruep
Michael Stenz

Supervisor:
Prof. Mag. Martin Huemer

Project Partner:
Solvistas GmbH

Leonding, April 4, 2025

I hereby declare that I have composed the presented paper independently and on my own, without any other resources than the ones indicated. All thoughts taken directly or indirectly from external sources are properly denoted as such. This paper has neither been previously submitted to another authority nor has it been published yet. The presented paper is identical to the electronically transmitted document.

Leonding, April 4, 2025   Michael Ruep & Michael Stenz

Abstract

SeatGen is a tool developed for Solvistas GmbH
to simplify stadium seating plan creation and man-
agement. It replaces the inefficient manual process
with an intuitive graphical interface, allowing event
organizers to design and edit seating layouts with-
out technical expertise.

Built with React, Spring Boot & Kotlin, and
Leaflet.js, SeatGen enables direct seat manipulation, real-time updates, and bulk
modifications. This thesis explores the challenges, technologies, and implementation
behind SeatGen.

Contents

1 Introduction

1.1 Initial Situation

The company Solvistas GmbH is a software development company, and one of their main products is the Ticketing project. Ticketing is a software solution that enables customers to purchase tickets for seats or sections in stadiums and other venues hosting events. The software is used by various sports clubs and event organizers to manage ticket sales for their events.

1.2 Problem Statement

The as just mentioned stadiums and venues have a lot of seats and different areas, and therefore the Ticketing software needs to know the layout of the seats. These layouts can have lots of complex shapes like curves and other irregular shapes. The current process of creating these so-called seat plans is done manually by editing text files. There are many problems, and it's a very tedious process when editing seat plans within a text editor. To name a few: When changing the layout of a stadium, all the text files have to be reworked by a schooled developer. This costs the customer a lot of money, and the developer a lot of time. Also, it's very hard to imagine how the rendered plan looks, when staring at text files.

Uploading the plan image is another tedious task when creating new plans. To convert the given SVG file into a functional map compatible with their system, the developer must manually upscale and slice the SVG into tiles, repeating this process for each zoom level—typically 5 to 7 times. Additionally, since each tile is divided into four smaller tiles at every zoom level, the number of tiles increases exponentially. As a result, a massive number of files must be uploaded to an AWS S3 bucket, making the process even more time-consuming.

1.3 Goal

The goal of the diploma thesis was to develop a custom solution for the company Solvistas and solve all these aforementioned problems with a tool that's intuitive to use and easy to learn, saving time and costs. The goal was to create a visual editor that creates and manages seat plans for events in a stadium. This editor allows customers to create and edit new seating plans themselves, making the process so accessible and easy to use that no more schooled developers are required to make changes in a seating plan.

2 Context / Environment Analysis

2.1 Overview

Stadiums are typically operated by sports clubs, concert organizers, or third-party management companies. These actors want to sell their tickets efficiently and accurately for a large amout of offers. The environment is therefore characterized by:

- **Varied Layouts:** Modern stadiums feature a curved layout, irregular seat patterns, and different pricing tiers. Managing the seat data in plain text is error-prone and time-intensive.

- **Frequent Configuration Changes:** Stadium layouts frequently change based on events and seasons, requiring continuous updates.

- **Limited Technical Staff:** The event organizers often rely on external developers to alter seat plan definitions, creating additional costs.

From a user-experience standpoint, these challenges make it clear that an intuitive, graphical editing interface is needed to replace the text-based seat data manipulation. This concept aligns with the principles of *direct manipulation interfaces*, as described by Hutchins, Hollan, and Norman, which emphasize reducing the cognitive distance between user intent and system actions by allowing direct interaction with visual representations [**?**]. In the context of SeatGen, users can place, move, and edit seats through an intuitive graphical interface rather than modifying abstract raw text data. Similar to how direct manipulation in statistical tools allows users to interact with data graphs instead of numbers, SeatGen provides a **spatially direct approach** to stadium seat planning.

2.2 Stakeholder Analysis

Several parties interact with the SeatGen tool:

- **Developers:** Historically, Solvistas' developers modified the seat layout text files. The new approach aims to minimize their involvement in the long term, except for initial and advanced configurations.

- **Event Organizers and Stadium Managers:** These stakeholders need the intuitive tools to make update to the seat maps without dealing with complex raw data formats.

- **Ticketing Platform Users:** The final seat layouts are used in Solvistas' ticketing service. Although these end-users do not edit the data themselves, the accuracy and clarity of seat layouts crucially impact their experience at the stadium.

By identifying these stakeholders and their needs, the main focus is placed on a graphical, user-friendly solution for seat map creation and maintenance.

2.3 Technical Environment

On the technical side, the SeatGen application integrates with:

- **AWS S3 Buckets:** Image tiles and map data are uploaded to and fetched from a secure Amazon Web Services (AWS) cloud environment.

- **React-based Frontend:** Provides an interactive user interface for managing and modifying seat layouts with real-time updates.

- **Spring Boot / Kotlin Backend:** Manages image processing and seat data handling.

- **PostgreSQL Database:** Stores seat configurations, categories, groups, sectors, and map data.

In practice, large or complex venues may have thousands of individual seat entities, potentially impacting performance if the data management is not optimized. Additionally, the zoom levels demand loads of memory- and compute-efficient image slicing and compression, to avoid excessive storage usage on Amazon Web Services. The design choices reflect these constraints and requirements in multiple areas.

2.4 Requirements and Challenges

A core requirement of SeatGen is "direct manipulation" [**?**], which states that users interact more effectively when objects can be selected, dragged, and dropped in a way that mimics the real-world arrangement of physical seats or standing areas. Therefore the usability and user experience is a key Challenge. The following elements are particularly relevant:

- **Immediate Feedback:** When a user modifies a seat position, the update is reflected instantly on the map. Dragging seats should feel instant and follow the user, mimicking real-life interactions

- **Simplicity and Learnability:** Familiar mouse-based interactions should allow users to perform specific and complex tasks—such as grouping seats, creating standing areas, or categorizing—without requiring specialized training or an understanding of coordinate systems.

- **Cognitive Offloading:** By representing seats visually, the mental effort to interpret raw text-based seat coordinates or stadium layouts is reduced significantly. Which improves efficiency and accuracy by a huge margin.

Leaflet, used within React, handles dynamic rendering and live interaction for the stadium seat map. Additionally, quick actions in the form of hotkeys further improve efficiency.

2.5 Summary

In summary, the environment for SeatGen includes a mix of business and technical constraints, demanding a straightforward, user-friendly and yet powerful visual editor. In the following chapters, the technical and logical aspects of how the set requirements are met will be further described.

3 Technologies

A complete visualization of the tech stack can be found in **??**. Further details and explanations can be found in the following sections.
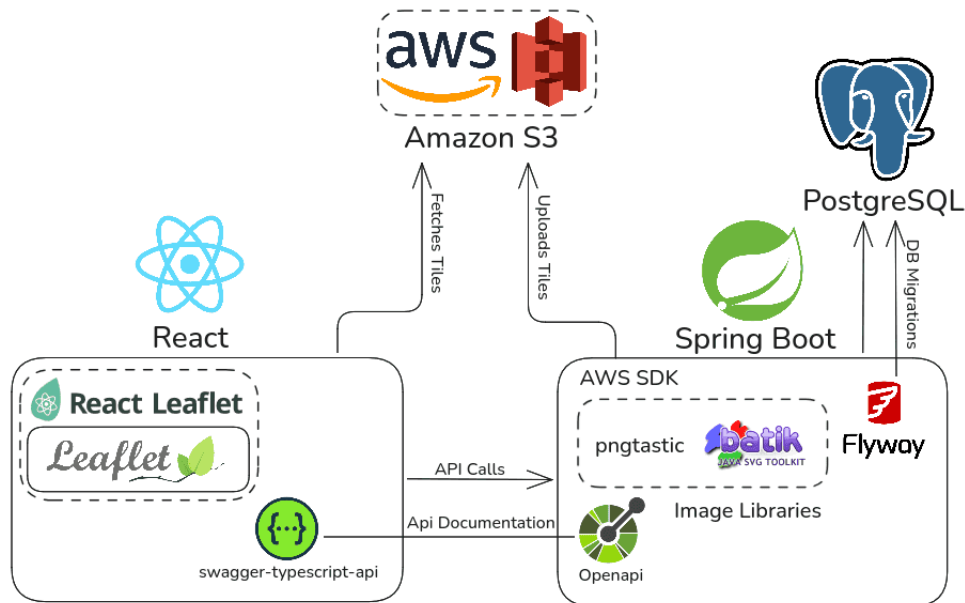

Figure 1: Tech Stack

3.1 React

The frontend of SeatGen is built using the React framework. This choice was primarily motivated by the existing expertise within Solvistas, ensuring that the company's developers could easily maintain and adjust the project to their needs. Additionally, React provides an ideal balance between flexibility, performance, and a vibrant ecosystem, which are factors that proved crucial when building an interactive seating plan editor for stadiums. Also, our team was already experienced with component-based single page application frontend frameworks.

3.1.1 Framework Background

React is an open-source JavaScript library developed and maintained by Meta (formerly known as Facebook) and a community of individual developers and companies.

Originally introduced in 2013 to power Facebook's dynamic news feed, React has since become renowned for creating data-driven web interfaces [**?**, **?**]. Rather than manually manipulating the Document Object Model (DOM), developers can simply declare how the interface should appear based on the underlying data. This declarative approach allows React to handle updates internally, ensuring smoother user interactions, which are especially beneficial for large or frequently changing data sets [**?**, **?**].

3.1.2 Virtual DOM

React's Virtual DOM architecture is particularly advantageous for applications requiring frequent updates and complex UI interactions. In SeatGen, each seat on the map can be added, moved, or deleted in real time, causing rapid changes that must be reflected in the user interface without compromising speed. By selectively re-rendering only components that have actually changed, the Virtual DOM mechanism helps maintain excellent performance even under heavy load [**?**]. This aligns with the findings in [**?**], where React demonstrates superior rendering speed and user satisfaction in Single Page Applications (SPAs) requiring dynamic content updates.

3.1.3 Component-based Architecture

React's component-based architecture keeps each feature modular to make the project easier to maintain as it grows. Instead of bundling all functionality into a single monolithic view, UI features are developed as self-contained components. In our case for example:

- Seat Map Component
- Toolbar Component
- Detail Component

Each of those elements has its own component, allowing developers to modify or expand individual features without affecting unrelated parts of the application. This approach simplifies debugging, since issues can often be traced to a specific component rather than across the entire codebase. It is also suitable for collaborative development by letting team members work on separate components in parallel, which significantly accelerated our development process. Overall, React's modular design reduces complexity which assists a more organized and maintainable codebase. [**?**, **?**, **?**]

3.1.4 Integration of Libraries

One of SeatGen's central requirements is to enable direct manipulation for seat layouts. React's flexible architecture allows for the easy incorporation of third-party libraries. For example, Leaflet was integrated to render the stadium map using its efficient, canvas-based engine. React uses its Context API to manage global state and user interactions, while Leaflet is responsible for the map rendering. This separation ensures that intensive mapping operations do not interfere with overall UI responsiveness. Reacts Context was used to manage global state effectively. This approach meets the specific requirements of seat manipulation, multi-layered zoom levels, and user interactions.

3.1.5 Developer Familiarity and Team Expertise

Since React was already in use at Solvistas, its adoption ensured that the company's developers could seamlessly work with and extend the project. Additionally, our team had prior experience with component-based frontend frameworks, making it easy to understand React's structure, including concepts such as routing, state management, data binding, and components. This familiarity enabled a quick understanding and utilization of React, leading to the rapid development of the first running prototype.

Comparison to Other Frameworks

Compared to alternative frameworks such as Angular or Vue studies have shown that React and Vue generally demonstrate superior rendering performance and faster load times in dynamic applications [?]. Additionally, React is preferred by developers for SPAs with frequent UI updates, with a reported 34% higher satisfaction rate compared to other frameworks [?].

3.1.6 Summary

React's widespread adoption, strong ecosystem, and proven efficiency in building dynamic web applications make it a reliable choice for modern frontend development. Its Virtual DOM ensures optimized rendering performance, while its component-based architecture keeps the application modular and maintainable. Additionally, React Context provides a lightweight yet effective solution for managing global state. This allows seamless integration with external libraries such as Leaflet for real-time seat rendering [?, ?].

With React already in use within the company, adopting it ensured maintainability and smooth collaboration. Its performance advantages in Single Page Applications (SPAs), along with high developer satisfaction rates [**?**], further validated its suitability for our interactive seating plan editor.

3.2 Spring Boot and Kotlin

For the backend logic, Spring Boot was chosen as it is a core technology in Solvistas' tech stack. This decision ensures that the project remains maintainable by Solvistas developers in the long run. The backend has several key responsibilities, including:

- Handling the storage of the seatplan metadata
- Converting SVGs into image tiles
- Uploading the converted tiles to an S3 bucket
- Serving all of this data to the frontend via REST

For image processing tasks such as resizing and slicing SVGs and PNGs, Python was initially considered due to its well-documented and easy-to-use image manipulation libraries like CairoSVG and OpenCV. However, the decision was ultimately made to keep the processing within the Java/Kotlin ecosystem, using libraries like Batik[**?**] and ImageIO[**?**]. While Java/Kotlin image processing is not as straightforward as Python due to less extensive documentation and fewer community resources, this choice allowed for a consistent backend technology stack. One challenge with Java-based image processing is memory management—heap size and garbage collection must always be considered, especially when processing large images. For file uploads, Amazon S3 provides excellent support for Java and Kotlin through the AWS SDK[**?**], accompanied by extensive documentation and examples. This integration made it easy to incorporate S3 into the backend for efficiently storing and retrieving image tiles.

3.2.1 Kotlin

As for the language, Kotlin was used in Spring Boot, even though it is not commonly employed in many of Solvistas' projects. However, Kotlin was deemed the better option because it is a modern language that is fully interoperable with Java and offers many features that facilitate writing clean and concise code, thereby reducing errors and improving readability and maintainability. It eliminates much of the boilerplate code

required in Java and provides a rich standard library with numerous built-in utility functions, significantly reducing development time. While Kotlin does not offer essential functionalities that Java cannot provide, it is more modern and has a more concise syntax.

Additionally, Kotlin introduces powerful features such as null safety, which helps create more robust applications with fewer runtime errors. Furthermore, Kotlin provides strong support for functional programming, including higher-order functions, lambda expressions, and extension functions, making it easier to write expressive and reusable code. Another key advantage is Kotlin's coroutines, which enable highly efficient asynchronous programming without the complexity of Java's traditional thread management. This makes Kotlin particularly well-suited for handling concurrent tasks, such as processing multiple image transformations simultaneously, significantly reducing processing time.

Kotlin's seamless integration with Spring Boot also allows for idiomatic DSLs (Domain-Specific Languages), which can simplify configuration and reduce verbosity in code. The language's structured concurrency and intuitive syntax contribute to cleaner, more maintainable backend services, ensuring long-term scalability. Finally, Kotlin's growing adoption within the Spring ecosystem, along with first-class support from JetBrains and the Spring team, makes it a viable choice for modern backend development. Its developer-friendly nature, combined with reduced verbosity and enhanced safety features, positions it as a forward-thinking investment despite its lower adoption within Solvistas' existing projects.

In the end, Spring Boot with Kotlin was chosen due to the team's expertise with the language and the fact that all other components of the Ticketing software were already written in Spring Boot.

3.2.2 API Documentation with Swagger

SeatGen also utilizes SwaggerUi and SwaggerUi codegen to generate REST API documentation and client code for the frontend. This allows for easy integration of the backend with the frontend and ensures that the frontend developers always have the most up-to-date API documentation. This is done via the OpenApi gradlew plugin, which generates the SwaggerUi documentation and client code for all the API endpoints and required models. The generated docs can be fetched by the frontend developers

with a script within the `package.json` file, under the name `fetch-openapi-docs`. This script fetches the api-docs.yaml file from the backend and saves it in the frontend project. When starting or building the frontend, the `swagger-typescript-api` plugin generates the client code from this file. The frontend developer now can use the generated client code to interact with the backend API through the generated functions and models without having to manually maintain the API client code.

3.3 Database

PostgreSQL was chosen as the database for several key reasons. A relational database is required since the data follows a structured design that is best represented through classical relational models. The structure of the database is visualized in the class diagram **??**. Additionally, utilizing a relational database simplifies the process of exporting generated data into the Ticketing database, which also adheres to a relational structure.

The system is designed to be compatible with multiple relational databases, not just PostgreSQL, as the Java Persistence API (JPA) is utilized as the Object-Relational Mapping (ORM) framework. To maintain database flexibility, PostgreSQL-specific commands were deliberately avoided. While leveraging PL/pgSQL for business logic could have provided benefits such as enhanced security, improved performance, and greater data consistency, database interchangeability was prioritized.

For database connectivity in the Spring Boot application, the Spring Data JPA library was utilized. This library streamlines the process of connecting to a database and executing queries while implementing the repository pattern. Through this pattern, custom queries are defined in an interface, which Spring Boot automatically implements at runtime. This approach simplifies query management, making it easier to maintain and use repository methods directly within the codebase.

To manage database migrations efficiently, the Flyway [**?**] library is used. Flyway enables the to define database changes through SQL scripts that execute automatically when the application starts. This ensures that the database schema remains consistent with the latest changes, simplifying deployment and mitigating potential conflicts across different environments. Managing migrations this way also helps prevent issues arising from different database versions among team members. Additionally, since Flyway migrations consist of entire SQL scripts, both Data Definition Language (DDL) and
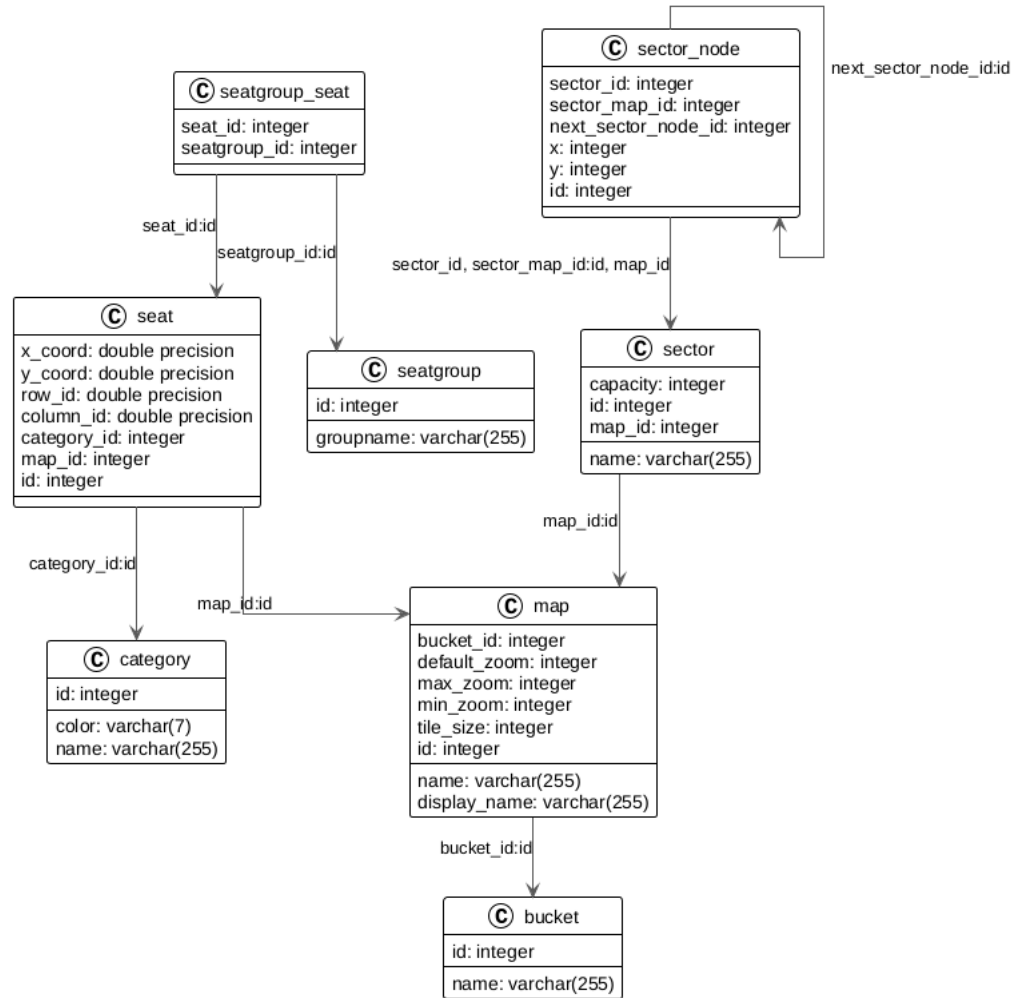
Figure 2: Database Model

Data Manipulation Language (DML) commands can be executed. This capability is particularly beneficial for tasks such as migrating data between tables, altering column data types, and implementing other business logic-related transformations.

When selecting a migration tool, both Liquibase [**?**] and Flyway were evaluated. While both are open-source and provide seamless integration with Spring Boot and other Java frameworks, the decision was ultimately made to opt for Flyway due to its simplicity and specific use case. Since the team is small with infrequent parallel database changes, Flyway's linear migration approach suits the workflow without introducing complications. Although this approach might present challenges in larger teams with concurrent database modifications, it remains a practical choice for current needs.

Flyway also offers a cleaner versioning system by requiring migration filenames to follow a structured naming convention: `VX.X.X__migration_name.sql` (where X.X.X is the version of the migration). In contrast, Liquibase utilizes changelog files, which provide additional features but introduce unnecessary complexity for the use case. These changelog files can be written in SQL, XML, YAML, or JSON, but they require extensive Liquibase-specific formatting. The following example illustrates a Liquibase-formatted SQL changelog file **??**. Flyway's approach, which relies on plain SQL migration files, enhances readability and maintainability.

Listing 1: Liquibase example changelog

```sql
--liquibase formatted sql

--changeset nvoxland:1
create table test1 (
    id int primary key,
    name varchar(255)
);
--rollback drop table test1;

--changeset nvoxland:2
insert into test1 (id, name) values (1, 'name 1');
insert into test1 (id,  name) values (2, 'name 2');

--changeset nvoxland:3 dbms:oracle
create sequence seq_test;
```

To ensure database consistency, Flyway generates a flyway_schema_history table that tracks all executed migrations. This table stores metadata for each migration, including the version, description, execution timestamp, and a checksum. The checksum prevents modifications to previously applied migrations, ensuring consistency but potentially causing unexpected errors during local development. In such cases, manual intervention in the flyway_schema_history table may be required, but except for these rare cases the flyway_schema_history table should not be manipulated manually.

By maintaining this history, Flyway can determine which migrations have been applied and which are still pending. Each migration also has a state, which can be pending, applied, failed, undone, and more—detailed in the Flyway documentation. These states allow system administrators to quickly identify and resolve migration and deployment issues.

When considering how to store image data, PostgreSQL's built-in options, including BLOBs (Binary Large Objects) and TOAST [**?**] (The Oversized-Attribute Storage Technique), were evaluated. While these mechanisms allow PostgreSQL to handle large binary files, the decision was ultimately made against using them due to performance concerns, maintenance overhead, scalability limitations, and company reasons. Even though, TOAST is very performant and automatically compresses and stores large column values outside the main table structure, making it a more attractive option than traditional BLOBs, accessing and manipulating the stored images via SQL queries can become a bottleneck. ORMs like Hibernate tend to retrieve large column values by default unless explicitly configured otherwise, potentially leading to performance degradation when dealing with frequent queries. This means extra effort would be required to optimize database queries to avoid unnecessary data retrieval, increasing development complexity.

3.4 AWS - S3

For image processing tasks such as resizing and slicing SVGs and PNGs, Python was initially considered due to its well-documented and easy-to-use image manipulation libraries like CairoSVG and OpenCV. However, the decision was ultimately made to keep the processing within the Java/Kotlin ecosystem, utilizing libraries such as Batik and ImageIO. While image processing in Java/Kotlin is not as straightforward as in Python due to less extensive documentation and fewer community resources, maintaining consistency within the backend technology stack was prioritized. One challenge associated with Java-based image processing is memory management—heap size and garbage collection must always be considered, especially when processing large images. For file uploads, Amazon S3 provides excellent support for Java and Kotlin through the AWS SDK, with extensive documentation and examples. This facilitated seamless integration of S3 into the backend for efficient storage and retrieval of image tiles.

In the Ticketing project, all image tiles are stored in an AWS S3 bucket. S3 was required due to its robust performance, reliability, and seamless integration with the AWS ecosystem, which is already in use at Solvistas. By utilizing the Amazon S3 SDK, the file upload process is automated, reducing manual effort and minimizing the risk of errors.

Using S3 also improves frontend performance by ensuring that image retrieval does not depend on the backend server's speed. Instead of acting as a middleware for serving images, the backend delegates this task directly to S3, reducing its workload and enhancing response times.

AWS S3 was the only option considered, as it is the cloud platform used by Solvistas, and the infrastructure costs are funded by the company.

3.5 Leaflet

Leaflet is an open-source JavaScript library designed for interactive maps. Developed by Vladimir Agafonkin and maintained by a large community, it is widely used for its lightweight nature and ease of integration with modern web applications [**?**]. Unlike heavyweight mapping solutions such as Google Maps or OpenLayers, Leaflet is specifically optimized for rendering custom vector layers and handling dynamic user interactions efficiently. These characteristics make it an ideal choice for SeatGen's stadium seat visualization, where real-time updates and performance optimization are critical.

3.5.1 Reasons for Leaflet

For SeatGen, choosing the right mapping library was crucial to ensuring smooth and interactive seat visualization. Leaflet was selected due to its lightweight architecture, extensibility, and strong performance when rendering custom vector layers. Unlike Google Maps or OpenLayers, which offer extensive GIS (geographic information system focused) functionalities but often introduce unnecessary overhead, Leaflet is designed for fast, customizable, and lightweight mapping solutions [**?**].

A key advantage of Leaflet is its low dependency footprint. Unlike other mapping solutions that rely on external APIs or heavy SDKs, Leaflet provides a standalone JavaScript library that integrates seamlessly with React. This lightweight approach

ensures excellent map performance, even when handling large stadiums with thousands of seats. In contrast, frameworks like Google Maps API enforce rate limits and external API calls, which can introduce latency and unnecessary costs.

Leaflet's customizability also played a significant role in our decision. SeatGen requires custom zoom levels, and real-time updates, all of which are efficiently handled using Leaflet's open architecture. Unlike proprietary mapping tools, Leaflet allows full control over rendering logic, making it easier to optimize performance and adjust the visualization to match stadium layouts precisely [**?**]. Furthermore, existing Leaflet functions were adapted for tasks such as seat selection and the grid tool. This significantly accelerated the development process

By selecting Leaflet, SeatGen ensures efficient handleing of multi-layered rendering, interactive zoom, custom maps, and seamless seat selection, all while maintaining a lightweight and scalable frontend architecture.

3.5.2 Key Features Used in SeatGen

Leaflet provides several core functionalities that are essential for our interactive seating plan editor. The following features were particularly valuable in implementing a performant and user-friendly seat visualization system:

- **Dynamic Seat Rendering:** Leaflet enables the efficient rendering of thousands of seat markers without significantly impacting performance. Since stadiums can contain a large number of seats, rendering was optimized using Leaflet layers to manage visibility at different zoom levels.

- **Custom Zoom Levels and Scaling:** Leaflet allows the definition of custom zoom levels, ensuring precise seat selection when zoomed in and a full view of the stadium's structure when zoomed out.

- **Interactive Seat Selection:** By leveraging Leaflet's built-in event handling system, users can click and modify seats in real time. This is crucial for intuitively adjusting seating arrangements.

- **Grid-Based Seat Placement:** Leaflet's selection, polygon, and coordinate functions were used to implement functions like the grid tool and selection tool, allowing for structured seat placement. This feature speeds up the process

of generating rows and sections by automatically aligning seats according to predefined parameters.

- **Real-Time State Management with React Context:** Since Leaflet does not natively integrate with React's state management, React Context was used to synchronize seat selections, updates, and modifications across the application. This ensures that any seat change is reflected immediately in both the UI and the underlying data model.

These features collectively make Leaflet a powerful tool for handling the seat visualization requirements. By leveraging Leaflet's efficient rendering engine and customization capabilities, a seamless user experience was created that allows for real-time adjustments and intuitive stadium navigation.

3.5.3 Integration with React

Integrating Leaflet with React was straightforward thanks to React Leaflet, a library that provides React components for Leaflet [**?**]. This greatly simplified the integration process, as Leaflet elements could be managed within React components without directly manipulating the DOM.

One challenge faced was handling state management and data binding between Leaflet and React. Since Leaflet operates independently of React's Virtual DOM [**?**], synchronizing real-time seat selections, modifications, grid placements, and similar interactions required a structured approach.

Beyond standard Leaflet functionality, existing features were extended and customized to meet specific needs. Tools like the seat selection tool and grid tool leveraged Leaflet's built-in functions but were adapted to suit SeatGen's requirements. By understanding and modifying Leaflet's core functions, a tailored solution was developed, aligning with the requirements for real-time seat arrangement and stadium visualization.

3.5.4 Summary

Leaflet's lightweight architecture, flexibility, and strong customization capabilities made it the ideal choice for an interactive seating plan editor. Unlike heavier GIS-focused alternatives, Leaflet provided a high-performance mapping solution tailored for real-time seat rendering and selection.

Its custom zoom levels allowed for the creation of an intuitive and responsive seating visualization tool. Additionally, React Leaflet streamlined the integration process, enabling faster development within React [**?**].

By leveraging Leaflet's existing functionality and extending its core features to suit stadium seat planning, a clean and efficient tool suite was ensured, enabling smooth real-time interaction. This solidifies Leaflet as an essential part of the frontend architecture.

4 Implementation

4.1 Frontend Architecture

To maintain a modular and scalable UI, SeatGen's frontend follows a structured component-based architecture using React. Each UI section, including the map, toolbar, menus, and detail editor, is encapsulated in independent components. These components communicate through React's state management system and the MapContext to ensure that UI updates remain efficient and consistent.
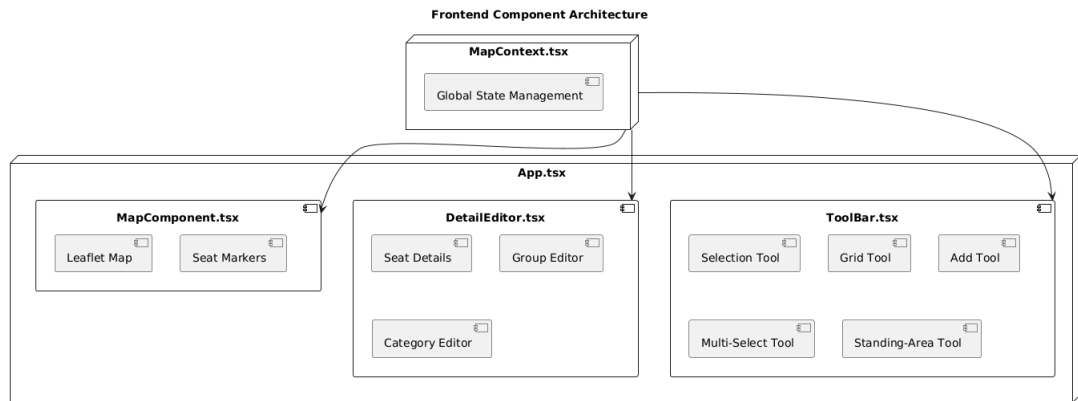

Figure 3: Frontend Component Architecture Overview (Rough Overview)

The diagram in Figure **??** illustrates the core interactive components of the SeatGen frontend. However, it does not represent all UI elements, such as the menu, landing page, modals, and settings, which are also essential for the overall user experience.

Beyond the primary seating map, detail editor, and tool system, SeatGen's frontend also includes:

- **Landing Page (LandingPage.tsx)**: The first screen that users see upon opening SeatGen. It provides an introduction to the tool and a project overview for first-time users.

- **Home Menu (Home.tsx)**: This serves as the main project selection interface. Users can choose between loading an already saved seating plan or creating a

new one. The UI provides an intuitive and minimalistic selection process while ensuring users can access their projects quickly.

- **Menu and Navigation (Menu.tsx)**: The global navigation system connects different views within SeatGen. It allows users to switch between the seating map editor, settings, and export functionalities. The navigation is designed to remain persistent throughout the application to provide seamless transitions between different tasks.

- **Settings Panel (SettingsPanel.tsx)**: This panel allows users to configure global preferences that influence the overall seating arrangement experience. Options include:

  - **Default seat categories**: Users can predefine categories to streamline seat assignments.

  - **Theme selection**: Dark or light mode based on user preference.

  - **Seat map scaling**: Allows fine-tuned adjustments of grid seat density.

- **Modals and Popups**: The frontend includes various popups and confirmation dialogs:

  - **Modal.tsx**: Used for confirmations, warnings, and additional actions.

  - **LeavePagePopup.tsx**: Warns users about unsaved changes before leaving.

These elements, though not included in the diagram, play a significant role in navigating and managing seat plans efficiently and further improving the user experience.

4.1.1 MapComponent.tsx: Interactive Map Rendering

The **MapComponent.tsx** is one of the most critical components in SeatGen, responsible for rendering and managing the stadium seating map. It integrates with **Leaflet.js** to provide interactive seat visualization, selection, and manipulation.

Key Responsibilities:

- Initializes and manages the **Leaflet map instance**.

- Loads seat and standing area data dynamically from the backend.

- Renders seats and standing areas as **interactive markers and polygons**.

- Handles user interactions such as **clicking, selecting, and dragging seats**.

- Implements **state synchronization** with the global context (**MapContext.tsx**).

- Provides support for **tool-based seat editing** (e.g., adding, deleting, moving).

Component Structure:

- **State Management:** Uses `useState`, `useEffect`, and `useContext` to manage map data.

- **Leaflet Integration:** Utilizes `MapContainer` from `react-leaflet` for seamless map rendering.

- **Performance Enhancements:** Implements `useCallback` and `useRef` to optimize re-renders.

Map Initialization and Leaflet Integration

The **MapComponent.tsx** initializes the Leaflet map using the react-leaflet `MapContainer` component.

Listing 2: Initializing Leaflet Map in React

```
const MapComponent: FC<MapProps> = ({ editable: initialEditable }) => {
    const context = useMapContext(); // Access global state (MapContext.tsx)
    const { bucketName, mapName } = useParams(); // Retrieve map identifiers from
        URL params

    const [editable, setEditable] = useState(initialEditable ?? true);
    const mapRef = useRef<L.Map>(null);

    return (
        <MapContainer
            crs={L.CRS.Simple} // Uses a flat, pixel-based coordinate system
            className="leaflet-container h-full w-full"
            ref={mapRef}
            center={[context.mapInfo.tileSize / (-2), context.mapInfo.tileSize /
                (2)]}
            zoom={context.mapInfo.defaultZoom}
            maxZoom={context.mapInfo.maxZoom}
            minZoom={context.mapInfo.minZoom}
            scrollWheelZoom={true}
            zoomControl={false}
            doubleClickZoom={false}
            preferCanvas={true}
            dragging={true}
            tap={false}
            renderer={L.canvas()} // Use Canvas for better performance
        >
            <TileLayer tms={true}
                url={`${context.mapInfo.mapDto.baseUrl}/{z}/{x}/{y}.png`} />
        </MapContainer>
    );
};
```

Custom CRS (Coordinate Reference System):

- Uses **L.CRS.Simple**, a **2D pixel-based coordinate system**.

- Unlike traditional geographic maps, SeatGen does not need a curved map.

- All coordinates can be converted to a Cartesian X/Y grid.

Custom Rendering Engine:

- Uses **L.canvas()** instead of SVG for performance.

- Enables handling of thousands of seat markers.

- Reduces DOM load by rendering elements in a drawing surface.

Dynamically Loading Map Tiles:

- The tile URL is dynamically constructed based on `context.mapInfo`.

- Tiles are loaded asynchronously to improve map load times.

Fetching Seat and Standing Area Data

The MapComponent.tsx fetches seat and standing area data from the backend when the component mounts. This is done using the api client in the useEffect hook.

Listing 3: Fetching Seat and Category Data

```
1   useEffect(() => {
2       if (bucketName && mapName) {
3           // Fetch basic stadium map information
4           seatgenApiClient.api.info(bucketName, mapName)
5               .then(response => context.setMapInfo(response.data))
6               .catch(error => console.error('Error fetching map info:', error));
7
8           // Fetch seat categories before retrieving individual seats
9           seatgenApiClient.api.getAllCategories().then((r) => {
10              if (r.ok) {
11                  context.setCategories(r.data);
12                  seatgenApiClient.api.getSeatsByMap(bucketName,
                        mapName).then(response => {
13                      if (r.ok) {
14                          context.setSeats(response.data.map((s) => ({
15                              id: s.seatId!,
16                              position: { lat: s.xcoord!, lng: s.ycoord! },
17                              category: r.data.find(it => it.id === s.categoryId) ??
                                    null
18                          })));
19                      }
20                  });
21              }
22          });
23
24          // Fetch standing areas
25          seatgenApiClient.api.getSectorByMap(bucketName, mapName).then((r) => {
26              if (r.ok) {
27                  context.setStandingAreas(r.data.map((data) => ({
28                      id: data.id!,
29                      name: data.name!,
30                      capacity: data.capacity,
31                      coordinates: data.coordinates?.map((c) => new L.LatLng(c.x!,
                            c.y!)) ?? [],
32                      selected: false
33                  })));
34              }
35          });
36
37      } else {
```

```
38          console.error("BucketName or MapName not set");
39      }
40  }, [bucketName, mapName]);
```

Breakdown of Logic:

- Fetches stadium metadata (mapInfo) and sets it globally.

- Retrieves seat categories.

- Retrieves seat positions from the backend and maps them into React state.

- Ensures data consistency by linking seats to their corresponding categories.

Example of Mapped Seat Object:

Listing 4: Seat Object in State

```
1  {
2      id: 1234,
3      position: { lat: 48.3069, lng: 14.2858 }, // Example coordinates
4      category: { id: 2, name: "VIP", color: "#FFD700" } // Associated category
5  }
```

Fetching Standing Area / Sector Data

In addition to seats, the component also retrieves standing areas, which are handled separately.

Listing 5: Fetching Standing Areas

```
1  seatgenApiClient.api.getSectorByMap(bucketName, mapName).then((r) => {
2      if (r.ok) {
3          context.setStandingAreas(r.data.map((data) => ({
4              id: data.id!,
5              name: data.name!,
6              capacity: data.capacity,
7              coordinates: data.coordinates?.map((c) => new L.LatLng(c.x!, c.y!)) ??
                    [],
8              selected: false
9          })));
10     }
11  });
```

Explanation:

- The API returns a list of sector polygons.

- Each area consists of a unique ID, name, capacity, and a list of coordinates.

- Coordinates are transformed into Leaflet's LatLng format to be rendered as a polygon.

Example of a Standing Area Object in State:

Listing 6: Standing Area Object in State

```
1  {
2      id: 69420,
3      name: "Sektor 69",
4      capacity: 187,
5      coordinates: [
6          { lat: 48.3069, lng: 14.2858 },
7          { lat: 48.3075, lng: 14.2862 },
8          { lat: 48.3080, lng: 14.2856 }
9      ],
10     selected: false
11 }
```

State Management and Performance

- `useEffect` Dependency Array:

  - Ensures the API calls only run when `bucketName` or `mapName` change.

  - Prevents unnecessary re-fetching on every render.

- Efficient State Updates:

  - Avoids unnecessary re-renders by batching state updates for seats and standing areas.

  - No prop-drilling by storing globaly needed fetched data in the `MapContext`.

- Error Handling:

  - If any API call fails, an error is logged, and the operation is skipped.

  - Ensures that failures in one request do not crash the entire component.

Rendering Seats and Handling Selection

Each seat in the stadium is rendered as a Leaflet marker, allowing users to interact with them dynamically. The selection mechanism is designed to provide an intuitive experience while supporting multi-selection for bulk operations.

Rendering Seat Markers:

Listing 7: Rendering and Selecting Seats

```
1  const handleSeatClick = useCallback((id: number, event: L.LeafletMouseEvent) => {
2      const isCtrlPressed = event.originalEvent?.ctrlKey;
3
4      context.setSeats(prevSeats => prevSeats.map(seat => {
5          if (seat.id === id) {
6              const selected = isCtrlPressed ? !seat.selected : true;
7              return { ...seat, selected };
8          }
9          return isCtrlPressed ? seat : { ...seat, selected: false };
10     }));
11
```

```
12        setOpenSideBar(true);
13   }, [context]);
```

How This Works:

- Clicking a seat toggles its selected state.

- Holding **Ctrl** allows multi-selection, useful for bulk actions.

- Clicking a seat opens the **DetailEditor sidebar** for further modifications.

Implementing Live Seat Dragging and Moving

In SeatGen, users can drag and reposition multiple selected seats dynamically. The challenge was ensuring that **all** selected seats move smoothly and live directly following the cursor while keeping their relative distances intact. To achieve this, a real-time position tracking mechanism was implemented using Leaflet events and React state updates.

Tracking Initial Positions

Before moving seats, their initial positions are stored so that relative offsets can be preserved:

Listing 8: Storing Initial Positions Before Dragging

```
1    const initialPositionsRef = useRef<{ [key: number]: L.Point }>({});
2
3    const storeInitialPositions = useCallback(() => {
4        const map = mapRef.current;
5        if (!map) return;
6
7        initialPositionsRef.current = {};
8        selectedSeats.forEach(seat => {
9            initialPositionsRef.current[seat.id] =
                  map.latLngToLayerPoint(seat.position);
10       });
11
12       // Register move action for undo functionality
13       context.doAction(new MoveAction(selectedSeats, context.setSeats));
14   }, [selectedSeats, context]);
```

How This Works:

- A reference (`initialPositionsRef`) is created to store the pixel positions of selected seats by converting the Latitude and Longitude into Layer Points which are x and y Cartesian System points.

- These positions are saved when the user starts dragging a seat.

- The relative distance between seats is maintained, preventing unwanted misalignment.

Updating Seats in Real-Time During Dragging

While the user drags a seat, the drag distance is calculated and apply it to all selected seats:

Listing 9: Updating Seat Positions During Dragging

```
1  const updateSelectedSeatsPosition = useCallback((draggedSeatId: number,
       newLatLngPosition: { lat: number; lng: number }) => {
2      const map = mapRef.current;
3      const primarySeat = context.seats.find(s => s.id === draggedSeatId);
4
5      if (!map || !primarySeat || !primarySeat.selected) return;
6
7      const newPosition = map.latLngToLayerPoint(newLatLngPosition);
8      const oldPosition = initialPositionsRef.current[draggedSeatId];
9
10     const deltaX = newPosition.x - oldPosition.x;
11     const deltaY = newPosition.y - oldPosition.y;
12
13     context.setSeats(prevSeats =>
14         prevSeats.map(seat => {
15             if (seat.selected) {
16                 const initialPosition = initialPositionsRef.current[seat.id];
17                 const newPosX = initialPosition.x + deltaX;
18                 const newPosY = initialPosition.y + deltaY;
19                 const newLatLngPos = map.layerPointToLatLng(new L.Point(newPosX,
                       newPosY));
20                 return { ...seat, position: newLatLngPos };
21             }
22             return seat;
23         })
24     );
25
26     // Update MoveAction for undo tracking
27     const currentAction = context.getCurrentAction();
28     if (currentAction instanceof MoveAction) {
29         currentAction.setNewSeats(selectedSeats);
30     }
31 }, [context.seats, context]);
```

Breakdown of Logic:

- The drag delta (change in X/Y position) between the starting position and the new cursor position is calculated.

- The same delta is applied to all selected seats, ensuring they move together.

- Positions are converted between LatLng (geo-coordinates) and pixel points, so dragging works consistently at different zoom levels.

- Changes using `MoveAction` are tracked, allowing the operation to be undone if needed.

Optimizations and Challenges

Major Challenges Encountered:

- Preventing position drift when switching between zoom levels.

- Ensuring smooth movement with large seat selections.

- Avoiding excessive re-renders that slow down performance.

Performance Optimizations:

- Used `useRef` to store initial positions instead of state (prevents extra re-renders).

- Applied batch updates for all selected seats instead of updating them individually.

- Optimized Leaflet latLng to pixel conversion for to be able to drag the seats naturally (direct manipulation [?]).

Conclusion

The live dragging implementation allows users to dynamically reposition multiple seats while keeping their relative distances intact. By leveraging Leaflet's coordinate system and real-time state updates, a fluid and high-performance dragging experience whilst also beeing able to undo and redo the movement was achieved.

Managing Standing Areas and Sectors

SeatGen allows the definition of standing areas / sectors, which differ from regular seats by not being assigned individual markers but instead represented as polygonal sectors. Each standing area has a defined capacity, ensuring ticketing restrictions.

Standing Area Features:

- **Custom Polygons:** Users define standing areas by selecting points on the map.

- **Capacity Control:** Limits the number of tickets available per standing area.

- **Category Assignment:** Standing areas can be assigned different categories.

- **Real-time Editing:** Areas can be renamed, and deleted dynamically.

Fetching Standing Areas from the Backend:

Listing 10: Fetching Standing Areas

```
1  seatgenApiClient.api.getSectorByMap(bucketName, mapName).then((r) => {
```

```
2        if (r.ok) {
3            context.setStandingAreas(r.data.map((data) => {
4                const standingArea: StandingArea = {
5                    id: data.id!,
6                    name: data.name!,
7                    capacity: data.capacity,
8                    coordinates: data.coordinates?.map((c) => {
9                        return new LatLng(c.x!, c.y!);
10                   })??[],
11                   selected: false
12               }
13               return standingArea
14           }))
15       }
16  });
```

Standing Area Selection:

Listing 11: Handling Standing Area Selection

```
1   const handlePolygonClick = useCallback((id: number, event: L.LeafletMouseEvent) =>
        {
2       const isCtrlPressed = event.originalEvent?.ctrlKey;
3
4       // Deselect all seats when a polygon is clicked
5       context.setSeats(prevSeats => prevSeats.map(seat => ({
6           ...seat,
7           selected: false
8       })));
9
10      const selectedStandingAreaIds: number[] = [];
11      context.setStandingAreas(prevAreas => {
12          return prevAreas.map(area => {
13              if (area.id === id) {
14                  const selected = isCtrlPressed ? !area.selected : true;
15                  if (selected) selectedStandingAreaIds.push(area.id);
16                  return { ...area, selected };
17              }
18              if (!isCtrlPressed) {
19                  return { ...area, selected: false };
20              }
21              if (area.selected) selectedStandingAreaIds.push(area.id);
22              return area;
23          });
24      });
25      // Update the selected standing area IDs in context
26      context.setSelectedStandingAreaIds(selectedStandingAreaIds);
27      setOpenSideBar(true);
28  }, [context, setOpenSideBar]);
```

Selection and Editing Process:

- Clicking a standing area toggles its selected state.

- In the Detail Editor panel renaming, capacity adjustments, and category (including color coding) updates are possible.

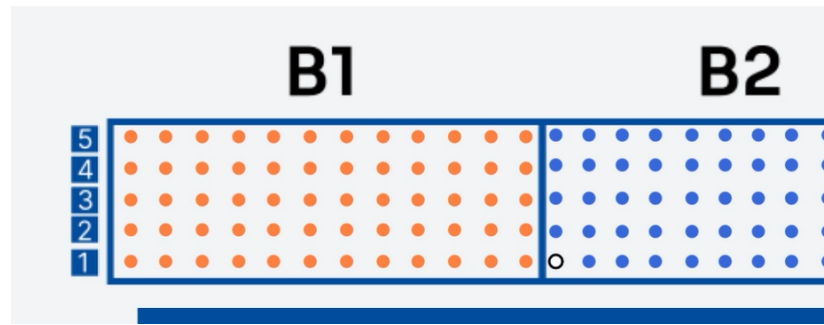- The user can resize the polygons on creation to modify the area covered.

Figure 4: Map, Seats with Category Color, and Selected Seat in MapComponent.tsx

Event Handling in Leaflet

SeatGen relies heavily on Leaflet event handling to manage user interactions dynamically.

Global Click Handling:

```
Listing 12: Handling Global Click Events
1  const MapEvents = () => {
2      useMapEvents({
3          click(e) {
4              context.setSeats(prevSeats => prevSeats.map(seat => ({ ...seat,
                   selected: false })));
5              context.setSelectedStandingAreaIds([]); //Deselects Standing Areas
                   because Seats are selected
6          }
7      });
8      return null;
9  };
```

Drag Events for Seat Movement:

```
Listing 13: Handling Seat Drag Events
1  // Function definitions for handling drag events
2  const handleDragStart = (seatId: number) => {
3      storeInitialPositions();
4  };
5
6  const handleDragEnd = (seatId: number, newPosition: L.LatLng) => {
7      updateSelectedSeatsPosition(seatId, newPosition);
8  };
9
10 // Applying event listeners to each seat marker
11 const renderedSeats = useMemo(() => {
12     return context.seats.map(seat => (
13         <Seat
14             key={seat.id}
15             id={seat.id}
16             position={seat.position}
17             updatePosition={updateSeatPosition}
18             updateSelectedSeatsPosition={updateSelectedSeatsPosition}
19             storeInitialPositions={storeInitialPositions}
20             tooltipText={seat.tooltipText}
21             onClick={(event: LeafletMouseEvent) => handleSeatClick(seat.id, event)}
22             onDragStart={() => handleDragStart(seat.id)}
23             onDragEnd={(event) => handleDragEnd(seat.id, event.target.getLatLng())}
24             draggable={editable}
25             selected={seat.selected}
26             category={seat.category}
27         />
28     ));
29 }, [context.seats, selectedSeats]);
```

- The handleDragStart function is called when a user begins dragging a seat. It stores the initial positions of selected seats to ensure relative movement.

- The handleDragEnd function finalizes the new seat positions when dragging stops.

- Event listeners (onDragStart and onDragEnd) are added to each Seat component to trigger these functions dynamically.

- The useMemo hook optimizes rendering performance by ensuring seat markers are not unnecessarily re-created during re-renders.

Event Handling Summary:

- Click Events: Used for selecting seats and standing areas.

- Drag Events: Applied to dynamically reposition seats.

- Map Events: Ensure global deselection when clicking outside elements.

Saving and Syncing with the Backend

SeatGen implements an asynchronous saving mechanism that synchronizes data with the backend.

Warning Users Before Leaving Without Saving (LeavePagePopup.tsx):

The `LeavePagePopup.tsx` component ensures that users are warned before navigating away from the application when they have unsaved changes. This prevents accidental data loss and provides an opportunity to save progress before exiting.

- Detects unsaved changes via the `hasUnsavedChanges` prop.

- Attaches a `beforeunload` event listener to prevent accidental exits.

- Displays a native browser warning when users try to leave.

- Removes the event listener when no longer needed to optimize performance.

Listing 14: Auto-Saving on Unload

```
1    interface FormPromptProps {
2        hasUnsavedChanges: boolean;
3    }
4
5    export const LeavePagePopup: FC<FormPromptProps> = ({ hasUnsavedChanges }) => {
6        useEffect(() => {
7            const onBeforeUnload = (e: BeforeUnloadEvent) => {
8                if (hasUnsavedChanges) {
9                    e.preventDefault();
10                   e.returnValue = "";
11               }
12           };
13           window.addEventListener("beforeunload", onBeforeUnload);
```

```
14              return () => {
15                  window.removeEventListener("beforeunload", onBeforeUnload);
16              };
17          }, [hasUnsavedChanges]);
18      };
```

How It Works:

- The `hasUnsavedChanges` prop determines whether to enable the warning.

- When unsaved changes are detected, a `beforeunload` event listener is added.

- If the user attempts to leave the page, the browser displays a warning.

- The event listener is removed when the component unmounts to prevent memory leaks.

Batch Save:

Listing 15: Batch Save Mechanism

```
1  const saveChanges = useCallback(() => {
2      seatgenApiClient.api.saveSeats(bucketName, mapName, context.seats.map(seat =>
           ({
3          id: seat.id,
4          x: seat.position.lat,
5          y: seat.position.lng,
6          categoryId: seat.category?.id
7      })))
8      .then(() => enqueueSnackbar("Changes saved successfully!", { variant:
           "success" }));
9  }, [context.seats]);
```

Batch Save Mechanism:

- Instead of saving each individual seat change separately, SeatGen groups multiple seat updates into a single API request to reduce network overhead and improve efficiency.

- The `useCallback` hook ensures that changes get only saved when `context.seats` changes, preventing redundant function executions.

- `seatgenApiClient.api.saveSeats` sends the updated seat data to the backend, including the seat ID, coordinates, and category ID.

- Upon a successful save, a snackbar notification is displayed to confirm the operation.

- SeatGen maintains the action history so users can revert unintended changes before saving.

This saving mechanism ensures that the application remains responsive while keeping data integrity intact, even in scenarios where users forget to manually save their progress they will be reminded.

Summary

The **MapComponent.tsx** is the core of SeatGen's interactive seating system. It efficiently manages:

- Interactive Leaflet-based Map Rendering

- Seat and Standing Area Rendering

- Group Selection and Bulk Editing

- Drag-and-Drop Seat Repositioning

- Event Handling for User Interaction

- Synchronizing data with global state (textttMapContext.tsx)

- Asynchronous Backend Synchronization

It ensures smooth operation even for large stadium layouts with thousands of seats.

4.1.2 DetailEditor.tsx: Editing Attributes

The **DetailEditor.tsx** component provides an interface for modifying selected seats, managing seat groups, categories, and configuring standing areas. It plays a huge role in SeatGen's user interaction system by allowing users to efficiently modify stadium layouts in a structured and intuitive manner.
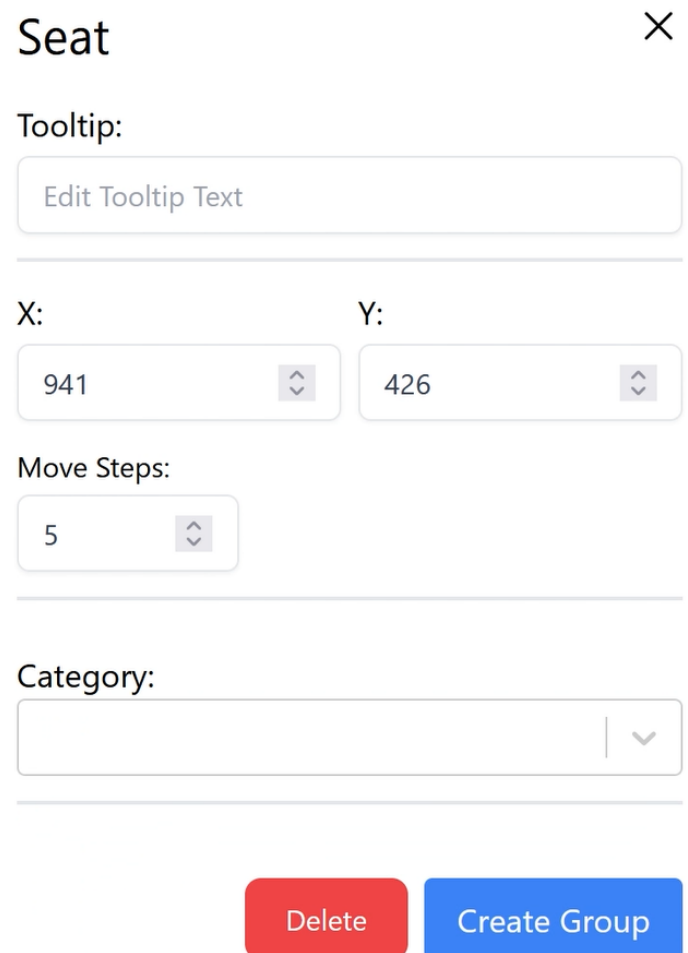
Key Responsibilities:

- **Editing Individual Seats:** Users can update seat tooltips, positions, and categories.

- **Managing Seat Groups:** Enables users to create, merge, and delete groups of seats.

- **Standing Area Editing:** Supports renaming and capacity adjustments for standing areas.

- **Category Assignment:** Allows users to assign pricing tiers and colors to seats.

4.1.3 Component Structure

The **DetailEditor.tsx** has the following core sections:

- **Seat Editing Panel:** Displays detailed information for selected seats and allows modifications.

- **Group Management Panel:** Handles seat grouping and bulk operations.

- **Standing Area Editing Panel:** Enables editing of standing areas, including name and capacity.

- **Deletion and Bulk Actions:** Supports removing selected seats, groups, or selected standing areas.

Figure 5: Seat Editing Panel in DetailEditor.tsx

Figure **??** illustrates the seat editing panel, where users can adjust tooltips, assign categories, and delete seats.

4.1.4 Managing Seat Attributes

When a seat is selected, the editor provides multiple options for modification.

Updating Tooltip Text

Users can edit tooltip descriptions for seats, making it easier to add position information or other special notes.

```
Listing 16: Updating Tooltip for Selected Seats
1  <input
2      type="text"
3      value={props.currentTooltip}
4      onChange={props.handleExternalTooltipChange}
5      placeholder="Edit Tooltip Text"
6  />
```

How It Works:

- The text input dynamically updates the tooltip for all selected seats by calling the `handleExternalTooltipChange` function.

- Changes are stored in the global state.

- Provides immediate visual feedback to the user.

Adjusting Seat Position via UI

Instead of manually dragging seats on the map, users can fine-tune their positions numerically. Also, adjusting the steps of one increment when using the arrow keys to position the seats perfectly is possible.

```
Listing 17: Updating Seat Coordinates
1  const updateSeatPosition = (x: number, y: number) => {
2      const latLng = props.mapRef.current!.layerPointToLatLng(new L.Point(x, y));
3      props.updateSelectedSeatsPosition(selectedSeats[0].id, { lat: latLng.lat, lng:
           latLng.lng }, true);
4  };
```

How It Works:

- Converts user-inputted X/Y values into map coordinates.

- Updates seat position dynamically.

- With this the movement is optimized for both manual entry and real-time adjustments.

4.1.5 Group Management and Multi-Selection

Grouping seats allows users to efficiently manage large stadium sections.

Groups and Multi Selection and Deletion

SeatGen supports the concept of seat groups, allowing users to efficiently manipulate multiple seats at once. Grouping seats enables bulk operations such as movement, category assignment, and section-wide modifications, making it particularly useful for managing large stadium layouts.

Use Cases for Seat Groups:

- **Bulk Editing:** Modify multiple seat attributes simultaneously.

- **Efficient Repositioning:** Move multiple seats while maintaining relative positioning.

- **Category Assignment:** Assign pricing tiers and access restrictions to an entire section.

- **Simplified Deletion:** Remove entire seat groups without manually selecting each seat.

Creating a Seat Group:

```
Listing 18: Creating Seat Groups
1   const createGroup = useCallback(() => {
2       context.doAction(new CreateGroupAction(setSeatGroups, selectedSeats));
3   }, [selectedSeats, context]);
```

How It Works:

- The function retrieves all currently selected seats.

- The CreateGroupAction is executed, registering the selected seats as a group.

- The group ID is assigned, and the group is stored in the global state.

Deleting a Seat Group:

```
Listing 19: Deleting Seat Groups
1   const deleteGroup = useCallback((groupId: number) => {
2       context.doAction(new DeleteGroupAction(setSeatGroups, groupId));
3   }, [context]);
```

Group Deletion Process:

- The DeleteGroupAction removes all seats within that group using the groupId.

- The global context state updates accordingly.

- The operation is reversible via the undo stack.

Furthermore, users can merge existing groups or split them dynamically, enabling flexible seat management. This allows for unlimited subgrouping, making the editor more intuitive and efficient.

Seat Categories

Seat categories allow users to classify seating arrangements based on (pricing) tiers, accessibility, and special designations such as VIP areas or restricted sections. In SeatGen, every seat can be assigned a category that determines its visual representation and ticketing attributes.
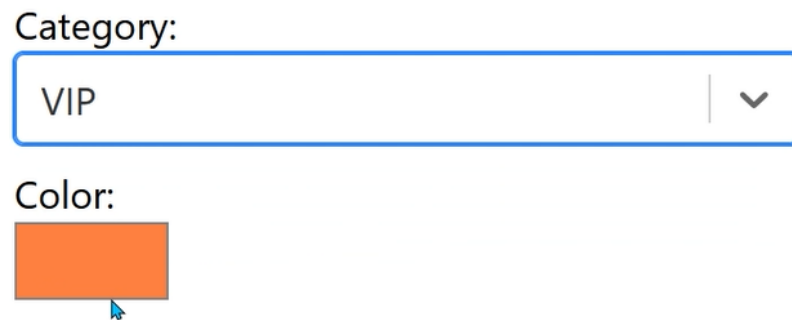

Figure 6: Category Selection in DetailEditor.tsx

Features of Seat Categories:

- **Color-Coding:** Each category is assigned a color for clear visualization.

- **Pricing Information:** Categories define pricing tiers, ensuring correct ticket pricing.

- **Flexible Assignments:** Seats can be reassigned to different categories as needed.

- **Bulk Category Updates:** Multiple seats can be assigned a category simultaneously.

Category Data Model

Each seat category is stored as an object that holds classification data:

Listing 20: Seat Category Data Model

```
1  interface Category {
2      id?: number;
3      name: string; // Example: "VIP", "General Admission"
4      color: string; // Hex code for UI representation
5      price: number; // Ticket price associated with this category
6  }
```

Assigning Categories to Seats

When a user selects a seat, they can assign or change its category using the **DetailEditor.tsx**.

Listing 21: onChange Assigning Categories to Selected Seats

```
1  onChange={(newVal) => {
2      console.log(newVal)
3      context.doAction(new UpdateCategoryAction(selectedSeats, newVal ? newVal.value
           : null))
4      setCurrentSelectedCategory(newVal ? newVal.value : null);
5  }}
```

Listing 22: Assigning Categories to Selected Seats

```
1  execute = () => {
2      this._oldCategories = this._selectedSeats.map(seat => seat.category)
3      this._selectedSeats.forEach((seat) => {
4          seat.category = this._newCategory
5      });
6  }
```

How It Works:

- The function in `UpdateCategoryAction` loops through all selected seats.

- The new category is assigned based on the provided `categoryId`.

- The UI updates instantly, applying the new color and classification.

Category Management in the UI

Users can manage categories by:

- Creating new categories with custom colors and pricing.

- Editing existing categories, updating names, prices, or colors.

- Deleting unused categories.

Listing 23: Managing Categories

```
1  onCreateOption={(inputValue) =>
2      seatgenApiClient.api.createCategory({name: inputValue, color:
           "#3768db"}).then((r) => {
3          if (r.ok) {
```

```
 4              enqueueSnackbar("Successfully created category", {variant: 'success'})
 5              selectedSeats.forEach((seat) => {
 6                  seat.category = r.data;
 7              });
 8              context.setCategories(prevCategories => [...prevCategories, r.data]);
 9              context.categoryChecksums.current.set(r.data.id!, objectHash(r.data))
10              setCurrentSelectedCategory(r.data)
11          } else {
12              console.error('Error creating category:', r.statusText);
13              enqueueSnackbar('Error creating category', {variant: 'error'});
14          }
15      })
16  }
```

Listing 24: Managing Category Color

```
1  <input
2      type="color"
3      value={currentSelectedCategory.color ?? "#3768db"}
4      onChange={(newColor) => {
5          selectedSeats[0].category!.color = newColor.target.value
6          setCurrentSelectedCategory({...currentSelectedCategory, color:
               newColor.target.value})
7      }));
8  }}/>
```

Category Visualization on the Map

Seat categories are visually represented by color-coded markers in **MapComponent.tsx**.
Each seat marker dynamically updates based on its assigned category.

Listing 25: Rendering Seat Markers with Categories

```
1  const renderedSeats = seats.map(seat => (
2      <SeatMarker
3          key={seat.id}
4          seat={seat}
5          color={seat.category?.color || "gray"}
6          onClick={() => handleSeatClick(seat.id)}
7      />
8  ));
```

How It Works:

- Each seat marker inherits the category's color.

- Unassigned seats default to a neutral color (gray).

- Selecting a seat allows users to change its category.

Conclusion

Categories play a crucial role in SeatGen, providing a structured way to manage
ticket pricing and seat classification. By integrating category assignment with group
selection and bulk operations, users can efficiently update stadium layouts with minimal
effort.

4.1.6 Standing Area Editing

Standing areas in SeatGen are defined as polygonal sectors rather than individual seats. Unlike seats, which are represented as distinct markers, standing areas are implementd using a defined boundary of polygons. Each standing area has a name, and a maximum capacity.

Key Features of Standing Areas:

- **Custom Polygonal Boundaries:** Users can define standing areas by selecting points on the map.

- **Capacity Control:** Each area has a maximum capacity limit.

- **Real-Time Editing:** Users can rename standing areas and adjust their capacities dynamically.

- **Deletion and Reconfiguration:** Existing standing areas can be removed or modified at any time.

Standing Area Data Model

Each standing area is stored as an object in the application state.

Listing 26: Standing Area Data Model

```
1  interface StandingArea {
2      id: number;
3      name: string; // Display name of the standing area
4      capacity: number; // Maximum allowed attendees
5      coordinates: L.LatLng[]; // List of boundary points defining the area
6      selected: boolean; // Boolean flag indicating selection state
7  }
```

Creating and Selecting Standing Areas

Standing areas are created by defining polygonal boundary coordinates on the map. Once an area is selected, it becomes editable in the `DetailEditor.tsx`.

Renaming a Standing Area

Users can rename standing areas directly in the `DetailEditor.tsx` panel.

Listing 27: Renaming Standing Areas

```
1  const handleStandingNameChange = (name: string) => {
2      context.setStandingAreas(prev => prev.map(area =>
3          context.selectedStandingAreaIds.includes(area.id)
4              ? { ...area, name: name }
```

```
5                 : area
6        ));
7    };
```

How It Works:

- The function updates the name of all selected standing areas.

- Changes are reflected instantly in the UI.

- The new name is stored persistently for future sessions.

Updating Standing Area Capacity

To control attendee limits, each standing area has an adjustable capacity.

Listing 28: Updating Standing Area Capacity

```
1    const handleStandingCapacityChange = (capacity: string) => {
2        context.setStandingAreas(prev => prev.map(area =>
3            context.selectedStandingAreaIds.includes(area.id)
4                ? { ...area, capacity: Number(capacity) }
5                : area
6        ));
7    };
```

Capacity Adjustment Process:
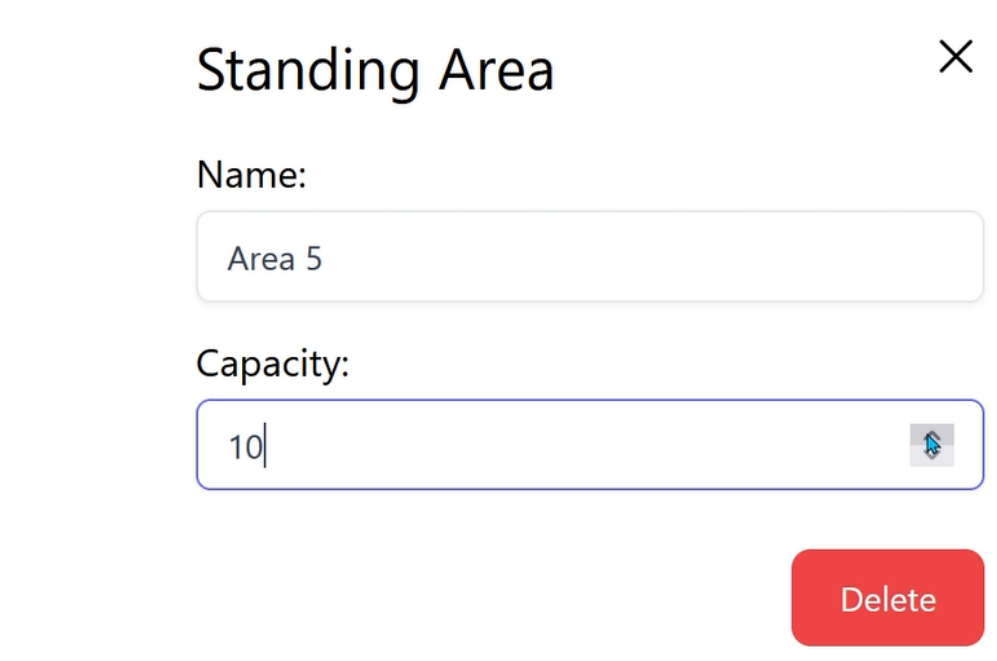
- The function modifies the capacity of all selected standing areas.

- Input validation ensures that only numeric values are accepted.

- The UI dynamically updates, reflecting the new ticketing constraints.

Deleting a Standing Area

Standing areas can be removed using `DeleteStandingAreaAction` when no longer needed which is also undoable for user convenience.

Listing 29: Deleting Standing Areas

```
1    context.doAction(new DeleteStandingAreaAction(selectedStandingAreas, context))
```

Figure 7: Editing Standing Areas in DetailEditor.tsx

Figure **??** illustrates the interface for editing standing areas, including renaming, capacity adjustment, and deletion.

Standing areas in SeatGen offer a structured approach to handling non-seated sections of a stadium. By allowing dynamic capacity control and easy renaming, the system ensures that standing areas remain flexible and adaptable. Users can efficiently create, edit, and remove standing areas based on event needs, making stadium layouts highly customizable.

Beyond general standing areas, this feature can also be used to designate specific sections for specialized needs. For example, stadiums may allocate certain sectors for **wheelchair-accessible areas** or **priority seating for individuals with mobility impairments**. This flexibility ensures that accessibility requirements can be met while maintaining a clear and organized seating plan.

Summary

DetailEditor.tsx is a key component within SeatGen, offering intuitive tools for modifying stadium layouts. It enables:

- **Seat Editing:** Real-time adjustments to tooltips, positions, and assignments.

- **Group Management:** Merging, splitting, and deleting seat groups efficiently.

- **Category Assignment:** Bulk updates with intuitive color-coded visualization.

- **Standing Area Modifications:** Easy editing of boundaries and capacities.

- **Seamless Integration:** Ensures consistent updates via `MapContext.tsx`.

With its structured approach and live updates, **DetailEditor.tsx** ensures flexible and efficient stadium configuration.

4.1.7 ToolBar.tsx: Tool Management and User Interaction

The **ToolBar.tsx** component provides an intuitive interface for tool selection, map management, and saving operations within SeatGen. It enables users to interact efficiently with the stadium layout by offering a structured toolbar for tool activation, map naming, and data persistence.

Key Responsibilities:

- **Tool Selection:** Provides quick access to various editing tools.

- **Map Title Editing:** Allows renaming the map for better organization.

- **Saving and Previewing:** Ensures seamless data persistence and view-only mode.

Component Structure

The **ToolBar.tsx** consists of the following elements:

- **Tool Icons:** Represent different editing modes (selection, adding seats, grid placement, area selection, standing area placement).

- **Map Name Editor:** Enables modifying the displayed name of the stadium layout.

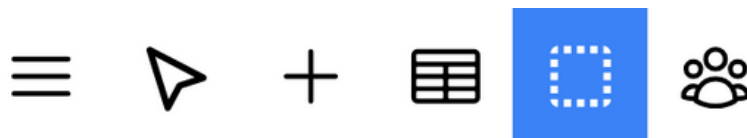- **Save and Preview Buttons:** Handles storing changes and switching to a preview mode.

Figure 8: Toolbar Icons in ToolBar.tsx

Figure **??** illustrates the available tool icons, where users can select different functionalities. The currently active tool is highlighted.

Tool Selection and Activation

The toolbar supports multiple tools, each represented by an icon. Users can switch between tools to perform various stadium layout modifications.

```
1   const tools: Tool[] = [
2       {
3           id: "addTool",
4           icon: <PlusIcon></PlusIcon>,
5           onSelect: () => handleToolSelect((e) => {
6               props.addSeat(e.latlng.lat, e.latlng.lng)
7           }, "addTool", "cell"),
8           hotkey: "c"
9       },
10      {
11          id: "addGridTool",
12          icon: <TableCellsIcon></TableCellsIcon>,
13          onSelect: () => handleToolSelect(() => {
14          }, "addGridTool", "cell"),
15          hotkey: "g"
16      },
17      {
18          id: "standingAreaTool",
19          icon: <UserGroupIcon />,
20          onSelect: () => handleToolSelect(() =>console.log("standingtool"),
                   "standingAreaTool", "crosshair"),
21          hotkey: "s"
22      }
23  ];
```
Listing 30: Defining Available Tools

How It Works:

- Each tool has an icon and an activation function.

- Hotkeys allow quick switching between tools.

- Selected tools modify the user's interaction mode (e.g., clicking, selecting, adding seats).

Editing the Map Name

The toolbar includes an editable text field for renaming the stadium layout. Clicking the pencil icon toggles the edit mode.

```
1   {
2       !editName ? (
3           <>
4               <p>{displayName}</p>
5               <PencilIcon onClick={() => setEditName(true)} className="mapNameEdit
                       cursor-pointer" />
6           </>
7       ) : (
8           <>
9               <input
10                  autoFocus
11                  type='text'
12                  className="text-center"
13                  value={tempDisplayName}
14                  onChange={(e) => setTempDisplayName(e.target.value)}
15              />
```
Listing 31: Editing Map Name

```
16              <CheckIcon onClick={() => {
17                  seatgenApiClient.api.updateMapName({ bucketName:
                        props.mapDto.bucketName, mapName: props.mapDto.mapName,
                        displayName: tempDisplayName })
18                     .then((r) => setDisplayName(r.data.second))
19                  setEditName(false);
20              }} className="cursor-pointer" />
21          </>
22      )
23  }
```

How It Works:

- Clicking the pencil icon allows editing the map title.

- The name is updated and saved via an API request.

- The change is immediately reflected in the UI.

Save and Preview Buttons

The toolbar also contains buttons for saving progress and switching to a preview mode.

Figure 9: Save and Preview Buttons in ToolBar.tsx

```
Listing 32: Saving and Previewing
1  <button className={SgButtonType.primarySmall} onClick={() => {
2      seatgenApiClient.api.saveSeats(context.seats.map((s) => ({
3          id: s.id,
4          x: s.position.lat,
5          y: s.position.lng,
6          categoryId: s.category?.id
7      })))
8      .then(context.updateSaveIndex);
9  }} disabled={isSaveLoading}>
10     {isSaveLoading ? <Spinner size={4} /> : <span>Save</span>}
11 </button>
12
13 <button className={SgButtonType.primarySmall} onClick={() => {
14     props.setEditable(false);
15     setSelectedTool(null);
16 }}>
17     Preview
18 </button>
```

How It Works:

- The `Save` button persists all seat and standing area modifications to the backend.

- A checkmark icon confirms successful saving, while a warning icon indicates unsaved changes.

- The `Preview` button switches the interface to a non-editable mode.

Summary

ToolBar.tsx plays a critical role in SeatGen by providing an accessible interface for stadium layout editing. It includes:

- **Interactive Tool Selection:** Enables switching between different seat and area modification tools.

- **Map Naming:** Allows users to rename and organize stadium layouts.

- **Save and Preview Mechanism:** Ensures data integrity and view-only mode.

By offering an efficient and intuitive interface, **ToolBar.tsx** enhances the overall usability of SeatGen.

4.1.8 React Components and Hooks

4.1.9 MapContext and Global State

4.1.10 Tool System and Event Handling

4.2 Leaflet Integration

To integrate Leaflet[**?**] in SeatGen the React Leaflet library[**?**] was utilized as a wrapper for Leaflet, because of the easier React implementation. The library provides a set of React components for Leaflet maps, instead of just having to use the javascript functions of the Leaflet library. To get started with the integration of a basic map the `MapContainer` component and a map reference. The `MapContainer` is the area in the frontend where the map is displayed. The map reference is used to interact with the map, like adding layers or markers. The following code snippet shows how to create a basic map with the React Leaflet library. To make a map appear, there needs to be a `TileLayer` as a child of the `MapContainer`. There can also be multiple `TileLayer` components, to display different map layers, but it's not needed in the usecase of SeatGen. The one required property of the `TileLayer` is the `url` property, which is the URL of the map tiles. In the `url` property the place where `x`, `y` and `z` are placed is defined by the `{x}`, `{y}` and `{z}` placeholders. The `z` is the zoom level, `x` and `y` are the coordinates of the tile. A key part of the integration is the use of a map reference (mapRef), which allows programmatic interaction with the map instance. This reference

is created using React's useRef hook and is used to manipulate the map dynamically, such as adding layers, adjusting zoom levels, or panning to specific locations.

The integration of custom markers in the map, the `Marker` component is used. The `Marker` component is a child of the `MapContainer` and has a position property.

Lots of different components utilize positions to display them on the map. Leaflet has two kinds of positions, the `LatLng` and the `Point`. The `LatLng` is a geographical point with a latitude and longitude. The `Point` is a point with x and y coordinates in pixels. The `Point` is used to position elements on the map, like markers or popups.

Latitude and longitude are used for representation of Earth's surface. Latitude specifies the north-south position and ranges from -90° (South Pole) to +90° (North Pole). Longitude specifies the east-west position and ranges from -180° to +180°. These coordinates are used in geographic coordinate systems, which are essential for positioning objects on a global scale, but not useful for the usecase of SeatGen. When not transforming the coordinates correctly and using a marker, it can happen, that when moving a marker in a straight line, it will move in a curved direction. This is because of the aforementioned logic of the surface of the Earth. To avoid this, the `LatLng` coordinates can be converted to `Point` coordinates by providing the map reference. An example of such a conversion in the code of Seatgen is in listing **??**

Listing 33: Latitude Longitude and Point conversion

```
1    //Point to LatLng
2    const latLngPosition = map.layerPointToLatLng(new L.Point(x, y));
3
4    //LatLng to Point
5    const pointPosition: Point = map.latLngToLayerPoint(new L.Lat(lat, lng));
```

Leaflet also provides a lot of features which can be used by some part for this editor. This ranges for from fully usable features, that don't need a lot of reconfiguration to work for SeatGen's usecase, to features that need to be reworked or where only a small part of the feature is utilized, and the rest is rewritten. Some of the features that could be just used as they were, were:

- Zoom
- Movement in the map
- Tooltips of markers

SeatGen has a lot more of Leaflet's features implemented, but they are heavily modified. For example: The marker feature was utilized for displaying seats, but other than

the base features everything else isn't provided by Leaflet, but it's implemented here instead.

4.2.1 Writing Extensions

For the bigger changes inside Leaflet itself, SeatGen uses extensions, to modify existing features or even overwrite them. Leaflet provides an easy way for developers to do such a thing like modifying leaflet functionalities. This can be done with the `extend` method that is provided by some Leaflet classes. When overwriting functions, knowledge of the functionality of the leaflet internal functions that want to be overwritten is required. It's recommended to look into Leaflet's source code and study the class before overwriting it. When doing so, it is possible to create new subclasses of the existing class and integrating these new modded subclasses into the map. An example of this in SeatGen is in listing **??**

Listing 34: Modifying Leaflet Features

```
1  L.Map.Multiselect = L.Map.BoxZoom.extend({
2
3      _onMouseDown: function (e) {
4          //...
5          //Business logic for overwriting here
6          //...
7      },
8
9      _onMouseMove: function (e) {
10         //...
11         //Business logic for overwriting here
12         //...
13     },
14
15     _onMouseUp: function (e) {
16         //...
17         //Business logic for overwriting here
18         //...
19     },
20
21     _finish: function () {
22         //...
23         //Business logic for overwriting here
24         //...
25     },
26
27 })
28
29 L.Map.mergeOptions({boxPrinter: true});
30 L.Map.addInitHook('addHandler', 'boxPrinter', L.Map.Multiselect);
31
32 L.Map.mergeOptions({boxZoom: false});
```

In the provided Leaflet extension code, mergeOptions is used to introduce and modify configuration options for the L.Map class. This method allows developers to add custom options to existing Leaflet classes without modifying the core library. By merging options and using addInitHook, the new feature seamlessly integrates with Leaflet maps.

The result is that whenever a map instance is created, new Feature replaces the default BoxZoom functionality if boxPrinter is enabled.

4.2.2 Event Handling

Leaflet provides an extensive event system that allows developers to listen for and handle various user interactions within the map. This event system is crucial for SeatGen, as it enables dynamic updates and interactions based on user actions. Events in Leaflet can be categorized into different types, such as mouse events, keyboard events, and map-specific events.

Commonly used events in SeatGen include:

- `click`: Triggered when a user clicks on the map or an element.
- `mousemove`: Fires whenever the mouse moves over the map, useful for hover effects.
- `drag`: Used to detect when a user drags an element like a marker.

Handling events in Leaflet is straightforward using `useMapEvents`. When using this hook and adding it to the Map, a lot of events can be caught and handled. An example for the usage of this hook with the `click` event is shown in listing **??**.

Listing 35: Handling Events in Leaflet

```
1   const MapEvents = () => {
2       useMapEvents({
3           click(e) {
4               console.log("clicked")
5
6               if (context.selectedToolId === "addTool" && mapClickCallBack) {
7                   console.log("Map clicked for add tool");
8                   mapClickCallBack(e);
9               }
10
11              if (context.selectedToolId === "default" || context.selectedToolId ===
                    "") {
12                  context.setSeats(prevSeats => prevSeats.map(seat => ({ ...seat,
                        selected: false })));
13                  context.setSelectedStandingAreaIds([]);
14              }
15
16          },
17      });
18      return null;
19  };
```

4.3 Landing Page

A landing page was created to give the entire application a more complete look, and instead of dropping the user directly into the application some information about the developers, the application, the technologies and a visual demonstration in the form of

a GIF that shows how a basic workflow with Seatgen could look like. The landing page also has a similar color scheme to the actual application, but it is way more visually pleasing and filled with more animations and gradients, because SeatGen focuses more on the functionality and the speed of the workflow instead of fancy animations that could just distract the user from the work. The landing page is shown in figure **??**.
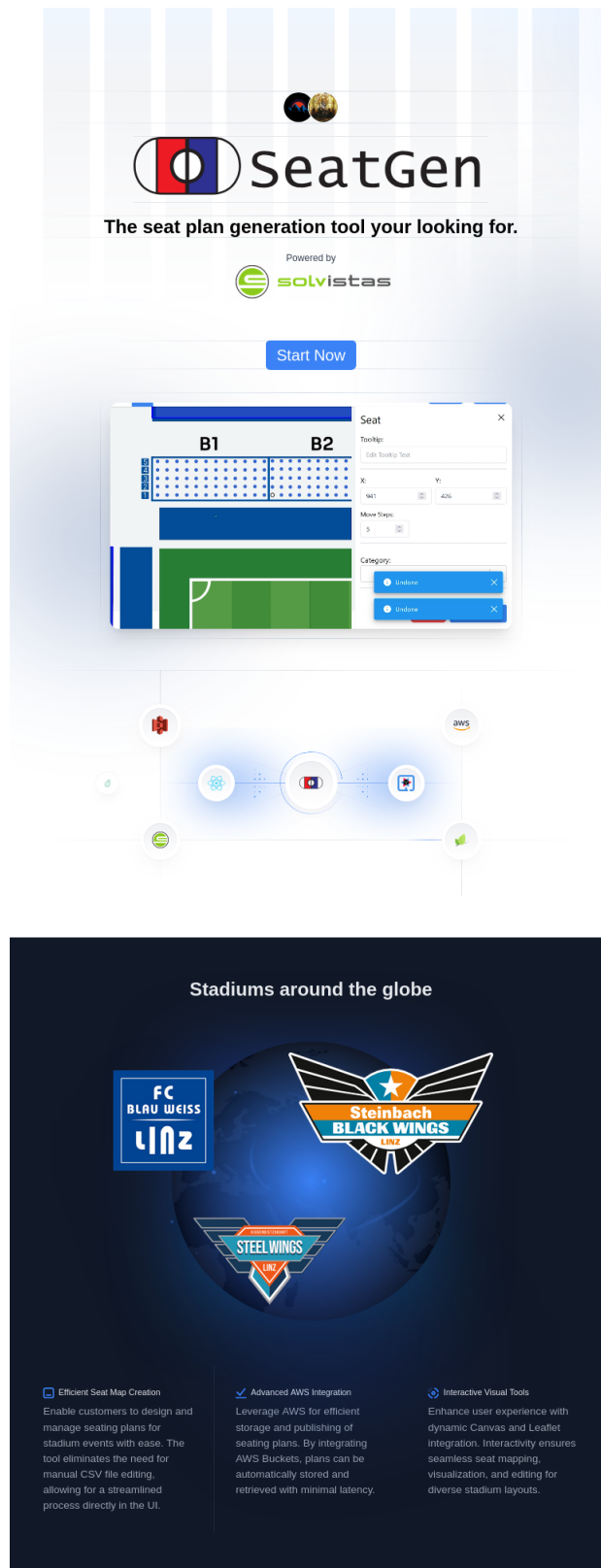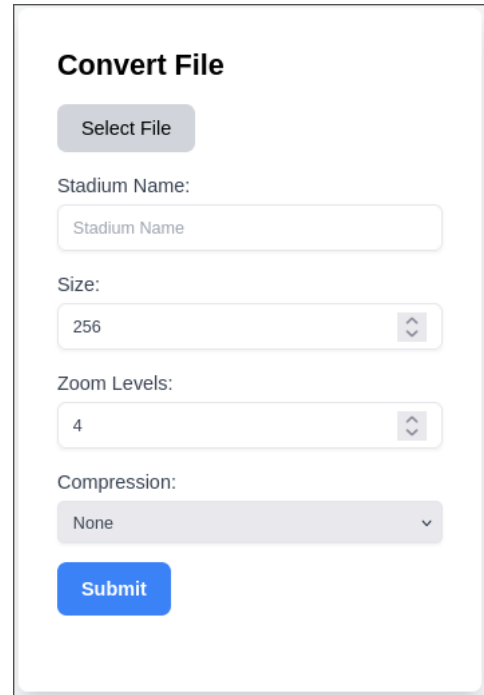
Figure 10: Landing Page

4.4 Map Generation

A significant milestone in the project was the development of the map generation functionality. This feature enables the user to generate an empty seat plan, which relies on the map to serve as the basic visual representation of the venue. The map is interactive in the frontend, and the user can configure several key parameters:

- The venue plan

- The name of the map

- The size of each tile (default is 256x256px for most use cases)

- The number of zoom levels

- The image compression algorithm, which is a dropdown menu with the options: No Compression, Default Algorithm and Zopfli

Figure 11: New Map Mask

These configuration options are accessible under the "New Map" button, as depicted in Figure **??**.

The venue plan is always provided in SVG format, as the application does not support other file formats. To render the map, Leaflet, a JavaScript library designed for interactive maps, is used. A Leaflet map is structured as a 3-dimensional pyramid of tiles, where each tile represents an image. The map's zoom dimension can be considered the "z-axis," while the horizontal and vertical axes correspond to the "x" and "y" axes of the map. Importantly, the x and y axes remain consistent across all zoom levels.

As a result, each tile is defined by a 3-dimensional coordinate in the map. These tiles are retrieved from an S3 bucket and processed by Leaflet in the frontend. The map's structure follows the form of a 3-dimensional pyramid with a square base, progressively expanding as the zoom level increases. The number of tiles per zoom level grows exponentially by a factor of two.

For a given zoom level z, the number of tiles at that level is calculated as:

$$\text{Number of tiles}(z) = 4^{(z-1)}$$

The length of the side of the square base of the pyramid is:

$$\text{Length of side}(z) = 2^{(z-1)}$$

The total number of tiles is given by the sum:

$$S(z) = \sum_{k=1}^{z} 4^{(k-1)} = \frac{4^z - 1}{3}$$

As the number of zoom levels grows, the number of tiles that need to be processed rises exponentially, resulting in a significant increase in the total number of tiles very quickly. For example, there are already 4095 256x256px images with 6 zoom levels.

4.4.1 Step 1: Convert SVG to PNG

The first step in generating this map structure is to convert the SVG file into a PNG image. This process is handled by the backend using the Batik image transcoder. Apache Batik is a robust, pure-Java library developed by the Apache Software Foundation for rendering, generating, and manipulating Scalable Vector Graphics (SVG). Batik provides various tools for tasks such as:

- Rendering and dynamic modification of SVG files

- Transcoding SVG files into raster image formats (as done in this project)

- Transcoding Windows Metafiles to SVG

The size of the image is determined by the current zoom level. The width and height are calculated based on the logic described earlier and implemented in the Kotlin code snippet in Listing **??**.

Listing 36: Image dimensions calculation

```
Dimension(frameSize * 2.0.pow(zoomLevel).toInt(), frameSize *
    2.0.pow(zoomLevel).toInt())
```

If the image is not square, it is centered within a square canvas, with the remaining area filled with white. The resulting image is then converted to PNG format and written to a Java ByteArrayOutputStream, which is used in the subsequent processing step.

4.4.2 Step 2: Slicing the Image into Tiles

In this step, the PNG image generated in the previous step is sliced into smaller tiles. The size of the tiles is determined by the user, with 256x256px being the default. Given that the image is always square and its dimensions are divisible by the tile size, the image can be split into an integer number of tiles without complications.

The slicing process works by iterating through the image and extracting a sub-image of the specified tile size. This is done by calculating the appropriate coordinates for each tile and using the `Graphics.drawImage` method to copy the respective portion of the image into a new `BufferedImage` for each tile.

Here is the Kotlin code implementation for the slicing process:

Listing 37: Image Slicing Implementation

```
1  val subImage = BufferedImage(sliceSize, sliceSize, BufferedImage.TYPE_INT_ARGB)
2  val graphics = subImage.createGraphics()
3  graphics.drawImage(image, 0, 0, sliceSize, sliceSize, x * sliceSize, y *
       sliceSize, (x + 1) * sliceSize, (y + 1) * sliceSize, null)
4  graphics.dispose()
```

In this code, `sliceSize` represents the size of each individual tile (e.g., 256x256px), and `x` and `y` are the coordinates of the current tile. The image is drawn on the `subImage` BufferedImage, which is a sub-region of the original image.

The resulting sub-images are saved as individual PNG files, each representing one tile of the map at the specified zoom level. These tiles are then uploaded to the S3 bucket so that the frontend can fetch them as needed. By splitting the image into tiles, it is possible to load and display the map interactively, only fetching the tiles that are currently in view. This tiling strategy is essential for efficient handling of large map layers at the later zoom levels.

LZ77 algorithm is a dictionary-based compression technique that replaces repeated occurrences of data with references to previous instances, reducing redundancy. Huffman coding, on the other hand, assigns shorter binary codes to more frequently occurring byte sequences, further optimizing storage efficiency. Together, these methods enable PNG files to achieve significant compression while maintaining full image fidelity.

4.4.3 Step 3: Compression

To optimize AWS costs and improve image loading speed in the frontend of the Ticketing project, images are compressed before being uploaded to the S3 bucket. However, this

presents a challenge, as Solvistas requires PNG format for their project. Unlike lossy formats such as JPEG, which achieve smaller file sizes by discarding some image data, PNG is a lossless format, meaning it retains all original data. While this ensures sharp and clear images, it also results in larger file sizes, which can be problematic when numerous images are loaded from AWS in a web environment.

During the map generation process, users can choose from the following compression algorithms:

- **None** – No compression applied (fastest processing time, largest file size).
- **Default** – Standard compression using Deflate (balanced efficiency).
- **Zopfli** – Advanced, high-efficiency compression (better compression rates, slower processing).

The None option results in a 0% compression rate, making it the fastest but least efficient choice.

For the other two compression options, the Pngtastic library is utilized. Pngtastic is a lightweight, pure Java library with no dependencies. It provides a simple API for PNG manipulation, supporting both file size optimization and PNG image layering.

The Default option, uses the Deflate algorithm, which is used for as a base for many lossless compression algorithms, which combines LZ77 and Huffman coding.

- **LZ77** is a dictionary-based compression method that reduces redundancy by replacing repeated sequences of data with references to earlier occurrences, thus minimizing file size without loss of quality.
- **Huffman coding** optimizes storage efficiency by assigning shorter binary codes to frequently occurring byte sequences, further improving compression rates.

Together, these methods enable PNG files to achieve significant compression while maintaining full image fidelity.

The Zopfli algorithm[**?**], developed by Google engineers Lode Vandevenne and Jyrki Alakuijala in 2013, offers an advanced, high-efficiency compression technique. While it still utilizes the Deflate algorithm, it applies exhaustive entropy modeling and shortest path search techniques to achieve a higher compression ratio than standard Deflate and zlib[**?**] implementations. Zopfli achieves superior data compression by extensively analyzing different possible representations of the input data and selecting the most efficient encoding. By default, Zopfli performs 15 iterations to refine compression,
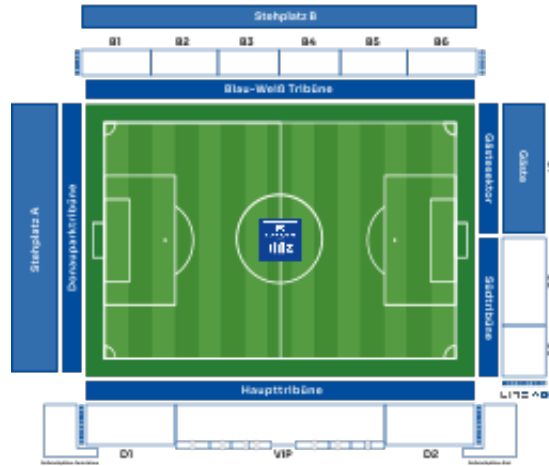
Figure 12: BW-Linz Stadium

though this can be adjusted for higher or lower processing times. Under standard settings, Zopfli output is typically 3–8% smaller than zlib's maximum compression, but it is approximately 80 times slower due to its computational intensity.

According to Google developers: [**?**]

> ...we believe that Zopfli represents the state of the art in Deflate-compatible compression.

While Zopfli is significantly slower than standard Deflate or zlib, but this isn't a huge problem for this usecase, because time can be sacrificed once for the optimization and speed improvement for the user.

Analysis of Compression Algorithms

By testing the compression algorithms within this application with different parameters, which range from the three algorithms, different maps, and different numbers of zoom levels, it can be observed that Zopfli is the best option for the compression of the images, provided that time is allocated for compressing the data. All the test results are visualized in the following table, and the image used for testing is the BW-Linz stadium **??**, which is an example taken from production.

Zoom Level	Size Before Compression (Bytes)	Size After Compression (Bytes)	Time Taken (ms)	Percent Saved
0	12148	12148	98	0%
1	28218	28218	243	0%
2	61540	61540	652	0%
3	131470	131470	1424	0%
4	289606	289606	2815	0%
5	700382	700382	7275	0%

Table 1: Compression Results for NONE Compression Method

Zoom Level	Size Before Compression (Bytes)	Size After Compression (Bytes)	Time Taken (ms)	Percent Saved
0	12148	10043	533	17.47%
1	28218	24710	1477	12.43%
2	61540	58638	4534	4.99%
3	131470	131226	14397	0.19%
4	289606	289134	51388	0.16%
5	700382	699310	180054	0.15%

Table 2: Conversion Results for DEFAULT Compression Method

Zoom Level	Size Before Compression (Bytes)	Size After Compression (Bytes)	Time Taken (ms)	Percent Saved
0	12148	9745	13179	19.73%
1	28218	23343	35591	17.29%
2	61540	54519	111915	11.41%
3	131470	119405	363794	9.18%
4	289606	266976	1166758	7.82%
5	700382	658512	3004721	5.98%

Table 3: Conversion Results for ZOPFLI Compression Method

Compression Method	NONE	DEFAULT	ZOPFLI
Total Size Before Compression (Bytes)	1223364	1223364	1223364
Total Size After Compression (Bytes)	1223364	1213061	1132500
Total Bytes Saved (Bytes)	0	10303	90964
Total Percent Saved	**0%**	**0.84%**	**7.44%**
Total Time Taken (ms)	7519	195929	4696028
Total Time Taken (min)	0.13	3.27	78.27

Table 4: Summary of Conversion Results

To ensure that this test data is viable, the calculation has been computed with 4 dedicated processors, that were configured like this as java vm options:

`-XX:ActiveProcessorCount=4`

As observed in the summary table **??**, Zopfli has amazing compression but is a very time-intensive process, as expected. In the end, it saves 6.6% more than the default algorithm, but it takes approximately 24 times longer, and waiting times of over an hour should be anticipated when using the algorithm. This trade-off must be considered. Although this would only be a one-time process, the decision could favor Zopfli; however, in comparison to the total 90,964 bytes (88.83 KiB) saved, this is still not a significant amount of data saved when considering the time taken. Ultimately, all the data produced is not very large, so the operator must decide whether the time is worth the saved data. If the time and resources are not desired to be spent for such a low storage and performance improvement, the default algorithm remains a good choice, as a waiting period of 3 minutes and 16 seconds is still acceptable for a small optimization. As mentioned previously, it is crucial that the user can decide which algorithm should be used due to various factors. When the program is hosted on an external cloud provider with dynamic cost calculations, the user may not want to incur the extra costs associated with the Zopfli algorithm, as the expense for the additional time could exceed the savings from storage. If the program is executed locally or on servers with sufficient free resources, the Zopfli algorithm is an excellent choice.

Another significant decision during development was whether to compress the images before or after the slicing process. Ultimately, the decision was made to compress the images before slicing them into tiles. This approach was favored for several reasons.

Compressing the entire image as a whole is generally more efficient than compressing individual tiles. Compression algorithms benefit from analyzing the entire dataset, allowing them to identify and eliminate redundancies more effectively. When an image is compressed in its entirety, the algorithm can exploit correlations and patterns that might not be as apparent when processing smaller segments. This leads to a better overall compression ratio, resulting in reduced file sizes without sacrificing quality.

Had the choice been made to first slice the images and then compress the individual tiles in parallel, potential issues with resource contention would have arisen. In such a scenario, multiple instances of the Zopfli compression algorithm could run simultaneously, each consuming considerable CPU cycles and memory. Given Zopfli's high computational demands, this could overwhelm the heap space, leading to memory exhaustion or, at the very least, severely impacting overall system performance. In extreme cases, excessive resource usage could degrade the performance of the entire operating system, causing bottlenecks and slowdowns.

4.4.4 Step 4: Uploading Tiles to S3

The final step in the map generation process involves uploading the generated tiles to an Amazon S3 bucket. This is achieved using the AWS SDK for Java, which provides a robust and efficient way to interact with AWS services. The SDK allows creating an S3 client, which facilitates seamless communication with the S3 bucket. The only required configuration parameters for the client are the AWS region (set to eu-central-1 in SeatGen's case), the access key, and the secret key. Once configured, as demonstrated in Listing **??**, the S3Client instance provides a range of operations, including putObject, getObject, and listObjectsV2, among others.

Listing 38: Configuring the S3 Client

```
1  @ConfigurationProperties(prefix = "aws")
2  data class S3Config @ConstructorBinding constructor(
3      val awsRegion: String,
4      val accessKey: String,
5      val secretKey: String
6  ){
7      @Bean(destroyMethod = "close")
8      fun s3Client() : S3Client {
9          return S3Client
10             .builder()
11             .overrideConfiguration(ClientOverrideConfiguration.builder()
12                 .apiCallTimeout(Duration.ofSeconds(10)).build()
13             )
14             .region(Region.regions()
15                 .find { region -> region.toString() == awsRegion }
16             )
17             .credentialsProvider(
18                 StaticCredentialsProvider.create(
19                     AwsBasicCredentials.create(accessKey, secretKey)
20                 )
```

```
21                    )
22                    .build()
23   }
```

The configuration is managed using the @ConfigurationProperties(prefix = "aws") anno-
tation, which enables automatic injection of required properties. These values—defined
in the primary constructor with @ConstructorBinding—are retrieved from an external
properties file under the aws prefix. This approach ensures that configuration values
remain externalized rather than hardcoded, making it easier to switch between envi-
ronments such as development, testing, and production. The relevant configuration in
application.yml is illustrated in Listing **??**.

Listing 39: AWS Configuration in application.yml

```
1   aws:
2       awsRegion: ${AWS_REGION:eu-central-1}
3       access-key: ${AWS_ACCESS_KEY}
4       secret-key: ${AWS_SECRET_KEY}
```

By using environment variables for sensitive credentials, security is enhanced while main-
taining flexibility in deployment configurations.The SDK's S3Client.builder() method is
used to instantiate and configure the client with the required credentials and region
settings. Because the client is defined as a Spring bean, it can be easily injected into any
class requiring interaction with the S3 bucket. This is a key advantage in Spring-based
applications, as it promotes modularity and maintainability. Unlike in Quarkus, where
dependency injection is handled differently, Spring allows defining such functions as
beans and seamlessly injecting them where necessary.

The name provided by the user doesn't have any major restrictions for special characters,
that's because the customer shouldn't be bothered with technical restrictions. They
should be able to choose the name they want. Technically there are still some restrictions.
The name provided by the user will be used in two situations, that have limitations.

1. Directory names in the S3 buckets
2. Path in the URL in the frontend for editor page

S3 looks like a standard file system, but actually it's not, and therefore the name for
the "directory" doesn't have huge limitations.

Normally data in an Amazon S3 bucket, is stored in a flat structure instead of a
hierarchy one as seen in standard file systems. Amazon still supports the organization
of data like in file systems. This is done by giving all the grouped objects a shared

string prefix. The prefix is therefore the folder name. The data, is actually still stored in a flat structure, but it's visualized like folders in the Amazon S3 console. [**?**]

The second place where the name is utilized is the URL path in the frontend. This is more restricted because it is part of the URL, and therefore some characters could lead to errors. The following characters are reserved and cannot be used in the URL path: `/&?=:%`. Using these characters leads to errors. For this reason, the name is prepared for the URL path by replacing these characters with an underscore. To still provide the user with the requested name, the original name is stored in the database alongside the prepared name. The original name is used solely for display purposes in the frontend.

Then the tiles are uploaded with the prefixes according to their coordinates. The final filenames are in the format `<mapName>/<z>/<x>/<y>.png`. The map name is the name provided by the user, and the zoom level, x, and y are the coordinates of the tile. The tiles are uploaded to the S3 bucket in a hierarchical structure, with each zoom level containing a set of directories for the x and y coordinates.

After executing all of these steps, they have to be repeated for each zoom level asked for.

4.4.5 Optimizations & Memory

This process involves repetitive and computationally demanding tasks such as image slicing, format conversion, and compression, making it well-suited for multi-threading. However, parallel execution introduces challenges, particularly with Java heap space management. During slicing and compression, a large amount of data is stored in memory at the same time. This includes both the upscaled source image and the processed image tiles, leading to high memory usage. At higher zoom levels, storage requirements can reach several gigabytes, potentially exceeding the allocated heap space and causing OutOfMemoryError exceptions.

To address this, the Java heap size can be manually adjusted using:

```
java -Xmx6g seatgen
```

This increases the maximum heap allocation to `6 GB`, allowing for more memory-intensive operations. On `32-bit` systems, the heap size should not exceed 2 GB, as Java will reject larger values and fail with an invalid memory allocation error.

While manually increasing the heap size is a possible solution, it was important to ensure that the application runs efficiently without requiring users to adjust memory settings, although this is recommended when planning to use the Zopfli algorithm. To achieve this, the number of parallel threads was limited to prevent excessive memory usage, and a cap was placed on the maximum zoom level. At higher zoom levels, the processing demands grow exponentially, making it impractical to handle them within a Java-based backend. If future requirements necessitate even higher zoom levels, a more efficient approach could involve using a language like C, Rust, or Python, which offer better memory management for such intensive operations. However, since the company currently does not require zoom levels beyond level 6, this remains an optimization for future development.

To further optimize performance and manage concurrency effectively, Kotlin coroutines were utilized. Coroutines provide a lightweight and efficient way to handle asynchronous programming, allowing tasks like image slicing, compression, and uploading to be performed in a non-blocking manner. Unlike traditional threads, coroutines are more memory-efficient and can be launched in large numbers without overwhelming the system.

For example, during the slicing and compression phases, coroutines were used to parallelize tasks such as processing individual rows of tiles. This approach maximized CPU utilization while keeping memory usage under control. By structuring the workflow with coroutines, it was possible to ensure that tasks like garbage collection and memory cleanup could be triggered at appropriate intervals, preventing memory leaks and excessive heap usage.

The limitation logic for the number of threads for parallelization is based on the algorithm used for compression, because for Zopfli, this is the most memory-critical part. When using Zopfli, which is particularly memory-intensive, parallelism is limited to a single thread during the compression phase. This approach ensures that the system's memory is not overwhelmed, allowing for more efficient processing. However, the slicing of image rows can still be executed with a maximum of four coroutines running concurrently, striking a balance between performance and resource usage. For the DEFAULT compression algorithm, a more aggressive approach is adopted, utilizing half of the available threads. This strikes a balance between efficient processing and maintaining manageable memory consumption. In scenarios where no compression methods are applied, the full use of the available processor threads is permitted. By

limiting the number of concurrent coroutines when using Zopfli or the default algorithm, the risk of exceeding heap space during high-demand processes like compression and slicing is mitigated.

After each complete calculated zoom level, a garbage collection is triggered to free up memory that is no longer needed. This is done by calling the `System.gc()` method, which is a hint to the JVM to run the garbage collector. While this is not a guarantee that the garbage collector will run, it provides a good hint for the JVM to do so.

Because of memory problems, it was also decided not to upload all the data of the images when everything is finished. Instead, the data for each row of tiles is uploaded as soon as it is completed. This approach provides a good balance between memory usage and efficiency, as storing all the data in memory while waiting for other rows to be computed is avoided. Uploading the tiles in the form of rows is also more efficient than uploading every tile individually to the S3 bucket, as it significantly reduces the overhead. For example, on zoom level 6, only 63 requests need to be made instead of 1365 requests.

4.5 AWS - S3

As already mentioned in the technology section, integrating AWS services into the project required a reliable way to interact with AWS APIs. The AWS SDK provides language-specific libraries that simplify communication with AWS services, including S3.

For administrators and developers, AWS provides multiple ways to interact with its services, each suited for different use cases:

- **AWS Management Console (Web UI)** – A user-friendly graphical interface for managing AWS services, ideal for beginners or when making quick changes.
- **AWS Command Line Interface (CLI)** – A powerful command-line tool that allows users to manage AWS resources via scripts and commands, enabling automation and repeatability.
- **AWS SDKs – Language-specific libraries** (such as those for Python, Java, and JavaScript) that facilitate programmatic interaction with AWS services, making integration into applications seamless.
- Others can be found in the AWS documentation.

For configuring the S3 Pod, the AWS CLI tool was utilized instead of the AWS Management Console due to its greater efficiency and the ability to save previously used queries in a text format. While the CLI is less beginner-friendly than the Web UI, it offers significantly more powerful functionalities. When configuring an S3 Pod with the AWS CLI tool for the first time, thoroughly reviewing AWS documentation is recommended to understand the underlying concepts, as numerous configuration options impact the security of the application. Cost considerations were not a concern in this project, as the IAM user provided by the company lacked permissions to modify billing-related settings. However, certain cost factors remain dependent on the development process rather than the configuration of the bucket itself.

To set up the CLI tool, the for the developers operating system appropriate installation method has to be chosen. The specifics are well documented on the AWS's documentation page. For the initial configuration of the CLI tool, the command `aws configure` has to be executed. This command will prompt the user to enter the access key, secret key, region, and output format. The access key and secret key can be obtained from the AWS Management Console. The region is the geographical location where the S3 bucket will be created, in this case it is `eu-central-1`, and the output format can be set to JSON, text, or table. The configuration is stored in two files located under linux in `~ /.aws/` directory. In this directory lie the `config` and `credentials` files. The `config` file contains the region and the output format, while the `credentials` file contains the access key and the secret key. The configuration can be changed at any time by executing the `aws configure` command again. These files can contain multiple profiles, for multiple developers. This is very useful when working in a team, or when working on multiple projects. The profile can be specified by adding the `--profile` flag to the `aws` command.

Other than for testing and managing bucket configuration, the AWS CLI was a very useful tool during development, because it allows the developer to manipulate the data in the buckets manually, as well as reading and listing the data with additional statistics.

Some of the useful utility commands used during the development process are listed in listing **??**

Listing 40: Usefull AWS CLI Commands

```
1  # Lists all buckets
2  aws s3 ls --profile myprofile
3
```

```
 4   # Command to recursively delete every item inside a directory
 5   aws s3 rm --profile --recursive s3://ticketing-stadium-creator-dev/my-bucket/
 6
 7   # List all the data inside directory and provide statistics, like file size,
         object count and tota filesize
 8   s3 ls --profile solvistas --summarize --human-readable --recursive
         s3://ticketing-stadium-creator-dev/my-bucket
 9
10   # Count the number of objects in a bucket (similar but more compact results than
         in the previous command)
11   aws s3 ls --profile solvistas --recursive s3://ticketing-stadium-creator-dev/ | wc
         -l
12
13   # Listing all the applied bucket policies
14   aws s3api get-bucket-policy --profile solvistas --bucket
         ticketing-stadium-creator-dev
```

4.5.1 S3 Bucket Configuration

The configuration of an S3 bucket is a crucial step in setting up an S3 pod, as it determines access permissions, storage classes, and other settings that influence the bucket's behavior. Several key configuration options were implemented in this project:

To enable access to the S3 bucket for all users, the aws s3api put-bucket-policy command was utilized. This command applies a bucket policy that defines specific permissions for the bucket. In this case, public read access to the objects in the bucket was required. The following command was executed: **??** with the `bucket-policy.json` configuration **??**.

Listing 41: AWS CLI command to set a bucket policy

```
1   aws s3api put-bucket-policy --bucket ticketing-stadium-creator-dev --policy
        file://bucket-policy.json
```

Listing 42: Bucket policy JSON configuration

```
 1   {
 2       "Version": "2012-10-17",
 3       "Statement": [
 4           {
 5           "Effect": "Allow",
 6           "Principal": "*",
 7           "Action": "s3:GetObject",
 8           "Resource": "arn:aws:s3:::ticketing-stadium-creator-dev/*"
 9           }
10       ]
11   }
```

This policy allows any user (`Principal:  "*"`) to perform the `s3:GetObject` action on all objects (`Resource:  "arn:aws:s3:::ticketing-stadium-creator-dev/*"`) within the bucket.

Amazon S3 provides **Access Control Lists (ACLs)** to manage access to buckets and objects. ACLs are a legacy access control mechanism, but they are still useful for simple use cases. Here are some important ACLs:

- **Private**: The bucket and objects are accessible only by the bucket owner. This is the default ACL for new buckets.

- **Public Read**: The bucket and objects are readable by anyone on the internet. This is useful for hosting static websites or publicly accessible files.

- **Public Read-Write**: The bucket and objects are readable and writable by anyone on the internet. This is generally not recommended due to security risks.

- **Authenticated Read**: The bucket and objects are readable by any authenticated AWS user (not just the bucket owner).

While ACLs are easy to use, they are less flexible than bucket policies or IAM policies. For more granular control, it is recommended to use bucket policies or IAM policies. IAM (Identity and Access Management) policies provide fine-grained access control to AWS resources. Unlike bucket policies, which are attached to the bucket, IAM policies are attached to IAM users, groups, or roles. For this use case bucket policies were sufficient, because the bucket was only used to store static files, which should be accessible by everyone.

4.6 Add-Tool

The **Add-Tool** in SeatGen provides the ability to manually place individual seats on the stadium map. It is primarily used for precise editing and complements bulk placement features such as the Grid-Tool.

Key Responsibilities:

- Enables manual placement of seats at a specific position on the map.

- Internally dispatches an `AddSeatAction` to update global state.

- Supports undo and redo via the action system.

4.6.1 Triggering the Action

The seat is created externally and passed to the `AddSeatAction` class, which is then executed via the `doAction` method from context.

Listing 43: Calling AddSeatAction

```
1   context.doAction(new AddSeatAction(newSeat, context.setSeats));
```

Explanation:

- The `newSeat` is constructed outside and passed to the action.

- The action handles the insertion of the seat into global state.

- This mechanism ensures consistent integration with undo/redo logic.

4.6.2 AddSeatAction Implementation

The actual seat insertion logic is encapsulated in the `AddSeatAction` class.

Listing 44: AddSeatAction Implementation

```
export default class AddSeatAction extends Action {
    private readonly newSeat: Seat;
    private readonly setSeats: React.Dispatch<React.SetStateAction<Seat[]>>;

    constructor(newSeat: Seat, setSeats:
        React.Dispatch<React.SetStateAction<Seat[]>>) {
        super();
        this.newSeat = newSeat;
        this.setSeats = setSeats;
    }

    execute(): void {
        this.setSeats(prev => [...prev, this.newSeat]);
    }

    undo(): void {
        this.setSeats(prev => prev.filter(seat => seat.id !== this.newSeat.id));
    }
}
```

How It Works:

- `execute()` appends the new seat to the seat list.

- `undo()` removes it again by filtering the ID.

- This implementation ensures full undo/redo compatibility.

4.6.3 Conclusion

The **Add-Tool** provides a foundational editing feature in SeatGen, allowing users to place single seats precisely. It ties into the system's action-based architecture and contributes to a robust and reversible seat editing experience.

4.7 Multiselect-Tool

A very important feature was enabling the user to select multiple seats at once. This was especially crucial because a venue can have many seats, and selecting them one by one
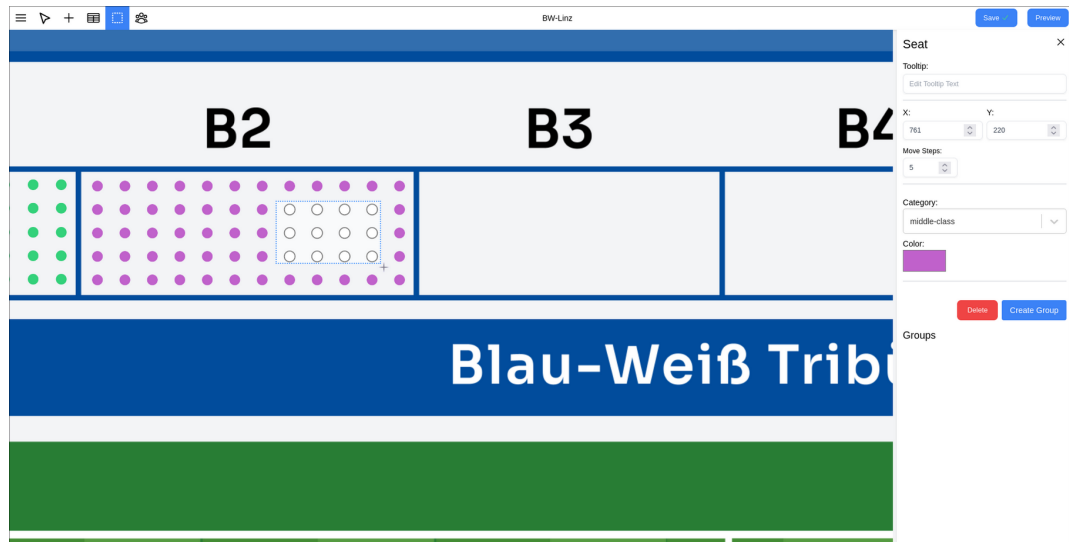
Figure 13: Multiselect Tool

would be very time-consuming. A common operation for the user is also moving entire sectors. To tackle this challenge, the decision was made to develop the multiselect tool. The multiselect tool draws a rectangle when selected and dragged on the map, selecting everything inside this rectangle. For the rectangle part, Leaflet already provides a feature that uses a rectangle, which works on the user's drag interaction. This feature is called BoxZoom. The BoxZoom feature is a built-in capability of Leaflet that allows the user to draw a rectangle on the map and zoom into the area of the rectangle. This feature served as a good starting point for the multiselect tool because it already provides rectangle drawing and drag interaction. The BoxZoom feature was extended to develop the multiselect tool. The multiselect tool is a subclass of the BoxZoom feature and overrides the functions responsible for zooming. Instead of zooming, the multiselect tool selects all the seats inside the rectangle. The multiselect tool is shown in listing **??**. The finished functionality for selecting seats is shown in figure **??**.

Listing 45: Multiselect Tool

```
1   const MapBoxSelect = (props) => {
2       const propsRef = useRef(props);
3
4       useEffect(() => {
5           propsRef.current = props;
6       }, [props]);
7
8       let _startPoint;
9       let _currFunction;
10
11      function checkIfNothingSelected() {
12          const currentProps = propsRef.current;
13          return (currentProps.currentSelectedTool === undefined ||
                  currentProps.currentSelectedTool === null ||
                  !(currentProps.currentSelectedTool.id in currentProps.handleDrag));
14      }
15
16      L.Map.Multiselect = L.Map.BoxZoom.extend({
17          _onMouseDown: function (e) {
18              if (checkIfNothingSelected()) {
19                  return false;
```

```
20                  }
21                  _currFunction =
                        propsRef.current.handleDrag[propsRef.current.currentSelectedTool.id]
22                  L.DomUtil.disableTextSelection();
23                  propsRef.current.mapRef.current?.dragging.disable()
24
25                  _startPoint = this._map.mouseEventToLayerPoint(e);
26
27                  this._box = L.DomUtil.create('div', 'leaflet-zoom-box', this._pane);
28                  L.DomUtil.setPosition(this._box, this._startLayerPoint);
29
30                  this._container.style.cursor = propsRef.current.currentSelectedTool;
31
32                  L.DomEvent
33                      .on(document, 'mousemove', this._onMouseMove, this)
34                      .on(document, 'mouseup', this._onMouseUp, this)
35                      .on(document, 'keydown', this._onKeyDown, this)
36                      .preventDefault(e);
37
38                  this._map.fire('boxzoomstart');
39              },
40
41          _onMouseMove: function (e) {
42                  if (checkIfNothingSelected()) {
43                      return false;
44                  }
45                  var startPoint = _startPoint,
46                      box = this._box,
47
48                      layerPoint = this._map.mouseEventToLayerPoint(e),
49                      offset = layerPoint.subtract(startPoint),
50
51                      newPos = new L.Point(
52                          Math.min(layerPoint.x, startPoint.x),
53                          Math.min(layerPoint.y, startPoint.y));
54
55                  L.DomUtil.setPosition(box, newPos);
56
57                  box.style.width = (Math.max(0, Math.abs(offset.x) - 4)) + 'px';
58                  box.style.height = (Math.max(0, Math.abs(offset.y) - 4)) + 'px';
59              },
60
61          _onMouseUp: function (e) {
62                  if (checkIfNothingSelected()) {
63                      return false;
64                  }
65
66                  this._finish();
67                  const map = this._map,
68                      layerPoint = map.mouseEventToLayerPoint(e);
69                  const bounds = new L.LatLngBounds(
70                      map.layerPointToLatLng(layerPoint),
71                      map.layerPointToLatLng(_startPoint)
72                  )
73
74                  if (_currFunction != null) {
75                      _currFunction(bounds);
76                  }
77                  _currFunction = null
78              },
79
80          _finish: function () {
81                  propsRef.current.mapRef.current?.dragging.enable()
82                  if (Array.from(this._pane.children).includes(this._box)) {
83                      this._pane.removeChild(this._box);
84                  }
85                  this._container.style.cursor = '';
86
87                  L.DomUtil.enableTextSelection();
88
89                  L.DomEvent
90                      .off(document, 'mousemove', this._onMouseMove)
91                      .off(document, 'mouseup', this._onMouseUp)
92                      .off(document, 'keydown', this._onKeyDown);
93              },
94
95      })
96
97      L.Map.mergeOptions({boxPrinter: true});
98      L.Map.addInitHook('addHandler', 'boxPrinter', L.Map.Multiselect);
99
100     L.Map.mergeOptions({boxZoom: false});
```

```
101   }
102
103   export default MapBoxSelect
```

This code uses parts of the original code concepts, and adapts it for the selecting of seats. The original source can be found in Leaflet's source code.

This box is utilized not only for the multi select tool, but also for the grid tool which is explained in more detail in section **??**. The modification of the ZoomBox only need to be done once, because they can be reused by both tools. The modified functionality just disables the zoom of the original feature, and accepts a function that is called onMouseUp with the boundaries of the drawn rectangle as parameters. When the rectangle select is needed it can be dynamically loaded into the map component.

Except for this tool another way of selecting multiple seats at once was implemented, because of usability reasons, and the expectancy of the user. In editors ranging from Photoshop, Gimp, to File Explorers, the holding of the `strg` or `cmd` key while clicking on an object, allows the user to select multiple objects. This was also implemented in SeatGen. The user can hold this key and click on a seat, to select more than one seat.

4.8 Grid-Tool

4.9 Grid-Tool

The **Grid-Tool** allows the user to generate a rectangular grid of seats based on a user-defined area on the map. It is ideal for efficiently placing large blocks of seats in structured stadium sections.

Key Responsibilities:

- Lets users draw a bounding box to define the grid area.

- Opens a modal dialog to configure the number of rows, columns, and padding.

- Internally dispatches an `AddMultipleSeatAction` to add the generated seats.

4.9.1 User Interaction

Once the `Grid-Tool` is selected, users can draw a selection box on the map. This triggers the following method:

Listing 46: Handling Box Selection

```
1  const handleAddSeatGridBox = useCallback((bounds: L.LatLngBounds) => {
2      setBounds(bounds);
3      setOpenModal(true);
4  }, []);
```

Explanation:

- Stores the selected bounds.

- Opens the modal to allow the user to input grid parameters.

4.9.2 Generating the Seat Grid

When the user confirms the modal, the tool computes and inserts a seat grid into the map:

Listing 47: Add Grid Logic

```
1  const addSeatGrid = useCallback((startPoint: L.LatLng, rows: number, columns:
       number, latStep: number, lngStep: number) => {
2      const newSeats = [];
3      for (let row = 0; row < rows; row++) {
4          for (let col = 0; col < columns; col++) {
5              const lat = startPoint.lat - row * latStep;
6              const lng = startPoint.lng + col * lngStep;
7              const position = L.latLng(lat, lng);
8
9              const newSeat: Seat = {
10                 id: -1,
11                 position: {lat: position.lat, lng: position.lng},
12                 draggable: false,
13                 tooltipText: `Row ${row + 1} Seat ${col + 1}`,
14                 selected: false,
15                 category: null
16             };
17             newSeats.push(newSeat);
18         }
19     }
20     context.doAction(new AddMultipleSeatAction(newSeats, props.setSeats))
21  }, [props]);
```

How It Works:

- The bounds are converted into a grid layout using latitude and longitude step sizes.

- Margin (padding) is applied to space the grid evenly.

- All seats are bundled into an `AddMultipleSeatAction` for batch insertion and undo support.

4.9.3 Modal Interface

The modal lets the user configure rows, columns, and padding interactively:

Listing 48: Modal Input Fields

```
1  <input type="number" value={rows} onChange={(e) =>
       setRows(Number(e.target.value))} />
2  <input type="number" value={cols} onChange={(e) =>
       setCols(Number(e.target.value))} />
3  <input type="range" min={"0"} max={"100"} value={padding} onChange={(e) =>
       setPadding(Number(e.target.value))} />
```
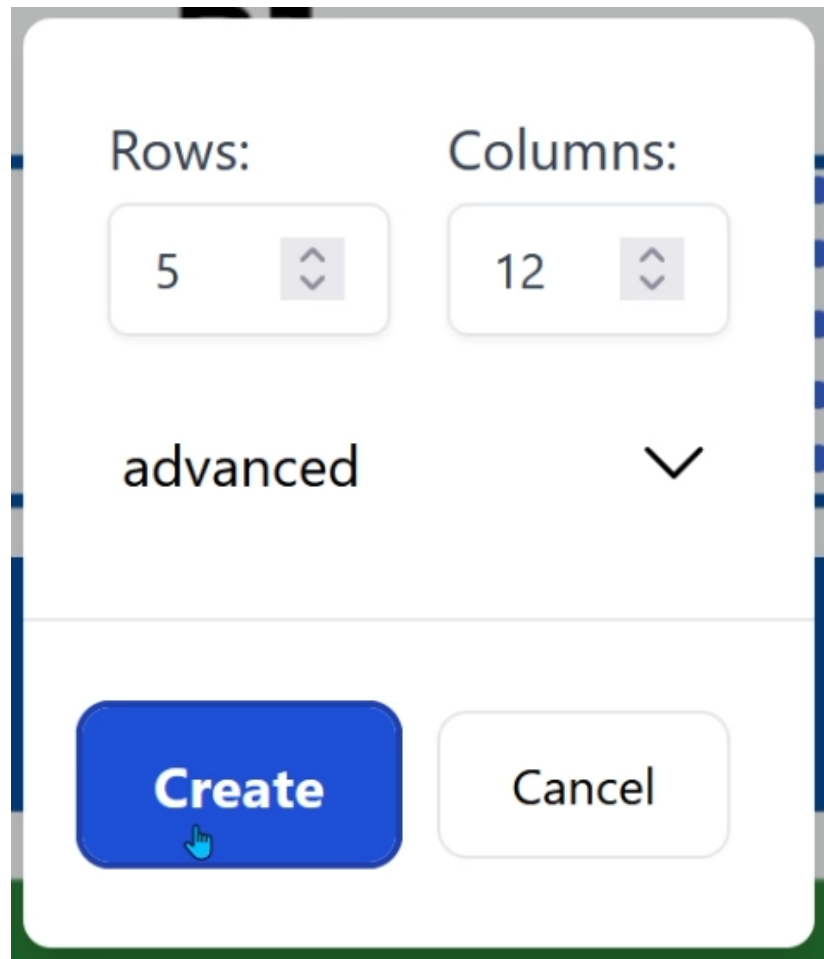

Figure 14: Toolbar with Grid-Tool Selected and Configuration Modal Open

4.9.4 Conclusion

The **Grid-Tool** is designed for speed and precision when placing large quantities of seats. Its modal-based configuration, combined with interactive box selection and undoable batch insertion, makes it a powerful tool for large-scale seat placement in structured layouts.

4.10 Standing-Area-Tool

4.10.1 Frontend

The **Standing-Area-Tool** enables users to define polygonal sectors directly on the map by selecting a series of points. These sectors, referred to as standing areas, are used to represent sections of a stadium that are not associated with fixed seating.

Key Responsibilities:

- Allows users to draw a polygon on the map by clicking multiple points.

- Once finalized, dispatches an `AddStandingAreaAction` to insert the area into global state.

- Automatically assigns an ID, empty name, and zero capacity by default.

- Supports full undo and redo through the action system.

Capturing the Area

Users activate the tool and click on the map to add polygon points:

Listing 49: Adding Polygon Points

```
1  const handleMapClick = (event: L.LeafletMouseEvent) => {
2      setPolygon([...polygon, event.latlng]);
3  };
```

Each click appends a new vertex to the polygon. The current polygon is stored in local state and displayed as an in-progress shape.

Finalizing the Standing Area

Once the polygon is complete, the user confirms by clicking a button that triggers the following logic:

Listing 50: Finalizing Standing Area

```
1  const handleDone = () => {
2      if (polygon.length < 3) return;
3
4      const newArea: StandingArea = {
5          id: Date.now(),
6          name: "",
7          capacity: 0,
8          coordinates: polygon,
9          selected: false
10     };
11
12     context.doAction(new AddStandingAreaAction(newArea, context.setStandingAreas));
13     setPolygon([]);
14 };
```

Explanation:

- Ensures at least three points are present to form a valid polygon.

- Constructs a new `StandingArea` object with default values.

- Dispatches an `AddStandingAreaAction` to insert the new area into state.

- Resets the drawing state for the next operation.

Integration with the Action System

The tool leverages the action-based architecture of SeatGen to ensure consistency and undoability.

Listing 51: AddStandingAreaAction

```
1  export default class AddStandingAreaAction extends Action {
2      constructor(private standingArea: StandingArea, private setStandingAreas:
           React.Dispatch<React.SetStateAction<StandingArea[]>>) {
3          super();
4      }
5
6      execute(): void {
7          this.setStandingAreas(prev => [...prev, this.standingArea]);
8      }
9
10     undo(): void {
11         this.setStandingAreas(prev => prev.filter(a => a.id !==
               this.standingArea.id));
12     }
13 }
```

Highlights:

- `execute()` appends the new area to the list.

- `undo()` removes the inserted area by filtering it out by ID.

Summary

The **Standing-Area-Tool** provides an intuitive polygon-drawing interface for adding sectors that do not rely on individual seats. Through real-time polygon tracking and integration with the action system, it supports flexible and reversible sector creation tailored to the needs of modern stadium planning.

4.10.2 Backend

Because a standing area is in the for of nodes, which are connected to each other it can be viewed as a circular linked list. These have to be saved in the database. The
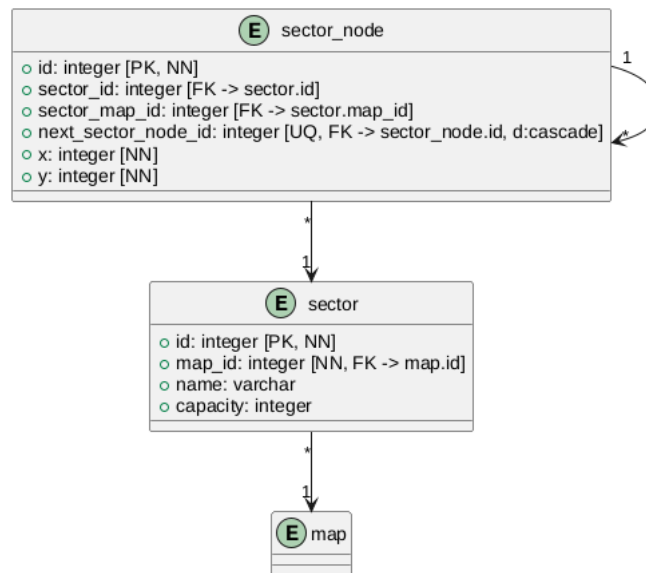
Figure 15: Standing Area Database Model

database model is shown in figure **??**. All the other tables are left out for simplicity's sake.

The sector_node entity in the database has a foreign key, that references itself. This allows modeling a hierarchy, and in this case this hierarchy looks like a loop. To ensure database integrity, a unique constraint is set on the next_sector_node_id. This makes sure, that there is only one sector node, that is the successor of a given sector node. When fetching a standing area two options were considered to do so.

1. Fetch all sector nodes from a map and build the standing area in the backend.
2. Make a recursive query in the database, that fetches all sector nodes of a standing area.

The advantage of the second approach is, that only the needed nodes are fetched, and the logic is closer to the db. The big problem is, that SQL queries in Postgres don't have a built-in way to do recursion like oracle db has with its `connect by` clause. The solution for Postgres would have to be implemented with PostgreSQL specific keywords. A query that fetches all the items in a loop would look as seen in listing **??**.

Listing 52: Recursive Query

```
1  WITH RECURSIVE cte AS (
2      -- select first node with level 1
3      SELECT , 1 AS level
4      FROM   sector_node
5      WHERE  (select id from sector_node where map_id = 1 and sector_id = 1 LIMIT 1)
           = id
6
7      UNION  ALL
8      SELECT  sn., c.level + 1 as level
9      FROM cte c
```

```
10              JOIN sector_node sn ON  c.next_sector_node_id = sn.id where (select id
                    from sector_node where map_id = 1 and sector_id = 1 LIMIT 1) != sn.id
11  )
12  SELECT id, x, y
13  FROM    cte
14  ORDER   BY level;
```

Here the `WITH RECURSIVE` clause is used to define a recursive query. The query selects
the first node and then iterates through the linked list by repeatedly joining the
sector_node table with itself using the next_sector_node_id field. This process
continues until all nodes in the loop have been retrieved. The big disadvantage of this
approach is, that it uses a PostgreSQL specific keyword, this makes it not possible to
After executing all of these steps, they have to be repeated for each zoom level asked
seamlessly switch to another database. This is why the first approach was chosen. The
standing areas of the map are fetched in the backend and ordered recursively in a way,
that represents the loop, like in listing **??**.

Listing 53: Standing Area Backend

```
1  override fun getSectorNodeBySector (sector: Sector): Set<SectorNode> {
2      val startNode = db.getFirstSectorNodeBySector (sector.id!!)
3      return startNode.collectSectorNodes (startNode)
4  }
5
6  private fun SectorNode.collectSectorNodes (startNode: SectorNode): Set<SectorNode> {
7      return setOf(this) + (
8              if (next == startNode) emptySet ()
9              else next?.collectSectorNodes (startNode) ?: emptySet ()
10          )
11  }
```

4.11 Saving

SeatGen uses a save button to save the currently edited state and changes in the map.
This has some advantages and disadvantages, against saving every time an action occurs.
Some advantages are:

Advantages:

- The user can undo changes without saving them
- The user can save the changes when they are done, and try out stuff without instantly
  saving it, and overwriting the last state
- There is less load on the server, because multiple changes are saved at once
- A lot of load is avoided when doing a lot of small changes with lots of objects
- Only the important changes are saved, and not every step in between
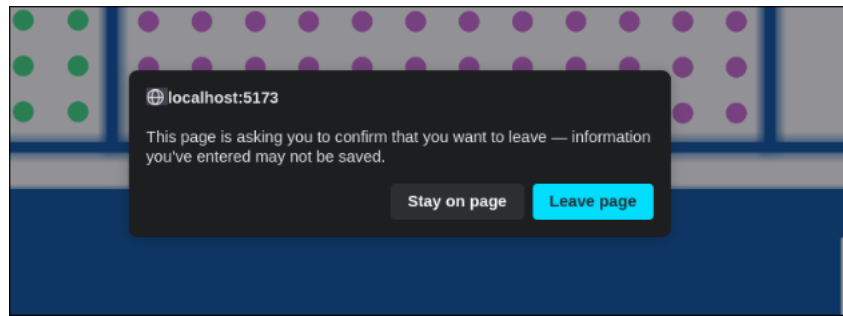
Disadvantages:

Figure 16: Leave Confirmation Dialog

Figure 17: Save Button States

- If the user forgets to save, all changes are lost

- There is an extra step the user has to execute

To mitigate these disadvantages, SeatGen uses a some techniques, like when the user is trying to leave the page via, reloading or closing the tab, the browser will ask the user if they are sure they want to leave the page, because there are unsaved changes. This is done by using the `beforeunload` event, which is triggered when the user tries to leave the page. This event is used to show a confirmation dialog to the user, which asks if they are sure they want to leave the page **??**.

To ensure that the user doesn't forget to save, the save button is always visible, and always when the user has made changes, the save button is highlighted **??**. This is done by the controller that manages the undo and redo functionality that is explained in section **??**. The code **??** checks if there have been any actions by the user that are not the currently saved index, or if there are any standing areas that have to be saved or deleted. The standing areas have been managed differently for technical reasons.

Listing 54: Check for Unsaved Changes

```
1  function checkForUnsavedChanges() {
2      setHasUnsavedChanges(
3          historyIndex !== savedIndex ||
4          standingAreasToSave.length != 0 ||
5          standingAreasToDelete.length != 0
6      )
7  }
```

When saving the changes, first a checksum that is the hash of all seats and their properties is calculated and compared to the checksum of the initial loading of the map. If the checksums are the same, no seats will be saved, because they didn't change. If

the checksum is different, a snapshot of the current seats is sent to the backend, for saving. For the standing areas there are two variables. `standingAreasToSave` and `standinAreasToDelete`. These are set relative to the last saved state. When saving, the toSave get saved in the backend, and the to delete get deleted. After the save is successive both variables are empty. When creating a new standing area or deleting an existing one the variables get filled with the respective standing area. A big advantage over saving every time an action occurs is, that when creating and deleting an area without saving in between, there are no unsaved changes, and no traffic has to be sent instead of saving sending a request on its creation and deletion.

4.12 Design-Patterns

During the development of the application, various design patterns were incorporated to efficiently address specific challenges. Different sectors of the application required distinct patterns, particularly in the interactive editor and tool functionalities. Given the complexity of these components, tailored solutions were necessary. For instance, implementing an undo functionality—a widely expected usability feature in modern editors, both text-based and visual—required careful design considerations. This feature is commonly applied across software products, with multiple solutions available. Specialized implementations of design patterns were employed in the following aspects of the application:

- Undo/Redo functionality of Actions
- Tool System
- Backend services

4.12.1 Undo/Redo

The undo/redo mechanism is essential for usability, providing users with the flexibility to revert and reapply actions efficiently. Multiple approaches exist for implementing this feature:

1. **State Snapshot Approach**: This method involves saving the entire application state at each change and reverting to the previous state when undoing. While simple to implement, this approach is inefficient due to excessive memory consumption and redundant data storage. An advantage is, that old states can easily be restored

without any additional logic and calculations. This makes it not very prone to bugs and errors.

2. **Differential State Storage**: Instead of storing complete states, this approach records only the differences between successive states, similar to version control systems such as Git. While more efficient, this method becomes complex as the number and types of objects increase (in this case it would be Seats and Standing-Areas).

3. **Command Pattern**: Actions are encapsulated as objects that implement a common interface, containing methods for execution and reversal. This approach allows flexible and scalable undo/redo functionality, making it ideal for complex interactive applications. It also allows to execute additional business logic when undoing an action, like deleting additional data that was created by the action, or sending requests to a backend. This makes it an excellent choice when the states are distributed.

4. **Memento Pattern**: This pattern captures and externalizes an object's internal state so that it can be restored later without violating encapsulation. While useful for preserving an object's complete state, it can be memory-intensive when storing multiple versions.

Given the application's complexity, a variation of the Command Pattern has been implemented for the undo/redo functionality. This approach ensures scalability, efficiency, and maintainability while minimizing redundancy of data and code.

The implementation defines an abstract `Action` class that all actions must implement. An overview of this is seen in **??**. This class enforces the inclusion of `execute` and `undo` methods, ensuring a standardized approach to action management.

Listing 55: Action class

```
1  export abstract class Action {
2      execute: (() => void) | undefined;
3      undo: (() => void) | undefined;
4  }
```

For both of these properties a function is expected that can be called to execute the action or to `undo` the `action`. This is flexible approach and can be used for a lot of different actions.

Another importance for this application was to allow the execution of business logic while undoing specific actions, like sending requests to the backend. This is very easy to impliment because each Action has its individual `execute` and `undo` function.
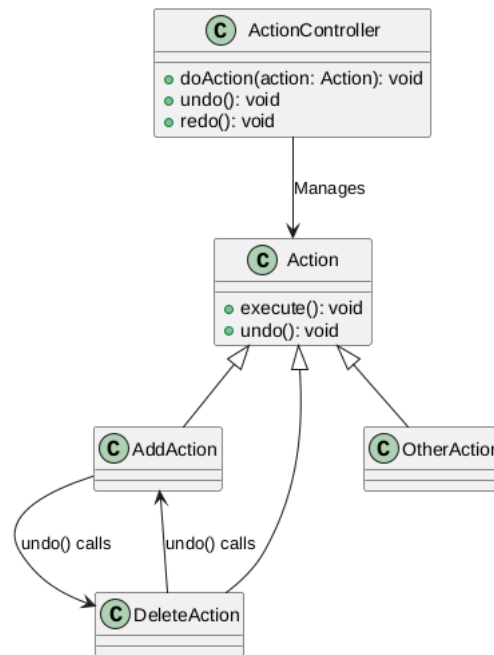
Figure 18: Command Pattern in SeatGen

While this seems like a lot more logic is needed than in the other design patterns, the logic demanded by this is actually important for usability reasons, and lots of it is reusable. Because the undo function should be able to be executed manually by the user, calling the `undo()` shouldn't be the only way to reverse an action, for example when creating a new seat, you should be able to delete it again by calling the `undo()` function as well as a separate way like a delete button. So the developer should always provide both ways for usability reasons. This approach incentivizes developers to do it because it's mandatory to implement the logic for undoing and action anyway.

Here is an implementation of an action that creates a Standing-Area with its counterpart action that deletes the standing area. Both actions can reference each other for the undoing part to reduce code redundancy, because the opposite of creating a standing area is deleting it. That's why deleting it is the undo action of creating it and the other way around.

Listing 56: Add standing-area action implementation

```
1  export class AddStandingAreaAction implements Action {
2      private readonly _newArea: StandingArea;
3      private readonly _context: MapContextValue;
4
5      constructor(newArea: StandingArea, context: MapContextValue) {
6          this._newArea = newArea;
7          this._context = context;
8      }
9
10     execute = () => {
11         this._context.setStandingAreas((prevAreas) =>
12             [...prevAreas, this._newArea]
13         );
14         if (this._context.standingAreasToDelete.includes(this._newArea.id))
```

```
15            {
16                this._context.setStandingAreasToDelete((prev) =>
17                    prev.filter(id => id !== this._newArea.id)
18                );
19            } else {
20                this._context.setStandingAreasToSave((prev) => {
21                    const updated = [...prev, this._newArea.id];
22                    return updated;
23                });
24            }
25        };
26
27        undo = () => {
28            new DeleteStandingAreaAction([this._newArea], this._context).execute()
29        };
30    }
```

Listing 57: Delete standing-area action implementation

```
1   export class DeleteStandingAreaAction implements Action {
2       private readonly _deletedAreas: StandingArea[];
3       private _undoAreas: StandingArea[] | undefined;
4       private readonly _context: MapContextValue;
5
6       constructor(deletedAreas: StandingArea[], context: MapContextValue) {
7           this._deletedAreas = deletedAreas;
8           this._context = context;
9       }
10
11      execute = () => {
12          const deletedAreaIds = this._deletedAreas.map(area => area.id);
13          this._context.setStandingAreas((prevAreas) =>
14              prevAreas.filter((area) => !deletedAreaIds.includes(area.id))
15          );
16          this._context.setStandingAreasToDelete((prev) =>
17              [...prev, ...deletedAreaIds]
18          );
19          deletedAreaIds.forEach(id => {
20              if (this._context.standingAreasToSave.includes(id))
21              {
22                  this._context.setStandingAreasToSave(this._context
23                  .standingAreasToSave
24                  .filter(savedId => savedId !== id));
25              } else
26              {
27                  this._context.setStandingAreasToDelete((prev) => [...prev, id]);
28              }
29          });
30      }
31
32      undo = () => {
33          if (this._undoAreas) {
34              this._deletedAreas.forEach((area) => {
35                  new AddStandingAreaAction(area, this._context).execute()
36              })
37          }
38      }
39  }
```

Int the code in Listing **??** and **??** you have the functions with the business logic for creating and deleting a standing area. When the `undo()` function is called, actually a new `DeleteStandingAreaAction` is created, and it's execute is called, because it implements the correct business logic for undoing the action. Same is true for the call of the call of the `undo()` function in the `DeleteStandingAreaAction`. With this code both functionalities can be implemented by separate buttons or something similar, and the undo and redo functionality is implemented as well. The needed contexts and functions to execute the business logic correctly for both classes can be defined individually in the constructor of the classes. The class also has to store the information

to undo its actions, for example the move action has to store the old position of the object to be able to move it back to the old position.

The actual undoing logic is defined by a controller. When an action should be undo and redoable it has to be passed to the controller. The controller manages the function and can be called to undo or redo the last action. It also manages the stack of actions, so that all the actions can be undone and then redone again, until a new action is executed. When this happens the controller ignores all of the "future" actions that would have come after the current action. For example: Action1, Action2, Action3 and Action4 have been executed. The latest action saved by the Controller is currently Action4. Currently, all the actions can be undone and then redone in a stack like way. This means Action4 is undone, then Action3, then Action2 and so on. Then they can all be reapplied in the same revered order. When actions have been undone, to Action2 for example, and then a new Action5 is executed, Action3 and Action4 will be scrapped, because a new "future" has been created. This is a common behavior in undo and redo functionalities.

The implementation of the controller is as shown in listing **??**.

Listing 58: Action controller implementation

```
1  const loopSize = 50;
2  const actionHistory: (Action | undefined)[] = new Array(loopSize).fill(undefined);
3
4  let historyIndex = 0;
5  let maxIndex = 0;
6  let minIndex = -1;
7  let savedIndex = 0
8
9  const loopSize = 50;
10 const actionHistory: (Action | undefined)[] = new Array(loopSize).fill(undefined);
11
12 let historyIndex = 0;
13 let maxIndex = 0;
14 let minIndex = -1;
15 let savedIndex = 0
16
17
18 const doAction = (action: Action) => {
19     historyIndex = increase(historyIndex);
20
21     while (maxIndex != historyIndex) {
22         actionHistory[maxIndex] = undefined
23         maxIndex = decrease(maxIndex)
24     }
25
26     actionHistory[minIndex] = undefined
27     minIndex = increase(maxIndex);
28     actionHistory[historyIndex] = action;
29     action.execute!();
30     checkForUnsavedChanges()
31 };
32
33 const updateSaveIndex = () => {
34     savedIndex = historyIndex
35     checkForUnsavedChanges()
36 }
37
38 function increase(num: number): number {
39     return num !== loopSize - 1 ? num + 1 : 0;
40 }
```

```
41
42   function decrease(num: number): number {
43       return num !== 0 ? num - 1 : loopSize - 1;
44   }
45
46   const undo = () => {
47       if (historyIndex !== minIndex && actionHistory[historyIndex] !== undefined) {
48           actionHistory[historyIndex]!.undo!()
49           historyIndex = decrease(historyIndex);
50           checkForUnsavedChanges()
51           enqueueSnackbar("Undone", {variant: "info"})
52       }
53   };
54
55   const redo = () => {
56       if (historyIndex !== maxIndex && actionHistory[increase(historyIndex)] !==
            undefined) {
57           historyIndex = increase(historyIndex);
58           actionHistory[historyIndex]!.execute!();
59           checkForUnsavedChanges()
60       }
61   };
62
63   const getCurrentAction = (): Action | null => actionHistory[historyIndex] ?? null;
```

To register a new Action in the controller, the `doAction` function has to be called with the action as a parameter like in this listing **??**. The context referenced here is the context containing business logic for the map as well as containing the logic for the undo and redo controller.

Listing 59: Registering a new action in the controller

```
1   context.doAction(new AddSeatAction(context.setSeats, lat, lng))
```

This controller stores all the actions in the form of a loop, with the size defined by the `loopSize` variable. The variable is set to 50 because more than 50 undoable actions back are not necessary. A circular buffer for storing this kind of data is very advantageous because when the buffer is full, the oldest action is overwritten by the newest action because the oldest data is not needed anymore. Other very important variables are the `minIndex` and `maxIndex` variables. They define the range of the actions that can be undone and redone. When undoing, it's checked that the current index which is represented by the `historyIndex` is not the same as the `minIndex` and there is also an undoable action, because then there would be no more actions to undo. Only if these conditions are fulfilled, there are actions to undo, and the `undo()` this is called, and the `historyIndex` is decreased. A similar logic is applied for the `redo()` function, but with the `maxIndex` and the `increase()` function. The `doAction()` function is used to handle new actions. It increases the `historyIndex` and sets the `maxIndex` to the `historyIndex` and overwriting all the no longer needed Actions with undefined. The `minIndex` is set to the `increase(maxIndex)` to ensure that when the loop is full, that the changes that are too old they are removed. At last the action is added to the list of actions and the `execute()` function is called.

The `increase()` and `decrease()` functions are used to increase and decrease the index in a circular way. This is necessary because the buffer is circular and when the end of the buffer is reached, the index has to be set to the beginning of the buffer again. This is done by checking if the index is at the end of the buffer and then setting it to the beginning of the buffer again.

4.12.2 Tools

Tools played a significant role during development, making it essential to ensure that the implementation of new tools was as easy and fast as possible. To achieve this, a Tool interface was designed, where instances only need to be added to an existing array containing the tools. The final version of this interface is presented in Listing. **??**.

Listing 60: Tool interface

```
export interface Tool {
    id: string
    icon: ReactNode,
    onSelect?: ()=>void,
    hotkey?: string
}
```

The attributes of the interface are the following:

- **id**: The id, which is used to identify the tool. Normally this is a string which is the name.
- **icon**: The icon of the tool, which is displayed in the toolbar. This has the type `ReactNode` because first a simple string was used, to pass it to an icon component, but then it was decided to use the `ReactNode` type, because it's more flexible, and the icons are not only limited to the icons of one UI library, but any icon can be used, including SVG icons.
- **onSelect**: The function that is called when the tool is selected. This is optional because not every tool needs a function to be called when it's selected. Some tools are handled externally.
- **hotkey**: The hotkey is a string that defines the hotkey for the tool. This is optional because not every tool needs a hotkey.

In listing **??** list of all the implemented Tools and how they look, when the coresponding image **??** of how they are rendered.

Listing 61: Implemented Tools

```
const tools: Tool[] = [
    {
```
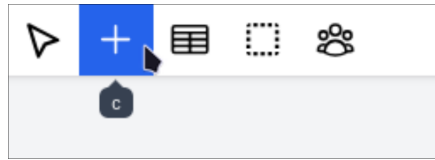
Figure 19: Tools in SeatGen

```
3              id: "mouseTool",
4              icon: <svg viewBox="0 0 24 24" xmlns="http://www.w3.org/2000/svg">
5                  <path
6                      d="..."
7                  />
8              </svg>,
9              onSelect: () => handleToolSelect(() => {
10             }, "mouseTool", "default"),
11             hotkey: "v"
12         },
13         {
14             id: "addTool",
15             icon: <PlusIcon></PlusIcon>,
16             onSelect: () => handleToolSelect((e) => {
17                 props.addSeat(e.latlng.lat, e.latlng.lng)
18             }, "addTool", "cell"),
19             hotkey: "c"
20         },
21         {
22             id: "addGridTool",
23             icon: <TableCellsIcon></TableCellsIcon>,
24             onSelect: () => handleToolSelect(() => {
25             }, "addGridTool", "cell"),
26             hotkey: "g"
27         },
28         {
29             id: "squareSelectTool",
30             icon: <svg xmlns="http://www.w3.org/2000/svg" viewBox="0 -960 960 960">
31                 <path
32                     d="..."/>
33             </svg>,
34             onSelect: () => handleToolSelect(() => console.log("Clicked"),
                   "squareSelectTool", "crosshair"),
35             hotkey: "a"
36         },
37         {
38             id: "standingAreaTool",
39             icon: <UserGroupIcon />,
40             onSelect: () => handleToolSelect(() =>console.log("standingtool"),
                   "standingAreaTool", "crosshair"),
41             hotkey: "s"
42         }
43     ];
```

The `handleToolSelect` function is a function that is called when a tool is selected. It's responsible for setting the current tool and the cursor as shown in listing **??**. The cursors supported are the names of all the cursors names supported by the browser. These can be viewed on the Mozilla developer documentation [**?**].

Listing 62: Handle tool select function

```
1  const handleToolSelect = useCallback(((toolFunction: (e: LeafletMouseEvent) =>
       void, toolId: string, cursorIcon?: string) => {
2      const cursorName =
           L.DomUtil.getClass(props.map.current!.getContainer()).split(" ").find((it)
           => it.startsWith("cursor-"));
3      if (cursorName) {
4          L.DomUtil.removeClass(props.map.current!.getContainer(), cursorName);
5      }
6      if (cursorIcon !== undefined) {
7          L.DomUtil.addClass(props.map.current!.getContainer(),
               `cursor-${cursorIcon}`);
8      }
```

```
 9        setActiveToolFunction(() => toolFunction);
10        context.setSelectedToolId(toolId);
11  }, [props.map, setActiveToolFunction]);
```

After all this is set, the tools are simply rendered with a component, that handles all these properties like hotkeys and the icon.

4.12.3 Backend Services

For the backend services SeatGen uses a very general and exchangeable approach that works with Spring. For the services, an interface is defined, which is then implemented by the respective service classes. This interface is used to define the methods that are needed for the services. In the controller the service is injected, and spring automatically creates an instance of the correct service implementation and injects it into the controller. This is a very common approach in Spring and is used in many projects.

5 Summary

5.1 Michael Ruep

My focus in this project was on the frontend implementation of SeatGen. I was responsible for building the interactive stadium editing experience using React, TypeScript, and Leaflet. This included implementing the core component structure around the `MapComponent.tsx`, managing state updates via `MapContext.tsx`, and also partially integrating undo/redo functionality through a command pattern using actions.

One of the most challenging but rewarding parts was implementing the live multi-seat dragging logic. This required a deep understanding of Leaflet's coordinate system and careful handling of relative positioning during real-time updates. Additionally, I worked on the `DetailEditor.tsx`, which provides a comprehensive interface for editing seats, assigning categories, managing groups, and configuring standing areas. I also did the `ToolBar.tsx` with my team partner, which ties all interactive tools together, and implemented the logic for triggering actions through tool selection.

Throughout the project, I learned how to structure complex, interactive applications in a scalable and modular way. I also improved my ability to collaborate on a shared codebase by ensuring that my parts were reusable and extensible. In terms of workflow, the communication and division of work between my colleague and me was seamless. We were able to work independently on different parts of the project at all time and merge our work together without any major conflicts.

Looking back, one of the most important things I learned is how important it is to fully understand a library like Leaflet before diving into implementation. There were many instances where understanding a subtle detail in the documentation helped save hours of programming. Overall, I'm very happy with the functionality I was able to implement and proud of how our diploma thesis turned out.

5.2 Michael Stenz

For my part I was responsible for most of the technology choices in the backend and some design choices in the frontend. All these choices had a huge impact on the entire development of the application because some featured centered around these libraries and patterns. In my opinion some choices were very good, but some choices probably weren't the most optimal as stated in the thesis, like java vs python for the backend. A big limitation were the technology demands of the Solvistas company, but it's important to also address these challenges and to find solutions that fit the company's needs. I personally learned a lot of new patterns that are used in the industry and I think that this new knowledge can be very useful in future projects. When it comes to splitting the tasks, it was a very smooth process because my partner and I had a very good communication and were able to both work on the tasks and tools that fitted more our expertise and interests. Also, after the minimal implementation of the project had been done, we were able to expand it feature by feature very easily because in the end the project more or less looked like a small framework, with all the patterns that enabled this easy expandability. A big takeaway from this project is that it can be overwhelming at first when working with huge libraries like Leaflet. When dealing with these challenges it's very important to start with a small prototype and read the documentation very carefully because most of the time the solution is hidden somewhere in the documentation, and a lot of time can be saved when reading it instead of trying to figure out the solution by yourself.

The decision to collaborate with a company was both good and bad in different aspects. On the one hand, it was very good because we were able to get a lot of feedback from the company, and we were able to address a real-world problem. On the other hand a part of our task was to understand parts of the Ticketing project which has a huge codebase, and we also had to understand code of the csv-stadium-creator tool and the workflow of the employees that are responsible for the maintenance and creation of the stadium.

List of Figures

List of Tables

List of Listings

Appendix