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 internal tool for the company Solvistas. We cooperated with Solvistas to help them with their organization and management of their product, which sells tickets for sport-events which take place in stadiums. Seatgen allows the members of Solvistas to create and edit stadium plans in a fraction of the time that it used to take. With a selection of our handy tools, the workflow to create an entire seating plan gets very efficient and allows the people to create, move and edit seats, areas and more. The tool is designed to be user-friendly and intuitive so that the people at Solvistas don't have to spend a lot of time learning new software.

Inhaltsverzeichnis

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. We wanted to create a visual editor to create and manage 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.

@Huemer Milestomes ????

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 [1]. 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. Our 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, we put the main focus 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. Our design choices reflect these constraints and requirements in multiple areas.

2.4 Requirements and Challenges

A core requirement of SeatGen is "direct manipulation" [1], 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 we will further describe the technical and logical aspects of how the set requirements are met.

3 Technologies

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 [2, 3]. 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 [4, 3].

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 [4]. This aligns with the findings in [3],

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. [5, 6, 7]

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 us to easily incorporate third-party libraries. For example, we integrated Leaflet 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. We used Reacts Context to manage global state effectively, rather than relying on more complex solutions. 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 allowed us to quickly understand and use React, and we immediately had our 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 [3]. Additionally, React is preferred by developers for SPAs with frequent UI updates, with a reported 34% higher satisfaction rate compared to other frameworks [3].

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 [4, 5].

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 [3], further validated its suitability for our interactive seating plan editor.

3.2 Spring Boot and Kotlin

For the backend logic, we chose Spring Boot 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. Our backend had 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, we initially considered Python due to its well-documented and easy-to-use image manipulation libraries like CairoSVG and OpenCV. However, we ultimately decided 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, it allowed us to keep our backend technology stack consistent. Additionally, using Java/Kotlin ensured we did not need to manage separate runtime environments. 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, with extensive documentation and examples. This made it easy to integrate S3 into our backend for storing and retrieving image tiles efficiently.

As for the language, we used Kotlin in Spring Boot even though it's not used in many of Solvistas' projects. We still decided that Kotlin was the better option because it is a modern language that is fully interoperable with Java and has many features that make it easier to write clean and concise code, thus reducing errors, improving readability, and maintainability. It eliminates much of the boilerplate code required in Java and provides a rich standard library with many built-in utility functions, significantly reducing development time. Kotlin has no essential functionalities that Java couldn't provide, but 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 allow for 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 proces-

sing multiple image transformations simultaneously which reduces our image processing time by a factor a lot.

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, makes it a forward-thinking investment despite its lower adoption within Solvistas' existing projects.

In the end, we chose Spring Boot with Kotlin because of our team's expertise with the language and the fact that all other components of the Ticketing software were already written in Spring Boot.

3.3 Database

We chose PostgreSQL as our database for several key reasons. First, we required a relational database since our data follows a structured design that is best represented through classical relational models. Additionally, using a relational database simplifies the process of exporting generated data into the Ticketing database, which also adheres to a relational structure.

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

For database connectivity in our Spring Boot application, we utilized the Spring Data JPA library. This library streamlines the process of connecting to a database and executing queries while implementing the repository pattern. Through this pattern, we define custom queries 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 our codebase.

To manage database migrations efficiently, we adopted the Flyway library. Flyway enables us to define database changes through SQL scripts that execute automatically when the application starts. This ensures our database schema remains consistent with the latest changes, significantly 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, we can execute both Data Definition Language (DDL) and Data Manipulation Language (DML) commands. 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, we evaluated both Liquibase and Flyway. While both are open-source and provide seamless integration with Spring Boot and other Java frameworks, we ultimately opted for Flyway due to its simplicity and our specific use case. Since we are a small team with infrequent parallel database changes, Flyway's linear migration approach suits our workflow without introducing complications. Although this approach might present challenges in larger teams with concurrent database modifications, it remains a practical choice for our 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 our 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 1. Flyway's approach, which relies on plain SQL migration files, makes it more readable and easier to maintain.

Listing 1: Liquibase example changelog

```
1      --liquibase formatted sql
2
3      --changeset nvoxland:1
4      create table test1 (
5          id int primary key,
6          name varchar(255)
7      );
8      --rollback drop table test1;
9
10     --changeset nvoxland:2
11     insert into test1 (id, name) values (1, 'name 1');
12     insert into test1 (id,  name) values (2, 'name 2');
13
```

```
14        --changeset nvoxland:3 dbms:oracle
15        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 our image data, we evaluated PostgreSQL's built-in options, including BLOBs (Binary Large Objects) and TOAST (The Oversized-Attribute Storage Technique). While these mechanisms allow PostgreSQL to handle large binary files, we ultimately decided 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

Amazon S3 (Simple Storage Service) is a scalable object storage service provided by Amazon Web Services (AWS). It allows users to store and retrieve large amounts of data in the cloud, with a focus on high availability, durability, and security. S3 is widely used for storing static assets such as images, videos, documents, and backups, and it is designed to provide low-latency access to data from anywhere in the world and in our

case the image tiles of the seatplan. With features such as data encryption, versioning, and lifecycle policies, S3 offers a flexible and cost-effective solution for managing large datasets. S3 also provides an extensive API that allows developers to interact with their storage buckets programmatically. In this application we access the API via the AWS SDK for Java which is provided and maintained by Amazon.

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 [8]. 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 Why We Chose 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 [8].

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 gives us great 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 [8]. Furthermore, we adapted existing Leaflet functions for tasks such as seat selection and the grid tool. This significantly accelerated the development process

By selecting Leaflet, we ensured that SeatGen could efficiently handle 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 allows us to render thousands of seat markers efficiently without significantly impacting performance. Since stadiums can contain a large number of seats, we optimized rendering using Leaflet layers to manage visibility at different zoom levels.

- **Custom Zoom Levels and Scaling:** Leaflet enables us to define custom zoom levels. This ensures that users can zoom in for precise seat selection or zoom out to get a full view of the stadium's structure.

- **Interactive Seat Selection:** By leveraging Leaflet's built-in event handling system, we allow users to click and modify seats in real time. This is crucial as it enables us to intuitively adjust 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, we used React Context 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, we created a seamless user experience 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 [9]. This greatly simplified the integration process, as we could manage Leaflet elements within React components without direct manipulation of the DOM.

One challenge we faced was handling state management and data binding between Leaflet and React. Since Leaflet operates independently from React's Virtual DOM [9], synchronizing real-time seat selections, modifications, grid placements and so on required a structured approach.

Beyond standard Leaflet functionality, we also extended and customized existing Leaflet features to meet our specific needs. Tools such as the seat selection tool and grid tool, leveraged Leaflet's built-in functions but were adapted and modified to SeatGen's requirements. By understanding and modifying Leaflet's core functions, we were able to create a tailored solution that aligned with our 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 us to create an intuitive and responsive seating visualization tool. Additionally, React Leaflet streamlined the integration process, enabling us to build faster within React [9].

By leveraging Leaflet's existing functionality while extending its core features to suit stadium seat planning, we ensured a clean, efficient tool suite and smooth real-time interaction. This makes Leaflet an essential part of our 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.
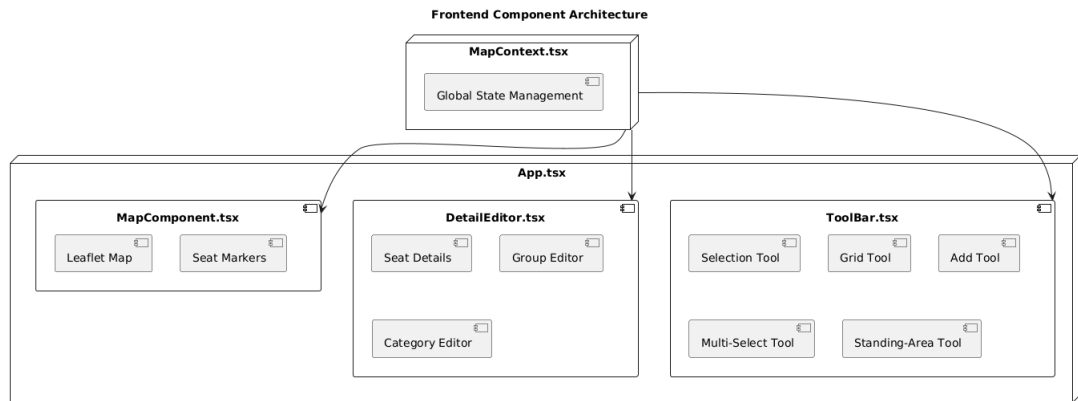

Abbildung 1: Frontend Component Architecture Overview (Rough Overview)

The diagram in Figure 1 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.

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

```
1   const MapComponent: FC<MapProps> = ({ editable: initialEditable }) => {
2       const context = useMapContext(); // Access global state (MapContext.tsx)
3       const { bucketName, mapName } = useParams(); // Retrieve map identifiers from
            URL params
4
5       const [editable, setEditable] = useState(initialEditable ?? true);
6       const mapRef = useRef<L.Map>(null);
7
8       return (
9           <MapContainer
10              crs={L.CRS.Simple} // Uses a flat, pixel-based coordinate system
11              className="leaflet-container h-full w-full"
12              ref={mapRef}
13              center={[context.mapInfo.tileSize / (-2), context.mapInfo.tileSize /
                    (2)]}
14              zoom={context.mapInfo.defaultZoom}
15              maxZoom={context.mapInfo.maxZoom}
16              minZoom={context.mapInfo.minZoom}
17              scrollWheelZoom={true}
18              zoomControl={false}
19              doubleClickZoom={false}
20              preferCanvas={true}
21              dragging={true}
22              tap={false}
23              renderer={L.canvas()} // Use Canvas for better performance
24          >
25              <TileLayer tms={true}
                    url={`${context.mapInfo.mapDto.baseUrl}/{z}/{x}/{y}.png`} />
26          </MapContainer>
27      );
28  };
```

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.

19

- 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

```
useEffect(() => {
    if (bucketName && mapName) {
        // Fetch basic stadium map information
        seatgenApiClient.api.info(bucketName, mapName)
            .then(response => context.setMapInfo(response.data))
            .catch(error => console.error('Error fetching map info:', error));

        // Fetch seat categories before retrieving individual seats
        seatgenApiClient.api.getAllCategories().then((r) => {
            if (r.ok) {
                context.setCategories(r.data);
                seatgenApiClient.api.getSeatsByMap(bucketName,
                    mapName).then(response => {
                    if (r.ok) {
                        context.setSeats(response.data.map((s) => ({
                            id: s.seatId!,
                            position: { lat: s.xcoord!, lng: s.ycoord! },
                            category: r.data.find(it => it.id === s.categoryId) ??
                                null
                        })));
                    }
                });
            }
        });

        // Fetch standing areas
        seatgenApiClient.api.getSectorByMap(bucketName, mapName).then((r) => {
            if (r.ok) {
                context.setStandingAreas(r.data.map((data) => ({
                    id: data.id!,
                    name: data.name!,
                    capacity: data.capacity,
                    coordinates: data.coordinates?.map((c) => new L.LatLng(c.x!,
                        c.y!)) ?? [],
                    selected: false
                })));
            }
        });

    } else {
        console.error("BucketName or MapName not set");
    }
}, [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: 5678,
3      name: "Sektor 1",
4      capacity: 500,
```

```
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, we implemented a real-time position tracking mechanism using Leaflet events and React state updates.

Tracking Initial Positions

Before moving seats, we store their initial positions 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:

- We create a reference (`initialPositionsRef`) 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, we calculate the drag distance and apply it to all selected seats:

Listing 9: Updating Seat Positions During Dragging

```
const updateSelectedSeatsPosition = useCallback((draggedSeatId: number,
    newLatLngPosition: { lat: number; lng: number }) => {
    const map = mapRef.current;
    const primarySeat = context.seats.find(s => s.id === draggedSeatId);

    if (!map || !primarySeat || !primarySeat.selected) return;

    const newPosition = map.latLngToLayerPoint(newLatLngPosition);
    const oldPosition = initialPositionsRef.current[draggedSeatId];

    const deltaX = newPosition.x - oldPosition.x;
    const deltaY = newPosition.y - oldPosition.y;

    context.setSeats(prevSeats =>
        prevSeats.map(seat => {
            if (seat.selected) {
                const initialPosition = initialPositionsRef.current[seat.id];
                const newPosX = initialPosition.x + deltaX;
                const newPosY = initialPosition.y + deltaY;
                const newLatLngPos = map.layerPointToLatLng(new L.Point(newPosX,
                    newPosY));
                return { ...seat, position: newLatLngPos };
            }
            return seat;
        })
    );

    // Update MoveAction for undo tracking
    const currentAction = context.getCurrentAction();
    if (currentAction instanceof MoveAction) {
        currentAction.setNewSeats(selectedSeats);
    }
}, [context.seats, context]);
```

Breakdown of Logic:

- We calculate the drag delta (change in X/Y position) between the starting position and the new cursor position.

- 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.

- We track changes using `MoveAction`, allowing the operation to be undone if needed.

Finalizing the Seat Position After Dragging

When the user releases the dragged seat, we commit the final position to the global state:

Listing 10: Finalizing Seat Position After Dragging

```
1  const finalizeSeatPosition = useCallback(() => {
2      context.setSeats(prevSeats =>
3          prevSeats.map(seat => ({ ...seat, draggable: false }))
4      );
5  }, [context]);
```

How This Works:

- Ensures that seats remain in their new positions after releasing the mouse.

- Prevents further unnecessary state updates.

- Syncs the final positions with the backend when saving changes.

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 [1]).

Dragging Summary

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, we achieved a fluid and high-performance dragging experience whilst also beeing able to undo and redo the movement.

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 11: 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 });
```

Standing Area Selection:

Listing 12: Handling Standing Area Selection

```
1  const handleStandingAreaClick = useCallback((id: number) => {
2      context.setStandingAreas(prevAreas => prevAreas.map(area => ({
3          ...area,
4          selected: area.id === id ? !area.selected : false
5      })));
6  }, [context]);
```

Selection and Editing Process:

- Clicking a standing area toggles its selected state.

- In the Detail Editor panel renaming, capacity adjustments, and category updates are possible.

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

Event Handling in Leaflet

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

Global Click Handling:

Listing 13: 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 14: 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 15: 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 16: 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.

Final Summary of MapComponent.tsx

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

- Renders the Leaflet-based interactive seating map.

- Seat and Standing Area Rendering

- Group Selection and Bulk Editing

- Drag-and-Drop Seat Repositioning

- Event Handling for User Interaction

- Synchronizes data with global state (MapContext.tsx).

- Asynchronous Backend Synchronization

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

4.1.1 React Components and Hooks

4.1.2 MapContext and Global State

4.1.3 Tool System and Event Handling

4.2 Leaflet Integration

4.2.1 Writing Extensions

4.3 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 rate

Abbildung 2: New Map Mask

These configuration options are accessible under the "New Map"button, as depicted in Figure 2.

The venue plan is always provided in SVG format, as the application does not support other file formats. To render the map, we use Leaflet, a JavaScript library designed for interactive maps. 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
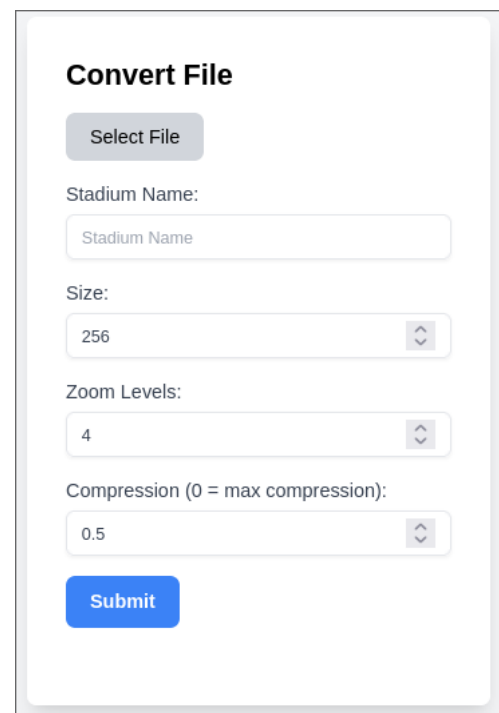
ß-axis,"while the horizontal and vertical axes correspond to the *änd "yäxes of the map. Importantly, the x and y axes remain consistent across all zoom levels.

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 we need to process rises exponentially, and we reach a huge number of tiles very fast. For example, we already have 4095 256x256px images with 6 zoom levels.

4.3.1 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 17.

Listing 17: Image dimensions calculation

```
1   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.

4.3.2 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 18: 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 going to be uploaded to the S3 bucket, so that the fronzend can fetch them as needed.

By splitting the image into tiles, we can 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.

4.3.3 AWS

4.3.4 Image Processing

4.4 Add-Tool

4.5 Multiselect-Tool

4.6 Grid-Tool

4.7 Standing-Area-Tool

4.7.1 Frontend

4.7.2 Backend

4.8 Optimizations

4.9 Design-Patterns

5 Summary

Literaturverzeichnis

[1] J. D. H. Edwin L. Hutchins und D. A. Norman, „Direct Manipulation Interfaces," *Human–Computer Interaction*, Vol. 1, Nr. 4, S. 311–338, 1985. Online verfügbar: https://doi.org/10.1207/s15327051hci0104_2

[2] React Team, *React - A JavaScript Library for Building User Interfaces*, 2025, Accessed: 19/02/2025. Online verfügbar: https://react.dev/

[3] P. S. Emmanni, „Comparative Analysis of Angular, React, and Vue.js in Single Page Application Development," *International Journal of Science and Research (IJSR)*, Vol. 12, 06 2023.

[4] Ibadehin Mojeed, *What is the virtual DOM in React?*, 2022, Accessed: 19/02/2025. Online verfügbar: https://blog.logrocket.com/virtual-dom-react

[5] Hamir Nandaniya, *A Guide to Component-Based Architecture: Features, Benefits and more*, 2024, Accessed: 19/02/2025. Online verfügbar: https://marutitech.com/guide-to-component-based-architecture

[6] Lya Laurent, *React's Component-Based Architecture: A Case Study*, 2023, Accessed: 19/02/2025. Online verfügbar: https://appmaster.io/blog/react-component-based-architecture

[7] Chintan Gor, *Why React's Component-Based Architecture Simplifies Web Application Development?*, 2024, Accessed: 19/02/2025. Online verfügbar: https://www.esparkinfo.com/blog/react-component-based-architecture.html

[8] Leaflet, *Leaflet*, 2025, Accessed: 20/02/2025. Online verfügbar: https://leafletjs.com/

[9] React Leaflet, *React Leaflet Documentation*, 2025, Accessed: 20/02/2025. Online verfügbar: https://react-leaflet.js.org/

Abbildungsverzeichnis

Tabellenverzeichnis

List of Listings

Appendix