HTL Leonding
next level

HÖHERE TECHNISCHE
BUNDESLEHRANSTALT
LEONDING

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

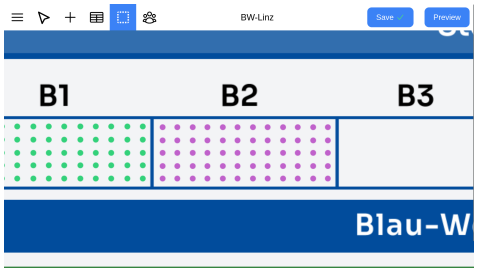
Leonding, April 4, 2025

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 an 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:** Depending on ongoing events and seasons, stadium layouts are frequently restructured, requiring updated seat plans.
- **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 actors 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 AWS. Our design choises underly these constraints and requirements in many places.

2.4 Requirements and Challanges

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 Challange. The following elements are particularly relevant:

- **Immediate Feedback:** When a user modifies a seat position, the update is reflected instantly on the map. Hence dragging seats should happen instant and following the user just like if you would take something in real-life.
- **Simplicity and Learnability:** Due to Familiar mouse-based interactions performing tasks rather specific and complex tasks such as grouping seats, creating standing areas, or categorizing should be possible without any specialized training or introduction to the meaning of coordinates.
- **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 Maps in React were chosen primarily to streamline the development of such dynamic, visually rich interfaces. Additionally for further efficiency improvements quick actions in form of hotkeys are added.

2.5 Summary

In summary, the environment for SeatGen includes a mix of business and technical constraints, demanding a straighforward, userfriendly 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

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 processing 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.

SeatGen also utilizes SwaggerUi and SwaggerUi codegen to generate a 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

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

4 Implementation

4.1 Frontend Architecture

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

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 2

Listing 2: 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 3

Listing 3: Modifying Leaflet Features

```
L.Map.Multiselect = L.Map.BoxZoom.extend({

    _onMouseDown: function (e) {
        //...
        //Business logic for overwriting here
        //...
    },

    _onMouseMove: function (e) {
        //...
        //Business logic for overwriting here
        //...
    },

    _onMouseUp: function (e) {
        //...
        //Business logic for overwriting here
        //...
    },

    _finish: function () {
        //...
        //Business logic for overwriting here
        //...
    },

})

L.Map.mergeOptions({boxPrinter: true});
L.Map.addInitHook('addHandler', 'boxPrinter', L.Map.Multiselect);

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.

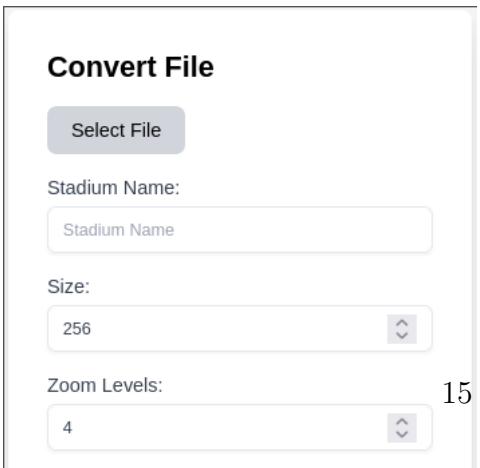
Handling events in Leaflet is straightforward using `useMapEvents`. When using this hook and adding it to the Map, a lot of events can be caught and handled. An example for the usage of this hook with the `click` event is shown in listing 4.

Listing 4: 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 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

These configuration options are accessible under the "New Map"button, as depicted in Figure 1.

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 5.

Listing 5: 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 6: 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.

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.

4.3.3 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, we utilize the Pngtastic library, 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, 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: [2]

> The smaller compressed size allows for better space utilization, faster data transmission, and lower web page load latencies. Furthermore, the smaller compressed size has additional benefits in mobile use, such as lower data transfer fees and reduced battery use.

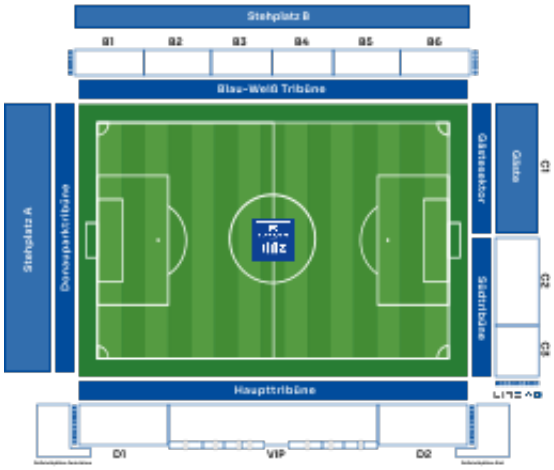


Abbildung 2: BW-Linz Stadium

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, that range from the 3 algorithm, different maps and different number of zoom levels, we can see that Zopfli is the best option for the compression of the images, if the time is willing to be spent on compressing the data. All the test results are visualized in the following table, the image used for testing is the BW-Linz stadium 2, which is an example taken out of 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%

Tabelle 1: Compression Results for NONE Compression Method

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:

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%

Tabelle 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%

Tabelle 3: Conversion Results for ZOPFLI Compression Method

`-XX:ActiveProcessorCount=4`

As you can observe in the summary table 4 and as expected has amazing compression but is a very time intensive process. In the end we save 6.6% more than the default algorithm, but it takes 24 times longer, and waiting times for over an hour should be calculated when using the algorithm. This is a trade-off that has to be considered. Because this would only be a one time process the decision could fall for Zopfli, but in comparison to the total 90964 Bytes (88.83=KiB) saved this is still not a huge amount of data saved, when considering the time taken. In the end all the data produced is not very large, so the operator has to decide if the time is worth the saved data. If the time and resources don't want to be spent for such a low storage and performance improvement, the default algorithm is still a good choice, because a waiting period of 3 minutes 16 seconds is still acceptable for a small bit of optimization. As mentioned previously it's very important, that the user can decide what algorithm should be used because of lots of factors. When the program is hosted on an external cloud provider with dynamic cost calculation, the user probably doesn't want to spend the extra money for the Zopfli algorithm, because the cost for the extra time could be higher than the

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

Tabelle 4: Summary of Conversion Results

saved money for the storage. If the program is executed locally or servers that have enough free resources the Zopfli algorithm is an excelent choice.

Another significant decision during development was whether to compress the images before or after the slicing process. Ultimately, we decided 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 we chosen to first slice the images and then compress the individual tiles in parallel, we would have faced potential issues with resource contention. 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.

4.3.4 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 us to create

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 our case), the access key, and the secret key. Once configured, as demonstrated in Listing 7, the S3Client instance provides a range of operations, including putObject, getObject, and listObjectsV2, among others.

Listing 7: Configuring the S3 Client

```
@ConfigurationProperties(prefix = "aws")
data class S3Config @ConstructorBinding constructor(
    val awsRegion: String,
    val accessKey: String,
    val secretKey: String
){
    @Bean(destroyMethod = "close")
    fun s3Client() : S3Client {
        return S3Client
            .builder()
            .overrideConfiguration(ClientOverrideConfiguration.builder()
                .apiCallTimeout(Duration.ofSeconds(10)).build()
            )
            .region(Region.regions()
                .find { region -> region.toString() == awsRegion }
            )
            .credentialsProvider(
                StaticCredentialsProvider.create(
                    AwsBasicCredentials.create(accessKey, secretKey)
                )
            )
            .build()
}
```

The configuration is managed using the @ConfigurationProperties(prefix = “aws”) annotation, 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 environments such as development, testing, and production. The relevant configuration in application.yml is illustrated in Listing 8.

Listing 8: AWS Configuration in application.yml

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

By using environment variables for sensitive credentials, we enhance security while maintaining 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 our URL 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 name is utilized, is the URL path in the frontend. This is a bit more restricted, because it's part of the URL and therefore some characters could lead to errors. These following characters are reserved and can't be used in the URL path: `/&?=:%`. When these are used this leads to errors. This is why we prepare the name for the URL path by replacing these characters with an underscore. To still give the user the requested name, we store the original name in our database as well as the prepared name. The original name is only used for displaying 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.

4.3.5 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, we wanted to ensure that the application runs efficiently without requiring users to adjust memory settings, although it's recommended when planning to use the Zopfli algorithm. To achieve this, we limited the number of parallel threads to prevent excessive memory usage and placed a cap 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. Leveraging Kotlin Coroutines for Concurrency

To further optimize the performance and manage concurrency effectively, we utilized Kotlin coroutines. Coroutines provide a lightweight and efficient way to handle asynchronous programming, allowing us to perform tasks like image slicing, compression, and uploading 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, we used coroutines to parallelize tasks such as processing individual rows of tiles. This allowed us to maximize CPU

utilization while keeping memory usage under control. By structuring the workflow with coroutines, we were able 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, we limit the parallelism to a single thread during the compression phase. This approach ensures that we do not overwhelm the system's memory, 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, we adopt a more aggressive approach, 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, we allow for the full use of the available processor threads. By limiting the number of concurrent coroutines when using Zopfli or the default algorithm, we mitigate the risk of exceeding heap space during high-demand processes like compression and slicing.

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

4.4 AWS - S3

As already mentioned in the technology section, integrating AWS services into our 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, we used the AWS CLI tool. We chose this option over the AWS Management Console, because it is more efficient and already used queries can be saved in a text format. It's not as beginner-friendly as the Web UI, but it is way more powerful. When trying out to configure an S3 pod with the AWS CLI tool for the first time, it's recommended to read a lot of documentation to understand the concepts of AWS first, instead of just trying out commands, because there are lots of options that affect the security of the application. Worrying about cost was not necessary in this project, because the by the company provided I AM user didn't even have the rights to mess with the billing critical settings. Although a part of the cost is still dependent on our development process, but not 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 9

Listing 9: Usefull AWS CLI Commands

```
# Lists all buckets
aws s3 ls --profile myprofile

# Command to recursively delete every item inside a directory
aws s3 rm --profile --recursive s3://ticketing-stadium-creator-dev/my-bucket/

# List all the data inside directory and provide statistics, like file size,
    object count and tota filesize
s3 ls --profile solvistas --summarize --human-readable --recursive
    s3://ticketing-stadium-creator-dev/my-bucket

# Count the number of objects in a bucket (similar but more compact results than
    in the previous command)
aws s3 ls --profile solvistas --recursive s3://ticketing-stadium-creator-dev/ | wc
    -l

# Listing all the applied bucket policies
aws s3api get-bucket-policy --profile solvistas --bucket
    ticketing-stadium-creator-dev
```

4.4.1 S3 Bucket Configuration

The configuration of an S3 bucket is a crucial step in the process of setting up an S3 pod. The bucket configuration determines the access permissions, storage classes, and other settings that affect how the bucket behaves. Here are some of the key configuration options that we used in our project:

To allow access to the S3 bucket for all users, we used the aws s3api put-bucket-policy command. This command applies a bucket policy that defines permissions for the bucket. In our case, we wanted to allow public read access to the objects in the bucket.

The following command has been used 10 with the `bucket-policy.json` configuration 11.

Listing 10: AWS CLI command to set a bucket policy

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

Listing 11: Bucket policy JSON configuration

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
        "Effect": "Allow",
        "Principal": "*",
        "Action": "s3:GetObject",
        "Resource": "arn:aws:s3:::ticketing-stadium-creator-dev/*"
        }
    ]
}
```

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.

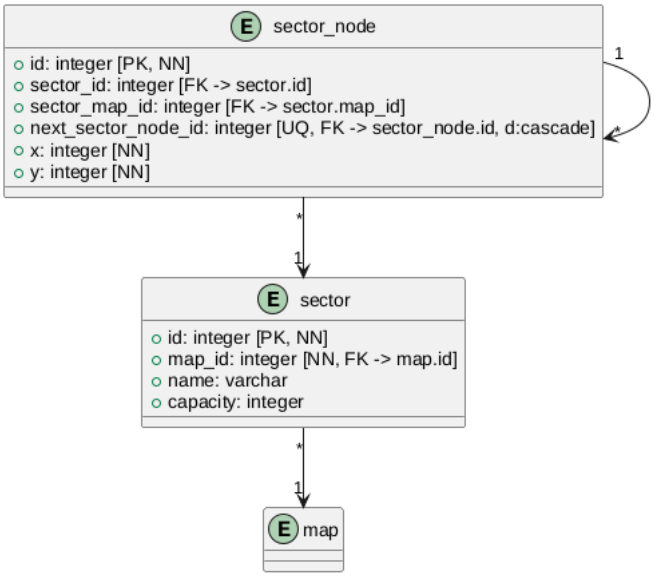


Abbildung 3: Standing Area Database Model

4.5 Add-Tool

4.6 Multiselect-Tool

4.7 Grid-Tool

4.8 Standing-Area-Tool

4.8.1 Frontend

4.8.2 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 database model is shown in figure 3. 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 us to model a hierarchy, and in our 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 12.

Listing 12: Recursive Query

```
WITH RECURSIVE cte AS (
    -- select first node with level 1
    SELECT , 1 AS level
    FROM   sector_node
    WHERE  (select id from sector_node where map_id = 1 and sector_id = 1 LIMIT 1)
        = id

    UNION  ALL
    SELECT  sn., c.level + 1 as level
    FROM cte c
        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
)
SELECT id, x, y
FROM   cte
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 13.

Listing 13: Standing Area Backend

```
override fun getSectorNodeBySector(sector: Sector): Set<SectorNode> {
    val startNode = db.getFirstSectorNodeBySector(sector.id!!)
    return startNode.collectSectorNodes(startNode)
}

private fun SectorNode.collectSectorNodes(startNode: SectorNode): Set<SectorNode> {
    return setOf(this) + (
            if (next == startNode) emptySet()
            else next?.collectSectorNodes(startNode) ?: emptySet()
        )
}
```

4.9 Design-Patterns

In the development of our application, we incorporated various design patterns to address specific challenges efficiently. Many sectors of the application required distinct patterns, including the interactive editor and tool functionalities. Because the interactive editor and tool system included a lot of complex features, we needed specific solutions. Undoing actions, for instance, is a widely expected usability feature in almost every modern editor, whether text-based or visual. This feature is applied broadly across software products, and various solutions exist. Special implementations of design patterns were utilized for these following parts of the application:

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

4.9.1 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

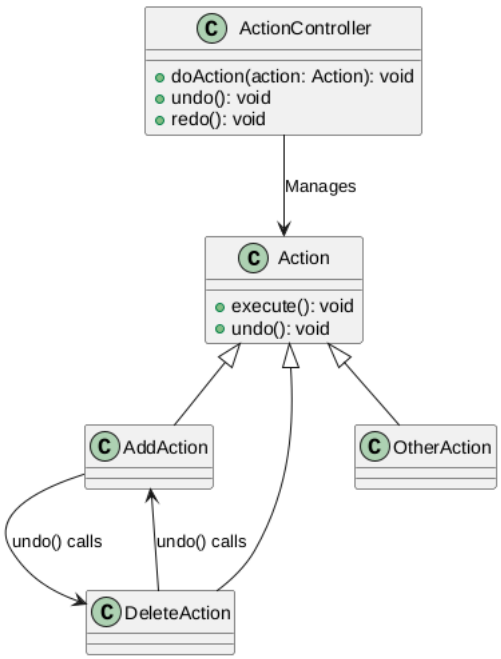


Abbildung 4: Command Pattern in SeatGen

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.

Our implementation defines an abstract `Action` class that all actions must implement. An overview of this is seen in 4. This class enforces the inclusion of `execute` and `undo` methods, ensuring a standardized approach to action management.

Listing 14: 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.

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 15: Add standing-area action implementation

```
export class AddStandingAreaAction implements Action {
    private readonly _newArea: StandingArea;
    private readonly _context: MapContextValue;

    constructor(newArea: StandingArea, context: MapContextValue) {
        this._newArea = newArea;
        this._context = context;
    }

    execute = () => {
        this._context.setStandingAreas((prevAreas) =>
            [...prevAreas, this._newArea]
        );
        if (this._context.standingAreasToDelete.includes(this._newArea.id))
        {
            this._context.setStandingAreasToDelete((prev) =>
                prev.filter(id => id !== this._newArea.id)
            );
        } else {
            this._context.setStandingAreasToSave((prev) => {
                const updated = [...prev, this._newArea.id];
                return updated;
            });
        }
    };

    undo = () => {
        new DeleteStandingAreaAction([this._newArea], this._context).execute()
    };
}
```

Listing 16: Delete standing-area action implementation

```
export class DeleteStandingAreaAction implements Action {
    private readonly _deletedAreas: StandingArea[];
    private _undoAreas: StandingArea[] | undefined;
    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 15 and 16 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 17.

Listing 17: Action controller implementation

```
const loopSize = 50;
const actionHistory: (Action | undefined)[] = new Array(loopSize).fill(undefined);

let historyIndex = 0;
let maxIndex = 0;
let minIndex = -1;
let savedIndex = 0

const loopSize = 50;
const actionHistory: (Action | undefined)[] = new Array(loopSize).fill(undefined);

let historyIndex = 0;
let maxIndex = 0;
let minIndex = -1;
let savedIndex = 0


const doAction = (action: Action) => {
    historyIndex = increase(historyIndex);

    while (maxIndex != historyIndex) {
        actionHistory[maxIndex] = undefined
        maxIndex = decrease(maxIndex)
    }

    actionHistory[minIndex] = undefined
    minIndex = increase(maxIndex);
    actionHistory[historyIndex] = action;
    action.execute!();
    checkForUnsavedChanges()
};

const updateSaveIndex = () => {
    savedIndex = historyIndex
    checkForUnsavedChanges()
}

function increase(num: number): number {
    return num !== loopSize - 1 ? num + 1 : 0;
}

function decrease(num: number): number {
    return num !== 0 ? num - 1 : loopSize - 1;
}

const undo = () => {
    if (historyIndex !== minIndex && actionHistory[historyIndex] !== undefined) {
        actionHistory[historyIndex]!.undo!()
        historyIndex = decrease(historyIndex);
        checkForUnsavedChanges()
        enqueueSnackbar("Undone", {variant: "info"})
    }
};

const redo = () => {
    if (historyIndex !== maxIndex && actionHistory[increase(historyIndex)] !==
        undefined) {
        historyIndex = increase(historyIndex);
        actionHistory[historyIndex]!.execute!();
        checkForUnsavedChanges()
    }
};

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 18. 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 18: 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.

4.9.2 Tools

Tools were a big part during development, so it was very important to make the implementation of new tools as easy and fast as possible. To achieve this we ended up

with a Tool interface, were instances of it just need to be added to an array where the tools are. This final version of the interface looks like seen in listing 19.

Listing 19: 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 20 list of all the implemented Tools and how they look, when the coresponding image 5 of how they are rendered.

Listing 20: Implemented Tools

```
const tools: Tool[] = [
    {
        id: "mouseTool",
        icon: <svg viewBox="0 0 24 24" xmlns="http://www.w3.org/2000/svg">
            <path
                d="..."
            />
        </svg>,
        onSelect: () => handleToolSelect(() => {
        }, "mouseTool", "default"),
        hotkey: "v"
    },
    {
        id: "addTool",
        icon: <PlusIcon></PlusIcon>,
        onSelect: () => handleToolSelect((e) => {
            props.addSeat(e.latlng.lat, e.latlng.lng)
        }, "addTool", "cell"),
        hotkey: "c"
    },
    {
        id: "addGridTool",
        icon: <TableCellsIcon></TableCellsIcon>,
        onSelect: () => handleToolSelect(() => {
        }, "addGridTool", "cell"),
```

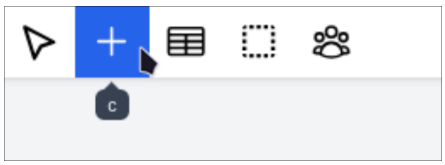


Abbildung 5: Tools in SeatGen

```
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 21.

Listing 21: 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.

4.9.3 Backend Services

For the backend services SeatGen uses a very general and exchangeable approach that works with Spring. For the services we define an interface that is implemented by the services. 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

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] Google Developers, „Compress Data More Densely with Zopfli,” 2013, Accessed: [18.2.2025]. Online verfügbar: https://developers.googleblog.com/en/compress-data-more-densely-with-zopfli/

Abbildungsverzeichnis

Tabellenverzeichnis

List of Listings

Appendix