

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 Rüp
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
1.1	Initial Situation	1
1.2	Problem Statement	1
1.3	Goal	2
2	Context / Environment Analysis	3
2.1	Overview	3
2.2	Stakeholder Analysis	3
2.3	Technical Environment	4
2.4	Requirements and Challenges	5
2.5	Summary	5
3	Technologies	6
3.1	React	6
3.2	Spring Boot and Kotlin	6
3.3	Database	7
3.4	AWS - S3	10
3.5	Leaflet	10
4	Implementation	11
4.1	Frontend Architecture	11
4.2	Leaflet Integration	11
4.3	Map Generation	11
4.4	Add-Tool	18
4.5	Multiselect-Tool	18
4.6	Grid-Tool	18
4.7	Standing-Area-Tool	18
4.8	Optimizations	18
4.9	Design-Patterns	18

5 Summary	19
Literaturverzeichnis	VI
Abbildungsverzeichnis	VII
Tabellenverzeichnis	VIII
List of Listings	IX
Appendix	X

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 Milestones ????

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 amount 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 choices underly these constraints and requirements in many places.

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. 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 straightforward, 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.

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

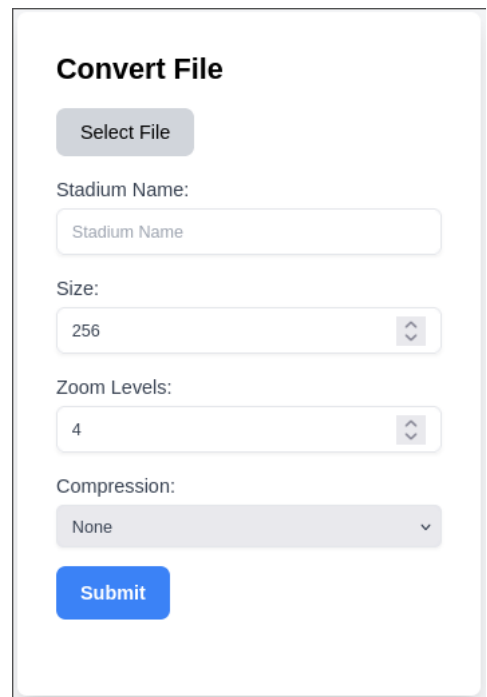
4.2 Leaflet Integration

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 algorithm, which is a dropdown menu with the options: No compression, Default algorithm and Zopfli



The screenshot shows a web form titled "Convert File". It contains the following elements: a "Select File" button; a "Stadium Name:" label followed by a text input field containing "Stadium Name"; a "Size:" label followed by a dropdown menu showing "256"; a "Zoom Levels:" label followed by a dropdown menu showing "4"; a "Compression:" label followed by a dropdown menu showing "None"; and a blue "Submit" button at the bottom.

Abbildung 1: New Map Mask

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 z -axis, while the horizontal and vertical axes correspond to the x and y axes of the map. Importantly, the x and y axes remain consistent across all zoom levels.

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

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

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

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

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

The total number of tiles is given by the sum:

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

As the number of zoom levels grows the number of tiles 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 2.

Listing 2: 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 3: 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 frontend 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.

To address this challenge, we employ lossless compression techniques. PNG compression primarily relies on the Deflate algorithm, 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.

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 compression, 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 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 can generate raw Deflate data streams or encapsulate them into gzip and zlib formats. It 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

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.

While Zopfli is significantly slower than standard Deflate or zlib, its higher compression ratio makes it particularly beneficial for static assets that are compressed once and distributed multiple times. This makes it ideal for optimizing web images, static assets, and other resources where storage and bandwidth efficiency are a priority.

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:

TABLE HERE TODO

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.

Firstly, 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.

Secondly, by compressing the images only once for each zoom level, we minimize the processing overhead. If we were to compress the images after slicing, we would have to compress the same visual data multiple times for different tile sizes or zoom levels. This redundancy not only wastes computational resources but also increases the time required for image processing, ultimately slowing down the application.

Additionally, compressing the images before slicing simplifies the process of parallelization. Given the extensive computational resources required for image compression, handling this task on the complete image allows us to better allocate resources and optimize performance.

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 4, the S3Client instance provides a range of operations, including putObject, getObject, and listObjectsV2, among others.

Listing 4: Configuring the S3 Client

```

1  @ConfigurationProperties(prefix = "aws")
2  data class S3Config @ConstructorBinding constructor(
3      val awsRegion: String,
4      val accessKey: String,
5      val secretKey: String
6  ){
7      @Bean(destroyMethod = "close")
8      fun s3Client() : S3Client {
9          return S3Client
10             .builder()
11             .overrideConfiguration(ClientOverrideConfiguration.builder()
12                 .apiCallTimeout(Duration.ofSeconds(10)).build()
13             )
14             .region(Region.regions()
15                 .find { region -> region.toString() == awsRegion }
16             )
17             .credentialsProvider(
18                 StaticCredentialsProvider.create(
19                     AwsBasicCredentials.create(accessKey, secretKey)
20                 )
21             )
22             .build()
23 }

```

The configuration is managed using the `@ConfigurationProperties(prefix = "aws")` 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 5.

Listing 5: AWS Configuration in `application.yml`

```

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

```

By using environment variables for sensitive credentials, 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.

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

4.3.5 AWS

4.3.6 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] 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

1 New Map Mask 11

Tabellenverzeichnis

List of Listings

1	Liquibase example changelog	9
2	Image dimensions calculation	13
3	Image Slicing Implementation	13
4	Configuring the S3 Client	16
5	AWS Configuration in application.yml	17

Appendix