

SeatGen - The Seating Plan Generation Tool For Stadiums

DIPLOMA THESIS

submitted for the

Reife- und Diplomprüfung

at the

Department of IT-Medientechnik

Submitted by:

Michael Ruep
Michael Stenz

Supervisor:

Prof. Mag. Martin Huemer

Project Partner:

Solvistas GmbH

Leonding, April 4, 2025

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

Leonding, April 4, 2025

Michael Ruep & Michael Stenz

Abstract

SeatGen is a tool developed for Solvistas GmbH to simplify stadium seating plan creation and management. It replaces the inefficient manual process with an intuitive graphical interface, allowing event organizers to design and edit seating layouts without technical expertise.

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



Contents

1 Environment Analysis

1.1 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:** Stadium layouts frequently change based on events and seasons, requiring continuous updates.
- **Limited Technical Staff:** The event organizers often rely on external developers to alter seat plan definitions, creating additional costs.

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

1.2 Stakeholder Analysis

Several parties interact with the SeatGen tool:

- **Developers:** Historically, Solvistas' developers modified the seat layout text files. The new approach aims to minimize their involvement in the long term, except for initial and advanced configurations.
- **Event Organizers and Stadium Managers:** These stakeholders need the intuitive tools to make update to the seat maps without dealing with complex raw data formats.
- **Ticketing Platform Users:** The final seat layouts are used in Solvistas' ticketing service. Although these end-users do not edit the data themselves, the accuracy and clarity of seat layouts crucially impact their experience at the stadium.

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

1.3 Technical Environment

On the technical side, the SeatGen application integrates with:

- **AWS S3 Buckets:** Image tiles and map data are uploaded to and fetched from a secure Amazon Web Services (AWS) cloud environment.
- **React-based Frontend:** Provides an interactive user interface for managing and modifying seat layouts with real-time updates.
- **Spring Boot / Kotlin Backend:** Manages image processing and seat data handling.
- **PostgreSQL Database:** Stores seat configurations, categories, groups, sectors, and map data.

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

1.4 Requirements and Challenges

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

- **Immediate Feedback:** When a user modifies a seat position, the update is reflected instantly on the map. Dragging seats should feel instant and follow the user, mimicking real-life interactions
- **Simplicity and Learnability:** Familiar mouse-based interactions should allow users to perform specific and complex tasks—such as grouping seats, creating standing areas, or categorizing—without requiring specialized training or an understanding of coordinate systems.
- **Cognitive Offloading:** By representing seats visually, the mental effort to interpret raw text-based seat coordinates or stadium layouts is reduced significantly. Which improves efficiency and accuracy by a huge margin.

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

1.5 Summary

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

2 Technologies

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

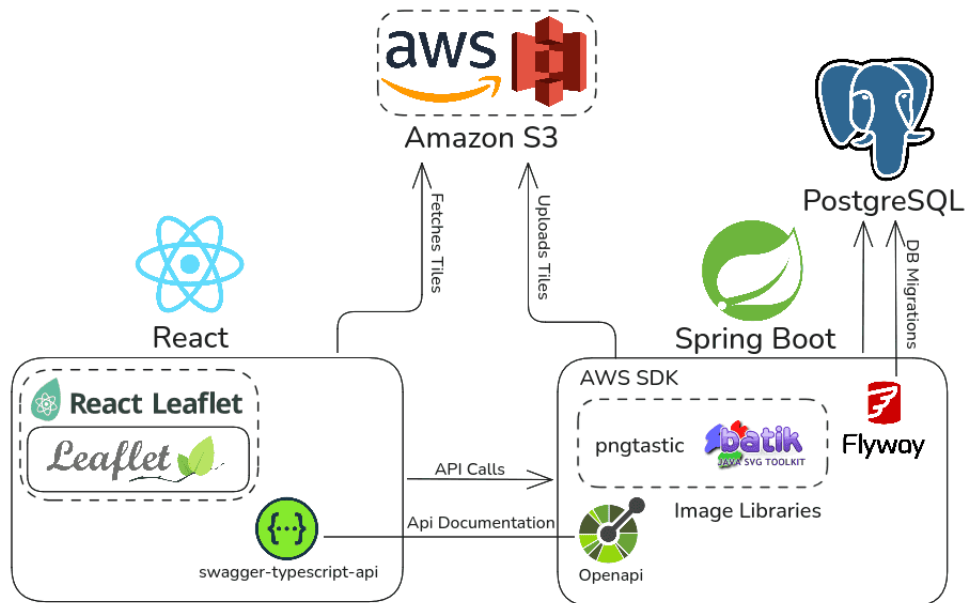


Figure 1: Tech Stack

2.1 React

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

2.1.1 Framework Background

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

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

2.1.2 Virtual DOM

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

2.1.3 Component-based Architecture

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

- Seat Map Component
- Toolbar Component
- Detail Component

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

2.1.4 Integration of Libraries

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

2.1.5 Developer Familiarity and Team Expertise

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

Comparison to Other Frameworks

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

2.1.6 Summary

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

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

2.2 Leaflet

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

2.2.1 Reasons for Leaflet

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

A key advantage of Leaflet is its low dependency footprint. Unlike other mapping solutions that rely on external APIs or heavy SDKs, Leaflet provides a standalone JavaScript library that integrates seamlessly with React. This lightweight approach ensures excellent map performance, even when handling large stadiums with thousands of seats. In contrast, frameworks like Google Maps API enforce rate limits and external API calls, which can introduce latency and unnecessary costs.

Leaflet's customizability also played a significant role in our decision. SeatGen requires custom zoom levels, and real-time updates, all of which are efficiently handled using Leaflet's open architecture. Unlike proprietary mapping tools, Leaflet allows full control over rendering logic, making it easier to optimize performance and adjust the visualiza-

tion to match stadium layouts precisely [?]. Furthermore, existing Leaflet functions were adapted for tasks such as seat selection and the grid tool. This significantly accelerated the development process

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

2.2.2 Key Features Used in SeatGen

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

- **Dynamic Seat Rendering:** Leaflet enables the efficient rendering of thousands of seat markers without significantly impacting performance. Since stadiums can contain a large number of seats, rendering was optimized using Leaflet layers to manage visibility at different zoom levels.
- **Custom Zoom Levels and Scaling:** Leaflet allows the definition of custom zoom levels, ensuring precise seat selection when zoomed in and a full view of the stadium's structure when zoomed out.
- **Interactive Seat Selection:** By leveraging Leaflet's built-in event handling system, users can click and modify seats in real time. This is crucial for intuitively adjusting seating arrangements.
- **Grid-Based Seat Placement:** Leaflet's selection, polygon, and coordinate functions were used to implement functions like the grid tool and selection tool, allowing for structured seat placement. This feature speeds up the process of generating rows and sections by automatically aligning seats according to predefined parameters.
- **Real-Time State Management with React Context:** Since Leaflet does not natively integrate with React's state management, React Context was used to synchronize seat selections, updates, and modifications across the application. This ensures that any seat change is reflected immediately in both the UI and the underlying data model.

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

2.2.3 Integration with React

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

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

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

2.2.4 Summary

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

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

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

3 Implementation

3.1 Frontend Architecture

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

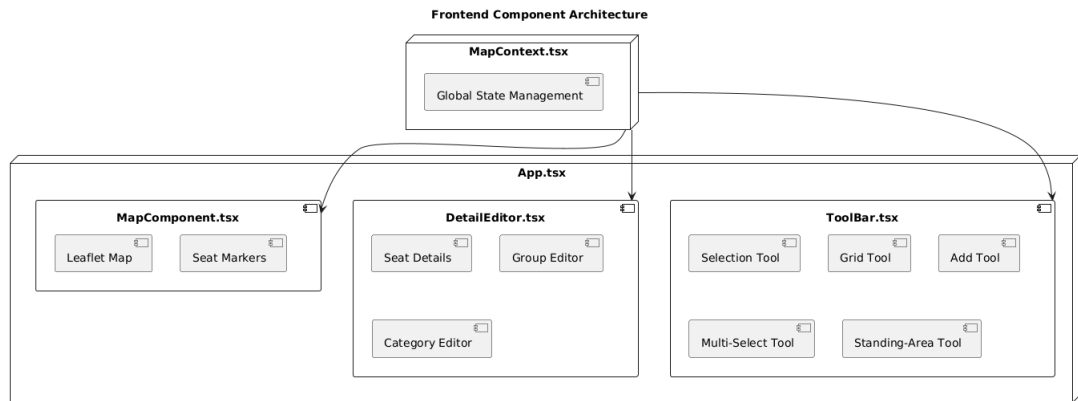


Figure 2: Frontend Component Architecture Overview (Rough Overview)

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

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

- **Landing Page (LandingPage.tsx):** The first screen that users see upon opening SeatGen. It provides an introduction to the tool and a project overview for first-time users.
- **Home Menu (Home.tsx):** This serves as the main project selection interface. Users can choose between loading an already saved seating plan or creating a

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

- **Menu and Navigation (Menu.tsx):** The global navigation system connects different views within SeatGen. It allows users to switch between the seating map editor, settings, and export functionalities. The navigation is designed to remain persistent throughout the application to provide seamless transitions between different tasks.
- **Settings Panel (SettingsPanel.tsx):** This panel allows users to configure global preferences that influence the overall seating arrangement experience. Options include:
 - **Default seat categories:** Users can predefine categories to streamline seat assignments.
 - **Theme selection:** Dark or light mode based on user preference.
 - **Seat map scaling:** Allows fine-tuned adjustments of grid seat density.
- **Modals and Popups:** The frontend includes various popups and confirmation dialogs:
 - **Modal.tsx:** Used for confirmations, warnings, and additional actions.
 - **LeavePagePopup.tsx:** Warns users about unsaved changes before leaving.

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

3.1.1 MapComponent.tsx: Interactive Map Rendering

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

Key Responsibilities:

- Initializes and manages the **Leaflet map instance**.
- Loads seat and standing area data dynamically from the backend.
- Renders seats and standing areas as **interactive markers and polygons**.

- Handles user interactions such as **clicking, selecting, and dragging seats**.
- Implements **state synchronization** with the global context (`MapContext.tsx`).
- Provides support for **tool-based seat editing** (e.g., adding, deleting, moving).

Component Structure:

- **State Management:** Uses `useState`, `useEffect`, and `useContext` to manage map data.
- **Leaflet Integration:** Utilizes `MapContainer` from `react-leaflet` for seamless map rendering.
- **Performance Enhancements:** Implements `useCallback` and `useRef` to optimize re-renders.

Map Initialization and Leaflet Integration

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

Listing 1: Initializing Leaflet Map in React

```

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

Custom CRS (Coordinate Reference System):

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

- Unlike traditional geographic maps, SeatGen does not need a curved map.
- All coordinates can be converted to a Cartesian X/Y grid.

Custom Rendering Engine:

- Uses **L.canvas()** instead of SVG for performance.
- Enables handling of thousands of seat markers.
- Reduces DOM load by rendering elements in a drawing surface.

Dynamically Loading Map Tiles:

- The tile URL is dynamically constructed based on `context.mapInfo`.
- Tiles are loaded asynchronously to improve map load times.

Fetching Seat and Standing Area Data

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

Listing 2: Fetching Seat and Category Data

```

1  useEffect(() => {
2    if (bucketName && mapName) {
3      // Fetch basic stadium map information
4      seatgenApiClient.api.info(bucketName, mapName)
5        .then(response => context.setMapInfo(response.data))
6        .catch(error => console.error('Error fetching map info:', error));
7
8      // Fetch seat categories before retrieving individual seats
9      seatgenApiClient.api.getAllCategories().then((r) => {
10         if (r.ok) {
11           context.setCategories(r.data);
12           seatgenApiClient.api.getSeatsByMap(bucketName,
13             mapName).then(response => {
14             if (r.ok) {
15               context.setSeats(response.data.map((s) => ({
16                 id: s.seatId!,
17                 position: { lat: s.xcoord!, lng: s.ycoord! },
18                 category: r.data.find(it => it.id === s.categoryId) ??
19                   null
20               })));
21             }
22           });
23         }
24       });
25
26       // Fetch standing areas
27       seatgenApiClient.api.getSectorByMap(bucketName, mapName).then((r) => {
28         if (r.ok) {
29           context.setStandingAreas(r.data.map((data) => ({
30             id: data.id!,
31             name: data.name!,
32             capacity: data.capacity,
33             coordinates: data.coordinates?.map((c) => new L.LatLng(c.x!,
34               c.y!)) ?? [],
35             selected: false
36           })));
37         }
38       });
39     }
40   }
41 } else {

```

```

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

```

Breakdown of Logic:

- Fetches stadium metadata (mapInfo) and sets it globally.
- Retrieves seat categories.
- Retrieves seat positions from the backend and maps them into React state.
- Ensures data consistency by linking seats to their corresponding categories.

Example of Mapped Seat Object:

Listing 3: Seat Object in State

```

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

```

Fetching Standing Area / Sector Data

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

Listing 4: Fetching Standing Areas

```

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

```

Explanation:

- The API returns a list of sector polygons.
- Each area consists of a unique ID, name, capacity, and a list of coordinates.
- Coordinates are transformed into Leaflet's LatLng format to be rendered as a polygon.

Example of a Standing Area Object in State:

Listing 5: Standing Area Object in State

```

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

```

State Management and Performance

- `useEffect` Dependency Array:
 - Ensures the API calls only run when `bucketName` or `mapName` change.
 - Prevents unnecessary re-fetching on every render.
- Efficient State Updates:
 - Avoids unnecessary re-renders by batching state updates for seats and standing areas.
 - No prop-drilling by storing globally needed fetched data in the `MapContext`.
- Error Handling:
 - If any API call fails, an error is logged, and the operation is skipped.
 - Ensures that failures in one request do not crash the entire component.

Rendering Seats and Handling Selection

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

Rendering Seat Markers:

Listing 6: Rendering and Selecting Seats

```

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

```

```

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

```

How This Works:

- Clicking a seat toggles its selected state.
- Holding **Ctrl** allows multi-selection, useful for bulk actions.
- Clicking a seat opens the **DetailEditor sidebar** for further modifications.

Implementing Live Seat Dragging and Moving

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

Tracking Initial Positions

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

Listing 7: Storing Initial Positions Before Dragging

```

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

```

How This Works:

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

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

Updating Seats in Real-Time During Dragging

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

Listing 8: Updating Seat Positions During Dragging

```

1  const updateSelectedSeatsPosition = useCallback((draggedSeatId: number,
2    newLatLngPosition: { lat: number; lng: number }) => {
3    const map = mapRef.current;
4    const primarySeat = context.seats.find(s => s.id === draggedSeatId);
5
6    if (!map || !primarySeat || !primarySeat.selected) return;
7
8    const newPosition = map.latLngToLayerPoint(newLatLngPosition);
9    const oldPosition = initialPositionsRef.current[draggedSeatId];
10
11    const deltaX = newPosition.x - oldPosition.x;
12    const deltaY = newPosition.y - oldPosition.y;
13
14    context.setSeats(prevSeats =>
15      prevSeats.map(seat => {
16        if (seat.selected) {
17          const initialPosition = initialPositionsRef.current[seat.id];
18          const newPosX = initialPosition.x + deltaX;
19          const newPosY = initialPosition.y + deltaY;
20          const newLatLngPos = map.layerPointToLatLng(new L.Point(newPosX,
21            newPosY));
22          return { ...seat, position: newLatLngPos };
23        }
24        return seat;
25      })
26    );
27
28    // Update MoveAction for undo tracking
29    const currentAction = context.getCurrentAction();
30    if (currentAction instanceof MoveAction) {
31      currentAction.setNewSeats(selectedSeats);
32    }
33  }, [context.seats, context]);

```

Breakdown of Logic:

- The drag delta (change in X/Y position) between the starting position and the new cursor position is calculated.
- The same delta is applied to all selected seats, ensuring they move together.
- Positions are converted between LatLng (geo-coordinates) and pixel points, so dragging works consistently at different zoom levels.
- Changes using MoveAction are tracked, allowing the operation to be undone if needed.

Optimizations and Challenges

Major Challenges Encountered:

- Preventing position drift when switching between zoom levels.
- Ensuring smooth movement with large seat selections.
- Avoiding excessive re-renders that slow down performance.

Performance Optimizations:

- Used `useRef` to store initial positions instead of state (prevents extra re-renders).
- Applied batch updates for all selected seats instead of updating them individually.
- Optimized Leaflet `latLng` to pixel conversion for to be able to drag the seats naturally (direct manipulation [?]).

Conclusion

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

Managing Standing Areas and Sectors

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

Standing Area Features:

- **Custom Polygons:** Users define standing areas by selecting points on the map.
- **Capacity Control:** Limits the number of tickets available per standing area.
- **Category Assignment:** Standing areas can be assigned different categories.
- **Real-time Editing:** Areas can be renamed, and deleted dynamically.

Fetching Standing Areas from the Backend:

Listing 9: Fetching Standing Areas

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

```

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

```

Standing Area Selection:

Listing 10: Handling Standing Area Selection

```

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

```

Selection and Editing Process:

- Clicking a standing area toggles its selected state.
- In the Detail Editor panel renaming, capacity adjustments, and category (including color coding) updates are possible.
- The user can resize the polygons on creation to modify the area covered.

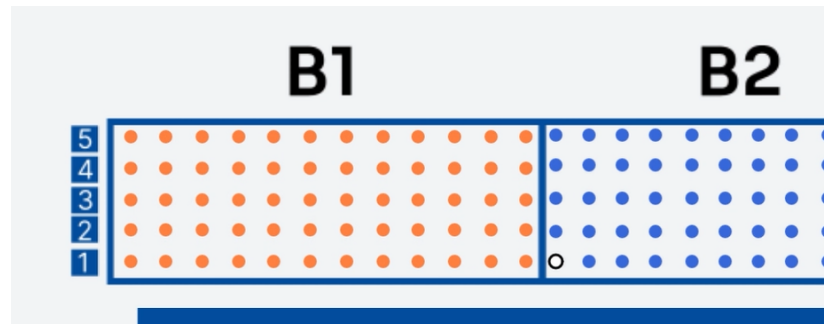


Figure 3: Map, Seats with Category Color, and Selected Seat in MapComponent.tsx

Event Handling in Leaflet

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

Global Click Handling:

Listing 11: Handling Global Click Events

```
1  const MapEvents = () => {
2    useMapEvents({
3      click(e) {
4        context.setSeats(prevSeats => prevSeats.map(seat => ({ ...seat,
5          selected: false })));
6        context.setSelectedStandingAreaIds([]); //Deselects Standing Areas
7          because Seats are selected
8      }
9    });
10   return null;
11 }
```

Drag Events for Seat Movement:

Listing 12: Handling Seat Drag Events

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


- The `handleDragStart` function is called when a user begins dragging a seat. It stores the initial positions of selected seats to ensure relative movement.
- The `handleDragEnd` function finalizes the new seat positions when dragging stops.
- Event listeners (`onDragStart` and `onDragEnd`) are added to each `Seat` component to trigger these functions dynamically.
- The `useMemo` hook optimizes rendering performance by ensuring seat markers are not unnecessarily re-created during re-renders.

Event Handling Summary:

- Click Events: Used for selecting seats and standing areas.
- Drag Events: Applied to dynamically reposition seats.
- Map Events: Ensure global deselection when clicking outside elements.

Saving and Syncing with the Backend

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

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

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

- Detects unsaved changes via the `hasUnsavedChanges` prop.
- Attaches a `beforeunload` event listener to prevent accidental exits.
- Displays a native browser warning when users try to leave.
- Removes the event listener when no longer needed to optimize performance.

Listing 13: Auto-Saving on Unload

```

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

```

```

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

```

How It Works:

- The `hasUnsavedChanges` prop determines whether to enable the warning.
- When unsaved changes are detected, a `beforeunload` event listener is added.
- If the user attempts to leave the page, the browser displays a warning.
- The event listener is removed when the component unmounts to prevent memory leaks.

Batch Save:

Listing 14: Batch Save Mechanism

```

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

```

Batch Save Mechanism:

- Instead of saving each individual seat change separately, SeatGen groups multiple seat updates into a single API request to reduce network overhead and improve efficiency.
- The `useCallback` hook ensures that changes get only saved when `context.seats` changes, preventing redundant function executions.
- `seatgenApiClient.api.saveSeats` sends the updated seat data to the backend, including the seat ID, coordinates, and category ID.
- Upon a successful save, a snackbar notification is displayed to confirm the operation.
- SeatGen maintains the action history so users can revert unintended changes before saving.

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

Summary

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

- Interactive Leaflet-based Map Rendering
- Seat and Standing Area Rendering
- Group Selection and Bulk Editing
- Drag-and-Drop Seat Repositioning
- Event Handling for User Interaction
- Synchronizing data with global state (`textttMapContext.tsx`)
- Asynchronous Backend Synchronization

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

3.1.2 DetailEditor.tsx: Editing Attributes

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

Key Responsibilities:

- **Editing Individual Seats:** Users can update seat tooltips, positions, and categories.
- **Managing Seat Groups:** Enables users to create, merge, and delete groups of seats.
- **Standing Area Editing:** Supports renaming and capacity adjustments for standing areas.
- **Category Assignment:** Allows users to assign pricing tiers and colors to seats.

3.1.3 Component Structure

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

- **Seat Editing Panel:** Displays detailed information for selected seats and allows modifications.
- **Group Management Panel:** Handles seat grouping and bulk operations.
- **Standing Area Editing Panel:** Enables editing of standing areas, including name and capacity.
- **Deletion and Bulk Actions:** Supports removing selected seats, groups, or selected standing areas.

Seat X

Tooltip:

Edit Tooltip Text

X: 941 Y: 426

Move Steps: 5

Category:

Delete Create Group

Figure 4: Seat Editing Panel in `DetailEditor.tsx`

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

3.1.4 Managing Seat Attributes

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

Updating Tooltip Text

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

Listing 15: Updating Tooltip for Selected Seats

```
1 <input
2   type="text"
3   value={props.currentTooltip}
4   onChange={props.handleExternalTooltipChange}
5   placeholder="Edit Tooltip Text"
6 />
```

How It Works:

- The text input dynamically updates the tooltip for all selected seats by calling the `handleExternalTooltipChange` function.
- Changes are stored in the global state.
- Provides immediate visual feedback to the user.

Adjusting Seat Position via UI

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

Listing 16: Updating Seat Coordinates

```
1 const updateSeatPosition = (x: number, y: number) => {
2   const latLng = props.mapRef.current!.layerPointToLatLng(new L.Point(x, y));
3   props.updateSelectedSeatsPosition(selectedSeats[0].id, { lat: latLng.lat, lng:
4     latLng.lng }, true);
5 };
```

How It Works:

- Converts user-inputted X/Y values into map coordinates.
- Updates seat position dynamically.
- With this the movement is optimized for both manual entry and real-time adjustments.

3.1.5 Group Management and Multi-Selection

Grouping seats allows users to efficiently manage large stadium sections.

Groups and Multi Selection and Deletion

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

Use Cases for Seat Groups:

- **Bulk Editing:** Modify multiple seat attributes simultaneously.
- **Efficient Repositioning:** Move multiple seats while maintaining relative positioning.
- **Category Assignment:** Assign pricing tiers and access restrictions to an entire section.
- **Simplified Deletion:** Remove entire seat groups without manually selecting each seat.

Creating a Seat Group:

Listing 17: Creating Seat Groups

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

How It Works:

- The function retrieves all currently selected seats.
- The CreateGroupAction is executed, registering the selected seats as a group.
- The group ID is assigned, and the group is stored in the global state.

Deleting a Seat Group:

Listing 18: Deleting Seat Groups

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

Group Deletion Process:

- The `DeleteGroupAction` removes all seats within that group using the `groupId`.
- The global context state updates accordingly.
- The operation is reversible via the undo stack.

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

Seat Categories

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



Figure 5: Category Selection in `DetailEditor.tsx`

Features of Seat Categories:

- **Color-Coding:** Each category is assigned a color for clear visualization.
- **Pricing Information:** Categories define pricing tiers, ensuring correct ticket pricing.
- **Flexible Assignments:** Seats can be reassigned to different categories as needed.
- **Bulk Category Updates:** Multiple seats can be assigned a category simultaneously.

Category Data Model

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

Listing 19: Seat Category Data Model

```

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

```

Assigning Categories to Seats

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

Listing 20: onChange Assigning Categories to Selected Seats

```

1 onChange={newVal => {
2   console.log(newVal)
3   context.doAction(new UpdateCategoryAction(selectedSeats, newVal ? newVal.value
4     : null))
5   setCurrentSelectedCategory(newVal ? newVal.value : null);
6 }}

```

Listing 21: Assigning Categories to Selected Seats

```

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

```

How It Works:

- The function in `UpdateCategoryAction` loops through all selected seats.
- The new category is assigned based on the provided `categoryId`.
- The UI updates instantly, applying the new color and classification.

Category Management in the UI

Users can manage categories by:

- Creating new categories with custom colors and pricing.
- Editing existing categories, updating names, prices, or colors.
- Deleting unused categories.

Listing 22: Managing Categories

```

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

```



```

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

```

Listing 23: Managing Category Color

```

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

```

Category Visualization on the Map

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

Listing 24: Rendering Seat Markers with Categories

```

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

```

How It Works:

- Each seat marker inherits the category's color.
- Unassigned seats default to a neutral color (gray).
- Selecting a seat allows users to change its category.

Conclusion

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

3.1.6 Standing Area Editing

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

Key Features of Standing Areas:

- **Custom Polygonal Boundaries:** Users can define standing areas by selecting points on the map.
- **Capacity Control:** Each area has a maximum capacity limit.
- **Real-Time Editing:** Users can rename standing areas and adjust their capacities dynamically.
- **Deletion and Reconfiguration:** Existing standing areas can be removed or modified at any time.

Standing Area Data Model

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

Listing 25: Standing Area Data Model

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

Creating and Selecting Standing Areas

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

Renaming a Standing Area

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

Listing 26: Renaming Standing Areas

```
1 const handleStandingNameChange = (name: string) => {
2   context.setStandingAreas(prev => prev.map(area =>
3     context.selectedStandingAreaIds.includes(area.id)
4     ? { ...area, name: name }
5   )
6 }
```

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

How It Works:

- The function updates the name of all selected standing areas.
- Changes are reflected instantly in the UI.
- The new name is stored persistently for future sessions.

Updating Standing Area Capacity

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

Listing 27: Updating Standing Area Capacity

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

Capacity Adjustment Process:

- The function modifies the capacity of all selected standing areas.
- Input validation ensures that only numeric values are accepted.
- The UI dynamically updates, reflecting the new ticketing constraints.

Deleting a Standing Area

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

Listing 28: Deleting Standing Areas

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

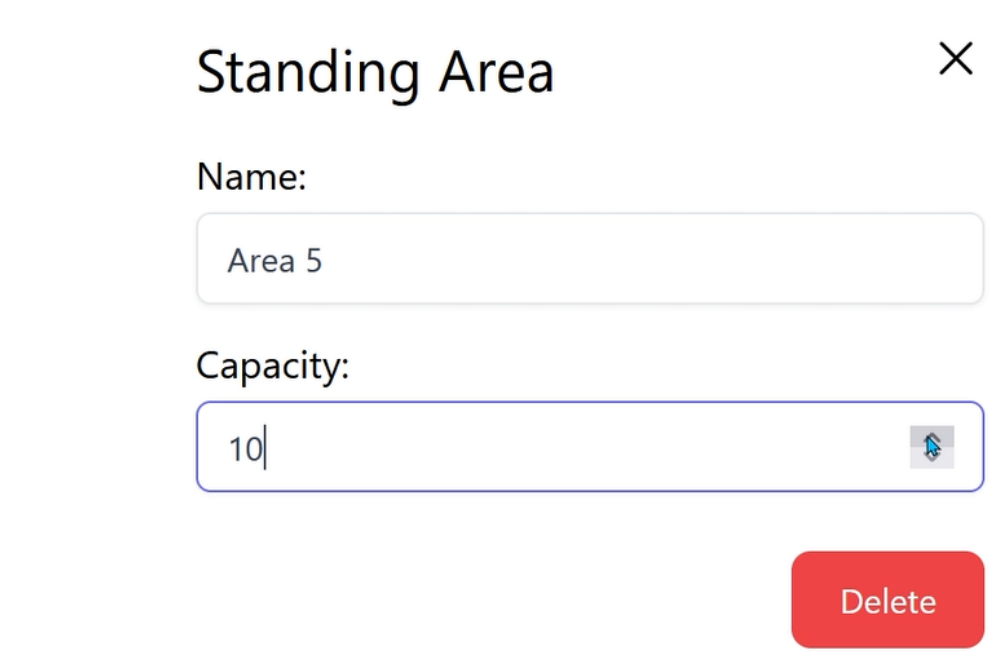


Figure 6: Editing Standing Areas in DetailEditor.tsx

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

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

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

Summary

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

- **Seat Editing:** Real-time adjustments to tooltips, positions, and assignments.
- **Group Management:** Merging, splitting, and deleting seat groups efficiently.
- **Category Assignment:** Bulk updates with intuitive color-coded visualization.

- **Standing Area Modifications:** Easy editing of boundaries and capacities.
- **Seamless Integration:** Ensures consistent updates via `MapContext.tsx`.

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

3.1.7 ToolBar.tsx: Tool Management and User Interaction

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

Key Responsibilities:

- **Tool Selection:** Provides quick access to various editing tools.
- **Map Title Editing:** Allows renaming the map for better organization.
- **Saving and Previewing:** Ensures seamless data persistence and view-only mode.

Component Structure

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

- **Tool Icons:** Represent different editing modes (selection, adding seats, grid placement, area selection, standing area placement).
- **Map Name Editor:** Enables modifying the displayed name of the stadium layout.
- **Save and Preview Buttons:** Handles storing changes and switching to a preview mode.

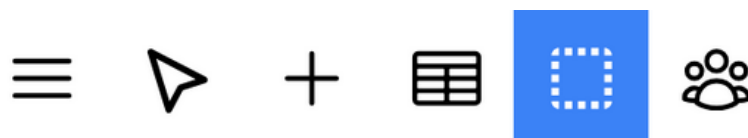


Figure 7: Toolbar Icons in ToolBar.tsx

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

Tool Selection and Activation

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

Listing 29: Defining Available Tools

```

1  const tools: Tool[] = [
2    {
3      id: "addTool",
4      icon: <PlusIcon></PlusIcon>,
5      onSelect: () => handleToolSelect((e) => {
6        props.addSeat(e.latlng.lat, e.latlng.lng)
7      }, "addTool", "cell"),
8      hotkey: "c"
9    },
10   {
11     id: "addGridTool",
12     icon: <TableCellsIcon></TableCellsIcon>,
13     onSelect: () => handleToolSelect(() => {
14       }, "addGridTool", "cell"),
15     hotkey: "g"
16   },
17   {
18     id: "standingAreaTool",
19     icon: <UserGroupIcon />,
20     onSelect: () => handleToolSelect(() => console.log("standingtool"),
21       "standingAreaTool", "crosshair"),
22     hotkey: "s"
23   }
24 ];

```

How It Works:

- Each tool has an icon and an activation function.
- Hotkeys allow quick switching between tools.
- Selected tools modify the user's interaction mode (e.g., clicking, selecting, adding seats).

Editing the Map Name

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

Listing 30: Editing Map Name

```

1  {
2    !editName ? (
3      <>
4        <p>{displayName}</p>
5        <PencilIcon onClick={() => setEditName(true)} className="mapNameEdit
6          cursor-pointer" />
7      </>
8    ) : (
9      <>
10       <input
11         autoFocus
12         type='text'
13         className="text-center"
14         value={tempDisplayName}
15         onChange={(e) => setTempDisplayName(e.target.value)}
16       />

```

```

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

```

How It Works:

- Clicking the pencil icon allows editing the map title.
- The name is updated and saved via an API request.
- The change is immediately reflected in the UI.

Save and Preview Buttons

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



Figure 8: Save and Preview Buttons in ToolBar.tsx

Listing 31: Saving and Previewing

```

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

```

How It Works:

- The **Save** button persists all seat and standing area modifications to the backend.
- A checkmark icon confirms successful saving, while a warning icon indicates unsaved changes.
- The **Preview** button switches the interface to a non-editable mode.

Summary

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

- **Interactive Tool Selection:** Enables switching between different seat and area modification tools.
- **Map Naming:** Allows users to rename and organize stadium layouts.
- **Save and Preview Mechanism:** Ensures data integrity and view-only mode.

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

3.2 Add-Tool

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

Key Responsibilities:

- Enables manual placement of seats at a specific position on the map.
- Internally dispatches an **AddSeatAction** to update global state.
- Supports undo and redo via the action system.

3.2.1 Triggering the Action

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

Listing 32: Calling AddSeatAction

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

Explanation:

- The **newSeat** is constructed outside and passed to the action.
- The action handles the insertion of the seat into global state.
- This mechanism ensures consistent integration with undo/redo logic.

3.2.2 AddSeatAction Implementation

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

Listing 33: AddSeatAction Implementation

```

1  export default class AddSeatAction extends Action {
2    private readonly newSeat: Seat;
3    private readonly setSeats: React.Dispatch<React.SetStateAction<Seat []>>;
4
5    constructor(newSeat: Seat, setSeats:
6      React.Dispatch<React.SetStateAction<Seat []>>) {
7      super();
9      this.newSeat = newSeat;
10     this.setSeats = setSeats;
11   }
12
13   execute(): void {
14     this.setSeats(prev => [...prev, this.newSeat]);
15   }
16
17   undo(): void {
18     this.setSeats(prev => prev.filter(seat => seat.id !== this.newSeat.id));
19   }
20 }

```

How It Works:

- `execute()` appends the new seat to the seat list.
- `undo()` removes it again by filtering the ID.
- This implementation ensures full undo/redo compatibility.

3.2.3 Conclusion

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

3.3 Grid-Tool

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

Key Responsibilities:

- Lets users draw a bounding box to define the grid area.
- Opens a modal dialog to configure the number of rows, columns, and padding.
- Internally dispatches an `AddMultipleSeatAction` to add the generated seats.

3.3.1 User Interaction

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

Listing 34: Handling Box Selection

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

Explanation:

- Stores the selected bounds.
- Opens the modal to allow the user to input grid parameters.

3.3.2 Generating the Seat Grid

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

Listing 35: Add Grid Logic

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

How It Works:

- The bounds are converted into a grid layout using latitude and longitude step sizes.
- Margin (padding) is applied to space the grid evenly.
- All seats are bundled into an AddMultipleSeatAction for batch insertion and undo support.

3.3.3 Modal Interface

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

Listing 36: Modal Input Fields

```

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

```

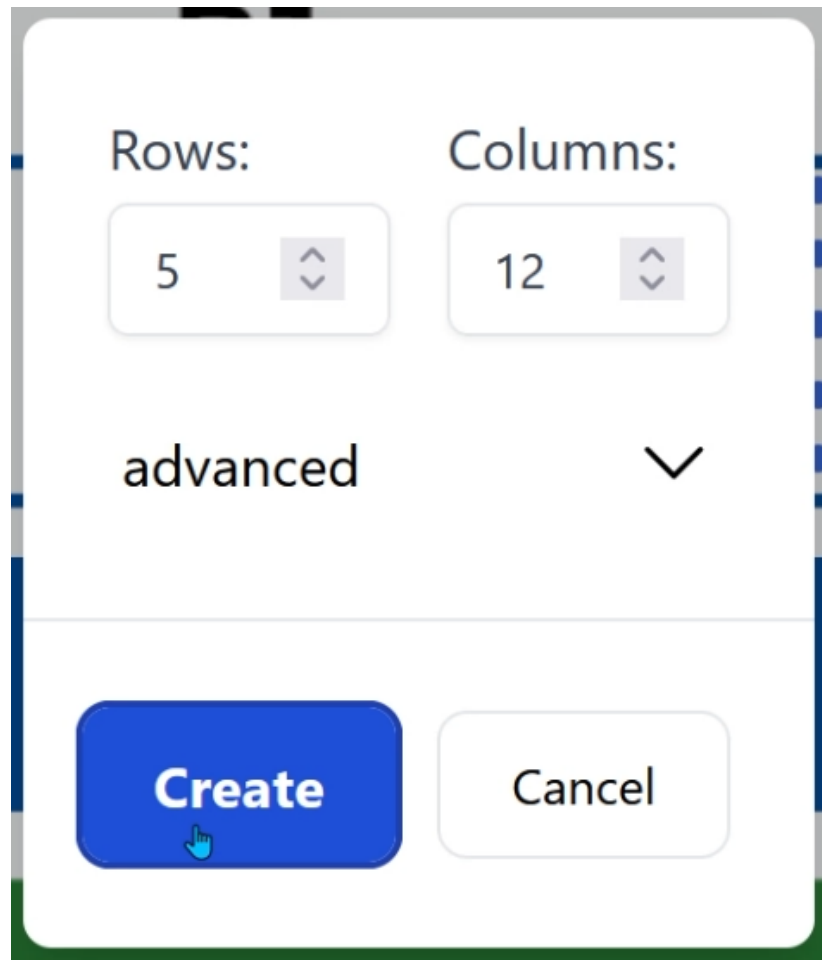


Figure 9: Toolbar with Grid-Tool Selected and Configuration Modal Open

3.3.4 Conclusion

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

3.4 Standing-Area-Tool

3.4.1 Frontend

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

Key Responsibilities:

- Allows users to draw a polygon on the map by clicking multiple points.
- Once finalized, dispatches an `AddStandingAreaAction` to insert the area into global state.
- Automatically assigns an ID, empty name, and zero capacity by default.
- Supports full undo and redo through the action system.

Capturing the Area

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

Listing 37: Adding Polygon Points

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

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

Finalizing the Standing Area

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

Listing 38: Finalizing Standing Area

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

Explanation:

- Ensures at least three points are present to form a valid polygon.
- Constructs a new `StandingArea` object with default values.
- Dispatches an `AddStandingAreaAction` to insert the new area into state.
- Resets the drawing state for the next operation.

Integration with the Action System

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

Listing 39: AddStandingAreaAction

```

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

```

Highlights:

- `execute()` appends the new area to the list.
- `undo()` removes the inserted area by filtering it out by ID.

Summary

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

4 Summary

4.1 Michael Ruep

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

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

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

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

List of Figures

List of Tables

List of Listings

1	Initializing Leaflet Map in React	12
2	Fetching Seat and Category Data	13
3	Seat Object in State	14
4	Fetching Standing Areas	14
5	Standing Area Object in State	15
6	Rendering and Selecting Seats	15
7	Storing Initial Positions Before Dragging	16
8	Updating Seat Positions During Dragging	17
9	Fetching Standing Areas	18
10	Handling Standing Area Selection	19
11	Handling Global Click Events	20
12	Handling Seat Drag Events	20
13	Auto-Saving on Unload	21
14	Batch Save Mechanism	22
15	Updating Tooltip for Selected Seats	25
16	Updating Seat Coordinates	25
17	Creating Seat Groups	26
18	Deleting Seat Groups	26
19	Seat Category Data Model	28
20	onChange Assigning Categories to Selected Seats	28
21	Assigning Categories to Selected Seats	28
22	Managing Categories	28
23	Managing Category Color	29
24	Rendering Seat Markers with Categories	29
25	Standing Area Data Model	30
26	Renaming Standing Areas	30
27	Updating Standing Area Capacity	31
28	Deleting Standing Areas	31
29	Defining Available Tools	34
30	Editing Map Name	34
31	Saving and Previewing	35
32	Calling AddSeatAction	36
33	AddSeatAction Implementation	37
34	Handling Box Selection	38
35	Add Grid Logic	38
36	Modal Input Fields	39
37	Adding Polygon Points	40
38	Finalizing Standing Area	40
39	AddStandingAreaAction	41

Appendix