SocialSpot — A Social Platform for Event Discovery

Technical Report: CL-TR-2025-42, June 2025

Andreas Hecht, Oliver Kneidl, Eva-Maria Maurer, Leon Minks,
Yannick Ondra, Christoph P. Neumann
CyberLytics-Lab at the Department of Electrical Engineering, Media, and Computer Science
Ostbayerische Technische Hochschule Amberg-Weiden
Amberg, Germany

***Abstract*—SocialSpot is a social media-style website focused on event discovery and interaction. It aims to fill the gap left by mainstream social media platforms like Facebook and Instagram, where events are often buried in personal content. SocialSpot instead places events at the heart of its user experience. By combining this event-first approach with familiar social media features, such as liking and commenting, SocialSpot creates an intuitive space for discovering, creating, and engaging with local events. Features include an interactive map, a real-time event feed, and OAuth-based user authentication. This report outlines the motivation, system architecture, and implementation of the platform.**

***Index Terms*—Web Application; GraphQL; OAuth; Event discovery; geolocation; interactive maps; user interfaces; PostgreSQL.**

I. Introduction and Objectives

SocialSpot is a specialized social media application designed to revolutionize event discovery and social coordination. Unlike traditional platforms that treat events as secondary content, SocialSpot positions events as the primary focus, creating what we envision as "the Instagram of Events".

The current social media landscape lacks a dedicated platform for event-centric social interaction. While platforms like Instagram and Facebook excel at general content sharing, they fail to provide specialized features for event discovery and promotion. Event organizers struggle to reach target audiences effectively, while potential attendees often miss relevant events due to algorithmic limitations and content oversaturation. Existing solutions either focus solely on event management (Eventbrite) or bury event content within broader social feeds, creating inefficient discovery experiences. SocialSpot addresses this gap by integrating familiar social media mechanics—sharing, liking, and commenting—with specialized event discovery features. The platform's event-first architecture prioritizes event content throughout the user experience, from feed algorithms to interface design. Key innovations include map-based event discovery, allowing users to visualize opportunities spatially, and interest-based recommendations that connect users with relevant local events.

This report documents the comprehensive development of SocialSpot, covering architectural decisions, implementation strategies, and technical challenges encountered in building a production-ready social media platform. The current implementation demonstrates our approach to creating a specialized social network that serves the underexplored niche of event-focused social interaction, combining location-based discovery with traditional social media engagement patterns.

II. Related Work And State Of The Art

Event websites have become more popular in recent years. Services like Eventbrite, Meetup, or the event modules on Facebook allow users to create, publish, and discover events, based on various filters such as date, category, or location. These platforms typically integrate user accounts, attendance tracking, and interaction features such as likes or comments.

From a technological standpoint, interactive maps are a common method for visualizing geographically distributed events. Tools like Leaflet.js and Google Maps API are widely used in web applications to offer intuitive spatial browsing. To enhance performance and usability when displaying a large number of markers, clustering libraries like Leaflet.markercluster are also commonly used. These libraries aggregate nearby events into a single cluster marker, improving map readability and reducing performance issues.

Modern event-based applications also make extensive use of reactive frontend frameworks such as React, Vue, or Svelte in combination with GraphQL or REST APIs to dynamically fetch and render data. This architectural style allows for efficient client-side filtering and real-time updates, based on user interaction.

In addition to map-based views, most applications also include feed views to provide alternative ways of browsing events. These lists are often sorted by time, relevance, or proximity and can be filtered by user-defined criteria.

In contrast to these established platforms and techniques, our implementation focuses on simplicity, and an open, localized event interaction model. The concrete design choices and architecture of our application are described in the following sections.

III. Architectural Goals

Figure 1 shows SocialSpot's external systems and how the user interacts with them. To use SocialSpot to its full potential, users need a Google account to authenticate themselves. We

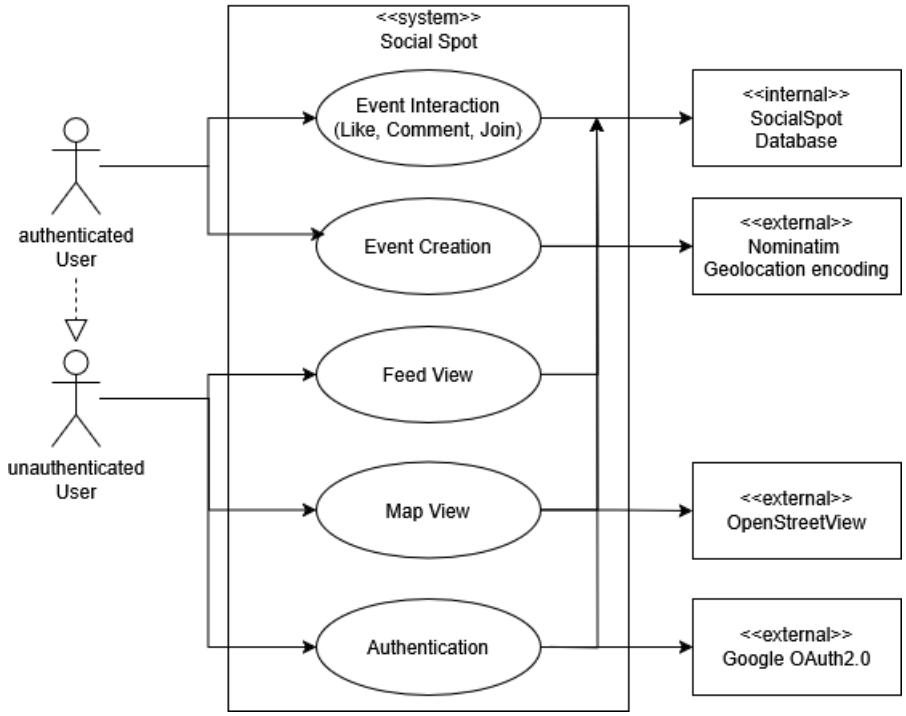

Figure 1. Visualization of Interactions and External Systems

want to provide an interactive and intuitive user interface that is also aesthetically pleasing and functional. We primarily used JavaScript for the backend as well as the Svelte framework for the frontend using TypeScript and JavaScript.

IV. Architecture of SocialSpot

A. Overall System

The base layer of the system is our PostgreSQL database, which stores important data like events and user-related information. Our Application Programming Interface (API) interacts directly with the database by querying and inserting data. The API layer itself consists of a RESTful API and a GraphQL API, both share the same middleware for session handling. The GraphQL API handles most user interactions, while the RESTful API is only used for authentication and image uploads. For authentication, we also rely on external services from the Google OAuth 2.0 API. The GraphQL API uses the Nominatim API for encoding addresses into geolocations for the map-related features of our website. Our frontend layer uses a Node.js server with Svelte and SvelteKit, which requires the OpenStreetView API for the map of our website. Lastly, we use NGINX as a reverse proxy to forward user requests to the desired service over HTTPS.

All our aforementioned layers are part of a single tier. Details on each one can be found in their corresponding section. Refer to Figure 2 for a visualization of the architecture. We loosely based our architecture on the architecture of the website StockSentinel [1].

B. Frontend

The frontend for SocialSpot was built using SvelteKit and consists of the following pages: Home, Map, Create Event, and User Profile. Additionally, we created a layout component that is shared across all pages to ensure consistent navigation throughout the website. For this navigation, we use Svelte's built-in routing system.

1) Home Page: The main feature of the Home page (see Figure 4) is the feed of events that the user can scroll through. Each event has its own box and can be clicked on to toggle a detailed view for that event. These features are created using the following components:

- *EventFeed:* This component's purpose is the arrangement of the events in a grid with two columns for the desktop, and one column for the mobile version. It receives the list of events from the home page and creates an EventBox for each event.
- *EventBox:* This component renders individual event cards within the feed. Each EventBox displays the following event information: image, title, date, time, location, and a truncated description. The component also includes several interactive features: a like button with real-time counter updates, a join/leave button for event attendance, and a display of the comment count. The like button and comment count are displayed over the image.
  The entire card is clickable and opens the corresponding EventDetailView when selected. To prevent event bubbling, the interactive buttons use the stopPropagation() method, which prevents the click event from bubbling up to the parent

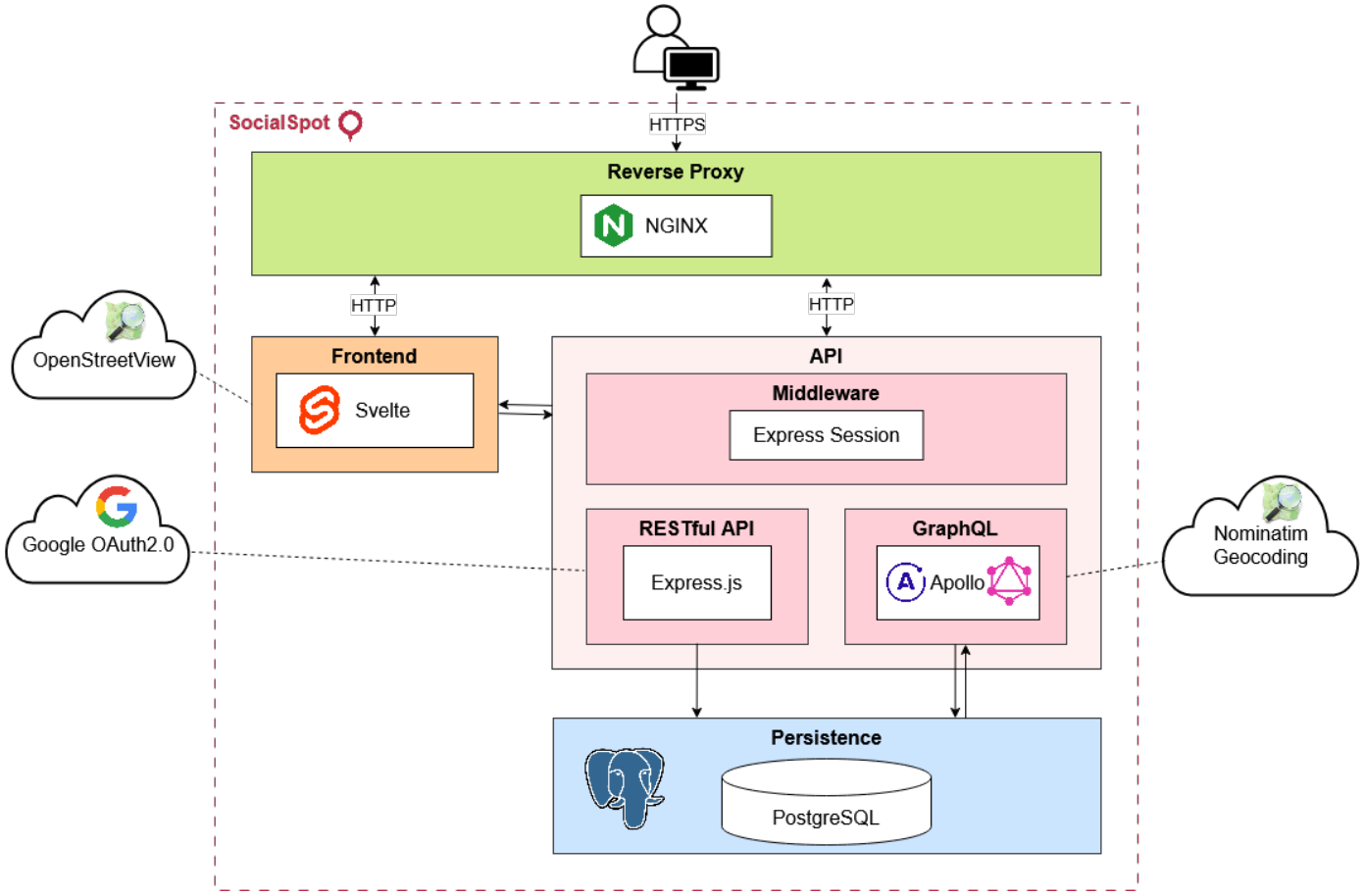

Figure 2. Architecture of SocialSpot

card element [2]. Otherwise, all the buttons would trigger the detail view unintentionally. The component utilizes Svelte's reactive stores for state management, automatically updating the UI when like or join status changes occur from other users or sessions.

- *EventDetailView:* This component renders an overlay that displays more detailed event information when an EventBox is clicked on. The component creates a full-screen overlay with a semi-transparent background that can be closed by clicking outside the event window or the x button in the top right corner. Once again, we use stopPropagation to prevent unintended closing when interacting with elements inside the detail view.

  The component displays an enlarged event image with the same overlay for liking and viewing comment counts as the home page. Below the image, the component shows the complete event details, including title, date, time, location, address, and full description. The interactive features are the same like and join/leave buttons as in the EventBox, with real-time updates through reactive stores. The reactive stores also mean that the like, comment, and join/leave functionality is consistent across the EventBox and EventDetailView. Without the stores, the EventBox would not update with the changes made in the EventDetailView.

- *eventInteractions:* To manage user interactions with events consistently across multiple components, we created this centralized store module that handles like and join/leave functionality. It creates reactive Svelte stores for each event's interaction state (liked status, joined status, like count, loading state) and provides toggle functions that handle both the UI updates and the GraphQL API calls. The module ensures data consistency between the frontend and backend while allowing multiple components to share the same interaction logic. This is used by the EventBox and EventDetailView components.

2) Map Page: The Map page (see Figure 5) allows users to explore all created events via an interactive OpenStreetMap interface. It is implemented as a reusable Svelte component and receives optional filter values for the user to search for events. This way, the component remains modular and does not require a different version for each use case. The map is implemented using Leaflet.js [3] [4] and contains additional features, like marker clustering for creating a heat map for events and a detail overlay to get more information for each event on the map. The Map page is composed of several Svelte components:

- *Map Component:* Displays the OpenStreetMap [5] view centered over Germany. It renders all events received from the

backend as map markers and automatically groups markers into clusters when zoomed out. This way, the user can see areas of events, just like a heat map.
- *EventFilter Component:* Allows users to filter events based on city name, title, or date of the event. The filter values are forwarded to the Map component and applied client-side after data fetching. The filter also includes a live search input for cities backed by the project's GraphQL API [6].
- *EventDetailView Component:* A full-screen overlay that is triggered when a marker is clicked. It shows detailed event information such as title, date, time, description, location, and an optional image. The component is the same as used on the Homepage.

The layout of the Map page is divided into two columns. On the left side, the Map is centered and scales dynamically to fit most of the screen. On the right, the EventFilter component is displayed in a separate column with vertically stacked input elements. Clicking on an event opens the overlay on top of the entire layout to see more information about the clicked event. By clicking on a heatpoint on the map, the map automatically zooms into the point and shows the events in the area or even more heatpoints.

3) Create Event Page: The CreateEvent page allows users to create their own events. When creating an event, the following information can be entered: event title, city, address, description, start date, start time, and optionally an image, which serves as a visual preview. Based on this data, it should be easy for other users to find suitable events in their area. To improve user experience, there is an automatic city name auto-complete feature, so that as little information as possible needs to be entered manually.

The page consists of the following parts:

- *Event Form*: This is where the main information is entered: event title, city, address, description, start date, and start time.
- *City Suggestion Input*: While typing, the program displays suggestions. With every additional letter typed, the suggestions update. This is intended to help users select the correct city as quickly as possible with minimal input. The selected city ID is then stored to provide accurate coordinates for the MapPage.
- *Image Upload*: Users can select an image file from their computer, which will be linked to the event. After selecting a file, the "Choose Image" button is replaced with a preview of the image. The image is uploaded separately, and the link is stored with the event.
- *Submission Dialog*: After clicking the "Create Event" button, the system checks whether all required data has been entered. A dialog then informs the user whether the event was successfully created.

4) User Profile Page: The User Profile page provides authenticated users with a personalized dashboard to manage their accounts and view their created events. The page implements conditional rendering based on authentication status. That means it displays different content for logged-in users, unauthenticated users, and during loading. For unauthenticated users, the page presents a Google OAuth login button that redirects to the backend authentication endpoint. Once logged in and authenticated, the page displays the user's profile information, including name, email, and profile picture, which are retrieved through a GraphQL query to the myUser endpoint.

The main feature of the authenticated view is a section showing all events created by the current user, fetched via the getCreatedEvents GraphQL query, and displayed using the same EventFeed component as the home page. This ensures consistent event presentation and interaction across the website. A logout button allows users to terminate their session and return to the login state.

C. Backend

1) NGINX: Since all applications live in the same Docker network, the web and backend servers are not exposed to the outside world. When the user interacts with the app, the requests are handled by a NGINX [7] webserver instead. NGINX acts as a reverse proxy and forwards the user requests to the corresponding backend or frontend container. It also allows easy integration of SSL certificates, so the user traffic is securely encrypted with HTTPS. NGINX also allows managing content caching or acting as a load balancer; however, the latter is not used in our case.

2) Sessions: Since our application has interactions that require user authentication, the state of the users has to be remembered even when they switch or reload a page. The most common solution to this problem is to use either JSON Web Tokens or Session-based Authentication. In our case, we use the latter approach with express-session as a middleware, which is used by the RESTful express server and the Apollo-GraphQL server. On successful authentication, the user session is stored, and the client receives a cookie with the signed session ID. On subsequent requests, the client's browser automatically sends the cookie back to the server. All endpoints that require authentication access the session and check if the user is logged in. When the user logs out, the session is destroyed. [8]

3) Authentication: OAuth 2.0 is a standard for authorizing a web application to access resources from a different app [9]. We use the OAuth 2.0 protocol to provide user authentication which is handled by Google in our case. Before using the protocol, the application has to be registered with the OAuth provider. During registration, we obtain a client_id and a confidential client_secret. We also have to register the redirect_uri and application origin. Both have to be secured by HTTPS and must match the actual URIs; otherwise, the request will be blocked for security reasons. [10]

When a user wants to log in to our application, the web server redirects the user to the Google authorization endpoint. Once Google has verified the authorization request, the user is sent back to the specified redirect_uri of our web server. There, our server fetches the user's identity from the Google API with the authorized access token.

Then the acquired data has to be matched with the database. For simplicity, login and registration are essentially the same

operation right now. So the database logic becomes an upsert query in which the acquired user data from the OAuth provider is inserted if the user does not exist and updated otherwise. At the end of the authentication process, user data is stored in the session.

4) GraphQL: For the backend of SocialSpot, we chose GraphQL [11] as the primary API interface between the frontend and backend services. GraphQL offers a flexible and efficient alternative to REST, allowing the client to request exactly the data they need. This efficiency is critical for SocialSpot's event-centric, user-driven architecture.

- *Schema Overview:* The GraphQL schema forms the contract between frontend and backend, defining three key types: User, City, and Event.
  The User type represents a registered individual on the platform, including fields such as name, email, and an optional profile Picture. City objects allow events to be geo-located, supporting queries to search by name or partial match.
  The Event type is central to SocialSpot's function, encompassing metadata like title, description, date, and geographic coordinates (latitude and longitude). To support interactivity, fields such as likeCount, likedByMe, attendCount, and attendedByMe are included, providing essential state data for personalized user views.
- *Queries and Mutations:* GraphQL queries in SocialSpot allow clients to retrieve lists of events (eventList), filter for user-created events (getCreatedEvents), or access user-specific data (myUser). Events are enriched with dynamic, real-time context, such as whether the current user has liked or is attending them, determined by user sessions managed via express-session.
  The GraphQL Mutation type enables interactive features essential for a social platform: creating new events (createEvent), managing event attendance, liking or unliking events, and posting comments. Each mutation requires authentication, ensuring that only logged-in users can perform these actions.
- *Backend Implementation:* On the backend, the schema is implemented using Apollo Server [12], integrated with express to serve the /graphql endpoint. The resolver logic connects the GraphQL operations to a PostgreSQL database, using pg for data persistence. Notably, the system includes geocoding support via OpenStreetMap's Nominatim API [13]. When creating an event, the provided address and city are geocoded into latitude and longitude, allowing for spatial queries and map displays on the frontend.
  The GraphQL architecture provides a robust, type-safe, and developer-friendly environment. It enables future scalability for SocialSpot, such as adding new types like GroupEvent or supporting nested querying for social interactions like threaded comments.

D. Persistence

For persistence, we use a relational PostgreSQL database. Our application has to store the user and event data together with the author, likes, comments, and attendees. We also store a small dataset of German cities for autocompletion and guiding the user to enter the correct city name. It's also planned to enable a friendship feature between users. The resulting schema is depicted in Figure 3. The extensions postGIS [14] and pg_trgm [15] allow for efficient geospatial queries and text similarity search with trigram matching. However, due to time constraints, our application does not make use of them yet.

E. Docker and Deployment

The services of SocialSpot run in isolated Docker containers. Docker Compose is used to bundle the different services and start them in the correct order. It also sets the environment variables required for each service. The containerization allows for easy deployment. In our case, we host Social-Spot as a single-node application on a Hetzner CX22 server.

V. Impediments

The development of SocialSpot gave us extensive insights into the planning and implementation of a fully-featured web application. Some features turned out to be significantly more difficult to implement than initially expected. In particular, the interaction between OAuth authentication, session management, reactive UI, and geodata processing presented several challenges.

Collaboration between backend and frontend developers also proved difficult at the beginning due to a lack of communication. However, this improved significantly over the course of the project, and we began coordinating more regularly and in greater detail. Clear task separation and regular code reviews helped to improve our workflow and reduce integration issues.

In our opinion, the website has achieved its goals, but as always, there is room for improvement. Also, the overall stability and usability of the platform at this stage met our expectations. Teamwork functioned well throughout the project, and the weekly meetings and milestones were mostly met successfully.

VI. Conclusion and Future Work

A. Possible Improvements

Currently, user sessions are saved in the default express session storage. According to express, this is not recommended for production environments as it will leak memory under most conditions and does not scale past a single process. Also, the session state is lost once the server has been shut down. A better solution would be to use a dedicated in-memory database like Redis to store the session data.

At the moment, the image upload is handled by a REST endpoint, which is embedded in our single backend service. The images are stored locally on the file system. In the future, we would change this to a more modular approach by separating the image upload into its own dedicated service. However, it is required to use the aforementioned session database, since we would then need to share the session between different services.

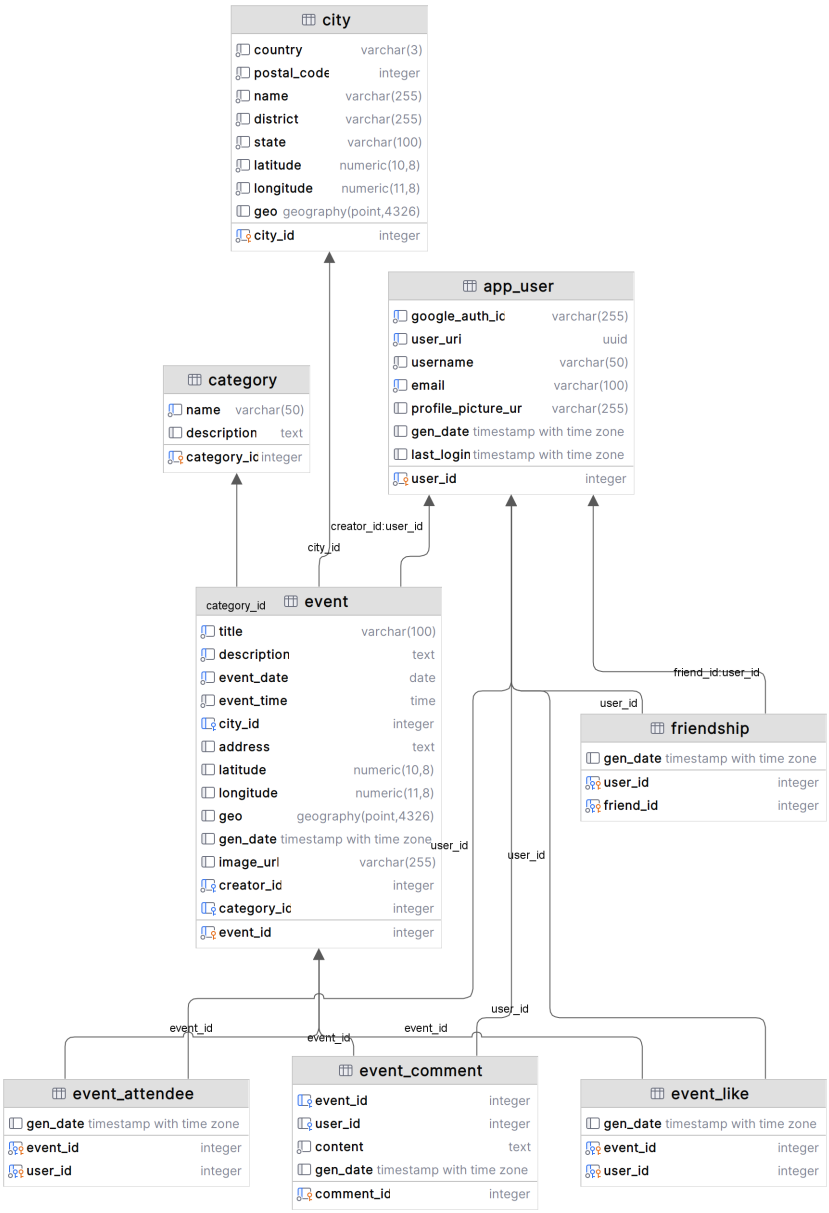

Figure 3. Database Visualization

We could also consider using MinIO, which is an open-source, S3-compatible object storage system that would provide better scalability, data redundancy, and seamless cloud integration compared to local file system storage.

Map and Feed View currently load all events that are stored in the database. This can lead to overfetching. To address this issue, we should implement pagination to load only a limited number of events per request rather than retrieving all events simultaneously, which significantly improves scalability and reduces server load.

B. Additional Features

There are additional features that can be added to the website for higher user comfort. Adding a filtering feature to the home page would make it easier to find events relevant to the user. We would allow filtering by date, like count and distance. In addition to those filters, we could add categories that the user can sort the events by. This could include categories like musicals, parties, festivals, and so on.

A feature for more interaction between users would be to add certain friend features. This would include a friend list with the ability to add and remove other users. This would then enable the user to see what events their friends have liked, commented on, or joined.

REFERENCES

[1] Andreas Hecht, Linus Heise, Oliver Kneidl, Eva-Maria Maurer, and Christoph P. Neumann. *StockSentinel: AI-Powered Web*

Tool for Analyzing the Markets Perception of Stocks. Technical Reports CL-2024-07. Ostbayerische Technische Hochschule Amberg-Weiden, CyberLytics-Lab an der Fakultät Elektrotechnik, Medien und Informatik, July 2024. DOI: 10.13140/RG.2.2.20457.30564.

[2] W3Schools. *stopPropagation() Event Method*. W3Schools. URL: https://www.w3schools.com/jsref/event_stoppropagation.asp (visited on 06/21/2025).

[3] Leaflet.js Contributors. *Leaflet.js – JavaScript library for interactive maps*. 2025. URL: https://leafletjs.com (visited on 06/12/2025).

[4] Leaflet.markercluster Contributors. *Leaflet.markercluster – Marker clustering plugin for Leaflet*. URL: https://github.com/Leaflet/Leaflet.markercluster (visited on 06/12/2025).

[5] OpenStreetMap contributors. *OpenStreetMap – The free wiki world map*. URL: https://www.openstreetmap.org/ (visited on 06/12/2025).

[6] GraphQL Request Contributors. *graphql-request – Minimal GraphQL client for Node and browsers*. URL: https://github.com/jasonkuhrt/graphql-request (visited on 06/12/2025).

[7] NGINX. *nginx*. NGINX. URL: https://nginx.org/en (visited on 06/21/2025).

[8] expressjs. *Express session middleware*. expressjs. URL: https://expressjs.com/en/resources/middleware/session.html (visited on 06/22/2025).

[9] Dick Hardt. *The OAuth 2.0 Authorization Framework*. URL: https://datatracker.ietf.org/doc/html/rfc6749 (visited on 06/22/2025).

[10] Google. *Using OAuth 2.0 to Access Google APIs*. Google. URL: https://developers.google.com/identity/protocols/oauth2?hl=de (visited on 06/22/2025).

[11] GraphQL. *GraphQL A query language for your API*. Meta Platforms Inc. URL: https://graphql.org/ (visited on 06/22/2025).

[12] Apollo GraphQL. *Introduction to Apollo Server*. Apollo GraphQL. URL: https://www.apollographql.com/docs/apollo-server (visited on 06/22/2025).

[13] Nominatim. *Overview*. Nominatim. URL: https://nominatim.org/release-docs/develop/api/Overview/ (visited on 06/22/2025).

[14] PostGIS. *About PostGIS*. PostGIS. URL: https://postgis.net/ (visited on 06/22/2025).

[15] Alexander Korotkov Oleg Bartunov Teodor Sigaev. *F.33. pg_trgm — support for similarity of text using trigram matching*. URL: https://www.postgresql.org/docs/current/pgtrgm.html (visited on 06/22/2025).

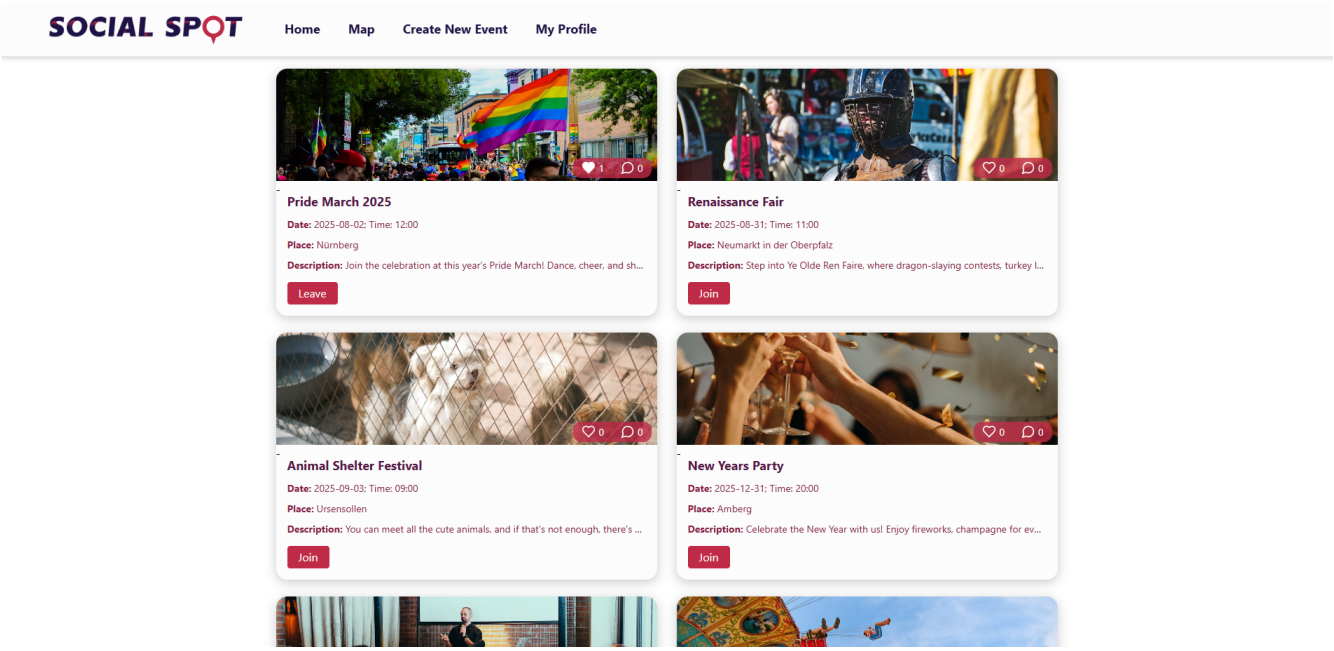

Figure 4. Home Page of SocialSpot

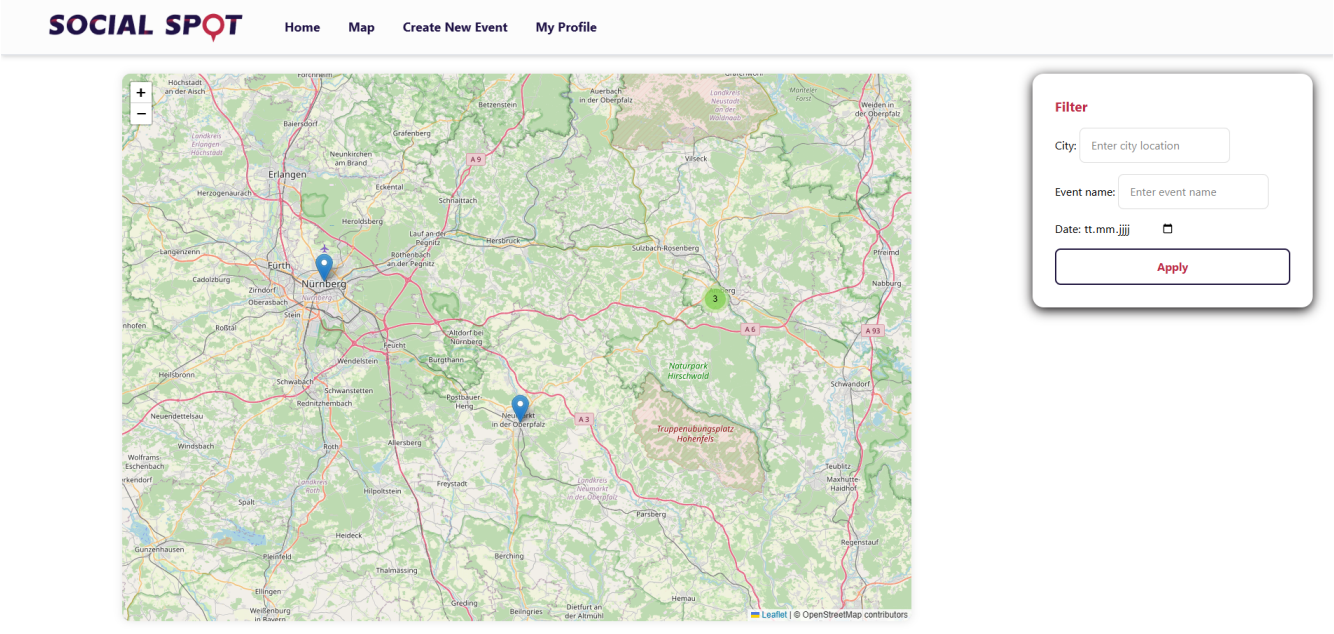

Figure 5. Map Page of SocialSpot