Schedulo: Self-Hosted Meeting Scheduler

Lorenz Aichner, Sarah Hörtnagel, Hagen Leitner, and Fabian Szakács

706.041 Information Architecture and Web Usability 3VU WS 2025/2026
Graz University of Technology

03 Feb 2026

Abstract

This report presents Schedulo, a self-hosted web application for scheduling meetings, developed as an open-source alternative to existing tools such as Doodle. Within the application, organisers can create polls with different time options and collect participant availability without requiring user registration, addressing the growing need for privacy-conscious scheduling solutions especially in academic and organisational contexts.

Schedulo is built using modern web technologies including Next.js, TypeScript, and React. The application features a type-safe API architecture, secure authentication for organisers, and a responsive user interface designed for simplicity and ease of use. It is designed to be self-hosted through Docker containerisation and file-based data storage, eliminating dependencies on external databases or cloud services.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

Coordinating schedules among multiple people is a common challenge in both working and scientific environments. Whether organising a team meeting, academic conference or group presentation, finding a time that works for everyone can be surprisingly difficult and time-consuming. Established tools like Doodle [Doodle 2026], When2Meet [When2Meet 2026], or Framadate [Framasoft 2026] have become popular solutions to this problem. However, they come with significant drawbacks, such as reliance on third-party hosting, data privacy concerns, or intrusive advertisements. For organisations and individuals, it is not secure to handle sensitive information on a hosted solution. It is important to make self-hosted alternatives available to give users full control over their data. Schedulo was developed to address these issues by providing a lightweight, privacy-friendly alternative which focuses on ease of use and can be deployed on one's own infrastructure.

The goal of Schedulo is to implement a simple web-based tool to schedule meetings. It should be built on modern standards with privacy at its core. A central design decision is that participants should not need to create accounts to vote in polls. Instead, they receive unique, secure URLs that enable them to participate immediately without any registration barriers. With this, the key objectives of this project are:

- *Data Control*: Enable deployment within a private or trusted cloud environment. This would give users full control over their scheduling data without depending on insecure, external providers.
- *User Experience*: Provide a clean, responsive, modern user interface, which works seamlessly across different devices.
- *Minimalism*: Reduce the collection of personal data to the absolute minimum required for the core functionality.
- *Technical Integrity*: Implementing a robust, full-stack JavaScript environment using Next.js, Node.js, and tRPC to ensure end-to-end type safety and reliability.

Chapter 2

Existing Meeting Schedulers

Meeting schedulers are widely used tools which support groups in finding suitable time slots for meetings or events. They typically allow an organiser to propose multiple options, which participants can then evaluate by indicating their availability. While such tools are popular and convenient, they differ significantly in terms of deployment model, privacy guarantees, usability and administrative control.

Many existing solutions are offered as centralized web services, which simplifies usage but often comes at the cost of reduced control over data and limited customisation options. In contrast, self-hosted alternatives aim to provide greater autonomy and privacy, but may lack the usability or polish of commercial platforms. This chapter briefly reviews two representative meeting schedulers: Doodle and Rallly.

2.1 Doodle

Doodle is one of the most well-known meeting scheduling platforms and is widely used in both professional and private contexts [Doodle 2026]. Figure 2.1 shows the Doodle poll creation interface, which allows an organiser to enter event details such as title, description, and location. Figure 2.2 shows the Doodle time selection interface, where an organiser can define multiple potential meeting slots. The organiser then shares a link with the participants, who are able indicate their availability without creating an account. This low barrier to entry contributes significantly to Doodle's popularity.

However, Doodle is offered exclusively as a centrally hosted service. Users have no control over where their data is stored or how it is processed, which may raise concerns in privacy-sensitive environments. In addition, advanced features such as administrative controls, branding options, and extended privacy settings are typically restricted to paid plans. While Doodle provides a modern and polished user interface, its closed-source nature and lack of self-hosting support limit its suitability for users seeking full data ownership or deployment flexibility.

2.2 Rallly

Rallly is a modern, open-source meeting scheduler which focuses on simplicity and privacy [Rallly 2026]. Similar to Doodle, it allows organisers to create polls and allows participants to vote without requiring user accounts. Rallly distinguishes itself by offering a clean and contemporary user interface, while remaining lightweight and easy to use. The poll creation interface of Rallly can be seen in Figure 2.3.

Unlike many commercial platforms, Rallly can be self-hosted, giving administrators greater control over deployment and data storage. This makes it an attractive option for privacy-conscious users and organisations. However, administrative features such as structured user management and role separation are limited and the system is primarily designed for small-scale or informal use cases. As a result, Rallly strikes a balance between usability and openness, but does not fully address scenarios requiring stronger administrative oversight or configurable access control.

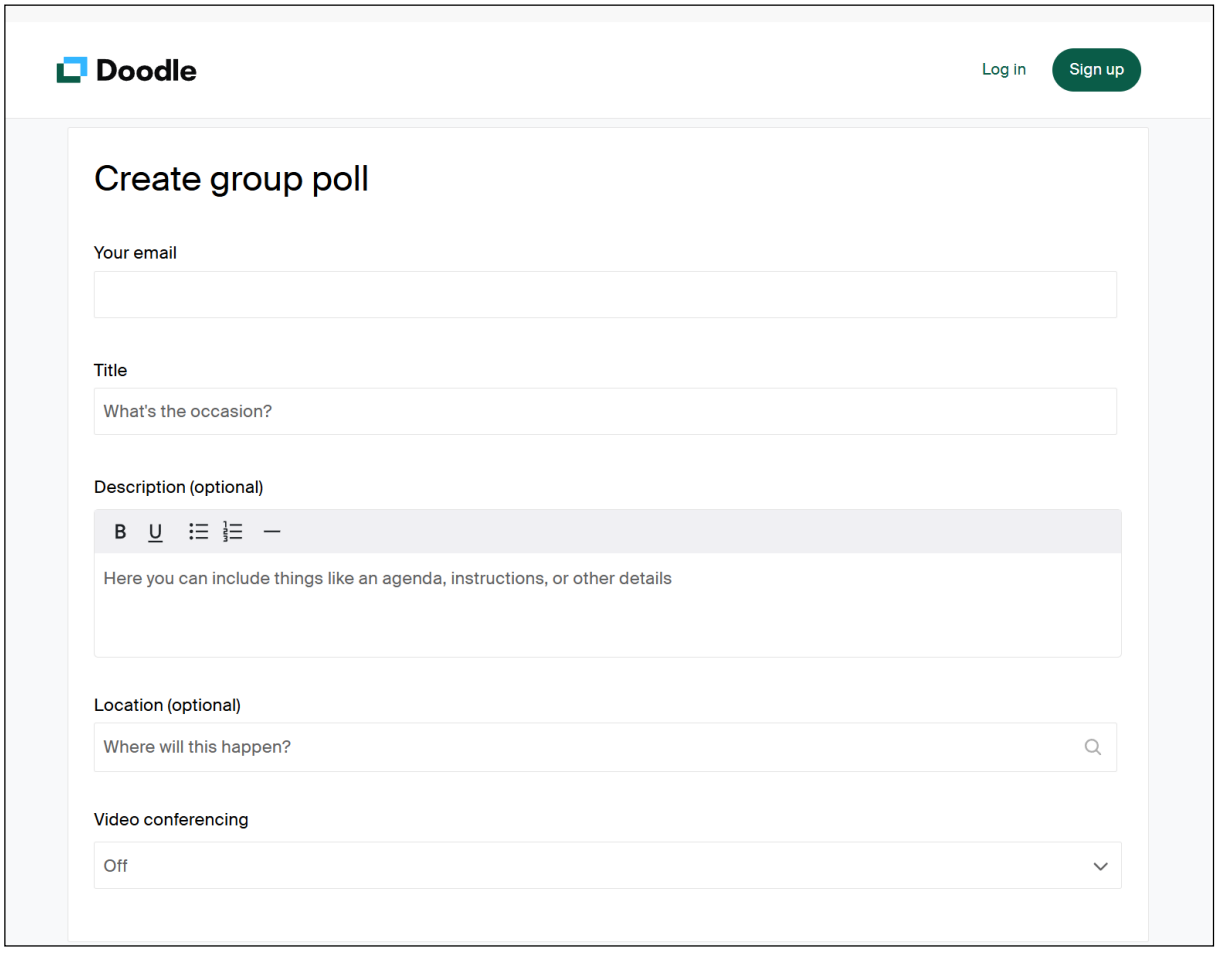

Figure 2.1: Doodle: The poll creation interface, showing event details such as title, description, and location. [Screenshot taken by Hagen Leitner.]

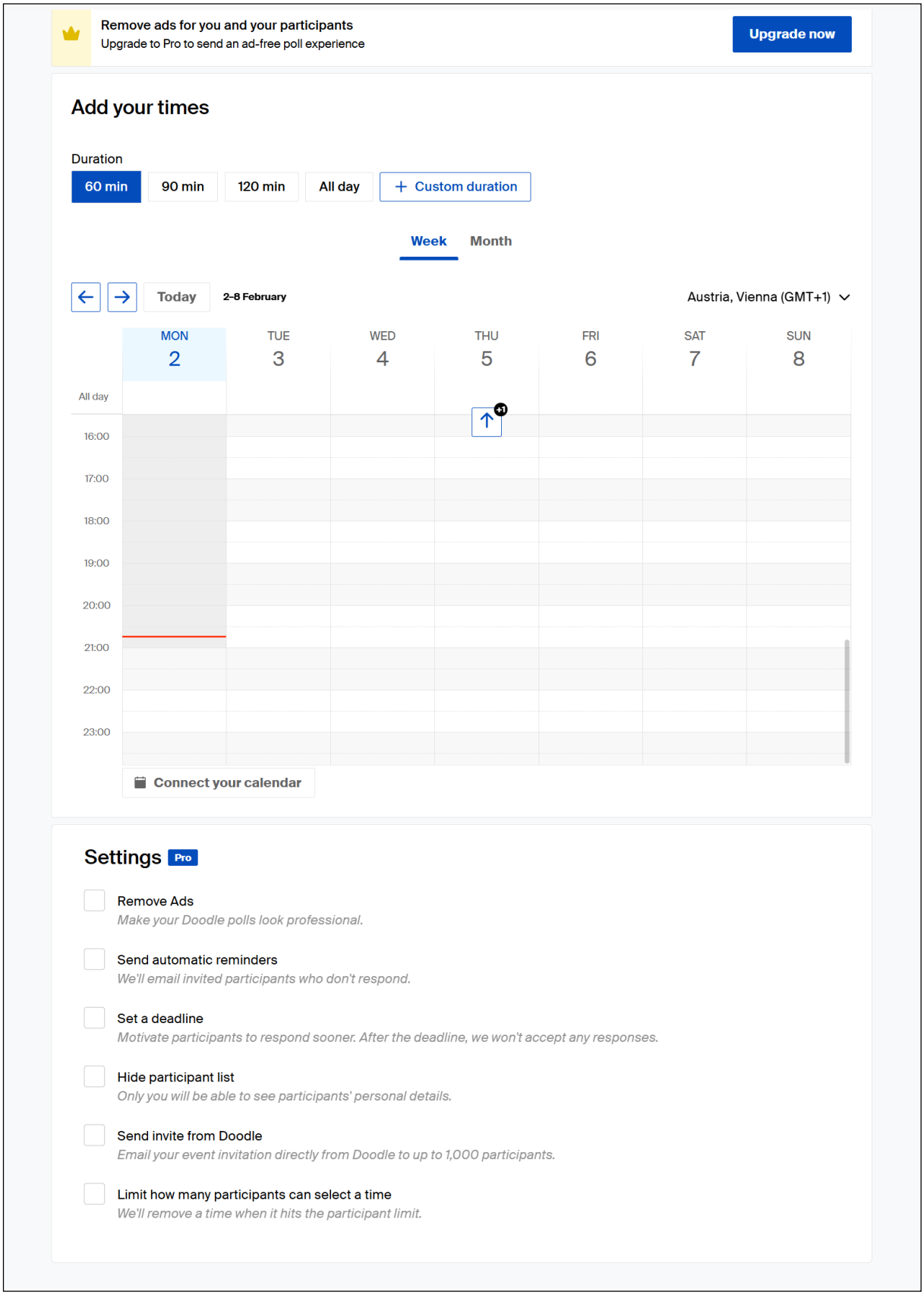

Figure 2.2: Doodle: The weekly time selection interface, letting organisers define multiple potential meeting slots. [Screenshot taken by Hagen Leitner.]

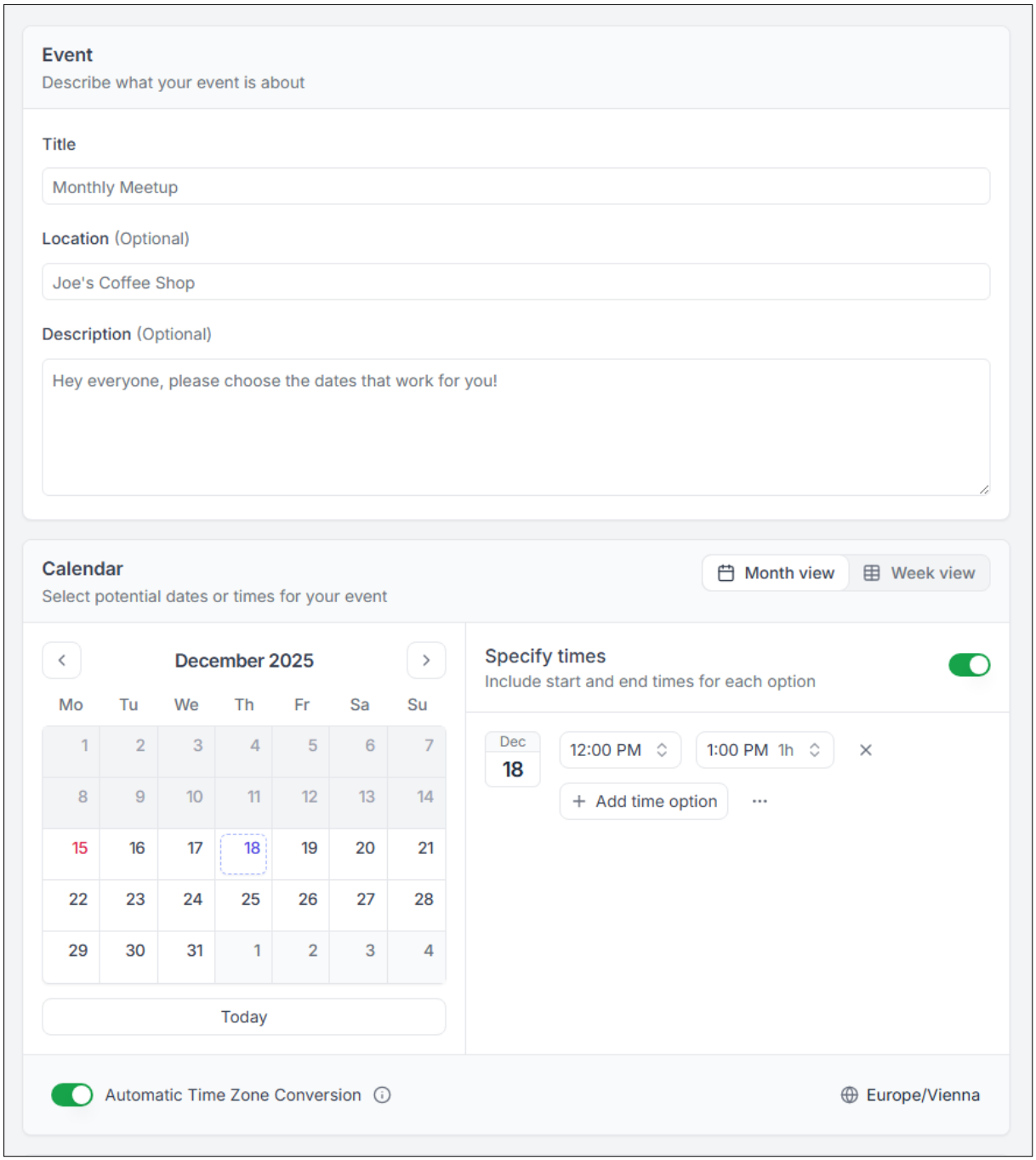

Figure 2.3: Rallly: The poll creation interface, including event metadata and calendar-based time selection. [Screenshot taken by Fabian Szakács.]

Chapter 3

Schedulo

Schedulo is a lightweight, self-hosted meeting scheduler designed for minimum effort and maximum comfort [Aichner et al. 2026]. The application is built upon the Next.js [Vercel 2025] framework, with Tailwind CSS [Tailwind 2026] and Mantine [Mantine 2026] components for the frontend, and tRPC [tRPC 2024] for end-to-end type-safe APIs.

3.1 System Requirements

The system requirements for Schedulo are derived from the goal of providing a lightweight, self-hosted alternative to existing meeting schedulers. Table 3.1 summarises the key criteria and highlights how Schedulo compares to related tools. The system requirements for Schedulo include:

- *Self-Hosted Deployment*: Schedulo should be self-hosted, giving full control and authority to the administrator as well as getting rid of unnecessary clutter such as advertisements. The administrator should be able to choose where to deploy the application and retain full ownership of all stored data. This approach allows the operation within private servers or trusted cloud environments, ensuring compliance with data protection requirements.

- *Accountless Participant Access*: Participants should be able to indicate their availability without the need to create or manage user accounts. Access to a meeting should solely be provided through a unique meeting URL shared by the organiser, removing the requirement for authentication or registration. This design simplifies the user experience and reduces the amount of personal data processed by the system.

- *Privacy-Friendly Design*: Minimising the collection and storage of personal data is essential. No tracking, analytics or third-party integrations should be required for the core functionality of the application. All data should remain under the control of the administrator.

- *Modern User Interface*: The system should provide a modern, clean, and responsive user interface. The interface should prioritise clarity and ease of use, so that users can propose meeting times, share meeting links and submit availability with minimal effort. In addition, the interface should adapt to different screen sizes and device types, including desktop and mobile environments.

- *Administrative User Management*: The application should provide a simple administrative interface for managing organiser accounts. A single administrator account should be responsible for creating, updating, and removing organiser access. Once access is granted, organisers should be able update their own password in the settings.

Feature	Doodle	DuD-Poll	When2Meet	Framadate
Self-Hosted	✗	✓	✗	✓
No Participant Accounts	✓	✓	✓	✓
Privacy-Friendly	✗	✓	✓	✓
Modern UI	✓	✗	✗	✓
Admin User Management	✓	✗	✗	✗

(a) Scheduling Tools (Part 1)

Feature	Nextcloud	Croodle	Rallly	Schedulo
Self-Hosted	✓	✓	✓	✓
No Participant Accounts	✗	✓	✓	✓
Privacy-Friendly	✓	✓	✓	✓
Modern UI	✓	✗	✓	✓
Admin User Management	✓	✗	✗	✓

(b) Scheduling Tools (Part 2)

Table 3.1: Feature comparison between Schedulo and seven related scheduling tools: Doodle [Doodle 2026], DuD-Poll [TU Dresden 2026], When2Meet [When2Meet 2026], Framadate [Framasoft 2026], Nextcloud [Rosenkranz et al. 2026], Croodle [Systemli 2026], and Rallly [Rallly 2026].

3.2 User Roles and Features

The application is structured around three user roles (access levels): Administrator, Organiser, and Participant. Each role is provided with functionality tailored to its specific use cases and responsibilities. The following subsections describe the features available to each access level.

3.2.1 Administrator

The Administrator dashboard serves as the central control point for managing organiser accounts in the application. A single admin account with a preset default password can access the Administrator dashboard to create, update, and delete organiser accounts. The Administrator dashboard displays all registered organisers in a table, each entry showing the organiser's username and account creation date, as can be seen in Figure 3.1. The admin can now add new organiser accounts, by simply providing a username and password for each new organiser. User management features also allow editing organiser accounts. With this, the admin is able to update existing organiser credentials, including modifying usernames or resetting passwords in case of organisational changes or security concerns. To prevent accidental account deletion, the system requires confirmation before removing any organiser account, which makes sure that administrative actions are intentional and carefully considered. This design choice also aligns with the Error Prevention principle from Nielsen's 10 usability heuristics [Andrews 2025, pages 93–94].

3.2.2 Organiser

Organisers with the role Organiser have comprehensive poll management capabilities. After authentication using valid organiser credentials, organisers are presented with a personalised dashboard showing all of their created polls, each marked with a visual indicator showing whether the poll is currently active and accepting votes, temporarily paused, or permanently closed. Organisers can also log out at any time or change their password from their account settings. The Organiser dashboard is shown in Figure 3.2.

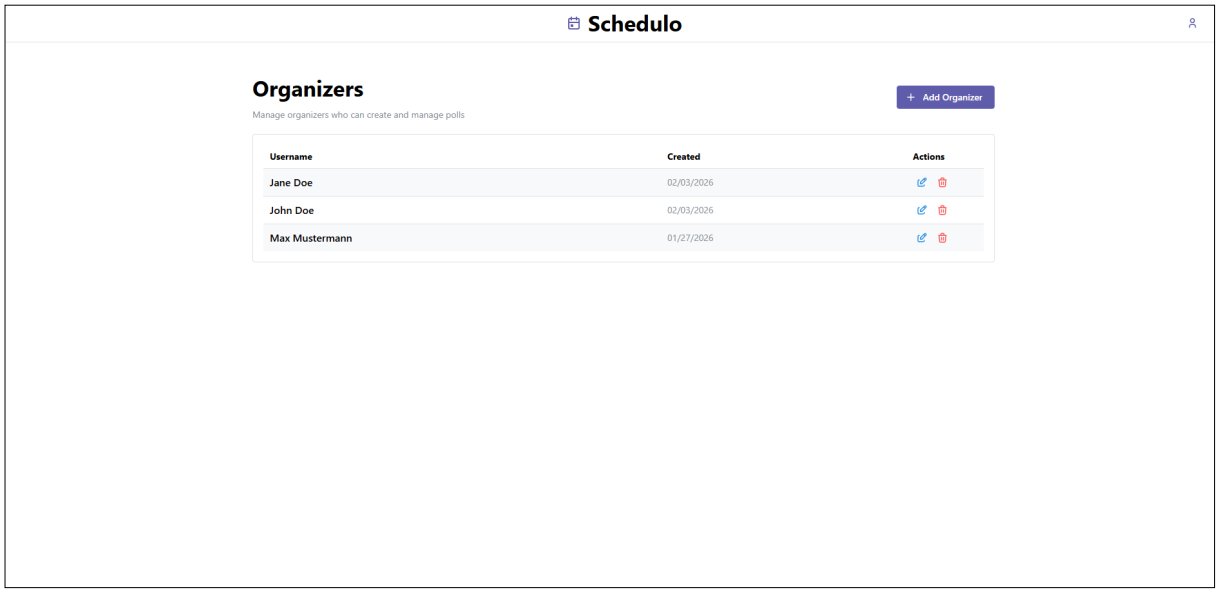

Figure 3.1: Administrator: Dashboard showing all existing Organiser accounts. [Screenshot taken by Sarah Hörtnagel.]

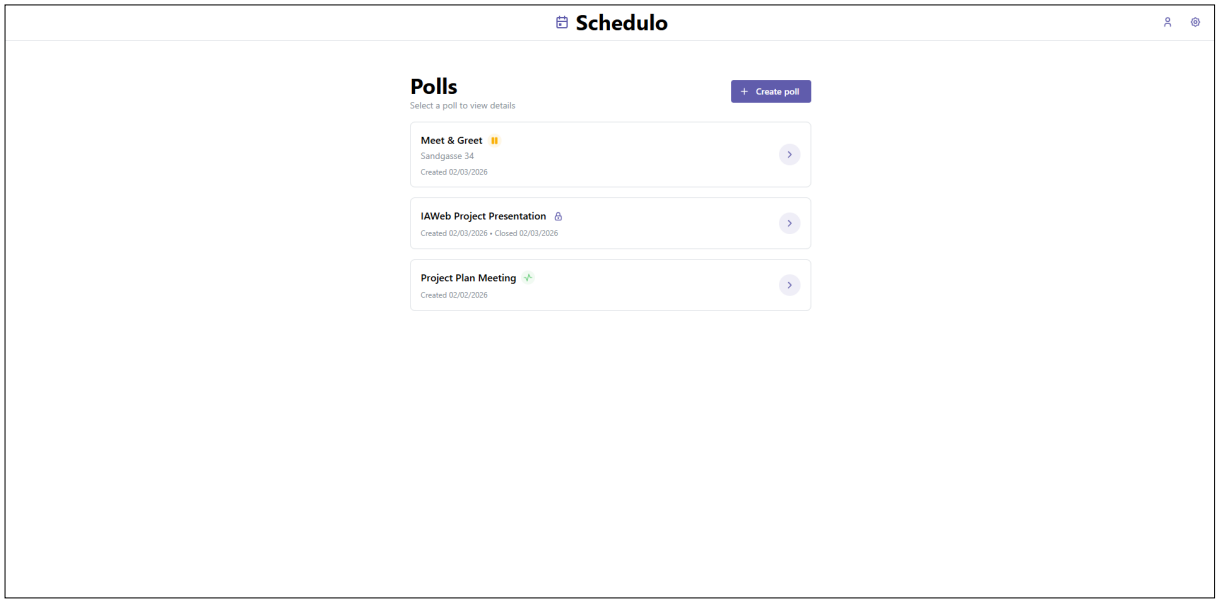

Figure 3.2: Organiser: Dashboard showing all created polls with status indicators. [Screenshot taken by Sarah Hörtnagel.]

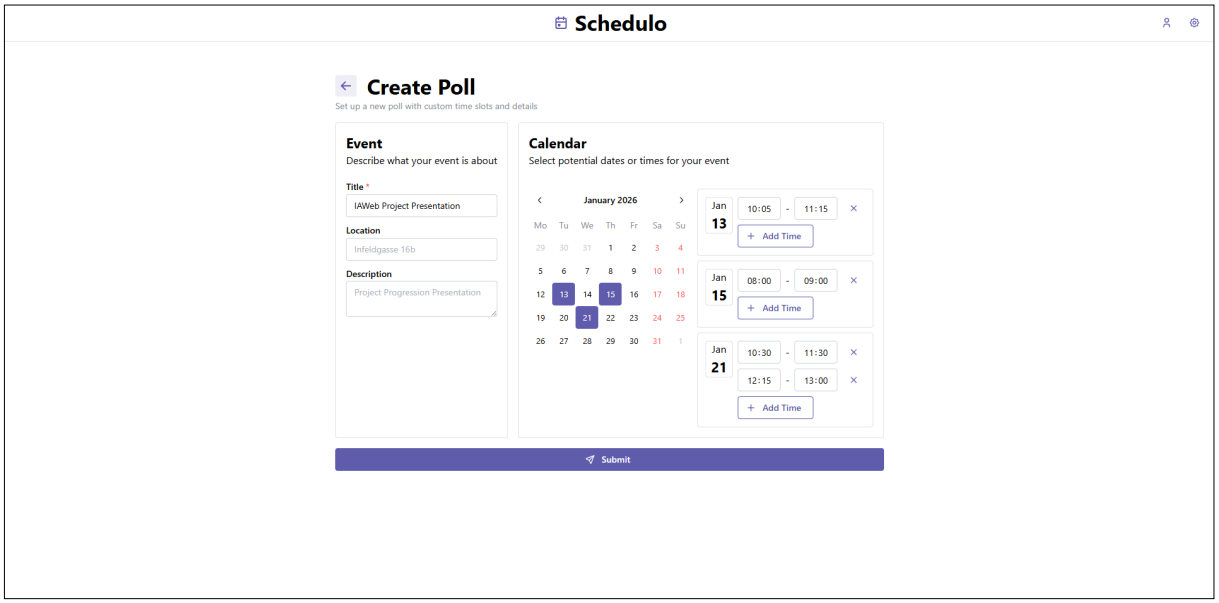

Figure 3.3: Organiser: Interface for creating a new poll. [Screenshot taken by Sarah Hörtnagel.]

Creating a new poll follows an intuitive workflow, as can be seen in Figure 3.3. An organiser creates a new poll by specifying a title, an optional location, and an optional description, along with the proposed time slots defined by date, start time, and end time. Additionally, the entered time slots are validated by the system, to prevent invalid time ranges where start times occur after end times. Once all details are entered, the organiser can submit the poll and is redirected to the results page of the new poll. After launching the poll, it is possible to generate a shareable voting URL for a poll which can be distributed to participants through external communication channels.

The results page provides organisers with detailed analytics of participant responses, as shown in Figure 3.4. If the poll already received user votes or the organiser manually selected a winning time slot, the current best time slot is shown at the top of the page. In addition, a detailed time slot summary table is provided, showing the vote counts categorised by the three voting options Yes, If Need Be and No for each proposed time slot. By default, this table is sorted in descending order of the number of Yes votes as the primary criterion and If Need Be votes as the secondary criterion. These results can also be re-sorted by date or a vote type in ascending or descending order to identify alternative possible meeting times, which can then be selected as the winning option by the organiser.

Furthermore, a user voting matrix is shown, where each row corresponds to a proposed time slot, showing the date and time range, while columns represent individual participants. The intersection of these rows and columns displays each participant's vote, whether they selected Yes (to indicate full availability), If Need Be (to signal conditional availability), or No (for unavailability). Participant comments are also displayed in the voting matrix table, offering additional context which might influence the final decision.

Beyond viewing results, organisers can perform several administrative actions on polls. Active polls can be paused to temporarily stop accepting new votes, reactivated after being paused, permanently closed to finalise results, or reopened to accept additional votes if needed. If a poll needs adjustments, the edit function allows modification of details or time slots, while preserving all existing votes. When necessary, organisers can even delete individual participant responses or remove entire polls from the system, with confirmation dialogues protecting against accidental deletions.

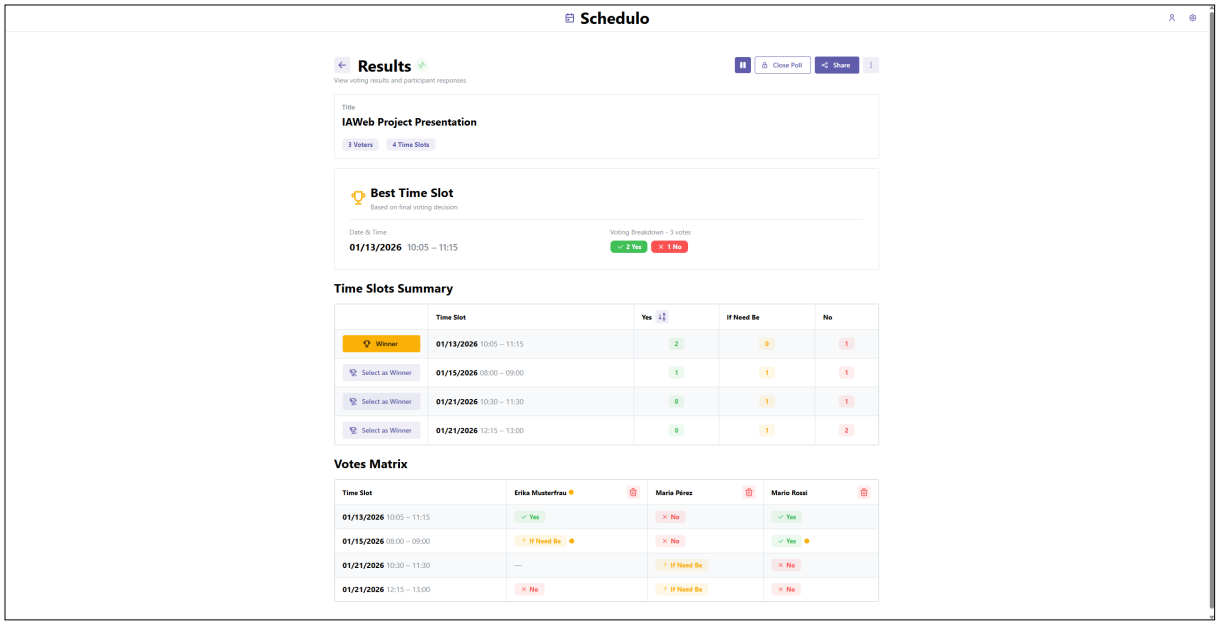

Figure 3.4: Organiser: Poll results page showing vote counts, time slot summary, and participant voting matrix. [Screenshot taken by Sarah Hörtnagel.]

3.2.3 Participant

Participants can vote in polls without creating accounts or logging in, which significantly lowers the barrier to participation. Accessing a poll through the shared URL from the organiser presents participants with the poll details including title, location, and description, along with a calendar interface displaying all proposed time slots. The corresponding interface is shown in Figure 3.5.

For each proposed time slot, participants can indicate their availability by selecting one of the three availability options. Participants may also choose not to vote for time slots. Before submitting their votes, a participant must enter their name, which is used to identify responses in the results table and to prevent completely anonymous votes. Additionally, a comment field allows participants to add contextual remarks regarding the overall meeting or just specific time slots, such as suggesting alternative times not listed or noting preferences which might help the organiser make a final decision. Participants can then submit their votes with a single click, upon which a confirmation dialogue is shown to indicate successful submission and to inform participants that the organiser will communicate the final meeting details after a decision has been made.

Once the voting phase of a poll is over and the poll has been closed by the organiser, participants can access the participant results page through a result link provided by the organiser. This is shown in Figure 3.6. It is a simplified version of the organiser's results page, showing only the general meeting information and the final selected meeting slot for the poll. If a participant attempts to access a poll which has been temporarily paused by the organiser, they receive a notification explaining that the poll is currently not accepting votes, along with a button to return to the home page. These state-aware messages ensure participants always understand the current status of any poll they try to access, which is again consistent with the principles outlined in Nielsen's usability heuristics [Andrews 2025, pages 93–94].

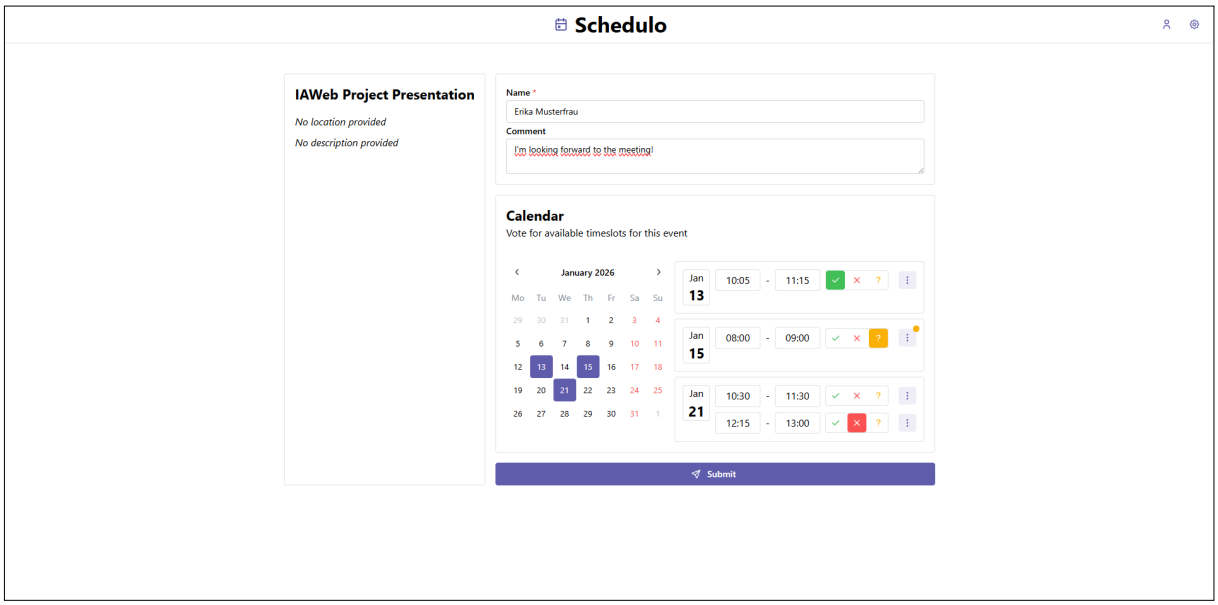

Figure 3.5: Participant: Voting interface allowing participants to indicate their availability for each proposed time slot in a poll. [Screenshot taken by Sarah Hörtnagel.]

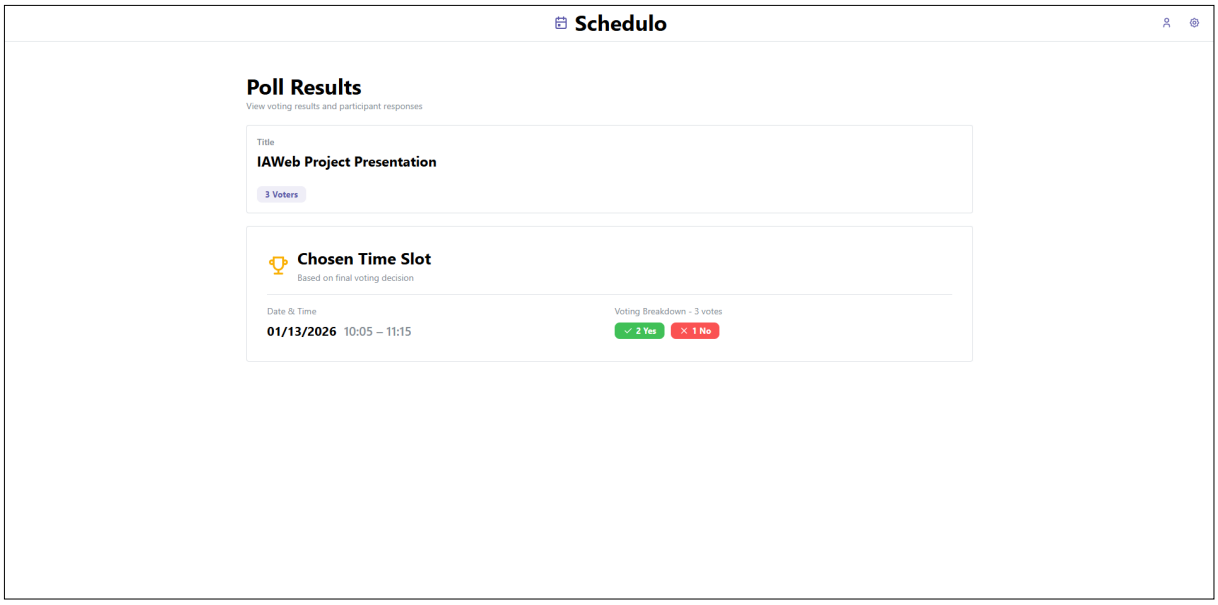

Figure 3.6: Participant: Simplified poll results page for participants, showing the finalised meeting time and general poll information. [Screenshot taken by Sarah Hörtnagel.]

Chapter 4

Technical Realisation

Schedulo is a full-stack TypeScript-based implementation, based on a simple client-server architecture, as shown in Figure 4.1. Users communicate via HTTPS with the frontend, which then sends API requests to the backend. The frontend is built with Next.js (React), Tailwind, and Mantine. The backend is implemented in Node.js. Docker is used for deployment.

4.1 Frontend

Schedulo uses Next.js, a React-based frontend framework. The UI components are built with Mantine and the layout is styled with Tailwind CSS.

4.1.1 Next.js

Next.js provides a structured foundation and a wide range of features for building production-ready web applications with React [Vercel 2025]. As Rupareliya [2025] explains, some of the key benefits of using Next.js for a web project include:

- *Rendering Performance*: Next.js supports server-side rendering and static site generation. This allows faster page loading, improving the overall user experience compared to purely client-rendered single-page applications.
- *Page Routing*: Next.js offers a file-based routing system that maps application routes directly to the project directory structure. This simplifies route management, making it easier and more effective for developers to create dynamic routes. Routes for specific polls, such as `/poll/[id]`, are essential for generating unique URLs for meeting polls.
- *Seamless Error Handling*: Next.js comes with built-in error handling, which allows developers to define custom error pages. When a developer encounters a similar error, the framework automatically displays the appropriate error page along with helpful debugging information, significantly reducing development time and effort.

4.1.2 React.js

React.js enables a component-based architecture for building complex web applications [Meta 2026]. The advantages of using React.js in a web project include:

- *Component Reusability*: React.js gives the possibility to break down the user interface into encapsulated, reusable components (for example the Calendar component). This improves the maintainability and allows developers to reuse code easily across the application.
- *Interactivity*: The state management mechanisms of React simplify the handling of state-based interactions. They allow developers to implement responsive and dynamic user interfaces.

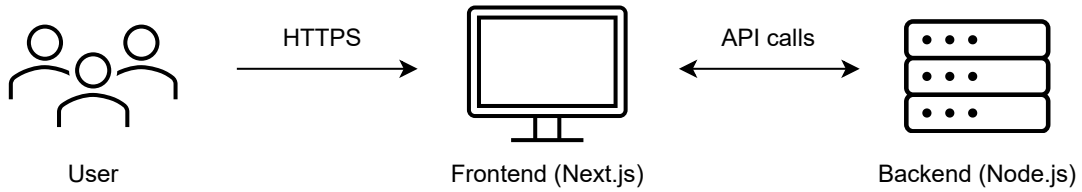

Figure 4.1: Schedulo: Client-server architecture. Users interact with the frontend via HTTPS requests. The frontend then communicates with the backend via API calls. [Diagram drawn by Sarah Hörtnagel.]

4.1.3 Mantine

The Mantine library provides modern and robust React UI components [Mantine 2026]. The idea is to reuse existing implementations, for example date pickers, modals or buttons, instead of building everything from scratch. Development time is reduced, while still ensuring functionality and a consistent visual design.

4.1.4 Tailwind CSS

Tailwind CSS is a utility-first CSS framework that enables styling directly within the markup [Tailwind 2026]. This allows rapid and flexible UI customization.

4.2 Backend

The backend of Schedulo is built on Node.js, which provides the server-side runtime environment to execute JavaScript logic [OpenJS 2026]. Using Node.js fulfils the project objective to use a full-stack JavaScript development environment. The primary responsibility of the backend is to handle business logic, such as evaluating polls, communicating with the frontend, and persisting data.

To realise smooth communication between the frontend and backend, Schedulo utilises the TypeScript Remote Procedure Call (tRPC) library. Unlike REST APIs or GraphQL APIs, tRPC allows the implementation of end-to-end type-safe APIs, without schema synchronisation or manual code adjustments. The key features of tRPC are the following, extracted from the official documentation [tRPC 2024]:

- *Full Static Type Safety*: tRPC provides complete type safety and autocompletion for inputs, outputs, and errors. This makes sure that the data sent from the Next.js frontend strictly matches the requirements of the Node.js backend.
- *Developer Efficiency*: By removing the need for a separate schema synchronisation step or manual code generation, tRPC allows a seamless development process where API changes are immediately reflected across the entire stack. If an input type (for example a field in `ZodPoll`) is modified in the backend, the TypeScript compiler will immediately flag an error in the frontend if the corresponding code is not updated.
- *Request Efficiency*: tRPC supports request batching, where multiple procedure calls made in the same time frame are automatically combined into a single HTTP request. This reduces network overhead when the Schedulo frontend needs to fetch various data points simultaneously, such as poll details and participant comments.
- *Zod Validation*: Schedulo uses Zod as the validation engine for all tRPC procedures. This ensures that all incoming data, such as `ZodVoteValue` or `ZodTimeSlot`, is strictly validated against the application's business logic before being processed or persisted to the local filesystem.

Environment Variable	Description
ADMIN_PASSWORD	Default administrator password for the application.
ORGANIZER_SESSION_SECRET	Secret key used to sign and verify session cookies. This value should be a long, random string to ensure session security.
NODE_ENV	Specifies the runtime environment as (development, test or production). This value is automatically set by Next.js when using pnpm dev or pnpm start, but must be defined explicitly for manual deployments.

Table 4.1: Schedulo: Environment variables required to configure and run the application.

4.3 Building and Deployment

Before starting the application, it is necessary to make sure that the environment variables described in Table 4.1 are correctly set. The application can be built and run by following a series of steps:

1. The required dependencies must be installed using the pnpm package manager:

   ```
   pnpm install
   ```

2. For local development, the application can then be started in development mode using:

   ```
   pnpm dev
   ```

3. To create a production build without containerization, the application can be built and started with the following commands:

   ```
   pnpm build
   pnpm start
   ```

By default, the application is accessible on port 3000.

For production deployment, Docker Compose is used to build and run the application in a containerised environment. This approach allows the application to run in a consistent environment across different systems. The application can be built and started using the command:

```
docker-compose up -d
```

This command builds the Docker image, which comes as a Dockerfile in the repository, starts the application container, and creates a persistent Docker volume for poll data storage. The service is exposed on port 3000 of the host system. The -d flag starts the container in detached mode, which lets Docker Compose run in the background without attaching logs to the current terminal session.

To stop the application, the following command is used:

```
docker-compose down
```

In a production environment, the application is typically deployed behind an existing web server or reverse proxy, such as Nginx, which handles concerns like HTTPS termination, domain routing, and access control.

4.4 Data Persistence Options

Schedulo stores poll data persistently, keeping created polls and submitted votes available across application restarts. During development and for simple deployments, poll data is written to the local file system in JSON format. This approach enables a lightweight setup without requiring external services and is sufficient for local testing and small-scale deployments.

For container-based deployments, data persistence is achieved using Docker volumes. By mounting a persistent volume into the container, poll data stored on the local file system remains available independently of the container lifecycle. This way the application container itself remains stateless, while preserving user-generated data across restarts and updates.

For production environments with higher reliability or scalability requirements, external storage solutions can be integrated. One option is to store poll data in an Amazon S3 bucket, which provides durable object storage and facilitates backups and horizontal scaling. This approach requires adapting the data access layer to use the AWS SDK instead of local file system operations.

Alternatively, a relational database such as PostgreSQL can be used to store poll and vote data. A database-backed persistence layer enables advanced querying, transactional guarantees and improved concurrency handling. This option is particularly suitable for deployments with a larger number of users or more complex data management requirements.

In container orchestration environments, such as Kubernetes, persistent storage can be managed using Persistent Volumes (PV) and Persistent Volume Claims (PVC). In this setup, the application pod mounts a PVC which is backed by a storage provider, letting poll data persist independently of pod restarts or rescheduling events. This approach combines the benefits of containerised deployment with reliable and scalable data persistence.

The choice of persistence strategy depends on the deployment context and operational requirements. While local file system storage and Docker volumes provide simplicity and ease of setup, cloud-based storage and database solutions offer increased robustness and scalability.

Further background information can be found in the official documentation of Docker [Docker 2026b], Docker Compose [Docker 2026a], Kubernetes storage concepts [Kubernetes 2021], Amazon S3 [AWS 2026], and PostgreSQL [PostgreSQL 2026].

4.5 Migrating from File System Storage to a Database

Although the current approach of using the local file system for storing persistent data is more than sufficient for a self-deployed application, administrators might want to use a database in the future. To migrate to a database-backed architecture, Prisma [Prisma 2026] can be introduced as an ORM layer to replace file system access with typed database queries. The migration enables efficient querying and filtering, improves consistency through transactional updates and relational constraints and reduces the risk of corrupted files under concurrent access.

To perform the migration, a relational data model must be derived from the existing Zod schemas. This includes creating Prisma models for `Poll`, `TimeSlot`, `Vote`, and `Comment` and defining primary keys and relations. Furthermore, cascading deletes and uniqueness rules can be specified directly in the Prisma schema.

Next, instead of reading and writing JSON files, all CRUD operations must be implemented using Prisma's client API. This means replacing file-based operations in the tRPC routes with Prisma queries. The rest of the application logic can stay untouched as long as the new relational data models are a exact copy of the defined schemas.

If the existing data should be preserved, a data migration step is required. The JSON data can be imported into the database using a one-time script which inserts records using Prisma or alternatively through Prisma seed functionality. During this step, existing UUIDs should be retained to avoid breaking existing URLs and foreign key references.

In order to migrate the application from the filesystem approach to the database-backed architecture the following steps are needed:

1. Install Prisma and the Prisma client:

```
pnpm install prisma @prisma/client npx prisma init
```

2. Configure the database connection in `.env` using `DATABASE_URL`.
3. Define the database models in `prisma/schema.prisma`.
4. Create and apply the initial migration:
   ```
   npx prisma migrate dev --name init
   ```
5. Generate the Prisma client:
   ```
   npx prisma generate
   ```
6. (Optional) Import existing file system data using a migration or seed script:
   ```
   npx prisma db seed
   ```

Chapter 5

Implementation

Certain implementation details require special mention.

5.1 Data Management

In order to ensure type safety when writing and reading from the JSON files in the local filesystem, Zod [McDonnell 2026] was used to define all necessary schemas. These schemas are then used to validate the data read from the files. Additionally, all TypeScript types are inferred directly from these schemas. Listing 5.1 shows the four schemas used in the project:

- `ZodComment`: The `ZodComment` schema represents a participant's comment on a poll and consists of three fields. The `comment` field stores the textual content, `name` contains the username provided by the participant, and `userId` serves as a foreign key referencing the participant's vote.
- `ZodTimeSlot`: The `ZodTimeSlot` schema models a proposed meeting timeslot and consists of 4 fields. The `id` field contains a UUID used for unique identification, while the `date` field stores the calendar date of the timeslot as a string. The `startTime` and `endTime` fields define the temporal boundaries of the timeslot and are also represented as strings.
- `ZodVote`: The `ZodVote` schema represents a participant's vote for a specific timeslot and consists of 6 fields. The `userId` uniquely identifies the participant, while `pollId` and `timeSlotId` serve as foreign keys referencing the associated poll and timeslot. The `name` field stores the participant's chosen display name. The `value` field stores the participant's availability using an enumerated type (`yes`, `no` or `ifNeedBe`). Finally, the `comment` field allows the attachment of a timeslot-specific remark.
- `ZodPoll`: The `ZodPoll` schema represents an organiser-created poll and consists of 12 fields. The `id` field uniquely identifies the poll, while `organizerId` acts as a foreign key referencing the poll's organiser. The `createdAt` field records the poll's creation timestamp. Poll state is managed through the `active` flag, which controls availability of the poll and the `closedAt` timestamp, which indicates whether the poll has been finalized. The descriptive metadata for the poll is stored in the `title`, `location` and `description` fields. The `dates` array contains all proposed timeslots, while the `votes` and `comment` arrays store submitted participant votes and poll-wide comments. Finally, the `winner` field is used when the organiser selects a timeslot which differs from the majority vote.

5.2 Authorisation and Authentication

At the beginning of the projec,t it was decided to use a role-based authorisation system to manage a user's access to the different parts of the application. As described in Section 3.2, Schedulo has 3 different access roles (levels): Administrator, Organiser, and Participant. Participants do not need any authentication, because of the project's decision to refrain from end-user accounts.

```
const ZodComment = z.object({
  userId: z.string(),
  comment: z.string(),
  name: z.string(),
});

const ZodTimeSlot = z.object({
  id: z.string(),
  date: z.string(),
  startTime: z.string(),
  endTime: z.string(),
});

const ZodVoteValue = z.enum(['yes', 'no', 'ifNeedBe']);

const ZodVote = z.object({
  userId: z.string(),
  pollId: z.string(),
  name: z.string().min(1).max(60),
  timeSlotId: z.string(),
  value: ZodVoteValue.optional(),
  comment: z.string().optional(),
});

const ZodPoll = z.object({
  id: z.string().optional(),
  title: z.string().min(1, 'Title is required'),
  location: z.string().optional(),
  description: z.string().optional(),
  dates: z.array(ZodTimeSlot),
  votes: z.array(ZodVote).optional(),
  createdAt: z.coerce.date(),
  organizerId: z.string(),
  closedAt: z.coerce.date().optional(),
  comment: ZodComment.array().optional(),
  active: z.boolean().optional(),
  winner: ZodTimeSlot.optional(),
});
```

Listing 5.1: The four Zod schemas used to ensure type safety between Schedulo's frontend, backend, and filesystem.

Authentication for the administrator and organisers is implemented using signed, HTTP-only cookies combined with server-side verification in Next.js middleware. Upon successful login, the server issues a session cookie containing a cryptographically signed payload using an HMAC-SHA256 signature with a server-side secret.

Session validity is enforced by embedding an expiration timestamp within the signed cookie payload, which is verified on every request. The middleware intercepts requests to protected routes and grants access only if a valid and non-expired session cookie is present. Cookies are configured with appropriate security attributes, including `HttpOnly`, `SameSite` and `Secure` (in production) to mitigate common web vulnerabilities such as cross-site scripting and cross-site request forgery.

Additionally, on critical tRPC API routes, the context is checked for a valid `organiserId` before executing any read or write operations. This identifier is derived from the verified organiser session and injected into the request context. Requests without a valid organiser context are rejected, ensuring that protected operations can only be performed by authorised organisers.

5.3 Dynamic Routes

Schedulo makes use of dynamic routing to provide access to polls without relying on user accounts. Each poll is identified by a unique identifier, which is embedded directly into the URL and used to resolve the corresponding poll data on the server.

For participants, the routes `/vote/[id]` and `/results/[id]` enable stateless access to a poll. The voting route allows participants the submission of their availability, while the results route provides read-only access to the final meeting date. This approach allows participants to interact with polls solely via a shared link, reducing friction and simplifying the user experience while avoiding persistent user identities.

For organisers, the routes `/organize/[id]/edit` and `/organize/[id]/results` provide access to poll management and evaluation features. Access to these routes is protected by middleware and additional server-side checks, ensuring that only the authenticated organiser associated with the poll can modify its data.

Chapter 6

Concluding Remarks

This report presented the design and implementation of Schedulo, a lightweight and self-hosted meeting scheduling application. The system was developed with a strong focus on simplicity and privacy. By relying on URL-based access and avoiding participant accounts, Schedulo enables accountless participation while minimising the collection of personal data. The use of modern web technologies, including Next.js, tRPC and Zod, ensures a responsive user interface, type-safe data handling and role-based authentication. Overall, the application fulfills the defined requirements for self-hosted deployment, privacy-friendly operation, and ease of use.

While Schedulo meets its core objectives, it can still be improved in multiple ways:

- *Database-Backed Architecture*: Migrating the application from file system storage to a database would improve scalability as well as querying and filtering capabilities. In addition, transactional guarantees and relational constraints would reduce the risk of data corruption under concurrent access. This change would also simplify future feature extensions, such as advanced result aggregation and auditing.

- *Mandatory Votes*: An important usability improvement would be the introduction of mandatory timeslots. Organisers could mark specific timeslots as mandatory, forcing participants to indicate their availability for these slots before submitting their vote. This approach prevents critical time options from being overlooked and allows the organiser to reliably determine whether a viable meeting date exists.

- *Variable Availability*: In some cases, the discrete options *Yes*, *No* and *If Need Be* may be too restrictive to express a participant's availability. As an alternative, a slider could be introduced, letting participants indicate their availability on a finer-grained scale. This approach would enable more nuanced responses and improve the expressiveness of votes when availability is uncertain or flexible.

Bibliography

Aichner, Lorenz, Sarah Hörtnagel, Hagen Leitner and Fabian Szakács [2026]. *Schedulo*. 02 Feb 2026. https://github.com/IAWeb-2025-G3/Schedulo (cited on page 7).

Andrews, Keith [2025]. *Human-Computer Interaction: Course Notes*. 27 May 2025. https://courses.isds.tugraz.at/hci/hci.pdf (cited on pages 8, 11).

AWS [2026]. *Amazon Simple Storage Service Documentation*. Amazon Web Services, 01 Feb 2026. https://docs.aws.amazon.com/s3 (cited on page 16).

Docker [2026a]. *Docker Compose Documentation*. 01 Feb 2026. https://docs.docker.com/compose (cited on page 16).

Docker [2026b]. *Docker Documentation*. 01 Feb 2026. https://docs.docker.com/ (cited on page 16).

Doodle [2026]. *Doodle*. 28 Jan 2026. https://doodle.com/ (cited on pages 1, 3, 8).

Framasoft [2026]. *Framadate*. 28 Jan 2026. https://framadate.org/ (cited on pages 1, 8).

Kubernetes [2021]. *Storage Concepts*. The Kubernetes Authors, 16 Jun 2021. https://kubernetes.io/docs/concepts/storage (cited on page 16).

Mantine [2026]. *Mantine*. 01 Feb 2026. https://mantine.dev/ (cited on pages 7, 14).

McDonnell, Colin [2026]. *Zod*. 01 Feb 2026. https://zod.dev/ (cited on page 19).

Meta [2026]. *React: The library for web and native user interfaces*. 28 Jan 2026. https://react.dev (cited on page 13).

OpenJS [2026]. *Node.js*. 01 Feb 2026. https://nodejs.org/ (cited on page 14).

PostgreSQL [2026]. *PostgreSQL Documentation*. 01 Feb 2026. https://postgresql.org/docs (cited on page 16).

Prisma [2026]. *Prisma ORM*. 21 Jan 2026. https://prisma.io/docs/orm (cited on page 16).

Rallly [2026]. *Rallly*. 28 Jan 2026. https://rallly.co/ (cited on pages 3, 8).

Rosenkranz, Vinzenz, René Gieling and Kai Schröer [2026]. *Polls*. Nextcloud, 18 Jan 2026. https://apps.nextcloud.com/apps/polls (cited on page 8).

Rupareliya, Kamal [2025]. *Why Use Next.js: Reasons & Benefits for Developers to Choose This Framework*. Intuz, 06 Oct 2025. https://intuz.com/blog/5-reasons-why-you-should-use-next.js-for-your-front-end-development (cited on page 13).

Systemli [2026]. *Croodle beta*. 01 Feb 2026. https://systemli.org/service/croodle (cited on page 8).

Tailwind [2026]. *Tailwind CSS*. 28 Jan 2026. https://tailwindcss.com/ (cited on pages 7, 14).

tRPC [2024]. *tRPC*. 28 Apr 2024. https://trpc.io/ (cited on pages 7, 14).

TU Dresden [2026]. *DuD-Poll*. Technische Universität Dresden, 01 Feb 2026. https://dud-poll.inf.tu-dresden.de/ (cited on page 8).

Vercel [2025]. *Next.js*. 30 Jul 2025. https://nextjs.org/ (cited on pages 7, 13).

When2Meet [2026]. *When2Meet*. 28 Jan 2026. https://when2meet.com/ (cited on pages 1, 8).