NEXTGEN
- DEVELOPMENT -

Designbeschreibung

NextGen Development

Version 1.0

21. April 2025

Teammitglieder:
Julian Lachenmaier
Ayo Adeniyi
Din Alomerovic
Cedric Balzer
Rebecca Niklaus

Verantwortlich für dieses Dokument:
Din Alomerovic

Inhaltsverzeichnis

1 Allgemeines **2**

2 Produktübersicht **2**

3 Grundsätzliche Struktur- und Entwurfsentscheidungen **4**

3.1 Technologien 4

3.2 Aufbau Flows 5

3.2.1 Trigger 6

3.2.2 Actions 8

3.2.3 Ablauf 9

3.3 Registrierungsprozess 9

3.4 Rollenverteilung 10

3.5 Erstellung eines Events 12

4 Grundsätzliche Struktur- und Entwurfsentscheidungen der einzelnen Pakete/Komponenten **14**

4.1 Backend 14

4.2 API 15

4.3 Frontend 18

4.3.1 Root-Layout und globale Provider 18

4.3.2 Layouts und Container 19

4.3.3 Wiederverwendbare UI-Komponenten 19

4.3.4 Utility-First CSS mit Tailwind 19

4.3.5 Modulares Routing 19

4.3.6 API-Interaktion 19

1 Allgemeines

Dieses Dokument soll das Softwaredesign der ersten Version unseres Event-Management-Systems beschreiben. Es basiert auf dem Design unseres Prototypen, welches hier jedoch noch präziser dargestellt und erweitert wird.

2 Produktübersicht

Die erste Version unseres Projekts wird folgende Funktionen bieten:

- Event konfigurieren: Der Organisator ist dazu in der Lage Events innerhalb seiner Organisation zu gestalten. Dazu kann er sich Vorlagen bedienen und eigene Flows erstellen.
- Auf der Plattform registrieren: Ein Benutzer kann sich mit seiner E-Mail-Adresse der Organisation auf der Plattform registrieren. Nach seiner Registrierung kann der Benutzer seiner Organisation beitreten und alle Events einsehen.
- Organisatoren einladen: Ein Organisator kann innerhalb seiner Organisation weitere Organisatoren einladen.
- Zu einem Event an-/abmelden: Ein Benutzer kann sich selbstständig zu einem Event innerhalb seiner Organisationen an- und abmelden.
- Flows konfigurieren: Ein Organisator kann Flows für die Organisation oder eventspezifisch erstellen.

Das Use-Case-Diagramm in Abbildung 1 fasst noch einmal alle Funktionen zusammen, die auch über die Funktionen einer möglichen ersten Version des Produktes hinausgehen.

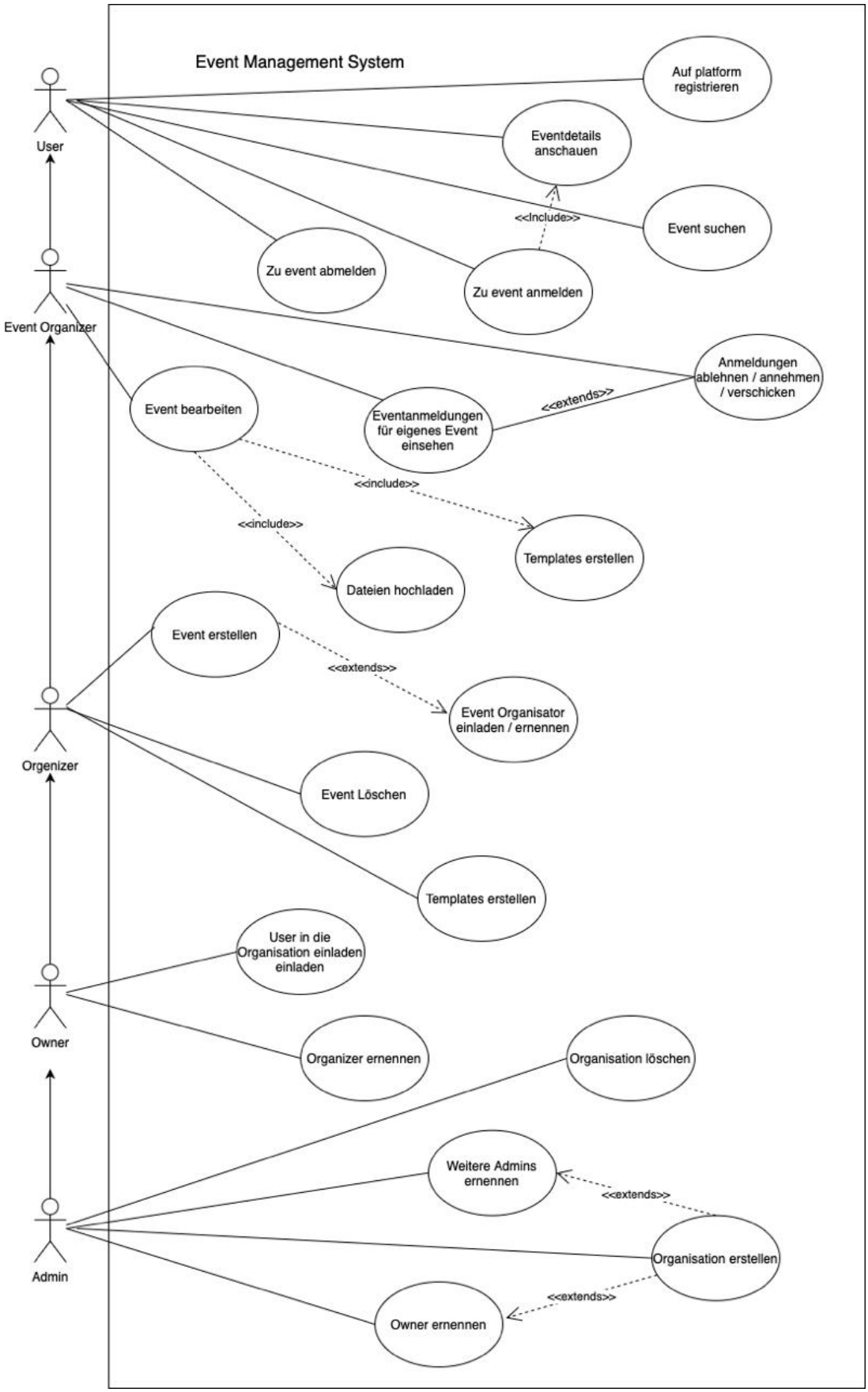

Abbildung 1: Use Case Diagramm

3 Grundsätzliche Struktur- und Entwurfsentscheidungen

Der vorliegende Abschnitt erläutert die wesentlichen Systementscheidungen und behandelt bereits zentrale inhaltliche Aspekte. Weil das Endprodukt direkt aus dem Prototyp hervorgeht, ähnelt die Grundarchitektur der ersten Produktversion weitestgehend der Prototypenstruktur. Aus diesem Grund implementieren wir auch in der ersten Produktversion eine dreischichtige Architektur aus Frontend, Backend und Datenhaltung.

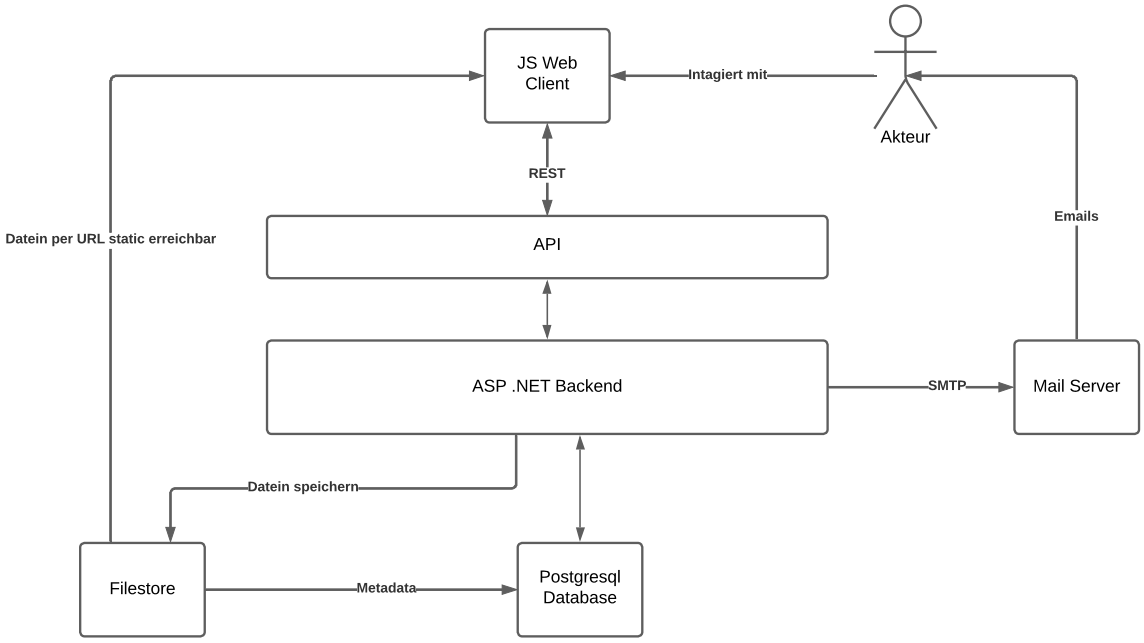

Abbildung 2: Genereller Aufbau

3.1 Technologien

Frontend: Das Benutzerinterface wird mit der JavaScript-Bibliothek React.js in Kombination mit dem Framework Next.js realisiert. Die Entscheidung für dieses Technologie-Stack basiert auf der positiven Erfahrung aus der Prototypenphase, insbesondere im Hinblick auf die flexible Komponentenarchitektur und die breite Community Unterstützung.

Backend: Im Backend setzen wir auf ASP.NET Core, um eine robuste, performante und testbare Serverarchitektur zu gewährleisten. Durch den Einsatz von ASP.NET Core profitieren wir von einem bewährten Framework, das sowohl moderne Entwicklungspraktiken

wie Dependency Injection als auch umfangreiche Sicherheitsfeatures bietet.

Datenbank: Als Datenbank wird uns auch weiterhin die PostgreSQL Datenbank dienen, da wir bei der Erstellung unseres Backends bereits viele wertvolle Erfahrungen sammeln konnten und nun bereits die Grundlage für die Datensicherung mit dieser Datenbank gelegt haben.

API: Die Interaktion zwischen Frontend und Backend erfolgt über eine REST-API. Diese Kommunikationsschnittstelle erlaubt es, die Präsentations- und Anwendungslogik klar zu trennen und eine lose Kopplung der Komponenten sicherzustellen. Die Routenstruktur und die wichtigsten Endpunkte werden im Abschnitt 4.2 beschrieben.

3.2 Aufbau Flows

Ein FlowTemplate hat allgemein folgenden Aufbau:

```
    id: string;
    name: string;
    description: string;
    trigger: Condition[];
    actions: Action[];
    createdAt: Date;
    updatedAt: Date;
    createdBy: string;
    updatedBy: string;
    isUserCreated: boolean;
```

Jedes FlowTemplate hat eine eindeutige ID und besteht aus einer Menge von Triggern, und einer Menge von Actions. Die Trigger sind Bedingungen, die wahr oder falsch sein können, wie z.B. sind es noch zwei Wochen bis Eventstart? Actions hingegen sind Aktionen die ausgeführt werden können, wie z.B. schicke eine Wilkommens-Email. Die Actions sollen dann ausgeführt werden, wenn alle Trigger erfüllt sind. Ist dies der Fall so wird ein Flow Run gestartet, also eine Instanz aus diesem Template, welches durchläuft und dann fehlschlagen kann oder erfolgreich beendet wird. Im Optimalfall werden hierbei Logs zu allen Actions aufgezeichnet um diese nachvollziehen zu können und auch wird das erfolgreiche Durchlaufen oder das Fehlschlagen der einzelnen Actions und nicht nur des gesamten Flow Runs dokumentiert.

3.2.1 Trigger

Ein Trigger hat allgemein folgende Form:

```
    id: string;
    type: ConditionType;
    details: ConditionDetails;
    // Etwaige andere Meta-Eigenschaften
```

Es gibt 5 Kategorien die ein Trigger haben kann:

```
    date
    relativeDate
    numOfAttendees
    status
    registration
```

Entsprechend abhängig von der ConditionType sind die ConditionDetails. Diese sind als JSON in der Datenbank gespeichert und grundlegend unterschiedlich. Nachfolgend wird jede dieser Kategorien kurz definiert.

```
    date {
        operator: "before" | "after" | "on";
        value: date,
    }
```

```
    relativeDate {
        operator: "before" | "after" | "equal",
        value: number,
        valueType: "hours" | "days" | "weeks" | "months" ,
        valueRelativeTo: "event.start" | "event.end",
        valueRelativeOperator: "before" | "after"
    }
```

Während der Date Trigger ein festes Datum (vor, nach oder zu einem bestimmten Zeitpunkt) betrachtet, bezieht sich der relativeDate Trigger auf den Beginn oder das Ende des Events, was sie flexibler macht.

```
numOfAttendees {
    operator: "greaterThan" | "lessThan" | "equalTo",
    valueType: "absolute" | "percentage",
    value: number,
}
```

Ist im valueType percentage ausgewählt, so ist die im Value gegeben Zahl als Prozente relativ zur maximalen Teilnehmeranzahl des Events zu verstehen.

```
status {
    operator: "is" | "isNot",
    value: "active" | "cancelled" | "completed" | "archived"
       | "draft",
}
```

Der Registration Trigger besitzt keine Details. Er ist dann wahr, wenn sich ein neuer Nutzer zu einem Event anmeldet. Dieser Nutzer kann in Actions mittels der ‘trigger.registration.user‘ Variable referenziert werden. Der Registration Trigger ist pro Flow unique und kann somit nur einmal vorkommen.

Verglichen zu den anderen Triggern handelt es sich beim Registration Trigger um einen Sonderfall. Alle anderen Trigger sind binär entweder dauerhaft wahr oder falsch. So ist der Status von einem Event entweder der angegebene oder nicht und damit unabhängig von der Anzahl und Art der Durchführung an Runs. Der Registration Trigger hingegen ist immer nur in dem Augenblick der Neuregistrierung eines bestimmten Users erfüllt. Dies hat zur Folge, dass jede Registrierung, also jedes Erfüllen diesen Trigger, eine Überprüfung der anderen dem Flow zugeordneten Triggern zum jeweiligen Zustandes des Events im Augenblick der Registrierung notwendig macht.

In der Praxis hat dies die Folge, dass mit jeder Registrierung zu einem Event überprüft werden muss, ob dieses Event ein Flow mit Registration Trigger besitzt und wenn dies der Fall ist, dieser Überprüft und ggf. instanziiert werden muss. Die mögliche Alternative, in welcher der Zustand des Events im Augenblick der Registrierung gespeichert wird, wird für die Praxis als nicht notwendig erachtet. Alle Betrachtungen diesbezüglich werden nicht weiter vorgenommen.

3.2.2 Actions

Es gibt 5 Arten von Actions diese sind:

```
    email
    statusChange
    fileShare
    imageChange
    titleChange
    descriptionChange
```

Grundsätzlich ist eine Action wie auch ein Trigger wie folgt aufgebaut:

```
    id: string;
    type: ActionType;
    details: ActionDetails;
    // Etwaige andere Meta-Eigenschaften
```

Entsprechend abhängig von dem ActionType sind die ActionDetails. Diese sind als JSON in der Datenbank gespeichert und grundlegend unterschiedlich.

Zum aktuellen Zeitpunkt wurden die email action und die fileShare action noch nicht im detail definiert. Eine Definition davon folgt in einer weiteren Version dieses Dokuments. Der imageChange, der titleChange und der descriptionChange haben jeweils ein String-Attribut welches das im Titel stehende Attribut eines Events verändert.

3.2.3 Ablauf

Das Aktivitätsdiagramm in Abbildung 3 soll verdeutlichen wie das Erstellen von Flows bzw. Flow-Templates funktioniert.

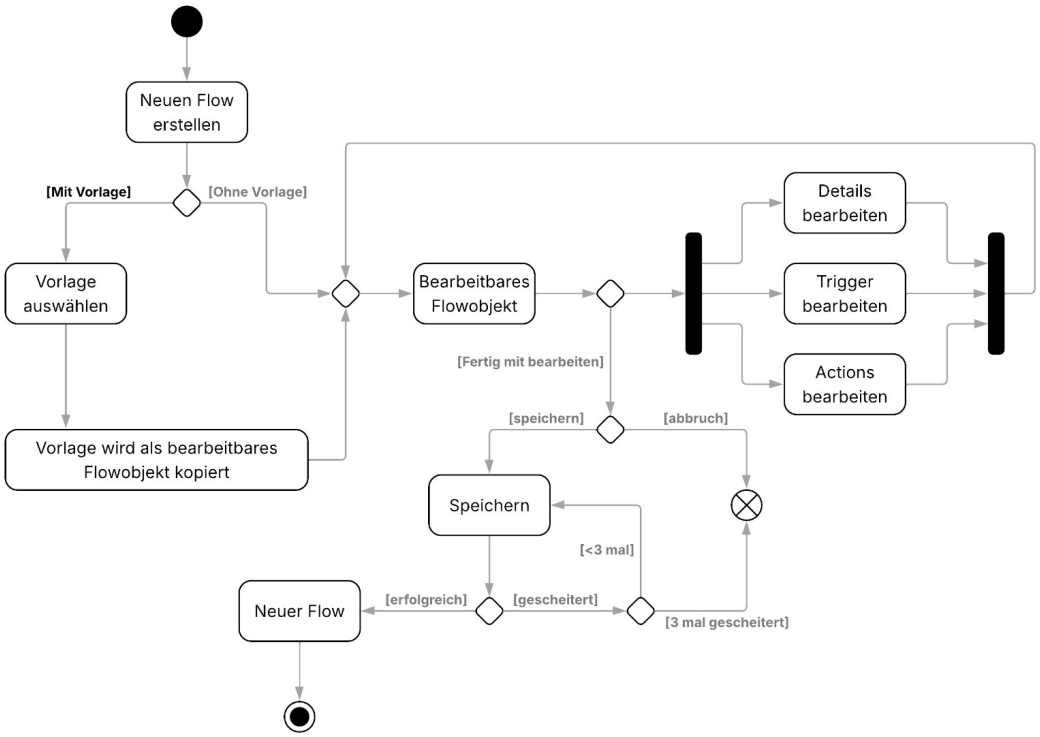

Abbildung 3: Erstellung von Flows

3.3 Registrierungsprozess

Abbildung 4 stellt ein Sequenzdiagramm dar, welches die Kommunikation der einzelnen Komponenten untereinander während der Registrierung eines Users veranschaulichen soll.

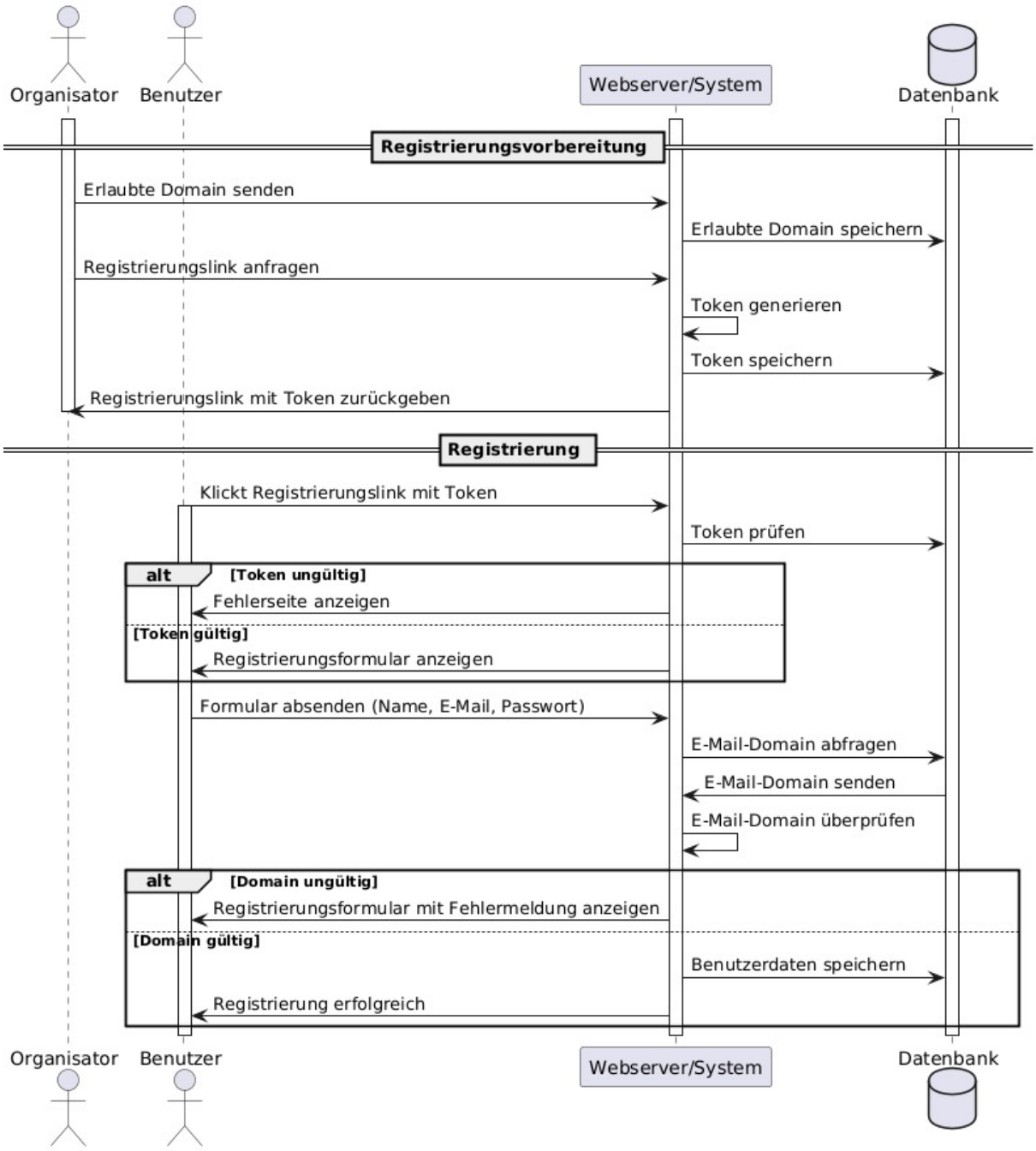

Abbildung 4: Registrierung eines Users

3.4 Rollenverteilung

In unserem System gibt es fünf zentrale Rollen, die jeweils mit spezifischen Rechten und Zuständigkeiten ausgestattet sind: Administrator (Admin), Owner (Organisationsinhaber), Organizer (Veranstaltungsmanager), Event Organizer (Event-Verantwortlicher) und User (Teilnehmer).

Der Administrator verfügt über die höchsten Berechtigungen und kann systemweit Organisationen und Events erstellen, bearbeiten sowie löschen. Zudem hat er die volle Kontrolle über alle Rollen und Benutzer, einschließlich der Möglichkeit, Owner zu entfernen.

Der Owner ist einer bestimmten Organisation zugeordnet und besitzt innerhalb dieser Organisation ähnlich umfangreiche Rechte wie der Admin. Er kann weitere Owner einladen und ist für die Verwaltung aller Organisationsmitglieder sowie deren Rollen verantwortlich.

Der Organizer kümmert sich um die Events innerhalb seiner Organisation. Er hat die Befugnis, Templates zu erstellen und Event Organizer zu ernennen. Dabei kann es in einer Organisation mehrere Organizer geben, die gemeinsam die Veranstaltungen verwalten.

Der Event Organizer ist für ein konkretes Event zuständig und darf innerhalb dieses Events alle notwendigen Anpassungen vornehmen. Sein Zugriff beschränkt sich jedoch ausschließlich auf das ihm zugewiesene Event; andere Events kann er nicht bearbeiten.

Die Rolle des Users ist auf die Teilnahme an Events beschränkt. User haben keine administrativen Rechte und können weder Events noch Organisationen verwalten.

Dieses Rollenkonzept gewährleistet eine klare Aufgabentrennung und ermöglicht eine effiziente Verwaltung des Systems, indem jeder Nutzer entsprechend seiner Rolle handeln kann.

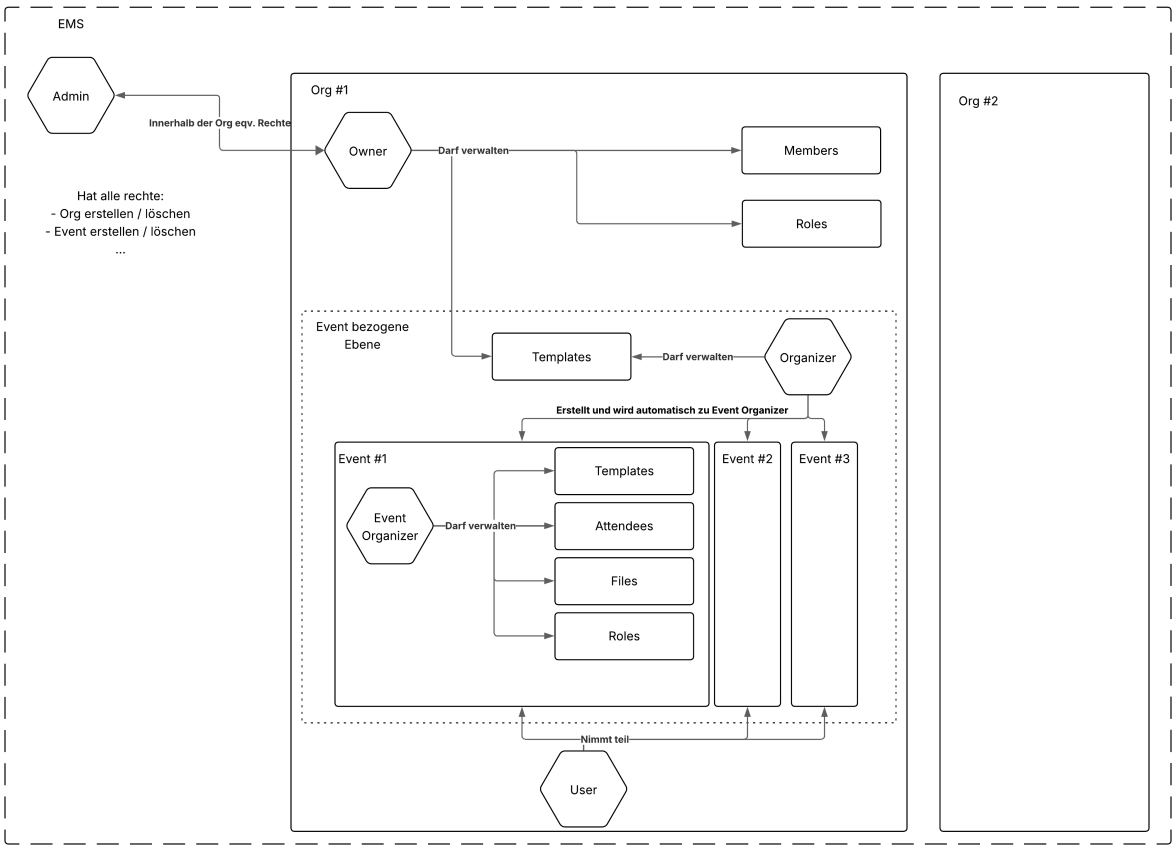

Abbildung 5: Übersicht Rollenverteilung

3.5 Erstellung eines Events

Das Sequenzdiagramm in Abbildung 6 soll verdeutlichen, wie die Kommunikation unserer Komponenten untereinander während der Erstellung eines Events funktioniert.

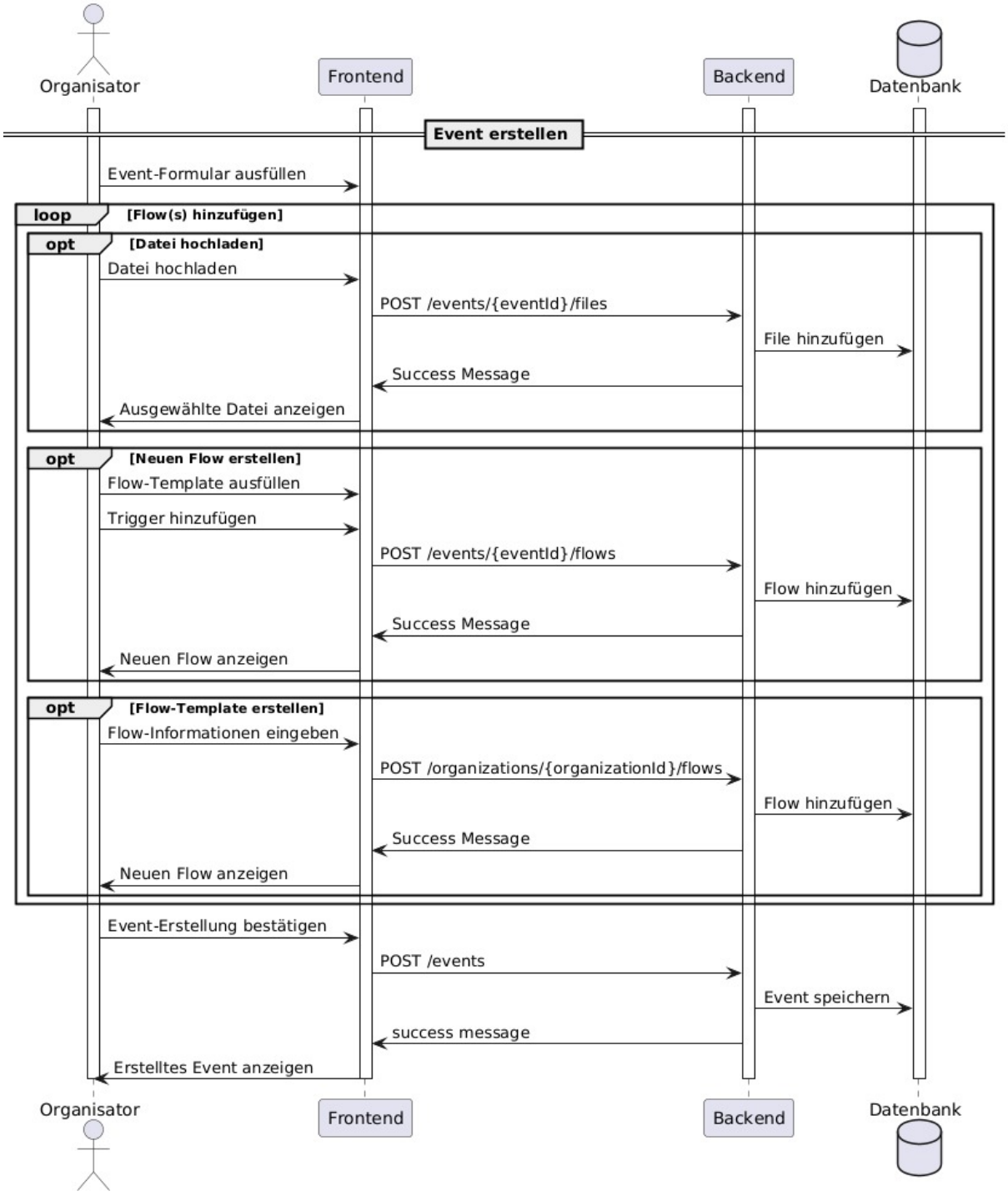

Abbildung 6: Event Erstellung

4 Grundsätzliche Struktur- und Entwurfsentscheidungen der einzelnen Pakete/Komponenten

4.1 Backend

Unser Backend ist modular aufgebaut und besteht aus mehreren zentralen Paketen, die jeweils spezifische Aufgaben übernehmen. Die Controller bilden die Schnittstelle zu den Endpunkten und sind für die Koordination der eingehenden Anfragen zuständig. Sie leiten die Anfragen weiter und stellen die entsprechenden Antworten bereit. Die eigentliche Geschäftslogik ist in den Services gekapselt, die die fachlichen Prozesse steuern und die notwendigen Berechnungen oder Datenverarbeitungsschritte durchführen. Die Kommunikation mit der Datenbank erfolgt über die Repository-Klassen, die für das Lesen und Speichern von Daten zuständig sind und so die Persistenzschicht des Systems darstellen. Wir haben uns für diese Aufteilung entschieden, damit es eine klare Trennung der Zuständigkeiten gibt und der Code strukturiert und gut wartbar ist.

Das Klassendiagramm in Abbildung 7 zeigt den Aufbau am Beispiel der User Klassen:

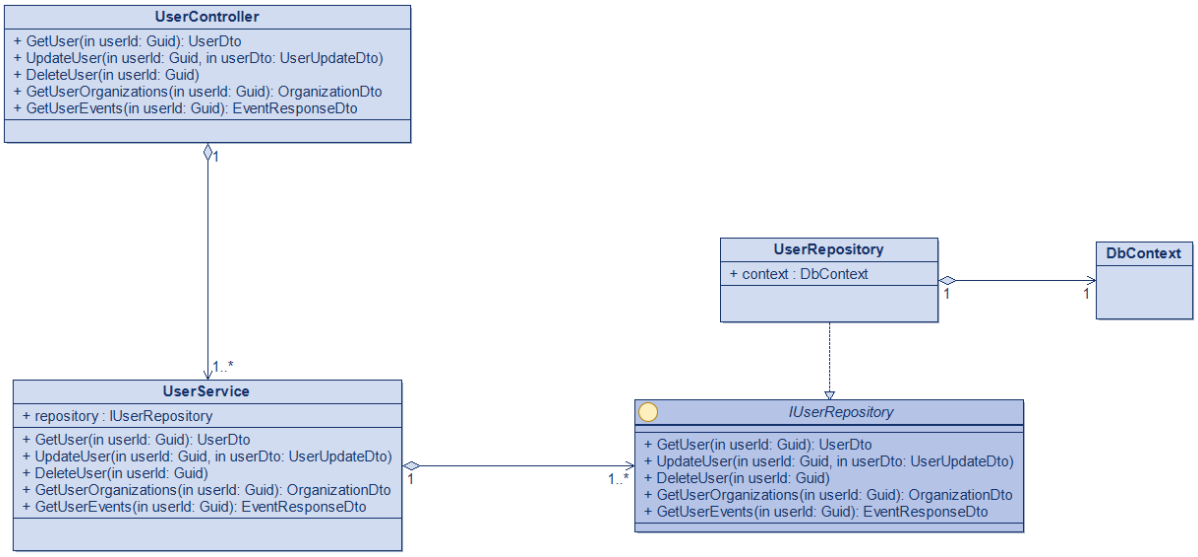

Abbildung 7: User Beispiel

Nach diesem Prinzip funktionieren auch die anderen Klassen, wie in Tabelle 1 gezeigt:

Controller	**Service**	**Repository**
UserController	UserService	UserRepository
OrgFlowController	OrgFlowService	OrgFlowRespository
OrganizationController	OrganizationService	OrganizationRepository
EventFlowController	EventFlowService	EventFlowRepository
EventController	EventService	EventRepository
EmailController	EmailService	EmailRepository
AuthController	AuthService	AuthRepository

Tabelle 1: Zuordnung von Controllern, Services und Repositorys

4.2 API

Dieser Abschnitt dokumentiert sämtliche Endpunkte der API und gibt einen Überblick über deren Funktionalität. Die hier beschriebene API-Architektur ist nicht ausschließlich für die erste Produktversion konzipiert, sondern soll langfristig in dieser oder einer ähnlichen Form Bestand haben.

Endpoint	**Beschreibung**
/auth/login	User login
/auth/register	User Registration
/orgs (GET)	Get a list of organizations
/orgs (POST)	Create a new organization
/orgs/{orgId} (GET)	Get detailed organization data by id
/orgs/{orgId} (PUT)	Update organization data by id
/orgs/{orgId} (DELETE)	Delete organization with given id
/orgs/{orgId}/emails (POST)	Create an email template
/orgs/{orgId}/members (GET)	Get members of an organization

Endpoint	**Beschreibung**
/orgs/{orgId}/members (POST)	Add a new member to an organization
/orgs/{orgId}/members/{userId} (PUT)	Update member data (mail)
/orgs/{orgId}/members/{userId} (DELETE)	Delete a member from an organization
/orgs/{orgId}/events (GET)	Get a list of events
/orgs/{orgId}/events (POST)	Create a new event
/orgs/{orgId}/events/{eventId} (GET)	Get an event by id
/orgs/{orgId}/events/{eventId} (PUT)	Update an event by id
/orgs/{orgId}/events/{eventId} (DELETE)	Delete an event by id
/orgs/{orgId}/events/{eventId}/email (POST)	Send an email
/orgs/{orgId}/events/{eventId} /attendees (GET)	Get attendees for an event
/orgs/{orgId}/events/{eventId} /attendees (POST)	Add a new attendee to an event
/orgs/{orgId}/events/{eventId} /attendees/{userId} (DELETE)	Remove an attendee from an event
/orgs/{orgId}/events/{eventId}/files (GET)	Get files for an event
/orgs/{orgId}/events/{eventId}/files (POST)	Add a new file to an event
/orgs/{orgId}/events/{eventId}/files /{fileId} (PUT)	Update an event file
/orgs/{orgId}/events/{eventId}/files /{fileId} (DELETE)	Delete an event file
/orgs/{orgId}/events/{eventId}/agenda (GET)	Get agenda entries for an event
/orgs/{orgId}/events/{eventId}/agenda (POST)	Add a new agenda entry to an event
/orgs/{orgId}/events/{eventId}/agenda /{agendaId} (PUT)	Update an agenda entry

Endpoint	Beschreibung
/orgs/{orgId}/events/{eventId}/agenda/{agendaId} (DELETE)	Delete an agenda entry
/orgs/{orgId}/events/{eventId}/flows (GET)	Get list of flows
/orgs/{orgId}/events/{eventId}/flows (POST)	Create a new flow for event
/orgs/{orgId}/events/{eventId}/flows/{flowId} (GET)	Get flow details
/orgs/{orgId}/events/{eventId}/flows/{flowId} (PUT)	Change flow for event
/orgs/{orgId}/events/{eventId}/flows/{flowId} (DELETE)	Remove flow from event
/orgs/{orgId}/events/{eventId}/flows/{flowId}/actions (GET)	Get list of actions
/orgs/{orgId}/events/{eventId}/flows/{flowId}/actions (POST)	Create a new action
/orgs/{orgId}/events/{eventId}/flows/{flowId}/actions/{actionId} (GET)	Get detailed action data
/orgs/{orgId}/events/{eventId}/flows/{flowId}/actions/{actionId} (PUT)	Update action
/orgs/{orgId}/events/{eventId}/flows/{flowId}/actions/{actionId} (DELETE)	Delete an action
/orgs/{orgId}/events/{eventId}/flows/{flowId}/triggers (GET)	Get list of triggers
/orgs/{orgId}/events/{eventId}/flows/{flowId}/triggers (POST)	Create a new trigger
/orgs/{orgId}/events/{eventId}/flows/{flowId}/triggers/{triggerId} (GET)	Get detailed trigger data
/orgs/{orgId}/events/{eventId}/flows/{flowId}/triggers/{triggerId} (PUT)	Update trigger
/orgs/{orgId}/events/{eventId}/flows/{flowId}/triggers/{triggerId} (DELETE)	Delete a trigger

Endpoint	Beschreibung
/orgs/{orgId}/flows (GET)	Get flows for an organization
/orgs/{orgId}/flows (POST)	Create a new flow for an organization
/orgs/{orgId}/flows/{flowTemplateId} (GET)	Get a specific flow template
/orgs/{orgId}/flows/{flowTemplateId} (PUT)	Update a flow template
/orgs/{orgId}/flows/{flowTemplateId} (DELETE)	Delete a process step for an organization
/users/{userId} (GET)	Get a user by id
/users/{userId} (PUT)	Update a user by id
/users/{userId} (DELETE)	Delete a user by id
/users/{userId}/events (GET)	Get events the user is signed up for
/users/{userId}/orgs (GET)	Get all organizations the user is a member of

4.3 Frontend

Entsprechend den React best practises verwenden wir wiederverwendbare React-Komponenten, um die Redundanz im Code zu vermeiden. Um Zugriff auf den angemeldeten Nutzer und die ausgewählte Organisation in der gesamten Applikation zu erhalten, werden React-Kontexte eingesetzt.

4.3.1 Root-Layout und globale Provider

Im Root-Layout werden zentrale Provider wie der QueryClientProvider für React Query, der ThemeProvider zur dynamischen Themenanpassung sowie der UserOrgProvider eingebunden. Die Provider ermöglichen den Zugriff auf die spezifischen Hooks, beispielsweise den useUser Hook um den aktuell angemeldeten User zu erhalten. Diese Schicht garantiert einen konsistenten globalen Zustand und ein einheitliches Erscheinungsbild.

4.3.2 Layouts und Container

Das Hauptlayout organisiert den Seiteninhalt in einer flexiblen Containerstruktur und sorgt so für eine responsive Darstellung. Die Anordnung der Sidebar, Kopfzeile, Content-Bereich und anderer UI-Elemente erfolgt über klar definierte Komponenten, die den statischen Aufbau der Seite widerspiegeln. Dies verhindert ebenfalls Redundanz und sorgt für ein einheitliches Page Design.

4.3.3 Wiederverwendbare UI-Komponenten

Dank ShadCn welches auf Radix Primitives basiert, werden Elemente wie Cards, Buttons, Badges, Tabellen und Dialoge in eigenen wiederverwendbaren Komponenten umgesetzt. Diese Komponenten sind stark parametrisiert und erlauben so eine flexible Wiederverwendung in unterschiedlichen Kontexten (z.B. in Event-Übersichten, Flow-Dashboards oder Dateiverwaltungen).

4.3.4 Utility-First CSS mit Tailwind

Die Gestaltung erfolgt überwiegend über Tailwind CSS, wodurch statische Layouts durch Utility-Klassen direkt in den JSX-Komponenten definiert werden. Dies ermöglicht eine schnelle Iteration und klare Trennung von Design und Logik.

4.3.5 Modulares Routing

Das Frontend folgt der Next.js-äppStruktur, in der einzelne Seiten und Teilbereiche (z.B. Events, Flows, Dateien) in eigenen Routing Modulen realisiert werden. Diese statische Ordnerstruktur unterstützt eine klare Trennung der Verantwortlichkeiten und verbessert die Übersichtlichkeit.

4.3.6 API-Interaktion

Um mit der API zu interagieren, verwenden wir TanStack Query für die Verwaltung von Lade- und Fehlerzuständen. Aus Gründen der Entwicklerfreundlichkeit wird jeder mögliche API-Request in einem Client gekapselt, der dann die TanStack-Query-Schnittstelle mittels eines Hooks bereitstellt.