NEXTGEN
- DEVELOPMENT -

Designbeschreibung

NextGen Development

Version 1.1

4. Mai 2025

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

Verantwortlich für dieses Dokument:
Din Alomerovic

Inhaltsverzeichnis

1 Allgemeines

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

2 Produktübersicht

Unser Event-Management-System 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.

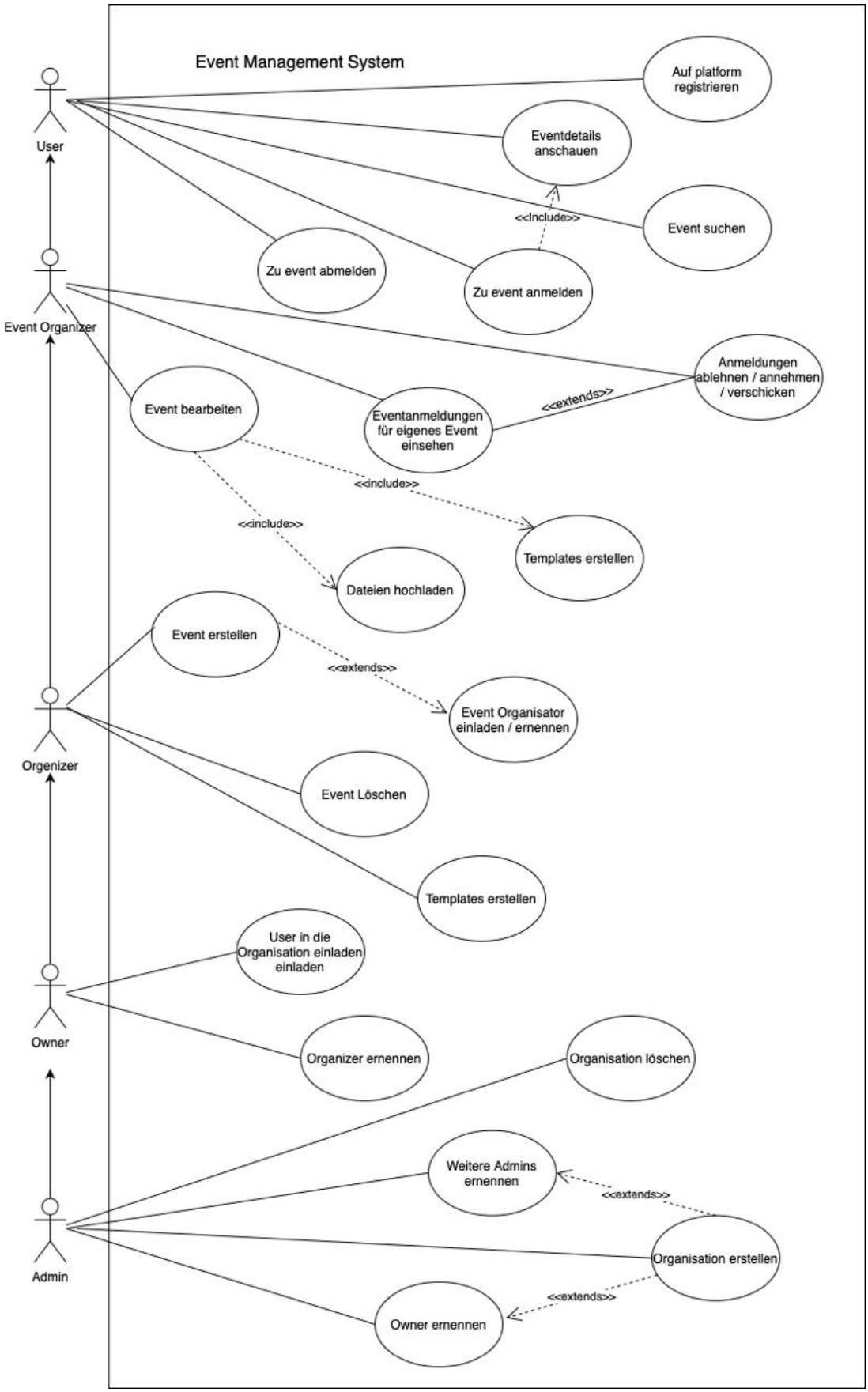

Abbildung 1: Use Case Diagramm

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 weitestgehend der Prototypenstruktur. Aus diesem Grund implementieren wir eine dreischichtige Architektur aus Frontend, Backend und Datenhaltung.

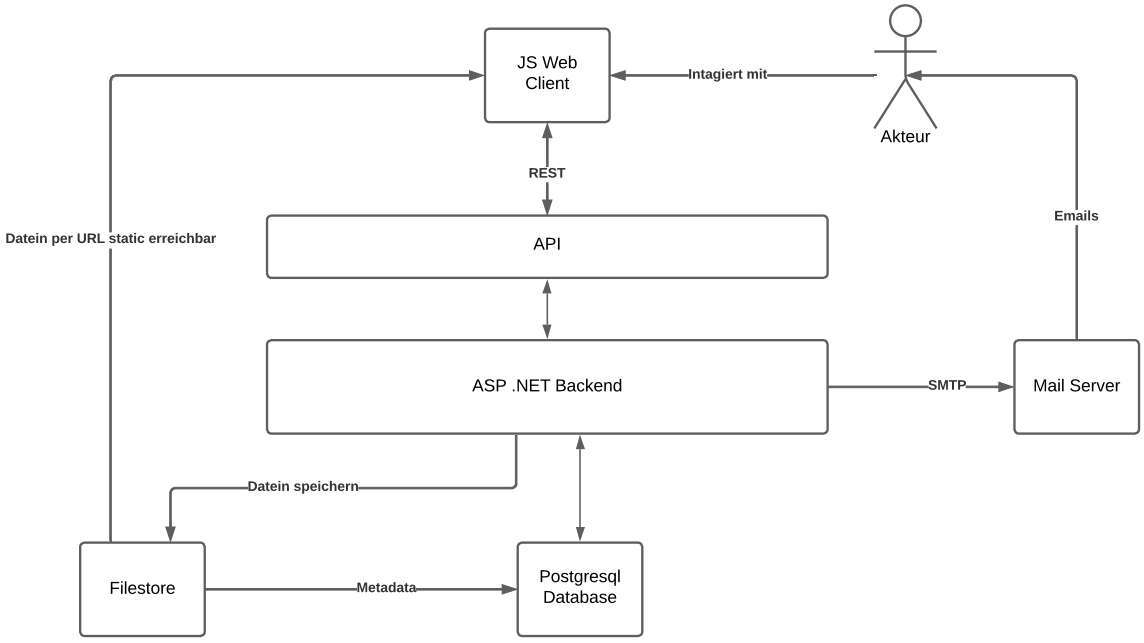

Abbildung 2: Genereller Aufbau

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 profi-

tieren 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.

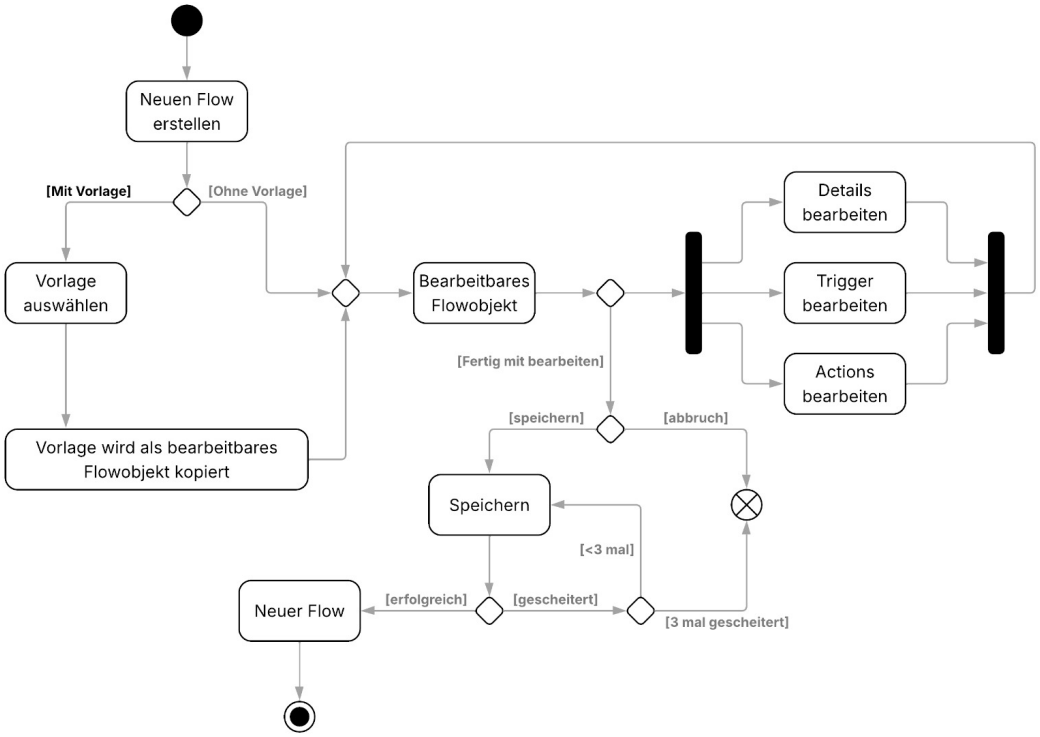

Abbildung 3: Erstellung von Flows

3.3 Benutzerregistrierung, Authentifizierung und Autorisierung

3.3.1 Benutzerregistrierung

Zur Registrierung füllt der Benutzer ein Formular mit folgenden Feldern aus:

- Vorname
- Nachname
- E-Mail-Adresse
- Passwort

Nach dem Absenden prüft das System automatisch, ob die E-Mail-Adresse bereits in der Datenbank existiert.

Fall 1: Die E-Mail-Adresse existiert bereits.
Die Registrierung wird abgebrochen, und der Benutzer wird aufgefordert, eine andere E-Mail-Adresse zu verwenden.

Fall 2: Die E-Mail-Adresse ist nicht vorhanden.
Die eingegebenen Daten werden gespeichert und ein neuer Benutzer wird angelegt. Standardmäßig erhält jeder neue Benutzer die Rolle `User`.

3.3.2 Benutzerdatenspeicherung

Die Speicherung erfolgt in einer PostgreSQL-Datenbank mittels `ASP.NET Core Identity`. Dabei werden folgende Informationen gespeichert:

- Vorname
- Nachname
- E-Mail-Adresse (eindeutig)
- Passwort (gehasht)

Für das Hashing kommt der PBKDF2-Algorithmus (mit HMAC-SHA256) zum Einsatz. Passwörter werden dadurch nicht im Klartext gespeichert. Die Kommunikation erfolgt verschlüsselt. Die Identity-Einstellungen (z.B. Passwortkomplexität, erlaubte Zeichen im Benutzernamen) werden in der Datei Program.cs wie folgt konfiguriert:

Listing 1: Identity-Konfiguration

```
services.Configure<IdentityOptions>(options =>
{
    options.Password.RequireDigit = true;
    options.Password.RequireLowercase = true;
    options.Password.RequiredLength = 8;
    options.User.AllowedUserNameCharacters =
        "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMN
--------OPQRSTUVWXYZ0123456789-._@+";
});
```

3.3.3 Authentifizierung (Login)

Identity-Konfiguration in `Program.cs`: Nach dem Login über ein Formular (E-Mail + Passwort) wird ein JSON Web Token (JWT) generiert und dem Client bereitgestellt.

Funktion des Tokens:

- Beweist die Identität des Benutzers bei HTTP-Requests
- Enthält Benutzerinformationen: ID, E-Mail, Rolle
- Ist digital signiert (z. B. mit HMACSHA256)

Beispielstruktur eines Tokens:

```
{
  "sub"   "user_id_123",
  "email"   "NextGen@mail.de",
  "role"   "User",
  "exp"   60
}
```

Das Backend verwendet JwtBearer zur Verarbeitung der Tokens. Die Konfiguration erfolgt in Program.cs wie folgt:

Listing 2: JWT-Konfiguration

```
services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
    .AddJwtBearer(options =>
    {
        options.TokenValidationParameters = new TokenValidationParameters
        {
            ValidateIssuerSigningKey = true,
            IssuerSigningKey = new SymmetricSecurityKey(
                Encoding.UTF8.GetBytes("SECRET_KEY")),
            ValidateIssuer = false,
            ValidateAudience = false,
            ClockSkew = TimeSpan.Zero
        };
    });
```

3.3.4 Verwendung des JWT-Tokens

Nach erfolgreicher Anmeldung erhält das Frontend den Token (z. B. im JSON-Body oder als HTTP-only-Cookie). Der Token dient fortan als Zugangsticket für geschützte Endpunkte.

Verwendungsszenarien:

- Zugriff auf geschützte Endpunkte (`/api/events`, `/api/orgs/{id}`)
- Aktionen wie Event erstellen, Benutzer aktualisieren usw.

Frontend-Verhalten:

- Speicherung des Tokens z. B. in einem `HttpOnly`-Cookie über HTTPS
- Mitsenden des Tokens im `Authorization`-Header:

  `Authorization: Bearer <JWT_TOKEN>`

- Dekodieren des Tokens zur Anzeige der Benutzerrolle oder E-Mail (keine Validierung, nur Lesezugriff)

3.3.5 Ablaufübersicht: Frontend → Backend

1. Benutzer loggt sich mit E-Mail und Passwort ein
2. Backend prüft Daten und erstellt Token
3. Frontend speichert Token
4. Weitere API-Aufrufe enthalten den Token im Header
5. Backend prüft Token-Gültigkeit und Rolle
6. Fehlerhafte Anfragen erhalten `401 Unauthorized` oder `403 Forbidden`

Token-Verfall und Logout:

- Jeder Token enthält ein Ablaufdatum (`exp`-Claim im Payload)
- Nach Ablauf: erneuter Login erforderlich
- Beim Logout: Token wird aus dem Frontend entfernt (z. B. Cookie gelöscht)

3.3.6 Autorisierung (Zugriffsrechte)

Rollenmodell im JWT:

User: Standardrolle nach Registrierung. Kann z. B. Events beitreten.

EventOrganizer: Darf Inhalte eines Events verwalten.

Organizer: Kann mehrere Events verwalten und erstellen.

Owner: Hat erweiterte Rechte über Events und Benutzer.

Admin: Systemweiter Administrator (z. B. Rollenzuweisung, Benutzer löschen)

Rollenzuweisung: Neue Benutzer erhalten automatisch die Rolle `User`. Höhere Rollen werden von berechtigten Nutzern vergeben.

```
{
  "sub"   "34523",
  "email"   "benutzer@gendev.de",
  "role"   "EventOrganizer",
  "exp"   60
}
```

Listing 3: Beispiel-Controller mit Rollen-Autorisierung

```
[Authorize]
[HttpGet("user/profile")]
public IActionResult GetUserProfile() { ... }
```

```
[Authorize(Roles = "Organizer")]
[HttpPost("event")]
public IActionResult CreateEvent(EventDto dto) { ... }

[Authorize(Roles = "EventOrganizer,Owner")]
[HttpPut("event/{id}")]
public IActionResult EditEvent(int id, EventUpdateDto dto) { ... }

[Authorize(Roles = "Admin")]
[HttpPut("user/role")]
public IActionResult ChangeUserRole(RoleUpdateDto dto) { ... }
```

3.3.7 Anwendungsszenarien

Zugriffssteuerung mit `[Authorize]`-Attribut:

Eventverwaltung (z. B. durch EventOrganizer):

- EventOrganizer dürfen eigene Events bearbeiten
- Organizer dürfen zusätzlich neue Events anlegen/löschen

Listing 4: Prüfung der Event-Zugehörigkeit

```
[Authorize(Roles = "EventOrganizer")]
[HttpPut("event/{id}")]
public IActionResult UpdateEvent(int id, EventUpdateDto dto)
{
    var userId = User.FindFirst(ClaimTypes.NameIdentifier)?.Value;
    var ev = _eventService.GetEventById(id);

    if (ev.EventOrganizerId != userId)
        return Forbid("Sie duerfen nur Ihre eigenen Events bearbeiten.");

    return Ok();
}
```

Zusammenfassung der Zugriffssicherung:

- Token ist gültig (Signatur + Ablauf)
- Benutzer besitzt erforderliche Rolle
- Zugriff nur auf eigene Ressourcen (z. B. Events)

Damit wird verhindert, dass:

- ein Benutzer Admin-Funktionen nutzt
- EventOrganizer fremde Events bearbeiten
- Admins systemweit Änderungen durchführen, wenn nötig

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.

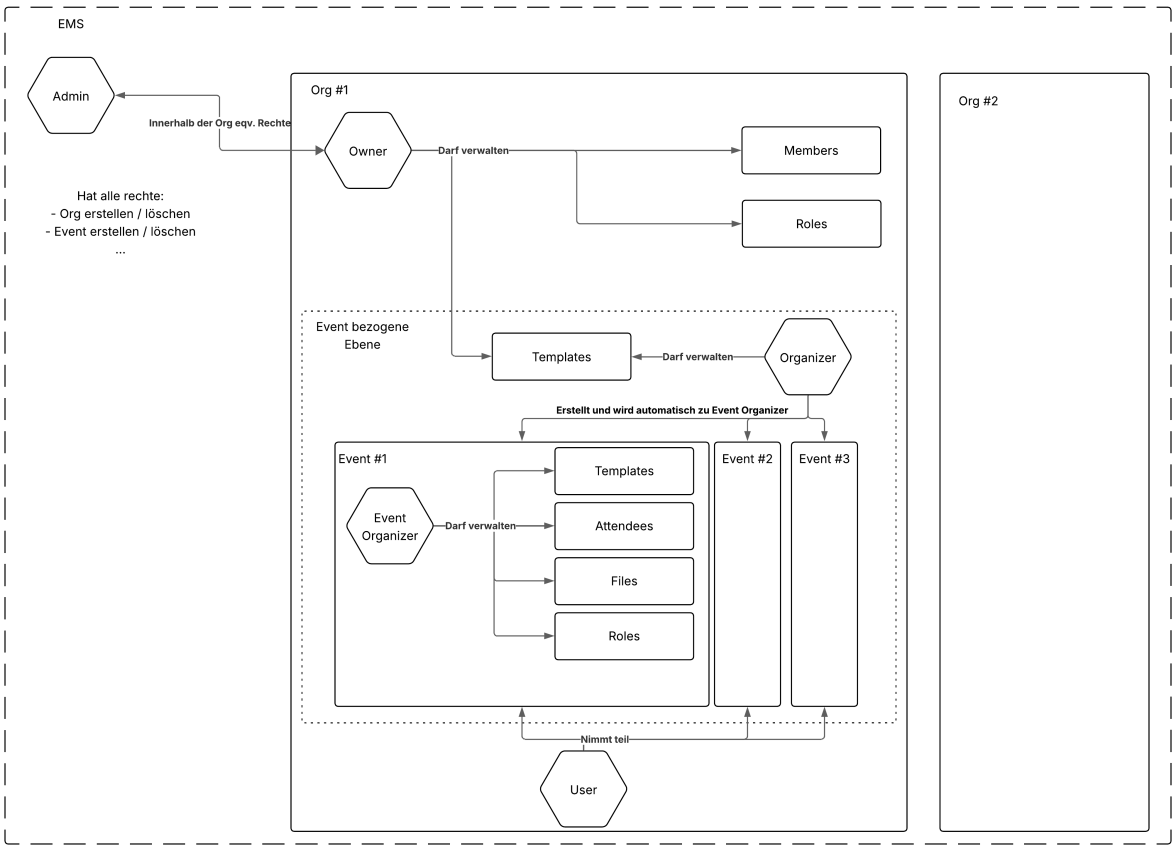

Abbildung 4: Übersicht Rollenverteilung

3.5 Erstellung eines Events

Das Sequenzdiagramm in Abbildung 5 soll verdeutlichen, wie die Kommunikation unserer Komponenten untereinander während der Erstellung eines Events funktioniert.

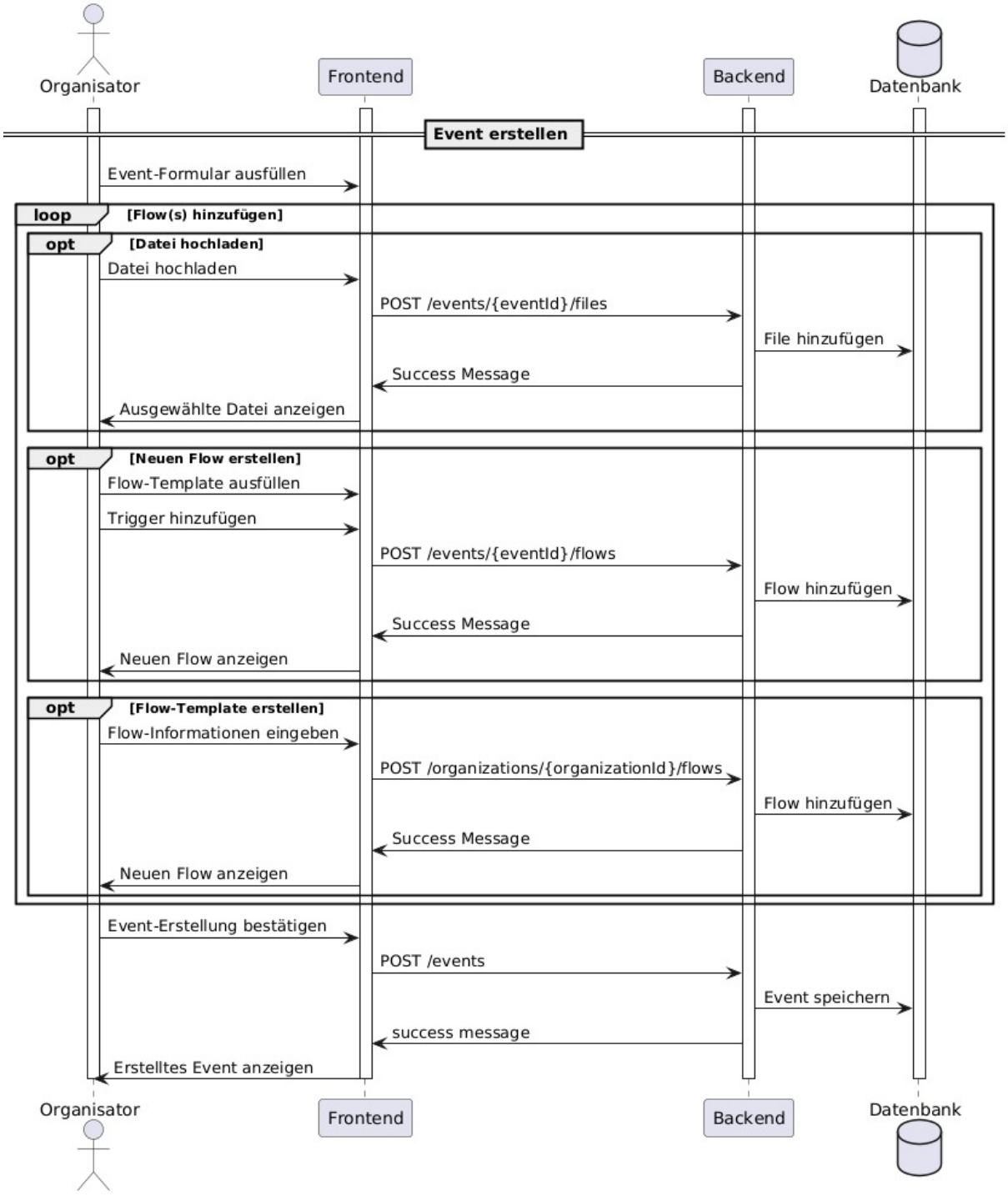

Abbildung 5: Event Erstellung

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 6 zeigt den Aufbau am Beispiel der User Klassen:

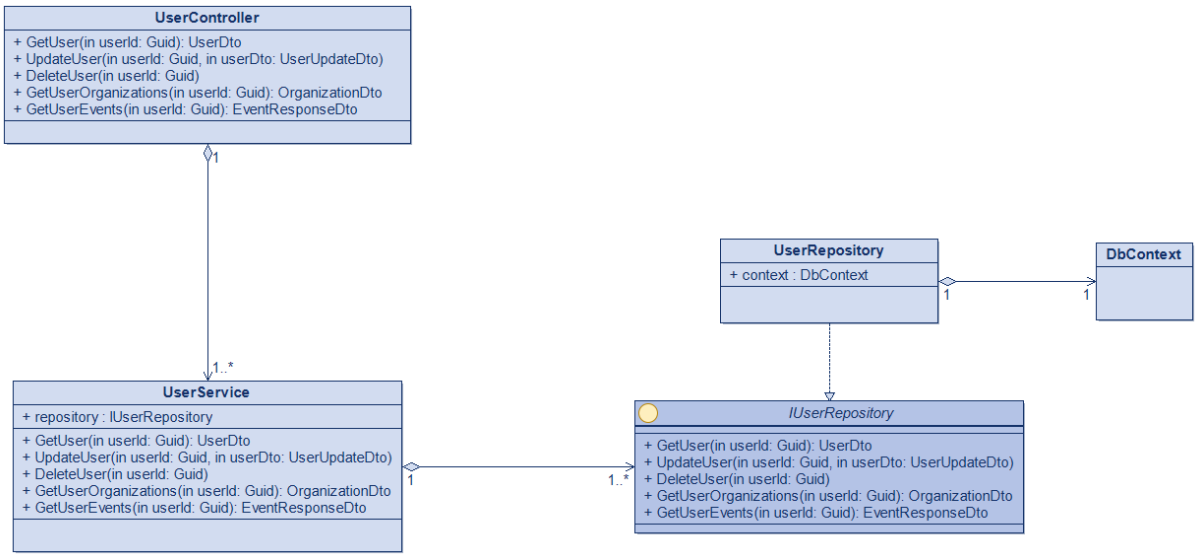

Abbildung 6: User Beispiel

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

Um die Beziehung zwischen den einzelnen Services zu verdeutlichen, bietet Abbildung 7 einen Überblick über unser Backend. Zu den dort dargestellten Klassen kommen zusätzlich jene hinzu, die in Abbildung 8 beschrieben sind. Diese umfassen unter anderem Jobs, welche in regelmäßigen Abständen das Eintreten von Flows überprüfen. Dabei kommt das Quartz-Package zum Einsatz, das für das automatisierte Scheduling und die Ausführung der Jobs verantwortlich ist. Alle zwei Minuten wird der CheckFlowsJob ausgeführt, der die Flows-Tabelle abfragt und prüft, ob ein Flow bereits ausgeführt wurde, aktuell ausgeführt wird oder mehrfach ausgeführt werden soll. Über die zugehörigen FlowIds werden alle relevanten Trigger ermittelt. Anschließend wird für jeden Flow ein Eintrag in der FlowRun-Tabelle erstellt. Ein FlowRun durchläuft dabei verschiedene Stati, die im Enum FlowRunStatus definiert sind. Die Trigger selbst speichern ihre Details in JSON-Form in der Datenbank, wobei der Aufbau je nach Typ variiert – für die Verarbeitung werden entsprechende Trigger-Klassen und ein zugehöriges Enum verwendet. Nach dem Abruf der benötigten Event-Daten aus der Datenbank wird basierend auf der definierten Aktion die entsprechende Execute-Methode ausgeführt.

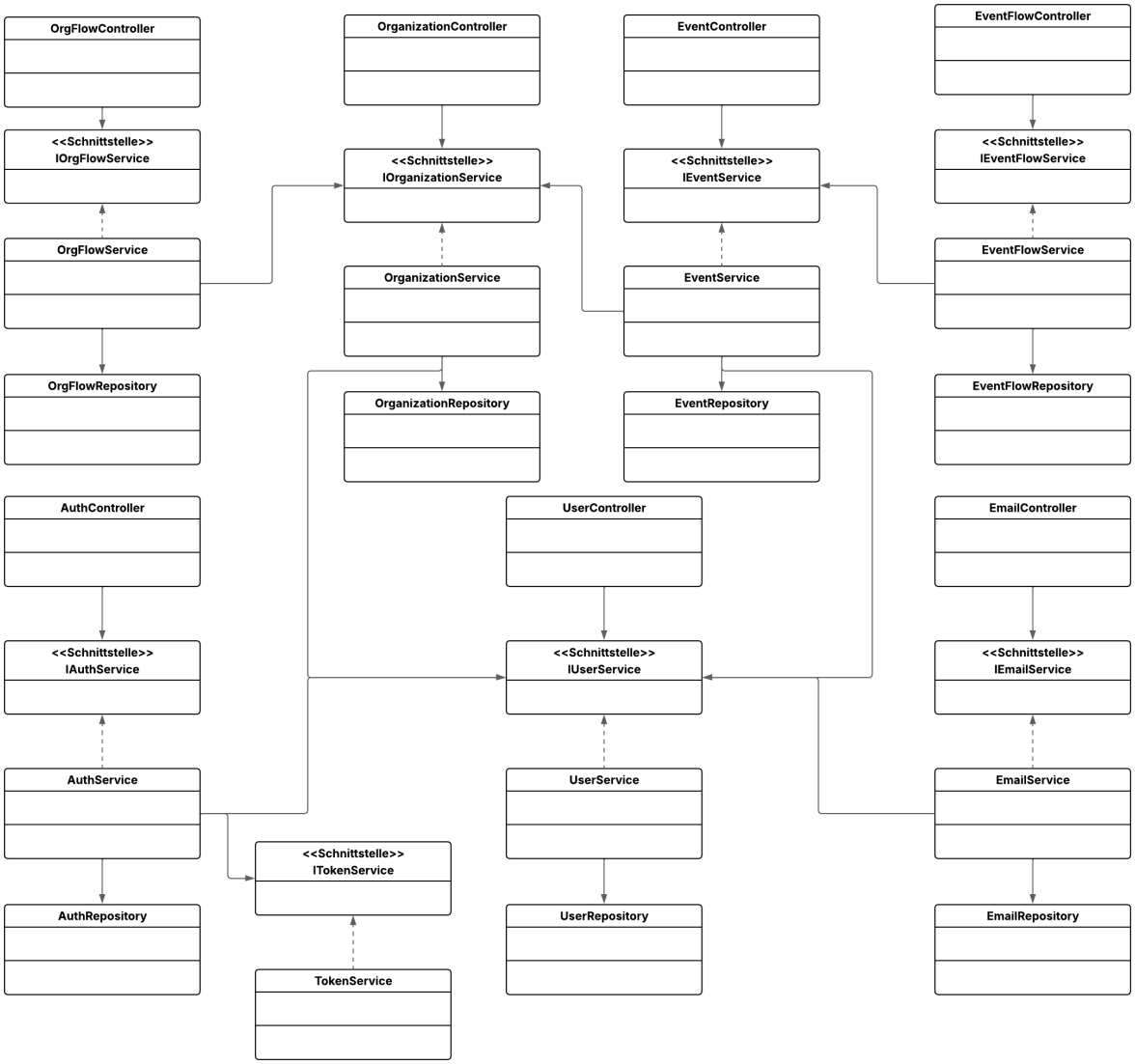

Abbildung 7: Grober Aufbau

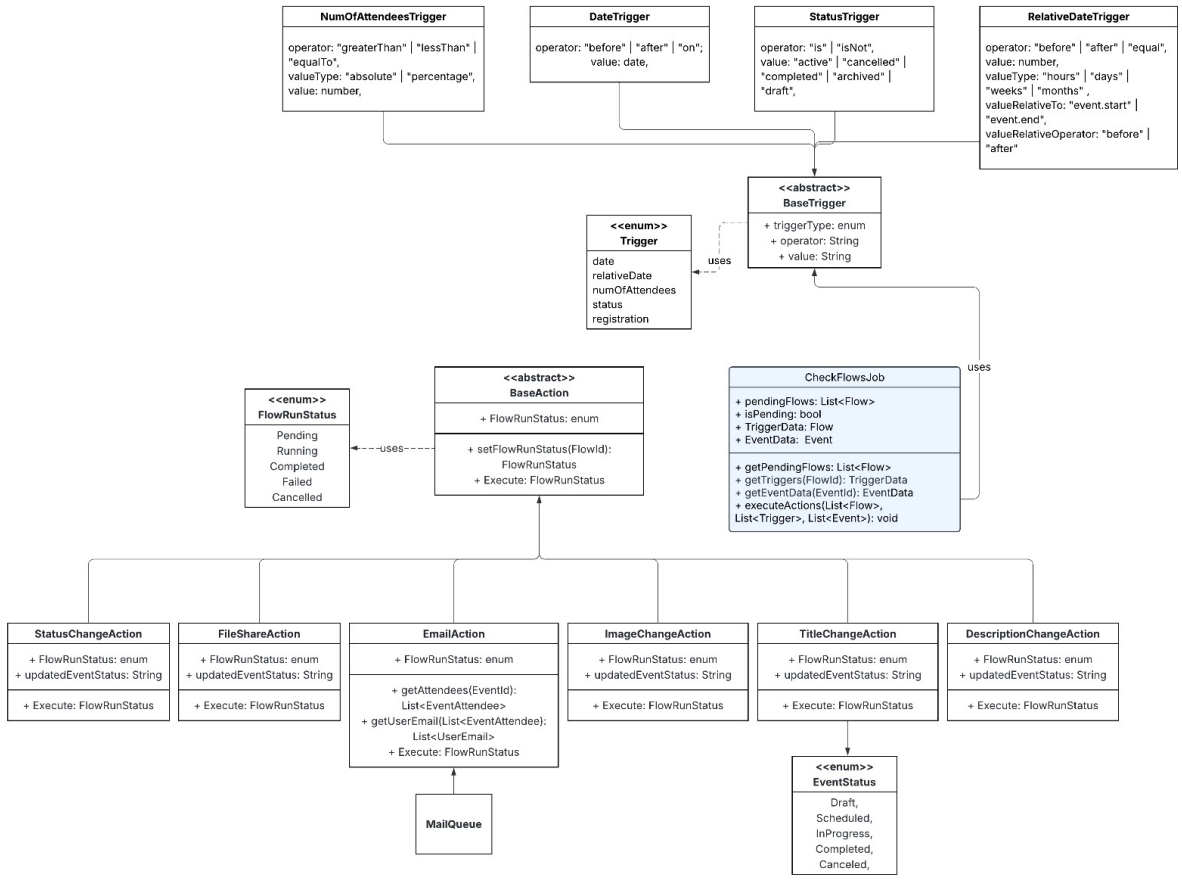

Abbildung 8: Aufbau der Jobs

4.2 API

Dieser Abschnitt dokumentiert sämtliche Endpunkte der API und gibt einen Überblick über deren Funktionalität.

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

Endpoint	**Beschreibung**
/orgs/{orgId}/emails (POST)	Create an email template
/orgs/{orgId}/members (GET)	Get members of an organization
/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

Endpoint	Beschreibung
/orgs/{orgId}/events/{eventId}/agenda/{agendaId} (PUT)	Update an agenda entry
/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

Endpoint	Beschreibung
/orgs/{orgId}/events/{eventId}/flows /{flowId}/triggers/{triggerId} (DELETE)	Delete a trigger
/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

4.3.1 Übersicht

Das Frontend unserer Anwendung wird mit React entwickelt. Dieses Framework ermöglicht eine komponentenbasierte Architektur, wodurch sich die Benutzeroberfläche modular und übersichtlich aufbauen lässt. Für die Gestaltung kommt TailwindCSS zum Einsatz. Das Utility-First-Framework erlaubt ein einheitliches, responsives Design und erleichtert die schnelle Umsetzung eines modernen, funktionalen Layouts.

Die Benutzeroberfläche ist für die Verwendung auf Desktop- und Laptop-Geräten ausgelegt, funktioniert aber auch zuverlässig auf Tablets wie dem iPad. Die Seitenstruktur besteht aus einer festen Sidebar zur Hauptnavigation sowie einer Breadcrumb-Leiste zur besseren Orientierung im Kopfbereich.

Zur Umsetzung der Navigation verwenden wir react-router-dom, womit eine clientseitige

Routenverwaltung realisiert wird. So können Nutzer zwischen den verschiedenen Ansichten – z.B. Dashboard, Eventübersicht, Eventverwaltung oder Organisation – wechseln, ohne dass die Seite neu geladen werden muss.

Das Frontend kommuniziert über eine RESTful API mit dem Backend. Die Daten werden dabei im JSON-Format ausgetauscht. Die Schnittstellenanbindung erfolgt über einen zentralen Service-Layer, wodurch die API-Aufrufe vom restlichen Anwendungscode getrennt sind und leicht gewartet werden können.

Je nach Benutzerrolle (User, Organizer, Owner, Admin) werden unterschiedliche Inhalte und Funktionen im Frontend angezeigt. Während einfache Nutzer lediglich die angebotenen Events und ihre Eventteilnahmen sehen, können Organisatoren Events anlegen und verwalten. Für administrative Rollen stehen zusätzliche Funktionen wie die Mitgliederverwaltung zur Verfügung.

Ein zentrales Feature der Anwendung ist die Möglichkeit, sogenannte „Flows“ zu erstellen. Diese dienen der Automatisierung von Abläufen – beispielsweise dem automatischen Versand von E-Mails oder der Veröffentlichung von Dateien zu bestimmten Zeitpunkten vor einem Event. Die Flows lassen sich flexibel definieren und individuell mit Events verknüpfen.

4.3.2 Architektur und Strukturprinzipien

Das Frontend basiert auf React mit TypeScript und ist modular aufgebaut. Ziel ist eine saubere Trennung der Verantwortlichkeiten sowie eine hohe Wiederverwendbarkeit und Wartbarkeit. Das Design folgt modernen UI-Prinzipien und verwendet TailwindCSS für schnelles, konsistentes Styling. Die Gesamtstruktur ist in funktionale Bereiche gegliedert, die klar voneinander abgegrenzt sind – z.B. Seiten, Komponenten, Services, Kontexte und Hilfsfunktionen.

4.3.3 Seitenaufbau und Navigation

Das Frontend nutzt das moderne App Router System von Next.js, das eine dateibasierte Routing-Struktur implementiert. Die Organisation der Seiten folgt einem hierarchischen Konzept mit definierten Anwendungsbereichen. Der Haupteinstiegspunkt befindet sich unter app/, wobei der (withSidebar)-Ordner als Container für alle authentifizierten und mit Seitenleiste versehenen Routen dient. Die Klammernotation [(...)] signalisiert, dass diese Gruppierung keine Auswirkung auf die URL-Struktur hat.

Die Verzeichnisstruktur bildet direkt die URL-Pfade ab, wie bei /organisation/events/-create, wodurch ein intuitives und vorhersehbares URL-Schema entsteht. Durch die Verwendung von page.tsx Dateien werden die tatsächlichen Seitenkomponenten definiert, die unter der entsprechenden Route gerendert werden. Die Client-Komponenten sind mit üse client"gekennzeichnet, um clientseitige Interaktivität zu ermöglichen. Zu den wichtigsten Seiten gehören:

- LoginPage: Login mit E-Mail und Passwort.
- HomeDashboard: Einstieg nach erfolgreicher Anmeldung und Übersicht der angemeldeten Events eines Users.
- EventÜbersicht: Übersicht und Filterung aller Events.
- EventErstellen: Formular zur Erstellung neuer Veranstaltungen.
- EventDetails: Detailinformationen und Interaktion zu einzelnen Events.
- Organisation: Verwaltungsansicht für Organisationseinträge.

Die Implementierung nutzt den useRouter-Hook für programmatische Navigation (z.B. router.push(/organisation/events") nach erfolgreicher Event-Erstellung) und ermöglicht so flüssige Übergänge zwischen verschiedenen Anwendungsbereichen. Die klare Trennung zwischen Routing-Struktur und Komponentenlogik erhöht die Wartbarkeit und erlaubt es, Seiten unabhängig voneinander zu entwickeln und zu aktualisieren, während das übergeordnete Navigationsgerüst konsistent bleibt.

4.3.4 Komponentenaufbau und UI-Konzept

Alle UI-Komponenten der Anwendung sind im Verzeichnis components/ organisiert und nach funktionalen sowie rollenbasierten Gesichtspunkten strukturiert. Dies fördert die Wiederverwendbarkeit, Übersichtlichkeit und Wartbarkeit des Codes.

Die Struktur gliedert sich hauptsächlich in folgende Bereiche:

- org/: Beinhaltet Komponenten zur Verwaltung der Organisation sowie spezifische Event- und Flow-bezogene Elemente.
- events/: Komponenten für Event-Tabellen oder -Karten.
- flows/: Komponenten zur Darstellung und Bearbeitung von Automatisierungsregeln (Flows).

- Weitere Komponenten: z.B. email-template-form.tsx, file-upload-dialog.tsx, team-members.tsx.
- user/: Beinhaltet Komponenten, die auf die Nutzeransicht zugeschnitten sind.
- dashboard/: Elemente für das User-Dashboard.
- event-layout/: Komponenten zur Filterung, Eingabe und Ansicht von Eventdaten (z.B. active-filters.tsx, filter-dropdown.tsx, date-input.tsx).
- ui/: Container für generische UI-Komponenten und universelle Elemente wie Buttons, Modals usw.

Weitere lose strukturierte Komponenten im components/-Root welche sowohl in der Organisator Ansicht als auch un der Use Ansicht verwendet werden:

- Navigationsleisten: z.B. nav-admin.tsx, nav-events.tsx, nav-main.tsx, nav-user.tsx, passend zu den jeweiligen Rollen.
- Formulare & Layouts: z.B. login-form.tsx, chart-area-interactive.tsx, data-table.tsx.
- Layout-Komponenten: wie app-sidebar.tsx für die Hauptnavigation.

4.3.5 Datenkommunikation und Service-Layer

Für die Kommunikation mit der API von unserem Backend wird ein Service Layer implementiert. Zu finden ist dies unter lib/backend und ist enstprechend den Kategorien der API in einzelne Datein aufgeteilt. Das Service Layer abstrahiert dabei die Next.Js fetch requests, bei dem es sich um eine um Caching und Revalidation erweiterte Web fetch() API handelt, welche unsere API ansprechen mittels exposed async functions. Diese wiederum werden, um die verwendenung im Client Side rendering Fall zu vereinfachen, mit dem Tanstack Query useQuery Hook gewrappt und exposed. Dieser übernimmt das loading state und error management und reduziert dadurch Boilerplate. Um die Autorization Requirements der API zu erfüllen wird der beim Anmelden von der API erhaltene JWT Bearer Token stets im Authorization Header mitgeschickt.

4.3.6 Zustandsmanagement und Systemmeldungen

Pagebezogene Zustände werden lokal in Komponenten über React Hooks (useState, useEffect) verwaltet. Global verwendete Systeme wie Auth und das Toast System werden mittels um das Root Layout gewrappte Contexte beretigestellt. Diese exposen dabei die in

den Kontexten vorhanden Variablen und Hooks an alle children des jeweiligen Kontextes, in diesem Fall also an alle Children des Root Layouts entsprechen an alles in der Applikation. Das zentrale Toast-System (ToastProvider) erlaubt es, aus beliebigen Komponenten heraus visuelle Rückmeldungen auszulösen – etwa zur Bestätigung von Aktionen oder zur Anzeige von Fehlern. Zur Sessionpersistenten Speicherung des JWT Bearer Tokens setzen wir auf einen HttpOnly Cookie und verwenden diesen ausschließlich in sicheren Transportmodi per HTTPS.

4.3.7 Benutzerführung und Interaktionsdesign

Die Benutzeroberfläche ist auf Klarheit und Effizienz ausgelegt. Zentrale Funktionen wie Event-Erstellung, Filterung, Suche oder Anmeldung sind sofort zugänglich. Buttons und UI-Elemente folgen einem einheitlichen Stil mit visueller Unterscheidung zwischen Haupt- und Nebenaktionen.

Mehrschrittige Prozesse wie die Event-Erstellung sind in übersichtliche Schritte untergliedert, mit Zwischenüberschriften, Zustandsanzeigen und klarer Benutzerführung. Das gesamte Interface ist responsive bis zur Tabletgröße (z.B. iPad) optimiert.

4.3.8 Technische Trennung und Wartbarkeit

Das Frontend ist nach dem Prinzip der Separation of Concerns aufgebaut, was eine klare technische Trennung zwischen Präsentationslogik, Geschäftslogik und Datenverwaltung gewährleistet. Komponenten folgen dem SSingle ResponsibilityPrinzip, sodass beispielsweise Formulare wie EventBasicInfoForm oder Filter-Logik in useEventFilters isoliert und eigenständig gewartet werden können. Die Modularisierung, besonders sichtbar in der Tabs-Struktur der Event-Erstellung, erlaubt es, einzelne Funktionalitäten unabhängig voneinander zu entwickeln und zu testen. Datenstrukturen und Validierungslogik wie in form-schemas.ts sind zentral definiert und wiederverwendbar. Die konsequente Verwendung von TypeScript mit klar definierten Interfaces und Props-Typisierungen erhöht die Wartbarkeit, da Typfehler bereits zur Entwicklungszeit erkannt werden. Die Kombination aus funktionalen React-Komponenten, Custom Hooks für geteilte Logik und der Nutzung von Shadcn-UI als konsistente Komponentenbibliothek sorgt für eine nachhaltige Codebasis, die einfach zu erweitern und zu warten ist.

4.4 Datenbank

Das Entity-Relationship Diagramm in Abbildung 9 stellt die Grundstruktur unserer PostgreSQL Datenbank dar.

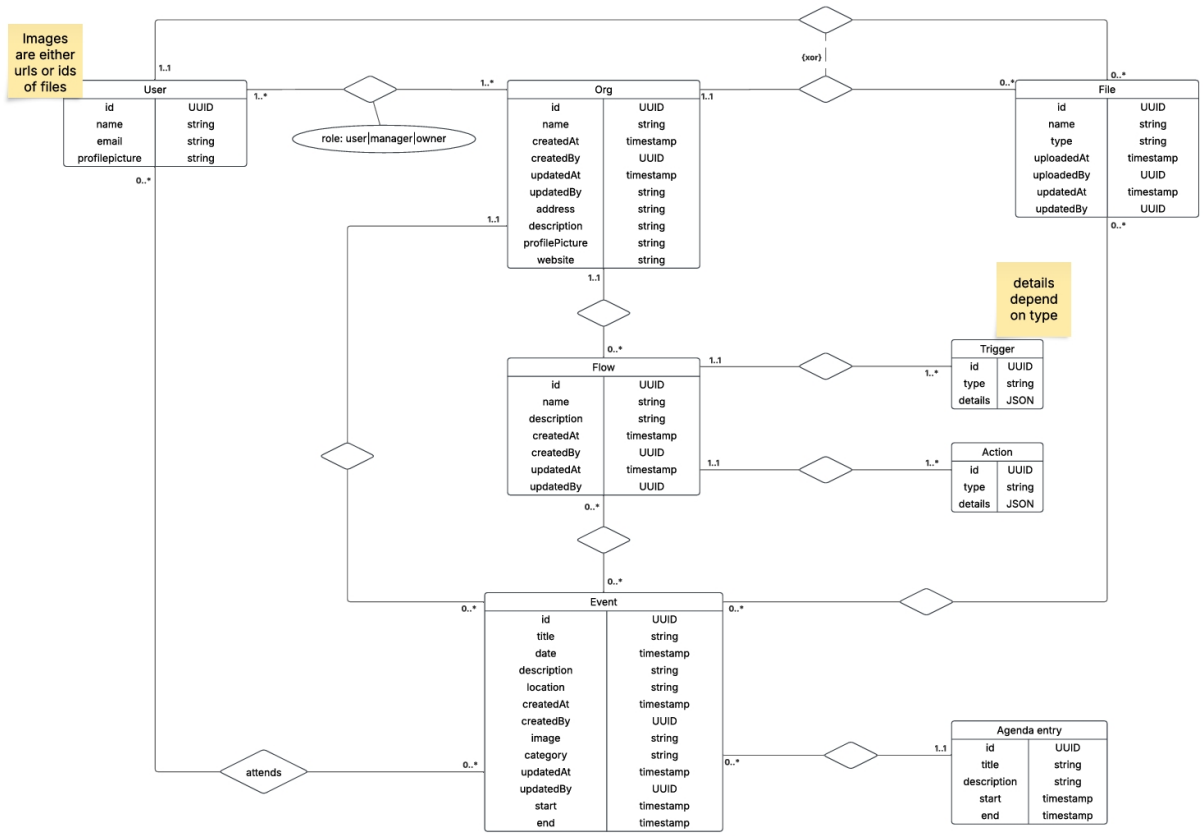

Abbildung 9: Aufbau der Datenbank