Projekt Dokumentation

Integrationsprojekt CAS Webapplikationen

Datum: 2.10.2025

Studierende:

- Damir Mavric
- Helen Aerni
- Thomas Wenger

Dozent: Jonas Bandi

1 Thema

Organisations- und Abstimmungstool für gemeinsame Aktivitäten.

2 Umfeld, Ausgangslage

Heutzutage haben alle immer viel los und volle Kalender. Die Organisation eines Freizeit-Events mit Freunden kann daher oft zur echten Herausforderung werden. Unterschiedliche Verfügbarkeiten, verschiedene Interessen und Vorstellungen vom "perfekten Abend" führen häufig zu langen, unübersichtlichen Chatverläufen und Abstimmungen, die sich über Tage oder Wochen ziehen. Um diesen Prozess effizienter und übersichtlicher zu gestalten, entwickeln wir eine App zur Terminfindung und Event-Organisation. Mithilfe mehrstufiger Abstimmungen – zu Datum, Aktivität und Ort – soll es Gruppen ermöglicht werden, gemeinsame Freizeitaktivitäten einfach, schnell und demokratisch zu planen. Unser Ziel ist es, die Koordination innerhalb von Freundesgruppen zu vereinfachen und spontane sowie geplante Treffen wieder unkomplizierter zu machen.

3 Aufgabenstellung & Ziel des Projektes

Das Ziel ist es, eine App zu haben, welche den Prozess der Erstellung/Planung eines Events für eine Person vereinfacht, Ihm die nötigen Optionen für das Planen eines Events frei zur Verfügung stellt und Ihm/Ihr die Möglichkeit gibt, die gewünschten Personen in diesem Event einzuladen.

Dem Organisator des Events sollten die Möglichkeiten gegeben werden zu bestimmen, wann das Event geplant wird, was gemacht werden kann und wo das Event stattfinden könnte.

Optionen können entweder als Umfrage erstellt, oder bei der Erstellung fix definiert werden.

Mehrere Optionen können vom Organisator eingetragen werden, um eine Umfrage zu starten.

Ein typischer Anwendungsfall wäre die Organisation eines gemeinsamen Kinoabends. Der Organisator erstellt dazu ein neues Event mit dem Titel „Kino Film schauen“ und möchte zunächst folgende Informationen von den eingeladenen Personen einholen:

- Wer möchte mitkommen?
- Welcher Film soll geschaut werden?
- In welchem Kino (Ort)?
- An welchem Datum?

Nachdem die erste Umfrage versendet wurde, antworten die eingeladenen Personen mit ihren Präferenzen. Auf Basis dieser Rückmeldungen kann der Organisator eine zweite, detailliertere Umfrage starten, um den Kinoabend final zu planen.

Dabei könnten folgende Punkte geklärt werden:

- Zu welcher Uhrzeit soll der Film geschaut werden?
- Wer übernimmt die Fahrt und kann andere mit dem Auto abholen?
- Wer verfügt über einen Studentenrabatt?

Mit diesem schrittweisen System lässt sich ein Event effizient und transparent organisieren, sodass am Ende ein Ergebnis entsteht, mit dem alle Beteiligten zufrieden sind.

4 MVP – Muss Funktionen

Der Einfachheit halber aus der Projektspezifikation übernommen und mit ✓ versehen, wenn es umgesetzt ist. Die nicht umgesetzten Punkte haben eine *kursive Begründung*.

4.1 Event-Erstellung

4.1.1 Generell

- ✓ Alle UserInnen können einen neuen Event erstellen.
- ✓ Die Person, die einen Event erstellt, wird automatisch als OrganisatorIn festgelegt und erhält besondere Verwaltungsrechte für diesen Event.
- ✓ OrganisatorInnen nehmen selbst an allen Abstimmungen teil.
- ✓ Ein Event kann erst vom Organistor für die eingeladenen User zur Abstimmung freigegeben werden, wenn mindestens eine weitere Person eingeladen wurde und mindestens eine Abstimmung definiert wurde.
- ✓ Der Organisator definiert, ob bei einer Abstimmung mehrere Antworten oder nur eine Anwort möglich ist. (Checkboxen oder Radiobuttons)
- ✓ Ein Event kann vom Organistor beendet werden. Das heisst, die TeilnehmerInnen können nicht mehr abstimmen und der Event erscheint als abgeschlossener Event.

4.1.2 Ort und Datum

- ✓ Damit ein Event vom Organistor beendet werden kann braucht er mind. ein Ort und ein Datum.
- ✓ Bei Ort und Datum kann beim Erstellen des Events ein “später definieren” gesetzt werden. So kann z.B. zuerst darüber Abstimmen was man machen möchte und erst später die Daten und die Orte hinzufügen.
- ✓ OrganisatorInnen können mehrere mögliche Termine und Orte hinzufügen.
- ✓ Bei mehreren Optionen entsteht automatisch eine Abstimmung.
- Bei nur einer Option gilt dieser Ort/dieses Datum als gesetzt.
    - *-> Diese Abfrage ist nicht Umgesetzt, es werden nur die Stimmen ausgewertet (wenn jemand für die einzige Option abgestimmt hat, ist es als Resultat drin)*

4.1.3 Individuelle Fragen und Folgeabstimmungen

- ✓ OrganisatorInnen können zusätzliche individuelle Abstimmungen erstellen (z. B. „Welchen Film möchten wir schauen?“ mit Antwortoptionen wie „Batman“, „Forrest Gump“, „Die nackte Kanone“).
- ✓ Auch nachträglich (wenn der Event schon für die User freigegeben wurde) können neue Abstimmungen hinzugefügt werden – z. B. aufbauend auf bereits getroffenen Entscheidungen.
- ✓ Die Anzahl zusätzlicher Fragen/Abstimmungen ist nicht begrenzt.

4.1.4 Event-Verwaltung

- Der Organisator kann die von ihm erstellten Events bearbeiten:
  - ✓ Neue Abstimmungen hinzufügen
  - Den Event als beendet erklären, wenn mind. ein Ort und ein Datum definiert wurden.
    - *Abfrage nicht vorhanden, aber Beenden ist möglich*
  - Abstimmungen löschen und bearbeiten, wenn noch niemand abgestimmt hat
    - *Löschen geht, Bearbeiten nicht, ob wer abgestimmt hat, ist dabei nicht berücksichtigt. Es gibt aber vor dem Erstellen eine Übersicht, um die Angaben zu prüfen.*
- Der Organisator kann nicht:
  - Bestehende Abstimmungen (mind. 1 User hat schon abgestimmt) verändern/löschen/erweitern.
    - *Sperre ist nicht drin*
- Events können vom Organisator jederzeit als beendet definiert werden, sofern mindestens ein Datum und ein Ort festgelegt wurden.
  - *Abfrage nicht vorhanden, aber Beenden ist möglich*

4.2 Teilnahme an Abstimmungen

4.2.1 Dashboard mit Event-Übersicht

- Übersicht über alle offenen Events, an denen der eingeloggte User beteiligt sind.
  - *Events, für die der User bereits abgestimmt hat, sind nur in den Messages aufgelistet, damit das Dashboard übersichtlicher bleibt.*
- ✓ Status-Anzeige pro Event:
  - Eigene Aktion erforderlich (Abstimmen)
  - Warten auf andere TeilnehmerInnen
  - Alle haben abgestimmt -> Event abgeschlossen

4.2.2 Event-Detailansicht

- ✓ Beim Klick auf ein Event im Dashboard gelangen UserInnen zur Detailansicht.
- ✓ In der Event-Detailansicht können User an laufenden Abstimmungen teilnehmen.
- ✓ Bereits abgegebene Stimmen anderer TeilnehmerInnen sind einsehbar (vergleichbar mit WhatsApp-Umfragen).
- Bei Ortsabstimmungen werden die vorgeschlagenen Orte auf einer Karte dargestellt.
  - *Wir sind leider nicht dazu gekommen, die Karte einzubinden*

4.3 Benachrichtigungen

- ✓ Die UserInnen erhalten Benachrichtigungen, wenn sie zu neuen Events und Abstimmugen eingeladen werden oder wenn Abstimmungen abgeschlossen wurden.
  - *Einfach nicht alles mit Live Aktualisierung*
- Der Organisator erhält eine Benachrichtigung, wenn alle eingeladenen User an allen Abstimmungen teilgenommen haben.
  - *Das ist nicht umgesetzt*

4.4 Abgeschlossene Events

- ✓ Übersicht aller Events, die von den OrganisatorInnen beendet wurden.
  - *Es werden auch die noch offenen Events aufgelistet*
- Geordnet nach Datum
  - *Neuste zuerst (anhand id) da auch Events, die noch nicht abgeschlossen sind, dabei sind*

4.5 Nice-to-have

- ✓ Profilseite für UserInnen
- Anzeige für abgeschlossene Events, die in der Vergangenheit liegen
- Stichentscheid bei unentschiedenen Umfragen

4.6 Gemockte Funktionen

- User-Registrierung -> *war nicht nötig*
- ✓ User-Login

5 Systemübersicht, Grob-Architektur, Deployment

Bestandteile des Projektes, Lösung, Systemgrenzen, Umsysteme, Deployment-Struktur

- ✓ Aufbau mit Supabase Database und React
- ✓ Entwicklung mit Vite, NPM
- ✓ Abgleich und Versionierung mit GitHub
- ✓ https://fun-organizer.ch/ für Prod-Build -> Deployment manuell per FTP
- ✓ Entwicklung Mobile First

6 Eingesetzte Technologien

Benutzte Frameworks, Libraries, Tools ..

- ✓ Supabase Database: Backend as a Service mit relationaler DB, Userverwaltung und Authentifizierung
- ✓ Supabase JavaScript Client Library: Erlaubt Datenbank Querys und unterstützt Typescript -> Typensicherheit zwischen Projekt und Datenbank
- ✓ React: Erlaubt den flexiblen Einsatz der geplanten Komponenten
- ✓ Typescript: Für Typesicherheit im Javascript / React
- ✓ Tanstack Router: Navigation / Anzeige richtiges Formular
- Mapbox API: Für Adresssuche und Map
  - *Sind wir zeitlich nicht dazu gekommen*
- ✓ Shadcn und Tailwind: Komponenten Library
- ✓ Zod: Für Überprüfung der Daten

7 Design und Umsetzung

Aufbau und Strukturierung der Umsetzung, Design-Entscheide mit Begründungen.

Dieses Kapital wurde komplett mit Claude Code generiert und noch leicht angepasst.

7.1 React und TypeScript

Für die Entwicklung des Frontends wurde React 19 in Kombination mit TypeScript gewählt. React ermöglicht durch seine komponentenbasierte Architektur eine klare Strukturierung der Anwendung. Im Fun-Organizer wurde dies konsequent umgesetzt: UI-Komponenten wie Buttons, Cards und Dialoge sind eigenständige, wiederverwendbare Einheiten. Beispielsweise wird die Button-Komponente an über 20 Stellen im Projekt verwendet, mit verschiedenen Varianten wie default, destructive und outline.

TypeScript erweitert JavaScript um statische Typen und ermöglicht Fehlerprüfung bereits während der Entwicklung. Im Projekt führte dies zu deutlich weniger Laufzeitfehlern. Besonders wertvoll war dies bei der Integration mit TanStack Router, wo Route-Parameter und Loader-Daten vollständig typisiert sind. Ein Beispiel: Die Route /events/$eventId hat einen typisierten eventId-Parameter, sodass fehlerhafte Zugriffe bereits beim Kompilieren erkannt werden.

7.2 TanStack Router

TanStack Router wurde als Routing-Lösung gewählt, da es vollständige TypeScript-Integration und file-based Routing bietet. Die URL-Struktur spiegelt direkt die Ordnerstruktur wider: Eine Datei unter routes/_authenticated/events/events.tsx wird automatisch zur Route /events/events.

Besonders wertvoll ist das Konzept der Layout Routes. Die Datei __root.tsx definiert das grundlegende Layout mit Header und Footer für alle Seiten. Die Datei _authenticated.tsx fungiert als Middleware-Layer und prüft in der beforeLoad-Funktion, ob der Benutzer authentifiziert ist. Alle Routes unterhalb von _authenticated/ erben diese Prüfung automatisch. Dies eliminiert Code-Duplikation und macht die Auth-Logik zentral wartbar.

Das integrierte Data Loading System verhindert Waterfall-Requests. Im Dashboard-Loader werden invitedEvents und answers parallel geladen, bevor die Komponente rendert. Dies ist deutlich performanter als der klassische useEffect-Ansatz, bei dem Komponenten zunächst mit leeren Daten rendern und dann Daten nachladen.

7.3 shadcn

Für UI-Komponenten wurde das shadcn/ui-Konzept gewählt. Anders als klassische Component Libraries werden Komponenten direkt ins Projekt kopiert, was volle Kontrolle über den Code ermöglicht. Die Basis bildet Radix UI, das headless UI-Komponenten ohne vorgefertigte Styles bereitstellt. Radix kümmert sich um Keyboard-Navigation, Focus-Management und ARIA-Attribute.

7.4 Supabase als Backend

Supabase wurde als Backend-as-a-Service-Plattform gewählt und bietet drei zentrale Komponenten: PostgreSQL-Datenbank, Authentifizierung und Realtime-Subscriptions. Die Entscheidung für Supabase gegenüber einer selbst entwickelten Backend-Lösung reduziert die Entwicklungszeit erheblich und ermöglicht den Fokus auf die eigentliche Applikationslogik.

7.5 Datenbank

Die PostgreSQL-Datenbank ist relational strukturiert mit klaren Beziehungen zwischen Entities. Die user-Tabelle speichert erweiterte Benutzerdaten wie emoji, nickname, first_name und last_name. Die event-Tabelle enthält name, state, comment und admin_id sowie Boolean-Flags für date_multiple_choice und location_multiple_choice. Das Feld event_results ist vom Typ JSONB und speichert die finalen Ergebnisse nach Event-Abschluss in flexibler Struktur. Die invites-Tabelle bildet die Many-to-Many-Beziehung zwischen Events und Usern ab. Die answer-Tabelle speichert Benutzer-Antworten mit einem JSONB-Feld, was flexible Antwortstrukturen für Datum-, Orts- und Custom-Question-Präferenzen ermöglicht.

Ein wichtiges Feature ist Row Level Security (RLS). Über PostgreSQL-Policies wird definiert, dass ein User nur Events sehen kann, zu denen er eingeladen wurde oder die er selbst erstellt hat. Diese Security-Logic läuft auf Datenbankebene und ist nicht umgehbar.

7.6 Authentifizierung

Für die Authentifizierung wird Supabase Auth verwendet. Die Entscheidung für echtes Supabase Auth statt eines Mock-Systems war bewusst und basiert auf mehreren Überlegungen.

Erstens bietet ein echtes Auth-System echte Sicherheit mit JWT-Tokens, automatischem Token-Refresh und serverseitiger Validierung. Ein gemocktes System hätte keine dieser Eigenschaften und wäre in einer produktiven Umgebung völlig unbrauchbar.

Zweitens vermittelt die Integration eines echten Auth-Systems praxisnahe Erfahrungen. Die Herausforderungen von Token-Lifecycle-Management, Session-Refresh, Logout-Flows und Error-Handling sind in Mock-Implementierungen nicht vorhanden. Diese Erfahrung ist jedoch wertvoll für die professionelle Entwicklung und spiegelt reale Anforderungen wider.

Im Fun-Organizer wird ein SupabaseAuthProvider implementiert, der React Context nutzt. Dieser Provider initialisiert die Session beim App-Start über getSession, subscribed auf Auth-State-Changes mit onAuthStateChange und stellt Login- und Logout-Funktionen bereit. Eine Besonderheit ist die Kombination von Supabase-Auth-Daten mit der eigenen User-Tabelle. Supabase Auth liefert nur email, id und role. Die erweiterten Daten kommen aus der user-Tabelle. Daher wird ein CompleteUser-Type definiert, der beide Objekte merged. Die setCompleteUser-Funktion lädt bei jedem Auth-State-Change die Daten aus der user-Tabelle mit getUserByAuthId, parsed sie mit Zod und merged sie mit den Supabase-Auth-Daten.

7.7 Supabase Realtime DB

Supabase Realtime ermöglicht das Subscriben auf Datenbank-Changes über WebSocket. Im Fun-Organizer wird dies genutzt, um Event-Details in Echtzeit zu aktualisieren, wenn andere Teilnehmer ihre Antworten abgeben.

Die Implementierung erfolgt über einen SubscriptionProvider mit React Context. Dieser erstellt einen Supabase-Channel und subscribed auf INSERT- und UPDATE-Events der answer-Tabelle mit einem Filter auf event_id, sodass nur relevante Antworten für das aktuell geöffnete Event empfangen werden.

In der Event-Detail-Seite wird über useSubscriptions auf neue Antworten reagiert. Bei einer neuen oder geänderten Antwort wird geprüft, ob die Antwortliste aktualisiert werden muss. Falls ja, werden die Event-Daten neu geladen mit getEventById. Dies führt zu einer nahtlosen User Experience, bei der Antworten anderer Teilnehmer automatisch erscheinen, ohne manuelle Seiten-Reload.

Die Alternative wäre Polling gewesen, bei dem in regelmäßigen Intervallen nach neuen Daten gefragt wird. Realtime-Subscriptions sind jedoch effizienter, da Updates sofort gepusht werden, und bieten eine deutlich bessere User Experience ohne Verzögerung.

7.8 Zod für Runtime-Validierung

TypeScript bietet Typsicherheit zur Compile-Time, aber zur Laufzeit ist JavaScript typlos. Zod schließt diese Lücke durch Runtime-Validierung. Im Fun-Organizer werden Zod-Schemas für alle Datenbank-Entities definiert: userSchema, eventSchema, eventDetailSchema, inviteSchema, answerSchema und weitere.

Der Vorteil ist zweifach: Erstens bietet Zod Runtime-Safety durch Validierung. Zweitens generiert z.infer automatisch TypeScript-Types aus Schemas, sodass keine separate Type-Definition nötig ist. Dies vermeidet Inkonsistenzen zwischen Types und Validierung.

7.9 State Management

Für State Management wurde bewusst auf Redux oder Zustand verzichtet. Stattdessen kommt React Context für globale States (Auth, Subscriptions) und das TanStack Router Loader Pattern für route-spezifische Daten zum Einsatz.

React Context wird minimal eingesetzt: SupabaseAuthProvider für Auth-Status und SubscriptionProvider für Realtime-Updates. Das Loader Pattern lädt Daten parallel, bevor Komponenten rendern. Im Dashboard-Loader werden invitedEvents und answers gleichzeitig geladen. Dies ist performanter als useEffect, das Waterfall-Requests verursacht.

7.10 Mobile-First Design

Die Anwendung wurde von Anfang an für Mobile-Geräte konzipiert. Die Footer-Navigation ist für Touch-Bedienung optimiert mit großen Icons und ausreichend Abstand. Touch-Targets haben eine Mindestgröße von 44x44 Pixeln entsprechend den iOS Guidelines. Layouts sind primär vertikal, was auf schmalen Bildschirmen besser funktioniert. Responsive Design erfolgt durch Tailwind-Breakpoints, wobei Basis-Styles für Mobile gelten und größere Bildschirme über sm:, md:, lg:-Prefixes angepasst werden.

8 Herausforderungen

Erläuterung der grössten Herausforderungen des Projektes und deren Lösung.

8.1 Tom

8.1.1 Login mit Supabase und Tanstack-Router

Es hat eine Weile gedauert, bis ich die Features von Tanstack Router zusammen mit dem React-Contextprovider begriffen und richtig umgesetzt hatte.

Siehe app.tsx, auth.tsx, _authenticated.tsx

8.1.2 Navigieren nach dem login

Zuerst habe ich einfach auth.login(…) aufgerufen und danach direkt zum dashboard navigiert ,was natürlich zu fehlerhaftem Verhalten geführt hat, weil das login ja Zeit braucht und die Supabase-Session und User noch gar nicht da waren.
Die Lösung war, dass ich mit einem pendingUser warte, bis ich vom useSupabaseAuth-Hook die Meldung erhalte, dass der User jetzt eingeloggt ist, erst dann navigiere ich zum dashboard.

Siehe login.tsx und profile.tsx

Beim logout habe ich dasselbe Pattern implementiert, dort funktioniert es leider nicht ganz zuverlässig und ich habe nicht herausgefunden warum.

8.1.3 Abstimmen

Weil pro User ein JSON mit all seinen Anworten zu einem bestimmten Event in der answer-Table geupdatet werden muss, führt das insbesondere bei den questions (mehrere Questions pro Event möglich) zu einem etwas komplizierten mapping…

event-details.tsx und detail-question.tsx

8.2 Helen:

8.2.1 Event erstellen

Die flexible Struktur der Abstimmung erfordert eine Aufteilung auf mehrere Tabellen. Da jedoch sämtliche Abstimmungspunkte wie Datum, Ort und Fragen jeweils einem spezifischen Event zugeordnet sind, ist die Event-ID erforderlich, um diese Optionen korrekt zu speichern.

Da der Erstellungsprozess über einen mehrstufigen Wizard realisiert ist, verteilt sich die Logik über mehrere Komponenten hinweg. Das zentrale Element dabei ist ein Event-Objekt, das als State in der Datei create-event.tsx gehalten wird. Dieses Objekt bildet die gesamte Struktur des Events ab und wird während des Workflows schrittweise aktualisiert.

In der Praxis stellte sich das Aktualisieren dieses Objekts zunächst als herausfordernd dar: Es waren mehrere Testläufe nötig, um sicherzustellen, dass jeweils die richtigen Daten aktualisiert wurden, ohne dabei versehentlich bestehende Informationen zu überschreiben oder zu löschen.

Ein besonderer Sonderfall ergab sich bei der Verwaltung der Fragen und deren Antwortoptionen. Da die möglichen Optionen von der jeweiligen Frage abhängen, reichte es hier

nicht aus, einfache Strings im State zu speichern. Also war der nächste Schritt ein Objekt für die Optionen und einen Array von Frageobjekten in einem useState einzurichten und wieder richtig zu aktualisieren.

Schliesslich wurde der Code noch einmal durch eine Optimierung von Tom verbessert: Er nahm eine saubere Anpassung des Datepickers vor und entfernte dabei überflüssige Zwischenschritte in der Logik, was die Implementierung insgesamt deutlich schlanker und wartungsfreundlicher macht.

Bsp. Voher:

```
function addDate() {
    console.log('add date', date);
    let dateObject = {id: randomId(), date: date, time: time};
    updateDates(dateObject);
    changeDate('');
    changeTime('');
}

// create-event.tsx

function updateDates(newDate: EventDates[number]) {
    //console.log('add date to event data', newDate);
    setEvent((prevEvent) => ({
        ...prevEvent,
        dates: [...prevEvent.dates, newDate],
    }));
}

function removeDate(id: number) {
    //console.log('remove date from event data', id);
    setEvent((prevEvent) => ({
        ...prevEvent,
        dates: [...prevEvent.dates.filter((d) => d.id !== id)],
    }));
}
```

Vs. Jetzt:

```
function onAdd(date: Date, time: string) {
    const formattedDate = formatDate(date);
    const newDate = { date: formattedDate, time: time };
    // avoid duplicates
    if (dates.find((d) => d.date === formattedDate && d.time === time)) {
        return;
    }
    updateEvent({ dates: [...dates, newDate] });
}

// delete function direkt auf dem Button
```

8.2.2 Supabase Realtime

Wir haben das Realtime von Supabase genutzt, um über neue Einladungen und Votes zu informieren. Die erste Integration bzw. der erste hat gerade aktualisiert also dachte ich, das ist nicht so kompliziert. Dann kamen keine Updates mehr rein, und nach diversen Anpassungen und Debugging hat es immer wieder mal funktioniert und dann beim nächsten Test nicht mehr. Ein Blick auf das Websocket das von der Realtime Einbindung erstellt wird, verriet, dass Deletes auf dem Kanal reinkamen, die anderen Änderungen aber nicht. Offenbar braucht es für das Delete nicht die gleiche Authentifizierung wie für die anderen Optionen. Aber ab und zu kamen die Daten, also konnte es auch nicht nur an den Policies von Supabase liegen.
Wir haben schon fast aufgegeben und es als Learning abgetan, als Tom in den Tiefen von Google die Lösung mit der Authentifizierung fand.

```
if (auth.user) {
    async function setUpSubscriptions() {
      const answerChannelName = 'answer:' + ':' + auth.user!.id.toString();
      const token = (await supabase.auth.getSession()).data.session?.access_token;
      supabase.realtime.setAuth(token);

      answerChannel = supabase
        .channel(answerChannelName)
        .on('postgres_changes', { event: '*', schema: 'public', table: 'answer' }, (payload) => {
          if (payload.eventType === 'UPDATE') {
            setAnswerChange(payload.new as SingleAnswer);
          }
        })
        .subscribe();
      }
    setUpSubscriptions();
  }

  return () => {
    console.log('unsubscribing...');
    if (answerChannel) {
      supabase.removeChannel(answerChannel);
    }
  };
}, [auth.user?.id]);
```

8.3 Damir

8.3.1 Visual Clarity im Code

Eine der Herausforderungen im Projekt war die visuelle Klarheit im Code, insbesondere beim Einsatz von Icons. Anstelle selbst erstellter Symbole haben wir die Library lucide-react verwendet. Dadurch konnten wir sowohl Zeit sparen als auch von den einheitlichen und ansprechenden Icons profitieren.

In den meisten Fällen war die Bedeutung der Icons, beispielsweise <CheckCircle />, sofort ersichtlich und führte nicht zu Missverständnissen. Eine Ausnahme stellte jedoch das Icon <Undo /> dar, welches im Create-Event-Prozess eingebaut wurde, um den Vorgang abzubrechen. Während das Icon in der Benutzeroberfläche die gewünschte Funktion klar signalisierte, führte die Bezeichnung Undo im Code zu Verwirrung. Einige Teammitglieder interpretierten dies fälschlicherweise als Funktion zum Rückgängigmachen von Eingaben, anstatt als Abbrechen-Button.

Lösung: Um solche Missverständnisse zu vermeiden, haben wir intern vereinbart, Icons und deren Verwendungszweck gemeinsam zu überprüfen. Das Icon wurde schlussendlich gelöscht.

8.3.2 Styling der Shadcn Komponenten

Eine weitere Herausforderung betraf das Styling der **Shadcn-Komponenten**. Die Library **shadcn/ui** ermöglichte uns, die Seiten effizient mit vorgefertigten Bausteinen wie Input-Feldern oder Checkboxen aufzubauen. Um jedoch unseren individuellen Designanforderungen gerecht zu werden, mussten wir einige Styles anpassen. Dies haben wir über eine eigene CSS-Datei gelöst.

Das Problem dabei: Der Name einer Komponente entsprach nicht immer den tatsächlich verwendeten HTML-Bausteinen. Besonders auffällig wurde dies bei den Checkboxen. Obwohl wir nur Input- und Button-Elemente im CSS angepasst hatten, übernahmen plötzlich auch Checkboxen diese Styles.

Nach einer genaueren Analyse stellten wir fest, dass die Checkbox-Komponente intern als **Button** aufgebaut war. Dadurch war nachvollziehbar, warum unsere Button-Styles auch auf die Checkbox angewendet wurden.

Lösung: Wir haben unsere CSS-Regeln präziser definiert und nur gezielt auf die Elemente angewendet, die tatsächlich angepasst werden sollten.

9 Lessons Learned

Projektablauf. Erfahrungen. Was hat gut funktioniert? Was könnte man ein nächstes mal besser machen? Ausblick

Helen

- Persönlich: Ich habe auf jeden Fall viel gelernt, die Theorie in die Praxis umzusetzen, hat doch mehr Zeit gebraucht als erwartet. Ich habe ein paar Sachen noch mal umgestellt, weil ich gemerkt habe, dass es andersrum besser funktioniert. Das ist eine Erfahrung, die ich sicher mitnehme.
- Als Gruppe: Da wir verschiedene Bereiche mit eigener Logik hatten, konnte sich jeder einem Problem annehmen, ohne dass allzu viele Abhängigkeiten zu den anderen Bereichen ein Hindernis gewesen wären. Trotzdem konnten wir uns unterstützen und auch mal auf eine schon bestehende Lösung zugreifen. Ich empfand die Zusammenarbeit als sehr gut 😊
- Learnings:
    - Supabase Realtime hat viel Zeit und Nerven gekostet, da hätte ich früher nach der Ursache Googeln können, als an meinem Code zu zweifeln und da alles umzustellen.
    - Tailwind hat sicher seine Berechtigung, aber bei meiner Gewohnheit erst Struktur und dann Styling zu machen, bleibt direkt css zu schreiben meine Präferenz.
- Ausblick:
    - Es gibt sicher noch vieles, dass sich eleganter lösen liesse, die vielen Abfragen zu Supabase liessen sich vermutlich noch besser memorisieren.
    - Die User sind jetzt zwar in Supabase vorhanden, aber es besteht noch keine richtige Login- und Registrierungsmaske
    - Wir sind leider nicht dazu gekommen, die Anbindung an Mapbox zu machen und Orte auf der Karte darzustellen.

Tom

- Eine App komplett von Anfang an zu bauen war eine super Erfahrung und ich habe viel über React und Tanstack-Router gelernt und bin von beiden sehr überzeugt. Ich würde meine nächste App wieder damit bauen.
- Supabase ist ebenfalls eine Top-Sache. Dass man gratis so ein Backend verwenden kann finde ich sensationell!
- Learnings:
    - Beim ERM würde ich die Supabase-User-AuthID direkt in die jeweiligen Tables (z.B. admin_id von events) verlinken. Wir haben einen Zwischenschritt über unsere User-Table gemacht, was die RLS (Row-Level-Securty) von Supabase etwas umständlicher gemacht hat.

Damir

- Da dies mein erstes Gruppenprojekt war, konnte ich viele neue Erfahrungen sammeln. Besonders spannend war es, erstmals einen vollständigen Git-Workflow mitzuerleben, inklusive der Arbeit mit Issues, Pull Requests und dem Lösen von Merge Conflicts. Auch die Teamkoordination, also Absprachen darüber, wer an welchem Teil arbeitet, war für mich neu und lehrreich.
- Darüber hinaus konnte ich meine ersten praktischen Kenntnisse in React anwenden und vertiefen. Auch im Bereich Design habe ich viel dazugelernt, insbesondere wie man ein in Figma erstelltes Design Schritt für Schritt in eine funktionierende Weboberfläche umsetzt.

- Learnings
    - Klare Absprachen zu Issues:
      Wenn ein Thema noch nicht als Issue erfasst ist, sollte es zuerst angesprochen oder direkt im GitHub-Repository angelegt werden. So kann vermieden werden, dass mehrere Personen parallel an demselben Problem arbeiten, wie es einmal im Projekt passiert ist.
    - Code-Review und einheitliche Struktur:
      Bevor neue Features umgesetzt werden, ist es hilfreich, sich den Code der Teammitglieder anzuschauen. So kann man ein gemeinsames Verständnis entwickeln und eine einheitliche Code-Struktur sicherstellen. Das verhindert, dass ähnliche Probleme auf drei verschiedene Arten gelöst werden (z. B. mit oder ohne Refresh, stattdessen besser durch einheitliche Nutzung von React Hooks).
- Ausblick
    - Die grundlegende App steht nun und ist funktionsfähig. Für die Zukunft könnten noch einige „Good-to-have“-Features umgesetzt werden, die wir in der Planungsphase diskutiert hatten. Dazu zählt insbesondere die Integration einer Map, die wir ursprünglich einbauen wollten, aber aus Zeitgründen verschieben mussten.