Integrationsprojekt CAS HTML5

Studierende: Schenk Jan, Rindisbacher Marcel, Hofer Severin, Aeschlimann David

MarbleCollector – ein cleveres KMS (Kinder Management System)

1 Thema

Gamification der Kindererziehung mithilfe eines Belohnungssystems für Kinder.

2 Umfeld, Ausgangslage

Kinder dazu zu bewegen, bestimmte Aufgaben oder Ämtli (z. B. Zähneputzen, Umziehen, pünktlich ins Bett gehen, Rasen mähen, Müll raustragen, Zimmer aufräumen, etc.) zu erledigen stellt häufig eine Herausforderung dar. Meist kann der Motivation der Kinder mit kleinen Belohnungen nachgeholfen werden. Eine solche Belohnung könnten auch Murmeln sein, welche die Kinder dann gegen bestimmte Belohnungen wie Süssigkeiten oder Medienkonsum (TV, Computer) eintauschen können.

Versuche mit echten Murmeln scheitern häufig aufgrund von inkonsequenter Umsetzung durch die Eltern und unklarer Verhältnisse über den Stand der Belohnungen.

Die im Projekt erstellte Anwendung soll die Eltern dabei unterstützen, dieses Belohnungssystem komfortabel und konsequent digital umzusetzen. Den Kindern sollen jederzeit einfach ihr aktueller "Belohnungslevel" angezeigt werden können, um sie so zur Erledigung ihrer Aufgaben und Ämtli zu motivieren.

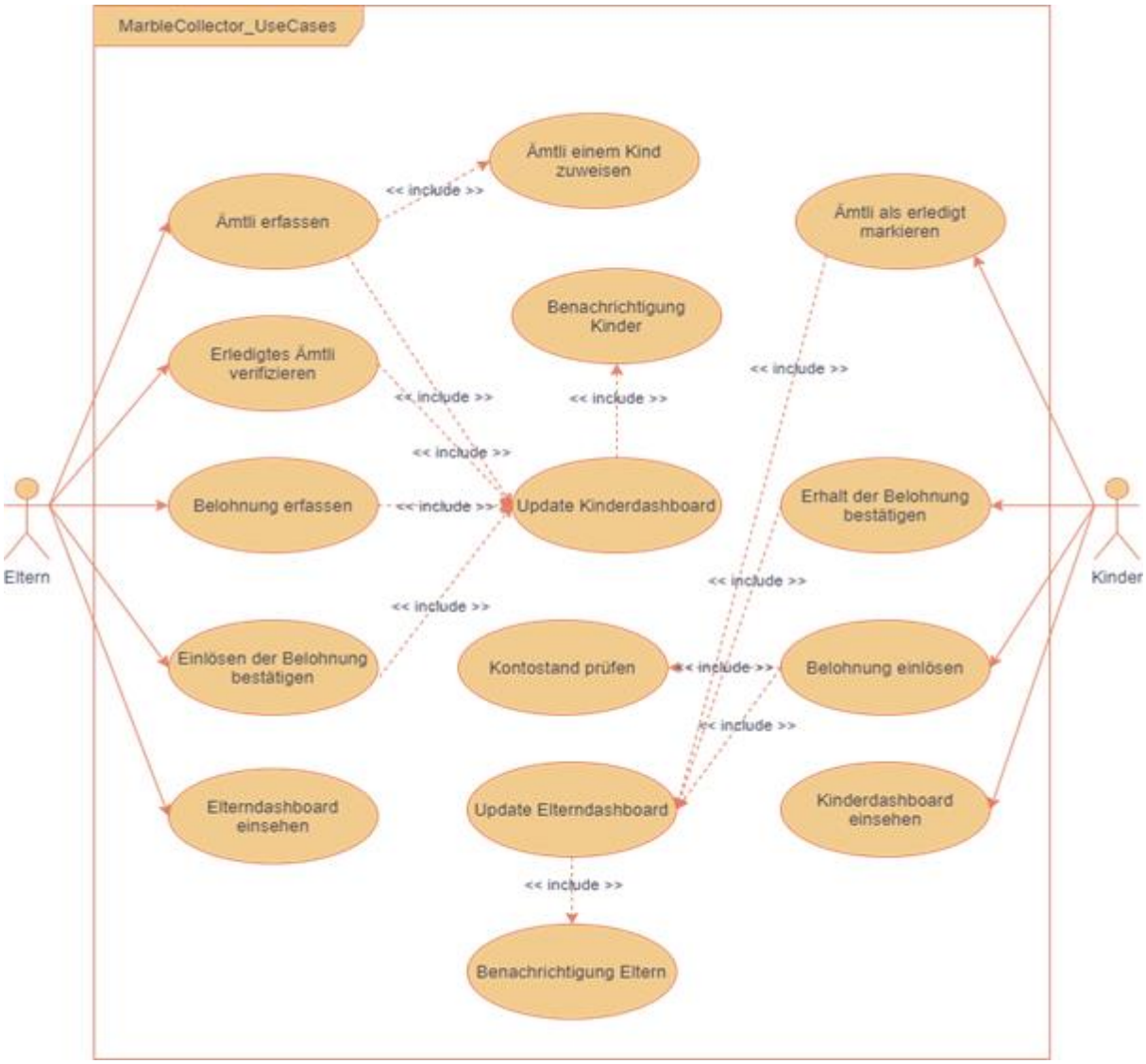
3 Aufgabenstellung & Ziel des Projektes

Es wird eine auf *React* basierende Webanwendung in *TypeScript* erstellt, die sowohl auf mobilen Geräten wie Smartphones und Tablets als auch auf Laptop- bzw. Desktopcomputern ansprechend aussieht und komfortabel bedienbar ist (per Finger und Maus).

Die Webapplikation ist interaktiv zu gestalten. Dies bedeutet, dass Änderungen an erfassten Daten automatisch auf den relevanten Dashboards aufpoppen sollen und Benachrichtigungen angezeigt werden.

3.1 Übersicht Use Cases und Akteure

Das folgende Use Case Diagramm verdeutlicht die in der Applikation involvierten Akteure und zeigt die wichtigsten Anwendungsfälle und deren Beziehung untereinander auf.



Die Applikation beinhaltet zwei Benutzerrollen, deren Aufgaben und Aktivitäten sich unterscheiden. Für die Erfassung der Hausarbeiten und Belohnungen sind die Eltern zuständig und die Kinder treten als Konsumenten und Verarbeiter dieser Informationen auf.

4 Scope des Projektes

Nachfolgend soll der Funktionsumfang des realisierten Produktes aufgezeigt werden. Hierbei ist zu erwähnen, dass sich während der Realisierungsphase die Priorisierung gegenüber der Projektspezifikation teilweise verschoben hat, und teilweise auch andere Anforderungen dazu kamen. Dies einerseits durch das Testen der Applikation von unserer Seite, andererseits waren wir in der glücklichen Lage, durch Jan und seine Familie, welche sich fleissig als Tester zur Verfügung stellten, direktes Benutzerfeedback zu bekommen und einfliessen zu lassen.

4.1 Umgesetzte Features

4.1.1 Ämtli

Eltern:

- Ämtli Dashboard mit übersichtlicher Darstellung aller Ämtli
- Erfassen eines Ämtlis in Pop-up Formular inklusive Validierung
- Sortierung der Ämtlis anhand der Relevanz (wo sind Benutzer Aktionen nötig)
- Ämtli Komponente:
    - Editieren eines erstellten Ämtlis (z. B. Murmeln, Beschreibung, Titel) durch direkten Klick/Touch des zu verändernden Wertes
    - Kopieren eines bestehenden Ämtlis
    - Löschen eines noch nicht durch Kinder gestarteten Ämtlis
    - Ämtli einem Kind zuordnen
    - Darstellung des Zustandes eines Ämtlis für die verschiedenen Kinder inklusive Bedienelemente für Benutzeraktionen der Eltern (Erledigung eines Ämtlis bestätigen oder verweigern)
    - Notification Batch, wenn Benutzereingabe bei einem Ämtli erforderlich

Kinder:

- Ämtli Dashboard mit übersichtlicher Darstellung aller Ämtli
- Ämtli Komponente:
    - Übersichtliche Darstellung des Zustandes eines zugeordneten Ämtlis
    - Stepper Element für eine einfache Benutzerinteraktion
    - Button für Interaktionen welcher sich dem Zustand des Ämtli anpasst
    - Notification Batch, wenn Benutzereingabe bei einem Ämtli erforderlich
    - Spezialeffekt (Konfetti), wenn die Murmeln für das erledigte Ämtli kassiert werden können, sorgen für eine zusätzliche Motivation der Kinder

4.1.2 Belohnung

Eltern:

- Belohnungs - Dashboard mit übersichtlicher Darstellung aller Belohnungen
- Erfassen einer Belohnung in Pop-up Formular inklusive Validierung
- Sortierung der Belohnungen anhand der Relevanz (wo sind Benutzer Aktionen nötig)
- Belohnungs Komponente:
    - Editieren einer erstellten Belohnung (z. B. Murmeln, Beschreibung, Titel)
    - Kopieren einer bestehenden Belohnung
    - Löschen einer noch nicht durch Kinder genutzte Belohnung
    - Belohnung einem Kind zuordnen
    - Darstellung des Zustandes einer Belohnung für die verschiedenen Kinder inklusive Bedienelemente für Benutzeraktionen der Eltern (Einfordern einer Belohnung bestätigen oder verweigern)
    - Notification Batch, wenn Benutzereingabe bei einer Belohnung erforderlich

Kinder:

- Belohnungs - Dashboard mit übersichtlicher Darstellung aller Belohnungen
- Belohnungs - Komponente:
    - Übersichtliche Darstellung des Zustandes einer zugeordneten Belohnung
    - Stepper Element für eine einfache Benutzerinteraktion
    - Notification Batch, wenn Benutzereingabe bei einer Belohnung erforderlich

4.1.3 Benachrichtigungen

Eltern:

- Benachrichtigung in der Navigationsleiste, wenn in einem der Dashboards eine Aktualisierung stattgefunden hat (mittels Notification Batch auf dem Icon des entsprechenden Dashboards)
- Ämtli:
    - Benachrichtigung, wenn ein Kind ein Ämtli fertiggestellt hat und die Eltern dies prüfen müssen
- Belohnung:
    - Benachrichtigung, wenn ein Kind eine Belohnung einfordern möchte

Kinder:

- Benachrichtigung in der Navigationsleiste, wenn in einem der Dashboards eine Aktualisierung stattgefunden hat (mittels Notification Batch auf dem Icon des entsprechenden Dashboards)
- Ämtli:
  - Benachrichtigung, wenn dem Kind ein neues Ämtli zugeordnet wurde
  - Benachrichtigung, ein Ämtli durch die Eltern geprüft wurde
- Belohnung:
  - Benachrichtigung, wenn dem Kind eine neue Belohnung zugeordnet wurde
  - Benachrichtigung, wenn dem Kind das Einlösen einer Belohnung verweigert oder gestattet wurde

Allgemein:

- Benachrichtigung des Benutzers mit zusätzlicher Browser Notifikation (opt-in)

4.1.4 Look & Feel

- Eigener Avatar für jeden User
- Responsives Design mit Fokus auf Mobile Devices
- Dark Theme zur Schonung des Akkus und der Augen
- Browser Notifikationen können auf dem Profil aktiviert werden
- Swipe Navigation in den Hauptscreens

4.1.5 Motivation & Gamification

- Spezialeffekt beim Einlösen der Murmeln
- Rangliste für zusätzliche Motivation der Kinder (umgesetzt anhand der verdienten Murmeln)
- User Statistik im Profil mit Übersicht der verdienten Murmeln und erledigten Ämtli
- Möglichkeit das Profil/Score eines anderen Benutzers anzuzeigen (Benutzerliste)

4.2 Mock, bewusst weglassen

- Kein User-Management, d. h. Eltern und Kinder sind hard-codiert bzw. vordefiniert und können nicht über die Applikation angepasst werden, 2 Elternteile, 3 Kinder.
- Logins sind ebenfalls hard-codiert bzw. vordefiniert, d. h. fixer Benutzername und Passwort.
- Die Lösung ist nicht mandantenfähig, d. h. es können über die Applikation keine weiteren Familien und Familienmitglieder hinzugefügt werden.

4.3 Nicht umgesetzt

Diverse Punkte, welche in der Projektspezifikation als "Nice to Have" definiert wurden, wurden nicht umgesetzt. Während der Realisierung stellte sich heraus, dass anderen Punkten eine höhere Priorität eingeräumt werden musste oder letztendlich die Zeit für die Realisierung fehlte. Im Kapitel 10 wird auf die nächsten Schritte für eine weitere Entwicklung eingegangen. Der Vollständigkeit halber sind hier trotzdem alle Punkte aus der Projektspezifikation aufgeführt, welche nicht realisiert wurden:

- Zeitlich terminierte Ämtli mit Benachrichtigung bei Ablauf des Termins.
  ➔ Die Erfassung eines Endtermines für Ämtli wurde vorbereitet
- Offline-Fähigkeit auch inklusive Nutzung der Notification API (Service Worker)
- Möglichkeit für Eltern, den Kindern Lob für ausgeführte Ämtli auszusprechen.
- Timer für zeitbasierte Belohnungen mit einer Benachrichtigung bei Ablauf der Zeitdauer (z. B. 30 min. Handybenutzung).
- Grafische Visualisierung des Verlaufs des "Kontostandes" der Kinder.
- Periodisch wiederkehrende Ämtli.
- Unterscheidung in Pflicht- und Bonusämtli.

5 Systemübersicht, Grob-Architektur, Deployment

5.1 Systemübersicht

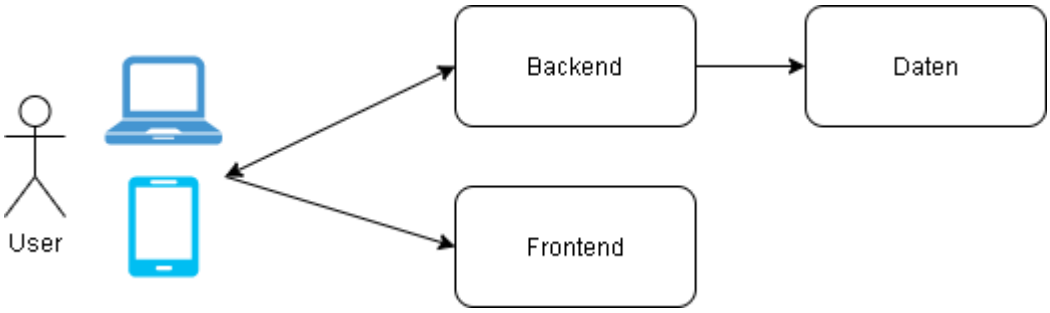
5.1.1 Systemgrenze

Die Webapplikation *MarbleCollector* ist ein in sich abgeschlossenes System, für welches keine Interaktionen mit Umsystemen oder anderen Applikationen vorgesehen sind.

Die optionale Anforderung für die Benachrichtigung eines Benutzers auch bei inaktiver Web-App, wurde aufgrund der hohen Komplexität und des für die Projektarbeit beschränkten Nutzens nicht umgesetzt. Dadurch gibt es auch keinen *Push Messaging Service,* der zum System dazugehört oder mit ihm interagiert.

5.1.2 Systemkomponenten

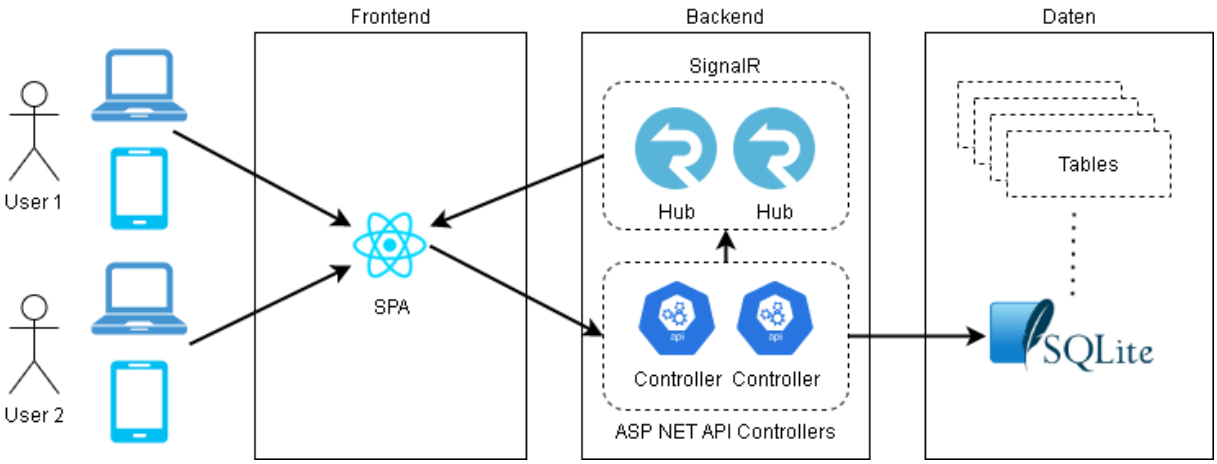
Die Webapplikation *MarbleCollector* besteht aus den folgenden technischen Komponenten.



Das **Frontend** wird an beliebige Webclients ausgeliefert und soll als *Single Page Application (SPA)* aufgebaut werden. Die *Single Page Application* wird im Browser ausgeführt und ruft Funktionen des **Backends** auf, um mit den persistierten **Daten** interagieren zu können. Das Backend seinerseits soll in der Lage sein, Nachrichten oder Daten ans Frontend zu senden und dadurch die Benutzer zu benachrichtigen oder weiterführende Aktionen auszulösen.

5.1.3 Systemarchitektur

Die zuvor vorgestellten Systemkomponenten werden in diesem Abschnitt noch genauer aufgeschlüsselt und einzelne Bestandteile im Detail erklärt.

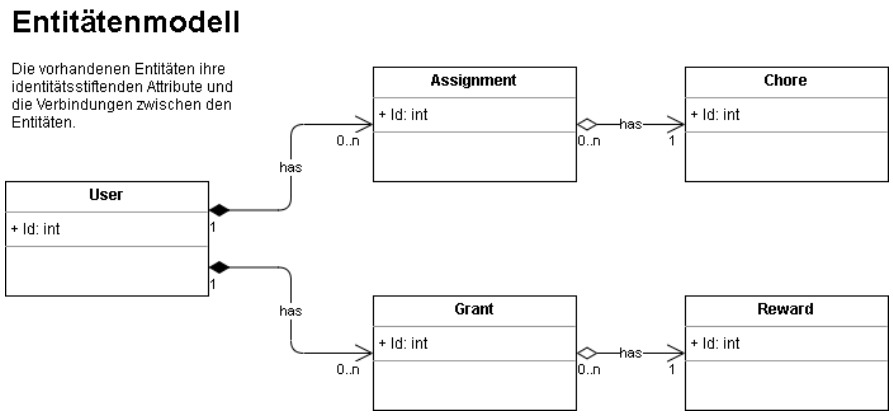


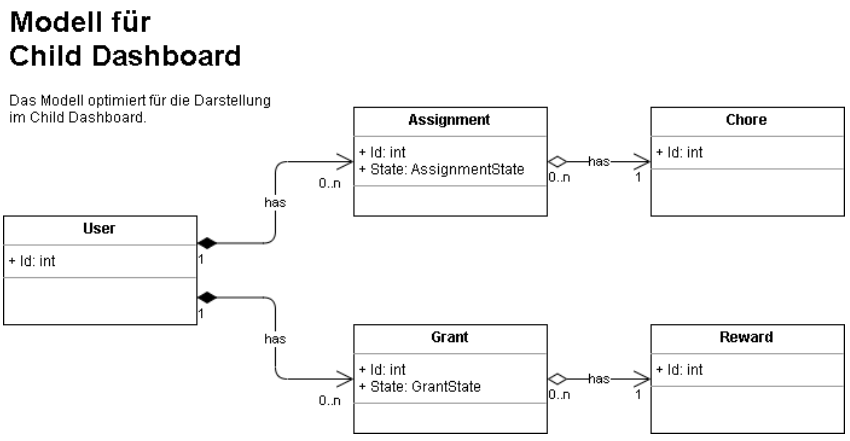
Die *Single Page Application* wird an verschiedene Benutzer und Clients ausgeliefert und verhält sich entsprechend des eingeloggten Benutzers und dessen Rolle. Die Daten werden initial über die *REST API* geladen, dies kann beispielsweise bei der Navigation auf eine neue Ansicht geschehen. Bei Änderungen der Daten geschieht dies ebenfalls über die *REST API*. Der jeweilige *API Controller* entscheidet, ob die Daten in der *SQLite* Datenbank mutiert werden müssen oder nicht. Die Datenbank wird beim Start des Backends erstellt, sofern sie nicht bereits existiert und das Schema auf den letzten Stand gebracht.

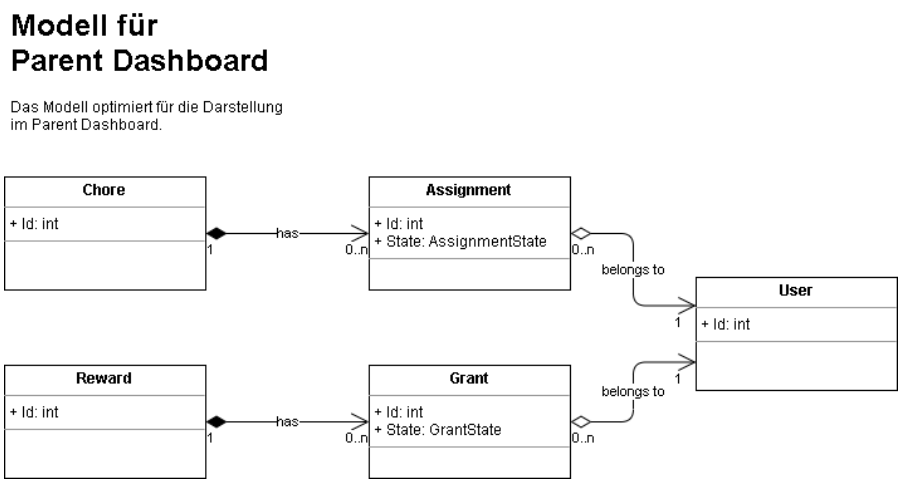
Ebenso entscheidet der *API Controller,* ob die vorliegende Mutation eine *SignalR* Notification auslösen soll. Diese SignalR Notification wird über eine Websocket Connection übermittelt, welche beim Start der SPA initialisiert wird. Falls eine Notification nötig ist, wird die Benachrichtigung an alle Clients des jeweiligen *Hubs* versandt, welche darauf reagieren können, in dem sie bspw. eine Benachrichtigung anzeigen und Daten nachladen. Die Kommunikation in der Gegenrichtung via *Websockets* von der *SPA* zum *SignalR Hub* wird zurzeit nicht genutzt, weshalb der Pfeil nur unidirektional ist.

5.1.4 Datenstruktur

In der Entwurfsphase wurden die folgenden Diagramme erstellt, um die Relationen zwischen den Daten aufzuzeigen. Bei der Analyse der Anforderungen wurde festgestellt, dass sich die Daten mit einer relativ einfachen Struktur abbilden lassen. Diese Strukturierung wurde als Grundlage für die Implementation des Datenmodells genutzt.

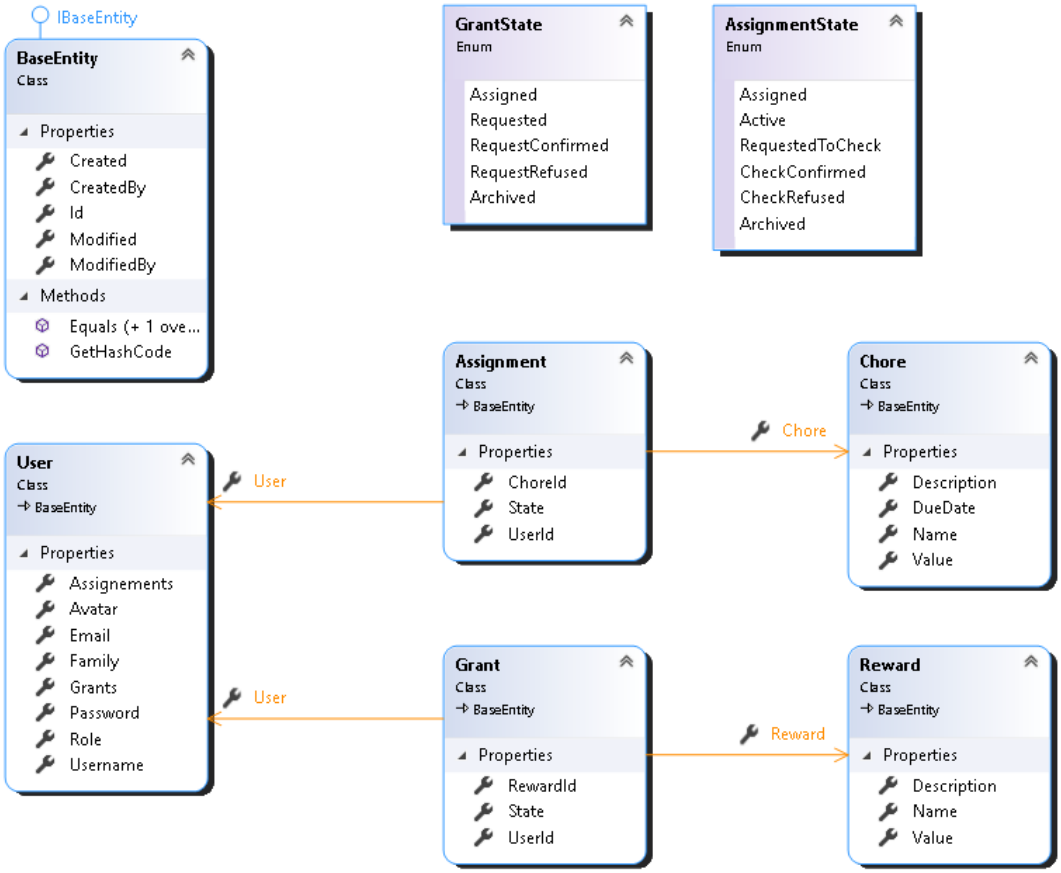






Wie in den obigen Abbildungen ersichtlich, war der Grundgedanke des Datenmodells, die **User**, in diesem Fall die Kinder, in eine Beziehung mit ihren Ämtli (**Chore**) und Belohnungen (**Rewards**) zu setzen. Dies konnte relativ einfach dadurch realisiert werden, dass eine Verbindungstabelle mit den Zuweisungen für die Ämtli (**Assignment**), und eine Verbindungstabelle mit den Zuweisungen für die Belohnungen (**Grants**) erstellt wurde. In diesen Verbindungstabellen wurde auch der Zustand der jeweiligen Zuweisungen gehalten (**GrantState** / **AssignmentState**).

Nach Fertigstellung des Projektes wurde das folgende Entitätendiagramm direkt aus dem *Visual Studio* exportiert, in Projekt als zusätzliche Datei abgelegt und hier der Vollständigkeit halber ein Bild davon abgelegt. Wie sich gezeigt hat, konnte das ursprünglich erstellte Datenmodell genutzt werden und musste nicht angepasst werden.



5.2 Projektaufbau

Der gesamte Source Code und auch die Anleitung wie das Projekt initialisiert wurde, befindet sich auf dem öffentlichen *GitHub* Repository *MarbleCollector*.

https://github.com/TashunkoWitko/MarbleCollector

Der Ordner *api* enthält das *.NET Core* Backend Projekt und der Ordner *client* das *React* Frontend Projekt. Hier werden die wichtigsten Gedankengänge hinter der Struktur erklärt.

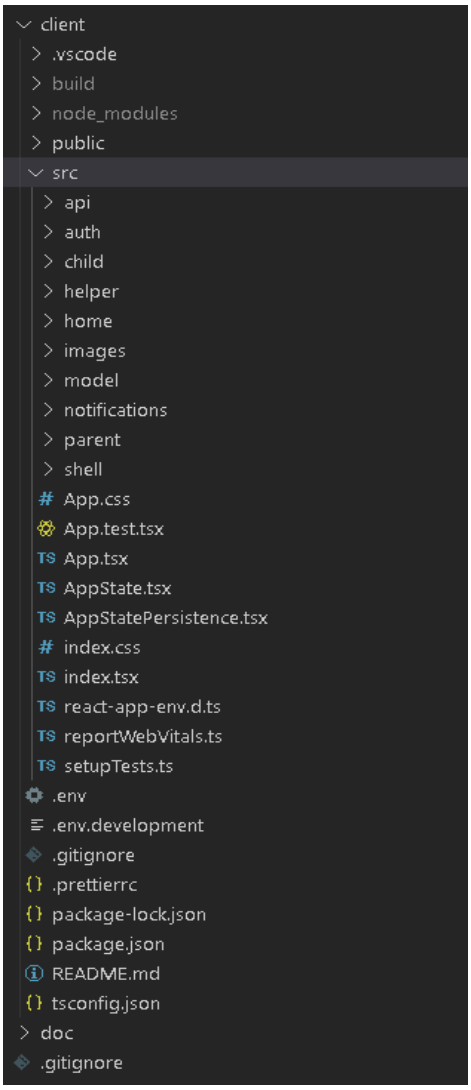
5.2.1 Frontend

Das Projekt wurde mit *create-react-app* mit dem *typescript template* initialisiert.

```
npx create-react-app marblecollector-client --template typescript
```

Da *React* wenig Vorgaben betreffend der Struktur des Frontend Projektes vorgibt, hat das Projektteam die folgende Strukturierungsformen gewählt:

- Strukturierung nach Screen
- Strukturierung nach Funktion/Modul



Toplevel Files

Die Files auf der ersten Ebene sind für global relevante Komponenten, Styles oder States reserviert (App*).

Strukturierung nach Screen

Als primäres Strukturierungskriterium wurde die Zugehörigkeit der Files zu einem der Hauptscreens gewählt (parent, child).

Strukturierung nach Funktion/Modul

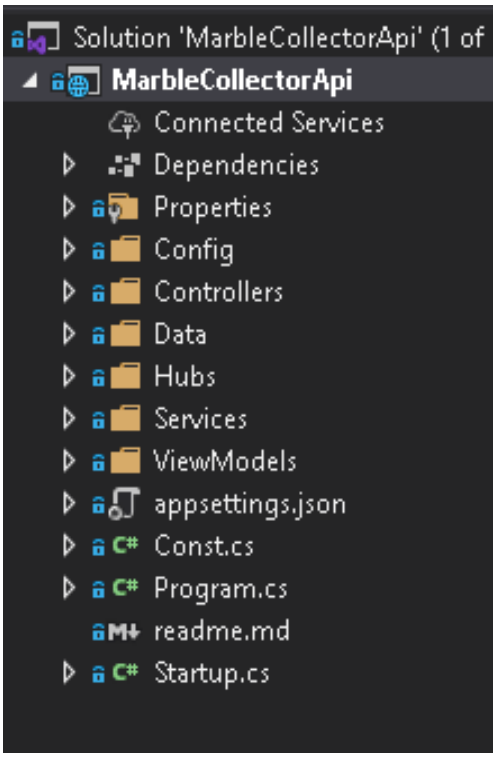
Während der Entwicklung wurde klar, dass viele der erstellten Funktionen/Komponenten für mehrere Screens verwendet werden konnten. Diese Bestandteile wurden als Module auf der obersten Ebene positioniert, damit aus den Hauptscreens einfach darauf zugegriffen werden kann.

Aktuelle Strukturierung

Da ein grosser Teil der Entwicklung sehr agil geschah, wurden die oben beschriebenen Prinzipien nicht komplett durchgängig umgesetzt. Ein Refactoring des aktuellen Standes nach den oben beschriebenen Prinzipien und zur Reduktion von Code Duplication wäre auf jeden Fall sinnvoll. Das Projektteam hat entschieden, dies nicht mehr vor der Abgabe des Projektes zu tun, um unvorhergesehene Nebeneffekte kurz vor der Abgabe zu vermeiden.

5.2.2 Backend

Das Projekt wurde aus dem *Visual Studio* heraus mit der *ASP .NET Web API* Vorlage leer erstellt, das bedeutet möglichst wenig initialen Boilerplate Code mit dazu genommen.



Ziel war es das Backend so schlank wie möglich zu halten. Hier die wichtigsten Eckpunkte:

Controllers

Enthält die API Controllers und alle Methoden, welche durch das Frontend konsumiert werden können.

Data

Enthält die Klassen zur Erstellung und zur Interaktion mit der *SQLite* Datenbank, mehr dazu im Teil Datenhaltung.

Hubs

Enthält die *SignalR* Hubs, sowie alles was mit Notifications zu tun hat.

Services

Enthält spezialisierte Services, welche komplexere Funktionen des Backends abstrahieren. Dazu gehört bspw. die Logik zur Berechnung der Benutzer Score.

ViewModels

Die Klassen welche ans Frontend versandt werden (Grundlage für TS interfaces).

Program.cs

Einstiegspunkt der *Dotnet* Applikation, welche die Middleware initialisiert und startet.

Startup.cs

Die Start-up Klasse konfiguriert die Middleware im Detail und definiert was die API macht.

5.2.3 Datenhaltung

Für die Datenhaltung kam das *Entity Framework* von *Dotnet* und daraus die *Code First* Strategie zum Einsatz. Die Gründe dafür waren wie folgt:

- Es gab keine bestehende Datenbank
- Das Datenbankschema ist relativ simpel
- Wir wollen möglichst schnell/effizient eine Datenbank haben

Um möglichst wenig Overhead zu erzeugen, wurde als Provider *SQLite* gewählt. Das Datenbankfile wird auf dem jeweiligen Host des Backends im Filesystem gespeichert.

Models

Enthält die Entitätsklassen des Datenmodells, welche für die Persistierung in der Datenbank verwendet werden. Die Models können über die dazugehörigen EntityTypeConfiguration Klassen konfiguriert werden (z. B. Datenbank Index für Spalte).

Repositories

Für den Zugriff auf die Datenbank wurden dem *Repository Design Pattern* entsprechend Repository-Klassen erstellt, welche den Datenzugriff abstrahieren. *Microsoft Entity Framework* bietet die benötigen Datenbanktreiber für den Zugriff auf die *SQLite* Datenbank.

Migrations

Hier befinden sich die mithilfe des *Microsoft Entity Frameworks* generierten Migrationsanweisungen, um die Datenbank auf eine bestimmte Version zu transformieren.

Dateninitialisierung

Die Datenbank wird für Entwicklungs- und Demonstrationszwecke mit Beispieldaten befüllt (*Seeding*). Hierfür bestehen zwei verschiedene Datensets, die in den *Seeder*-Klassen im *Init* Verzeichnis abgelegt sind. Im Code können auch für die lokale Entwicklung, die für den Democase verwendeten Daten erzwungen werden. Achtung: Im Demoszenario wird die ganze Datenbank zurückgesetzt und neu mit Daten befüllt.

5.2.4 Zusammenspiel

Damit alle Komponenten miteinander zusammenspielen, mussten die folgenden Schritte umgesetzt werden.

CORS

Damit die *SPA* mit dem Backend aus dem Browser heraus kommunizieren darf, mussten im Backend alle nötigen Frontend Urls freigeschalten und erlaubt werden.

Damit die Authentication funktioniert, musste „*AllowCredentials*“ hinzugefügt werden.

Damit *PUT* Requests funktionieren, musste „*AllowAnyMethod*“ hinzugefügt werden.

Backend API Url Konfiguration

Damit das Frontend weiss, unter welcher Url die *API* sowie die *SignalR* Hubs anzusprechen sind, auch insbesondere im Deployment-Szenario, wurden custom environment variables verwendet.

Dazu gibt es im Root des Frontend Projektes je ein *.env** File welches die Url für die lokale Entwicklung oder das Deployment Szenario enthält.

Backend zu Datenbank

Im Backend wird im *appsettings.json* File der *ConnectionString* für die Datenbank hinterlegt. Für *SQLite* ist dies ein Pfad auf die Datei im Filesystem. Auch hier gibt es ein File für die lokale Entwicklung und eines fürs Deployment-Szenario. Der Standardwert für das lokale Entwickeln ist für Windows Geräte ausgelegt, er kann jedoch problemlos angepasst werden.

5.3 Build & Deployment

Das Projekt wurde in zwei Szenarien entwickelt, getestet und genutzt:

1. Auf der lokalen (Windows)-Maschine eines Entwicklers
2. Deployed in *Microsoft Azure PaaS* Services

Damit das Projekt von *GitHub* nach *Azure* gelangt wurde *Azure DevOps* eingesetzt.

5.3.1 Lokales Entwickeln

Die genauen Schritte fürs Ausführen der beiden Projekte sind im *GitHub* Repository dokumentiert. Sämtliche Projektmitglieder verwendeten Windows Clients, weshalb nur dieses Szenario getestet wurde. Grundsätzlich ist jedoch *Dotnet* plattformunabhängig und sollte nach der Installation der entsprechenden *SDK (V5)* überall ausführbar sein.

https://github.com/TashunkoWitko/MarbleCollector#run-it-locally

5.3.2 Azure Deployment

Für das Hosting verwenden wir das Cloud Computing Angebot von *Microsoft Azure*. Über das Studentenangebot (https://azure.microsoft.com/de-de/free/students/) steht ein Grundguthaben zur Verfügung für ein eingeschränktes Set an Services, welches allerdings für die Zwecke dieser Projektarbeit genügen sollte.

Für das *.NET Core* Backend Projekt wird ein App-Service mit entsprechender Runtime verwendet. Auf dem Filesystem dieses App-Services befindet sich die *SQLite* Datenbank. Gestartet hat das Projektteam mit einem Linux-based App-Service, doch aufgrund von *SQLite* und App Service Linux Filesystem Inkompatibilitäten musste auf Windows gewechselt werden (Referenz).

Für das Frontend Projekt wird ein *Storage Account* verwendet, für welchen die statische Website Option aktiviert wird. Natürlich muss programmatisch sichergestellt werden, dass Frontend und Backend miteinander sprechen können (*CORS, Cross Origin Resource Sharing*). Im *GitHub* Repository befinden sich jeweils die aktuellen Urls auf das Frontend und das Backend.

Die folgenden *Azure* Ressourcen wurden über das Azure Portal manuell erstellt:

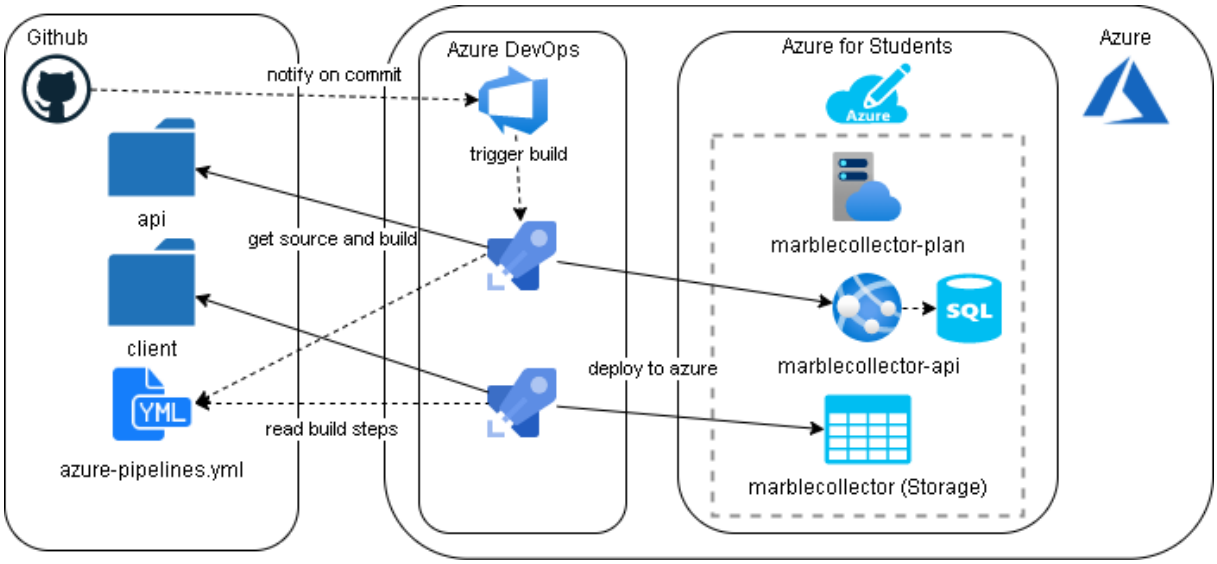
Ressource	Name	Beschreibung
Resource Group	marblecollector	Container für alle Ressourcen
App Service Plan	marblecollector-plan	Definiert Preis/Leistung des App Service
App Service	marblecollector-api	Enthält die API, Runtime & PaaS
Storage Account	marblecollector	Enthält die SPA und statische Files

5.3.3 Azure DevOps

Als Build Service kommt *Azure DevOps* zum Einsatz. *Azure DevOps* wird durch *GitHub* benachrichtigt, wenn ein Pull Request auf den *main* Branch des Repository vorgenommen wird. *Azure DevOps* erstellt für beide Projekte einen Build und deployed die Artefakte in die konfigurierten *Azure Ressourcen* via einer *Azure Service Connection*.

GitHub löst den Build in *Azure DevOps* bei jedem *Pull Request* aus, was eine Unschönheit ist. Eine Verbesserung wäre die Aufteilung in eine Build Pipeline, welche nur den Build macht und eine Release Pipeline, welche beim Abschliessen des Merge geschieht.

Das folgende Diagramm verdeutlicht das Zusammenspiel der verschiedenen Komponenten.



Die im Diagramm abgebildeten Schritte sind wie folgt:

1. Es wird auf *GitHub* ein neuer *Pull Request* erstellt
2. *GitHub* benachrichtigt *Azure DevOps*
3. *Azure DevOps* startet ein neuer Build Prozess basierend auf der Pipeline Definition, welche im Repository im *azure-pipelines.yml* File deklarativ beschrieben ist
4. Die Pipeline beinhaltet die folgenden Schritte
    a. Checkout des Repositories
    b. Backend
        i. Build der *Dotnet* App.
        ii. Package der *Dotnet* App.
        iii. Publish der *Dotnet* App in *Azure App Service.*
        iv. Restart des *App*-Services -> Datenbank wird aktualisiert.
    c. Frontend
        i. Installation der *node* Abhängigkeiten.
        ii. Build der *SPA.*
        iii. Publish der statischen Files in *Azure Storage Account.*

6 Eingesetzte Technologien

6.1 Frontend

Für das Frontend Projekt kam *React* als Single Page Application (*SPA*) Framework zum Einsatz und das Projekt wurde basierend auf dem *react script typescript template* initialisiert.

Die *JavaScript* Library *React* wurde aufgrund ihrer recht leichtgewichtigen Natur als Basis gewählt. *React* als Library konzentriert sich auf die Umsetzung von komponenten-basierten User Interfaces und gibt kaum Vorgaben zu anderen Aspekten einer *JavaScript* Frontendapplikation. Das *React* Ökosystem bietet aber eine Vielzahl von zusätzlichen Libraries, um die fehlenden Aspekte abzudecken. Damit lässt sich ein "Framework" ganz auf die Bedürfnisse der umzusetzenden Applikation zusammenstellen.

Die folgende Tabelle wurde direkt aus dem *package.json* File des Frontend extrahiert und enthält eine Übersicht aller eingesetzten Libraries/Packages, welche im Anschluss im Detail einzeln oder gruppiert nach Thema erklärt werden.

Package	Thema	Kurzbeschreibung
@date-io/date-fns	Date	Date Utility Library.
@material-ui/core	Material UI	*React* Komponentenlibrary basierend auf Material Design.
@material-ui/icons	Material UI	Icons für die Material UI Komponentenlibrary.
@material-ui/lab	Material UI	*Material UI* Komponenten, die sich noch in der Entwicklung befinden.
@material-ui/pickers	Material UI	Spezielle Date und Time Picker Komponenten für *Material UI*.
@microsoft/signalr	SignalR	*SignalR* Client-Library.
@testing-library/*	Test	Nicht aktiv verwendet
@types/*	Type	Typen für *Typescript*
axios	HTTP	Library für HTTP Abstraktion
formik	Form	Formular Library.
formik-material-ui	Form	*Formik* Komponenten speziell für die Verwendung mit *React*.
formik-material-ui-pickers	Form	*Formik* Date Picker Komponenten speziell für die Verwendung mit *React*.
immer	State	Hilfsmittel für den Umgang mit Immutable State

react	React Core	*JavaScript* Library für User Interfaces.
react-confetti	Confetti	Library für das einfache Einbinden eines Konfetti Effektes in die App.
react-dom	React Core	Methoden um *React* User Interfaces in das HTML DOM zu rendern.
react-query	HTTP	Library zum Fetchen, Cachen und Updaten von asynchronen Daten.
react-router-dom	Routing	Komponentenlibrary für die Navigation bzw. das Routing.
react-scripts	React Bootstrap	Scripts und Konfiguration für *React* Applikationen.
recoil	State	Statemanagement Library.
recoil-persist	State	Library zum Persistieren von *Recoil* State im Storage.
typescript	Type	Library für die statische Typisierung von *JavaScript*.
web-vitals	React Core	Library für die Messung von *Web Vitals*. Nicht aktiv verwendet.
yup	Form	Library für die Validierung von Daten aus Formularen.

6.1.1 Material UI

Um die Entwicklung der Applikation zu beschleunigen, wurde entschieden eine der bekannten UI Komponentenbibliotheken einzusetzen. Die Wahl fiel auf *Material-UI*, da diese vermutlich in zukünftigen Projekten der Projektteammitglieder zum Einsatz kommen wird. Ausserdem hatten einige Projektteammitglieder bereits Erfahrungen mit Bootstrap gesammelt und wollten Know-how mit einer anderen UI Komponentenbibliothek sammeln.
Material-UI bietet eine grosse Auswahl an UI Komponenten, eine Vielzahl von vordefinierten Icons und verschiedene Möglichkeiten/APIs das Styling der UI-Komponenten zu beeinflussen. Es bietet allerdings bereits standardmässig ein stimmiges Styling/Theming, das sehr schnell zu einem ansprechenden Resultat führt und nur sehr gezielt angepasst werden muss.

6.1.2 HTTP-Requests

Für die Umsetzung von HTTP-Requests wurde *Axios* gewählt. Das simple API dieser promise-basierten HTTP-Clientlibrary mit seinen an die HTTP-Verben angelehnten Funktionen gab den Ausschlag für die Wahl von *Axios*.

Die Library *React-Query* bietet ein hook-basiertes API zur Vereinfachung des Fetch-, Cache- und Update-Handlings, welches durch geschickte Voreinstellungen die asynchrone Kommunikation mit HTTP-basierten Schnittstellen erleichtert. Sie bietet einfache Möglichkeiten den Lade- und Fehlerzustand der laufenden Abfragen zu handhaben. Dadurch ist es nicht nötig diese Zustände umständlich im globalen Zustand der Applikation manuell nachzuführen. Das führt zu einem aufgeräumten globalen Applikationszustand.
Durch die Invalidierung von Queries kann nachschreibenden Operationen sichergestellt werden, dass der Zustand automatisch wieder vom Backend geladen und somit zwischen Backend und Frontend synchronisiert wird.

6.1.3 SignalR

ASP.NET SignalR ist eine Library für die einfache Integration von real-time Web-Funktionalität in Applikationen. Sie ermöglicht unter anderem die Ausführung von *Remote Procedure Calls (RPC)* durch das Backend in der Frontend-Applikation.

In der Applikation wurde *SignalR* für die Umsetzung der Benachrichtigungsfunktionalität verwendet. Die Frontend-Applikation subskribiert sich in den *SignalR Hubs* im Backend und erhält über diese Hubs Benachrichtigungen über geänderte Daten im Backend.

6.1.4 Statische Typisierung

Die statische Typisierung mithilfe von *Typescript* wurde gewählt, da die Projektteammitglieder mehrheitlich Erfahrung mit statisch typisierten Programmiersprachen im objekt-orientierten Umfeld hatten und damit der Einstieg einfacher fiel. Der etwas grössere Aufwand durch die explizite statische Typisierung zahlte sich in vielen Situationen durch die dadurch vorhandenen Type-Checks und Zusatzinformationen in der Entwicklungsumgebung (z. B. *Intellisense*) wieder aus.

6.1.5 Form

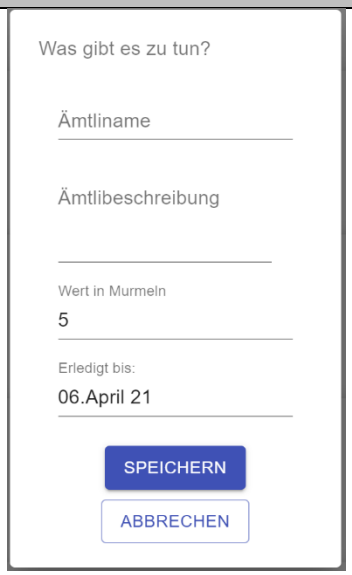
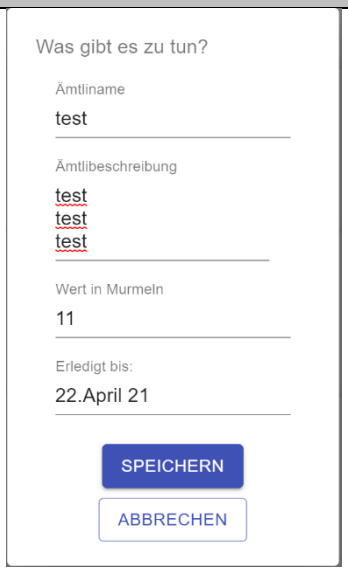
Formik ist prädestiniert mit wenig Code simpel und schnell Formulare umzusetzen. Somit können die Inputs, deren Validierung sowie Ausgabe und Weiterverwendung der Daten einfach gehandhabt werden. In vier grundlegenden Etappen kann demnach ein Eingabeformular für den User erstellt werden:

Grobe Code-Architektur

```
<Formik initialValues,validate,onSubmit > {({ submitForm }) => (
 <MuiPickersUtilsProvider locale={de} utils={DateFnsUtils}>
  <Form> ... </Form>
 </MuiPickersUtilsProvider>
)} </Formik>
```

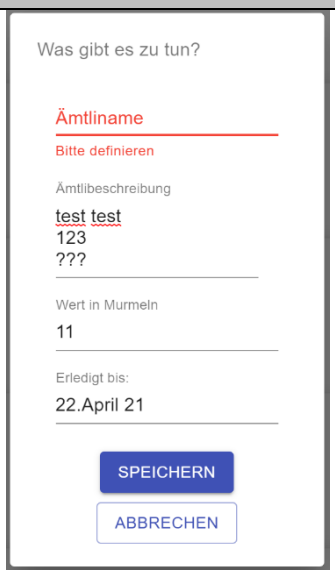
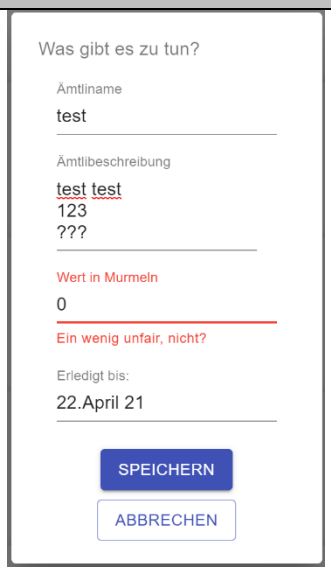
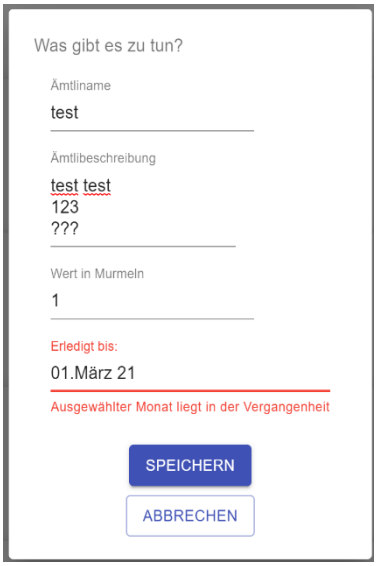
Initialisierung “initialValues”

Die erste Definition beinhaltet die Darstellung der einzelnen Inputs des Formulars zu Beginn aus der User-Perspektive. Beispielsweise können diese leer gelassen, mit dem heutigen Datum oder mit einem Platzhalter abgefüllt werden.

initialisiert	bearbeitet
	

Validierung “validate”

Um ein Formular auf dem Frontend zu validieren wird auch hier direkt ein eigener Definitionsabschnitt zur Verfügung gestellt. In diesem können alle einzelnen Inputs angesprochen und für die gewünschten Inhalte konfiguriert werden.

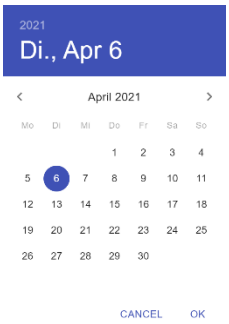
Input leer	Input akzeptiert nur > 0	Input akzeptiert nur “Zukunft”
		

Weitergabe “onSubmit”

Hier wird das Verhalten des Formulars nach Betätigung des Bestätigungsbuttons definiert (es muss innerhalb des Formulars ein Button definiert werden, dessen Funktion sich innerhalb des *Formik*-Containers befindet. In unserem Beispiel “*Speichern*”).

Formular “<Form>”

An dieser Stelle werden nun jegliche Inputfelder und Buttons definiert. Um die jeweiligen Felder mit dem *Formik* zu verknüpfen wird jeweils der zugehörige Komponentenname angegeben (component).



Um Inputfelder mit Datumsbehandlung einfach zu erstellen, dient *formik-material-ui-pickers.* Hier kann ein umfängliches Kalenderdesign unterhalb des *Formiks* angegeben werden, um diese schliesslich innerhalb des Formulars konsumieren zu können. Für die Verwaltung der Datumsangaben dient die Library *date-fns*. Dabei können bspw. Format und Lokalität definiert werden.

Einige der Validierungen von Eingabedaten wurden versuchsweise mit der *Yup* Library implementiert. Die *Formik* Library bietet verschiedene APIs für die Validierung von Eingabedaten, bietet aber auch ein spezielles Interface für die Validierung mit der *Yup* Library. Sie bietet das Konzept des *Validation-Schemas*, um die Validierungsregeln und deren Fehlermeldungen in Form von Objekten einfach lesbar zu definieren und für die Validierung im Formular zu verwenden.

6.1.6 State

React basiert auf dem Konzept von *Immutable State*, um Veränderungen im Zustand zu erkennen und damit verbunden ein Rerendering des UIs auszulösen. Um den Umgang mit *Immutable State* zu vereinfachen wurde die *Immer* Library verwendet. Sie bietet die Möglichkeit Veränderungen an einem Objekt/State über einen temporären Draft-State vorzunehmen und diesen, nach Abschluss der benötigten Änderungen, in den nächsten gültigen Zustand umzuwandeln.

Global geteilter Applikationszustand wurde mithilfe der *Recoil* Library gemanaged. Diese simple Statemanagement-Library genügte für die Anforderungen der Applikation. Der Einsatz von komplexeren Statemanagement-Libraries, wie z. B. Redux, war für die vorliegende Applikation nicht nötig.
Mit *Recoil* wurde der global geteilte Applikationszustand in verschiedenen State-Files thematisch gruppiert als Sammlung von *Atoms* und *Selectors* implementiert.

Für die Persistierung von global in der Applikation geteiltem Zustand wurde *Recoil-Persist* verwendet. *Recoil-Persist* unterstützt verschiedene Storage-Möglichkeiten. Für die Persistierung der relevanten Zustand-Atoms wurde *LocalStorage* verwendet.
Damit wurde unter anderem die Information über den aktuell eingeloggten Benutzer, d.h. die Authentifikationsinformation, persistent auf dem System des Benutzers abgelegt.

Typischerweise muss im global geteilten Applikationszustand auch Information über den Lade- bzw. Fehlerzustand von laufenden Backendzugriffen gehandhabt werden. Ausserdem werden die geladenen Daten ebenfalls häufig im Applikationszustand abgelegt. Durch den Einsatz von *React-Query* konnte dieses Handling aus dem global geteilten Applikationszustand in eine spezialisierte Library ausgelagert werden. Siehe hierzu auch das entsprechende Unterkapitel zu *HTTP-Requests*.

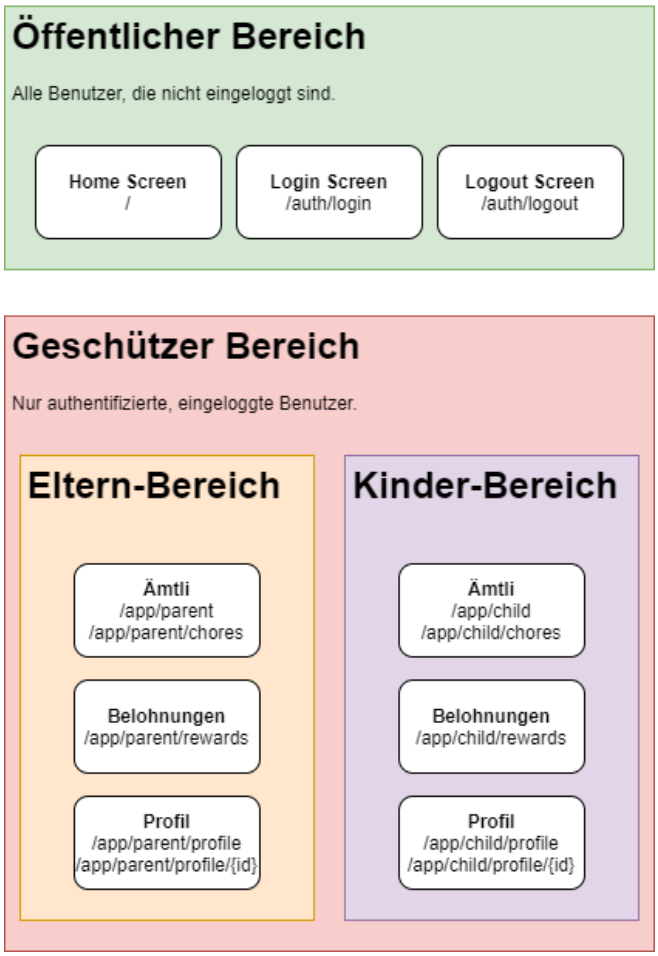
6.1.7 Confetti

Enthält eine Komponente, mit der ein konfigurierbarer Konfetti Effekt über die bestehenden Komponenten gelegt werden kann. Als Properties müssen u.a. die Länge und Breite des Effektes übergeben werden. Diese wurden durch das Auslesen der Properties des umgebenden Elementes bestimmt (via *Ref* auf dieses Element).

6.1.8 Routing

Für das Routing wurde die Library *React Router* eingesetzt. Damit wurde ein *HTML Routing* umgesetzt.

Die Applikation ist in einen öffentlichen und in einen geschützten Bereich unterteilt:



Öffentlicher Bereich

Auf den öffentlichen Bereich können alle Benutzer zugreifen. Hier befinden sich der Home-, Login- und Logout-Screen. Nicht eingeloggte Benutzer werden automatisch auf den Login-Screen umgeleitet, wenn sie auf irgendeine unbekannte Route navigieren.

Geschützter Bereich

Auf Routen im geschützten Bereich kann nur durch authentifizierte und eingeloggte Benutzer navigiert werden. Nach dem Login wird der Benutzer entsprechend seiner Rolle (Elternteil oder Kind) in die passende Route geleitet. Das Routing stellt sicher, dass der Benutzer nicht aus seinem, durch die Rolle definierten, Applikationsbereich ausbrechen kann. Navigiert der Benutzer auf eine unbekannte oder für die Rolle nicht erlaubte Route, dann wird der Benutzer automatisch in den erlaubten Bereich umgeleitet.

6.2 Backend

Das Backend wurde als *ASP.NET Web API* Projekt initialisiert und danach mit den benötigten Bestandteilen erweitert. Die Möglichkeit *REST API*s und *SignalR Hubs* zu hosten, müssen nicht explizit als Pakete dazu genommen werden, sondern sind Teil von *ASP.NET*.

Die folgende Tabelle wurde direkt aus dem Project File (*MarbleCollectorApi.csproj*) des Backend extrahiert und enthält eine Übersicht aller Abhängigkeiten, welche im Anschluss im Detail einzeln oder gruppiert nach Thema erklärt werden.

Package	Thema	Kurzbeschreibung
Microsoft.AspNetCore.Authentication.JwtBearer	Authentication	Tokens
Microsoft.EntityFrameworkCore.Design	Database	Entity Framework
Microsoft.EntityFrameworkCore.Sqlite	Database	Entity Framework
Microsoft.IdentityModel.Tokens	Authentication	Tokens
Swashbuckle.AspNetCore	API Doc	API Doc Generator
System.IdentityModel.Tokens.Jwt	Authentication	Tokens

6.2.1 REST API

Wie bereits erwähnt ist die *REST API* Teil von *ASP.NET* und es wurde grösstenteils der Standard verwendet, weshalb wir an dieser Stelle auf die offizielle Dokumentation verweisen.

6.2.2 SignalR Hub

Wie bereits erwähnt, ist *SignalR* ein Teil von *ASP.NET* und musste deshalb nicht speziell konfiguriert werden. Die folgenden Schritte waren nötig, um es nutzen zu können:

1. SignalR Hub erstellen
2. SignalR konfigurieren

6.2.3 Authentication/Authorization

Wie bereits erwähnt wurde das User Management grösstenteils weggelassen und die Testbenutzer in der Datenbank hardcodiert. Nichtsdestotrotz wollten wir eine saubere Authentication und Authorization auf dem Frontend und auch im Backend implementieren.

Als Vorlage diente ein Blog Post, welcher auf unsere Applikation adaptiert wurde. Das Projekt setzt auf selbst ausgestellte *JWT Tokens*, welche beim Login vergeben werden.

Configuration

Im *appsettings.json* gibt es eine eigene Konfiguration, welche für die Erstellung der Tokens verwendet und durch den *AuthService* konfiguriert wird.

```
"Authentication": {
  "TokenSecret": "???",
  "TokenLifespan": 2592000,
  "TokenIssuer": "MarbleCollectorApi-DEV",
  "TokenAudience": "MarbleCollectorApi-DEV-Clients"
 }
```

Middleware Integration

Für die Integration der Tokens ins *Dotnet* Ökosystem werden die Standardmechanismen von *Dotnet* verwendet. So wird die Authentication und die Authorization im Startup aktiviert und konfiguriert. Für den Schutz der API Controllers können die normalen *[Authorize] Attribute* auf den *API Controller* Klassen eingesetzt werden.

Rollen

Bei der Erstellung der *JWT Tokens* wird ein *Claim* für die Rolle des Benutzers eingefügt. Dieses *Claim* wird später auch durch die *Dotnet* Middleware verwendet, beispielsweise wenn ein Endpunkt der API nur für eine gewisse Rolle verfügbar sein soll.

So sollen nur Eltern einem Kind eine Hausarbeit zuweisen können.

6.2.4 Database

Zur Persistierung der anfallenden Daten haben wir eine relationale Datenbank, in unserem Fall *SQLite* verwendet. Sie schien uns für unseren Einsatz optimal, da die Datenstruktur nicht sonderlich komplex war, und alle Daten in einem File abgelegt sind, was das Handling erleichtert. Für den Zugriff auf die *SQLite* Datenbank haben wir das *Entity Framework* von *.Net Core* verwendet, da dies einen einfach abstrahierten Zugriff auf die Datenbank bietet und einige aus unserem Team auch bereits Erfahrungen damit gesammelt haben.

6.2.5 API Documentation

Die Dokumentation der *REST API* Endpoints bzw. Ressourcen wurde durch einen separaten *Swagger*-Endpoint gelöst. Mithilfe der *Swashbuckle* Library wird der *Swagger*-Endpoint dynamisch durch die *ASP.NET* Applikation aufgrund der vorhandenen Code-Dokumentation generiert.

Der *Swagger*-Endpoint dokumentiert alle vorhandenen *REST API* Endpoints.

Die *SignalR Hubs* für die Notifizierung sind nicht Teil der *Swagger*-Dokumentation.

6.3 Tools

6.3.1 IDEs

Visual Studio Code

Für die Entwicklung des Frontend-Codes wurde *Visual Studio Code* verwendet. Durch den Einsatz von *Prettier* wurde die Formatierung des Codes standardisiert. Informationen zum Setup von *Prettier* finden sich im *README.md* File im *client*-Verzeichnis des Projekt-Repositories.

Visual Studio

Für die Entwicklung des Backend-Codes wurde *Visual Studio* verwendet. Hier wurden keine speziellen Einstellungen vorgenommen.

6.3.2 API Type Generation

Mit Hilfe des Typewriter Tools konnten aus den im *API* Projekt erstellten *DTO's* direkt *Typescript* Code generiert werden, welcher im Frontend Projekt verwendet werden konnte. Damit konnte der Aufwand und Fehlerquellen minimiert werden.

6.3.3 DB Browser for SQLite

Der *DB Browser for SQLite* ist ein kleines, leichtgewichtiges Open Source Tool, um *SQLite* Datenbankfiles zu öffnen und zu bearbeiten. Wir haben das Tool in erster Linie dazu genutzt, in einem frühen Stadium der Entwicklung die Struktur und den Inhalt der abgelegten Daten zu prüfen und bei Bedarf zu bearbeiten.

7 Design und Umsetzung

7.1 Screens

7.1.1 Vorgehen

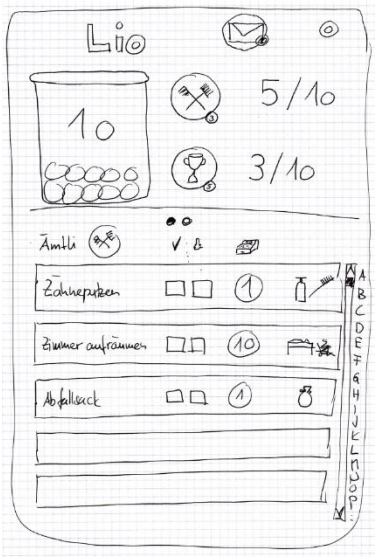
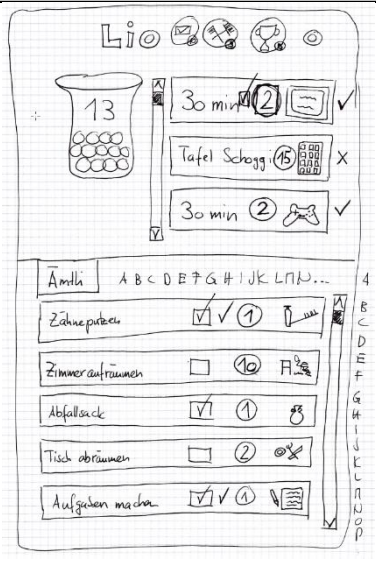
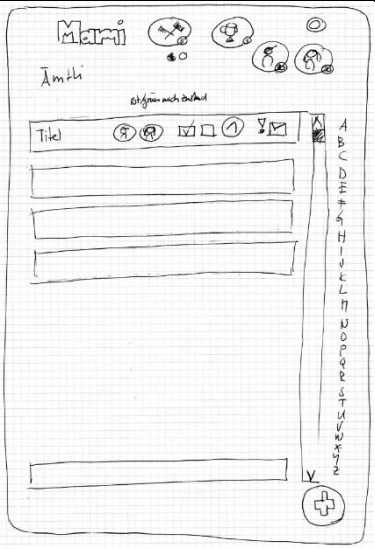
Inspiriert durch bzw. unter Anleitung im Kurs *Usability* wurde in mehreren Iterationen unter Zuhilfenahme des *UCD*-Prozesses (*User Centered Design*) das Design der Applikation entwickelt. Nach verschiedenen Iterationen wurden anhand von User Tests die bisherigen Ergebnisse bzw. Implementationsschritte verifiziert (siehe auch Kapitel *User Tests*).

- Anforderungsanalyse und Erstellung erster Papier-Prototypen zusammen mit Vertretern der Benutzergruppen *Eltern* und *Kind* (siehe auch Schritt 1 im Kapitel *User Tests*).
- Verfeinerung des Papier-Prototypen in zwei Iterationen innerhalb des Projektteams.
- Erstellung eines klickbaren Prototyps mithilfe des Wireframing-Tools Balsamiq.cloud als Vorbereitung auf einen ersten User Test (siehe auch Schritt 2 im Kapitel *User Tests*).
- Implementation einer ersten Version des Parent-Dashboards und deren Überprüfung in einem User Test (siehe auch Schritt 3 im Kapitel *User Tests*).
- Überarbeitung der Implementation des Parent-Dashboards und Fertigstellung einer ersten Version des Child-Dashboards. Überprüfung in einem End-2-End User Test mit Vertretern aus den Benutzergruppen *Eltern* und *Kind* (siehe auch Schritt 4 im Kapitel *User Tests*).
- Überarbeitung der Implementation anhand der Erkenntnisse aus dem End-2-End User Test.

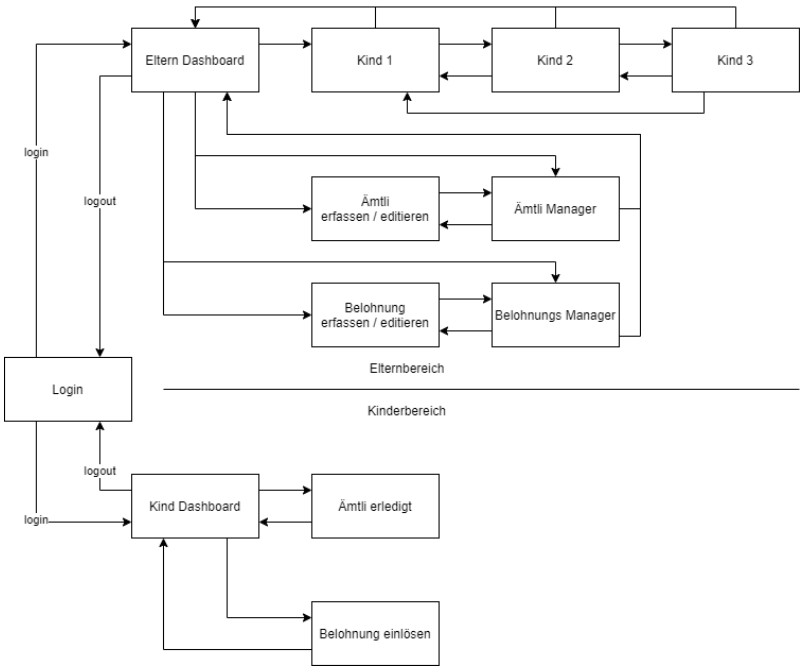
7.1.2 Entwicklungsschritte

Erste Entwürfe

In ersten Entwürfen für das Design der Dashboards wurden viele Bedienelemente bzw. Informationen auf dem beschränkten Platz des Bildschirms angeordnet:

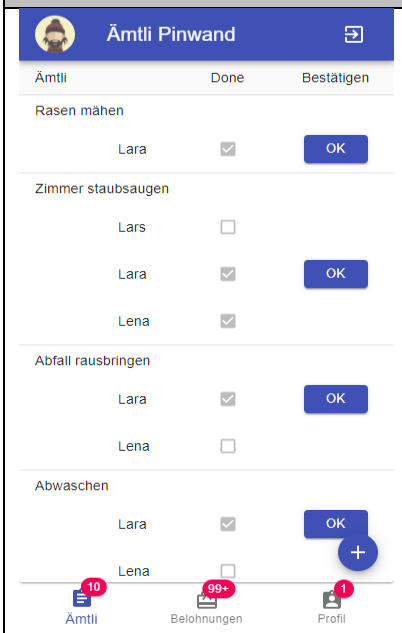
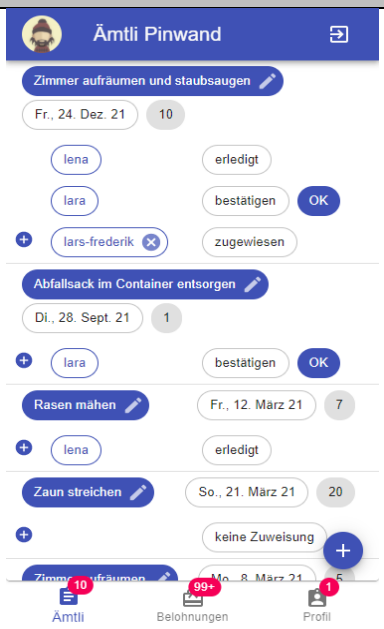
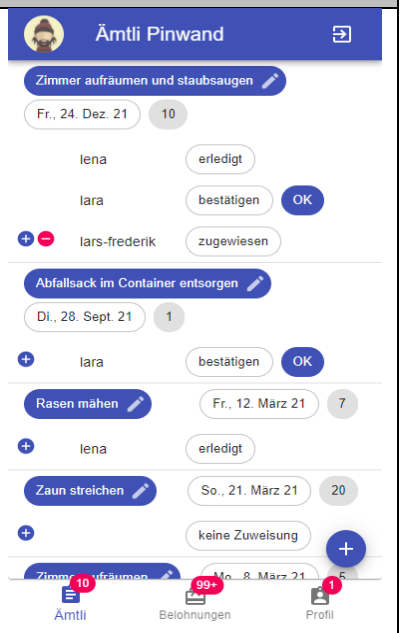
Kind-Dashboard Variante 1	Kind-Dashboard Variante 2	Eltern-Dashboard
		

Ein erster Entwurf des Screen-Flows:



Erste Implementationen

Die nachfolgenden Screenshots zeigen anhand des Eltern-Dashboards die initiale Implementation und mögliche überarbeitete Varianten. Es ist zu sehen, wie versucht wurde mehr Information und mehr Funktionalität bzw. Bedienelemente im Design unterzubringen:

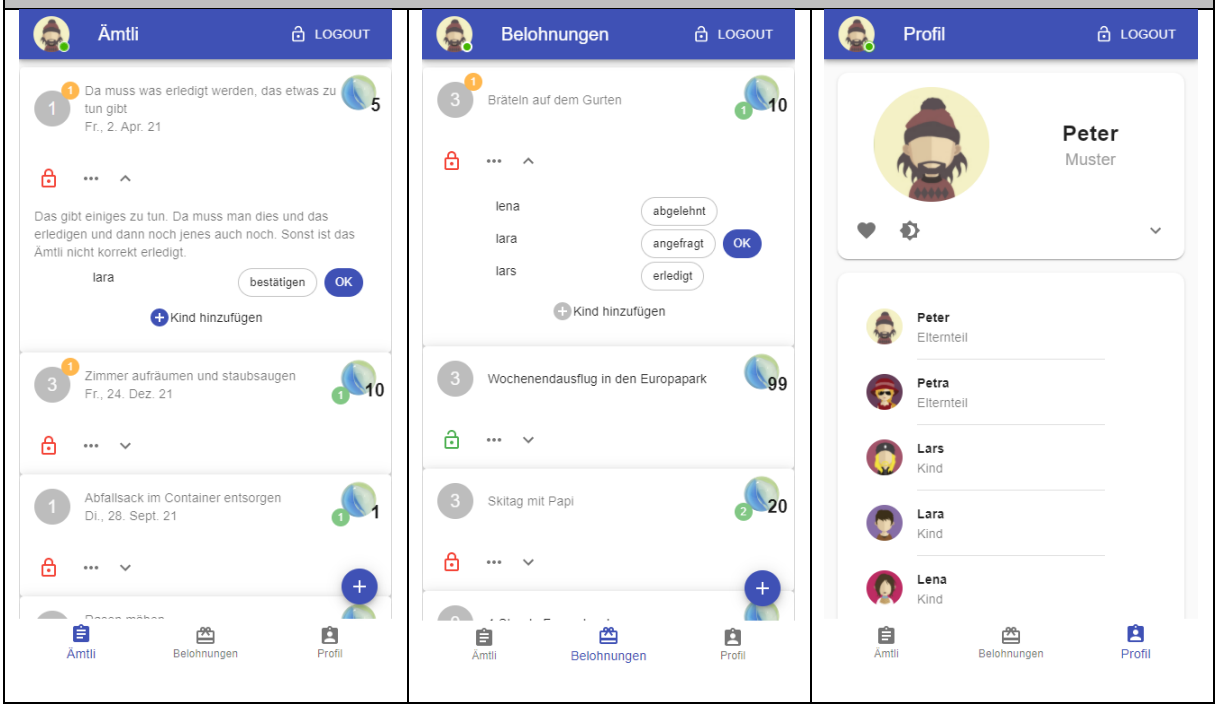
Eltern		
Initiale Implementation		
		

Der Screen-Flow wurde beinahe unverändert zum ursprünglichen Entwurf übernommen.

Finales Design

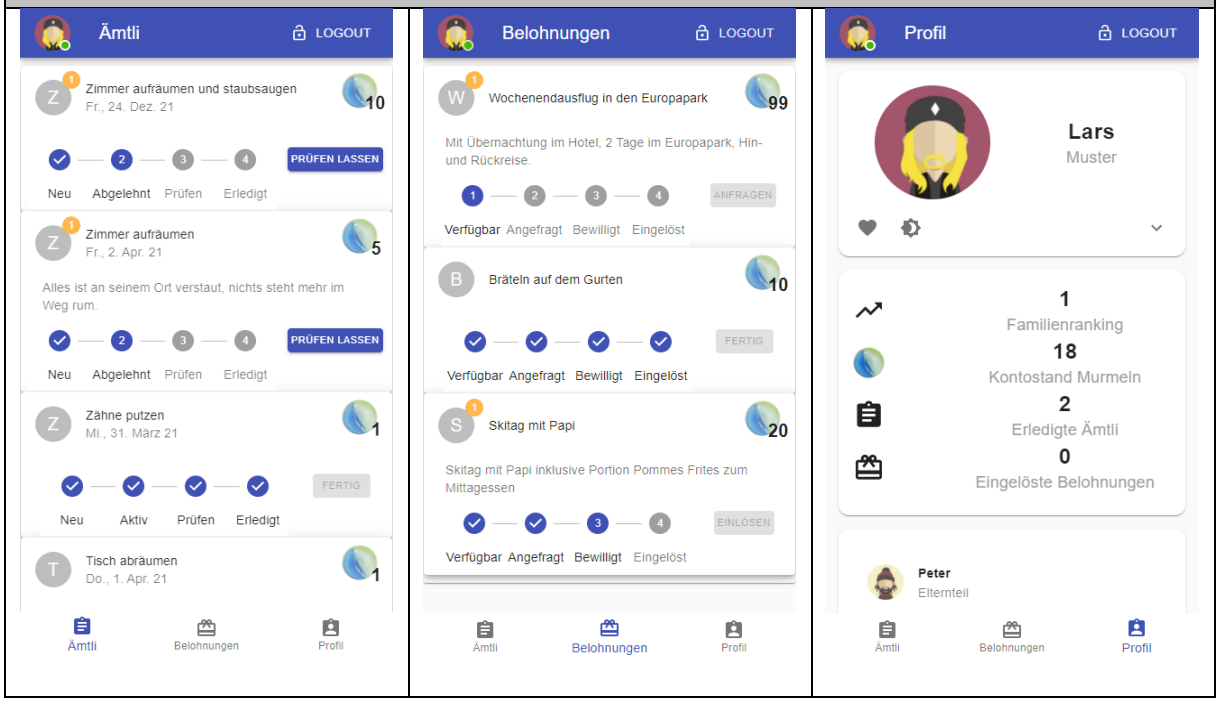
In mehreren Iterationsschritten wurde versucht die Menge an dargestellter Information auf dem Eltern-Dashboard geschickt in geeignete Bedienelementen zu kombinieren. Ebenfalls wurden Farben und Bilder bzw. Icons als Informationsträger eingefügt. Durch die Möglichkeit Inhalt ein- bzw. auszublenden wurde der Platzbedarf von einzelnen Ämtli- oder Belohnungselementen reduziert. Für den Benutzer ist auf wenig Platz viel Information vorhanden.

Eltern



Auf dem Kind-Dashboard ist grundsätzlich weniger Information darzustellen. Hier ist allerdings wichtig, die für das Kind relevanten Informationen übersichtlich und einfach verständlich darzustellen. Aus diesem Grund wurde die *Stepper* Komponente von *Material UI* gewählt, welches auf einen Blick den Zustand der einzelnen Elemente anzeigt. Durch gezieltes Anpassen der jeweiligen Texte auf den Button und *Stepper* Elementen konnte das Feedback an den User nochmals verbessert werden. Das restliche Design des Dashboards wurde an das Eltern Dashboard angelehnt.

Kind



7.2 Technische Aspekte

Da das Kapitel 6 bereits den grössten Teil der technischen Aspekte dieser Arbeit abdeckt, soll in diesem Abschnitt auf diese nicht mehr eingegangen werden.

7.2.1 Benachrichtigungen

Ein zentraler Bestandteil dieses Projekts sind die Benachrichtigungen, welche an die verschiedenen Benutzer verschickt werden. Auf diesen Aspekt soll hier genauer eingegangen werden.

Durch die Struktur und die Funktionalität unserer Anwendung haben wir uns für zwei Hubs entschieden:

ParentNotificationHub

Der *Parent Notification Hub* wird getriggert, wenn in den Controllern ein Update durch die Kinder stattfindet (Anfrage für Belohnung, Ämtli überprüfen, etc.), welches für das Parent Dashboard relevante Informationen enthält. Der Aufruf geschieht direkt in der entsprechenden Methode der Controller. Die zu pushende Nachricht enthält jeweils die UserId, die Id der Zuweisung, sowie einen Notifcation Typen. Mit diesen Informationen wird im Frontend dann das Nachladen der entsprechenden Daten angestossen.

ChildrenNotificationHub

Der *Children Notification Hub* wird getriggert, wenn in den Controllern ein Update durch die Eltern stattfindet (neue Belohnung zugeordnet, Ämtli überprüft, etc.), welches für das Children Dashboard relevante Informationen enthält. Der Aufruf geschieht direkt in der entsprechenden Methode der Controller. Die zu pushende Nachricht enthält jeweils die UserId, die Id der Zuweisung, sowie einen Notifcation Typen. Mit diesen Informationen wird im Frontend dann das Nachladen der entsprechenden Daten angestossen.

Typen von Notifications

Die folgenden Notifikationstypen kennt der MarbleCollector:

Typ	Auslöser	Beschreibung
AssignmentCreated	Eltern	Ein Ämtli wurde zugewiesen, Nachricht an Kind
AssignmentUpdated	Eltern Kind	Der Status eines Ämtlis wurde verändert, Nachricht an Kind / Eltern, je nachdem, wer den Status verändert hat.
AssignmentDeleted	Eltern	Eine Zuweisung zu einem Ämtli wurde gelöscht, Nachricht an Kind
GrantCreated	Eltern	Eine Belohnung wurde zugewiesen, Nachricht an Kind.
GrantUpdated	Eltern Kind	Der Status einer Belohnung wurde verändert, Nachricht an Kind / Eltern je nachdem, wer den Status verändert hat.
GrantDeleted	Eltern	Eine Zuweisung zu einer Belohnung wurde gelöscht, Nachricht an Kind.

7.3 Testing

7.3.1 Unit Tests

Aufgrund der kurzen Projektdauer, der zu erwartenden kurzen Lebensdauer des Projekts und der relativ geringen Komplexität der Businesslogik der Applikation wurde auf umfangreiche automatisierte Unit Tests verzichtet, dies sowohl im Front- wie auch im Backend. Implementierte Funktionalität wurde durch manuelle Tests durch das Projektteam verifiziert. Das Projektteam ist sich bewusst, dass damit auftretende Regressionen nicht automatisch entdeckt werden können, hat diesen Nachteil aber wohlweislich in Kauf genommen.

7.3.2 User Tests

An verschiedenen Punkten im Projekt wurden User Tests durchgeführt:

- Ein erster Entwurf des UIs wurde in Zusammenarbeit mit Vertretern der Benutzergruppen *Eltern* und *Kinder* ausgearbeitet.
    - Der direkte Input der Vertreter der Benutzergruppen liefert interessante Ansätze für das Design.
- Nach der iterativen Verfeinerung des ersten Entwurfs durch das Projektteam wurde dieser verfeinerte Entwurf in einer Online-Session mit einer Vertreterin der Benutzergruppe *Eltern* unter Zuhilfenahme des *Wireframing*-Tools *Balsamiq.cloud* einem User Test unterzogen.
    - Hier war interessant zu beobachten, wie bestimmte Aspekte des Designs von der Vertreterin der Benutzergruppe anders verstanden wurden als vom Projektteam angedacht. Auch waren vom Projektteam als wichtig eingestufte Anforderungen für die Vertreterin der Benutzergruppe weniger wichtig. Grundsätzlich konnte aber der verfeinerte Entwurf den User Test bestehen.
- Eine erste Implementation des Parent-Dashboards wurde mit einer Vertreterin der Benutzergruppe *Eltern* in einer Vor-Ort-Sitzung einem User Test unterzogen.
    - Es zeigte sich, dass das Design der Applikation nicht durchgängig selbsterklärend ist. Der Benutzer benötigt eine kurze Einführung über die Konzepte der Applikation, um deren Bedienelemente korrekt interpretieren zu können.
- Nach Abschluss der Implementation einer ersten Version der Applikation wurde ein ausgiebiger User Test mit einer ganzen Familie bestehend aus zwei Elternteilen und zwei Kindern durchgeführt. Ausgerüstet mit Smartphones wurde in einem Real-World-Szenario die grundsätzliche Funktionalität und Bedienbarkeit überprüft. Die Eltern waren angehalten “echte” Ämtli und Belohnungen zu erstellen und ihren Kindern zuzuweisen.
    - Der User Test war ein voller Erfolg. Die Kinder erledigten ihre Ämtli mit riesigem Enthusiasmus, sodass den Eltern nach kurzer Zeit die Ideen für mögliche Ämtli ausgingen. Der Test hat auch einige kleinere Schwachstellen im Design der Applikation gezeigt, aber trotzdem bewiesen, dass das Design grundsätzlich die Anforderungen erfüllt. Die Erkenntnisse flossen soweit möglich in die Verbesserung des Designs und der Implementation ein oder wurden für spätere Verbesserungen gesammelt.
- Ein erster Release-Kandidat wurde mit einem anderen Vertreter der Benutzergruppe *Eltern* in einer Vor-Ort-Sitzung einem User Test unterzogen.
    - Erneut zeigte sich, dass das Design der Applikation nicht durchgängig selbsterklärend ist. Auch dieser Benutzer benötigte eine kurze Einführung über die Konzepte der Applikation, um deren Bedienelemente korrekt interpretieren zu können.

7.3.3 End-2-End Tests

Auf automatisierte End-2-End Tests wurde aufgrund der kurzen Projektdauer, der zu erwartenden kurzen Lebensdauer des Projekts und der relativ geringen Komplexität der Applikation verzichtet. Implementierte Funktionalität wurde durch manuelle Tests durch das Projektteam zu verschiedenen Zeitpunkten verifiziert. Das Projektteam ist sich bewusst, dass damit auftretende Regressionen nicht automatisch entdeckt werden können, hat diesen Nachteil aber wohlweislich in Kauf genommen.

Mit einem ausgiebigen manuell durchgeführten End-2-End User Test wurde die grundsätzliche Funktionalität der Lösung in einem Real-World-Szenario überprüft (siehe Kapitel *User Tests*).

8 Herausforderungen

Durch die verschiedenen Aufgaben, welche die Mitglieder des Teams im Laufe des Projektes realisiert haben, und auch den unterschiedlichen Erfahrungsschatz, den jeder der Studenten in das Projekt mitgebracht hat, sind die Herausforderungen im Projekt sehr unterschiedlich ausgefallen. Wir haben uns deshalb dazu entschieden, die Herausforderungen personenbezogen aufzulisten.

8.1 Marcel

ReactQuery – relativ komplexe Angelegenheit

Eine der Aufgaben von mir war es, ein zentrales Modul für den *API* Zugriff zu erstellen, welches dann in den verschiedenen Komponenten wiederverwendet wurde. Wir haben uns im Projekt dafür entschieden, *React Query* für den *API* Zugriff zu verwenden. Dabei musste ich feststellen, dass die Lernkurve für *React Query* relativ steil ist, vor allem, wenn die Möglichkeiten und Vorteile ausgeschöpft werden sollen.

Branches und Pull Requests

Durch die parallele Entwicklung, und die teilweise enge “Verzahnung” der einzelnen Komponenten und Funktionen, waren wir sehr auf die Arbeitsergebnisse der anderen Teammitglieder angewiesen. Wir haben uns von Anfang an dazu entschieden, die Branches und *Pull Requests* relativ feingranular zu gestalten. Dadurch stieg aber der Aufwand für das Abschliessen von *Pull Requests* und die Anzahl Branches. Hier den Überblick zu bewahren war nicht immer ganz einfach.

8.2 David

SignalR – Notifications in Kombination mit Recoil

Der ursprüngliche Plan die *SignalR* Connection direkt als globalen State der App zu speichern ist fehlgeschlagen, da es beim Setzen des States einen unbekannten und nicht lösbaren Fehler in *Recoil* gab. Aus diesem Grund musste ein Workaround gefunden werden, damit die Vision trotzdem realisiert werden konnte, dass sich Komponenten auf Events, welche sie interessieren abonnieren können.

Die Connection wurde im Anschluss als lokaler State einer Komponente angelegt («*DashboardNotificationHandler*»), welche immer in der Dashboard Shell gemounted ist. Diese Komponente bezieht über den *Recoil*-state die benötigten Subscribers/Events und legt die Event Callbacks auf der Connection an. Die Notifications werden danach ebenfalls durch die Event Callbacks in einen globalen State geschrieben, dessen Konsumation durch ein Set von *Hooks* («*NotficationHooks.ts*») abstrahiert wird. Mit diesen Hooks können sich Komponenten bspw. auch merken welche Notifications sie bereits behandelt haben und welche nicht.

Wie man an dieser Erklärung sieht, war der ganze Umgang mit den Notifikationen eine Herausforderung und es gäbe sicher noch Verbesserungspotenzial was die Performance und Architektur betrifft.

Routing mit Protected Routes

Die Absicherung des “geschützten” Bereichs der App mit “protected Routes” war eine meiner Aufgaben, die sich als grössere Herausforderung herausstellte als ursprünglich angenommen.

Das Routing als deklarativer Approach war ein Mindset Change, an den ich mich zuerst gewöhnen musste. Insbesondere wenn mal etwas nicht funktioniert hat, war es teilweise schwierig dem Problem auf die Schliche zu kommen.

Auch jetzt ist die Lösung sicher noch nicht ideal, da die Komponenten jeweils geladen werden und erst im Rendering das Redirect vollzogen wird (Bsp. *ChildScreen.tsx*). Jedoch funktioniert das Routing so wie wir es definiert haben und ein User kann nicht auf einen nicht für ihn vorgesehenen Bereich zugreifen.

Authentication mit JWT Tokens

Die Implementation der Authentication mit *JWT Tokens* war sehr interessant, aber auch ein bisschen herausfordernd. Wahrscheinlich hätte man es für diese Projektarbeit einfacher lösen können, dennoch war es eine interessante Erfahrung, auch besonders für Real Life Anwendungen.

Koordination der Aufträge als Team/Merge Konflikte

Wie bereits von Marcel erwähnt, waren wir als Team parallel an der Entwicklung von Komponenten und Funktionen beteiligt, die eng im Zusammenhang standen. Die Koordination und Abstimmung dieser Funktionen, und auch das anschliessende Mergen, resp. das Lösen von Merge Konflikten war nicht immer ganz einfach, da einem dabei teilweise die Blickweise des Teamkollegen fehlte.

8.3 Severin

Mobile Devices als Zielplattform

Einer meiner grossen Aufgaben war das Design des Children Dashboards. Die Fokussierung auf Mobilgeräte als Zielplattform stellte sich als eine der grossen Herausforderungen dar. So waren viele Entwürfe, welche auf einer Tablet-Auflösung noch gefällig erschienen, auf einem Handy mit kleiner Auflösung nicht mehr realisierbar oder überladen. Das führte zu einigen Designiterationen mehr, als wir das wohl ursprünglich geplant hatten.

Koordination innerhalb des Teams / Unterschiedliche Velocity

Wie bereits in der Einleitung erwähnt, brachten die verschiedenen Teammitglieder unterschiedliche Erfahrung in das Projekt ein. Dies führte auch zu einer unterschiedlichen Velocity bei der Umsetzung von Arbeitspaketen. Eine der grossen Herausforderungen im Projekt war es, die Grösse und Verteilung dieser Pakete so zu koordinieren, dass alle Mitglieder in ihrem Tempo an den Paketen arbeiten konnte, und dabei keines der Mitglieder zu lange auf Pakete warten mussten.

Abweichendes Styling von Material UI Komponenten

Bei einigen Komponenten war es durch die engen Platzverhältnisse nötig, mehr als nur die standardmässigen Styling Möglichkeiten von *Material UI* zu nutzen. Hier zeigte sich, dass dabei der Aufwand sofort um einiges grösser wurde, als wenn nur die “normale” API der Komponente genutzt wurde.

8.4 Jan

Datenreload vom Backend

Im vorliegenden Projekt mussten durch die hohe Interaktivität häufig Daten nachgeladen werden. Eine der grossen Herausforderungen dabei war, sicherzustellen, dass dabei immer genau die richtigen Daten nachgeladen wurden, ohne dabei zu viele Daten nachzuladen. Beim aktuellen Stand des Projektes sind wir dabei eher den “sicheren” Weg gegangen und haben im Zweifelsfalle eher mehr Daten nachgeladen als grundsätzlich nötig.

Styling von Material UI Komponenten

Eine der “Herausforderungen” war auch das Styling der einzelnen *Material UI* Komponenten, da verschiedene APIs von *Material UI* unterstützt werden und hier nicht immer ganz klar war, welcher Weg der richtige, respektive der effizientere ist.

Mobile Devices als Zielplattform

Wie bereits von Severin erwähnt war auch eine meiner grössten Herausforderungen der eingeschränkte Platz der mobilen Devices als Zielplattform. Vor allem ältere Geräte bieten hier sehr wenig Platz, um die benötigte Information dem Benutzer zugänglich zu machen.

Informationsdichte auf Elterndashboard

Das Elterndashboard stellte eine sehr grosse Herausforderung dar, was die darzustellende Dichte von Informationen und Bedienelementen betrifft. Wir hatten uns zum Ziel gesetzt, ein Design zu entwerfen, welches von allen Bedienern schnell und ohne grosse Einführung verstanden werden soll. Dieses sollte trotzdem alle nötigen Informationen darstellen und dabei mit wenigen Bedienschritten bedient werden können soll. Erste Tests mit meiner Familie als Testpersonen haben sich dabei als sehr vielversprechend herausgestellt.

9 Lessons Learned

Wie bei den Herausforderungen, haben sich auch die Lessons Learned sehr individuell gestaltet. Deshalb möchten wir auch hier auf alle Teammitglieder einzeln eingehen.

9.1 Severin

Positiv

Super Team, jeder hat sich aktiv daran beteiligt das Projekt zu einem erfolgreichen Abschluss zu bringen. Angenehme Zusammenarbeit, immer konstruktive Kritik, ohne dabei den Fokus für das wesentliche zu verlieren. Teilweise erstaunlich detailliertes Feedback auf *Pull Requests*.

Mit dem Ergebnis unserer Arbeit können wir (aus meiner Sicht) sehr zufrieden sein.

Die Wahl von *Material UI* hat sich für mich persönlich als sehr gut herausgestellt. Das Erstellen von Frontend Komponenten geht sehr rasch und intuitiv, ohne einen grossen Einarbeitungsaufwand. Ich würde *Material UI* auf jeden Fall noch in weiteren Projekten nutzen.

Einsatz von Libraries wie *Recoil* / *Immer* / *React Query* hat uns die Arbeit deutlich erleichtert, obwohl der Einstieg nicht ganz leichtgefallen ist. Würde ich bei zukünftigen Projekten wieder einsetzen.

Das Projekt hat sich sehr gut dazu geeignet, um meine Erfahrungen in *React* zu vertiefen.

Verbesserungspotential

Der Aufwand bei der Umsetzung hat sich für mich als grösser herausgestellt, als ich das ursprünglich erwartet hätte. Auch die Abstimmungssitzungen waren meist zeitaufwändiger als ich mir das gewünscht habe.

Fazit

Wie bereits oben erwähnt, bin ich persönlich mit dem Erreichten sehr zufrieden. Auch mit der Wahl des Teams konnte ich mich sehr glücklich schätzen. Wie sich wieder einmal bewahrheitet hat ist "*learning by doing*" die beste Methode, theoretisch Erlerntes zu festigen und auszubauen. Beim nächsten Mal würde ich versuchen, den Abstimmungsaufwand zu verringern und die Abstimmung der einzelnen Tasks zu verbessern.

Auch würde ich ein anderes Repo als *GitHub* wählen, da das *Pull Request* Handling hier etwas mühsam scheint.

9.2 Marcel

Positiv

Ich war überwältigt von meinem Team, mit welchem ich arbeiten durfte. Obwohl ich merklich zu wenig Knowhow mitbringen konnte, um dieses *CAS* angemessen zu bewältigen, konnte ich stets auf ihre Hilfe zählen. Somit konnte ich ebenfalls meinen Anteil an diesem spannenden, und umfangreichen Projekt leisten.

Diese Arbeit eignete sich ausgezeichnet, um einen grossen Einblick in *React* und *TypeScript* zu erfahren. Beides war für mich Premiere. Mit *Material UI* haben wir ebenfalls eine super Wahl getroffen! Es ist einfach gehalten trotz seiner Komplexität und hat eine umfangreiche Dokumentation vorzuweisen. Ich werde diese Konstellation gerne weiterverwenden. *React Query* hat es mir besonders angetan, obwohl es sehr schwierig zu meistern ist. Allgemein vereinfacht es aber die ganze “Datenschieberei” extrem. Auch mit *Recoil* konnte ich nur positive Erfahrungen sammeln, trotz anfänglicher Skepsis bezüglich Foren und Hinweise.

Verbesserungspotential

Beim nächsten Projekt würde ich meinen, dass die Arbeitsaufteilung besser koordiniert werden könnte. Mit der Zeit wurde des Öfteren ersichtlich, dass sich einige Arbeiten überschneiden und dementsprechend auch zu erledigenden Tasks mit Zeitaufwand wieder stimmig gemacht werden mussten. Wenn wir von Anfang an die Cases klarer trennen würden, könnte hier auch wieder Zeit eingespart werden.

Fazit

Alles in allem bin ich mit dem Ergebnis überaus zufrieden und würde gerne wieder in einem solchen Projekt mithelfen. Ich konnte mir vieles aus diesem *CAS* insbesondere dieser Arbeit mitnehmen, dass ich in meinen Arbeitsalltag einfliessen lassen werde. Abschliessend würde ich somit das Projekt als sehr gelungen markieren!

9.3 Jan

Positiv

Spannende Zusammenarbeit mit unterschiedlichen Charakteren im Projektteam. Trotz der speziellen Umstände (Covid-19), in denen wir uns kaum persönlich kennenlernen konnten, hat die Zusammenarbeit sehr gut funktioniert.
Die wöchentlichen Online-Statusmeetings haben sich bewährt, um die laufenden Arbeiten zwischen den Teammitgliedern zu koordinieren.
Trotz der unterschiedlichen Arbeitstempi der Teammitglieder konnten wir ein sehr ansprechendes Projekt in einer angenehmen Arbeitsatmosphäre umsetzen.

Die eingesetzten Libraries haben sich durchwegs bewährt.
Mit *Material UI* lassen sich mit verhältnismässig wenig Aufwand ansprechende User Interfaces bauen, wobei eine gewisse Einarbeitungszeit speziell in die verschiedenen Styling-Möglichkeiten nicht vernachlässigt werden darf.
Recoil hat sich als sehr einfache Alternative für das State-Management gezeigt. Zusammen mit *Recoil Persist* kann der Applikationszustand sehr einfach auch im lokalen Storage persistiert werden.
React Query hat sich als sehr mächtige Library für die Kommunikation mit dem Backend herausgestellt. Diese Library erleichtert den Umgang mit Fetching, Caching und Reloading von Daten enorm. Allerdings lässt sich das ganze Potenzial dieser Library kaum in so kurzer Zeit erfassen und einsetzen. Hier ist einiges mehr an Einarbeitungszeit nötig.

Verbesserungspotential

Das Handling der *Pull-Requests* in *GitHub* ist etwas umständlich, da neue Commits im Ziel Branch zu keiner automatischen Anpassung der Änderungen im *Pull-Request* führen.

Die wöchentlichen Online-Statusmeetings könnten etwas effizienter gestaltet werden. Teilweise haben wir uns in Diskussionen verloren, anstatt die anstehenden Entscheide vorwärtszutreiben und konkrete Lösungen zu Problemen zu finden.

Fazit

Die Umsetzung eines konkreten Projektes hat sich als sehr gute Möglichkeit zum Erlernen einer *JavaScript* UI Library herausgestellt. *Learning by doing* ist die einzige wirkliche Möglichkeit die tatsächlichen Problemstellungen kennenzulernen und deren Lösung zu ergründen. Das durch diese Praxis erworbene Wissen ist viel besser gefestigt als in endlosen Theoriestunden gebüffelter Input.

Viele der im Projekt erarbeiteten Lösungsansätze konnte ich mittlerweile direkt in Projekten im Geschäftsleben einsetzen.

React und dessen Ökosystem bietet eine vergleichsweise flache Lernkurve, es kann mit recht wenig Aufwand eine ansprechende *Single Page Application* entwickelt werden. Allerdings ist es nicht immer einfach den Überblick im riesigen Bibliotheken-Ökosystem von *React* zu behalten. Hier war der Theorieteil ein wichtiger Kompass, um Entscheide für den Einsatz von bestimmten Libraries fällen zu können.
Die Einarbeitungszeit in die verschiedenen Libraries darf dabei aber natürlich auch nicht vernachlässigt werden, denn es gilt verschiedenste Konzepte zu verstehen.

Die regelmässig durchgeführten User Tests haben jeweils sehr interessante Einblicke und Feedbacks durch die Benutzer zutage gefördert. Als Entwickler verliert man häufig den Blick auf das Ganze und die Probleme der Benutzer, das entwickelte Design verstehen und effizient einsetzen zu können.

9.4 David

Positiv

Die Projektarbeit war für mich eine positive Erfahrung. Es hat Spass gemacht sich mit einem coolen Team an eine Aufgabenstellung heranzumachen, welche nichts mit normalen Arbeitsthemen zu tun hatte. Die Zusammenarbeit hat grundsätzlich trotz Remote Work gut funktioniert, auch wenn die ganzen Videokonferenzen mit der Zeit müde machen.

Ich kannte *React* bisher von der Entwicklung von *SharePoint/M365* Komponenten her und konnte mit der Projektarbeit in einen anderen Teil des *React* Ökosystems Einblick gewinnen, welcher mir bisher weniger bekannt oder gar unbekannt war. Die viel grössere Freiheit in der Wahl der Abhängigkeiten habe ich geschätzt. Die Libraries, welche wir für das Projekt einsetzten, fand ich attraktiv. Insbesondere *Material UI* und *Recoil (persist)* haben es mir angetan und ich werde diese in Zukunft auf jeden Fall vermehrt verwenden.

Auch das Routing mit *React Router* war ein interessanter Aspekt, sowie *SignalR* als Framework kennenzulernen war eine Bereicherung.

Sicher auch positiv war es, dass wir relativ früh einen funktionierenden Projektsetup hinbekommen haben, welcher als Durchstich die wichtigsten Kernbausteine/Challenges abgedeckt hat und auch die Build und Deployment vorbereitet war.

Verbesserungspotential

In der Zusammenarbeit als Team könnten wir sicher die Meetings ein bisschen zielgerichteter und kürzer gestalten, dafür lieber mal eines mehr machen.

Als *GIT* Repository würde ich heute wohl eher auf *Bitbucket* setzen und nicht *GitHub*, da dort das *Pull Request* Handling and *Code Review* ein bisschen intuitiver gemanaged wird. Als *Task Management* Tool würde ich eher etwas anderes einsetzen, dafür von Anfang an konsequent.

Fazit

Das Projekt war für mich ein voller Erfolg, ich konnte vieles Lernen und auch direkt für den Arbeitsalltag profitieren. Ich fühle mich nun viel sicherer im Umgang mit *Function Components*, *Hooks* & co was auch eines der Ziele war, die ich mir fürs Projekt gesetzt hatte. Mit dem Ergebnis bin ich zufrieden, auch wenn es noch viel zu verbessern gäbe.

10 Ausblick

Minimum Viable Product

Damit die Applikation als *MVP* für eine Familie eingesetzt werden kann, müssen mindestens die nachfolgenden Punkte angegangen werden:

- Ablage der Daten in einer Datenbank ausserhalb des App-Containers. Die verwendete *SQLite*-Datenbank wird bei jedem Rollout der App gelöscht und die Daten gehen verloren.
- Keine hard-codierten Benutzer, inkl. Möglichkeit für die Benutzer ihre Passwörter anzupassen.

Erweiterungen für eine Kommerzialisierung

Um den *Marble Collector* in ein vermarktbares Produkt zu entwickeln sind unter anderem folgende Punkte umzusetzen:

- Mandantenfähigkeit:
  - Mehrere Familien werden unterstützt.
  - Benachrichtigungen werden nur an die Familienmitglieder verschickt.
  - Authentification des Zugriffs auf die *SignalR Hubs*.
  - Implementation einer “echten” Benutzerverwaltung.
- Deployment in eine produktive (bezahlte) und bezahlte Betriebsumgebung.
- Bereitstellung der Betriebsumgebung mittels Infrastructure as Code (Iac)
- Hardening des Backends.
  - Das Backend wurde mit minimalem Aufwand entwickelt und es wurde keinerlei Wert auf nicht-funktionale Anforderungen, wie z. B. Performance, Input-Validation, etc. gelegt.
- Abbau von technischen Schulden:
  - Projektstruktur des Frontend-Projekts. Der gewachsenen Ordnerstruktur vermischt verschiedene Konzepte.
  - Überarbeitung und Streamlining der Notifikationsarchitetkur
  - Wiederverwendung von ähnlichen/gleichen Komponenten auf den verschiedenen Screens der Applikation.
  - Durchgehende Implementation von Input-Validierungen mit der *yup* Library.

Verbesserungen in der User Experience

Die User Experience der Applikation kann in verschiedenen Bereichen noch verbessert werden:

- Optimierung der Logik zum Laden der Daten vom Backend.
  - Zurzeit werden immer alle Ämtli bzw. Belohnungen nach einer Änderung/Notification neu geladen.
  - In vielen Fällen ist das nicht nötig und führt zu unnötigem Netzwerkverkehr.
- Kurzes How-To mit Erklärungen zu den grundsätzlichen Konzepten und Bedienelementen.
- Umgang mit Ämtlis, welche sich bereits in einem angefangenen Zustand befinden und abgebrochen oder verändert werden sollen.
- Umgang mit Belohnungen, welche sich bereits im angefragten Zustand befinden und bei welchen die Zuordnung verändert werden soll, oder bei denen die Belohnung als solches editiert werden soll.
- Manuelle Eingriffe in den Kontostand der Kinder durch die Eltern, beispielsweise um einen Fehler bei der Erfassung eines Ämtlis auszugleichen
- Verbesserung des GUI Feedbacks bei Notifications. Was hat sich genau wo geändert?
- Konzept für die Archivierung von abgeschlossenen Ämtlis und eingelösten Belohnungen erarbeiten und umsetzen.
- Beim Ablehnen eines erledigten Ämtlis oder beim Anfordern einer Belohnung könnte es auch hilfreich sein, dem Kind eine Nachricht zu senden, was der Grund für das Ablehnen ist.
- Darstellung der Zustände der Ämtlis/Belohnungen auf dem Eltern-Dashboard mit Stepper analog zum Kinder-Dashboard.
- Filterung/Sortierung der Ämtlis/Belohnungen:
  - Nach Kind.
  - Nach Bestätigungsanfragen.
  - Nach abgelaufen bzw. noch offen.
- Lazy-Loading für die verschiedenen Bereiche/Screens der Applikation.
- Bessere Ausnützung des Screen-Estate auf Geräten mit grossen Bildschirmen.

Weitere mögliche Erweiterungen

In Zukunft sind weitere Erweiterungen der Funktionalität denkbar:

- Wöchentliche/tägliche Challenges für die Erledigung von Ämtlis.
- Anpassen des User-Profils mit weiteren Avatar-Bildern.
- Möglichkeit bei einer Ablehnung durch einen Elternteil, beim anderen Elternteil eine Erlaubnis für eine Belohnung einzuholen.
- Vorschlag für die Erledigung eines nächsten Ämtlis.
- Zeitlicher Verlauf der Familienrangliste.
- Anpassbare Skins/Themes für das User Interface.
- Offline Fähigkeit der Applikation und Browser Notifikationen mit Service Worker