Abschlussarbeit am LG Kooperative Systeme der FernUniversität in Hagen

YReduxSocket: Ein Werkzeug zur Synchronisation und Konsistenz in Webanwendungen

Ricardo Stolzlechner

9463470

ricardo.stolzlechner@gmail.com

Informatik Bachelor of Science

Betreuerin: Dr. Lihong Ma

14. Januar 2024

Abstract

Inhaltsverzeichnis

Abbildungsverzeichnis

Algorithmenverzeichnis

Kapitel 1

Einleitung

1.1 Hintergrund und Motivation

Die Webanwendung „Yoshie.io“[1] ist eine kollaborative Plattform, spezialisiert auf das Baugewerbe. Der Autor dieser Arbeit fungiert hier als technischer Leiter. Im Baugewerbe operieren die verschiedenen Gewerke häufig in isolierten Datensilos. Baupläne und ähnliche Dokumente liegen bei den Beteiligten in unterschiedlichen Versionen vor, was während der Bauphase oft zu Fehlern und damit verbundenen Mehrkosten führt. „Yoshie.io“ bietet eine Lösung, indem es eine Plattform zur Verfügung stellt, auf der sämtliche für ein Bauvorhaben erforderlichen Dokumente zentralisiert, versioniert und zugänglich gemacht werden.

Während der Entwicklung dieser Anwendung wurde offensichtlich, dass eine der zentralen Herausforderungen darin besteht, die Konsistenz der dargestellten Daten zu gewährleisten. Diese Anforderung lässt sich in zwei Hauptbereiche unterteilen, die unterschiedliche technologische Ansätze erfordern. Einerseits muss die Datenkonsistenz innerhalb der verschiedenen Komponenten desselben Clients sichergestellt werden. Andererseits ist es notwendig, Datenänderungen zwischen verschiedenen Clients zu synchronisieren. Basierend auf Tanenbaum und van Steen musste ein clientbasiertes Konsistenzmodell implementiert werden, welches die Eigenschaften *monotones Lesen*, *monotones Schreiben* sowie eine *Writes Follow Reads* Konsistenz aufweist [TS08, S. 322–325]. In diesem Zusammenhang wurden ein auf Redux basierender Datenstore und das WebSocket-Kommunikationsprotokoll als technologische Lösungen eingesetzt.

Der Aufwand für die Implementierung und Wartung der Anwendung im Bereich der Konsistenzerhaltung stieg zunehmend. Daher wurde ein Konzept entwickelt, das beide Technologien kombiniert, um so den Entwicklungsaufwand erheblich zu reduzieren. Dieses Konzept erhielt intern den Namen *YReduxSocket*. Im Rahmen dieser Arbeit soll das genannte Konzept detailliert betrachtet und spezifiziert werden.

[1] `https://www.yoshie.io/`

1.2 Problemstellung

Wie bereits in Abschnitt 1.1 erwähnt, ergeben sich im Kontext der Konsistenzerhaltung in Webanwendungen zwei unterschiedliche Problemfelder. Diese werden in den folgenden Abschnitten näher erläutert.

1.2.1 Datenkonsistenz am selben Client

JavaScript-Frameworks wie *React* oder *Angular* basieren auf einer komponentenorientierten Architektur. Hierbei stellt jede Komponente ein User-Interface-Element dar, das an verschiedenen Stellen innerhalb der Anwendung eingesetzt werden kann. Diese Komponenten können weitere Unterkomponenten umfassen, wodurch ein Komponentenbaum entsteht. Typischerweise gibt es in Webanwendungen zahlreiche verschiedene Komponenten, deren Daten im Baum ausgetauscht und synchronisiert werden müssen.

Häufig sollen verschiedene Komponenten identische Daten anzeigen. In solchen Fällen ist es die Aufgabe der Entwicklerin, sicherzustellen, dass die Daten zwischen diesen Komponenten synchronisiert bleiben. Aufgrund der Notwendigkeit einer losen Kopplung, bei der Unterkomponenten keine direkte Kenntnis von ihren Elternkomponenten haben, ist die Synchronisation dieser Daten keine triviale Aufgabe.

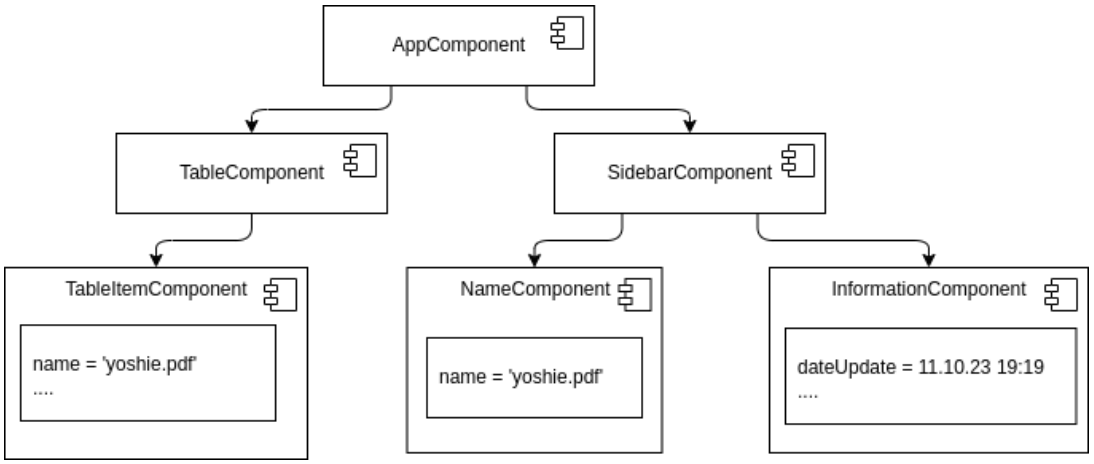

Abbildung 1.1: UML-Diagramm eines einfachen Komponentenbaums

In Abbildung 1.1 wird ein einfacher Komponentenbaum gezeigt. Der zugehörige Bildschirmausschnitt dieses Baums ist in Abbildung 1.2 zu sehen. Die Komponenten `TableItemComponent` und `NameComponent` zeigen jeweils dasselbe Datenfeld an. Wird der Name in der `NameComponent` geändert, muss der Entwickler dafür sorgen, dass diese Information auch an die `TableItemComponent` weitergegeben wird. Zudem muss das Aktualisierungsdatum in der `InformationComponent` aktualisiert werden.

Verschiedene Ansätze, um die notwendige Synchronisierung zu erreichen, werden in späteren Kapiteln detailliert betrachtet und miteinander verglichen.

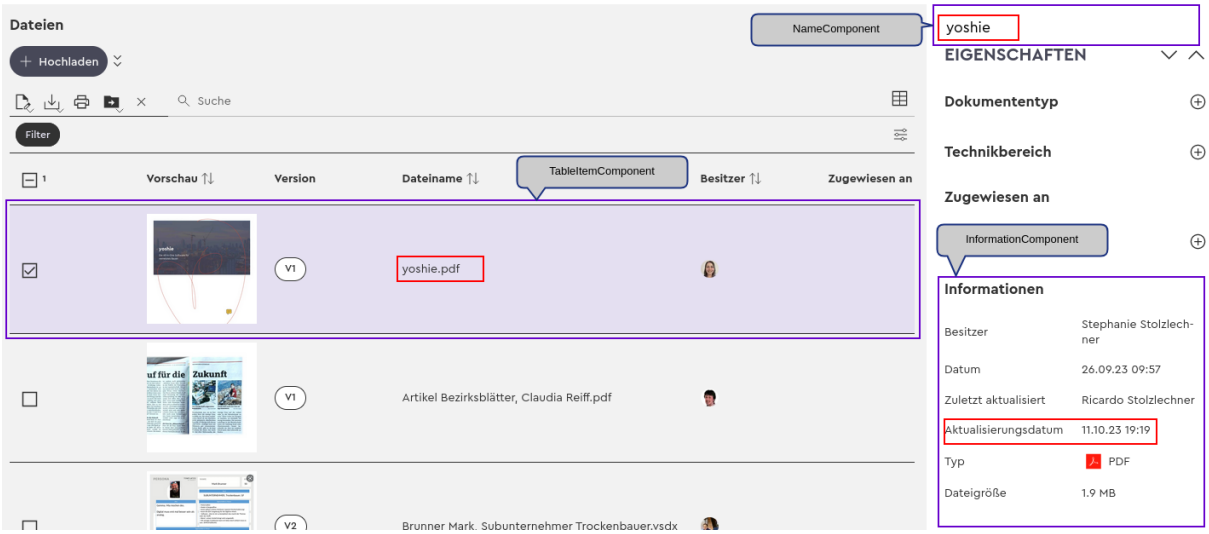

Abbildung 1.2: Bildschirmausschnitt eines einfachen Komponentenbaums

1.2.2 Datenkonsistenz zwischen verschiedenen Clients

In einer Webanwendung arbeiten verschiedene Clients oft mit denselben Daten. Wenn beispielsweise zwei Benutzerinnen gleichzeitig dieselbe Ansicht, wie in Abbildung 1.2 dargestellt, geladen haben, und eine von ihnen den Namen über die `NameComponent` ändert, muss gewährleistet sein, dass diese Änderung auch bei der anderen Benutzerin sichtbar wird. Daher ist es erforderlich, einen Mechanismus zu etablieren, der bei Datenänderungen alle betroffenen Teilnehmer synchron hält. Verschiedene Ansätze zur Lösung dieses Problems werden in späteren Abschnitten vorgestellt.

1.3 Ziele der Arbeit

Wie bereits erwähnt, existieren diverse Ansätze zur Lösung der beiden beschriebenen Probleme. Das Konzept *YReduxSocket* zielt darauf ab, beide Konsistenzanforderungen zu vereinheitlichen und gemeinsam zu adressieren. Hierbei wird auf jedem Client ein eigner auf Redux basierender Datenstore verwendet. Diese Datenstores werden anschließend mithilfe des WebSocket-Kommunikationsprotokolls, welches von einer zentralen Serverinstanz gesteuert wird, synchronisiert. Ein besonderer Fokus bei der Entwicklung von *YReduxSocket* lag auf der Entwicklerfreundlichkeit. Das Konzept minimiert die bei Funktionserweiterungen notwendigen Änderungen bezüglich des Konsistenzverhaltens. Dies trägt zur Reduzierung von Entwicklungsfehlern bei und verbessert somit das Konsistenzverhalten von Webanwendungen.

Das primäre Ziel dieser Arbeit besteht darin, *YReduxSocket* und die zugrundeliegenden Technologien generisch zu spezifizieren. Ein besonderes Augenmerk liegt auf der detaillierten Untersuchung des Konsistenzverhaltens des Konzepts. Zusätzlich wird die vorgestellte Spezifikation implementiert, um deren praktische Anwendbarkeit zu testen.

1.4 Struktur der Arbeit

Zu Beginn werden in Kapitel 2 die grundlegenden Konzepte erörtert, die für das Verständnis der weiteren Arbeit essenziell sind. Dabei liegt der Fokus auf Single Page Applications, der komponentenbasierten Webentwicklung, dem zentralen Redux-basierten Datenstore sowie dem WebSocket-Kommunikationsprotokoll. Ferner wird definiert, welche Voraussetzungen für den Einsatz von *YReduxSocket* erfüllt sein müssen. Im abschließenden Teil dieses Kapitels wird ein Ansatz vorgestellt, wie ein Datenstore mittels WebSocket-Nachrichten synchronisiert werden kann.

Das anschließende Kapitel 3 ist dem Konzept *YReduxSocket* gewidmet. Hier wird das Konzept zunächst generisch vorgestellt und spezifiziert. Der Abschnitt 3.3 vertieft das Thema, indem das Konsistenzverhalten von *YReduxSocket* genauer beleuchtet wird. Das Kapitel schließt mit einem Vergleich von *YReduxSocket* mit dem in Abschnitt 2.5 vorgestelltem Ansatz.

In Kapitel 4 wird vorgestellt, wie *YReduxSocket* implementiert werden kann. Abschnitt 4.1 geht zunächst auf die gewählte Software-Architektur ein. Der Abschnitt 4.2 behandelt die clientseitige Implementierung, welche dem Framework Angular zugrunde liegt. Abschließend behandelt der Abschnitt 4.3 die Implementierung der Serverseite im Frameworek ASP.Net.

Die Arbeit wird mit Kapitel 5 abgerundet. In diesem Abschnitt werden die erzielten Ergebnisse zusammengefasst, die Lösung validiert und potenzielle Verbesserungen des Konzepts aufgezeigt.

Kapitel 2

Grundlagen

In diesem Kapitel werden grundlegende Konzepte vorgestellt, die eine solide Basis für das Verständnis des weiteren Inhalts dieser Arbeit bilden. Es werden außerdem jene Voraussetzungen definiert, die für den erfolgreichen Einsatz von *YReduxSocket* notwendig sind.

2.1 Single Page Applications (SPAs)

Single Page Applications (SPAs) zeichnen sich dadurch aus, dass die gesamte Anwendung im Webbrowser des Benutzers ausgeführt wird. Eine SPA wird von einem Webserver bereitgestellt, der eine `index.html`-Datei enthält. Diese wird ausgeliefert, sobald die entsprechende URL im Webbrowser aufgerufen wird. In der `index.html` befindet sich ein `script`-Tag, das auf den erforderlichen JavaScript-Code für die Ausführung der Anwendung verweist. Dieser Code wird vom Browser heruntergeladen und ausgeführt. Nach dem initialen Ladevorgang beschränkt sich die Rolle des Webservers darauf, statische Ressourcen wie Schriftarten oder Bilder nachzuladen. Die benötigten HTML-Elemente für die Darstellung werden ausschließlich über JavaScript generiert und aktualisiert. Für die Entwicklung von SPAs werden verschiedene etablierte Frameworks wie *Angular*, *React* und *Vue* verwendet, die die DOM-API des Webbrowsers abstrahieren und somit die Entwicklung vereinfachen. Dies steht im Gegensatz zu klassischen Webanwendungen, bei denen der HTML-Code serverseitig generiert und ausgeliefert wird, bekannt als serverseitiges Rendering [HZ20, S. 1].

YReduxSocket setzt den Einsatz einer Single Page Application voraus. Zusätzlich wird eine separate Serverinstanz benötigt, die von den Clients zum Abrufen und Bearbeiten von Datensätzen verwendet wird. Diese Serverinstanz bietet eine HTTP-API und ermöglicht die Einrichtung einer WebSocket-Kommunikation. Jegliche Kommunikation zwischen den Clients läuft über diese Serverinstanz, die als eigenständiger Prozess zu verstehen ist und nicht mit dem Webserver, der die SPA bereitstellt, verwechselt werden sollte. Im weiteren Verlauf der Arbeit bezieht sich der Begriff „Server“ stets auf diese separate Serverinstanz. Die in Abbildung 2.1 dargestellte Struktur illustriert die von *YReduxSocket* benötigte Prozessarchitektur, wobei jedes Rechteck im Diagramm einen unabhängigen Prozess repräsentiert.

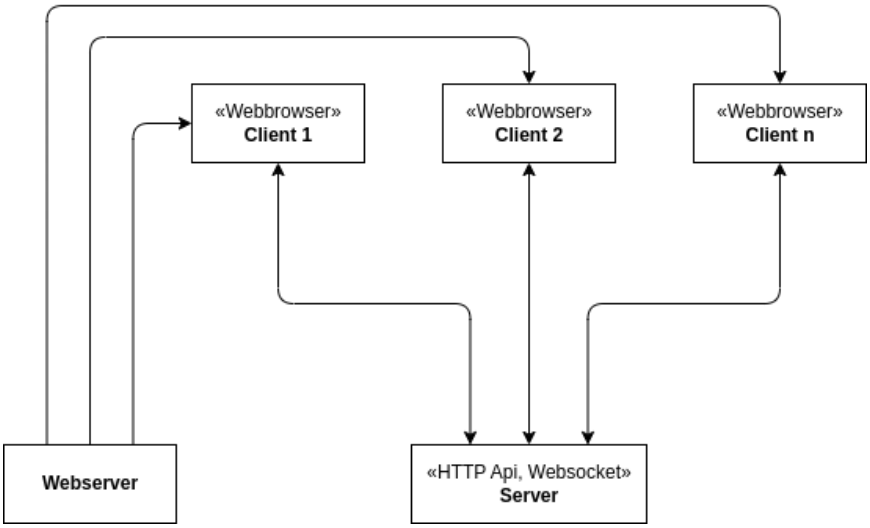

Abbildung 2.1: Von *YReduxSocket* vorausgesetzte Prozessarchitektur

2.2 Komponentenbasierte Webentwicklung

Wie im Abschnitt 2.1 erwähnt, erleichtern Frameworks den Aufbau von Single Page Applications (SPAs), indem sie auf einer komponentenbasierten Architektur aufbauen. Eine Komponente gliedert sich grundsätzlich in zwei Teile: Der erste Teil, in einer Art HTML definiert, beschreibt die Darstellung der Komponente im Browser. Der zweite Teil, mit JavaScript oder TypeScript (einer typisierten Variante von JavaScript) formuliert, definiert die logische Funktionalität der Komponente, einschließlich Methoden für Nutzerinteraktionen.

Jede Komponente kann im HTML-Teil weitere Unterkomponenten beinhalten, wodurch ein Komponentenbaum gebildet wird. Die Komponenten stehen so in einer Eltern-Kind-Beziehung zueinander.

In Webanwendungen müssen viele Komponenten miteinander kommunizieren. Dafür stellen Frameworks verschiedene Datenbindungsmechanismen zur Verfügung. Eine Eltern-Komponente kann Daten über einen Eingabeparameter an eine Kind-Komponente übergeben. Kind-Komponenten können Ausgabeparameter bereitstellen, über die sie die Eltern-Komponenten bezüglich Datenänderungen informieren. Wenn Kind-Komponenten diese Ausgabeparameter aktualisieren, wird die Eltern-Komponente automatisch benachrichtigt, sofern sie sich auf diese registriert hat.

Abbildung 2.2 demonstriert den Datenfluss im Komponentenbaum bei Nutzung der Datenbindungsmechanismen. Ändert sich beispielsweise der Name in der `NameComponent`, löst diese Komponente einen Ausgabeparameter aus, für den sich die übergeordnete `SidebarComponent` registriert hat. Diese informiert dann die `AppComponent` und aktualisiert einen Eingabeparameter für die `InformationComponent`, um das geänderte Aktualisierungsdatum zu melden. Die Hauptkomponente aktualisiert daraufhin ihren Eingabeparameter für die `TableComponent`, die wiederum eine `TableItemComponent` benachrichtigt. Letztgenannte kann dann ihren angezeigten Wert aktualisieren.

Bei größeren Komponentenbäumen oder wenn mehrere Komponenten denselben Datensatz anzeigen sollen, stößt dieser Ansatz jedoch an seine Grenzen. Hier bietet sich die

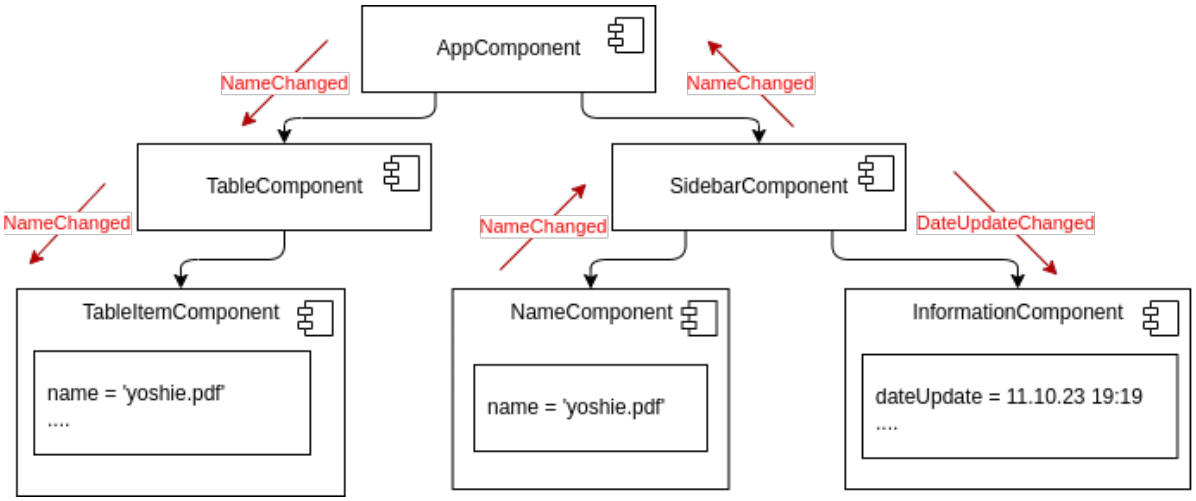

Abbildung 2.2: Datenbindungsmechanismen: Datenfluss im Komponentenbaum

Verwendung von Services an. Ein Service ist eine global verfügbare JavaScript-Klasse, die einmalig im Programm initialisiert wird. Die verschiedenen Frameworks bieten Mechanismen, um solche Services zu erstellen und global verfügbar zu machen.

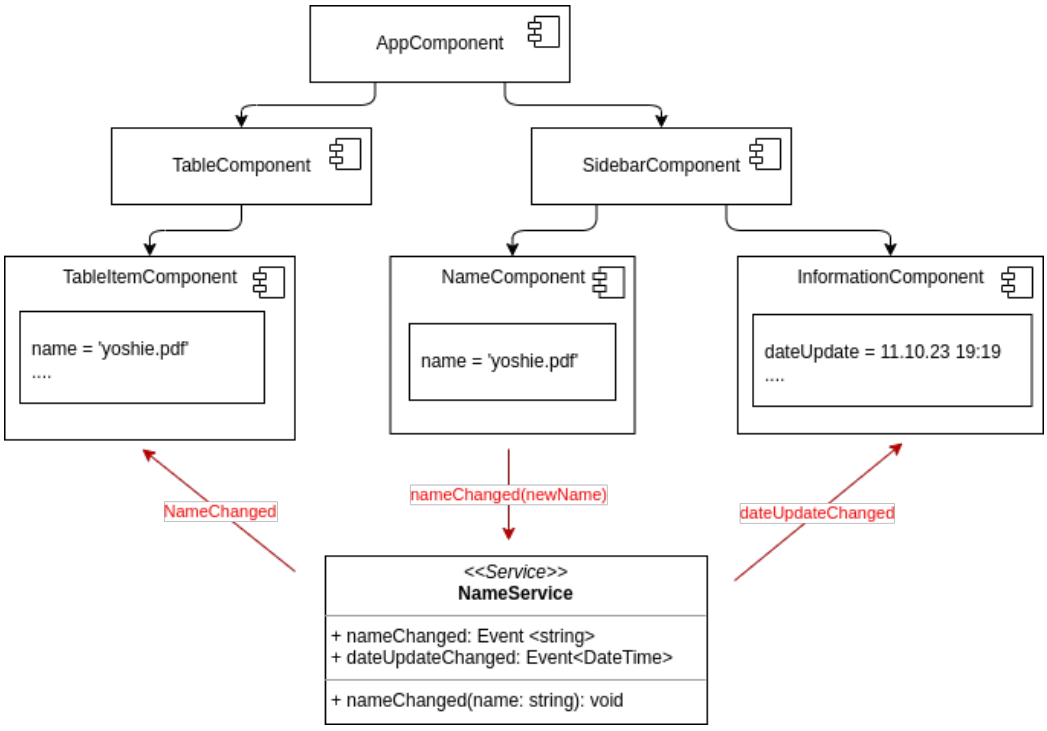

Abbildung 2.3: globaler Service: Datenfluss im Komponentenbaum

Im Beispiel in Abbildung 2.3 wird ein `NameService` genutzt. Dieser Service bietet eine Methode `nameChanged`, die von der `NameComponent` bei Änderungen aufgerufen wird. Die Methode löst dann die Events `nameChanged` und `dateUpdateChanged` aus. Haben sich die `TableItemComponent` und die `InformationComponent` auf diese Events registriert, werden sie entsprechend benachrichtigt und können ihre angezeigten Werte aktualisieren.

Bei der Entwicklung mit Services bleibt jedoch das Problem der doppelten Datenhaltung bestehen. Im besprochenen Beispiel existiert die Zeichenkette `name` sowohl in der

`TableItemComponent` als auch in der `NameComponent`. Dieses Problem lässt sich durch die Verwendung eines zentralen Datenstores auf Basis von Redux umgehen. Dieses Konzept wird im nächsten Abschnitt näher erläutert.

2.3 Zentraler Redux-basierter Datenstore

Mit der Zunahme an Komponenten in einer Anwendung steigt auch die Komplexität des zugrundeliegenden Codes. Diese Komplexitätssteigerung resultiert aus der Notwendigkeit, Daten zwischen den Komponenten auszutauschen, um sie in einem konsistenten Zustand zu halten. Zur Reduzierung dieser Komplexität eignet sich der Einsatz eines zentralen, auf Redux basierenden Stores (siehe Abschnitt 2.2).

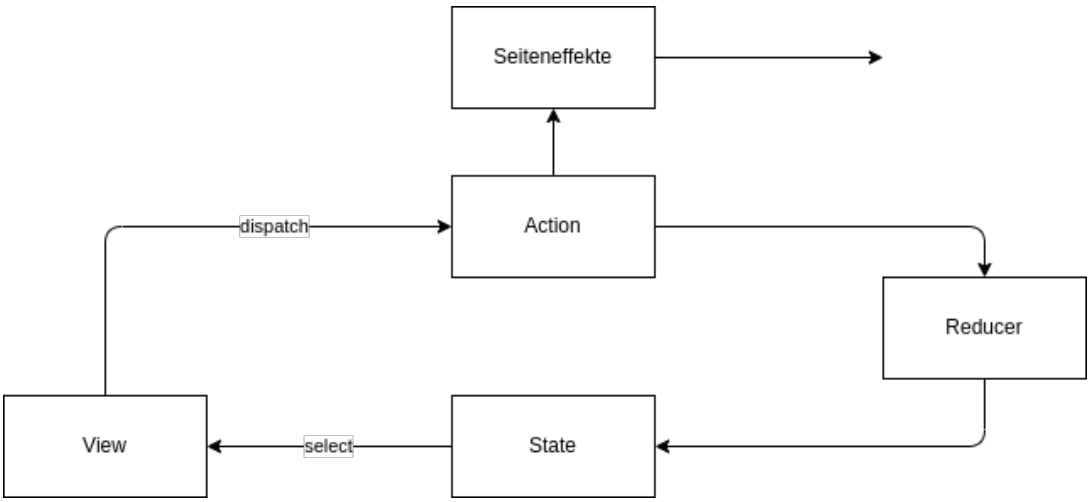

Abbildung 2.4: Von Redux vorgeschriebener Datenfluss

Redux, entwickelt von Dan Abramov, ist eine Implementierung des von Facebook definierten Designpatterns Flux. Dieses Pattern sieht einen unidirektionalen Datenfluss vor, wie in Abbildung 2.4 dargestellt. Der Store verwaltet den gesamten Anwendungszustand bezüglich der darzustellenden Daten. Er bietet Vorteile gegenüber den im vorherigen Abschnitt beschriebenen Konzepten, indem er Datenstrukturen und die gesamte Logik zur Zustandsaktualisierung an einer definierten Stelle konzentriert – bekannt als Single Source of Truth. Zusätzlich verbessert ein zentraler Store nicht nur die Performance, sondern erleichtert auch die Testbarkeit [Far17, S. 31–32].

Für eine detaillierte Beschreibung eines Redux-basierten Stores wird auf das bereits verwendete Beispiel zurückgegriffen (siehe auch Abschnitt 2.2). Dabei wird die Implementierung NgRx [1], welche im Framework Angular eingesetzt werden kann, verwendet.

In Abbildung 2.5 wird illustriert, wie die Komponenten mit dem Store interagieren können, um Daten abzurufen oder Datenänderungen zu veranlassen. Der Store, der wie ein Service global zugänglich ist, bietet zwei Methoden an: Die Methode `select`, der ein `Selektor` übergeben wird, steuert, welcher Teil des States zurückgegeben wird. Die Methode gibt kein festes Ergebnis, sondern ein `Observable` zurück, eine Art Stream. Die Komponente kann sich bei diesem `Observable` anmelden und wird bei jeder Datenänderung

[1] `https://ngrx.io/`

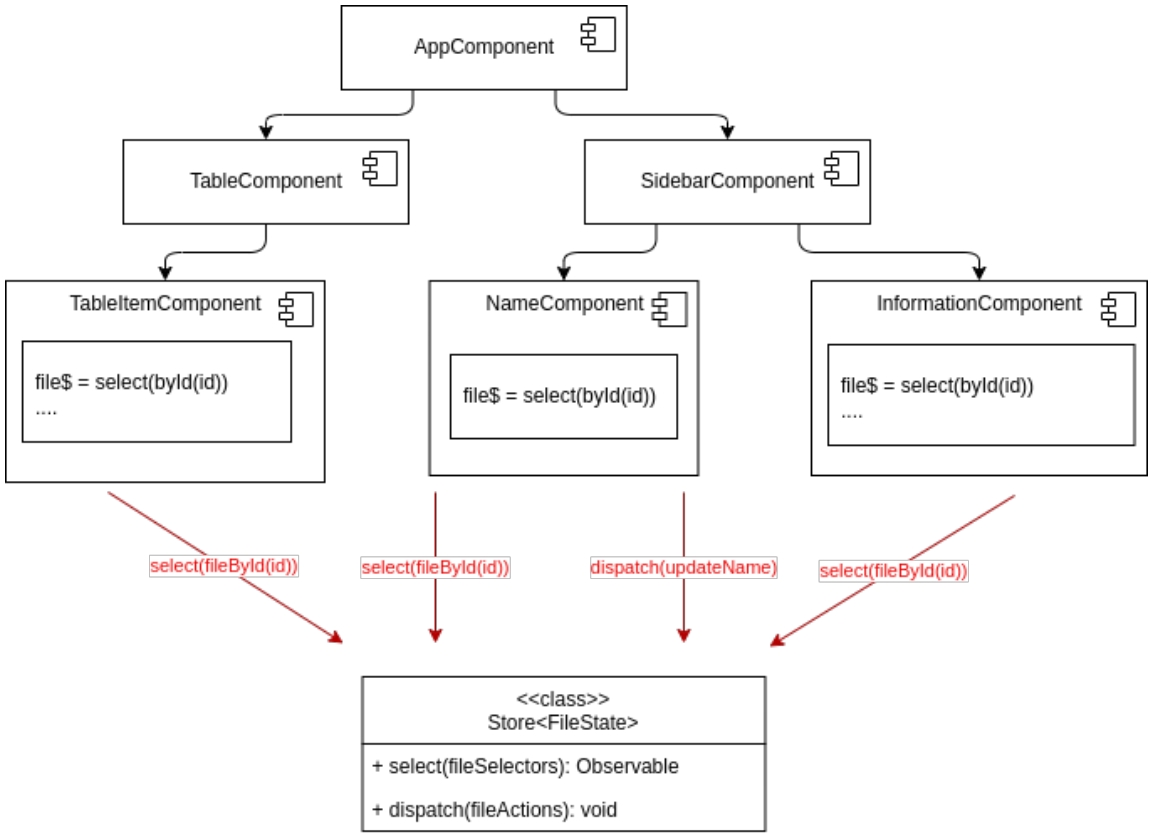

Abbildung 2.5: Redux-basierter Store: Datenfluss im Komponentenbaum

informiert. Um eine Datenänderung zu veranlassen, wird die Methode `dispatch` verwendet. Dieser wird eine `Action` übergeben, die definiert, wie die Änderung aussehen soll. Die `dispatch` Methode gibt keinen Wert zurück, betrifft die gewünschte Änderung jedoch einen Teil des Stores, auf den sich eine Komponente vorab über den Selektor angemeldet hat, wird im gelieferten `Observable` automatisch ein neuer Wert emittiert.

Im Folgenden werden die Kernkonzepte eines Redux-basierten Stores – der State, die Selektoren, die Actions, der Reducer und die Seiteneffekte – detailliert erläutert.

2.3.1 State

Der State wird in Form eines einfachen JavaScript-Objekts implementiert und zu Beginn der Anwendung mit vorgegebenen Startwerten initialisiert. Es ist wichtig zu betonen, dass der State ausschließlich lesbar ist; Änderungen am State sind ausschließlich über definierte Reducer-Funktionen möglich, die mittels der `dispatch`-Funktion aufgerufen werden. Für die Definition des JavaScript-Objekts wird in NgRx ein Interface erstellt, in dem alle benötigten Felder definiert sind. Weiterhin muss eine Konstante definiert werden, die dem Interface entspricht und den initialen Zustand des State festlegt.

Algorithmus 2.1 zeigt die Statedefinition für das Beispiel, wie in Abbildung 2.5 dargestellt. Die beiden Flags `loaded` und `loading` geben an, ob die Daten bereits geladen wurden bzw. ob der Ladevorgang bereits gestartet wurde. Die tatsächlichen Daten sind in einem JavaScript-Objekt in Form eines Dictionaries oder einer HashMap abgelegt, wobei

Algorithmus 2.1 Statedefinition in NgRx

```
export interface FileState {
   entities: {[id: string]: FileObject};
   loaded: boolean;
   loading: boolean;
}

export const initialFileState : FileState = {
    entities: {},
    loaded: false,
    loading: false
}
```

die `id` des `FileObject` dem Schlüssel und das `FileObject` selbst dem Wert entspricht. Der Vorteil dieses Ansatzes ist, dass bei bekannter `id` des `FileObject` in $O(1)$ auf das Objekt zugegriffen werden kann. Würde man stattdessen ein Array verwenden, müsste bei bekannter `id` über das gesamte Array iteriert werden, was $O(n)$ Zeit in Anspruch nehmen würde. [Bae19, S. 54]

2.3.2 Selektoren

Selektoren ermöglichen es Komponenten, auf bestimmte Bereiche des States zuzugreifen. NgRx bietet hierfür die Methoden `createFeatureSelector` und `createSelector` an. Die erstere Methode erzeugt einen Selektor, der den gesamten State umfasst. Dafür muss der Methode ein String übergeben werden, der mit dem State-Namen übereinstimmt, der in der Reducer-Definition (siehe 2.3.4) vergeben wird. Die zweite Methode, `createSelector`, dient dazu, einen Teil des States zu extrahieren, um so kleinere Teile nach außen zu geben.

Algorithmus 2.2 Selektorendefinition in NgRx

```
const state = createFeatureSelector<FileState>('files');
const loaded = createSelector(state, (state) => state.loaded);
const loading = createSelector(state, (state) => state.loading);
const entities = createSelector(state, (state) => state.entities);
//entities as array
const files = createSelector(entities, (entities) =>
    Object.keys(entities).map((id) => entities[id]));
const fileById = (id: string) =>
    createSelector(entities, (entities) => entities[id]);

export const fileSelectors = {
    loaded, loading, files, fileById
};
```

Algorithmus 2.2 zeigt die Selektorendefinition, um auf die für das Beispiel benötigten Teile des States zuzugreifen. Auf die eigentliche Generierung der bereits erwähnten

`Observables` wird in Abschnitt 2.3.6 näher eingegangen.

2.3.3 Actions

Actions in NgRx bestehen aus einer eindeutigen ID und einem optionalen Payload. Zum Erstellen von Actions bietet NgRx die Methode `createActionGroup()` an, die es ermöglicht, spezifische Actions zu definieren. Die ID einer Action wird aus dem Property-Namen des Events und dem String, der als `source` angegeben wird, generiert. Die Methoden `emptyProps()` und `props()` werden verwendet, um den Payload der Action zu spezifizieren. Algorithmus 2.3 zeigt die für das Beispiel notwendigen Action-Definitionen. Die definierten Actions können entweder von außen (siehe Abschnitt 2.3.6) oder durch einen Seiteneffekt (siehe Abschnitt 2.3.5) dispatched werden. Dieses Dispatchen führt in der Regel dazu, dass eine Reducer-Funktion aufgerufen wird, die einen neuen State liefert.

Algorithmus 2.3 Actiondefinitionen in NgRx

```
export const fileActions = createActionGroup({
    source: 'files',
    events: {
        load: emptyprops(),
        loadSuccess: props<{files: FileObject[]}>,
        updateName: props<{id: string, name: string}>,
        updateNameSuccess: props<{id: string, name: string}>
    }
})
```

2.3.4 Reducer

Über den Reducer wird festgelegt, wie der Store modifiziert werden soll, wenn eine Action ausgeführt (dispatched) wird. Dabei wird für jede Action, die eine Datenänderung bewirken soll, eine sogenannte reine Funktion (pure function) definiert. Dies sind Funktionen, die bei gleichen Eingabeparametern stets dasselbe Ergebnis liefern. Die Parameterwerte der Funktion umfassen den aktuellen State und die ausgeführte Action, wobei der State unverändert bleiben muss (Immutability) [McF19, S. 21].

Algorithmus 2.4 zeigt die Reducerdefinition in NgRx für das gewohnte Beispiel. Dabei wird ein sogenanntes Feature definiert, dessen `name` dem String entsprechen muss, der beim Erzeugen des Hauptselektors (der den gesamten State umfasst) verwendet wurde. Die Reducer-Funktionen werden über die Methode `createReducer` erzeugt. Der erste Parameter muss dabei dem initialen FileState entsprechen (siehe Abschnitt 2.3.1). Danach wird mittels der Methode `on` bestimmt, welche Funktion bei welcher Action aufgerufen wird. Wichtig dabei ist zu beachten, dass die Eingabewerte nicht modifiziert werden dürfen, das heißt, es muss, wie in der Methode `onUpdateNameSuccess` ersichtlich, ein neues State-Objekt retourniert werden.

Algorithmus 2.4 Reducerdefinitionen in NgRx

```
const onLoad = (state, action) =>
    ({ loading: true, loaded: false, entities: {} });
const onLoadSuccess = (state, action) => {
    const entities = arrayToEntities(action.files);
    return { loading: false, loaded: true, entities };
}
const onUpdateNameSuccess = (state, action) => {
    return { ...state, entities: { ...state.entities,
                [action.id]: {
                    ...state.entities[action.id],
                    name: action.name }
            }};
}

export const fileFeature = createFeature({
    name: 'files',
    reducer: createReducer(
        initialFileState,
        on(fileActions.load,onLoad),
        on(fileActions.loadSuccess, onLoadSuccess),
        on(fileActions.updateNameSuccess, onUpdateNameSuccess)
    )})
```

2.3.5 Seiteneffekte

Seiteneffekte werden in einem Redux-basierten Store für die Verwaltung von asynchronen Operationen wie API-Aufrufen, Datenpersistenz oder komplexeren Geschäftslogiken verwendet. Sie ermöglichen es, auf bestimmte Actions zu reagieren, ohne den Store direkt zu manipulieren. Diese Operationsweise ist besonders nützlich, wenn externe Prozesse oder asynchrone Aktivitäten involviert sind. In NgRx werden Seiteneffekte mit der `createEffect`-Funktion definiert, die es ermöglicht, auf spezifische Actions zu reagieren und daraus resultierende Aktionen zu steuern. Im Normalfall wird nach Abschluß der durch die eingehende Action ausgelöste Operation eine ausgehende Action retourniert.

Im Beispiel, das in Algorithmus 2.5 dargestellt ist, wird auf die Actions `load` und `updateName` reagiert. Die Klasse `FileEffects` wird mit einem Stream aller eintreffenden Actions (`actions`) initialisiert. Dieser Stream wird verwendet, um auf eingehende Actions zu reagieren. Beispielsweise, wenn die Action `updateName` eintrifft, löst dies einen API-Aufruf über den `FileHttpService` aus. Bei erfolgreicher Ausführung des Serveraufrufs wird eine weitere Action, in diesem Fall `updateNameSuccess`, dispatched.

2.3.6 Fassade

Um die Nutzung des Stores für Komponenten zu vereinfachen und eine lose Kopplung zwischen den Komponenten und den Stores zu erreichen, empfiehlt es sich, eine Fassade

Algorithmus 2.5 Seiteneffektsdefinitionen in NgRx

```
@Injectable()
export class FileEffects {
    constructor(actions: Actions, http: FileHttpService) {}

    load$ = createEffect(() =>
        this.actions.pipe(
            ofType(fileActions.load),
            switchMap(() =>
                this.http.load().map((files) =>
                    fileActions.loadSuccess(files))
        ));

    updateName$ = createEffect(() =>
        this.actions.pipe(
            ofType(fileActions.load),
            switchMap(({id, name}) =>
                this.http.updateName(id, name).map(() =>
                    fileActions.updateNameSuccess(id, name))
        ));
}
```

zwischen diesen einzuziehen. Eine Fassade fungiert als Abstraktionsschicht, die den Store, die Selektoren und die Actions verbirgt. Dadurch werden öffentliche Methoden nach außen angeboten, sodass die Komponenten keine direkte Kenntnis vom eigentlichen Store haben müssen [Fre+21, S. 266]. Algorithmus 2.6 zeigt eine Implementierung einer solchen Fassadenklasse, wie sie für das Beispiel benötigt wird.

Abbildung 2.6 fasst die Abhängigkeiten zwischen den einzelnen Store-Komponenten zusammen und zeigt, wie durch die Fassade eine lose Kopplung zwischen Store und Komponente erreicht wird.

2.4 WebSocket-Kommunikation

Wie bereits in Abschnitt 1.2.1 erwähnt, besteht in Webanwendungen, insbesondere wenn sie kollaborativ genutzt werden, das Problem, dass bei Datenänderungen alle Clients, die diese Daten geladen haben, aktualisiert werden müssen. Die standardmäßige HTTP-Kommunikation zwischen einem Webclient und dem Server ist zustandslos und kann nur vom Client initiiert werden. Aus der Zustandslosigkeit des HTTP-Protokolls folgt, dass der Server keine Informationen darüber hat, welche Clients gerade verbunden sind und welche Daten diese geladen haben. Um dieses Problem zu lösen, gibt es verschiedene Ansätze wie Long Polling oder WebSockets [Ack21, S. 182]. Da *YReduxSocket* von einer WebSocket-Verbindung ausgeht, wird im Folgenden ausschließlich auf dieses Protokoll eingegangen.

Das WebSocket-Protokoll baut auf TCP auf und ermöglicht bidirektionale Verbindungen

Algorithmus 2.6 Fassade: Beispielimplementierung für den FileStore

```
export class FileFacade {
    constructor(private store: Store<FileState>) {}

    async load() : Promise<boolean> {
        const loaded = await this.store.select(
            fileSelectors.loaded).toPromise();
        if(loaded) return true; //bereits geladen

        store.dispatch(fileActions.load()); //ladevorgang starten
        return await this.store.select(
            fileSelectors.loading).pipe(filter(x=>!x))
            .toPromise(); //warten bis Ladevorgang abgeschlossen
    }

    selectAll(): Observable<FileObject[]> =>
        this.store.select(fileSelectors.files());

    selectById(id: string): Observable<FileObject> =>
        this.store.select(fileSelectors.fileById(id));

    updateName(id: string, name: string): void =>
        this.store.dispatch(fileActions.updateName({id, name}));
}
```

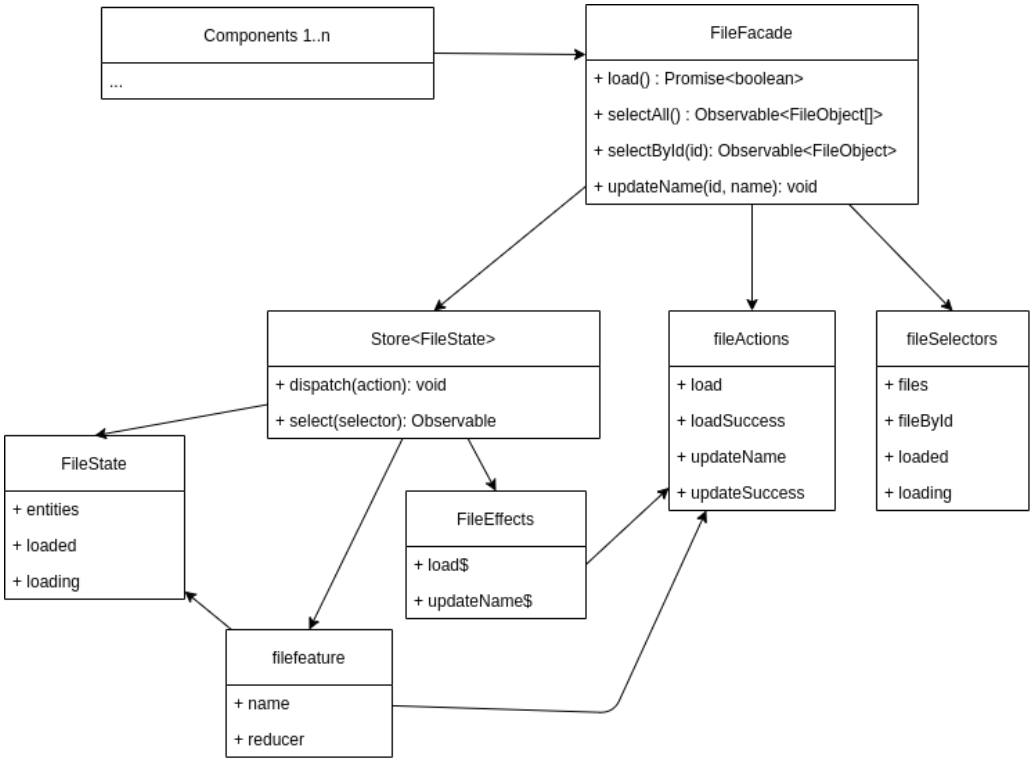

Abbildung 2.6: FileStore Abhängigkeiten

zwischen Clients und einem Server. Der initiale Verbindungsaufbau muss vom Client ausgehen, doch sobald die Verbindung besteht, hat auch der Server die Möglichkeit, Daten an den Client zu senden [Ack21, S. 184–185]. Es kann erforderlich sein, dass sich der Client für relevante Informationen anmeldet, um zu vermeiden, dass der Server bei jeder Datenänderung einen Broadcast (Nachricht an alle verbundenen Clients) senden muss. In Abbildung 2.7 werden zwei Clients dargestellt, die über das WebSocket-Protokoll verbunden sind. Client A führt eine Aktion aus, die Daten ändert. Nach der erfolgreichen Änderung können beide Clients vom Server aus aktualisiert werden.

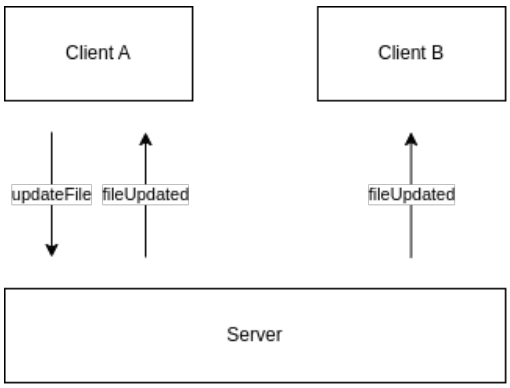

Abbildung 2.7: einfache WebSocket Kommunikation

2.5 Datenstore mit WebSocket: herkömmlicher Ansatz

In diesem Abschnitt wird die Kombination der in 2.3 und 2.4 beschriebenen Technologien betrachtet, insbesondere im Kontext von Datenänderungen. Der Prozess einer Datenänderung beginnt damit, dass der Client, der eine Änderung vornehmen möchte, eine entsprechende Action dispatched. Dies löst einen Seiteneffekt aus, der wiederum einen Serveraufruf nach sich zieht. Auf dem Server wird der Änderungswunsch validiert und bei Erfolg umgesetzt. Bei erfolgreicher Änderung sendet der Server einen Statuscode zurück an den Client, der den Erfolg der Änderung bestätigt. Zudem ist es erforderlich, anderen Clients, die sich für Benachrichtigungen über Änderungen registriert haben, diese über eine WebSocket-Nachricht mitzuteilen. Die Clients reagieren nun auf den erhaltenen Statuscode oder die WebSocket-Benachrichtigung, indem sie eine Success-Action dispatchen. Die Success-Action aktiviert dann den Reducer, der wiederum den Store aktualisiert.

Abbildung 2.8 illustriert diesen Ablauf anhand eines Interaktionsdiagramms. Es wird angenommen, dass Client A den Aktualisierungsvorgang initiiert, während Client B sich im Voraus für Aktualisierungsbenachrichtigungen angemeldet hat. Für die Implementierung jedes neuen Aktualisierungsvorgangs, der möglicherweise durch ein neues Feature entsteht, sind folgende Schritte erforderlich:

1. Definition einer Action, die den Seiteneffekt auslöst.

2. Implementierung eines neuen HTTP-Endpunkts, sowohl server- als auch clientseitig.

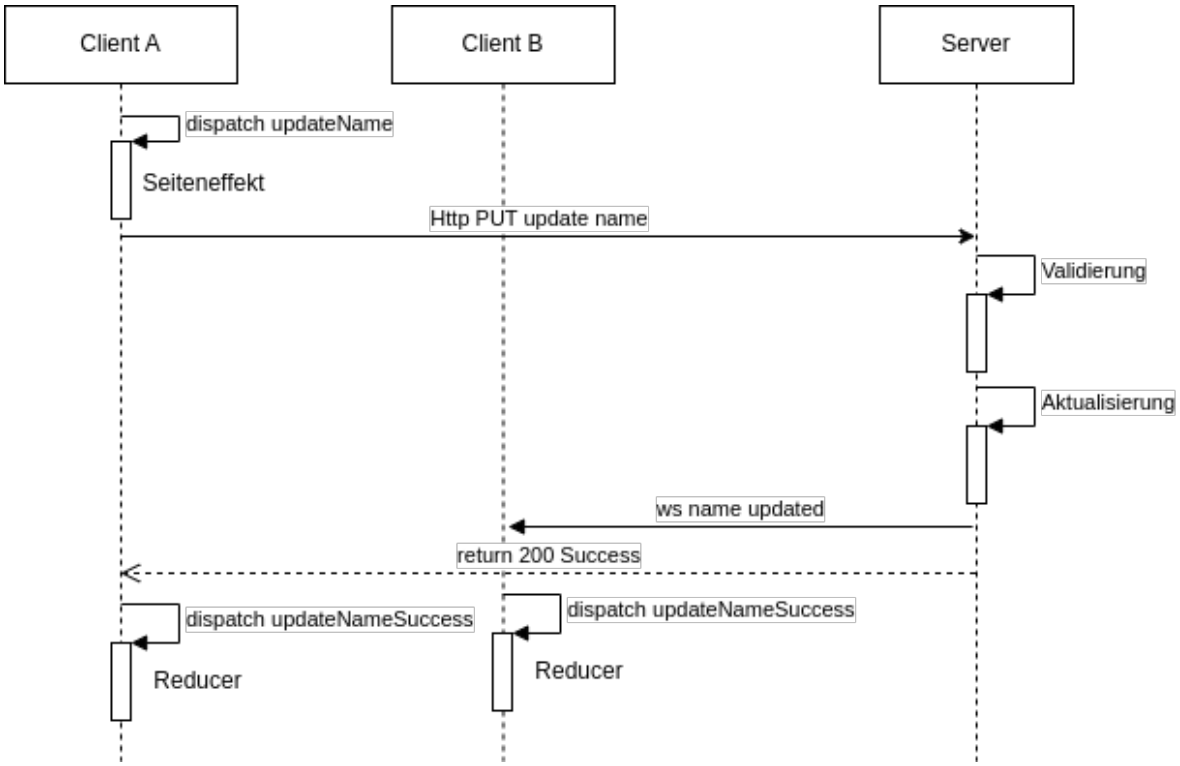

Abbildung 2.8: Interaktionsdiagramm: Datenstore mit WebSocket, herkömmlicher Ansatz

3. Implementierung des Seiteneffekts.
4. Implementierung der serverseitigen Validierungs- und Aktualisierungslogik.
5. Implementierung eines neuen WebSocket-Endpunkts, ebenfalls server- und clientseitig.
6. Definition einer Action, die die Reducer-Funktion auslöst.
7. Implementierung der Reducer-Funktion zur Anpassung der Client-Daten.

Die Notwendigkeit, diese Schritte für jede neue Funktionalität zu wiederholen, macht deutlich, dass dieser Ansatz langfristig sehr aufwendig ist. Um diese Herausforderungen zu bewältigen und den Implementierungsaufwand zu reduzieren, wurde *YReduxSocket* entwickelt, das im nächsten Kapitel detailliert beschrieben wird.

Kapitel 3

YReduxSocket

In diesem Kapitel widmen wir uns ausführlich dem Konzept des *YReduxSocket*. Zunächst beginnen wir mit einer kompakten Einführung, in der wir das Konzept von *YReduxSocket* kurz und prägnant umreißen. Anschließend entwickeln wir eine detaillierte generische Konzeption und Spezifikation, die sowohl einen Überblick als auch tiefergehende Einblicke in die Funktionsweise und Besonderheiten von *YReduxSocket* bietet. Danach wird ein besonderer Fokus auf das Konsistenzverhalten von *YReduxSocket* gelegt. Zum Abschluss des Kapitels erfolgt ein Vergleich zwischen dem *YReduxSocket*-Ansatz und dem herkömmlichen Ansatz, wie er im Abschnitt 2.5 beschrieben wird. Dieser Vergleich soll die Vorzüge und möglichen Verbesserungen durch *YReduxSocket* im Kontext bestehender Methoden hervorheben.

3.1 Kurzbeschreibung

Die Kernidee von YReduxSocket zielt darauf ab, den in Abschnitt 2.5 dargestellten traditionellen Ansatz zu vereinfachen. Dies wird erreicht, indem Actions sowohl auf Server- als auch auf Clientseite definiert werden. Anstelle der Übermittlung der Action an den Store wird sie direkt an den Server gesendet. Der Server bedient sich eines einzigen HTTP-Endpunkts zum Empfangen der Action. Bei diesem Vorgang wird abhängig von der spezifischen Kennzeichnung der Action entschieden, welche Validierungen oder Datenbankmodifikationen erforderlich sind. Nach Abschluss der Operationen sendet der Server eine Action per WebSocket zurück an den Client. Diese Rückübermittlung erfolgt über eine bereits festgelegte Methode auf der Clientseite. Der Client muss in dieser Methode sicherstellen, dass die Action an den Store weitergeleitet wird, woraufhin der Reducer aktiv wird und entsprechende Anpassungen im Store vornimmt. Ein weiterer Vorteil dieses Ansatzes ist, dass bei Bedarf neuer Endpunkte auf zusätzliche Seiteneffekte, die die Server-Client-Kommunikation betreffen, verzichtet werden kann. Um die Server-Client-Kommunikation mit YReduxSocket zu erweitern, sind folgende Schritte von einer Entwicklerin durchzuführen:

1. Definition von zwei Actions auf der Client- und der Serverseite.

2. Versenden einer Action vom Client an den Server über den bestehenden Endpunkt.

3. Implementierung der Validierungs- und Anpassungslogik auf der Serverseite.
4. Auslösen der serverseitigen Antwort über den bestehenden Client-Endpunkt unter Übergabe der zweiten Action.
5. Implementierung der Reducer-Funktion auf der Clientseite zur Anpassung der Client-Daten.

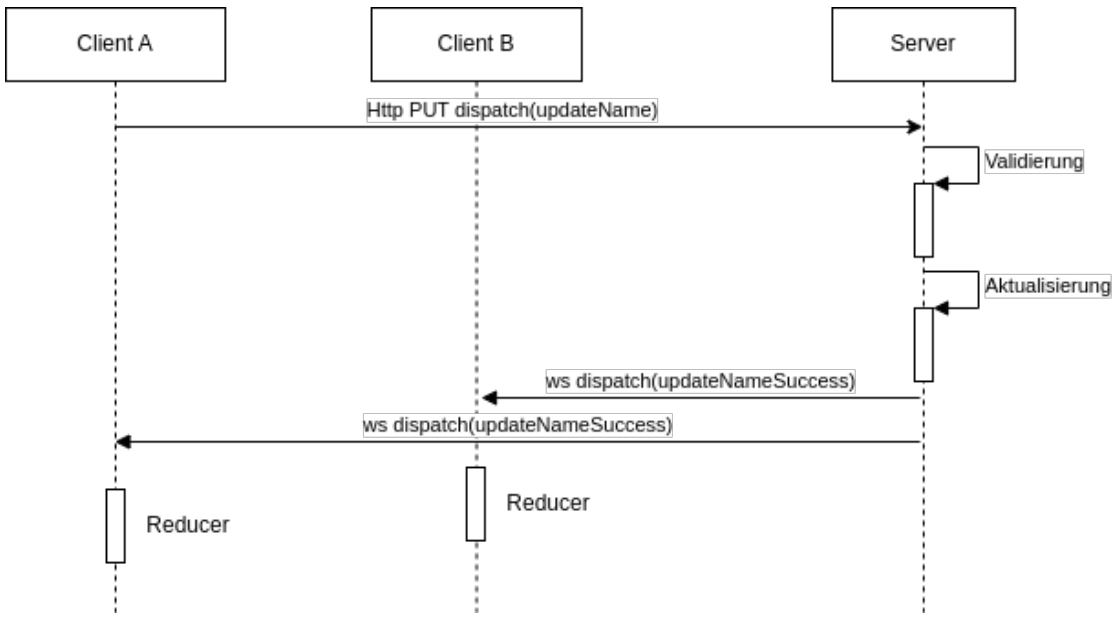

Abbildung 3.1: Interaktionsdiagramm: Datenstore mit WebSocket, YReduxSocket

Das Interaktionsdiagramm in Abbildung 3.1 illustriert die bei Datenänderungen involvierten Schritte. Ein Vergleich mit den Umsetzungsschritten bzw. mit dem Interaktionsdiagram (Abbildung 2.8) in Abschnitt 2.5 zeigt, dass der YReduxSocket-Ansatz einen deutlich geringeren Implementierungsaufwand erfordert. Für einen detaillierten Vergleich sei auf Abschnitt 5.2 verwiesen.

3.2 Generische Konzeption und Spezifikation

Dieser Abschnitt konzentriert sich auf die generische Spezifikation von *YReduxSocket*. Zuerst wird der initiale Verbindungsaufbau zwischen einem Client und dem Server dargelegt. Danach folgt eine Beschreibung des Anmeldeprozesses für Clients, um bei Datenänderungen entsprechend informiert zu werden. Weiterhin wird der HTTP-Endpunkt erläutert, der den Clients ermöglicht, Datenänderungen zu initiieren. Abschließend wird der WebSocket-Endpunkt vorgestellt, der von den Clients bereitgestellt wird, um vom Server über erfolgreiche Aktualisierungen benachrichtigt zu werden.

3.2.1 Initialer Verbindungsaufbau

Die Verwendung von *YReduxSocket* erfordert zunächst das Aufbauen einer WebSocket-Verbindung zwischen dem Client und dem Server. Serverseitig muss dafür ein spezifischer

Endpunkt bereitgestellt werden, den der Client ansprechen kann. Dieser initiiert die WebSocket-Verbindung. Clientseitig ist es erforderlich, beim ersten Laden des Skripts eine Verbindung zu diesem Endpunkt herzustellen. JavaScript stellt hierfür eine spezifische WebSocket-API (Application Programming Interface) zur Verfügung[1]. Ähnlich bieten die meisten Server-Frameworks entsprechende Bibliotheken zur Nutzung von WebSockets an, wie zum Beispiel die WebSocket-Unterstützung in ASP.Net[2].

Falls der Server eine Authentifizierung, beispielsweise mittels eines JsonWebTokens, verlangt, muss der Client sicherstellen, dass dieses Token sowohl beim Verbindungsaufbau als auch bei jeder nachfolgenden WebSocket-Nachricht mitgesendet wird. Auf der Serverseite ist es entscheidend, das übertragene Token bei jeder eingehenden Nachricht zu validieren.

Eine stabile Verbindung ist für den effektiven Einsatz von *YReduxSocket* unerlässlich. Daher ist die Implementierung einer Fehlerbehandlung für Verbindungsschwierigkeiten von großer Bedeutung. Ein gängiger Ansatz ist das Anzeigen einer Fehlermeldung beim Nutzer während einer Unterbrechung der Verbindung, kombiniert mit dem Versuch, die Verbindung automatisch wiederherzustellen. Nach einer erfolgreichen Wiederverbindung ist es aus Gründen der Datenkonsistenz wichtig, alle Daten neu zu laden. Dies stellt sicher, dass keine Änderungen übersehen werden, die während einer Offline-Phase des Clients aufgetreten sein könnten.

Algorithmus 3.1 zeigt schematisch den Verbindungsaufbau und die Wiederherstellung. Zu Beginn wird eine WebSocket-Verbindung hergestellt und die Verbindungsinformationen in einer Variablen gespeichert. Es folgt eine Endlosschleife, in der in regelmäßigen Abständen, hier jede Sekunde, überprüft wird, ob die Verbindung noch besteht. Bei einer Unterbrechung wird dem Nutzer eine Fehlermeldung angezeigt und versucht, die Verbindung erneut aufzubauen. Nach erfolgreicher Wiederverbindung wird der `reduxState` auf den initialen Zustand zurückgesetzt, um ein erneutes Laden der Daten vom Server zu erzwingen.

Algorithmus 3.1 Verbindungsaufbau und Wiederherstellung clientseitig

```
socketConnection ← connect to server, store connection
while true do
    if socketConnection is not connected then
        Display Reconnection message
        socketConnection.reconnect()
        if socketConnection is connected then
            reduxState ← initialState                ▷ Force a reload of data
            Remove Reconnecting message
        end if
    end if
    Wait for 1 second
end while
```

[1] `https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API`

[2] `https://learn.microsoft.com/en-us/aspnet/core/fundamentals/websockets?view=aspnetcore-8.0`

3.2.2 Anmeldung auf Datenänderungen

In einem typischen Szenario sind zahlreiche Clients gleichzeitig mit dem Server verbunden. Eine einfache Methode, Datenänderungen über den WebSocket zu verteilen, wäre die Verwendung eines Broadcasts, bei dem die Aktualisierungsnachrichten an alle verbundenen Clients gesendet werden. Dieser Ansatz führt jedoch zu einer Flut unnötiger Nachrichten, was die Skalierbarkeit des Systems beeinträchtigen kann.

Ein effizienterer Ansatz besteht darin, die Clients in spezifische Gruppen zu organisieren. Hierbei registrieren sich die Clients für Updates zu Datenänderungen, die für sie relevant sind. Diese Vorgehensweise ermöglicht eine gezielte Nachrichtenverteilung und reduziert die Netzwerklast. Die Gruppeneinteilung bei „Yoshie.io“ ist in Abbildung 3.2 dargestellt. Nachfolgend wird ein detaillierter Blick auf diese Gruppenstruktur und ihre Funktionsweise geworfen.

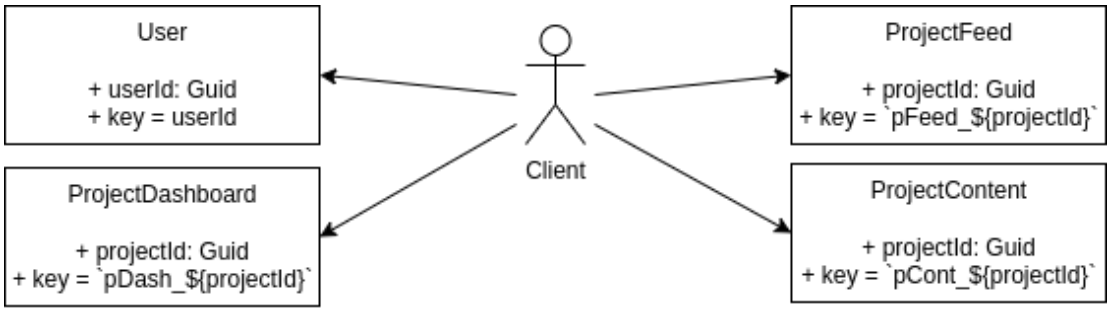

Abbildung 3.2: WebSocket-Gruppen bei „Yoshie.io“

In „Yoshie.io“ sind Datensätze als sogenannte `YNode` in Projekten strukturiert. Ein Projekt ist dabei selbst eine `YNode`. Jeder Datensatz verfügt über eine `projectId`, die auf ein bestimmtes Projekt verweist. Für die eindeutige Identifikation einer Gruppe wird ein spezieller Schlüssel definiert, der für die jeweilige Gruppe einzigartig ist. Bei den in Abbildung 3.2 dargestellten Gruppen wird dies durch Verwendung der einzigartigen ID der betreffenden Entität – in diesem Fall der Nutzer oder Projekte – erreicht. Existieren mehrere Gruppen derselben Entitätsart (hier Projekte), wird ein Präfix hinzugefügt, um Eindeutigkeit zu gewährleisten.

Stellt ein Client eine Verbindung zum WebSocket her, wird er automatisch der Gruppe `User` zugeordnet. Die Information über die UserId des Clients wird im Authentifizierungstoken übermittelt. Bei einem Verbindungsabbruch wird der Client wieder aus der Gruppe `User` entfernt. Diese Gruppe dient dazu, WebSocket-Nachrichten an alle Clients zu senden, die mit demselben Benutzerkonto verbunden sind. Ein praktisches Beispiel hierfür ist die Erstellung eines neuen Projekts. Zum Zeitpunkt der Erstellung ist kein anderer Nutzer außer dem Ersteller mit dem Projekt verbunden, sodass nur diesem Nutzer die Information übermittelt werden muss. Auf diese Weise kann sichergestellt werden, dass der Nutzer, wenn er sowohl über sein Smartphone als auch über seinen Desktop verbunden ist, das neu angelegte Projekt sofort auf beiden Geräten sieht.

Der Client hat die Option, alle Projekte, zu denen er Zugang hat, zu laden. Sobald dies geschieht, werden die Projektdaten in seinem Redux-basierten Speicher (Store) abgelegt. Um Live-Updates zu Änderungen dieser Projekte zu erhalten – beispielsweise Modifikationen einzelner Datenwerte wie Namen – ist es notwendig, dass der Client sich für die

Gruppe `ProjektDashboard` registriert. Dies betrifft alle `projectIds`, die geladen wurden. Durch diese Registrierung ist der Client in der Lage, Echtzeit-Updates zu erhalten, sobald Änderungen an den Projektdaten vorgenommen werden.

Ein Client kann auch ein einzelnes Projekt öffnen, was bedeutet, dass alle Inhalte dieses Projekts in den Store geladen werden. Unter Projektinhalt versteht man bei „Yoshie.io“ `YNode`-Entitäten, denen die `projectId` des jeweiligen Projekts zugeordnet ist. Beispiele hierfür sind Dateien, Tags oder Aufgaben. Ein Client kann jeweils nur Inhalte eines Projekts geladen haben. Beim Laden eines Projekts muss sich der Client für die Gruppe `ProjectContent` der entsprechenden `projectId` registrieren. Hatte er zuvor Inhalte eines anderen Projekts geladen, muss er sich von dieser Gruppe abmelden.

Schließlich gibt es die Gruppe `ProjectFeed`. Ein Projekt-Feed entspricht Einträgen welche Aktualisierungen eines Projekts bzw. dessen Inhalte beschreiben. Jedes Projekt bietet eine eigene Ansicht des Feed. Zudem gibt es eine Gesamtübersicht, der den Feed aller Projekte eines Nutzers zusammenfasst. Lädt der Client einen spezifischen Projekt-Feed in seinen Store, muss er sich für die Gruppe des entsprechenden Projekts registrieren und sich gegebenenfalls vom vorherigen Projekt-Feed abmelden. Beim Laden der Gesamtübersicht erfolgt eine Anmeldung bei allen `ProjectFeed`-Gruppen, zu denen der Client Zugriff hat, ähnlich der Registrierung bei der `ProjectDashboard`-Gruppe.

3.2.3 Definition von Actions

Wie bereits in Abschnitt 2.3.3 erläutert, können Actions mit der `dispatch`-Methode an den Store übermittelt werden. Diese Actions dienen entweder dazu, eine Reducer-Funktion aufzurufen oder einen Seiteneffekt auszulösen. Mit der Einführung von *YReduxSocket* ergibt sich jedoch eine leichte Modifikation dieses Prozesses. Es ist wichtig, zwischen zwei Arten von Actions zu unterscheiden: solchen, die vom Client an den Server gesendet werden, um beispielsweise eine Datenänderung zu veranlassen (siehe Abschnitt 3.2.4), und solchen, die den Erfolg einer Operation vom Server an den Client übermitteln (siehe Abschnitt 3.2.5).

Zur klaren Unterscheidung dieser beiden Action-Typen werden die Begriffe *trigger-Action* und *success-Action* eingeführt. Eine *trigger-Action* bezieht sich auf eine Action, die vom Client an den Server gesendet wird, während eine *success-Action* eine Action ist, die vom Server zum Client gesendet wird, um den erfolgreichen Abschluss einer Operation zu signalisieren.

Bei der Verwendung von *YReduxSocket* ist es erforderlich, Actions sowohl auf der Client- als auch auf der Serverseite zu definieren. Auf der Clientseite bleibt die Definition der Actions gemäß Abschnitt 2.3.3 unverändert. Allerdings entfallen die herkömmlichen HTTP-Aufrufe als Seiteneffekte, da *trigger-Actions* nun direkt an den Server gesendet werden. Die *success-Actions* lösen weiterhin den Aufruf einer Reducer-Funktion aus.

Auf der Serverseite ist nun ebenfalls eine Definition der Actions erforderlich. Hierbei muss für jede Action eine eindeutige Identifikation sowie der zu übermittelnde Payload festgelegt werden. Zusätzlich muss bei *trigger-Actions* spezifiziert werden, welche *success-Actions* im Falle einer erfolgreichen Operation auf welcher WebSocket-Gruppe (siehe Abschnitt 3.2.2) zurückgemeldet werden sollen. Abbildung 3.3 veranschaulicht das beschriebene nochmals als Klassendiagramm. Dabei handelt es sich bei `TriggerAction` und bei `SuccessAction` um abstrakte Basisklassen, die Actions `UpdateName` und `UpdateNameSuccess`

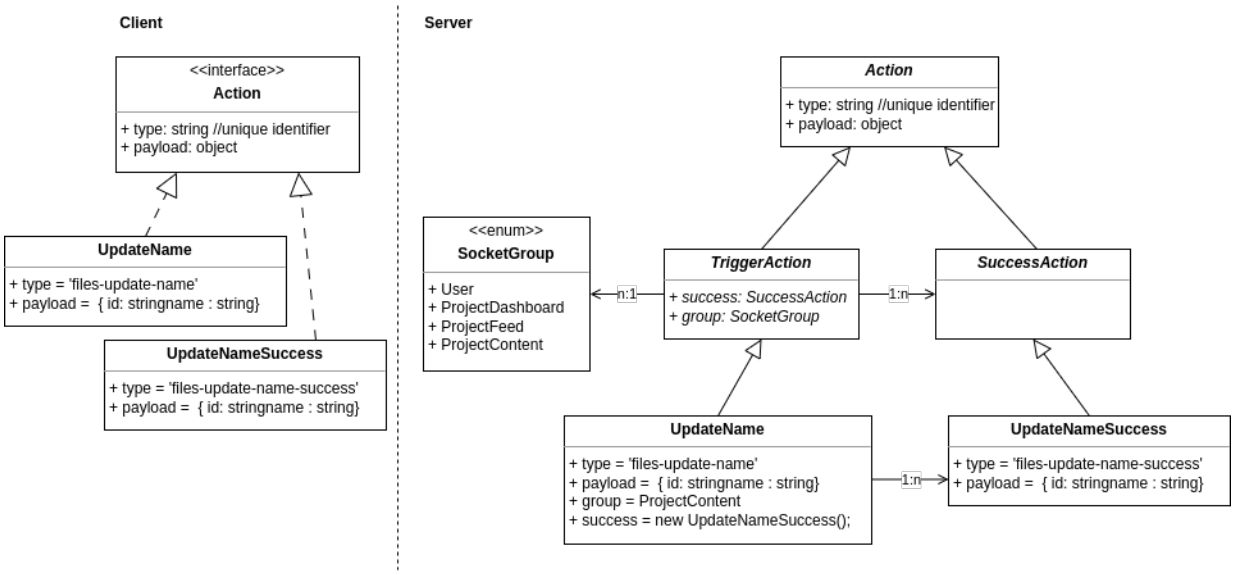

Abbildung 3.3: Klassendiagramm: YReduxSocket Actions

sind Beispielimplementierungen dieser Basisklassen. In der Regel existieren mehrere solcher Implementierungen.

3.2.4 Http Endpunkt `dispatch(triggerAction)`

YReduxSocket setzt voraus, dass der Server einen HTTP-Endpunkt bereitstellt, über den der Client `triggerActions` direkt an den Server senden kann. Dies unterscheidet sich vom konventionellen Ansatz, bei dem das Auslösen einer Aktion einen Seiteneffekt erzeugt, der wiederum einen spezifischen HTTP-Endpunkt aufruft.

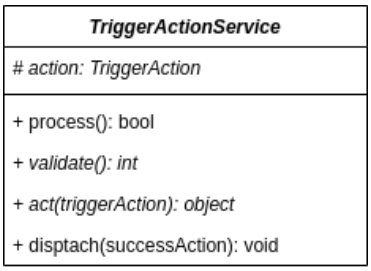

Abbildung 3.4: UML-Klassendiagramm: `TriggerActionService`

Auf dem Server gibt es die abstrakte Klasse `TriggerActionService`, dargestellt im UML-Klassendiagramm in Abbildung 3.4. Für jede `TriggerAction` sollte eine Implementierung dieses Services existieren. In dieser Implementierung müssen die abstrakten Methoden `validate()` und `act(triggerAction): object` definiert werden. `validate()` analysiert den Payload der `TriggerAction` und gibt einen HTTP-Statuscode zurück. Bei erfolgreicher Validierung sollte der Statuscode 200 (OK) sein. Die Methode `act` führt die Aktualisierung durch und gibt ein Objekt zurück, das als Payload für die an den Client gesendete `SuccessAction` dient.

Wenn eine `TriggerAction` über den `dispatch`-Endpunkt am Server eintrifft, wird die Methode `process(): boolean` der zur Action gehörigen `TriggerActionService` Implementierung aufgerufen.

3.2.5 WebSocket Endpunkt `dispatch(successAction)`

3.3 Konsistenzverhalten

Durch den Einsatz eines Redux-basierten Datenstores verfügt jeder beteiligte Client über seine eigene „Single Source of Truth“. Es ist jedoch wichtig zu beachten, dass die tatsächliche Wahrheit in der von einem Server verwalteten Datenbank liegt. Hier können bei Lese-Schreib- oder Schreib-Schreib-Konflikten sowie bei kurzzeitigen Unterbrechungen der WebSocket-Verbindung Inkonsistenzen zwischen den einzelnen Stores der Clients auftreten. Das JavaScript-Objekt welches vom Datenstore verwendet wird existiert im RAM des Webbrowsers, was bedeutet, dass beim Neuladen der Anwendung die Daten erneut vom Server abgerufen werden und der Datenstore wieder dem Zustand der Datenbank entspricht. Darüber hinaus werden die Daten aktualisiert, wenn eine neue Erfolgsmeldung eines Aktualisierungsvorgangs eintrifft. In diesem Kontext kann laut Tannenbaum und Van Steen von „eventual consistency“ gesprochen werden [TS08, S. 319, 322].

Für viele Anwendungsfälle ist diese „eventual consistency“ durchaus ausreichend. Es gibt jedoch Szenarien, in denen eine hohe Konsistenz zwischen den einzelnen Clients bzw. Client-Stores erforderlich ist. Dabei wurde ein Algorithmus von Marijn Haverbeke verwendet [Hav15], der in YReduxSocket implementiert wurde. Vereinfacht funktioniert der Algorithmus folgendermaßen:

- Am Server werden die Actions (beispielsweise über eine Redis-Datenbank) an zentraler Stelle gesammelt.
- Alle Actions desselben Typs haben eine Versionsnummer.
- Auch die Clients haben die zuletzt gesendete Versionsnummer gespeichert.
- Wenn ein Client eine neue Action auslösen möchte, sendet er sie zusammen mit seiner Versionsnummer an den Server.

Nun unterscheidet der Algorithmus 2 Fälle.

Die Versionsnummer des Clients stimmt mit der des Servers überein:

- Wenn die Versionsnummer am Server mit der des Clients übereinstimmt, wird die Action ausgeführt (Datenbankanpassung) und im Redis-Cache gespeichert.
- Darüber hinaus werden alle anderen Clients, die sich für diese Action angemeldet haben, über eine NewActionMeldung benachrichtigt.
- Anschließend erhöhen alle Clients ihre Versionsnummer.

Die Versionsnummer des Clients stimmt nicht mit der des Servers überein:

- In diesem Fall verwirft der Server einfach die empfangene Action.
- Da die Versionsnummern nicht übereinstimmen, wurde eine „NewAction“-Meldung an den ausführenden Client gesendet.
- Wenn die „NewAction“-Meldung eintrifft, holt sich der Client alle Actions, die größer sind als seine eigene Versionsnummer, vom Server und wendet sie auf seinem Redux-Store an.
- Danach sendet er seinen Änderungswunsch (Action) erneut an den Server.

Es ist erwähnenswert, dass das Konsistenzproblem nicht nur bei YReduxSocket, sondern auch im herkömmlichen Modell auftritt.

3.3.1 eventual consistency

3.3.2 Algorithmus von H.

Kapitel 4

Implementierung und Tests

4.1 Software Architektur

4.2 Client: Angular

4.3 Server: ASP.Net

Kapitel 5

Fazit

5.1 Zusammenfassung

5.2 Vergleich mit herkömmlichen Ansatz

5.3 Verbesserungspotential

Literatur

[Ack21] Philip Ackermann. *Webentwicklung: Das Handbuch für Fullstack-Entwickler*. Bonn, GERMANY: Rheinwerk Verlag, 2021. ISBN: 978-3-8362-6884-4. URL: http://ebookcentral.proquest.com/lib/fuhagen-ebooks/detail.action?docID=6748877 (besucht am 04. 01. 2024).

[Bae19] Sammie Bae. *JavaScript Data Structures and Algorithms: An Introduction to Understanding and Implementing Core Data Structure and Algorithm Fundamentals*. en. Berkeley, CA: Apress, 2019. ISBN: 978-1-4842-3987-2. DOI: 10.1007/978-1-4842-3988-9. URL: http://link.springer.com/10.1007/978-1-4842-3988-9 (besucht am 03. 01. 2024).

[Far17] Oren Farhi. „Adding State Management with ngrx/store". en. In: *Reactive Programming with Angular and ngrx: Learn to Harness the Power of Reactive Programming with RxJS and ngrx Extensions*. Hrsg. von Oren Farhi. Berkeley, CA: Apress, 2017, S. 31–49. ISBN: 978-1-4842-2620-9. DOI: 10.1007/978-1-4842-2620-9_3. URL: https://doi.org/10.1007/978-1-4842-2620-9_3 (besucht am 20. 09. 2023).

[Fre+21] Eric Freeman u. a. *Entwurfsmuster von Kopf bis Fuß*. Nov. 2021. ISBN: 978-3-96010-503-9. URL: https://content.select.com/de/portal/media/view/603e7135-4b3c-4b0f-9357-05bfb0dd2d03 (besucht am 04. 01. 2024).

[Hav15] Marijn Haverbeke. *Collaborative Editing in ProseMirror*. en. Okt. 2015. URL: https://marijnhaverbeke.nl/blog/collaborative-editing.html (besucht am 04. 10. 2023).

[HZ20] Nils Hartmann und Oliver Zeigermann. *React: Grundlagen, fortgeschrittene Techniken und Praxistipps - mit TypeScript und Redux*. ger. 2., überarbeitete und erweiterte Auflage. Heidelberg: dpunkt.verlag, 2020. ISBN: 978-3-96088-419-4.

[McF19] Timo McFarlane. „Managing State in React Appli- cations with Redux". en. In: (2019). URL: https://www.theseus.fi/bitstream/handle/10024/265492/McFarlane_Timo.pdf (besucht am 22. 09. 2023).

[TS08] Andrew S. Tanenbaum und Maarten van Steen. *Verteilte Systeme: Prinzipien und Paradigmen*. ger. 2., aktualisierte Auflage. it-informatik. München Harlow Amsterdam Madrid Boston San Francisco Don Mills Mexico City Sydney: Pearson, 2008. ISBN: 978-3-8273-7293-2.

Anhang A

Eigenständigkeitserklärung

„Ich erkläre, dass ich die schriftliche Ausarbeitung zur Abschlussarbeit selbstständig und ohne unzulässige Inanspruchnahme Dritter verfasst habe. Ich habe dabei nur die angegebenen Quellen und Hilfsmittel verwendet und die aus diesen wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht. Die Versicherung selbstständiger Arbeit gilt auch für enthaltene Zeichnungen, Skizzen oder graphische Darstellungen. Die Ausarbeitung wurde bisher in gleicher oder ähnlicher Form weder derselben noch einer anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht. Mit der Abgabe der elektronischen Fassung der endgültigen Version der Ausarbeitung nehme ich zur Kenntnis, dass diese mit Hilfe eines Plagiatserkennungsdienstes auf enthaltene Plagiate geprüft werden kann und ausschließlich für Prüfungszwecke gespeichert wird.“

Ricardo Stolzlechner