

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**

19. Februar 2024



# Abstract



# Inhaltsverzeichnis

<b>1</b>	<b>Einleitung</b>	<b>11</b>
1.1	Hintergrund und Motivation . . . . .	11
1.2	Problemstellung . . . . .	12
1.2.1	Datenkonsistenz am selben Client . . . . .	12
1.2.2	Datenkonsistenz zwischen verschiedenen Clients . . . . .	13
1.3	Ziele der Arbeit . . . . .	13
1.4	Struktur der Arbeit . . . . .	14
<b>2</b>	<b>Grundlagen</b>	<b>15</b>
2.1	Single Page Applications (SPAs) . . . . .	15
2.2	Client-Kommunikation . . . . .	15
2.2.1	asynchrone Kommunikation . . . . .	16
2.2.2	synchrone Kommunikation . . . . .	17
2.2.3	synchrone und asynchrone Kommunikation . . . . .	18
2.3	Transaktionen und ACID-Eigenschaften . . . . .	19
2.4	Komponentenbasierte Webentwicklung . . . . .	20
2.5	Zentraler Redux-basierter Datenstore . . . . .	21
2.5.1	Schnittstellen . . . . .	22
2.5.2	State . . . . .	23
2.5.3	Selektoren . . . . .	24
2.5.4	Actions . . . . .	24
2.5.5	Reducer . . . . .	25
2.5.6	Side Effects . . . . .	26
2.5.7	Fassade . . . . .	27
2.6	Datenstore mit WebSocket: herkömmlicher Ansatz . . . . .	27

<b>3</b>	<b>YReduxSocket</b>	<b>31</b>
3.1	Kurzbeschreibung . . . . .	31
3.2	Generische Konzeption und Spezifikation . . . . .	32
3.2.1	Initialer Verbindungsaufbau . . . . .	32
3.2.2	Anmeldung auf Datenänderungen . . . . .	34
3.2.3	Definition von Actions . . . . .	35
3.2.4	HTTP-Endpunkt <code>dispatch(triggerAction)</code> . . . . .	36
3.2.5	WebSocket Endpunkt <code>dispatch(successAction)</code> . . . . .	37
3.3	Analyse des Konsistenzverhaltens von YReduxSocket . . . . .	38
3.3.1	Transaktionen . . . . .	38
3.3.2	Monotones Lesen . . . . .	38
3.3.3	Monotones Schreiben . . . . .	39
3.3.4	Read Your Writes . . . . .	39
3.3.5	Write Follows Reads . . . . .	40
3.3.6	Haverbeke-Algorithmus . . . . .	40
3.4	Vergleich mit herkömmlichen Ansatz . . . . .	45
<b>4</b>	<b>Implementierung und Tests</b>	<b>49</b>
4.1	Software Architektur . . . . .	49
4.2	Client: Angular . . . . .	49
4.3	Server: ASP.Net . . . . .	49
4.4	Vergleich mit herkömmlichen Ansatz . . . . .	49
<b>5</b>	<b>Fazit</b>	<b>51</b>
5.1	Zusammenfassung . . . . .	51
5.2	Verbesserungspotential . . . . .	51

# Abbildungsverzeichnis

1.1	UML-Diagramm eines einfachen Komponentenbaums . . . . .	12
1.2	Bildschirmausschnitt eines einfachen Komponentenbaums . . . . .	13
2.1	Prozessarchitektur von Single Page Applications . . . . .	16
2.2	Prozessarchitektur von SPA's mit asynchroner Kommunikation . . . . .	17
2.3	Prozessarchitektur von SPA's mit synchroner Kommunikation . . . . .	17
2.4	Von <i>YReduxSocket</i> vorausgesetzte Prozessarchitektur . . . . .	18
2.5	Datenbindungsmechanismen: Datenfluss im Komponentenbaum . . . . .	20
2.6	globaler Service: Datenfluss im Komponentenbaum . . . . .	21
2.7	Redux-basierter Store: Schnittstellen . . . . .	22
2.8	Redux-basierter Store: Datenfluss im Komponentenbaum . . . . .	23
2.9	FileStore Abhängigkeiten . . . . .	28
2.10	Interaktionsdiagramm: Datenstore mit WebSocket, herkömmlicher Ansatz .	29
3.1	Interaktionsdiagramm: Datenstore mit WebSocket, YReduxSocket . . . . .	32
3.2	WebSocket-Gruppen bei „Yoshie.io“ . . . . .	34
3.3	Klassendiagramm: YReduxSocket Actions . . . . .	36
3.4	UML-Klassendiagramm des <b>TriggerActionService</b> . . . . .	36
3.5	Haverbeke-Algorithmus ein Client . . . . .	43
3.6	Haverbeke-Algorithmus zwei Clients . . . . .	44





# Algorithmenverzeichnis

2.1	Transaktion in einer relationalen Datenbank . . . . .	19
2.2	Statedefinition in NgRx . . . . .	24
2.3	Selektorendefinition in NgRx . . . . .	25
2.4	Actiondefinitionen in NgRx . . . . .	25
2.5	Reducerdefinitionen in NgRx . . . . .	26
2.6	Side Effect Definitionen in NgRx . . . . .	27
2.7	Fassade: Beispielimplementierung für den FileStore . . . . .	28
3.1	Verbindungsaufbau und Wiederherstellung clientseitig . . . . .	33
3.2	Methode <code>process()</code> in der Klasse <code>TriggerActionService</code> . . . . .	37
3.3	Haverbeke-Algorithmus: serverseitige Erweiterung . . . . .	41
3.4	Haverbeke-Algorithmus: clientseitige Erweiterung . . . . .	42



# Kapitel 1

## Einleitung

### 1.1 Hintergrund und Motivation

Die Webanwendung „Yoshie.io“<sup>1</sup> 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*, *Read Your Writes* 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 Konsistenzhaltung 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.

---

<sup>1</sup><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 muss eine Anwendung entwickelt werden, die sicherstellt, 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.

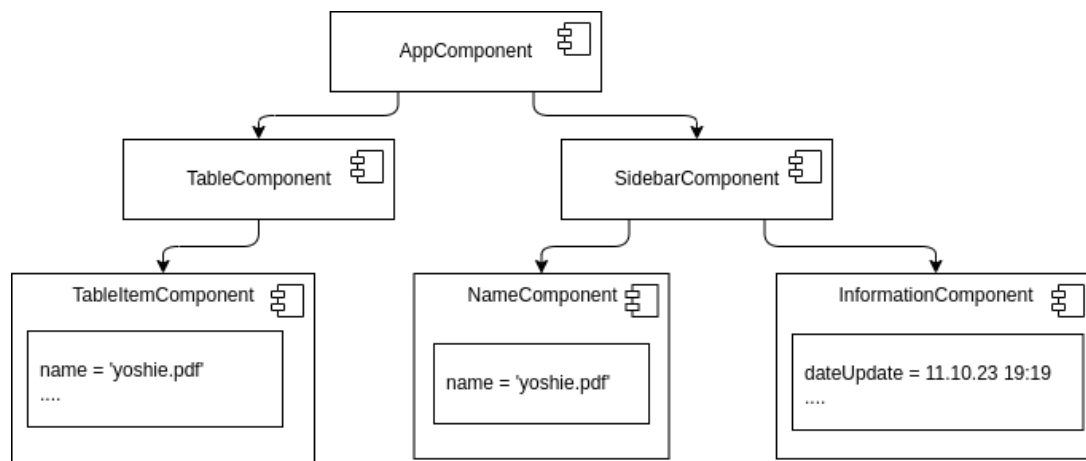


Abbildung 1.1: UML-Diagramm eines einfachen Komponentenbaums

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 die Anwendung 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.

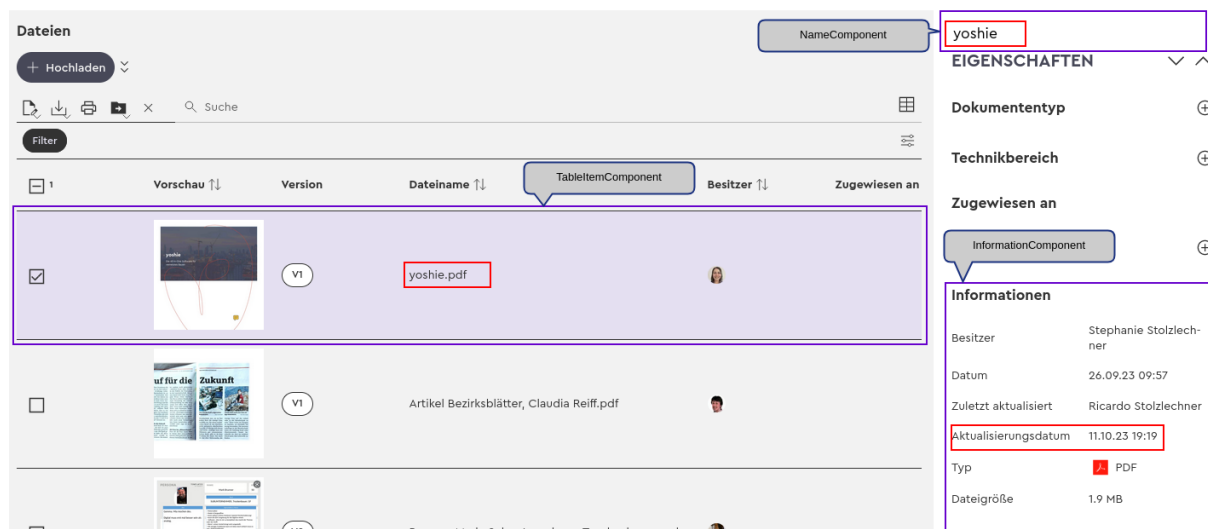


Abbildung 1.2: Bildschirmausschnitt eines einfachen Komponentenbaums

### 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 eigener 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 2.6 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 Framework 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 Webserver 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 Webrowsers 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].

### 2.2 Client-Kommunikation

In Abbildung 2.1 ist die Prozessarchitektur von Single Page Applications graphisch dargestellt. Dabei ist ersichtlich, dass es für die Clients ohne Erweiterung keine Möglichkeit gibt, untereinander zu kommunizieren.

Die Clients können nun auf zwei verschiedene Arten kommunizieren: synchron oder asynchron. Unter synchroner Kommunikation versteht man dabei eine Form der Kommunikation, bei der der Austausch für den Sender und den Empfänger gleichzeitig abläuft.

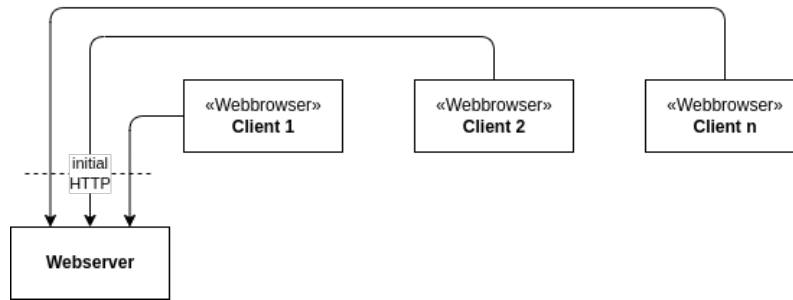


Abbildung 2.1: Prozessarchitektur von Single Page Applications

Bei der asynchronen Kommunikation kann das Empfangen durchaus zu einem anderen Zeitpunkt als das Senden passieren [CC01, S. 1]. Ein prominentes Beispiel für eine synchrone Kommunikation ist ein Telefongespräch, hingegen ist das Versenden einer E-Mail ein asynchroner Kommunikationsvorgang.

In den folgenden Abschnitten wird beleuchtet, welche Architekturerweiterungen an einer SPA getroffen werden müssen, um zunächst synchrone Kommunikation, dann asynchrone Kommunikation und abschließend beide Arten zu ermöglichen.

### 2.2.1 asynchrone Kommunikation

Um asynchrone Kommunikation zwischen den Clients einer Single-Page-Anwendung (SPA) zu realisieren, ist eine zentrale Datenbasis erforderlich. Diese lässt sich effektiv mittels einer Datenbank umsetzen. Um Interaktionen mit der Datenbank – wie das Abrufen, Hinzufügen oder Modifizieren von Daten – zu erleichtern, empfiehlt es sich, einen Server als Vermittler zwischen den Clients und der Datenbank einzusetzen. Dieser Server kann eine HTTP-API (Application Programming Interface) bereitstellen, die spezifische Funktionen für Datenbankoperationen über HTTP-Methoden anbietet. Darüber hinaus kann der Server genutzt werden, um verschiedene Sicherheitsmaßnahmen zu implementieren. Ein Beispiel hierfür ist die Authentifizierung der Clients, die notwendig sein kann, bevor sie Datenbankabfragen starten dürfen. Abbildung 2.2 illustriert die Erweiterung der SPA-Architektur, um asynchrone Kommunikation über einen gemeinsamen Speicher zu ermöglichen.

Ein wesentlicher Nachteil dieser Kommunikationsmethode ist die Unfähigkeit, Echtzeit-Aktualisierungen zu unterstützen. Angenommen, sowohl Client 1 als auch Client 2 haben dieselben Daten geladen, und Client 1 beschließt, diese Daten zu ändern. Aufgrund der Beschaffenheit des HTTP-Protokolls, welches zustandslos ist und keine dauerhafte Verbindung zu seinen Clients unterhält, kann der Server Client 2 nicht proaktiv über die Änderungen informieren. Der Server reagiert lediglich auf eingehende HTTP-Anfragen mit entsprechenden Antworten. Infolgedessen verbleiben die Daten auf Client 2 so lange veraltet, bis dieser erneut eine Anfrage an den Server stellt, um die von Client 1 modifizierten Daten abzurufen.



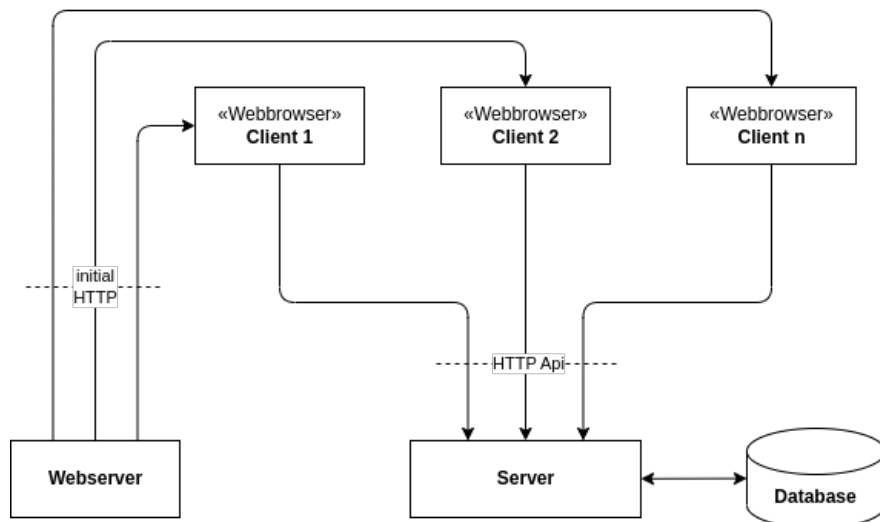


Abbildung 2.2: Prozessarchitektur von SPA's mit asynchroner Kommunikation

### 2.2.2 synchrone Kommunikation

Um den Clients einer Anwendung die Möglichkeit zur synchronen Kommunikation zu bieten, können Technologien wie Long Polling oder WebSockets zum Einsatz kommen [Ack21, S. 182]. In diesem Kontext konzentriert sich die Diskussion auf das WebSocket-Protokoll, welches auf der Basis von TCP bidirektionale Verbindungen zwischen Clients und Server ermöglicht, da *YReduxSocket* spezifisch eine WebSocket-Verbindung voraussetzt.

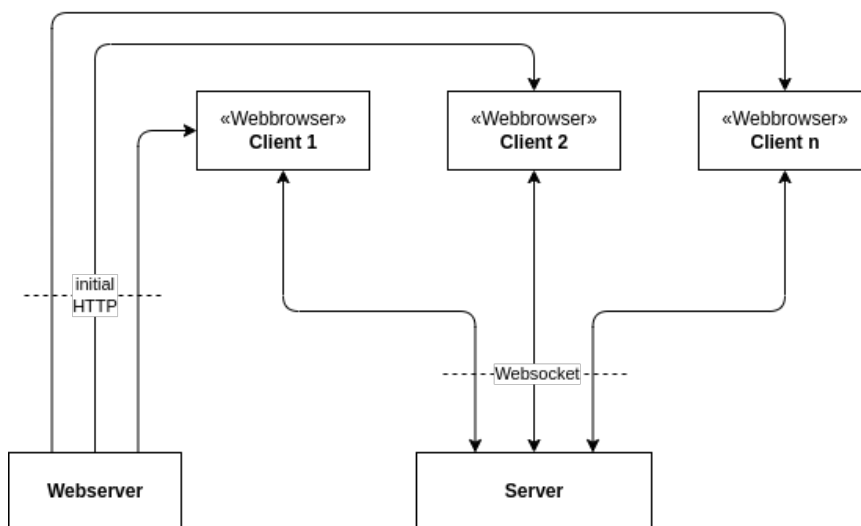


Abbildung 2.3: Prozessarchitektur von SPA's mit synchroner Kommunikation

Bei einer WebSocket-Verbindung muss der Client den Verbindungsaufbau initiieren. Ist die Verbindung einmal etabliert, kann jedoch auch der Server aktiv Daten an den Client übermitteln [Ack21, S. 184–185]. Um eine Überflutung mit irrelevanten Daten zu vermeiden, kann es notwendig sein, dass Clients sich für bestimmte Updates registrieren, anstatt dass der Server bei jeder neuen Nachricht einen Broadcast an alle verbundenen Clients sendet. Abbildung 2.3 illustriert die Erweiterung der SPA-Prozessarchitektur um synchrone Kommunikation mittels WebSocket zu ermöglichen. So kann beispielsweise eine

Nachricht von Client 1 über den Server an Client 2 weitergeleitet werden.

Ein Nachteil der rein synchronen Kommunikation ist, dass alle Clients permanent mit dem Server verbunden sein müssen. Ein Client, der sich zu einem späteren Zeitpunkt verbindet, kann keine Nachrichten empfangen die vor seiner Verbindung gesendet wurden. Um dieses Problem zu lösen und somit den Zugriff auf frühere Nachrichten zu ermöglichen, ist der Einsatz eines Datenspeichers erforderlich.

### 2.2.3 synchrone und asynchrone Kommunikation

Wie bereits in den Abschnitten 2.2.1 und 2.2.2 diskutiert, bringt jede Kommunikationsform – die synchrone als auch die asynchrone – spezifische Nachteile mit sich. Die Kombination beider Ansätze ermöglicht es jedoch, diese Einschränkungen zu überwinden, indem die Stärken der jeweils anderen Technologie genutzt werden. Durch die Verwendung einer Datenbank als zentralen Speicher können Änderungen in Echtzeit über das WebSocket-Protokoll an alle verbundenen Clients kommuniziert werden.

Für die Implementierung einer Single Page Application (SPA), die sowohl synchrone als auch asynchrone Kommunikation unterstützt, wie es *YReduxSocket* voraussetzt, ist eine dedizierte Serverinstanz erforderlich. Diese Instanz stellt eine HTTP-API bereit und ermöglicht gleichzeitig die Einrichtung einer WebSocket-Verbindung. Zudem verwaltet und speichert sie die Anwendungsdaten in einer Datenbank. Alle Kommunikationsprozesse zwischen den Clients werden über diese Serverinstanz abgewickelt, die als eigenständiger Prozess fungiert und nicht mit dem Webserver, der die SPA ausliefert, verwechselt werden sollte. Im weiteren Verlauf wird der Begriff „Server“ speziell für diese separate Serverinstanz verwendet. Abbildung 2.4 zeigt die für *YReduxSocket* erforderliche Prozessarchitektur, wobei jedes Rechteck im Diagramm einen unabhängigen Prozess symbolisiert.

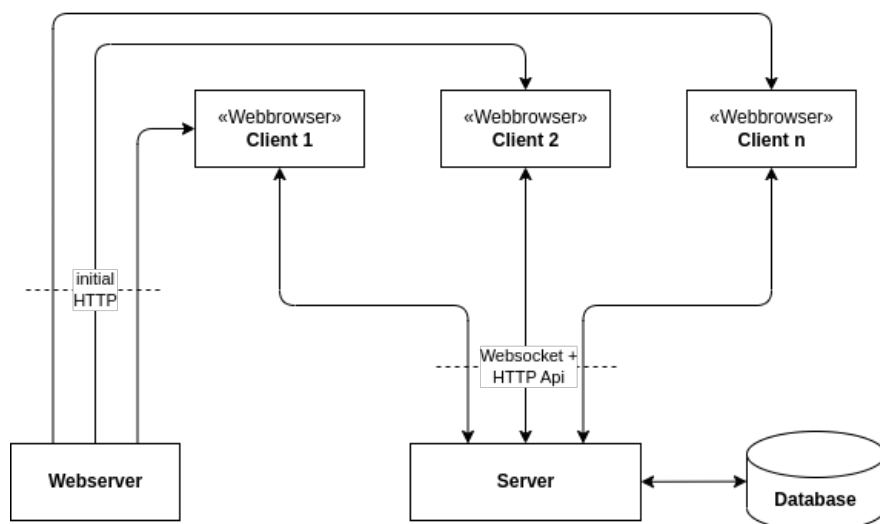


Abbildung 2.4: Von *YReduxSocket* vorausgesetzte Prozessarchitektur

Es ist zudem wichtig zu betonen, dass *YReduxSocket* die Nutzung genau einer Datenbank vorsieht, die als verlässliche Quelle der Wahrheit (Single Source of Truth) dient.

## 2.3 Transaktionen und ACID-Eigenschaften

Wie in Abbildung 2.4 dargestellt, greifen mehrere Clients gleichzeitig auf den Server zu. In solchen Szenarien kann es vorkommen, dass Clients gleichzeitig dieselben Daten in der Datenbank manipulieren wollen. Um parallele Serverzugriffe zu ermöglichen, setzt *YReduxSocket* den Einsatz einer Datenbank voraus, die Transaktionen unterstützt, welche den ACID-Eigenschaften entsprechen. Dies trifft beispielsweise auf gängige relationale Datenbanken zu.

---

**Algorithmus 2.1** Transaktion in einer relationalen Datenbank

---

```
transaction = createTransaction();
transaction.start();
try {
    act(); // Führt Datenbankänderungen durch
    transaction.commit();
} catch {
    transaction.rollback();
}
```

---

Algorithmus 2.1 illustriert das Transaktionsprinzip in einem an JavaScript angelehnten Pseudocode. Zunächst wird eine Transaktion erzeugt und gestartet. Anschließend wird versucht, Datenbankänderungen mit der Methode `act()` durchzuführen. Bei Erfolg wird die Transaktion mit `transaction.commit()` bestätigt. Vor dieser Bestätigung haben die Änderungen keine nach außen sichtbaren Auswirkungen. Schlägt der Versuch jedoch fehl, werden alle in `act()` vorgenommenen Änderungen durch `transaction.rollback()` rückgängig gemacht.

Nach Tanenbaum und van Steen muss eine Transaktion folgende Schlüsseleigenschaften aufweisen [TS08, S. 38, 39]:

1. **Atomic:** Die Transaktion definiert eine atomare Operation, d.h., sie wird entweder vollständig oder gar nicht ausgeführt.
2. **Consistent:** Die Transaktion muss konsistent sein. Nach ihrem Abschluss müssen alle Invarianten, die an die Daten gestellt werden, erfüllt sein. Wurde beispielsweise ein Datensatz während der Transaktion verändert, muss ein eventuell vorhandenes Aktualisierungsdatum entsprechend gesetzt sein.
3. **Isolated:** Transaktionen müssen isoliert oder serialisierbar sein. Dies bedeutet, dass sich parallel ablaufende Transaktionen nicht gegenseitig beeinflussen dürfen.
4. **Durable:** Diese Eigenschaft beschreibt die Dauerhaftigkeit einer Transaktion. Sie gewährleistet, dass die Ergebnisse der Transaktion auch nachfolgende Systemfehler überstehen.

## 2.4 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.

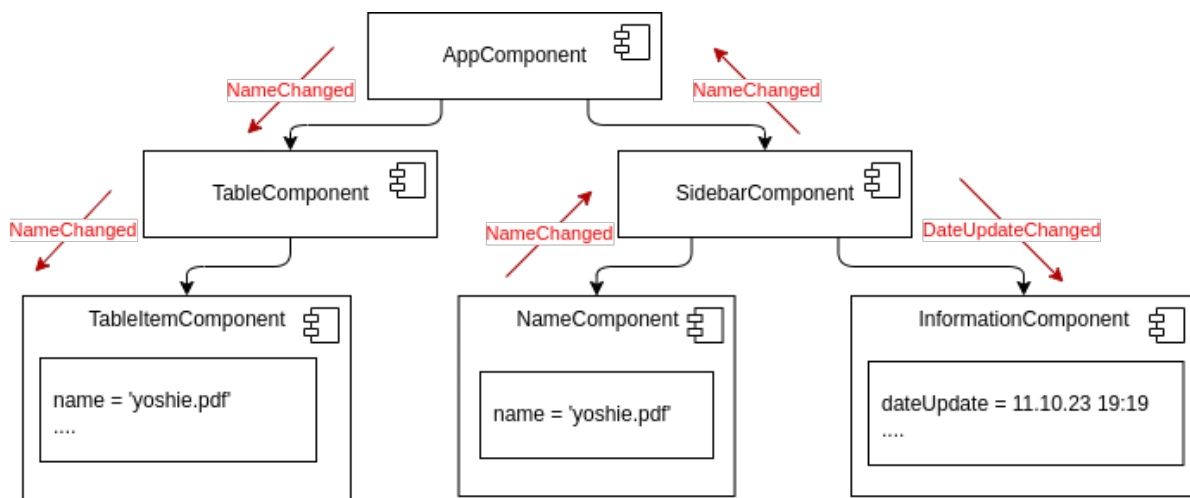


Abbildung 2.5: Datenbindungsmechanismen: Datenfluss im Komponentenbaum

Abbildung 2.5 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 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.

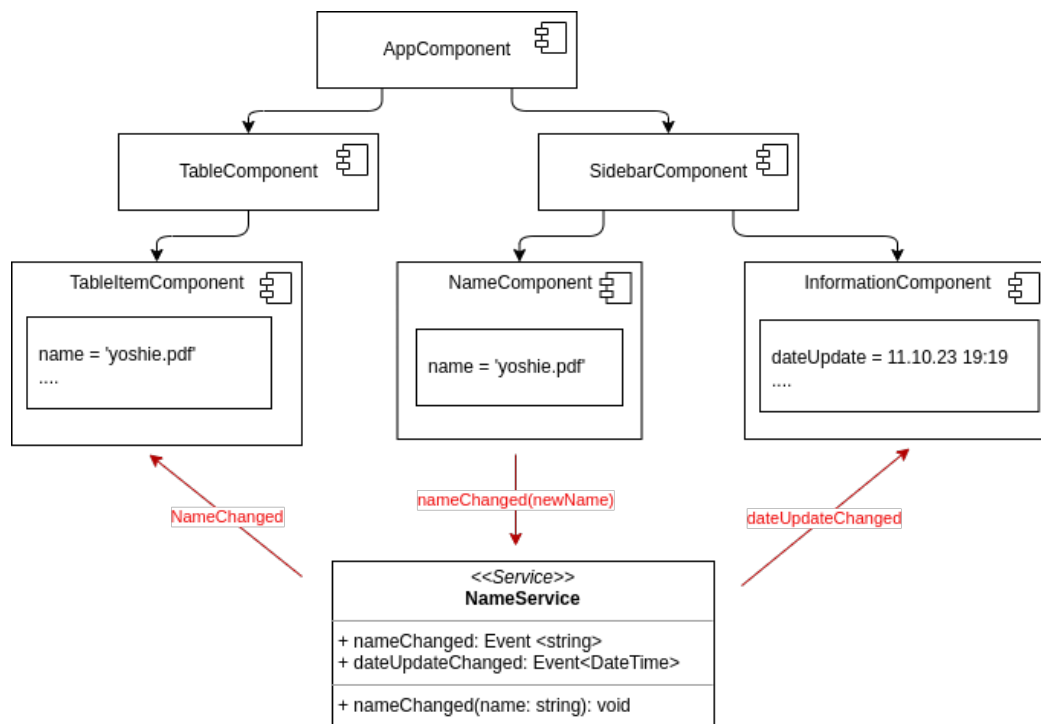


Abbildung 2.6: globaler Service: Datenfluss im Komponentenbaum

Im Beispiel in Abbildung 2.6 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.

## 2.5 Zentraler Redux-basierter Datenstore

Mit zunehmender Anzahl von Komponenten in einer Anwendung wächst auch die Komplexität des zugrundeliegenden Codes, hauptsächlich durch die Notwendigkeit, Daten zwischen den Komponenten zu synchronisieren und einen konsistenten Zustand zu gewährleisten. Um diese Komplexität zu verringern, empfiehlt sich die Verwendung eines zentralen Stores, der auf Redux basiert (siehe Abschnitt 2.4). Dieser Store wird auf der Clientseite implementiert und fungiert als eigenständige Softwareschicht, die zwischen dem User-Interface (den Komponenten) und der Serveranbindung vermittelt. Jeder Client verfügt somit über einen eigenen, individuellen Store.

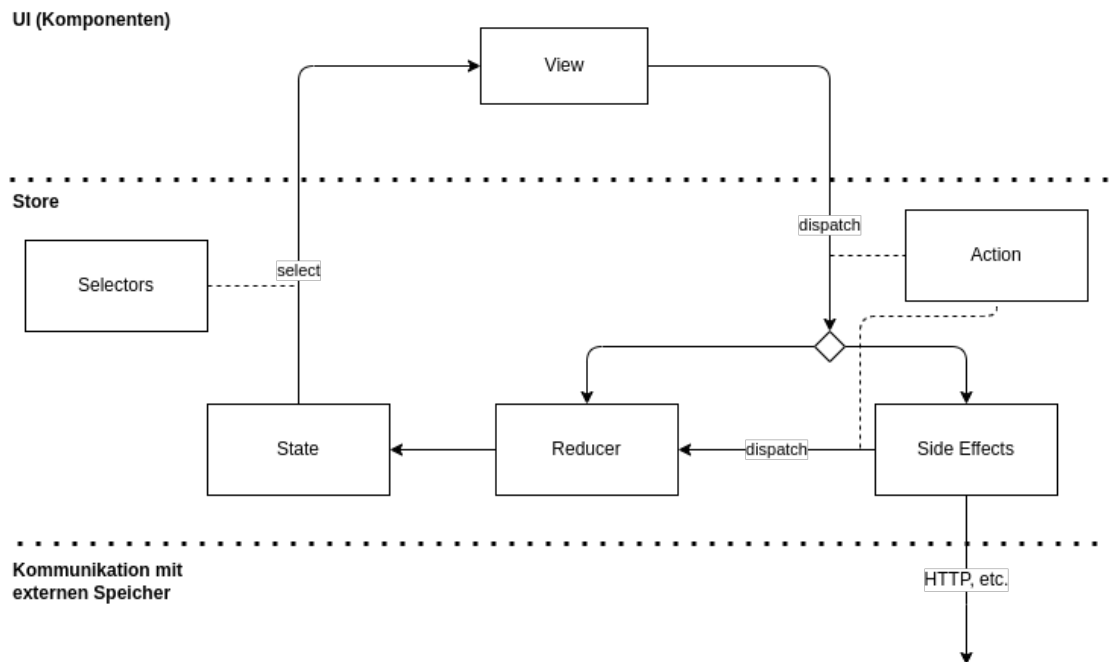


Abbildung 2.7: Redux-basierter Store: Schnittstellen

Redux, entwickelt von Dan Abramov, ist eine Implementierung des von Facebook definierten Designpatterns Flux. Dieses Pattern sieht einen unidirektionalen Datenfluss vor, wie in Abbildung 2.7 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 2.4). Dabei wird die Implementierung `NgRx`<sup>1</sup>, welche im Framework Angular eingesetzt werden kann, verwendet.

In Abbildung 2.8 wird illustriert, wie die Komponenten mit dem Store interagieren können, um Daten abzurufen oder Datenänderungen zu veranlassen. Im Folgenden wird zunächst detailliert auf die Schnittstellen eines Stores eingegangen. Anschließend werden die Kernkonzepte eines auf Redux basierenden Stores – darunter der State, die Selektoren, die Actions, der Reducer sowie die Side Effects – ausführlich erläutert.

## 2.5.1 Schnittstellen

Der Store, der wie ein Service global zugänglich ist, bietet als Schnittstelle zum User Interface (den Komponenten), zwei Methoden an: Mit der Methode `select`, der ein `Selektor` übergeben wird, kann ein Ausschnitt des States abgeholt werden. 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 informiert.

<sup>1</sup><https://ngrx.io/>

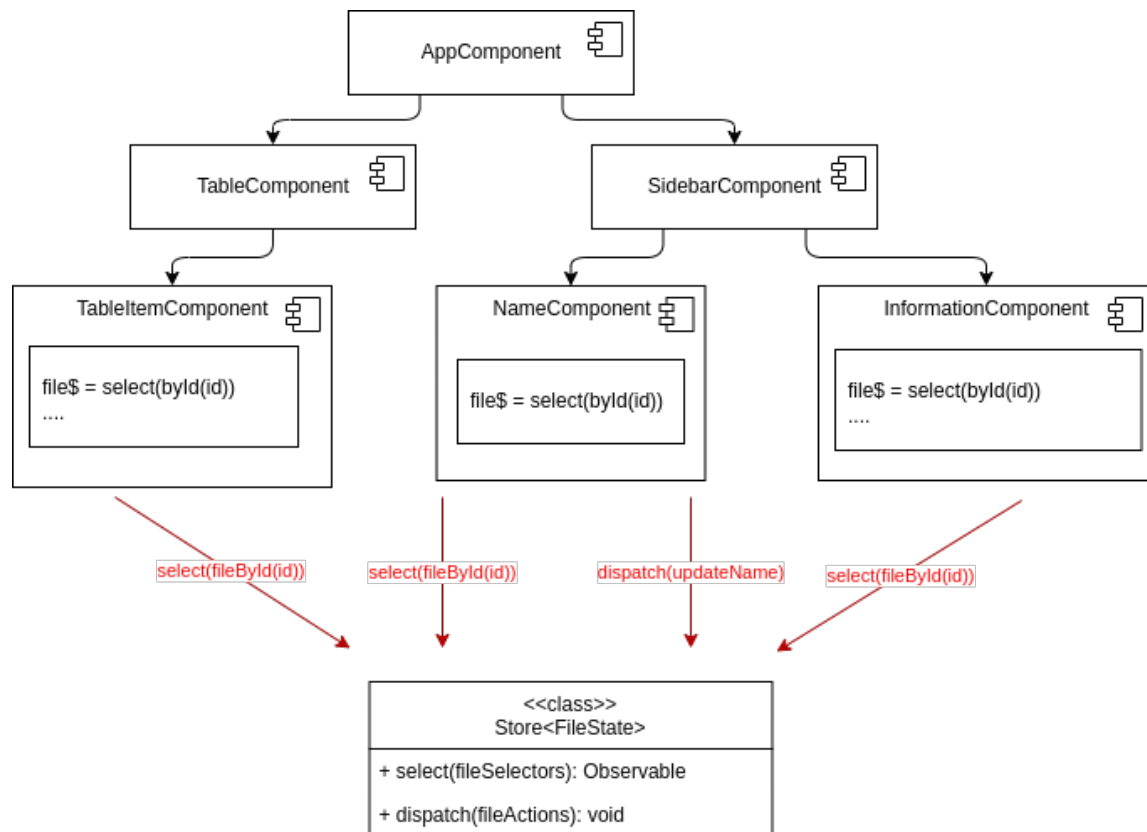


Abbildung 2.8: Redux-basierter Store: Datenfluss im Komponentenbaum

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.

Als weitere Schnittstelle bietet der Store die Möglichkeit, über sogenannte Side Effects mit externen Systemen zu kommunizieren, beispielsweise um Anfragen an externe Speicherlösungen zu senden, die etwa durch einen Server implementiert sein könnten. Eine detaillierte Erörterung der Side Effects erfolgt im Abschnitt 2.5.6.

## 2.5.2 State

Der State wird in Form eines einfachen JavaScript-Objekts implementiert. Dieses Objekt wird dazu verwendet den gesamten Zustand der Anwendung abzubilden. 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 2.2 zeigt die Statedefinition für das Beispiel, wie in Abbildung 2.8 dar-

---

**Algorithmus 2.2** Statedefinition in NgRx

---

```

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

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

```

---

gestellt. 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 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]

### 2.5.3 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 2.5.5) vergeben wird. Die zweite Methode, `createSelector`, dient dazu, einen Teil des States zu extrahieren, um so kleinere Teile nach außen zu geben.

Algorithmus 2.3 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 2.5.7 näher eingegangen.

### 2.5.4 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 2.4 zeigt die für das Beispiel notwendigen Action-Definitionen. Die definierten Actions können entweder von außen (siehe Abschnitt 2.5.7) oder durch einen Side Effect (siehe Abschnitt 2.5.6) dispatched werden. Dieses Dispatchen führt in der Regel dazu, dass



---

**Algorithmus 2.3** Selektorendefinition in NgRx

---

```

const state = createSelector<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
};

```

---

eine Reducer-Funktion aufgerufen wird, die einen neuen State liefert.

---

**Algorithmus 2.4** 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}>
    }
})

```

---

### 2.5.5 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 2.5 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 Hauptsektors (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 2.5.2). 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

---

**Algorithmus 2.5** 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)
  ))
}

```

---

dürfen, das heißt, es muss, wie in der Methode `onUpdateNameSuccess` ersichtlich, ein neues State-Objekt retourniert werden.

### 2.5.6 Side Effects

Side Effects 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 Side Effects 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 2.6 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.

---

**Algorithmus 2.6** Side Effect Definitionen 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))
      )));
}

```

---

### 2.5.7 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 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 2.7 zeigt eine Implementierung einer solchen Fassadenklasse, wie sie für das Beispiel benötigt wird.

Abbildung 2.9 fasst die Abhängigkeiten zwischen den einzelnen Store-Komponenten zusammen und zeigt, wie durch die Fassade eine lose Kopplung zwischen Store und Komponente erreicht wird.

## 2.6 Datenstore mit WebSocket: herkömmlicher Ansatz

In diesem Abschnitt wird die Kombination der in 2.5 und 2.2 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 Side Effect 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

**Algorithmus 2.7** 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}));
}

```

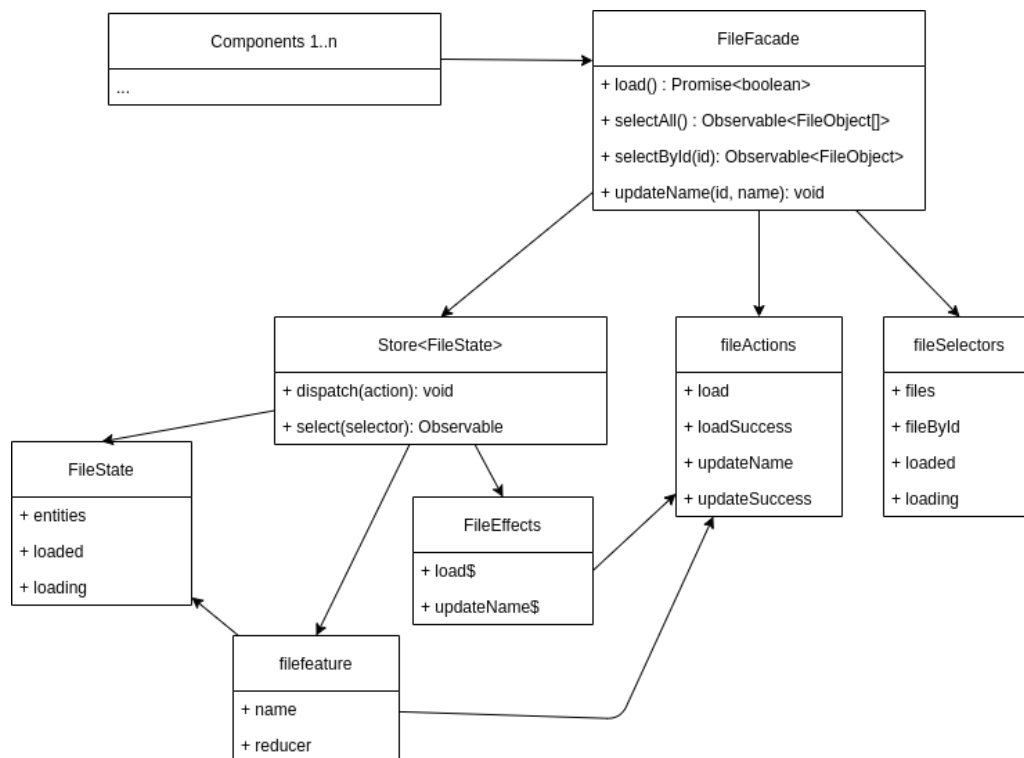


Abbildung 2.9: FileStore Abhängigkeiten

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.

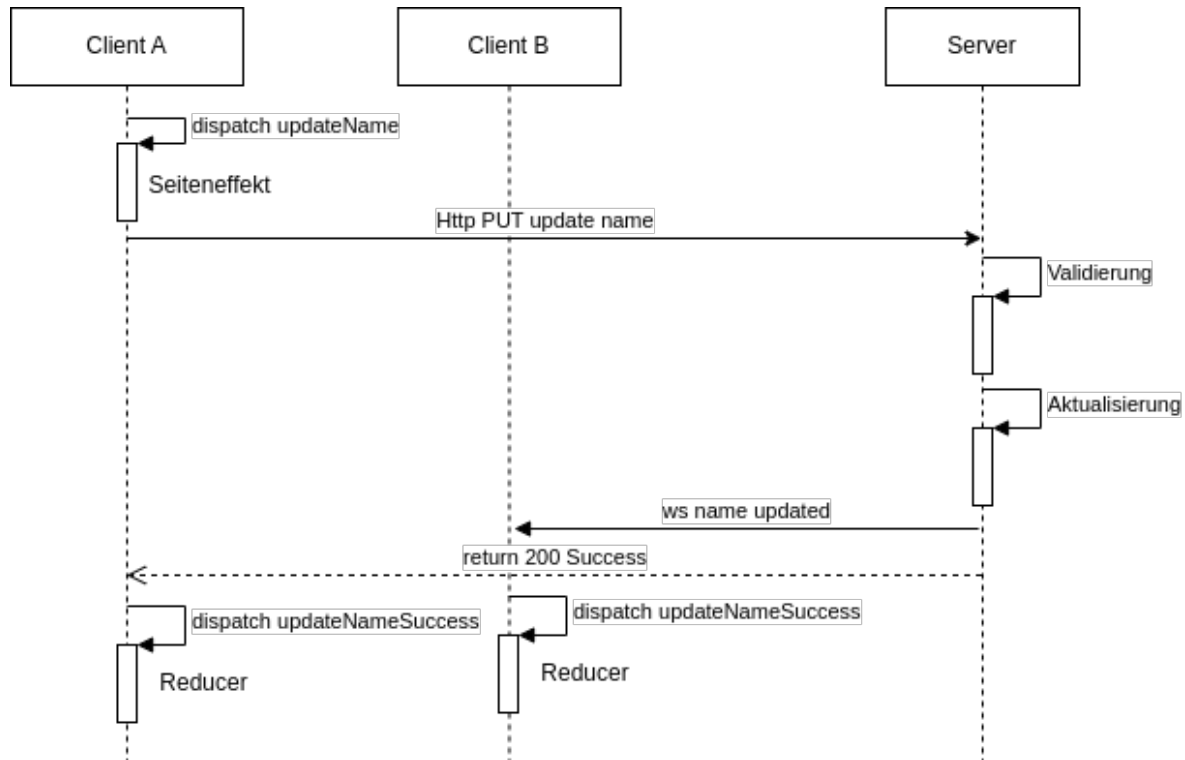


Abbildung 2.10: Interaktionsdiagramm: Datenstore mit WebSocket, herkömmlicher Ansatz

Abbildung 2.10 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 Side Effect auslöst.
2. Implementierung eines neuen HTTP-Endpunkts, sowohl server- als auch clientseitig.
3. Implementierung des Side Effects.
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 2.6 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 2.6 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 Side Effects, 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-Endpoint unter Übergabe der zweiten Action.
5. Implementierung der Reducer-Funktion auf der Clientseite zur Anpassung der Client-Daten.

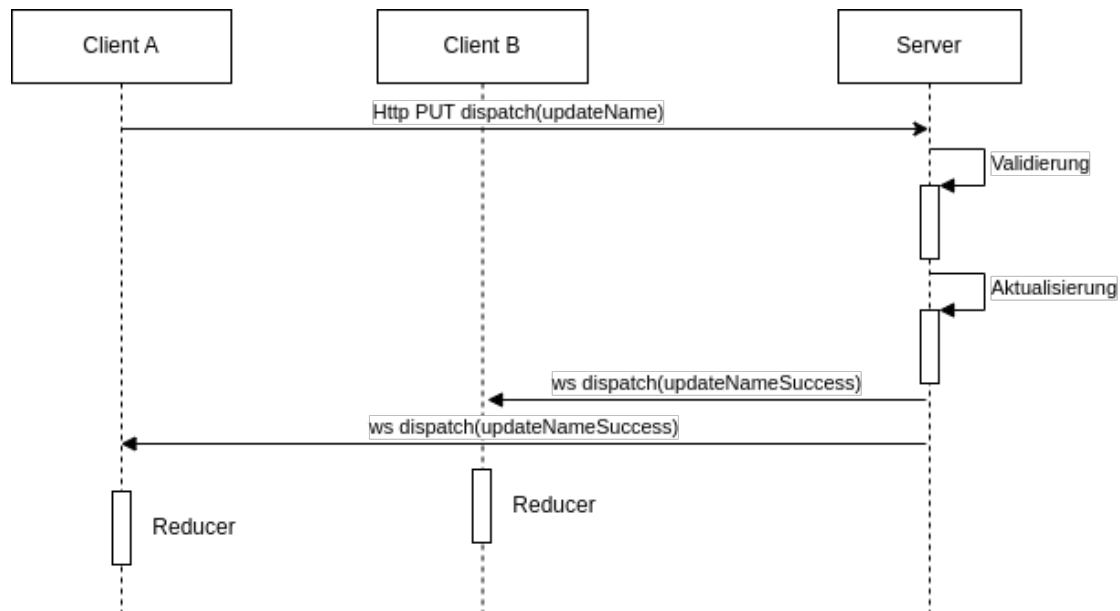


Abbildung 3.1: Interaktionsdiagramm: Datenstore mit WebSocket, YReduxSocket

Das Interaktionsdiagramm in Abbildung 3.1 illustriert die bei Datenänderungen involvierten Schritte. Ein Vergleich mit den Umsetzungsschritten bzw. mit dem Interaktionsdiagramm (Abbildung 2.10) in Abschnitt 2.6 zeigt, dass der YReduxSocket-Ansatz einen deutlich geringeren Implementierungsaufwand erfordert. Für einen detaillierten Vergleich sei auf Abschnitt 3.4 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 dargestellt. Danach folgt eine Beschreibung des Anmeldeprozesses für Clients, um bei Datenänderungen entsprechend informiert zu werden. Weiterhin wird der HTTP-Endpoint erläutert, der den Clients ermöglicht, Datenänderungen zu initiieren. Abschließend wird der WebSocket-Endpoint 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<sup>1</sup>. Ähnlich bieten die meisten Server-Frameworks entsprechende Bibliotheken zur Nutzung von WebSockets an, wie zum Beispiel die WebSocket-Unterstützung in ASP.Net<sup>2</sup>.

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
  
```

---



---

<sup>1</sup>[https://developer.mozilla.org/en-US/docs/Web/API/WebSockets\\_API](https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API)

<sup>2</sup><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.

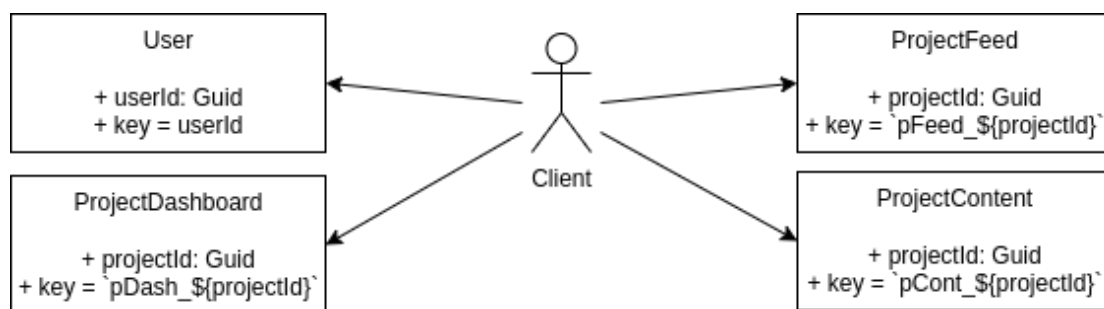


Abbildung 3.2: WebSocket-Gruppen bei „Yoshie.io“

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 `projectId`s, 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 Projekteinhalt 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 2.5.4 erläutert, können Actions mit der `dispatch`-Methode an den Store übermittelt werden. Diese Actions dienen entweder dazu, eine Reducer-Funktion aufzurufen oder einen Side Effect 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 2.5.4 unverändert. Allerdings entfallen die herkömmlichen HTTP-Aufrufe als Side Effects, 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`

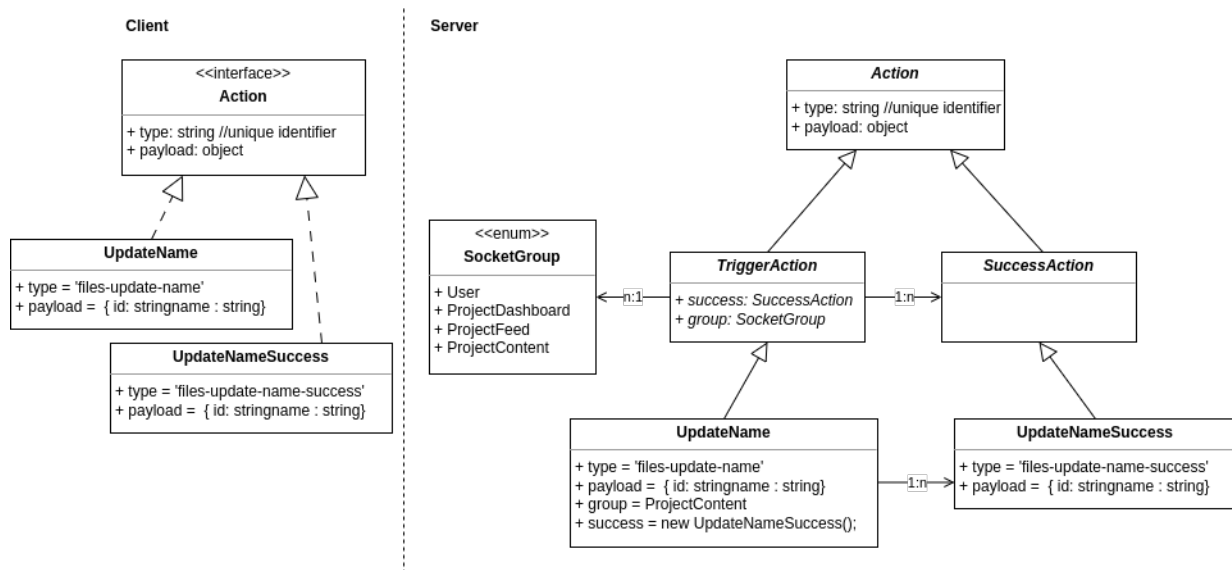


Abbildung 3.3: Klassendiagramm: YReduxSocket Actions

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, der es dem Client ermöglicht, `triggerActions` direkt an den Server zu senden. Dies unterscheidet sich vom konventionellen Ansatz, bei dem das Auslösen einer Aktion Side Effects erzeugt, die wiederum spezifische HTTP-Endpunkte aufrufen.

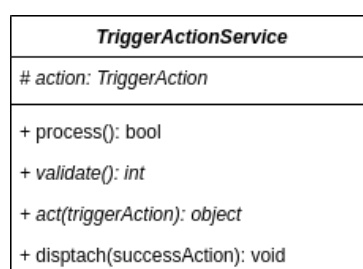


Abbildung 3.4: UML-Klassendiagramm des TriggerActionService

Die abstrakte Klasse `TriggerActionService` auf dem Server, dargestellt im UML-Klassendiagramm in Abbildung 3.4, ist zentral für dieses Konzept. Für jede `TriggerAction` sollte eine entsprechende Implementierung dieses Services existieren. In dieser Implementierung sind die abstrakten Methoden `validate()` und `act(triggerAction): object` zu definieren. Die Methode `validate()` analysiert den Payload der `TriggerAction` und gibt einen HTTP-Statuscode zurück, wobei der Code 200 (OK) eine erfolgreiche Validierung anzeigt. Die Methode `act` führt die notwendige Aktualisierung durch und liefert ein Objekt als Payload für die an den Client zu sendende `SuccessAction`.

Bei Eintreffen einer **TriggerAction** über den **dispatch**-Endpunkt auf dem Server wird die Methode `process(): boolean` der zugehörigen **TriggerActionService**-Implementierung aufgerufen. Algorithmus 3.2 veranschaulicht den Ablauf dieser Methode. Zunächst erfolgt eine Validierung der Anfrage. Bei einem Fehler wird der entsprechende HTTP-Statuscode zurückgegeben. Bei erfolgreicher Validierung wird die Aktualisierung durchgeführt, der Payload für die **SuccessAction** festgelegt und diese mittels **WebSocket** an alle Clients in der entsprechenden Gruppe versendet. Dieser Prozess ist in einem Try-Catch-Block eingebettet, um bei auftretenden Fehlern einen internen Serverfehler an den Client melden zu können.

---

**Algorithmus 3.2** Methode `process()` in der Klasse **TriggerActionService**

---

```
public process() {
    var status = validate();
    if(status != 200) {
        logError(status);
        return StatusCode(status);
    }

    try {
        var payload = act(this.action);
        this.action.success.payload = payload;
        dispatch(this.action.group, this.action.success);
        return Ok(payload);
    } catch(e) {
        logError(e);
        return InternalServerError();
    }
}
```

---

### 3.2.5 WebSocket Endpunkt `dispatch(successAction)`

Wie im vorherigen Abschnitt erläutert, benachrichtigt der Server Clients, die sich für eine durch eine **TriggerAction** definierte Gruppe registriert haben, mittels eines **WebSocket**-Aufrufs. Dieser Aufruf erfolgt über eine von den Clients bereitgestellte **WebSocket**-Methode, wobei die entsprechende **SuccessAction** (welche durch die **TriggerAction** definiert wurde) übergeben wird.

Auf der Clientseite wird in dieser Methode einfach die `dispatch`-Funktion des Redux-basierten Stores aufgerufen. Dies führt in der Regel dazu, dass eine Reducer-Funktion aktiviert wird, welche den Store entsprechend der erhaltenen **SuccessAction** aktualisiert. Es ist wichtig zu beachten, dass die Reducer-Funktion zusätzlich zur **SuccessAction** auf der Clientseite implementiert werden muss.

### 3.3 Analyse des Konsistenzverhaltens von YReduxSocket

Dieser Abschnitt konzentriert sich auf eine detaillierte Analyse des Konsistenzverhaltens von *YReduxSocket*. Zunächst wird vorgestellt wie in **YReduxSocket** Transaktionen eingesetzt werden um die ACID Eigenschaften zu erreichen. Danach, werden verschiedene Konsistenzmodelle, die auf der Clientseite anwendbar sind, schrittweise vorgestellt und untersucht. Ziel ist es, zu evaluieren, inwiefern *YReduxSocket* diesen Modellen entspricht oder Anpassungen erfordert.

#### 3.3.1 Transaktionen

Wie bereits in Abschnitt 2.3 erläutert, setzt der Einsatz von *YReduxSocket* eine Datenbank voraus, die Transaktionen mit ACID-Eigenschaften unterstützt.

Algorithmus 3.2 beschreibt die Verarbeitung von **TriggerActions** durch den Server. Während dieser Verarbeitung ist es entscheidend, dass der Aufruf der **act()**-Methode, welche die Datenbankmodifikationen durchführt, innerhalb einer Transaktion stattfindet. Dies gewährleistet, dass bei einer **SuccessAction** davon ausgegangen werden kann, dass die Änderungen atomar, konsistent, isoliert und dauerhaft erfolgt sind (siehe Abschnitt 2.3).

Um die Notwendigkeit der Transaktionsverarbeitung in jeder Implementierung der abstrakten Basisklasse **TriggerActionService** zu vermeiden, wird eine neue Methode **actBase(triggerAction): object** im **TriggerActionService** eingeführt. Diese Methode wird im Algorithmus 3.2 anstelle der **act()**-Methode aufgerufen. Algorithmus 2.1 aus Abschnitt 2.3 illustriert eine mögliche Implementierung dieser Methode.

#### 3.3.2 Monotones Lesen

Tanenbaum und van Steen definieren das clientbasierte Konsistenzmodell für monotone Lesevorgänge wie folgt:

„Wenn ein Prozess den Wert eines Datenelments  $x$  liest, gibt jede anschließende Leseoperation dieses Prozesses auf  $x$  stets denselben oder einen aktuelleren Wert zurück.“ [TS08, S. 322]

Wie bereits erwähnt, stützt sich *YReduxSocket* auf die Verwendung einer einzigen Datenbank, die als zentrale Wahrheitsquelle (Single Source of Truth) angesehen wird. Diese Datenbank wird ausschließlich durch Aktionen modifiziert, die innerhalb von Transaktionen stattfinden und die ACID-Prinzipien einhalten.

Eine Leseoperation in *YReduxSocket* wird durch eine **TriggerAction** und eine **SuccessAction** umgesetzt. Möchte ein Client eine Leseoperation initiieren, sendet er eine **TriggerAction** an den Server (z.B. **loadFile(id)**). Der Server liest daraufhin den Datensatz aus der Datenbank und sendet ihn mit einer **SuccessAction** (z.B. **loadFileSuccess(file)**) an

den anfragenden Client zurück. Die gelesenen Daten werden für den Client sichtbar, sobald er die **SuccessAction** erhält.

Die eingesetzten Transaktionen bei Schreibvorgängen gewährleisten, dass nur konsistente Datenwerte nach außen hin sichtbar werden. Dies entspricht der Konsistenzeigenschaft der ACID-Prinzipien, welche sicherstellt, dass inkonsistente Zwischenergebnisse während der Transaktionen verborgen bleiben. Somit sind die von einem Client gelesenen Werte stets konsistent. Zusätzlich sorgt die Dauerhaftigkeitseigenschaft der ACID-Prinzipien dafür, dass einmal geschriebene Daten beständig und zuverlässig gespeichert werden.

Wenn ein Client denselben Datensatz zu verschiedenen Zeitpunkten liest, garantieren die ACID-Prinzipien, dass immer konsistente und bestätigte Daten übermittelt werden. Der Client erhält zwei **SuccessActions**, wobei die zuletzt eintreffende die vorherige aktualisiert. Jedoch kann nicht garantiert werden, dass diese **SuccessAction** stets die aktuellere ist, da sich die Aktionen auf dem Weg vom Server zum Client überholen können. Das Konsistenzmodell des monotonen Lesens kann daher nicht vollständig sichergestellt werden. In den meisten Anwendungsfällen ist dies jedoch nicht erforderlich. Ändert ein Teilnehmer im System den gelesenen Datensatz, wird an alle Clients, die diese Daten gelesen hatten, eine aktualisierende **SuccessAction** gesendet, wodurch der Datensatz auf den neuesten Stand gebracht wird. Dieses Verhalten wird als „eventual consistency“ bezeichnet, was bedeutet, dass der Zustand des Clients über die Zeit mit dem Zustand in der Datenbank konvergiert [TS08, S. 319–321].

### 3.3.3 Monotones Schreiben

Zunächst die Definition des Konsistenzmodells laut Tannenbaum und van Steen:

„Eine Schreiboperation eines Prozesses an einem Datenelement  $x$  wird abgeschlossen, bevor eine folgende Schreiboperation auf  $x$  durch denselben Prozess erfolgen kann.“ [TS08, S. 323]

*YReduxSocket* kann das Prinzip des monotonen Schreibens nicht vollständig umsetzen. Wenn ein Client kurz hintereinander zwei **TriggerActions** sendet, die denselben Datensatz ändern sollen, ist die Reihenfolge der Ausführung nicht garantiert. Die zuerst ankommende Aktion wird angewendet, gefolgt von der nachfolgenden. Auch hier werden zwei **SuccessActions** versendet, wobei die zuletzt beim Client eintreffende Aktion den Store abschließend aktualisiert. Dieses Konsistenzverhalten ist für viele Anwendungsfälle ausreichend. Für Szenarien, die eine stärkere Konsistenz erfordern, kann *YReduxSocket* durch die Implementierung eines speziellen Algorithmus erweitert werden, wodurch die Konsistenzmodelle des monotonen Lesens und Schreibens erfüllt werden. Weitere Details dazu finden sich in Abschnitt 3.3.6.

### 3.3.4 Read Your Writes

Dieser Abschnitt stützt sich ebenfalls auf eine Definition von Tanenbaum und van Steen:

„Die Folge einer Schreiboperation eines Prozesses auf das Datenelement  $x$  wird für eine anschließende Leseoperation auf  $x$  durch denselben Prozess stets sichtbar sein.“ [TS08, S. 324]

Dank der bei Schreibvorgängen eingesetzten Transaktionen kann die ACID-Eigenschaft der Dauerhaftigkeit die Sichtbarkeit der Schreiboperationen für nachfolgende Leseoperationen garantieren. Wenn ein Client eine *TriggerAction* sendet, die eine Modifikation eines Datensatzes auslösen soll, wird diese Modifikation bei erfolgreicher Validierung innerhalb einer Transaktion durchgeführt. Ist der Vorgang erfolgreich, bleibt die Änderung dauerhaft in der Datenbank gespeichert. Jeder nachfolgende Lesevorgang, der auf diesen Datensatz zugreift, liefert als Ergebnis einen Wert, der auf dieser Transaktion basiert – entweder diesen Wert oder einen aktuelleren.

### 3.3.5 Write Follows Reads

Das abschließend untersuchte Konsistenzmodell basiert auch auf einer Definition von Tanenbaum und van Steen:

„Einer Schreiboperation eines Prozesses auf ein Datenelement  $x$ , die auf eine vorherige Leseoperation auf  $x$  durch denselben Prozess folgt, wird garantiert, dass sie auf demselben oder einem aktuelleren Wert von  $x$  stattfindet.“ [TS08, S. 326]

Auch diese Eigenschaft wird auch von *YReduxSocket* erfüllt. Wenn ein Wert aus der Datenbank gelesen wird, wird dieser mittels einem *dispatch* einer *SuccessAction* in den Client-Store geladen. Schreiboperationen basieren nun auf diesem Wert. Der Client sendet also einen modifizierten Wert an den Server, der auf dem zuvor geladenen Wert basiert. Auch hier unterstützt die Dauerhaftigkeitseigenschaft der ACID-Prinzipien die Konsistenz, da der Wert direkt aus der Datenbank geladen wurde. Daher kann zum Zeitpunkt des Eintreffens der *TriggerAction*, die eine Änderung auslösen soll, nur dieser Wert oder ein aktuellerer in der Datenbank vorhanden sein.

### 3.3.6 Haverbeke-Algorithmus

Eine zentrale Funktion der Plattform „yoshie.io“ ist der integrierte Editor, der kollaboratives Arbeiten in vollem Umfang unterstützt. Dieser Editor beruht auf dem Open-Source-Framework *ProseMirror*, welches von Marijn Haverbeke entwickelt wurde. *ProseMirror* bietet einen Entwicklerleitfaden, der einen Ansatz beschreibt, um einen kooperativ nutzbaren Editor zu implementieren [Hav24]. In diesem Leitfaden wird ein spezifischer Algorithmus detailliert dargelegt, der sowohl auf Server- als auch auf Clientseite umgesetzt wurde, um die hohen Anforderungen an den Editor zu erfüllen.

Wie in vorangegangenen Abschnitten diskutiert, unterstützt *YReduxSocket* die clientseitigen Konsistenzmodelle *Read Your Writes* und *Write Follows Read*. Allerdings ist die Einhaltung der Modelle *Monotones Lesen* und *Monotones Schreiben* nicht durchgängig sichergestellt. Für Anwendungsfälle, die eine strikte Konsistenz verlangen, lässt sich



*YReduxSocket* mittels einer Erweiterung anpassen. Diese stützt sich maßgeblich auf einen Blogbeitrag von Marijn Haverbeke [Hav15], in welchem der im Editor eingesetzte Algorithmus eingehend erläutert wird. Im Weiteren wird auf diese Erweiterung von *YReduxSocket* als Haverbeke-Algorithmus Bezug genommen.

## Der Algorithmus

---

**Algorithmus 3.3** Haverbeke-Algorithmus: serverseitige Erweiterung

---

```
public dispatchAction(action, version) {
    if(version!=triggerActions.length || lock()) return false;

    //Validierungslogik falls notwendig

    updateDatabase(action);
    triggerActions.append(action);

    clients.onNewAction();
    release();
    return true;
}

public actionsSince(version) {
    var actions = triggerActions.slice(version);
    return actions;
}
```

---

Die Erweiterung betrifft sowohl die Server- als auch die Clientseite. Algorithmus 3.3 beschreibt dabei den Serverteil. Der Einfachheit halber wird angenommen, dass der Server direkt auf eine Liste von `clients` zugreifen kann, indem er mittels Remote Procedure Call (RPC) mit den Clients kommuniziert. Weiterhin existiert auf dem Server eine Liste `triggerActions`, die alle Aktionen umfasst, die bereits zur Aktualisierung angewendet wurden. Zudem gibt es einen einfachen Locking-Mechanismus, repräsentiert durch die Methoden `lock()` und `release()`, der dazu dient, konkurrierende Zugriffe verschiedener Clients zu verhindern. Für die detaillierte Implementierung sei auf Kapitel 4 verwiesen.

Vom Server werden zwei Methoden bereitgestellt, die ein Client (beispielsweise via HTTP) aufrufen kann. Die Methode `dispatchAction(a, v)` erhält eine Aktion `a` und eine Versionsnummer `v` als Parameter und gibt einen booleschen Wert zurück. Dieser Rückgabewert informiert den aufrufenden Client darüber, ob die übergebene Aktion erfolgreich angewendet wurde. Zu Beginn der Methode überprüft der Server, ob die clientseitige Versionsnummer mit seiner eigenen übereinstimmt. Zusätzlich wird die Methode `lock()` verwendet, um zu prüfen, ob die Methode aktuell von einem anderen Client ausgeführt wird. Ist dies nicht der Fall, wird die Ausführung für andere Clients gesperrt. Kann die Aktion nicht angewendet werden, gibt die Methode `false` zurück. Im Erfolgsfall führt der Server Validierungen durch und passt die Datenbank mittels der übergebenen Aktion an. Anschließend benachrichtigt er alle Clients über die neue Aktion, hebt die Sperre auf und gibt `true` zurück.

Die zweite vom Server angebotene Methode trägt den Namen `actionsSince(version)`. Ein Client kann diese Methode nutzen, um alle Aktionen abzurufen, die eine spezifische Versionsnummer überschreiten.

---

**Algorithmus 3.4** Haverbeke-Algorithmus: clientseitige Erweiterung
 

---

```
public dispatch(action) {
    notDispatched.push(action);
    sendNotDispatchedToServer();
}

public onNewAction() {
    var actions = server.actionsSince(v);
    foreach(var action in actions) {
        store.dispatch(action);
        v++;
    }
    sendNotDispatchedToServer();
}

private sendNotDispatchedToServer() {
    if(!notDispatched.any()) return;
    var result = server.dispatchAction(v, notDispatched);
    if(result == true) {
        notDispatched = [];
    }
}
```

---

In Algorithmus 3.4 wird vorausgesetzt, dass jeder Client über eine Versionsnummer `v` verfügt, die seiner aktuellen geladenen Version entspricht. Zudem hat er Zugriff auf seinen State über ein Objekt `store` und kann Serveranfragen über ein Objekt `server` initiieren. Des Weiteren besitzt er eine Liste namens `notDispatched`, die alle Aktionen umfasst, die vom Client initiiert, aber noch nicht vom Server bestätigt wurden.

Auf dem Client existiert die Methode `dispatch(action)`, die eine Aktion entgegennimmt. Zuerst wird die Aktion in die Liste `notDispatched` eingefügt. Anschließend wird die private Methode `sendNotDispatchedToServer()` aufgerufen. Diese Methode prüft zunächst, ob nicht versandte Aktionen vorhanden sind. Falls solche Aktionen existieren, werden sie zusammen mit der aktuellen Versionsnummer an die Servermethode `dispatchAction` übermittelt. Der vom Server zurückgegebene Wert wird daraufhin ausgewertet. Wenn der Server die übermittelten Aktionen erfolgreich angewendet hat, wird die Liste `notDispatched` geleert.

Zusätzlich existiert noch die Methode `onNewAction()`, die vom Server jedes Mal aufgerufen wird, wenn eine neue Aktion erfolgreich angewendet werden konnte. Diese Methode ermöglicht es dem Client, zunächst alle Aktionen vom Server abzurufen, die noch nicht auf seinen lokalen Store angewendet wurden. Anschließend wendet der Client diese Aktionen nacheinander auf seinen Store an und inkrementiert mit jeder angewendeten Aktion seine Versionsnummer um 1.

### Ausführungsbeispiele

In diesem Abschnitt werden anhand zweier Beispiele die Funktionsweise und Anwendbarkeit des Haverbeke-Algorithmus detailliert dargestellt. Die Beispiele sollen die praktische Umsetzung des Algorithmus in unterschiedlichen Szenarien veranschaulichen und dabei helfen, die zugrundeliegenden Konzepte und Prozesse besser zu verstehen.

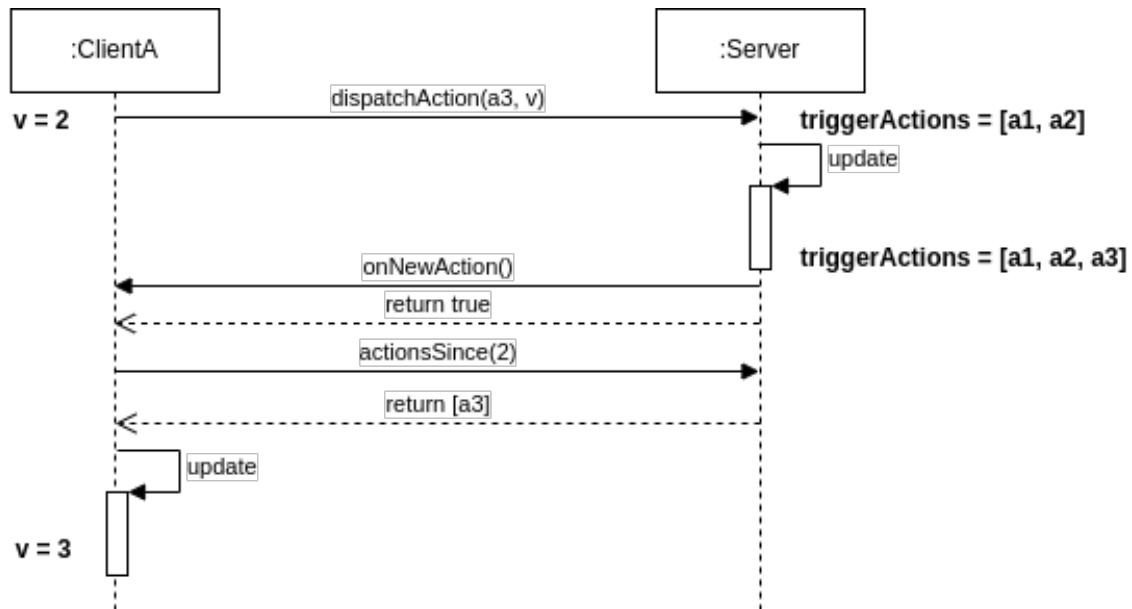


Abbildung 3.5: Haverbeke-Algorithmus ein Client

Abbildung 3.5 illustriert die Ausführung des Haverbeke-Algorithmus in einem Szenario mit einem einzelnen Client. Zu Beginn hält der Client eine Versionsnummer von 2, was darauf hinweist, dass beide, Client und Server, auf demselben Informationsstand sind, da auch das **triggerActions** Array auf dem Server zwei Einträge enthält. Möchte der Client nun eine weitere Action anwenden, sendet er diese zusammen mit seiner aktuellen Versionsnummer an den Server. Der Server vergleicht die übermittelte Versionsnummer mit seinem Stand und stellt fest, dass keine Diskrepanz vorliegt, woraufhin er die Action erfolgreich anwendet. Im Anschluss benachrichtigt der Server den Client mittels der Methode **onNewAction()**, dass eine neue Action verfügbar ist. Der Client ruft daraufhin mittels **actionsSince(version)** alle ihm unbekannten Actions vom Server ab und erhält die zuvor gesendete Action. Nachdem der Client diese Action auf seinen Store angewendet hat, inkrementiert er seine Versionsnummer entsprechend.

In Abbildung 3.6 wird ein komplexerer Prozess dargestellt, bei dem zwei Clients, nämlich **ClientA** und **ClientB**, versuchen, fast zeitgleich jeweils eine Aktion zu initiieren. Der von **ClientA** ausgelöste Ablauf ist in Blau und der von **ClientB** initiierte Ablauf in Grün markiert.

Zu Beginn befinden sich alle Beteiligten auf demselben Informationsstand. Beide Clients senden dann nahezu gleichzeitig eine neue Aktion an den Server. Da die Aktion von **ClientA** zuerst verarbeitet wird, wird die Aktion von **ClientB** aufgrund des Sperrmechanismus zurückgewiesen, und der Aufruf liefert **false** zurück. Der Ablauf für **ClientA** wird analog zum ersten Beispiel verarbeitet, mit dem Unterschied, dass auch **ClientB** eine Benachrichtigung über die Methode **onNewAction()** vom Server erhält und seinen Store

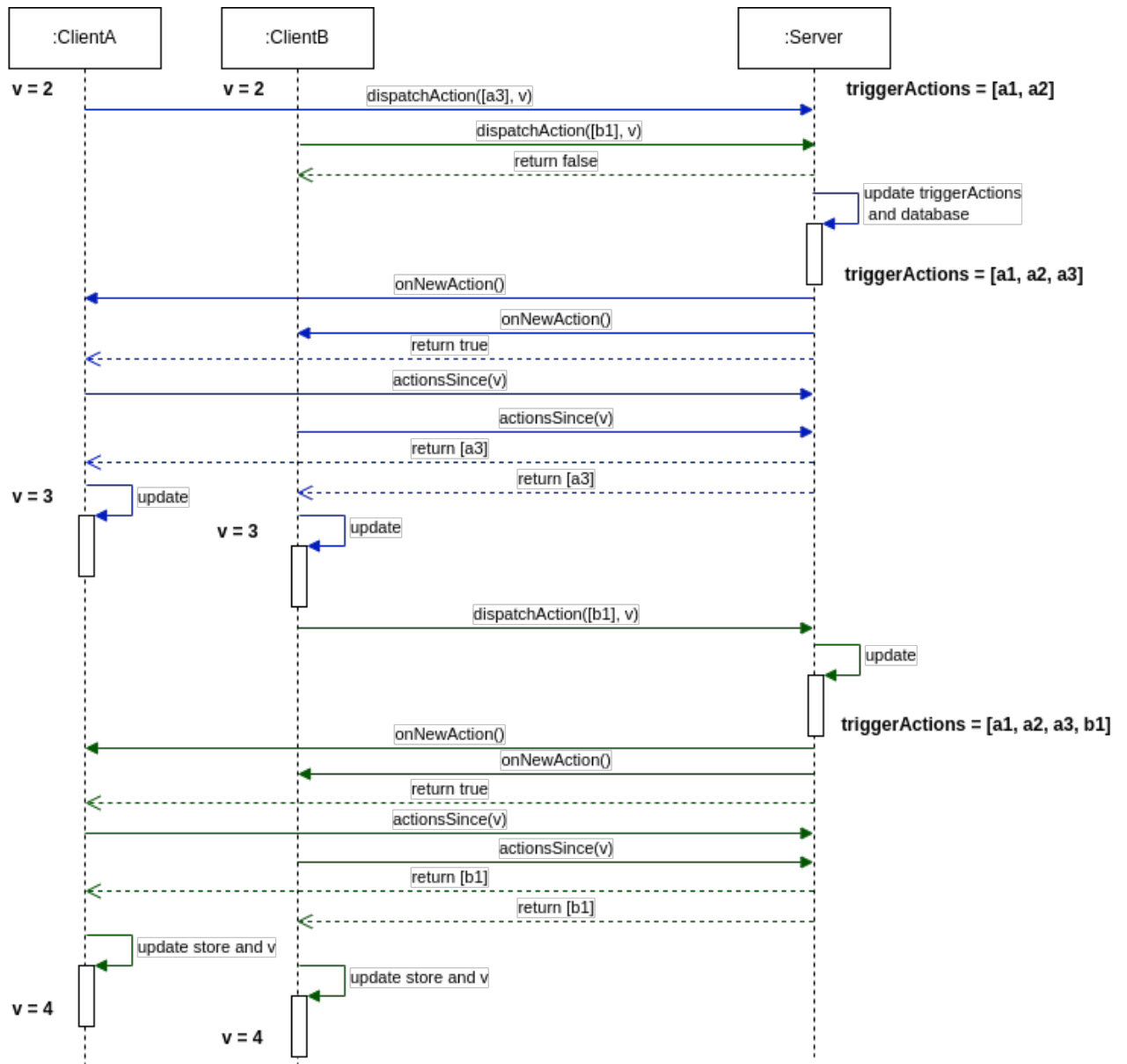


Abbildung 3.6: Haverbeke-Algorithmus zwei Clients

entsprechend aktualisiert. Nach Abschluss der Aktion von `ClientA` sind alle Teilnehmer wieder auf dem gleichen Informationsstand.

Sobald `ClientB` seinen Store aktualisiert hat, versendet er seine Aktion erneut an den Server. Diesmal wird die Aktion von `ClientB` erfolgreich angewendet, da keine Konflikte mehr vorliegen und die Versionsnummern übereinstimmen. Der Algorithmus stellt erneut sicher, dass alle Teilnehmer ihre Informationen aktualisieren und letztendlich denselben Versionsstand erreichen, in diesem Fall Version 4.

Deine Beschreibung des komplexeren Szenarios mit zwei Clients, die nahezu gleichzeitig Aktionen an den Server senden, ist sehr detailliert und veranschaulicht gut, wie der Haverbeke-Algorithmus in einem solchen Fall funktioniert. Die Erklärung zeigt, wie der Algorithmus Konsistenz gewährleistet und alle Teilnehmer synchron hält. Hier sind einige Vorschläge, wie du den Text eventuell noch verfeinern könntest:

In Abbildung 3.6 wird ein komplexerer Prozess dargestellt, bei dem zwei Clients, nämlich `ClientA` und `ClientB`, versuchen, fast zeitgleich jeweils eine Aktion zu initiieren. Der von `ClientA` ausgelöste Prozess ist in Blau und der von `ClientB` initiierte Prozess in Grün markiert.

Zu Beginn befinden sich alle Beteiligten auf demselben Informationsstand. Beide Clients senden dann nahezu gleichzeitig eine neue Aktion an den Server. Da die Aktion von `ClientA` zuerst verarbeitet wird, wird die Aktion von `ClientB` aufgrund des Sperrmechanismus zurückgewiesen, und der Aufruf liefert `false` zurück. Der Ablauf für `ClientA` wird analog zum ersten Beispiel verarbeitet, mit dem Unterschied, dass auch `ClientB` eine Benachrichtigung über die Methode `onNewAction()` vom Server erhält und seinen Store entsprechend aktualisiert. Nach Abschluss der Aktion von `ClientA` sind alle Teilnehmer wieder auf dem gleichen Informationsstand.

Sobald `ClientB` seinen Store aktualisiert hat, versendet er seine Aktion erneut an den Server. Diesmal wird die Aktion von `ClientB` erfolgreich angewendet, da keine Konflikte mehr vorliegen und die Versionsnummern übereinstimmen. Der Algorithmus stellt erneut sicher, dass alle Teilnehmer ihre Informationen aktualisieren und letztendlich denselben Versionsstand erreichen, in diesem Fall Version 4.

Dieses Beispiel demonstriert, wie der Algorithmus nicht nur automatisch alle Teilnehmer auf denselben Informationsstand bringt, sondern auch sicherstellt, dass alle Aktionen systemweit in derselben Reihenfolge angewendet werden. Der Einsatz dieses Algorithmus bietet den Vorteil einer starken Konsistenz im System. Darüber hinaus ermöglicht der Algorithmus eine einfache Implementierung von Datenversionierung oder einer Undo-History. Als Nachteile sind der erhöhte Implementierungsaufwand und der gesteigerte Speicherbedarf zu nennen.

## 3.4 Vergleich mit herkömmlichen Ansatz

Basierend auf Abschnitt 2.10 und den Abschnitten in diesem Kapitel, wird im folgenden versucht einen detaillierten Vergleich der beiden Ansätze erstellen. Das Ziel dieses Vergleichs ist es, die Unterschiede, Vor- und Nachteile sowie die jeweiligen Einsatzgebiete beider Technologien herauszuarbeiten.

## Konzept

Der herkömmliche Ansatz basiert auf der Nutzung separater Actions für Side Effects, die einen Serveraufruf nach sich ziehen, und der Implementierung spezifischer HTTP- und WebSocket-Endpunkte für jede neue Funktionalität. Dies erfordert eine umfangreiche Implementierungsarbeit, insbesondere bei der Definition neuer Actions, der Validierungslogik und der Aktualisierungslogik sowohl auf Server- als auch auf Clientseite.

*YReduxSocket* vereinfacht diesen Prozess durch die Definition von Actions auf Server- und Clientseite und die Nutzung eines einzigen HTTP-Endpunkts zum Empfangen und Verarbeiten dieser Actions. Serverseitig entscheidet die Kennzeichnung der Action über die erforderlichen Validierungen oder Datenbankmodifikationen. Die Rückübermittlung erfolgt über einen festgelegten WebSocket-Endpunkt, was die Notwendigkeit zusätzlicher Endpunkte für neue Features eliminiert.

## Implementierungsaufwand

Beim herkömmlichen Ansatz müssen für jede neue Funktionalität mehrere Schritte wiederholt werden, was den Implementierungsaufwand erhöht und zu einer komplexeren Codebasis führt.

*YReduxSocket* reduziert den Implementierungsaufwand erheblich, da lediglich zwei Actions definiert und die serverseitige Logik innerhalb eines einzigen Endpunkts implementiert werden muss. Dies führt zu einer schlankeren und wartungsfreundlicheren Codebasis.

## Konsistenzverhalten

Bei *YReduxSocket* wird ein besonderer Fokus auf das Konsistenzverhalten gelegt. Durch die Verwendung von Transaktionen und die Einhaltung von Konsistenzmodellen wie „Read Your Writes“ und „Write Follows Reads“ wird besonders auf die Konsistenz geachtet. Der Einsatz des Haverbeke-Algorithmus ermöglicht es, stärkere Konsistenzmodelle wie „Monotones Lesen“ und „Monotones Schreiben“ zu unterstützen, was für Anwendungsfälle mit hohen Konsistenzanforderungen vorteilhaft ist.

## Einsatzgebiete

Der herkömmliche Ansatz ist geeignet für Anwendungen mit geringerem Komplexitätsgrad und weniger häufigen Datenänderungen, wo der Mehraufwand für die Implementierung spezifischer Endpunkte vertretbar ist.

Dahingegen ist **YReduxSocket** ideal für komplexe Anwendungen mit häufigen Datenänderungen und hohen Konsistenzanforderungen. Durch die Vereinfachung der Implementierung und die Unterstützung starker Konsistenzmodelle eignet sich *YReduxSocket* besonders für Echtzeitanwendungen und kollaborative Plattformen.

**Fazit**

Der Vergleich zeigt, dass YReduxSocket signifikante Vorteile gegenüber dem herkömmlichen Ansatz bietet, insbesondere in Bezug auf den Implementierungsaufwand und das Konsistenzverhalten. Durch die Vereinfachung des Entwicklungsprozesses und die Unterstützung starker Konsistenzmodelle ermöglicht YReduxSocket die Entwicklung effizienter und zuverlässiger Echtzeitanwendungen. Der herkömmliche Ansatz kann jedoch in einfacheren Szenarien oder in Fällen, in denen die Implementierung spezifischer Endpunkte keine Einschränkung darstellt, nach wie vor eine valide Option sein.





# Kapitel 4

## Implementierung und Tests

### 4.1 Software Architektur

### 4.2 Client: Angular

### 4.3 Server: ASP.Net

### 4.4 Vergleich mit herkömmlichen Ansatz



# Kapitel 5

## Fazit

### 5.1 Zusammenfassung

### 5.2 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: 978-1-4842-3988-9. 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).
- [CC01] Diletta Cacciagrano und Flavio Corradini. „On Synchronous and Asynchronous Communication Paradigms“. en. In: *Theoretical Computer Science*. Hrsg. von Antonio Restivo, Simona Ronchi Della Rocca und Luca Roversi. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2001, S. 256–268. ISBN: 978-3-540-45446-5. DOI: 10.1007/3-540-45446-2\_16.
- [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](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).
- [Hav24] Marijn Haverbeke. *ProseMirror Guide*. 2024. URL: <https://prosemirror.net/docs/guide/#collab> (besucht am 07.02.2024).
- [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: 978-3-86490-552-0.
- [McF19] Timo McFarlane. „Managing State in React Applications with Redux“. en. In: (2019). URL: [https://www.theseus.fi/bitstream/handle/10024/265492/McFarlane\\_Timo.pdf](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.