Abschlussarbeit am LG Kooperative Systeme der FernUniversität in Hagen

YReduxSocket: Erhaltung der Konsistenz in webbasierten Anwendungen

Ricardo Stolzlechner

9463470

ricardo.stolzlechner@gmail.com

Informatik Bachelor of Science

Betreuerin: Dr. Lihong Ma

18. Oktober 2023

Abstract

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

1.1 Hintergrund und Motivation

Die Webanwendung „Yoshie.io“[1] ist eine kollaborative Plattform, spezialisiert auf das Baugewerbe, bei der der Autor dieser Arbeit als technischer Leiter tätig ist. Während der Entwicklung dieser Anwendung wurde deutlich, dass eine der zentralen Herausforderungen darin besteht, die Konsistenz der angezeigten Daten sicherzustellen. Diese Anforderung kann in zwei Hauptaspekte unterteilt werden, die verschiedene technologische Ansätze erfordern. Zum einen ist es erforderlich, die Datenkonsistenz zwischen den verschiedenen Komponenten innerhalb desselben Clients zu gewährleisten. Zum anderen muss sichergestellt werden, dass Datenänderungen zwischen den verbundenen Clients synchronisiert werden. In diesem Zusammenhang wurden zwei Technologien, nämlich ein auf Redux basierender Datenstore und das WebSocket-Kommunikationsprotokoll, als Lösungen zur Bewältigung dieser Herausforderungen eingesetzt.

Es zeigte sich, dass der Aufwand bezüglich der Implementierung und Wartung der Anwendung im Hinblick auf die Konsistenzerhaltung zunehmend stieg. Daher wurde ein Konzept entwickelt, das es ermöglichte, beide Technologien zu verbinden und so den Entwicklungsaufwand erheblich zu verringern. Dieses Konzept erhielt intern den Namen *YReduxSocket*. Im Zuge dieser Arbeit soll das erwähnte Konzept eingehend betrachtet und spezifiziert werden.

1.2 Problemstellung

Wie bereits im Abschnitt 1.1 erwähnt, ergeben sich im Zusammenhang mit der Konsistenzerhaltung in Webanwendungen zwei unterschiedliche Problemstellungen. Um diese beiden Probleme besser zu verstehen, werden sie in den folgenden Abschnitten näher erläutert.

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

1.2.1 Datenkonsistenz am selben Client

JavaScript-Frameworks wie *React* oder *Angular* basieren auf einer komponentenbasierten Architektur. Dabei repräsentiert jede Komponente ein UI-Element, das an verschiedenen Stellen innerhalb der Anwendung dargestellt werden kann. Diese Komponenten können zusätzliche Kindkomponenten einschließen, wodurch ein Komponentenbaum entsteht. In typischen Webanwendungen existieren zahlreiche verschiedene Komponenten, und die Daten, die von diesen Komponenten angezeigt werden, werden von übergeordneten Elternkomponenten an die entsprechenden Kindkomponenten als Eingabeparameter übergeben.

Es ist nicht ungewöhnlich, dass verschiedene Komponenten dieselben Daten anzeigen sollen. In solchen Fällen obliegt es dem Anwendungsprogrammierer, sicherzustellen, dass die Daten zwischen diesen Komponenten synchronisiert werden. Aufgrund der Notwendigkeit einer lose gekoppelten Architektur, bei der Kindkomponenten keine direkte Kenntnis von ihren Elternkomponenten haben, stellt die Synchronisierung dieser Daten keine triviale Aufgabe dar.

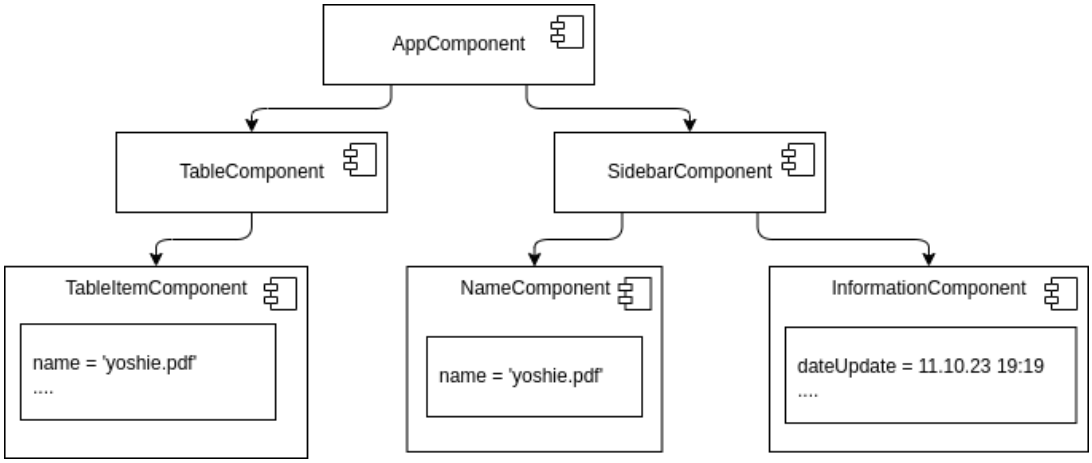

Abbildung 1.1: UML-Diagramm eines einfachen Komponentenbaums

In Abbildung 1.1 ist ein einfacher Komponentenbaum dargestellt, und der zugehörige Bildschirmausschnitt dieses Baums ist in Abbildung 1.2 zu sehen. Die Komponenten `TableItemComponent` und `NameComponent` zeigen jeweils dasselbe Datenfeld an. In der `NameComponent` kann der Nutzer den Namen ändern. Bei einer Änderung muss die Anwendungsprogrammiererin nun sicherstellen, dass die Information auch bei der `TableItemComponent` ankommt. Zudem muss das Aktualisierungsdatum, das in der `InformationComponent` dargestellt ist, angepasst werden.

Es gibt unterschiedliche Ansätze, um die Synchronisierung zu ermöglichen. Diese Ansätze werden in den folgenden Kapiteln näher beleuchtet und miteinander verglichen.

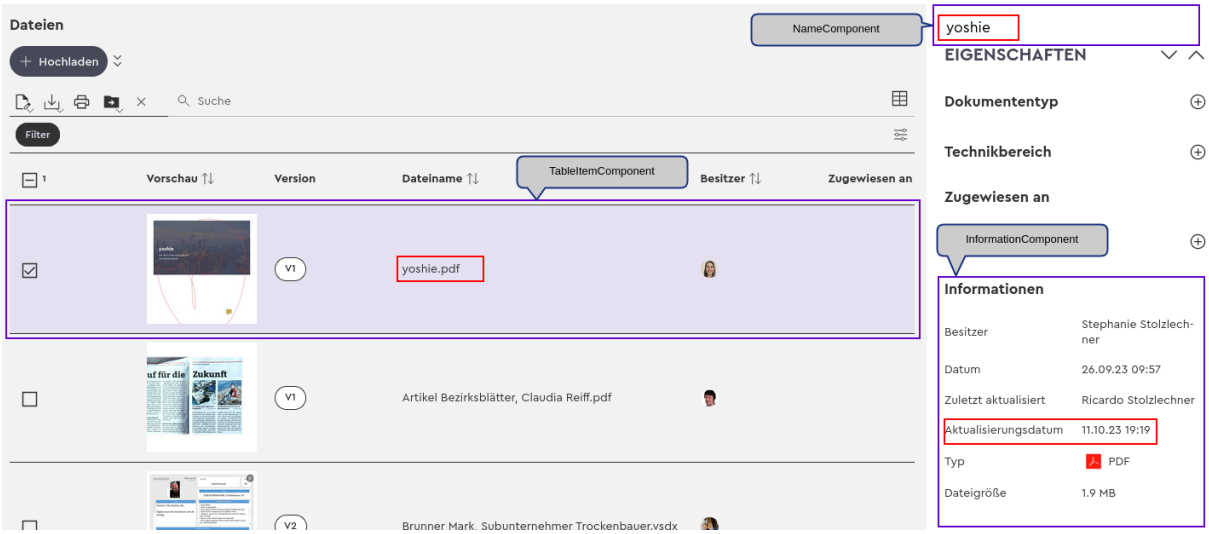

Abbildung 1.2: Bildschirmausschnitt eines einfachen Komponentenbaums

1.2.2 Datenkonsistenz zwischen verschiedenen Clients

In einer Webanwendung können verschiedene Clients mit denselben Daten arbeiten. Wenn beispielsweise zwei Benutzer gleichzeitig die Ansicht aus Abbildung 1.2 geladen haben und einer der Benutzer über die `NameComponent` den Namen ändert, muss sichergestellt werden, dass die Änderung auch beim anderen Benutzer angezeigt wird. Daher ist es notwendig, einen Mechanismus zu finden, um bei Datenänderungen alle beteiligten Teilnehmer synchron zu halten. Auch für dieses Problem gibt es verschiedene Ansätze, die in späteren Abschnitten vorgestellt werden.

1.2.3 YReduxSocket

Wie erwähnt gibt es verschiedene Ansätze, um die beiden beschriebenen Probleme zu lösen. Mit *YReduxSocket* wird ein Konzept eingesetzt, das versucht, beide Konsistenzanforderungen zu vereinheitlichen und gemeinsam zu lösen. Dabei wird auf jedem Client ein auf Redux basierter Datenstore eingesetzt. Diese Stores werden dann mithilfe des WebSocket-Kommunikationsprotokolls, das von einer zentralen Serverinstanz verwaltet wird, synchron gehalten. Bei der Entwicklung von *YReduxSocket* wurde besonders auf die Entwicklerfreundlichkeit geachtet. Das Konzept sorgt dafür, dass bei Funktionserweiterungen, die notwendigen Änderungen, die das Konsistenzverhalten betreffen, so gering wie möglich ausfallen. Dadurch können Entwicklungsfehler minimiert werden, was sich positiv auf das Konsistenzverhalten von Webanwendungen auswirkt.

1.3 Struktur der Arbeit

Zunächst werden in Kapitel 2 die Grundlagen vorgestellt, die notwendig sind, um der Arbeit im weiteren Verlauf folgen zu können. Dabei wird auf Single Page Applications, die komponentenbasierte Webentwicklung, den zentralen Redux-basierten Datenstore und

das WebSocket-Kommunikationsprotokoll eingegangen. Weiters wird dort definiert welche Voraussetzungen gegeben sein müssen um *YReduxSocket* einsetzen zu können. In einem abschließenden Abschnitt wird ein erster Ansatz vorgestellt, um einen Datenstore mithilfe von WebSocket-Nachrichten synchron zu halten.

Anschließend werden in Kapitel 3 verwandte Arbeiten und Konzepte aufgelistet und näher analysiert, die im Rahmen einer Literaturrecherche und auf Open-Source-Plattformen gefunden wurden.

Kapitel 4 widmet sich dem Konzept *YReduxSocket*. Zunächst wird das Konzept auf generische Weise vorgestellt und spezifiziert. Abschnitt 4.2 geht dann genauer auf das Konsistenzverhalten von *YReduxSocket* ein. Das Kapitel endet mit Vergleichen von *YReduxSocket* mit den Ansätzen, die in Abschnitt 2.5 und in Kapitel 3 vorgestellt wurden.

Daraufhin werden in Kapitel 5 verschiedene Implementierungen von *YReduxSocket* behandelt und vorgestellt. In Abschnitt 5.1 wird erläutert, wie *YReduxSocket* mit den Frameworks *Angular* und *ASP.NET* eingesetzt werden kann. Abschnitt 5.2 behandelt die Frameworks *React* und *Node.js*. Das Kapitel schließt mit einem Abschnitt, in dem vorgestellt wird, wie *YReduxSocket* auf gängigen Paketplattformen (im Detail *npm* und *NuGet*) eingesetzt wird.

Die Arbeit schließt mit Kapitel 6. Dort werden die erzielten Ergebnisse zusammengefasst. Außerdem wird die Lösung einer Validierung unterzogen, und potenzielle Verbesserungen des Konzepts werden aufgelistet.

Kapitel 2

Grundlagen

In diesem Kapitel werden zunächst einige grundlegende Konzepte erläutert, um eine bessere Grundlage für das Verständnis des Rests dieser Arbeit zu schaffen. Hierbei werden die Voraussetzungen definiert, die erfüllt sein müssen, um *YReduxSocket* erfolgreich einsetzen zu können.

2.1 Single Page Applications (SPAs)

Single Page Applications (SPAs) zeichnen sich dadurch aus, dass die gesamte Anwendung im Webbrowser des Benutzers ausgeführt wird. Um eine SPA zu nutzen, wird sie von einem Webserver bereitgestellt. Dieser Webserver enthält eine `index.html` Datei, die ausgeliefert wird, sobald die URL unter die der Webserver verfügbar ist in einem Webbrowser aufgerufen wird. In dieser `index.html` Datei befindet sich ein `script` Tag, das auf den erforderlichen JavaScript-Code verweist, der für die Ausführung der Anwendung notwendig ist. Dieser JavaScript-Code wird vom Webbrowser heruntergeladen und ausgeführt. Nach diesem anfänglichen Ladevorgang hat der Webserver nur noch die Aufgabe, statische Ressourcen wie Schriftarten oder Bilder nachzuladen. Die zur Darstellung benötigten HTML-Elemente werden ausschließlich über JavaScript generiert und aktualisiert. Es gibt verschiedene etablierte Frameworks, die für die Entwicklung von SPAs verwendet werden, darunter *Angular*, *React* und *Vue*. Diese Frameworks abstrahieren die DOM-API des Webbrowsers und vereinfachen so die Entwicklung. Im Gegensatz dazu stehen klassische Webanwendungen, bei denen der HTML-Code vom Webserver generiert und ausgeliefert wird. Dies wird auch als serverseitiges Rendering bezeichnet [HZ20, S. 1].

YReduxSocket setzt eine Single Page Application voraus. Außerdem wird eine separate Serverinstanz benötigt, die von den Clients verwendet wird, um Datensätze abzurufen und mit diesen zu arbeiten. Diese Serverinstanz stellt eine HTTP-API und die Möglichkeit zur Initialisierung einer WebSocket-Kommunikation zur Verfügung. Jegliche Kommunikation zwischen den Clients erfolgt über diese Serverinstanz. Diese beschriebene Serverinstanz ist ein eigenständiger Prozess und sollte nicht mit dem zuvor beschriebenen Webserver verwechselt werden. Im weiteren Verlauf wird mit dem Begriff „Server“ immer auf diese separate Serverinstanz Bezug genommen. Die in Abbildung 2.1 dargestellte Struktur zeigt die von YReduxSocket vorausgesetzte Prozess Architektur, wobei jedes Rechteck einen

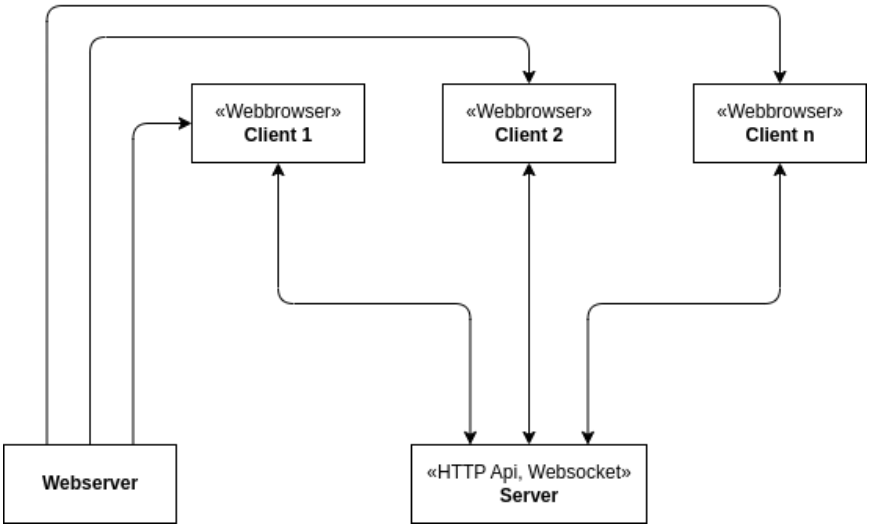

Abbildung 2.1: Prozess Architektur, die von YReduxSocket vorausgesetzt wird

eigenständig laufenden Prozess darstellt.

2.2 Komponentenbasierte Webentwicklung

Wie bereits im Abschnitt 2.1 erwähnt, wird der Aufbau von Single Page Applications durch den Einsatz von Frameworks vereinfacht. All diese Frameworks basieren auf einer komponentenbasierten Softwarearchitektur. Eine Komponente lässt sich grundsätzlich in zwei Teile gliedern. Der erste Teil ist in einer Art HTML definiert und beschreibt die Darstellung der Komponente im Browser. Der zweite Teil wird mit JavaScript oder TypeScript (eine typisierte Variante von JavaScript) definiert und beschreibt die logische Funktionalität der Komponente. Hier können beispielsweise Methoden definiert werden, die bei Nutzerinteraktionen ausgeführt werden.

Jede Komponente kann im HTML-Teil weitere Unterkomponenten enthalten. Ausgehend von einer Hauptkomponente entsteht so ein Komponentenbaum, wie er in Abbildung 1.1 dargestellt ist. Die eingesetzten Komponenten stehen also in einer Eltern-Kind-Beziehung.

In Webanwendungen gibt es in der Regel viele Komponenten, die miteinander kommunizieren müssen. Hierfür bieten Frameworks verschiedene Datenbindungsmechanismen. Eine Elternkomponente kann einen Datensatz über einen Eingabeparameter an die Kindkomponente übergeben. Zusätzlich können Kindkomponenten Ausgabeparameter bereitstellen, auf die sich eine Elternkomponente registrieren kann. Die Kindkomponenten haben die Möglichkeit, die Ausgabeparameter zu aktualisieren, wodurch automatisch alle Komponenten benachrichtigt werden, die sich zuvor darauf angemeldet haben.

Abbildung 2.2 zeigt, wie der Datenfluss im Komponentenbaum aussieht, wenn die genannten Datenbindungsmechanismen verwendet werden. Wenn sich beispielsweise der Name

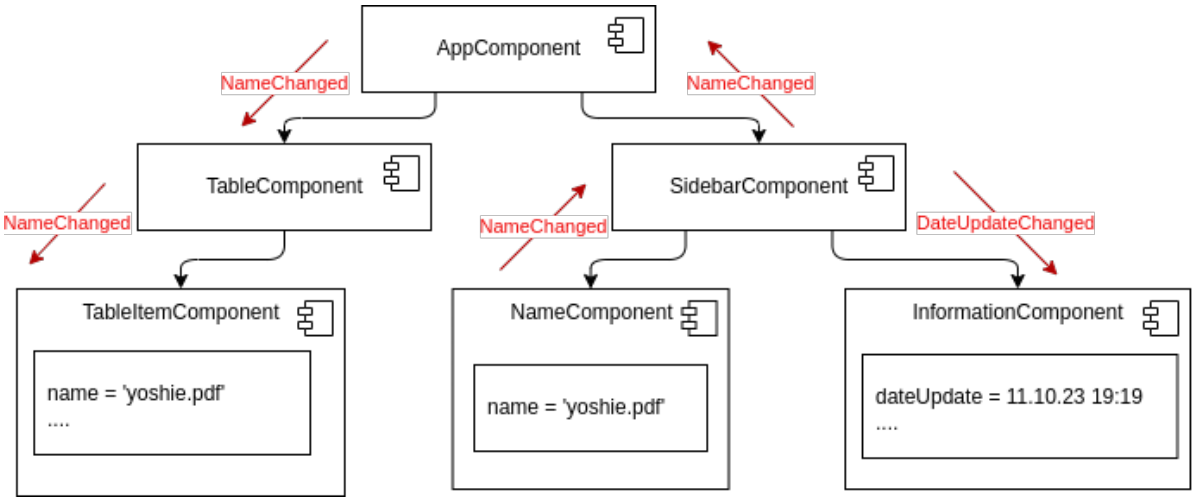

Abbildung 2.2: Datenfluss im Komponentenbaum, bei Verwendung von Datenbindungsmechanismen

in der Komponente `NameComponent` ändert, muss diese Komponente einen Ausgabeparameter auslösen, auf den sich die übergeordnete `SidebarComponent` registriert hat. Diese wiederum muss ebenfalls einen Ausgabeparameter haben, um die `AppComponent` zu benachrichtigen. Zudem aktualisiert sie einen Eingabeparameter, der an die `InformationComponent` meldet, dass sich das Aktualisierungsdatum geändert hat. Die Hauptkomponente muss daraufhin ihren Eingabeparameter für die `TableComponent` aktualisieren, die wiederum ihren Eingabeparameter an die `TableItemComponent` weiterreicht. Abschließend kann die `TableItemComponent` ihren angezeigten Wert aktualisieren.

Insbesondere bei größeren Komponentenbäumen, beispielsweise wenn die gemeinsame Wurzelkomponente weiter entfernt ist als im Beispiel oder wenn es noch mehr Komponenten gibt, die den Namen anzeigen sollen, stößt das beschriebene Verfahren an seine Grenzen. Hier bietet sich ein alternatives Konzept an, um Komponenten untereinander synchron zu halten. Bei diesem Konzept werden Services verwendet. Ein Service ist eine JavaScript-Klasse, die einmalig im Programm initialisiert wird und global für alle Komponenten verfügbar ist. Die Frameworks bieten verschiedene Mechanismen, um dies zu ermöglichen.

In dem in Abbildung 2.3 dargestellten Beispiel wird ein `NameService` verwendet. Da dieser global verfügbar ist, können die beteiligten Komponenten auf den Service zugreifen. Der Service bietet eine Methode `nameChanged`, die bei Änderungen von der `NameComponent` aufgerufen werden kann. Als Parameter wird der neue Komponentenname übergeben. In dieser Methode löst der Service die Events `nameChanged` und `dateUpdateChanged` aus. Wenn sich die `TableItemComponent` und die `InformationComponent` zuvor auf diese Events registriert haben, werden sie entsprechend benachrichtigt und können ihre angezeigten Werte aktualisieren.

Bei der Entwicklung mit Services besteht jedoch immer noch das Problem der doppelten Datenhaltung. Im beschriebenen Beispiel ist die Zeichenkette `name` sowohl in der `TableItemComponent` als auch in der `NameComponent` vorhanden. Dieses Problem kann vermieden werden, wenn ein zentraler Datenstore auf Basis von Redux verwendet wird.

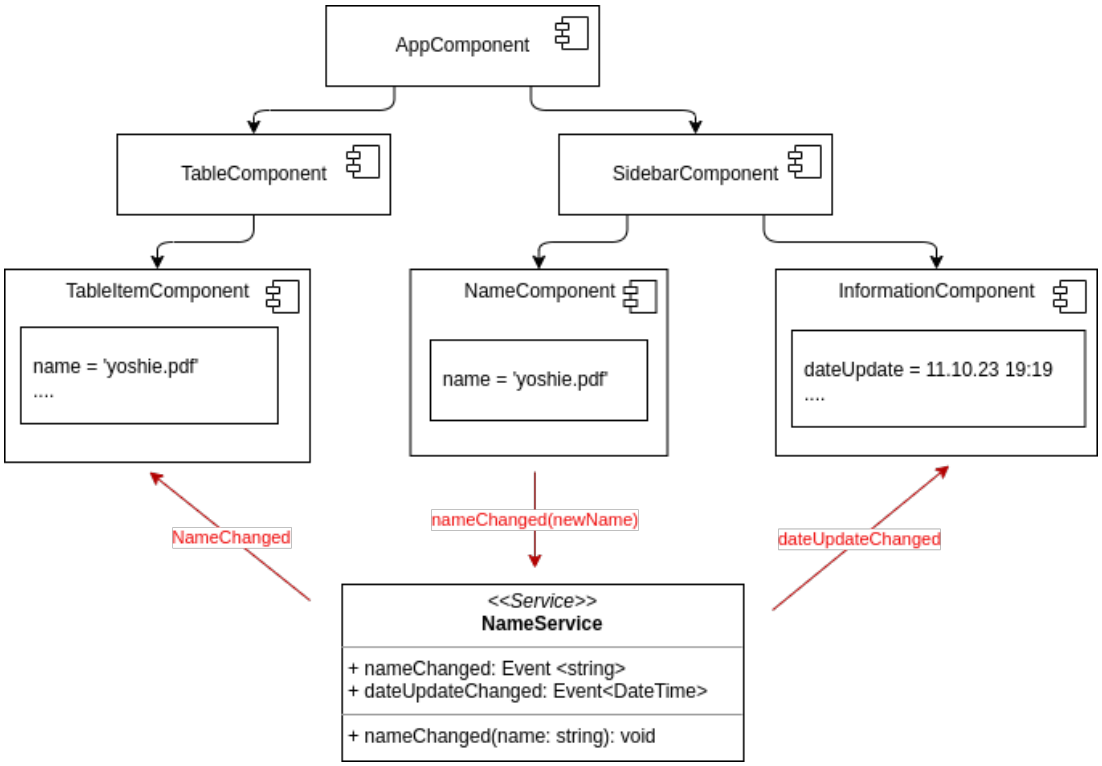

Abbildung 2.3: Datenfluss im Komponentenbaum, bei Verwendung eines Service

Auf dieses Konzept wird im nächsten Abschnitt näher eingegangen.

2.3 Zentraler Redux-Datenstore

In der Entwicklung von Webanwendungen werden häufig komponentenbasierte JavaScript-Frameworks eingesetzt [Sak19, S. 1]. Mit zunehmender Anzahl der verwendeten Komponenten steigt auch die Komplexität des zugrundeliegenden Codes. Oft ist es erforderlich, dass die Komponenten Daten untereinander austauschen, um auf dem aktuellen Stand zu bleiben. Um die durch den Datenaustausch entstehende Komplexität zu reduzieren, kann ein zentraler Redux basierter-Store verwendet werden. Dieser verwaltet den Zustand der Anwendung und kann als einzige Informationsquelle der anzuzeigenden Daten betrachtet werden. Der Store bietet darüber hinaus nicht nur den Vorteil einer verbesserten Leistung, sondern erleichtert auch die Testbarkeit [Far17, S. 31–32].

In einem Datenstore, der auf Redux basiert, können Komponenten mithilfe von Selektoren die für sie relevanten Daten abrufen. Wenn es zu gewünschten Datenänderungen kommt, beispielsweise aufgrund einer Nutzerinteraktion, können diese Daten nicht direkt manipuliert werden. Stattdessen wird eine sogenannte Action an den Datenstore gesendet. Eine Action ist im Wesentlichen eine Datenstruktur, die aus einer Parametersignatur und einer Identifikationszeichenfolge besteht. Das Senden der Action kann entweder bewirken, dass der Reducer aktiviert wird oder dass ein Seiteneffekt ausgelöst wird. Der Reducer führt eine Funktion aus, die den Datenstore aktualisiert. Mithilfe von Seiteneffekten kann bei Erhalt einer Action eine weitere Aktion ausgelöst werden, wie beispielsweise der Aufruf

einer Websocket-Methode [Tha20, S. 117–119].

Wird ein Redux-Store konsequent eingesetzt, kann gewährleistet werden, dass die Daten in einer konsistenten Art auf allen eingesetzten Komponenten angezeigt werden. Der gesamte Zustand der Anwendung ist in einem einzigen unveränderlichen (immutable) JavaScript-Objekt abgebildet, und Änderungen können ausschließlich durch das Auslösen (dispatchen) einer Action vorgenommen werden.

2.4 WebSocket-Kommunikation

Ein weiteres Problem bei der Entwicklung von Webanwendungen besteht darin, dass die standardmäßige HTTP-Kommunikation zwischen einem Webclient und dem Server nur vom Client aus initiiert werden kann. Der Server ist mit HTTP also nicht in der Lage, von sich aus aktiv zu werden um den Clients Nachrichten zu schicken. Durch den Einsatz des WebSocket-Kommunikationsprotokolls anstatt oder neben HTTP kann eine bidirektionale Kommunikation ermöglicht werden. So ist es möglich, die Daten der einzelnen Clients synchron zu halten [Fur11, S. 673].

2.5 Datenstore mit WebSocket: herkömmlicher Ansatz

In der Praxis werden oft beide der oben beschriebenen Technologien gemeinsam eingesetzt. Wenn nun eine WebSocket-Nachricht vom Server gesendet wird, führt dies normalerweise dazu, dass am Client eine entsprechende Aktion ausgelöst wird, die den zentralen Store aktualisiert. Jede neue WebSocket-Nachricht erfordert somit auch die Erstellung einer eigenen Client-Methode, in der auf die Nachricht reagiert und der Store entsprechend angepasst wird. Dadurch ist es nötig, pro WebSocket-Nachricht sowohl server- als auch clientseitige Logik zu implementieren.

Wenn nun beispielsweise eine Datenänderung veranlasst wird, die auch anderen Clients angezeigt werden soll, kann wie folgt vorgegangen werden: Der Client, der die Datenänderung auslöst, sendet eine Action an den Datenstore. Im Store ist ein Seiteneffekt definiert, der dazu führt, dass eine HTTP- oder WebSocket-Nachricht an den Server gesendet wird. Der Server verarbeitet den eingehenden Request, indem er beispielsweise Validierungen oder Datenbankänderungen durchführt. Danach sendet der Server eine WebSocket-Nachricht an alle Clients, die sich zuvor für Benachrichtigungen angemeldet haben. Die Clients empfangen die WebSocket-Nachricht und verarbeiten sie, indem sie eine weitere Action an den Store senden. Diese Action aktiviert den Reducer, der den Datenstore anpasst. Bei der Implementierung neuer Funktionen, die eine Server-Client-WebSocket-Kommunikation erfordern, muss der Entwickler die folgenden Schritte durchführen:

1. Definition einer Action, die den Seiteneffekt auslöst.
2. Definition einer Action, die für eine Anpassung durch den Reducer sorgt.

3. Implementierung eines neuen WebSocket-Endpunkts auf der Serverseite.
4. Implementierung des Seiteneffekts, der den WebSocket-Endpunkt aufruft.
5. Implementierung der serverseitigen Validierungs- und Anpassungslogik.
6. Implementierung eines neuen WebSocket-Endpunkts auf der Clientseite. Dieser sendet die Reducer-Action an den Store.
7. Aufruf des Client-Endpunkts auf der Serverseite.
8. Implementierung der Reducer-Funktion, die die eigentlichen Client-Daten anpasst.

Kapitel 3

Stand der Technik

3.1 Verwandte Arbeiten

In einer Arbeit von Qu et al. [QM19] wird ein Entwicklungsframework vorgestellt, das die WebSocket-Technologie mit einem Redux-basierten Datenstore verknüpft. Diese Arbeit konzentriert sich jedoch ausschließlich auf die Verwendung der JavaScript-Frameworks React auf der Clientseite und Node.js auf der Serverseite. Im Gegensatz dazu ist das Ziel dieser Arbeit die Entwicklung einer generischen Spezifikation, die unabhängig von den eingesetzten Frameworks implementiert werden kann.

Weitere verwandte Arbeiten, wie die von McFarlane [McF19] oder Tuomi [Tuo18], behandeln oder erwähnen ebenfalls das Thema, beschränken sich jedoch ebenfalls auf die Verwendung der Frameworks React oder Node.js.

Die oben genannten Arbeiten werden voraussichtlich in die im Kapitel 4 beschriebenen Vergleiche einbezogen, um einen umfassenden Überblick über die Leistungsfähigkeit und Anwendbarkeit der entwickelten Lösung zu erhalten.

3.2 Analyse der verwandten Arbeiten

Kapitel 4

YReduxSocket

4.1 Generische Konzeption und Spezifikation

Die Idee hinter YReduxSocket ist nun den in Abschnitt 2.5 beschriebenen Ansatz zu vereinfachen. Dabei werden die Actions sowohl auf der Server als auch auf der Clientseite definiert. Anstatt die Action nun an den Store zu senden, wird sie direkt an den Server übermittelt. Der Server benötigt lediglich einen einzigen WebSocket- oder HTTP-Endpunkt zum Empfangen der Action. Dieser Endpunkt führt dazu, dass eine Methode ausgeführt wird, in der je nach Identifikationszeichenfolge der Action entschieden wird, welche Validierungen oder Datenbankänderungen durchgeführt werden sollen. Nach Abschluss sendet der Server wieder eine Action per WebSocket an den Client. Dies geschieht ebenfalls über eine zuvor bereits definierte Client-Methode. In dieser Methode muss der Client lediglich sicherstellen, dass die Action an den Store weitergeleitet wird. Über die weitergeleitete Action wird dann der Reducer aktiviert, der den Store anpasst. Wenn neue Endpunkte benötigt werden, kann auf Seiteneffekte, die die Server-Client-Kommunikation behandeln, vollständig verzichtet werden. Mit dieser Idee muss eine Entwicklerin die folgenden Schritte durchführen, um die Server-Client-Kommunikation zu erweitern:

1. Definition von zwei Actions auf der Client- und der Serverseite.
2. Senden einer Action an den Server durch Aufruf des vorhandenen Endpunkts.
3. Implementierung der serverseitigen Validierungs- und Anpassungslogik.
4. Aufruf des vorhandenen Client-Endpunkts auf der Serverseite, wobei die zweite Action übergeben wird.
5. Implementierung der Reducer-Funktion, die die eigentlichen Client-Daten anpasst.

Ein Vergleich mit den Umsetzungsschritten aus Abschnitt 2.5 egibt, dass der vorgestellte Ansatz erheblich weniger Implementierungsaufwand erfordert.

4.2 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.

4.2.1 eventual consistency

4.2.2 Algorithmus von H.

4.2.3 Alternative Ansätze

- Vektoruhren?

4.3 Vergleich mit herkömmlichen Ansatz

4.4 Vergleich mit verwandten Arbeiten

Kapitel 5

Implementierung und Tests

5.1 Angular und ASP.NET

5.2 React und Node.js

5.3 npm und NuGet

Kapitel 6

Fazit

6.1 Zusammenfassung

6.2 Validierung der Lösung

6.3 Potenzielle Verbesserungen des Konzepts

Literatur

[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).

[Fur11] Y Furukawa. „Web-based Control Application using WebSocket“. en. In: (2011). URL: https://accelconf.web.cern.ch/icalepcs2011/papers/wemau010.pdf (besucht am 22. 09. 2023).

[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: 978-3-86490-552-0.

[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).

[QM19] Hao Qu und Kun Ma. „WebSocket-Based Real-Time Single-Page Application Development Framework“. en. In: *Advances on P2P, Parallel, Grid, Cloud and Internet Computing*. Hrsg. von Fatos Xhafa u. a. Lecture Notes on Data Engineering and Communications Technologies. Cham: Springer International Publishing, 2019, S. 36–47. ISBN: 978-3-030-02607-3. DOI: 10.1007/978-3-030-02607-3_4.

[Sak19] Elar Saks. „JavaScript frameworks: Angular vs React vs Vue“. en. In: (2019). URL: https://www.theseus.fi/bitstream/handle/10024/261970/Thesis-Elar-Saks.pdf (besucht am 21. 09. 2023).

[Tha20] Mohit Thakkar. *Building React Apps with Server-Side Rendering: Use React, Redux, and Next to Build Full Server-Side Rendering Applications*. en. Berkeley, CA: Apress, 2020. ISBN: 978-1-4842-5868-2. DOI: 10.1007/978-1-4842-5869-9. URL: http://link.springer.com/10.1007/978-1-4842-5869-9 (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.

[Tuo18] Jan-Sebastian Verner Tuomi. „Automated Initialization of Web Software Projects“. en. In: (2018). URL: https://jan.systems/files/docs/kandi.pdf (besucht am 22. 09. 2023).

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