Exposé zur Abschlussarbeit am LG Kooperative Systeme der FernUniversität in Hagen

„YReduxSocket“: Optimierung der Entwicklung von Webanwendungen mit zentralem Datenstore und WebSocket-Kommunikation

Ricardo Stolzlechner

9463470

ricardo.stolzlechner@gmail.com

Informatik Bachelor of Science

15. Oktober 2023 bis 15. April 2024

Betreuerin: Dr. Lihong Ma

1 Problemstellung

1.1 Motivation

Die Webanwendung „Yoshie.io“[1] ist eine kollaborative Plattform, die sich auf das Baugewerbe spezialisiert hat und bei der der Autor dieser Arbeit als technischer Leiter tätig ist. Im Verlauf 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, diesen Aufwand erheblich zu verringern. Dieses

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

Konzept erhielt intern den Namen „YReduxSocket“. Im Zuge dieser Arbeit soll das erwähnte Konzept eingehend betrachtet, spezifiziert und verbessert werden.

1.2 Aufgabenstellung

Die Aufgabe, die in dieser Arbeit behandelt wird, besteht darin, den Ansatz, der mit „YReduxSocket“ verfolgt wurde, möglichst generisch aufzubereiten. Dabei soll eine Spezifikation unter Einsatz von UML-Diagrammen und Pseudocode-Algorithmen entstehen, um „YReduxSocket“ auch für andere Softwareprojekte einsetzbar zu machen.

Im Zuge von Literaturrecherchen und der Betrachtung von Open-Source-Projekten sollen alternative Lösungsansätze gefunden werden, die die Websocket-Technologie mit einem auf Redux basierten Datenstore verbinden. Die identifizierten Ansätze sollen mit „YReduxSocket“ verglichen werden. Aus diesen Vergleichen sollen Verbesserungsmöglichkeiten erarbeitet werden, die anschließend in „YReduxSocket“ eingebracht werden.

Abschließend sollen Implementierungstests in verschiedenen Frameworks durchgeführt werden, um sicherzustellen, dass die aufgestellte Spezifikation auch tatsächlich anwendbar ist. Im Idealfall soll „YReduxSocket“ als eigenes Framework für gängige Paketmanager wie npm oder NuGet bereitgestellt werden.

1.3 Intendierte Ergebnisse

Die angestrebten Ergebnisse lassen sich in drei Hauptphasen gliedern:

Generische Konzeption und Spezifikation

Zunächst werden die eingesetzten Technologien, also der zentrale Redux-basierte Datenstore und das WebSocket-Protokolls in einführenden Kapiteln beschrieben. Danach wird „YReduxSocket“, in einer generischen Weise spezifiziert. Dabei liegt der Fokus darauf, eine Lösung zu entwickeln, die unabhängig von den eingesetzten Client- und Server-Frameworks anwendbar ist. Die Spezifikation wird in einer klaren und präzisen Formulierung ausgearbeitet, um die Umsetzung für Entwickler zu erleichtern. Ein Teil der Arbeit wird sich mit dem Konsistenzverhalten von „YReduxSocket“ beschäftigen.

Recherche und Vergleich von Alternativen

In der zweiten Phase wird eine umfassende Recherche durchgeführt, um alternative Ansätze zu identifizieren. Diese Alternativen werden eingehend analysiert und mit der in Phase 1 entwickelten Spezifikation verglichen. Der Fokus liegt dabei auf der Effizienz, der Skalierbarkeit und der Praktikabilität der verschiedenen Ansätze. Die Ergebnisse dieser Analyse dienen als Grundlage für Verbesserungen von „YReduxSocket“.

Implementierung und Tests

Abschließend wird die entwickelte Spezifikation und der generische Algorithmus in verschiedenen Client- und Server-Frameworks implementiert. Dabei werden umfassende Tests durchgeführt, um die Funktionalität und Leistungsfähigkeit der Lösung zu überprüfen. Diese Tests werden sorgfältig dokumentiert und dienen dazu, die praktische Anwendbarkeit des Konzepts in realen Szenarien zu validieren. Eventuelle Implementierungsherausforderungen und Lösungsansätze werden ebenfalls dokumentiert.

Die angestrebten Ergebnisse zielen darauf ab, ein effektives und leicht anwendbares Konzept zur Optimierung der Entwicklung von Webanwendungen mit zentralem Datenstore und WebSocket-Kommunikation zu schaffen. Dies soll die Entwicklungseffizienz steigern, und die Wartbarkeit verbessern. Durch Vergleiche mit alternativen Ansätzen und Implementierungstests wird angestrebt, ein umfassendes Verständnis für die Leistungsfähigkeit und die Anwendbarkeit der Lösung zu gewinnen.

2 Aktueller Stand der Technik

Um besser nachvollziehen zu können wie ein zentraler Datenstore mit dem WebSocket Kommunikationsprotokoll verknüpft werden kann wird zunächst kurz auf beide Technologien eingegangen. Danach wird ein einfaches mögliches Umsetzungsmodell erläutert. Dieser Abschnitt schließt mit einem Überblick über verwandte Arbeiten.

2.1 Redux basierter 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.

2.2 Das WebSocket Kommunikationsprotokoll

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].

2.3 Zentraler Redux-Datenstore mit WebSocket-Kommunikation

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.

2.3.1 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 Abschnitt 1.3 beschriebenen Vergleiche einbezogen, um einen umfassenden Überblick über die Leistungsfähigkeit und Anwendbarkeit der entwickelten Lösung zu erhalten.

3 Lösungsidee

3.1 Beschreibung

Die Idee hinter „YReduxSocket“ ist nun den in Abschnitt 2.3 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 2.3 egibt, dass der vorgestellte Ansatz erheblich weniger Implementierungsaufwand erfordert.

3.2 Konsistenz

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 Vorläufige Gliederung

1. Einleitung
   - 1.1 Hintergrund und Motivation
   - 1.2 Zielsetzung
   - 1.3 Struktur der Arbeit
2. Grundlagen
   - 2.1 Komponentenbasierte Webentwicklung
   - 2.2 Zentraler Redux-Datenstore
   - 2.3 WebSocket-Kommunikation
   - 2.4 Datenstore mit WebSocket: herkömmlicher Ansatz
3. Stand der Technik
   - 3.1 Verwandte Arbeiten
   - 3.2 Analyse der verwandten Arbeiten
4. YReduxSocket
   - 4.1 Generische Konzeption und Spezifikation
   - 4.2 Konsistenz
   - 4.3 Vergleich mit herkömmlichem Ansatz
   - 4.4 Vergleich mit verwandten Arbeiten

5. Implementierung und Tests
    5.1 Angular und ASP.NET
    5.2 React und Node.js
    5.3 Optional: npm und NuGet
6. Fazit
    6.1 Zusammenfassung
    6.2 Validierung der Lösung
    6.3 Potenzielle Verbesserungen des Konzepts

A Anhang

B Eigenständigkeitserklärung

5 Vorläufiger Zeitplan

Geplant ist, die Arbeit im Zeitraum vom 15. Oktober 2023 bis zum 15. April 2024 zu verfassen. Dies umfasst die Kalenderwochen 42 bis 52 im Jahr 2023 bzw. 1 bis 15 im Jahr 2024. Aus Gründen der Übersichtlichkeit erfolgt die Zeitplanung in Kalenderwochen (KW).

- KW 42: Grundstruktur aufbauen und Kapitel 1. Einleitung
- KW 43: Kapitel 2. Grundlagen
- KW 44 bis KW 45: Kapitel 4.1 Generische Konzeption und Spezifikation
- KW 46: Kapitel 4.2 Vergleich mit herkömmlichem Ansatz
- KW 47: Literaturrecherche für Kapitel 3
- KW 48: Kapitel 3.1 Verwandte Arbeiten
- KW 49: Kapitel 3.2 Analyse der verwandten Arbeiten
- KW 50: Kapitel 4.2 Konsistenz
- KW 51: Kapitel 4.3 Vergleich mit verwandten Arbeiten
- KW 51: Abgabe einer Zwischenversion der Arbeit
- KW 52 bis KW 4: Implementierungen und Tests durchführen
- KW 5: Kapitel 5.1 Angular und ASP.NET
- KW 6: Kapitel 5.2 React und Node.js
- KW 7: Kapitel 5.3 npm und NuGet (optional)
- KW 8 bis KW 9: Kapitel 6 Fazit und Abstract

- KW 9: Abgabe einer ersten vollständigen Version der Arbeit
- KW 10 bis KW 11: Erstellung einer PPT-Präsentation
- KW 12 bis KW 13: Überarbeitung der Arbeit
- KW 14 bis KW 15: Reserve
- KW 15: Abgabe der Arbeit

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).

[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).