Freie Universität Berlin

Bachelorarbeit am Institut für Informatik der Freien Universität Berlin

Arbeitsgruppe Software Engineering

Neuimplementierung und Weiterentwicklung der HTML-GUI von Saros

Marius Schidlack
Matrikelnummer: 4766056
marius.schidlack@fu-berlin.de

Betreuer:	Franz Zieris
Eingereicht bei:	Prof. Dr. Lutz Prechelt
Zweitgutachter:	Prof. Dr. Margarita Esponda-Argüero

Berlin, 2. August 2017

Zusammenfassung

Im Zuge der Bestrebungen, das Saros-Plugin nicht nur für Eclipse, sondern auch für andere IDEs anzubieten, wurde die HTML-GUI eingeführt. Die neue HTML-GUI soll im Gegensatz zu der alten, in Java implementierten Version nicht nur in Eclipse funktionieren, sondern eine plattformunabhängige Lösung für die Benutzeroberfläche darstellen. Ziel dieser Arbeit war anfangs eine Steigerung der Codequalität der HTML-GUI durch Neuimplementierung mit einem besser geeigneten Framework. Hierzu wählte ich das JavaScript-Framework *React* und überarbeitete die Architektur der Applikation. Anschließend sollte die Funktionalität der HTML-GUI erweitert werden. Diese Erweiterung war jedoch nicht so umfangreich, wie anfänglich geplant. Die Ziele, die ich mir gesetzt hatte, änderten sich im Laufe der Arbeit dahingehend, dass ich die bestehenden Features mit automatischen Tests abdecken wollte.

Eidesstattliche Erklärung

Ich versichere hiermit an Eides Statt, dass diese Arbeit von niemand anderem als meiner Person verfasst worden ist. Alle verwendeten Hilfsmittel wie Berichte, Bücher, Internetseiten oder ähnliches sind im Literaturverzeichnis angegeben, Zitate aus fremden Arbeiten sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

2. August 2017

Marius Schidlack

Inhaltsverzeichnis

1 Einleitung

1.1 Paarprogrammierung

Der Begriff Paarprogrammierung bezeichnet eine Arbeitsweise in der Softwareentwicklung, bei der zwei oder mehr Entwickler zusammen an einem Computer an Programmcode arbeiten. Ziel dieser Vorgehensweise ist eine erhöhte Softwarequalität und schnellere Produktentwicklung durch Wissensaustausch und Diskussion zwischen den Programmierern. Ein großer Nachteil von Paarprogrammierung ist jedoch die Notwendigkeit räumlicher Nähe und höhere Personalkosten. [16]

1.2 Saros

Saros ist ein Plug-in für die IDE *Eclipse*[1] und stellt eine Lösung zur *verteilten Paarprogrammierung* dar, die viele der Schwächen der herkömmlichen Paarprogrammierung beseitigen soll. Saros ermöglicht dem Nutzer der Entwicklungsumgebung, seine Projekte mit anderen Nutzern in sogenannten „Sessions“ zu teilen und kollaborativ daran zu arbeiten. Dabei können die Mitglieder einer Session in Echtzeit beobachten, woran die anderen gerade arbeiten. Die Idee für das Projekt entstand 2006 in der Arbeitsgruppe Software-Engineering der Freien Universität Berlin.[2] Das Projekt wird heute hauptsächlich im Rahmen von Abschlussarbeiten vorangetrieben.

1.3 Saros HTML-GUI

Eclipse ist nur eine von vielen Entwicklungsumgebungen und andere Lösungen dieser Art gewinnen mehr und mehr an Popularität. Das von JetBrains entwickelte *IntelliJ IDEA*[3] sticht dabei besonders ins Auge. Aufgrund dieser Entwicklung wurde entschieden Saros auch mit *IntelliJ* nutzbar zu machen (Saros/I).

Im Rahmen dieser Portierung traten viele Probleme auf, die Gegenstand diverser Arbeiten über Saros waren. Eines dieser Probleme war, dass die bestehende Saros-Benutzeroberfläche unter Verwendung der Java-Bibliothek *SWT*[4] implementiert wurde, *IntelliJ IDEA* jedoch auf *Swing*[5] und *AWT*[6] zurückgreift. Diese verschiedenen Bibliotheken sind nicht ohne weiteres kompatibel und die Benutzeroberfläche müsste daher teilweise doppelt implementiert werden.

Eine solche Code-Redundanz ist jedoch zu vermeiden, weil sie zu erhöhtem Wartungsaufwand und schlechterer Softwarequalität führt. Schreibt man Code mit derselben Funktionalität mehrmals, so muss bei einer Änderung der entsprechenden

[1] https://www.eclipse.org/
[2] http://www.saros-project.org/history
[3] https://www.jetbrains.com/idea/
[4] https://www.eclipse.org/swt/
[5] http://docs.oracle.com/javase/7/docs/technotes/guides/swing/index.html
[6] http://docs.oracle.com/javase/8/docs/technotes/guides/awt/

Funktion diese auch an mehreren Stellen durchgeführt werden. Das erfordert nicht nur mehr Zeit, sondern führt früher oder später zu Fehlern, da man sich oft nicht an alle betroffenen Code-Stellen erinnern kann.

Es kam die Idee auf, die GUI mit JavaScript und HTML zu implementieren und sie in einem in die IDE eingebetteten Browser auszuführen, anstatt hierfür eine Java-Bibliothek zu verwenden. Diese Möglichkeit untersuchte Christian Cikryt in seiner Masterarbeit [15]. Seine Analyse ergab, dass die Verwendung eines Browsers eine adäquate und plattformübergreifende Lösung für dieses Problem darstellt [15, 6.2]. Für den Moment entschied er sich einen auf *SWT* basierenden Browser zu verwenden. Diese Entscheidung kann jedoch aufgrund von Kompabilitätsproblemen mit neueren Java Versionen zukünftig noch einmal revidiert werden [15, 5.3.1].

Darauf folgte die Arbeit von Bastian Sieker[19], der die Basis für die HTML-GUI mit Hilfe des JavaScript-Frameworks *Ampersand.js*[7] implementierte, sowie eine Analyse zur Nutzerfreundlichkeit durchführte. Als nächstes nahm sich Nina Weber der HTML-GUI an und äußerte Zweifel an der Entscheidung *Ampersand.js* als Framework zu verwenden[21]. Sie entschied sich jedoch letzten Endes, doch auf der bestehenden Basis weiterzuarbeiten und den Einstieg für folgende Entwickler zu erleichtern.

1.4 Motivation

Als ich begann mich in das Saros Projekt einzufinden, wurde schnell klar, dass mit dem bestehenden Lösung für den JavaScript-Teil der HTML-GUI niemand so recht zufrieden war. Die von Nina Weber geäußerten Zweifel konnte ich schnell nachvollziehen und ich hatte trotz meiner relativ großen Erfahrung im Web-Development Schwierigkeiten den Code zu verstehen. Nach kurzer Einarbeitung sah ich in dem Code großes Verbesserungspotential. Ich kam zu der Einsicht, dass ich den Code überarbeiten wollte, um für die Zukunft eine schnellere und bessere Weiterentwicklung der HTML-GUI zu gewährleisten.

1.5 Zielsetzung und Abgrenzung

Das gewünschte Ergebnis dieser Arbeit ist vorrangig die Verbesserung der Codequalität der HTML-GUI. Diese Verbesserung soll nicht nur mich in die Lage versetzen schneller zu Arbeiten, sondern auch zukünftigen Entwicklern eine schnellere Einarbeitung und Erweiterung ermöglichen. Die Funktionalität der HTML-GUI soll dabei vollständig wiederhergestellt und anschließend erweitert werden. Das Erscheinungsbild der HTML-GUI soll an der in Java implementierten Benutzeroberfläche angelehnt werden, da diese bereits durch Nutzertests validiert wurde.[20]

[7] https://ampersandjs.com/

1.6 Vorgehensweise

Um die Codequalität zu steigern überarbeitete ich den Code komplett und implementierte ihn zu großen Teilen neu. Im Zuge dessen tauschte ich das Zugrunde liegende Framework *Ampersand.js* zugunsten einer besser geeigneten Alternative aus.

1.7 Gliederung

Diese Arbeit gliedert sich inhaltlich wie folgt:

In Kapitel 2 werde ich zunächst einige technische Begrifflichkeiten klären. Anschließend werde ich in Kapitel 3 den Stand der HTML-GUI vor meiner Arbeit vorstellen. Das darauf folgende Kapitel 4 widmet sich dem Prozess des Einarbeitens in den bestehenden Code, sowie mit der Reevaluation der von meinen Vorgängern getroffenen Entscheidungen. Darauf folgt die Entscheidung des Framework-Wechsels. In Kapitel 5 gebe ich eine kurze Einführung in die Technologien, die bei der Neuimplementierung zum Tragen kommen. Die Kapitel 6 und 7 handeln von dem Prozess der Rekonstruktion der bisherigen Funktionalität der HTML-GUI mit den neu gewählten Technologien, sowie der Erweiterung mit neuen Funktionen. Abschließend beschreibt das Kapitel 8 die Implementierung automatischer Tests und die dabei verwendeten JavaScript-Libraries.

2 Grundlagen

Nachdem ich nun die Ziele dieser Arbeit definiert habe, möchte ich zunächst einen kurzen Einblick in die Beschaffenheit der HTML-GUI geben.

Die Saros HTML-GUI besteht, wie die meisten Web-Applikationen, aus einem *Frontend* und einem *Backend*.

Frontend bezeichnet die eigentliche Benutzeroberfläche, das heißt den HTML- und JavaScript-Teil der Applikation. Im Frontend wird ein JavaScript-Framework verwendet, das grundlegende Aufgaben, wie das Rendern von Daten und Anfragen an das Backend abstrahiert.

Der Begriff *Backend* meint in diesem Fall den Kern von Saros, der die Funktionen bereitstellt, die mit Hilfe der Benutzeroberfläche genutzt werden können. Das Backend wird im Folgenden auch als *Saros-Core* bezeichnet. Auf die Funktionsweise des Saros-Core möchte ich an dieser Stelle nicht genauer eingehen, da er für das Verständnis dieser Arbeit keine große Rolle spielt. Für genauere Informationen über die Saros-Architektur empfehle ich daher die technische Dokumentation von Saros zu lesen[8].

[8] http://www.saros-project.org/techdoc

3 Status Quo

Im vorherigen Kapitel erklärte ich die Grundlegenden Begriffe, die für das Verständnis dieser Arbeit erforderlich sind. Es folgt eine Zusammenfassung des Standes der HTML-GUI, den ich zu Beginn meiner Arbeit vorfand.

3.1 Frontend

Die HTML-Benutzeroberfläche besteht vor dieser Arbeit aus drei Ansichten. Der erste Teil, die Hauptansicht der Applikationen, zeigt den Account, mit dem der Nutzer momentan eingeloggt ist, sowie eine Liste der Kontakte dieses Accounts (Abbildung 1). Der Connect/Disconnect-Button dient dazu, sich mit dem aktuellen Account zu verbinden, oder die Verbindung zu trennen. Man kann zudem in einem Dropdown-Menü einen anderen Account auswählen. Dies ist jedoch leider nicht funktionsfähig. Die Hauptansicht zeigt außerdem die Kontaktliste des Nutzers. Durch einen Rechtsklick auf einen Kontakt öffnet sich ein Menü, mit welchem man diesen umbenennen und löschen kann (Abbildung 3). Mit dem Add-Contact-Button wird die zweite Ansicht zum Hinzufügen von Kontakten geöffnet (Abbildung 2).

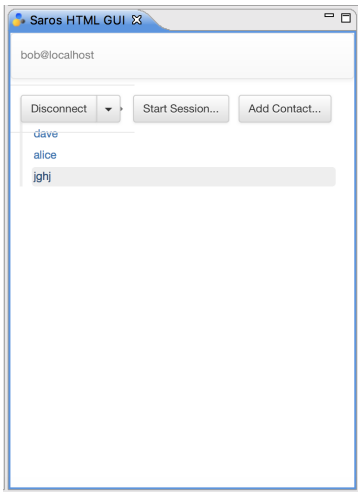

Abbildung 1: Die Hauptansicht

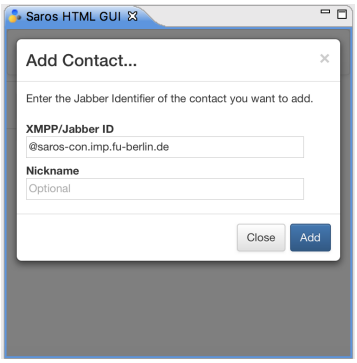

Abbildung 2: Kontakte hinzufügen

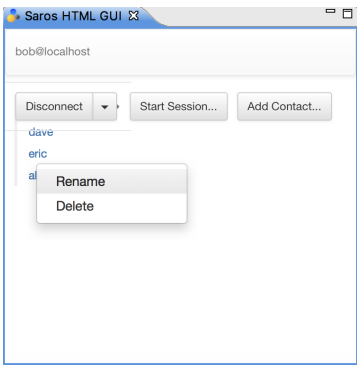

Abbildung 3: Kontakte umbenennen und löschen

Beim Klicken auf den Start-Session-Button öffnet sich die dritte Ansicht, der *Start-Session-Wizard* zum Konfigurieren einer Saros-Session. Im ersten Schritt wählt man die Dateien und Projekte aus, die geteilt werden sollen und im zweiten Schritt die Kontakte, mit denen zusammen gearbeitet werden soll.

3.2 Backend

Das Backend sammelt alle Informationen, die im Frontend dargestellt werden müssen und stellt diese über eine Browser-Schnittstelle bereit. Um das Frontend über Ereignis-

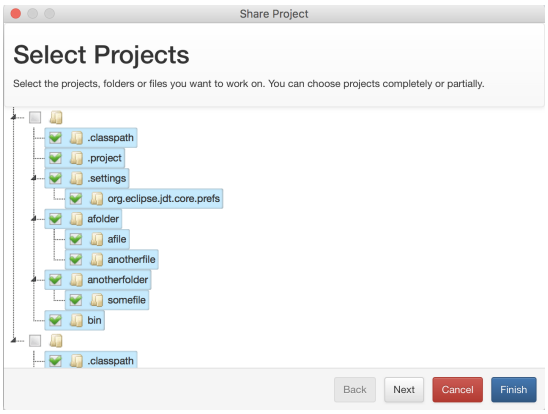

Abbildung 4: Projekte auswählen

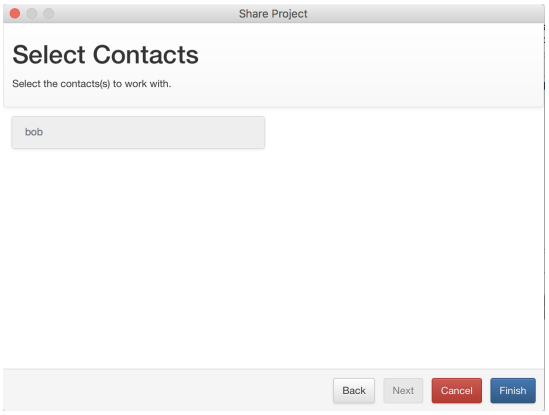

Abbildung 5: Kontakte auswählen

se zu informieren, gibt es sogenannte *Renderer*. Ein Renderer ist dabei verantwortlich für das Sammeln bestimmter Daten, wie z. B. der *AccountRenderer*, der das Frontend über Änderungen in den Nutzeraccounts informiert.

Umgekehrt kann das Frontend mit dem Backend über sogenannte *Browser-Functions* kommunizieren. Die Browser-Functions sind Java-Funktionen, die in den SWT-Browser injiziert werden. Im Browser können diese Funktionen dann wie herkömmliche JavaScript-Funktionen aufgerufen werden.

3.3 Bisher verwendete Frontend-Technologien

Wie bereits angedeutet, wurde die HTML-GUI mit dem JavaScript-Framework *Ampersand.js*[9] entwickelt. Das Framework wird von seinen Autoren wie folgt beschrieben:

> A highly modular, loosely coupled, non-frameworky framework for building advanced JavaScript apps.

Für die Darstellung der HTML-Elemente wurde die *Template-Engine Handlebars*[10] verwendet.

Template-Engine Eine Template-Engine wird genutzt um HTML zu generieren, das dann in einem Browser gerendert werden kann. In *Handlebars* werden z. B. Ausdrücke, die zwischen geschweiften Klammern stehen, zur Laufzeit mit Daten gefüllt.

Um die verschiedenen Dateien in ein großes Skript zu bündeln, wurde bisher *Browserify*[11] genutzt. All diese Module werden mit Hilfe des *Node Package Manager* (kurz NPM)[12] installiert.

[9] https://ampersandjs.com/
[10] http://handlebarsjs.com/
[11] http://browserify.org/
[12] https://www.npmjs.com/

4 Einarbeitung

4.1 Reevaluation

Ich begann mit der Einarbeitung in die bestehende Code-Basis und erprobte kleine Änderungen. Dabei stieß ich immer wieder auf Probleme, die selbst kleine Aufgaben zu einem Rätsel für mich machten. Trotz der verbesserten Ordnerstruktur [21, Kap. 3.1.1] und den gut verständlichen *Handlebars*-Templates fand ich mich schnell in derselben Lage wie meine Vorgängerin wieder. Ihre Prognose, dem nächsten Entwickler fiele es vermutlich genauso schwer wie ihr [21, Kap. 2.2], bewahrheitete sich, trotz ihrer Bemühungen, den Einstieg zu erleichtern. Mir fielen grundlegende Probleme an der Beschaffenheit des Codes und des verwendeten Frameworks *Ampersand.js* auf, die ich im Folgenden analysieren werde.

4.1.1 Intransparenter Kontrollfluss

Es ist sehr schwer, nachzuvollziehen welcher Code ausgeführt wird, wenn der Nutzer mit der GUI interagiert. Schuld daran ist die große Zahl an Indirektionen, die bei der Verarbeitung der Events auftreten. Beispiel: Der *Delete*-Button zum Löschen eines Kontakts wird geklickt. Sei dies das zugehörige HTML Element:

```
1 <li><a href="#" data-hook="delete">{{d.action.delete}}</a></li>
```

Und das die Events der entsprechenden Ampersand-View:

```
1 events: {
2     'click [data-hook=rename]': 'rename',
3     'click [data-hook=delete]': 'delete'
4 },
```

Beim Ausführen des Klicks wird das Event ausgelöst, das auf das geklickte HTML-Element passt. In diesem Fall also das in Zeile 3. Ampersand weiß nun, dass es die „delete“ Action ausführen muss. Dazu sucht es in der Ampersand-View eine Funktion mit ebendiesem Namen, und führt diese schließlich aus. Dieser Prozess ist meiner Meinung nach nicht nur unnötig kompliziert, sondern auch schlecht verständlich. Ginge das nicht auch einfacher? Beispielsweise indem man dem Element direkt eine Funktion übergibt, die es „onclick“ ausführen soll?

4.1.2 Kleine Community – Wenig Lernmaterial

Ein Punkt, den auch schon meine Vorgängerin ansprach, ist die extrem kleine Community von *Ampersand.js* Nutzern. Sie verdeutlichte dies durch einen einfachen Vergleich der Anzahl der Fragen die 2016 auf *Stack Overflow*[13] die mit dem Stichwort „ampersand.js“ (ca. 50 Stück) versehen waren, gegen die der Fragen mit „angularjs“

[13] https://stackoverflow.com/

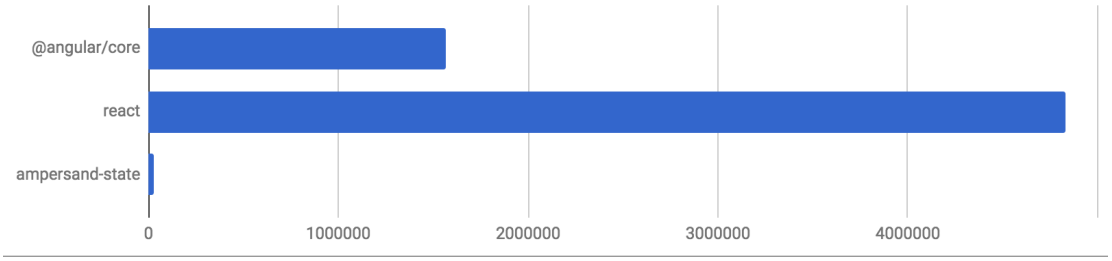

Abbildung 6: Monatliche NPM Downloads: React, Angular.js und Ampersand.js (Stand August 2017)

(ca. 180 000 Stück) [21, Kap. 2.2, Abs. 5]. Dieses Maß ist für sich genommen jedoch womöglich etwas vage, da es ebenso bedeuten könnte, dass Ampersand einfach weniger Fragen aufwirft als Angular. Ein zusätzlicher Indikator ist die Anzahl der Downloads mit NPM. Die Abbildung 6 vergleicht die NPM-Pakete *ampersand-state* [1], *@angular/-core* [2] und *react* [10]. Dieser Vergleich verdeutlicht erneut, wie klein die Ampersand-Community ist. Die Wahrscheinlichkeit, dass einer der Nachfolgenden Entwickler der HTML-GUI *Ampersand.js* kennt, ist also verschwindend gering. Ebenso gering ist die Menge des verfügbaren Lernmaterials für das Framework, was den Einstieg noch weiter erschwert.

4.1.3 Umfangreicher Code – Wenig Funktionalität

Auf den ersten Blick scheint es sehr viel Code zu geben – für eine überschaubare Menge an Funktionen. Der Code besteht zu großen Teilen aus *Boilerplate*.

Boilerplate bezeichnet Codefragmente, die an vielen Stellen ohne nennenswerte Abwandlungen wieder und wieder auftreten. Der Begriff wird häufig verwendet um zu beschreiben, dass ein Programmierer viel Code schreiben muss, um kleinste Aufgaben zu erledigen.

Laut Sieker ist dies ein Kompromiss, der zugunsten einer niedrigeren Abstraktionsstufe des Codes eingegangen werden kann. Ein weniger abstrakter Code, so Sieker, sei für einen Programmierer ohne Erfahrung in Web-Entwicklung einfacher zu verstehen. [19, Kap. 4.4.5, Abs. 3] Diese Prämisse war einer der Hauptgründe für die Entscheidung für Ampersand. Ich widerspreche der Annahme jedoch, denn höhere Abstraktion bedeutet in meinen Augen Fokussierung auf das Wesentliche. Boilerplate zu schreiben benötigt viel Zeit und macht den Code unleserlicher.

Aus ebendiesem Grund werden höhere Programmiersprachen wie z. B. Java oder C++ bei komplexen Problemen den weniger Abstrakten wie z. B. Assembler vorgezogen: Um besseren Code in kürzerer Zeit zu schreiben.

	Abstraction	Built-in features	Adaptibility / Flexibility	Learnability	Community
Angular	high	++	-	-	++
Ember	high	++	-	- -	+
Backbone	low	-	++	++	++
Ampersand	medium	+	+	+	-

Abbildung 7: Siekers Analyse von JavaScript Frameworks [19, S. 21, Tabelle 4]

4.1.4 Neustart von Eclipse bei jeder Änderung

Ein weiteres Problem, dass allerdings nichts mit dem Code zu tun hat, ist die Tatsache, dass man jedes mal, wenn man eine Änderung am JavaScript-Code durchführt, Eclipse komplett neu starten muss. Dies erschwert ein kleinschrittiges und testbasiertes Vorgehen, welches gerade beim Programmieren von Benutzeroberflächen so dringend benötigt wird. Da es sich bei JavaScript um eine Skriptsprache handelt, die nicht in Bytecode kompiliert werden muss, ist das aber eigentlich gar nicht nötig. Ich vermute der Grund, warum ein einfaches Schließen des HTML-GUI Fensters in Eclipse dennoch nicht ausreicht um die Skripte neu zu laden ist Folgender: Der SWT-Browser speichert das Skript beim ersten Laden im *Cache*. Wird die HTML-GUI neu geöffnet und der Dateipfad des Skriptes ist derselbe, wird die Datei aus dem Cache geladen, auch wenn der Code sich eigentlich geändert hat. Was also fehlt ist ein *Cache-Busting* Mechanismus.

Cache-Busting bedeutet in diesem Kontext, dass die JavaScript-Datei, die in dem HTML-Dokument verlinkt ist, jedes mal einen anderen Namen haben muss, wenn es eine neue Version von ihr gibt. Das wird üblicherweise durch das Anhängen einer zufälligen Zeichenkette an den Dateinamen erreicht [13].

4.1.5 Ampersand – Eine fragwürdige Wahl

All diese grundlegenden Probleme veranlassten mich, die Verwendung von Ampersand.js erneut zu überdenken. Bastian Sieker sah damals folgende Alternativen zu Ampersand.js: *Angular.js*, *Ember.js*[14] und *Backbone.js*[15]. Er verglich die Frameworks anhand der Kriterien Abstraktion, Menge an Features, Flexibilität, Erlernbarkeit und Community [19, Kap. 4.4.5] und kam zu dem Schluss, dass der Mittelweg der beste sei: ein flexibles, leicht erlernbares Framework, das aber gleichzeitig auch eine gewisse Menge an Features bereitstellt (siehe Tabelle 7).

Ich denke diese Kriterien sind gut geeignet zur Bewertung der Frameworks, jedoch missinterpretierte er meiner Meinung nach Abhängigkeiten der Kriterien, die sich untereinander ergeben.

[14] https://www.emberjs.com/
[15] http://backbonejs.org/

Beispielsweise folgt aus einer größeren Community durch die größere Menge an Lernmaterial auch eine bessere Erlernbarkeit und aus höherer Abstraktion folgt, wie bereits erwähnt (vgl. Abschnitt 4.1.3), nicht zwangsläufig eine schlechtere.

Darüber hinaus halte ich die Gewichtung der Kriterien, die hier gewählt wurde, nicht für adäquat. Sieker identifizierte hohe Flexibilität und hohe Erlernbarkeit als die wichtigsten Kriterien und für diese war in seinen Augen niedrige Abstraktion eine notwendige Bedingung. Ich kann jedoch nicht nachvollziehen, inwiefern hohe Flexibilität wichtig für die Entwicklung der Saros HTML-GUI ist. Die Aufgaben, die die HTML-GUI hat, können meines Erachtens nach mit jedem Framework problemlos bewältigt werden.

Zusammenfassend möchte ich diese Kriterien also revidieren:

Eine gute Erlernbarkeit ist zweifelsohne ein zentraler Aspekt bei der Wahl der Technologie. Ich vertrete jedoch die Auffassung, dass nicht niedrige Abstraktion, viel Boilerplate und hohe Flexibilität der Schlüssel zu guter Erlernbarkeit sind, sondern aussagekräftiger Code, eine verständliche Architektur, eine große Community, viel Lernmaterial und eine gute Dokumentation.

4.2 Auswahl der neuen Technologie

Nina Weber stellte bereits Überlegungen an, auf *Angular.js* zu wechseln [21, Kap. 2.2]. Als sie mit der Portierung begann stieß sie jedoch auf grundlegende Kompabilitätsprobleme, die die Ausführung von Angular.js 1 Applikationen in dem SWT-Browser unmöglich machten [21, Kap. 2.3]. Sie versuchte sich erneut an Angular version 2, doch scheiterte erneut an technischen Problemen [21, Kap. 2.4].

Angular.js fällt also als Alternative aus, ebenso möchte ich Backbone.js aufgrund seiner Ähnlichkeit zu Ampersand.js verwerfen.

4.2.1 React

Stattdessen möchte ich eine der momentan populärsten (vgl. Abbildung 6) JavaScript-Bibliotheken vorstellen, von der es mich wundert, dass sie nicht schon früher Erwähnung fand: Die von Facebook Inc. entwickelte JavaScript-View-Library *React*[16]. React wurde im Mai 2013 unter der BSD Lizenz auf GitHub veröffentlicht [7]. React ist *kein* vollständiges Framework, sondern lediglich eine *View*-Library [14]. Ihre Schöpfer betiteln es nüchtern:

> a javascript library for building user interfaces [8]

Trotz dieser Tatsache wird es oft mit JavaScript-Frameworks verglichen. Eine Umfrage belegt [17], dass React sich höchster Beliebtheit erfreut. So geben bei einer Umfrage 92% der Teilnehmer, die React schon einmal verwendet haben, an, React wieder verwenden zu wollen. Im Vergleich dazu stehen Angular 2 mit 65%, Ember mit 48% und

[16] https://facebook.github.io/react/

Backbone mit 32% [17]. Die Entwickler von React beschreiben es unter anderem mit folgenden Attributen[8]:

> **Declarative** React makes it painless to create interactive UIs. Design simple views for each state in your application, and React will efficiently update and render just the right components when your data changes. Declarative views make your code more predictable and easier to debug.
>
> **Component-Based** Build encapsulated components that manage their own state, then compose them to make complex UIs. [...]

In React werden View-Elemente in sogenannten *Komponenten* definiert. Eine Komponente ist zu vergleichen mit einem benutzerdefinierten HTML-Element. An die Komponente können dabei *Props* übergeben werden, die beim Rendern der Komponente genutzt werden können. Eine React-Applikation besteht in der Regel aus einer Hauptkomponente, die wiederum aus vielen *Child*-Komponenten besteht. Dabei reichen die *Parent* Komponenten durch *Props* Daten an ihre *Children* weiter.

Beispiel:

```
class Greeting extends React.Component {
  render() {
    return (
      <div>Hello {this.props.who}!</div>
    )
  }
}
```

Listing 1: Eine React-Komponente

Wir haben nun die Komponente „Greeting“ definiert. Die Komponente bekommt einen Namen als *Prop* übergeben, den sie grüßen soll.

```
class GreetApp extends React.Component {
  render() {
    return (
      <div>
        <Greeting who="Alice" />
        <Greeting who="Bob" />
      </div>
    )
  }
}
```

Rendern wir nun die *GreetApp* Komponente, erhalten wir den Auspruch „Hello Alice! Hello Bob!“ Des Weiteren kann man React-Komponenten ganz einfach Funktionen übergeben, die sie z. B. aufrufen sollen wenn sie geklickt wurden:

```
  class Greeter extends React.Component {

    greetMe = () => {
      console.log('Hello!')
    }

    render() {
```

```
8         <button onClick={this.greetMe}>Greet me!</button>
9       }
10    }
```

Zugegeben, die Syntax, die React verwendet, scheint auf den ersten Blick nichts mehr mit JavaScript zu tun zu haben. Darauf werde ich im folgenden Kapitel 5.1 noch weiter eingehen. Jedoch ist die Art, wie in React Views definiert werden, für meine Begriffe sehr intuitiv, aussagekräftig und vor allem lesbar. Man kann mit einen Blick auf die „render"-Methode erkennen, welche Daten wie visualisiert werden sollen und was passiert wenn der Nutzer mit der Oberfläche interagiert.

Mit React werden also viele der Punkte aufgegriffen, die ich an der bisherigen Lösung kritisiert habe (vgl. Abschnitt 4.1.5).

Hinzu kommt noch, dass meine Erfahrung mit React sich bereits über einige Jahre erstreckt, während ich mit Ampersand.js garkeine habe. Wenn ich React verwende, kann ich also zum einen meinen eigenen Entscheidungen wesentlich mehr vertrauen, zum anderen aber auch wesentlich schneller arbeiten. Zum Zeitpunkt dieser Arbeit bin ich zudem der einzige Entwickler, der an der HTML-GUI arbeitet – Noch ein Grund, aus dem es essentiell ist, dass ich genau weiß, was ich tue.

Aus genannten Gründen entschied ich also, einen Versuch zu starten und einen Prototypen in React zu entwickeln, um die Funktionsfähigkeit im SWT-Browser zu testen.

5 Neuimplementierung

Nachdem ich nun einen Überblick darüber gegeben habe, was React eigentlich ist, möchte ich nun zur eigentlichen Neuimplementierung der HTML-GUI kommen.

5.1 Techstack

Dazu gebe ich zunächst eine kurze Einleitung in die Technologien, die dabei zum Einsatz kamen. Bei der in den Beispielen verwendeten Programmiersprache handelt es sich zum einen um JavaScript ES6, die nächste offizielle Version von JavaScript nach ES5[17]. JavaScript ES6 ist eine Obermenge von ES5, das heißt jeder ES5-Code ist auch mit ES6 kompatibel. Diese neuere Version von JavaScript erweitert die Sprache um einige nützliche Features [4]. Darunter befinden sich:

Klassen Ähnlich den aus Java bekannten Klassen – Schlüsselwort *class*.

Konstanten Ähnlich dem *final* Schlüsselwort in Java – Schlüsselwort *const*.

Arrow-Functions Vor dem Pfeil die Parameter, dahinter der Rückgabewert.

```
(a, b) => 2 * a * b
```

[17] JavaScript ES6 ist eine Implementierung des ECMAScript 6 Standards

Template-Strings Diese ermöglichen einfache Einbindung von Variablen in Strings ohne den „+“ Operator zu verwenden.

```
‘Hello ${name}‘
```

Destruktion von Objekten Diese Funktion ermöglicht das einfache Zerlegen von Objekten in seine Bestandteile.

```
1 const people = { a: 'Alice', b: 'Bob' }
2 const { a, b } = people // nimmt die werte a, b aus people und
     erstellt daraus Konstanten
3 console.log(a, b) // Alice Bob
```

Imports In JavaScript ES6 können *Module* definiert werden. *Module* ermöglichen es ein Paket an Funktionalitäten abzukapseln und diese anderswo zu *importieren*. Dieses Feature mag dem ein oder anderen schon von *Node.js*[18] bekannt sein. Es ist zu vergleichen mit dem aus Java bekannten Import von Klassen, nur dass in JavaScript jede beliebige Ressource exportiert und importiert werden kann. Dies beinhaltet neben Klassen auch einzelne Funktionen, Variablen und Konstanten. Die benötigten Schlüsselworte sind *import* und *export*. Beispiel:

```
1 // modulA.js
2 export function add(a ,b){
3   return a + b
4 }
5 // modulB.js
6 import { add } from 'modulA'
7
8 add(1, 2)
```

Hier ein Code-Schnipsel geschrieben in JavaScript ES5:

```
1  var persons = [
2    { firstname: 'Alice', lastname: 'Bar' },
3    { firstname: 'Bob', lastname: 'Foo' },
4  ]
5
6  var greetings = persons.map(function(person) {
7    return 'Hello ' + person.firstname + ' ' + person.lastname + '!'
8  })
9  // greetings = ['Hello Alice Bar!', 'Hello Bob Foo!']
```

Der gleiche Code kann in JavaScript ES6 wie folgt geschrieben werden:

```
1  const persons = [
2    { firstname: 'Alice', lastname: 'Bar' },
3    { firstname: 'Bob', lastname: 'Foo' },
4  ]
5  const greetings = persons.map(({ firstname, lastname }) => ‘Hello ${
     firstname} ${lastname}!‘)
```

Zu beachten ist hierbei, dass die Funktion, die in Zeile 5 an die *Map*-Funktion übergeben wird, weiterhin nur *einen* Parameter hat (eine Person). Dieser Parameter wird jedoch durch die *Destruktion* in „firstname“ und „lastname“ zerlegt.

[18] https://nodejs.org/en/

Zum anderen werden Komponenten in React mit *JSX*[19] beschrieben. JSX ist lediglich eine JavaScript-Syntax-Erweiterung und keine eigene Programmiersprache. Diese wurde eigens zu dem Zweck entwickelt, React-Komponenten mit einer gut lesbaren HTML ähnlichen Syntax (siehe Code-Beispiel 1, Zeile 4) beschreiben zu können.

Sowohl JavaScript ES6, als auch JSX zu verwenden ist optional, bieten jedoch meiner Erfahrung nach einen großen Mehrwert durch weniger wortreichen Code.

5.2 Build-Prozess

5.2.1 Technologien

JSX kann nicht direkt im Browser ausgeführt werden. Hier kommt ein Build-Prozess ins Spiel, der JSX vor der Verwendung im Browser kompiliert. Im Fall der *Greeting*-Komponente sieht der kompilierte Code in etwa so aus:

```
1  class Greeting extends React.Component {
2    render() {
3      return React.createElement(
4        "div",
5        null,
6        "Hello ",
7        this.props.who,
8        "!"
9      );
10   }
11 }
```

JSX ist also letztendlich nichts weiter als syntaktischer Zucker für den Aufruf der „React.createElement“-Funktion.

Auch JavaScript ES6 ist eine relativ junge Neuerung und wird bisher lediglich von den großen Browsern wie Google Chrome und Firefox unterstützt. Um die Annehmlichkeiten, die es bietet, dennoch schon Nutzen zu können, muss es deshalb vor der Nutzung im SWT-Browser zu JavaScript ES5 kompiliert werden. Für diese Aufgabe werden folgende Werkzeuge benötigt [9]:

Bundler Ein Bundler bündelt alle benötigten Skripte des Projekts in ein Paket, das anschließend im Browser geladen werden kann. Dafür wird eine Datei als Einstiegspunkt definiert und alle Module die von dort aus importiert werden, landen mit in dem Bündel. In der Regel durchläuft der Code dabei eine Preprocessing-Pipeline, in der er mit Hilfe von Plug-ins transformiert wird. Diese Transformationen können z. B. das Übersetzen von einer Sprache in die andere sein, oder auch die Komprimierung des Codes. Die populärsten JavaScript-Bundler sind derzeit *Webpack*[20], sowie *Browserify*[21]

[19] https://facebook.github.io/jsx/
[20] https://webpack.js.org/
[21] http://browserify.org/

Transpiler Ein JavaScript-Transpiler wandelt den Quellcode, der in einer auf JavaScript basierenden Sprache geschrieben ist (in unserem Fall JavaScript ES6 mit JSX) in eine ältere, mit dem Ziel-Browser kompatible JavaScript Version um (ES5 oder ES3). Für React-Applikationen wird in der Regel *Babel*[22] verwendet.

Für den Prototypen entschied ich mich *Webpack* zu verwenden, da es meiner Erfahrung nach wesentlich besser und schneller zu Konfigurieren ist, als Browserify.

In Webpack wird der Build-Prozess in einer Konfigurationsdatei definiert[23]. Dort kann man sogenannte *Loader* einbinden, die für die Verarbeitung bestimmter Dateitypen verantwortlich sind. In unserem Fall nutzte ich den *babel-loader*, um Dateien mit der Endung „`.jsx`“ an den Babel-Transpiler zu übergeben. Babel hat wiederum seine eigene Konfiguration, die es anweist, erst die JSX-Syntax aufzulösen (vgl. Beispiele 1, 5.2.1) und dann den verbliebenen JavaScript ES6-Code in ES5 umzuwandeln. Für den Prototypen wies ich also Webpack an, den *babel-loader*[24] für die JSX-Dateien zu verwenden.

5.2.2 Nutzen

Das Aufsetzen des Build-Prozesses ist zugegebenermaßen etwas lästig. Ich sehe es dennoch nicht als Nachteil an, denn offensichtlich entsteht dieser Aufwand nur einmalig beim Aufsetzen des Projekts.

Für das Erstellen von React-Applikationen existiert auch ein *Generator*, konkret *create-react-app*[25]. Ich habe mich jedoch entschieden, diesen für die Implementierung des Prototypen nicht zu verwenden, weil die generierten Apps für die Verwendung in einem voll ausgestatteten Browser optimiert sind und viele der Features im SWT-Browser nicht lauffähig sind.

Durch den neuen Build-Prozess wird zudem ein weiterer Kritikpunkt an der bisherigen Lösung behoben: Webpack ermöglicht *Cache-Busting*, indem es bei jeder Neukompilierung des Codes eine zufällige Zeichenkette an den Dateinamen anhängt (siehe Absatz 4.1.4). Anstatt die Entwicklungsumgebung jedes mal neu starten zu müssen, wenn etwas geändert wurde, reicht es nun, das HTML-GUI Fenster neu zu öffnen. Wenn man dafür in Eclipse noch eine Tastenkombination definiert, kann man Änderungen praktisch ohne Verzögerung sehen.

5.3 Erster Versuch

Die Erfahrung von Nina Weber zeigt, dass der SWT-Browser eine eingeschränkte Funktionalität hat, die manche Frameworks, wie z. B. Angular.js, schon überschreiten [21, Kap. 2.3-2.4]. Nachdem ich den Build-Prozess aufgesetzt hatte, prüfte daher ich

[22] https://babeljs.io/
[23] Standardmäßig die webpack.config.js
[24] https://github.com/babel/babel-loader
[25] https://github.com/facebookincubator/create-react-app

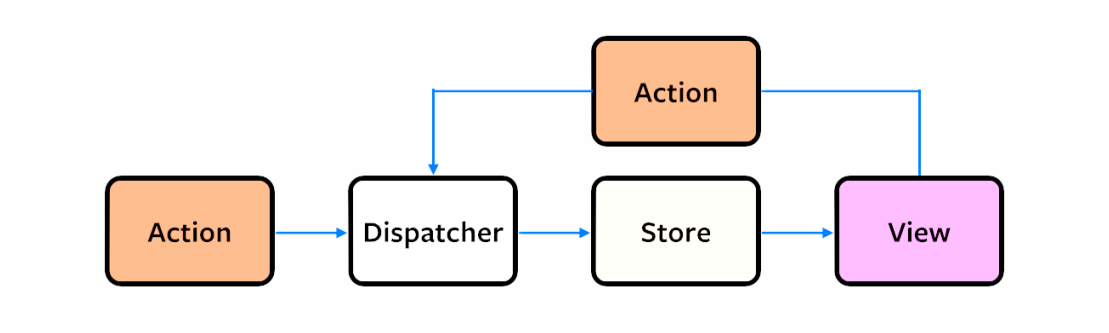

Abbildung 8: Die *Flux*-Architektur
Quelle: [3]

zunächst, ob React in dem SWT-Browser überhaupt lauffähig war. Dazu implementierte ich eine triviale Testkomponente und testete, ob sie ordnungsgemäß gerendert wird und ob die Interaktion mit den HTML-Elementen funktioniert. Zu meinem Erstaunen stieß ich auf keinerlei Probleme – React ließ sich problemlos in dem Browser Fenster ausführen. Einer Neuimplementierung stand also nichts im Wege.

5.4 State-Management

Wie bereits erwähnt, handelt es sich bei React lediglich um eine *View*-Library. Die Applikation benötigt aber auch eine Möglichkeit, den *UI-State*, also den Zustand der Benutzeroberfläche zu verwalten. Der Ort an dem das geschieht, sollte JavaScript seitig liegen, da der Zustand der HTML-GUI für die Java-Welt weitestgehend unerheblich ist. Die Entwickler von React empfehlen hierfür eine Architektur, die sie *Flux* getauft haben.

In *Flux* werden die Applikationsdaten in sogenannten *Stores* gespeichert. Die *View* reagiert auf Änderungen der Daten in den Stores. Des Weiteren existiert ein sogenannter *Dispatcher*. Dabei handelt es sich um ein Observer-Pattern. Das heißt konkret, dass sich die Stores bei dem Dispatcher registrieren um über Ereignisse benachrichtigt zu werden und darauf reagieren zu können. Diese Ereignisse heißen *Actions*. Actions können z. B. ausgelöst werden, wenn der Nutzer mit der GUI interagiert. Sie enthalten Informationen darüber, was passiert ist. (Siehe Abbildung 8)

Beispiel: Der „Delete Contact“-Button wird auf dem Kontakt „Alice“ geklickt. Eine *Action* wird ausgelöst:

```
1   { type: 'DELETE_CONTACT', name: 'Alice' }
```

Der Store wird benachrichtigt und weiß nun, dass er den Kontakt mit dem Namen „Alice“ löschen soll. Nachdem er dies getan hat, wird die Kontaktliste neu gerendert.

Der Vorteil dieser Architektur ist vor allem ein *unidirektionaler Datenfluss*. Die Views ändern nicht direkt den Zustand der Applikation, sondern schicken lediglich Nachrichten an den Dispatcher. Es ist somit sehr leicht, nachzuvollziehen, wie die Applikation auf welche Interaktion reagiert. Auch müssen so die Views von der Existenz

der Stores gar nichts wissen. Nach dem Prinzip des MV*-Musters werden View und Stores so zu vollkommen eigenständigen Komponenten, die unabhängig voneinander testbar sind.

Die Komponenten isoliert testen zu können ist essentiell für die Implementierung von Unit-Tests, die zukünftig die gesamte HTML-GUI abdecken sollen.

Flux ist wie gesagt lediglich ein Konzept. Es existieren jedoch bereits Lösungen, die einige Ideen dieses Konzeptes aufgreifen. Für die Saros-GUI zog ich dabei einige Lösungen in Betracht, die ich durch die Implementierung eines Minimalbeispiels miteinander verglich (siehe Anhang A.2). Ich kannte all diese Lösungen zwar bereits, aber von einem direkten und selbst durchgeführten Vergleich versprach ich mir größere Klarheit darüber, welche davon für die HTML-GUI von Saros besser geeignet ist. Im Folgenden werde ich diese Lösungen miteinander Vergleichen.

Redux Redux ist eine Bibliothek, die viele Ideen von Flux aufgreift und erweitert. Die Entwickler nennen es einen „predictable state container“ [11]. In Redux gibt es nur einen *einzigen* State für die gesamte Applikation. Es gibt zudem eine Funktion genannt *Reducer*, die jedes mal aufgerufen wird, wenn eine Action ausgelöst wird. Der Reducer bekommt als Argumente den aktuelle State, sowie die Action übergeben und gibt einen neuen State zurück. Dieser ersetzt dann den aktuellen State der Applikation. Ein Reducer ist eine Funktion *ohne Seiteneffekte*. Das heißt, dass durch die bloße Ausführung der Funktion keinerlei Zustandsänderungen in der Applikation herbeigeführt werden können. Die Funktion bekommt lediglich Argumente, und gibt einen Rückgabewert [11, Kapitel „Tree Principles“]. Daraus folgt zwangsläufig auch, dass der bestehenden State der Applikation nicht verändert wird, sondern jedes mal eine komplett neue Version des States geschaffen wird. Dieses Konzept ist bekannt als *Immutability* [18]. Immutability ist zum einen nützlich, weil es nicht zurückverfolgbare Änderungen an Objekten verhindert und zum anderen, weil es die Erkennung von Änderungen an einem Objekt extrem effizient macht: Eine Version des Objekts unterscheidet sich genau dann von einer anderen, wenn sie unterschiedliche Zeiger haben. Dies ermöglicht React sehr kostengünstig festzustellen, ob eine Komponente neu gerendert werden muss [12]. Redux greift somit einige Konzepte der funktionalen Programmierung auf [22]. Ein Nachteil von Redux, den ich bereits bei der Implementierung des Beispiels zu spüren bekam, ist die große Menge an Boilerplate, die geschrieben werden muss.

MobX MobX ist eine andere State-Management Lösung, die vor allem aufgrund ihrer Einfachheit beliebt ist. In MobX werden Daten, auf deren Änderung reagiert werden soll als *Observable* markiert, React-Komponenten, die darauf reagieren sollen, als *Observer* und Funktionen, die die Daten verändern als *Action* [5]. MobX erkennt dabei automatisch, welche Komponenten auf welche Änderungen reagieren müssen. Das möchte ich anhand eines kleinen Beispiels vor Augen führen.

```
1   class Person {
2     @observable firstname = 'John'
3     @observable lastname = 'Doe'
4   }
5
6   const person = new Person()
7
8   // Angenommen diese Komponente wird irgendwo gerendert
9   @observer
10  class Greeter {
11    render() {
12      return (
13        <div>Hello {person.firstname}!</div>
14      )
15    }
16  }
17
18  person.firstname = 'Max' // Das loest erneutes rendern aus
19  person.lastname = 'Mustermann' // Das nicht
```

Die Klasse Greeter verwendet in ihrer „render“-Methode einen Wert, der als *observable* markiert wurde: „person.firstname“. Jede Änderung an diesem Wert wird ein erneutes Rendern der *Greeter*-Komponente auslösen. Ändert man jedoch den „lastname“ der Person, wird nicht neu gerendert, da der Wert in der „render“-Methode nicht vorkommt. Das schöne (und vielleicht auch gefährliche) an MobX ist, dass es einfach funktioniert. Selten gibt es Fälle, an denen man überhaupt verstehen muss was es tut[26]. In MobX ist man sehr frei in der Art und Weise wie man seine Daten speichert, die eigentliche Architektur zu wählen, liegt beim Programmierer. Darin sehe ich allerdings auch den größten Nachteil von MobX: Unerfahrene Entwickler könnten eine unangebrachte Architektur wählen, die zu schlecht wartbarem Code führt. Im gegebenen Beispiel hängt die React-Komponente direkt von der „person“-Variable ab. Dies steht der Unabhängigkeit der View von dem Model im Wege, die für Unit-Tests erforderlich ist. Diese direkte Abhängigkeit kann jedoch im realen Anwendungsumfeld durch *Dependency-Injection* umgangen werden.

Die Beispielimplementierung hat hier mit Abstand den wenigsten Code.

Keine Library Die *Flux* Architektur ist einfach. So einfach, dass sie in kurzer Zeit selbst implementiert werden kann. Diese Lösung würde vollständige Transparenz über den Kontrollfluss bieten, auf der anderen Seite aber auch das Rad neu erfinden und mehr Code bedeuten. Schon in dem Minimalbeispiel war die Menge an Code nahezu doppelt so groß wie in den anderen beiden. Ebenfalls fehlen Performance-Optimierungen, die beide Libraries bieten.

Zur Wahl der in der Saros HTML-GUI verwendeten Lösung rief ich mir die zuvor erhobenen Anforderungen noch einmal vor Augen (siehe Abschnitt 4.1). Bei der HTML-GUI handelt es sich um eine relativ kleine Applikation, deren finale Größe beschränkt ist durch die bestehenden Saros Funktionen. Eine große Menge an Boilerplate würde

[26] https://mobx.js.org/best/react.html

also einen signifikanten Anteil der Code-Basis ausmachen. Ich beschloss daher, eine der beiden Libraries zu benutzen. Redux überzeugt vor allem durch die funktionalen Ansätze, die das Verhalten der Applikation besser nachvollziehbar machen. Wirklich nützlich finde ich das jedoch erst für sehr große Applikationen. Ebenfalls sehe ich für ein Projekt diesen Umfangs zudem keinen großen Nutzen in der Verwendung eines *Dispatch*-Systems. Das direkte Aufrufen von Funktionen auf dem Store ist für diese Zwecke meiner Einschätzung nach völlig ausreichend. Der Redux-Code hat zwar weniger Boilerplate, als die Lösung ohne Library, jedoch für meine Begriffe immer noch zu viel.

MobX scheint mir für die HTML-GUI von Saros die richtige Wahl zu sein. Es erfordert keine tiefere Kenntnis über funktionale oder sonstige Programmierparadigmen, reduziert die benötigte Code-Menge und kann ohne große Einarbeitungszeit genutzt werden. Ich entschied also, die Neuimplementierung mit React und MobX anzugehen.

6 Rekonstruktion

Nachdem nun die grundlegenden Technologien ausgewählt waren, konnte ich mit der Rekonstruktion der HTML-GUI beginnen. Das folgende Kapitel beschreibt die wichtigsten Unterschiede zur alten Lösung, sowie die größeren Entscheidungen, die ich dabei getroffen habe.

6.1 Saros-API

Der erste Schritt der Neuimplementierung war es, die Ereignisse, die der Saros-Core meldet, zu verarbeiten und die übergebenen Daten abzuspeichern. Für die Kommunikation mit dem Saros-Core gab es in der alten Lösung bereits eine zentrale Schnittstelle, die Saros-API. Diese konnte ich mit nur geringfügigen Anpassungen übernehmen.

Jedoch gab es in der alten Lösung für jeden Renderer im Backend ein Gegenstück im Frontend, dieses Gegenstück bestand aus einem *Model*. In dem Model wurden die Daten validiert und abgespeichert. Diese Architektur ist sehr solide und skalierbar, ich finde jedoch, dass sie dem Umfang der Applikation nicht angebracht ist. Die Daten zu validieren halte ich für unnötig, da dies zwangsläufig bereits Java-seitig geschieht. Ebenfalls sehe ich keine Notwendigkeit, die Daten an verschiedenen Orten zu speichern. Eine strenge Trennung der unterschiedlichen Daten ist meiner Meinung nach in großen Projekten von Vorteil, bei kleinen bedeutet es jedoch lediglich einen größeren Overhead und eine schlechtere Übersicht über das Projekt.

Ich ziehe daher die Lokalität der Daten vor: In der neuen Lösung gibt es nur eine einzigen Ort, an dem Informationen über den Zustand des Saros-Core liegen: Den *SarosStore*. Jedes mal wenn die Saros-API ein Ereignis meldet, z. B. „updateState“, werden die Daten in dem Store aktualisiert. Die betroffenen Daten habe ich alle als MobX-*Observable* deklariert, damit die View bei Änderungen automatisch aktualisiert

wird. Auf diese Weise bleibt der Zustand der HTML-GUI immer synchron mit dem Zustand des Saros-Core.

6.2 Die ersten Features

Nun galt es, die bestehende Funktionalität der HTML-GUI wiederherzustellen. Glücklicherweise ist Syntax der in der alten HTML-GUI verwendeten *Template-Engine Handlebars* der von JSX sehr ähnlich. Ich konnte daher Teile der *hbs*-Dateien mit geringer Modifikation wiederverwenden.

Die Implementierung folgender Features verlief sehr gradlinig und ich traf dabei auf keine nennenswerten Probleme: Accounts anzeigen, Kontakte anzeigen und Kontakte hinzufügen. Die Saros-API stellt eine Liste an Kontakten und an Accounts bereit, die es nur noch mit Hilfe einer React-Komponente darzustellen galt. Für das Hinzufügen von Kontakten war lediglich ein Formular erforderlich, das beim Absenden die dafür vorgesehene Browser-Function aufruft.

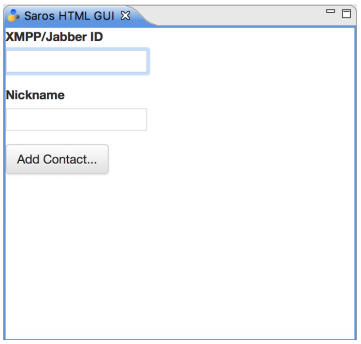

Abbildung 9: Kontakte hinzufügen

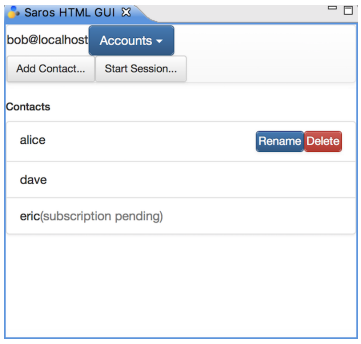

Abbildung 10: Kontakte anzeigen

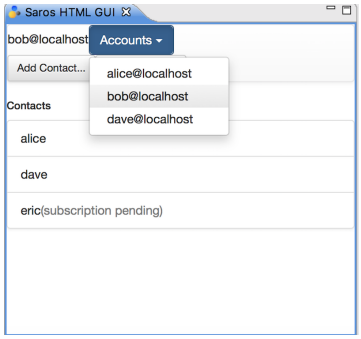

Abbildung 11: Account auswählen

6.3 Die Wizard-Komponente

Mehrere der in der Saros-GUI verwendeten Komponenten sind sogenannte *Wizards*, darunter sind der *Start-Session-Wizard*, sowie der *Configuration-Wizard* und der *Join-Session-Wizard*, die bisher in der HTML-GUI fehlen. Ein *Wizard* leitet den Nutzer Schritt für Schritt durch einen Konfigurationsprozess. In jedem Schritt trägt der Nutzer dann Informationen ein, die für die Konfiguration erforderlich sind. Ist der Nutzer fertig mit einem Schritt, kann er den *Next*-Button klicken und gelangt zum nächsten. Ist er auf der letzten Seite angelangt, kann er die Konfiguration mit dem *Finish*-Button beenden

Weil dieses Konzept in der GUI mehrmals vorkommt, bot es sich an, eine wiederverwendbare *Wizard*-Komponente zu entwickeln. Die Wizard-Komponente kann mit *Step*-Komponenten gefüllt werden. Dem Step kann ein Titel, sowie eine *Content*-Komponente übergeben werden, die er anzeigen soll. Anfangs versuchte ich, die in den einzelnen Steps eingetragenen Daten in der Wizard Komponente zu verwalten, dies

war jedoch sehr unpraktisch in Fällen, bei denen die Daten der verschiedenen Seiten voneinander abhingen (Wie es im später erwähnten *Configuration-Wizard* der Fall war). Ich entschied daher, dass die *Wizard*-Komponente lediglich die Funktion haben sollte, die Seiten anzuzeigen, und ein Callback aufzurufen, wenn die Konfiguration vom Nutzer als beendet erklärt wird. Für den Rest sind die *Content*-Komponenten selbst verantwortlich.

6.4 Start-Session-Wizard

Der *Start-Session-Wizard* war der erste Anwendungsfall meiner zuvor entwickelten *Wizard*-Komponente. Die Funktion dieses Wizards ist es aus technischer Sicht, ein Informationspaket zu schnüren, das dann der „sendInvitation“ Java-Schnittstelle übergeben wird. Dieses Paket enthält zum einen Informationen darüber, welche Dateien/Projekte geteilt werden sollen und zum anderen die Kontakte, mit denen gearbeitet werden soll. Wie gesagt sind die einzelnen Schritte des Wizards selbst dafür verantwortlich, die Daten weiterzugeben. Ich erstellte dazu einen neuen *Store*, den „SessionUIStore“, in dem der aktuelle Zustand des Informationspakets gespeichert wird. Beim Klick auf den Finish-Button wird dieses Paket an die Saros-API gesendet.

6.4.1 Projekte auswählen

Um die Projekte auszuwählen brauchte es eine Ansicht, die die Projekte des Nutzers als Datei-Baum darstellt. Bei der Implementierung dieser Ansicht sah ich mich das erste mal einer Herausforderung ausgesetzt. Ich musste aus den von der Saros-API übergebenen „projectTrees“ irgendwie eine React-Komponente rendern. Diese „projectTrees“ haben eine Baumstruktur. Jeder Knoten des Baumes, der kein Blatt ist, stellt dabei einen Ordner dar und jedes Blatt entweder einen leeren Ordner oder eine Datei. Ein Ordner kann wiederum beliebig viele Knoten enthalten. Glücklicherweise gibt es diverse Pakete auf *NPM*, die sich dieses Problems annehmen. Das Paket „rc-tree“ [6] schien mir dabei am ehesten dem gewünschten Ergebnis zu entsprechen. Es kann Ordner und Dateien mit Hilfe der React-Komponente „TreeNode“ darstellen, und bietet zudem die Funktion, diese mit einer Checkbox an- und abzuwählen.

Die bloße Darstellung des Datei-Baumes war damit einfach: Ich definierte eine rekursive Funktion namens „renderTreeNode“, die einen Knoten des Dateibaumes übergeben bekommt. Diesen Knoten stellt sie als „TreeNode“ dar und für jeden untergeordneten Knoten ruft sie wiederum sich selbst auf. Die Ergebnisse dieser Aufrufe werden dann als Kinder des „TreeNode“ eingefügt. Ist der Knoten eine Datei, wird der zugehörige „TreeNode“ auch als Datei dargestellt.

Was sich jedoch als schwieriger herausstellte war das an- und abwählen der Knoten. „rc-tree“ ruft beim An- oder Abwählen eines Knotens ein Callback mit den momentan selektierten Knoten auf. Ein Knoten wird dabei nicht direkt referenziert, sondern durch einen Baumpfad identifiziert. Beispielsweise hätte der dritte Knoten des zweiten Knotens der Wurzel den Pfad „2-3“. Diese Liste an Pfaden sei im Folgenden als

selectedPaths bezeichnet. Diese Pfade gilt es danach wieder auf den ursprünglichen Projektbaum zu übertragen. Ein Knoten, dessen Pfad in den selectedPaths enthalten ist, musste als „isSelectedForSharing“ markiert werden und alle anderen demzufolge nicht. Ich löste dies, indem ich eine Funktion schrieb, die über den Projektbaum traversiert, und für jeden Knoten eine Callback-Funktion mit dessen Pfad aufruft. In dem Callback wird dann „isSelectedForSharing“ auf „true“ gesetzt, falls der Pfad in selectedPaths vorkommt und ansonsten auf „false“. Dieser Prozess wird lediglich einmal ausgeführt, bevor das Paket mit den ausgewählten Dateien und Projekten, sowie den ausgewählten Kontakten an die „sendInvitation“ Java-Schnittstelle übergeben wird.

6.4.2 Kontakte auswählen

Das Auflisten von Kontakten war ein Problem, das ich bereits bei der Implementierung der Kontaktliste der Hauptansicht gelöst hatte. Was fehlte, war lediglich die Möglichkeit, diese Kontakte an- und abzuwählen. Dazu erweiterte ich die „ContactList“ Komponente so, dass man ihr optional eine Menge an IDs von Kontakten übergeben kann. Ist ein Kontakt in dieser Menge enthalten, wird er farbig hinterlegt. Die Menge der ausgewählten Kontakte speicherte ich wieder in dem „SessionUIStore“. Durch einen Klick auf einen Kontakt aus der Liste wird dieser der Menge hinzugefügt, beziehungsweise aus ihr entfernt.

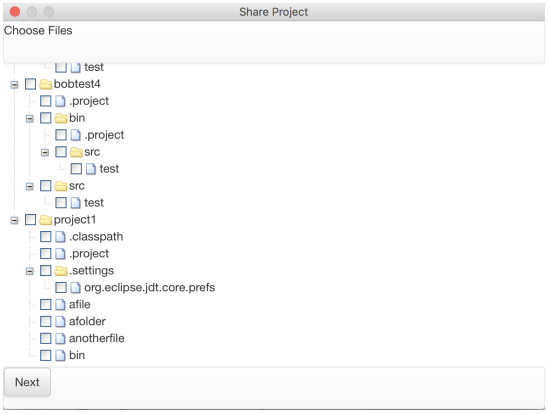

Abbildung 12: Projekte auswählen

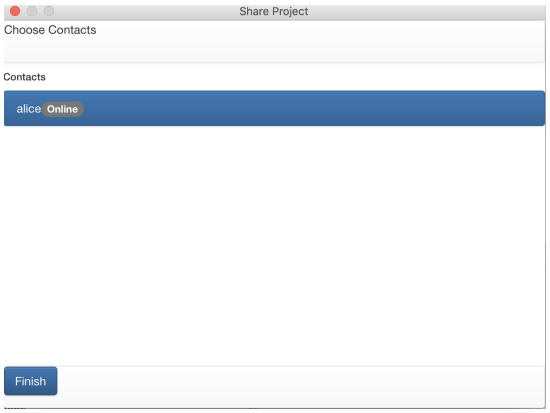

Abbildung 13: Kontakte auswählen

Der neue Start-Session-Wizard

6.5 Kontakte umbenennen und löschen

In der Kontaktliste der Hauptansicht musste man des Weiteren Kontakte umbenennen und löschen können. Dies erforderte das Anzeigen von Buttons in der Kontaktliste. Diese Buttons sollten aber natürlich nur in der Hauptansicht, und nicht im Start-Session-Wizard erscheinen. Deswegen erweiterte ich die Kontaktliste erneut, sodass man ihr optional eine *Operations*-Komponente übergeben konnte. In der Hauptansicht erstellte ich also eine Komponente die einen *Delete*- sowie eine *Rename*-Button darstellt und übergab diese der Kontaktliste.

Auch für das Umbenennen der Kontakte hatte ich bereits fast alle Komponenten an der Hand, die ich brauchte: Die Ansicht zum Hinzufügen von Kontakten unterschied sich nur sehr geringfügig von der, die ich für das Umbenennen von Kontakten vorsah. Dazu musste ich der *Add-Contact-View* aber irgendwie mitteilen, dass sie gerade einen Kontakt umbenennt und nicht hinzufügt. Ich führte dazu ein allgemeineres Konzept ein: Beim Öffnen einer View kann ein *Intent* übergeben werden, der Informationen über die darzustellende Ansicht enthält. Im Falle der Add-Contact-View ist das lediglich ein „isRename"-Flag.

6.6 Rückblick

An diesem Punkt hatte ich die Funktionalität der bestehenden HTML-GUI ungefähr wiederhergestellt. Ein guter Zeitpunkt, um meine Entscheidung, die GUI mit React neuzuimplementieren zu würdigen: Ich habe die Anzahl der Code-Zeilen verglichen, die meine Lösung zu diesem Zeitpunkt und die alte Lösung hatten (Für die genaue Vorgehensweise siehe Anhang A.3). Das Ergebnis war ein erstaunliches 915 zu 1741 für meine Lösung. Mein Code hatte also annähernd dieselbe Funktionalität mit kaum mehr als der Hälfte der Codezeilen. Auch wenn die Anzahl der Codezeilen natürlich per se nicht aussagekräftig ist, ist ein so massiver Unterschied nicht von der Hand zu weisen.

7 Erweiterung

Die Hauptansicht der HTML-GUI war an diesem Punkt zwar wieder auf dem Stand der Alten, jedoch fehlten ihr immer noch essentielle Features. Als nächsten Schritt wollte ich die Hauptansicht daher so weit wie möglich an die der originalen Saros-GUI angleichen. In dieser kann man neben dem Löschen und Umbenennen von Kontakten noch andere Operationen ausführen: Mit „Work together on..." kann man ein Projekt direkt mit einem Kontakt teilen, ohne den *Start-Session-Wizard* zu durchlaufen. Mit „Chat" kann ein Chat mit dem Kontakt gestartet werden – Eine Funktion die in der Saros HTML-GUI bisher gänzlich fehlt. Auch die „Send File" Funktion gibt es in der HTML-GUI noch nicht. Des Weiteren fehlten in der HTML-Version der Hauptansicht Informationen über die laufende Session.

Die „Chat" und „Send File" Funktionen sind meiner Einschätzung nach nicht Bestandteil der Kernfunktionalität von Saros, weshalb ich diese meinen Nachfolgern überlassen möchte. Die „Work together on..." Funktion sehe ich fürs erste als entbehrlich an, da dasselbe auch durch den Start-Session-Wizard erreicht werden kann. Weil die *Session-Info* nach meiner Einschätzung den größten Mehrwert bieten würde, entschied ich mich die Implementierung ebendieser anzugehen.

7.1 Session-Info

Funktion der *Session-Info* ist es, anzuzeigen mit welchen Kontakten der Nutzer sich gerade in einer Session befindet, wer welche Rolle hat und welche Datei die Nutzer gerade jeweils geöffnet haben.

In der Saros-API, die ich von der alten Lösung übernommen habe, fehlte jedoch bisher eine Schnittstelle, die Informationen über die laufende Session liefert. Ich entschied mich die Implementierung dieser Schnittstelle vorerst hinten anzustellen und zuerst die eigentliche View-Komponente zu implementieren. Dazu kreierte ich zunächst einen Test-Datensatz der die benötigten Informationen enthält. Dieser Datensatz besteht aus einer Liste an *Session-Membern*, die jeweils einen Namen, eine ID, eine Farbe, sowie einen *isHost*-Flag haben:

```
1 {
2   "members": [
3     {  "openedFile": "somefile.txt", "jid": "alice@localhost", "
          displayName": "alice", "isHost": true, "color": "blue" },
4     {  "openedFile": null, "jid": "bob@randomhost", "displayName": "bob",
           "color": "green" }
5   ]
6 }
```

Damit hatte ich das Datenformat spezifiziert, das die HTML-GUI in Zukunft von der Java-Schnittstelle erwartet. Auf der Basis dieser Annahme konnte ich nun die eigentliche View-Komponente schnell implementieren.

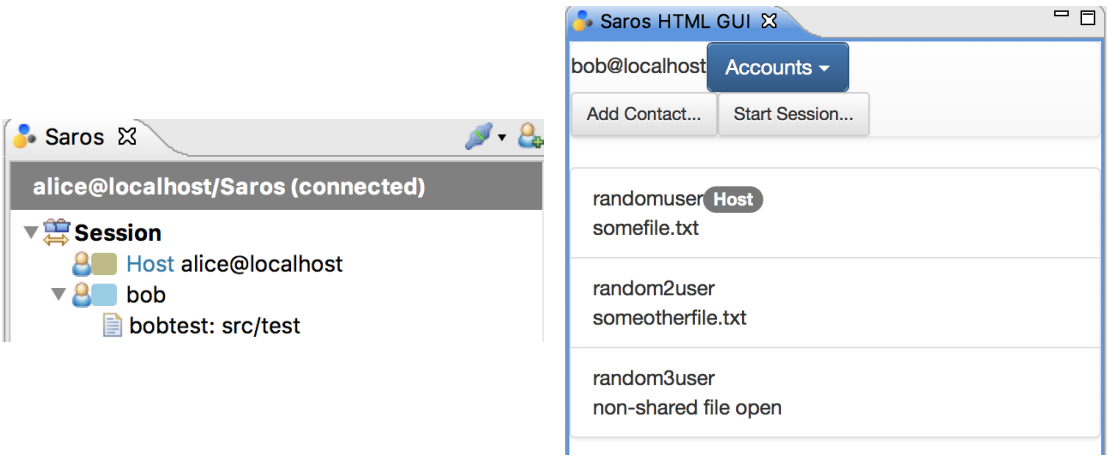

Abbildung 14: Die Session-Info in der Hauptansicht

Links die Java-GUI, rechts die HTML-GUI

Bisher hatte ich mich mit dem Java-Code von Saros kaum auseinandergesetzt, da es bis zu diesem Zeitpunkt auch nicht nötig war. Um die Java-Schnittstelle der Session Info zu implementieren, musste ich mich jedoch in diese Welt vorwagen.

Ich wollte für die Session einen *Renderer* implementieren, der alle für die *Session-Info* benötigten Informationen sammelt und an die HTML-GUI weiterleitet. Die Stellen an denen diese Informationen liegen, bieten oft eine *Listener*-Schnittstelle, bei der sich an-

dere Klassen registrieren können, um über Ereignisse informiert zu werden. Der Plan war also den Renderer als Listener für die *Session-Info* zu registrieren. Ich stieß dabei jedoch auf ein Problem: Es gab nicht *den* einen Ort, an dem Informationen über die laufende Session lagen. Die benötigten Informationen waren ein Sammelsurium, das ich mir aus vielen verschiedenen Stellen zusammenklauben musste. Eine schwierige und langwierige Aufgabe für mich, der ich in dem Java-Code von Saros ein Fremder war. Ich realisierte, dass ich die Zeit, die ich mit dieser Aufgabe verbrachte effizienter hätte Nutzen können, indem ich mich auf das konzentriere, was ich bereits kannte: Die HTML-GUI. An diesem Punkt entschied ich mich, die Implementierung der Schnittstelle erneut nach hinten zu schieben und stattdessen die Benutzeroberfläche weiterzuentwickeln.

7.2 Configuration-Wizard

Der *Configuration-Wizard* war eine weitere Komponente der Saros-GUI, die ich als essentiell ansah. Er ermöglicht dem Nutzer die Ersteinrichtung von Saros. Schritt für Schritt kann er dort einen XMPP-Account anlegen, eine Farbe zur Identifikation in den Sessions wählen und weitere Konfigurationen vornehmen. Der Wizard öffnet sich, sobald im Saros Menü auf die „Start Saros Configuration...“ Schaltfläche geklickt wird.

Auch beim *Configuration-Wizard* fehlte bisher eine Java-Schnittstelle. Konkret benötigte ich eine neue *Browser-Page*, in der der *Configuration-Wizard* angezeigt werden kann und erneut eine *Browser-Function* zum Übergeben des Informationspakets am Ende der Konfiguration. Des Weiteren würde auch ein neuer *Renderer* nötig sein, der die bisherige Konfiguration an die HTML-GUI übergibt. Da ersteres für die Entwicklung der Benutzeroberfläche zwingend notwendig war, machte ich erneut einen Ausflug in die Java-Welt. Dieses mal war es relativ einfach, den gewünschten Effekt zu erzielen. Ich musste dazu eine neue *Browser-Page* erstellen, wie es sie auch schon für die Hauptansicht und den Start-Session-Wizard gab. Danach musste ich die Funktion, die aufgerufen wird, wenn der Start-Configuration-Button geklickt wird, so modifizieren, dass sie, wenn die HTML-GUI aktiviert ist, statt dem bisherigen Fenster die neue *Browser-Page* öffnet.

Nachdem dies erledigt war, begann ich mit der Implementierung der View-Komponente. Dazu verwendete ich erneut die zuvor geschriebene Wizard-Komponente und einen weiteren Store, den „ConfigurationUIStore“, in dem, wie zuvor auch schon im „SessionUIStore“ die Konfiguration zwischengespeichert wird. Auch wenn die Benutzeroberfläche in Ihrem jetzigen Zustand noch nicht fertig ist, empfand ich sie ungefähr der des originalen Configuration-Wizards nach (vgl. Abbildung 7.2).

Ich definierte erneut Datenschemen für die Kommunikation mit dem Saros-Core, entschied mich jedoch auch hier die Grenze meiner Arbeit im JavaScript-Teil zu ziehen.

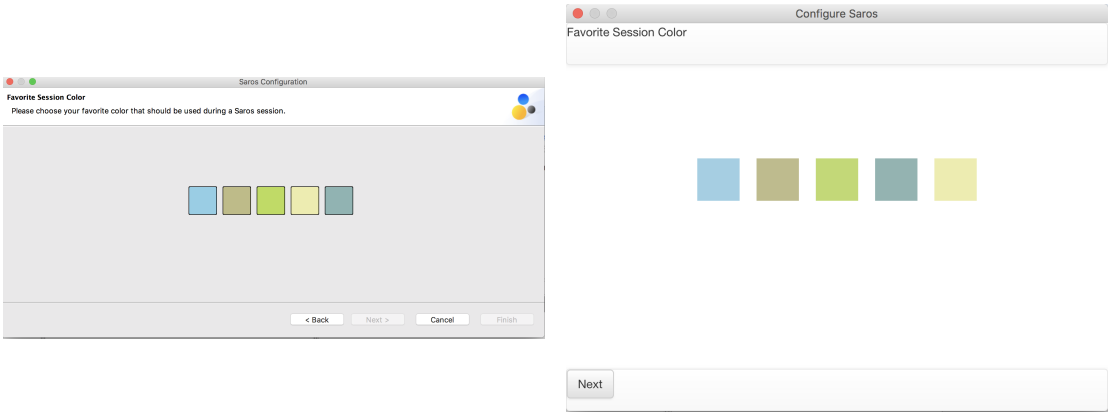

Abbildung 15: Der Farbauswahl-Schritt des Configuration-Wizard

Links die Java-GUI, rechts die HTML-GUI

8 Automatisiertes Testen

Ich hatte an diesem Punkt eine beachtliche Menge an Features implementiert, jedoch fehlte dem Software-Konstrukt eine solide Testabdeckung. Im Laufe meiner Arbeit an dem Saros-Projekt lernte ich viel dazu, insbesondere die starke Qualitätssicherung wie das ausgiebige automatische Testen und die gründlichen Code-Review-Prozesse beeindruckten mich. Mir waren diese Vorgehensweisen zwar durch mein Studium bekannt, ich hatte sie jedoch noch nie in Anwendung gesehen. Von diesen Erkenntnissen inspiriert überdachte ich mein anfängliches Ziel.

Die Implementierung automatischer Tests würde den Fortschritt der HTML-GUI zwar kurzfristig verlangsamen, jedoch bergen diese das Potential einer langfristigen Beschleunigung der Entwicklung. Die Tests verringern die Wahrscheinlichkeit, dass ein Entwickler versehentlich eine Regression der Funktionalität verursacht und helfen ihm/ihr so, sich auf neue Features zu konzentrieren.

Da ich den Code, den ich selbst geschrieben habe, auch am besten verstehe, erschien es mir nur sinnvoll, dass ich auch derjenige sein muss, der dafür automatische Tests implementiert.

Ich änderte daher meine Anfänglichen Ziele dahingehend, dass ich nicht einfach nur so viele Features wie möglich implementieren, sondern diese auch mit automatischen Tests abdecken möchte.

Aus diesem Grund sah ich von der Implementierung weiterer Features ab und konzentrierte mich stattdessen auf das Testen.

8.1 Verwendete Libraries

In der alten HTML-GUI wurde das Test-Framework *Mocha*[27] verwendet, um Unit-Tests durchzuführen. Ich verwendete weiterhin *Mocha*, da ich es bereits kannte und

[27] https://mochajs.org/

gute Erfahrungen damit gesammelt habe. Mocha ermöglicht das einfache Spezifizieren von Testfällen mit den Schlüsselworten „describe“ und „it“. „describe“ definiert dabei eine Kategorie von Tests, z. B. dass alle Tests die sich darin befinden zu einer bestimmten React Komponente gehören. Das „it“-Schlüsselwort beschreibt einen Test, von dem ein bestimmtes Ergebnis erwartet wird. Um *Assertions* (d.h. Ausdrücke, die bei Nichteinhalten bestimmter Bedingungen zu einem Error im Programm führen) in den Tests zu machen verwende ich *Chai*[28] mit dem Schlüsselwort „expect“. Beispiel:

```
1 describe('add', () => { // Sei 'add' eine Funktion die zwei Zahlen
      addiert
2   it('correctly adds numbers', () => {
3     expect(add(1,2)).to.equal(3)
4   })
5 })
```

Ich verwende zudem die Library *Sinon*[29]. Diese bietet vor allem einfache *Spy*-Funktionen, die Buch darüber führen, ob und mit welchen Parametern sie aufgerufen wurden. Diese *Spies* können in den Programmfluss injiziert werden, um festzustellen was passiert ist.

Zum Testen von React-Komponenten nutzte ich die Library *Enzyme*[30]. Mit Enzyme lassen sich React-Komponenten in sogenannten *Wrappern* rendern. Die Wrapper stellen viele nützliche Funktionen bereit, um den Zustand der React-Komponente zu untersuchen. Das folgende Beispiel zeigt einen Test der *Greeting*-Komponente aus Beispiel 1.

```
1  import { shallow } from 'enzyme'
2
3  describe('<Greeting />', () => {
4      const wrapper = shallow(
5          <Greeting who="Alice" />
6      )
7      it('Greets the person', () => {
8          expect(wrapper.text()).to.equal('Hello Alice!')
9      })
10 })
```

8.2 Die Tests

Bisher wurden lediglich eine Hand voll Tests implementiert, um die Auswahl eines Nutzeraccounts zu testen. Anstatt jedoch die einzelnen Komponenten bis ins Detail zu testen, wollte ich die verbliebene Zeit nutzen, um die für die Funktionsfähigkeit kritischen Punkte zu testen.

Um sicherzustellen, dass die HTML-GUI als ganzes funktioniert, erstellte ich daher eine Pseudo-Saros-API, die, anstatt mit dem Saros-Core zu kommunizieren, lediglich

[28] http://chaijs.com/
[29] http://sinonjs.org/
[30] http://airbnb.io/enzyme/

aus *Spies* besteht. So konnte ich feststellen, ob die HTML-GUI die Java-Schnittstellen korrekt anspricht.

In der Hauptansicht empfand ich folgende Tests als besonders dringend:

1. Ein Klick auf den „Add Contact“-Button löst einen Aufruf der Funktion aus, die für das Öffnen der Ansicht verantwortlich ist
2. Ein Klick auf den „Start Session“-Button ruft die Saros-API Schnittstelle „showStartSessionWizard“ auf

Zudem wollte ich durch einen Test feststellen, ob das Hinzufügen von Kontakten ordnungsgemäß funktioniert. Die Implementierung dieser Tests war einfach, denn ich konnte den Aufruf der Funktionen mit der Pseudo-Saros-API nachverfolgen.

Weil die Wizard-Komponente zudem an zwei Stellen verwendet wird und zukünftig noch mehr zum Einsatz kommt, sah ich das automatische Testen ebendieser als besonders nützlich an. Ich stellte dazu sicher, ob der Seitenwechsel funktioniert und ob die aktuelle Seite korrekt angezeigt wird. Zudem testete ich, ob beim Klick auf den *Finish*-Button auf der letzten Seite das entsprechende Callback aufgerufen wird.

Auch den *Start-Session-Wizard* sah ich aufgrund seiner Komplexität als besonders testenswert an. In der *Choose-Files-View* testete ich dafür, ob die vom Saros-Core übergebenen „projectTrees“ korrekt dargestellt werden. Dazu speicherte ich mir einen Beispieldatensatz, der dem Format der „projectTrees“ entspricht. Diese Daten übergab ich dann an die *Choose-Files-View* und testete stichprobenartig, ob einzelne Dateien dargestellt werden. Zudem überprüfte ich, ob das Auswählen der Dateien funktioniert. Am wichtigsten war jedoch der Test, ob das Datenpaket, das der *Start-Session-Wizard* schnürt, das korrekte Format hat und ob es korrekt die Konfiguration des Nutzers widerspiegelt. Dazu nutzte ich erneut den Beispieldatensatz und implementierte einen Test, der automatisch Aktionen auf dem Start-Session-Wizard ausführt. Anschließend vergleicht er das entstehende Datenpaket mit einem weiteren Testdatensatz, der die gewünschten Modifikationen enthält.

9 Fazit

Abschließend werde ich nun noch einmal die Ergebnisse dieser Arbeit zusammenfassen und in Kontrast zu den gesetzten Zielen setzen, sowie meine Entscheidungen rückblickend bewerten.

Die wohl größte Entscheidung, die ich während dieser Arbeit getroffen habe, war die bestehende HTML-GUI aufgrund der unpassenden Framework-Wahl größtenteils zu verwerfen und mit *React* neu zu implementieren. Obwohl diese Entscheidung viel Arbeit mit sich brachte, habe ich dennoch keine Zweifel, dass sie richtig war:

Die neue Lösung hat einen Bruchteil des Code-Umfangs der alten, bei größerer Funktionalität. Auch die Code-Qualität konnte ich meiner Einschätzung nach massiv steigern. Es gibt kaum Boilerplate und viel von dem Code wird effizient an mehreren Stellen eingesetzt. Ich konnte wiederauftretende Konzepte wie den *Wizard* und die *Kontaktliste* extrahieren und wiederverwenden.

Die Architektur des HTML-GUI wurde vereinfacht, sodass sie dem Umfang der Applikation angemessener ist. Diese Vereinfachung macht den Kontrollfluss des Codes wesentlich leichter nachvollziehbar und wartbar.

Durch die höhere Abstraktion von *React* und *MobX* gegenüber *Ampersand* war ich in der Lage mich auf das wesentliche zu konzentrieren und innerhalb kürzester Zeit die bestehende Funktionalität wiederherzustellen und zu übertreffen.

Ich bin zuversichtlich, dass der Code der neuen HTML-GUI deutlich verständlicher ist, als der der alten. Dies wird zukünftigen Entwicklern nicht nur helfen, sich schnell in den Code einzuarbeiten, sondern ihn auch schneller Erweitern zu können.

Während der Arbeit realisierte ich, dass für das Saros-Projekt nicht Quantität, sondern Qualität entscheidend ist und entschied daher automatische Tests, anstatt noch mehr Features zu implementieren. Diese Entscheidung war rückblickend ebenfalls richtig. Meine Tests können in den *Continuous-Intergration*-Prozess von Saros eingebettet werden und werden späteren Entwicklern helfen, mit einer größeren Sicherheit und Geschwindigkeit an der HTML-GUI zu arbeiten.

Auf der anderen Seite denke ich, dass ich in der mir zu Verfügung stehenden Zeit wahrscheinlich mehr hätte schaffen können. Ich investierte viel Zeit und Mühe mit dem Versuch, Java-Schnittstellen für den *Configuration-Wizard* und die *Session-Info* zu implementieren, nur um mir letztendlich einzugestehen, dass ich die Komplexität des Java-Teils der Applikation unterschätzt hatte.

Hinzu kam, dass das Code-Review-System, das für das Projekt verwendet wird, mich teilweise überfordert hat. Es war schwer gleichzeitig aktiv an der Diskussion über die verschiedenen Patches teilzunehmen, den Verbesserungsvorschlägen der anderen Entwickler nachzukommen und währenddessen noch neue Features zu implementieren. Dies führte zu einem Patch-Stau, weshalb viele meiner Patches zum Zeitpunkt der Abgabe dieser Arbeit noch offen sind.

Alles in allem bin ich mit dem Ergebnis der Arbeit zufrieden. Ich habe die Ziele, die

ich mir anfangs gesetzt habe, nicht in vollem Umfang erfüllt und dennoch glaube ich, dass ich durch die automatischen Tests letztendlich einen noch größeren Mehrwert geschaffen habe, als eine größere Menge an Features ihn bieten würde.

10 Unvollständige und fehlende Features

Im Folgenden werde ich noch einmal die Baustellen der HTML-GUI auflisten, die ich nach meiner Arbeit offen lies. Ich empfehle meinen Nachfolgern, hier mit ihrer Arbeit anzusetzen. Ich gebe zudem eine Einschätzung des Aufwands der Implementierung und des Nutzens dieser Features.

Main-View In der Hauptansicht muss noch verhindert werden, dass der Nutzer ohne sich eingeloggt zu haben, Kontakte hinzufügen und den Start-Session-Wizard öffnen kann. Ebenso sollte er diesen nicht öffnen können, wenn keine seiner Kontakte online sind. – ***Aufwand: Gering, Nutzen: Groß***

Configuration-Wizard Für den Configuration-Wizard fehlen, wie bereits erwähnt, noch die Java-Schnittstellen. Es wird ein *Renderer* benötigt, der die bisherige Konfiguration an den Wizard übergibt, sowie eine *Browser-Function*, die beim Klick auf den *Finish*-Button ausgelöst wird und die neue oder initiale Konfiguration speichert. Zudem benötigt die Benutzeroberfläche noch einen letzten Feinschliff. – ***Aufwand: Groß, Nutzen: Essentiell***

Session-Info Auch der Session-Info fehlt noch ein *Renderer*, der die benötigten Informationen bündelt und an die HTML-GUI weiterleitet. Das Schema der benötigten Daten ist bereits durch die Testdaten spezifiert. Es fehlt zudem die Möglichkeit, eine Session wieder zu verlassen. Auch hierfür wird eine *Browser-Function* benötigt. Das Erscheinungsbild der Session-Info ist bisher rein funktional und sollte an die Java-GUI angeglichen werden. – ***Aufwand: Groß, Nutzen: Essentiell***

Work together on... Diese Option fehlt in den Kontaktoperationen neben *Rename* und *Delete*. (siehe Abschnitt 7) – ***Aufwand: Mittel, Nutzen: Gering***

Join-Session-Wizard Ein weiterer Wizard, der erscheint, wenn der Nutzer zu einer Session eingeladen wurde. – ***Aufwand: Sehr Groß, Nutzen: Essentiell***

Send File Eine weitere Kontaktoperation. Damit kann einem bestimmten Kontakt eine Datei gesendet werden. – ***Aufwand: Groß, Nutzen: Gering***

Preferences Um die plattformübergreifende Funktionsfähigkeit der Saros-GUI zu gewährleisten, muss auch die *Preferences*-Seite mit JavaScript und HTML implementiert werden. – ***Aufwand: Extrem groß, Nutzen: Essentiell***

Literaturverzeichnis

[1] *ampersand-state* auf NPM – Eines der populärsten Pakete der Ampersand Familie. https://www.npmjs.com/package/ampersand-state.

[2] *@angular/core* auf NPM – Das Kernpaket einer jeden Angular Applikation. https://www.npmjs.com/package/@angular/core.

[3] Die Flux-Architektur. https://facebook.github.io/flux/docs/in-depth-overview.html#structure-and-data-flow.

[4] JavaScript ES6 Features. http://es6-features.org.

[5] MobX Documentation. https://mobx.js.org/index.html.

[6] rc-tree auf GitHub. https://github.com/react-component/tree.

[7] React auf GitHub. https://github.com/facebook/react.

[8] React Dokumentation. https://facebook.github.io/react/.

[9] React Installation Guide. https://facebook.github.io/react/docs/installation.html.

[10] *react* auf NPM. https://www.npmjs.com/package/react.

[11] Redux Documentation. http://redux.js.org/.

[12] The Power Of Not Mutating Data. https://facebook.github.io/react/docs/optimizing-performance.html#using-immutable-data-structures.

[13] What is Cache-Busting. https://www.keycdn.com/support/what-is-cache-busting/.

[14] Why did we build React? https://facebook.github.io/react/blog/2013/06/05/why-react.html#react-isnt-an-mvc-framework.

[15] Christian Cikryt. Evaluating the Use of a Web Browser to Unify GUI Development for IDE Plug-ins. Masterarbeit, 2015.

[16] R. Djemili. Entwicklung einer Eclipse-Erweiterung zur Realisierung und Protokollierungverteilter Paarprogrammierung. Diplomarbeit, Freie Universität Berlin, Inst. für Informatik, 2006.

[17] Sacha Greif. The State Of JavaScript: Front-End Frameworks. http://stateofjs.com/2016/frontend/, August 2016.

[18] Ben McCormick. What are mutable and immutable data structures? https://benmccormick.org/2016/06/04/what-are-mutable-and-immutable-data-structures-2/, 2016.

[19] Bastian Sieker. User-Centered Development of a JavaScript and HTML-based GUI for Saros. Masterarbeit, 2015.

[20] Maria Spiering. Verbesserung der Usability von Saros unter Verwendung eines User-Centered Design Ansatzes. Masterarbeit, Freie Universität Berlin, Inst. für Informatik, 2012.

[21] Nina Weber. Einstiegserleichterung für die Weiterentwicklung und Erweiterung der JavaScript- und HTML-GUI von Saros. Bachelorarbeit, 2016.

[22] Kate Wu. Taking a peek at functional programming with redux. https://wiredcraft.com/blog/redux-functional-programming/, May 2016.

A Anhang

A.1 Eingereichte Gerrit-Patches

A.1.1 Vollendet

#3333 [FEATURE][HTML] Started rewriting the HTML GUI with react

#3347 [BUILD][HTML] optimize build process, cleanup

#3348 [INTERNAL][HTML] Prepared HTML-GUI for being split into multiple views

#3349 [FEATURE][HTML] Implemented 'Add Contact' view

#3350 [FIX][HTML] Choose account not working

#3351 [FEATURE][HTML] Implemented reusable Wizard Component

#3352 [REFACTOR][HTML] Move contactlist into component folder for being reused

A.1.2 Offen

#3353 [FEATURE][HTML] Implemented session wizard UI

#3354 [NOP][HTML] Improved JavaScript linting

#3355 [FEATURE][HTML] Create browser page for configuration wizard

#3375 [FEATURE][HTML] Implement Configuration Wizard UI

#3376 [INTERNAL][HTML] Refactor and fix AddContactView

#3377 [FEATURE][HTML] Implement „Rename“ and „Delete“ operations for contacts

#3378 [FEATURE][HTML] Add cancel buttons

#3379 [FEATURE][HTML] Add „Back“ button to Wizard

#3380 [FEATURE][HTML] Implement UI for running session info

#3382 [HTML][JUNIT] Setup unit-testing environment for HTML-GUI

A.2 State-Management-Library Vergleich

```
import {  connect   } from 'react-redux'
import { createStore } from 'redux'
import Counter from './Counter'
// This is normally in a seperate File (actionTypes)
const INCREASE_COUNT = 'INCREASE_COUNT'
const DECREASE_COUNT = 'DECREASE_COUNT'
// This is normally in a seperate File (actions)
const increaseCount = () => ({ type: INCREASE_COUNT })
const decreaseCount = () => ({ type: DECREASE_COUNT })
// This is normally in a seperate File (the container container)
const mapStateToProps = count => ({ count })
const mapDispatchToProps = dispatch => ({
  onClickIncrease: () => dispatch(increaseCount()),
  onClickDecrease: () => dispatch(decreaseCount())
})

const reducer = (count = 0, action) => {
  switch (action.type) {
    case INCREASE_COUNT:
      return count + 1
    case DECREASE_COUNT:
      return count - 1
    default:
      return count
  }
}

const ReduxCounter = connect(mapStateToProps, mapDispatchToProps)(Counter
  )
export default ReduxCounter
export const store = createStore(reducer)
```

```
import { inject, observer } from 'mobx-react'
import { observable, action } from 'mobx'
import Counter from './Counter'

class CounterStore {
  @observable count = 0
  @action increase = () => {
    this.count = this.count + 1
  }
  @action decrease = () => {
    this.count = this.count - 1
  }
}

export const store = new CounterStore()

const mapStoresToProps = ({ store }) => ({
  count: store.count,
  onClickIncrease: store.increase,
  onClickDecrease: store.decrease
})

const MobxCounter = inject(mapStoresToProps)(observer(Counter))
```

```
export default MobxCounter
```

```
import React from 'react'
import EventEmitter from 'events'
import Counter from './Counter'

const emitter = new EventEmitter()
const dispatch = (action) => emitter.emit('dispatch', action)
const INCREASE_COUNT = 'INCREASE_COUNT'
const DECREASE_COUNT = 'DECREASE_COUNT'

class CounterStore {

  count = 0

  constructor() {
    emitter.on('dispatch', this.reduce)
  }

  reduce = (action) => {
    switch (action.type) {
      case INCREASE_COUNT:
        this.count = this.count + 1
        break;
      case DECREASE_COUNT:
        this.count = this.count - 1
        break;
      default:
        return
    }
    emitter.emit('update', this.count)
  }
}

const counterStore = new CounterStore()

class NoLibCounter extends React.Component {

  constructor(){
    super()
    this.state = this.getNextState()
    emitter.on('update', () => {
      this.setState(this.getNextState())
    })
  }

  getNextState() {
    return { count : counterStore.count}
  }

  render(){
    return (
      <Counter
        onClickIncrease={() => dispatch({ type: INCREASE_COUNT })}
        onClickDecrease={() => dispatch({ type: DECREASE_COUNT })}
        count={this.state.count}
      />
```

```
56      )
      }
    }

    export default NoLibCounter
```

A.3 Codezeilen-Vergleich

Ich verglich die neue Lösung auf dem Stand von Gerrit-Patch #3377/2[31] mit der alten Lösung auf dem Stand von GitHub-Commit 201dc94[32].

Dazu habe ich für meine Lösung folgenden Unix-Shell-Befehl verwendet:

```
wc -l src/**.jsx
```

Da die View-Komponenten in meiner Lösung auch Teil der JavaScript Dateien sind, zählte ich bei der alten Lösung auch die Zeilen der Handlebars-Templates dazu. Dazu nutzte ich folgenden Befehl:

```
wc -l src/**.{js,hbs}
```

[31] http://saros-build.imp.fu-berlin.de/gerrit/#/c/3377/
[32] https://github.com/saros-project/saros/commit/201dc9455c494ee12a394bc3145a3b1c4cebfe1a