RE TS

Technischer Bericht

Remo Dörig, Joel Fisch

19. Dezember 2019

Betreuer: Prof. Dr. Farhad Mehta
HSR Hochschule für Technik Rapperswil
Studienarbeit
Studiengang Informatik – Herbstsemester 2019/2020

Strongly typed, functional languages as an alternative to the popular React + Redux stack

Abstract

Der populäre Technologie-Stack TypeScript, React und Redux implementiert eine funktional inspirierte Architektur in einer nicht primär funktionalen Programmiersprache. Konstrukte aus der funktionalen Welt müssen deshalb imitiert werden. Wären funktionale Sprachen für neue Projekte in dieser Architektur eine bessere Alternative? Zwecks Beantwortung dieser Frage wurde in TypeScript, Reason und Elm je eine Beispielapplikation entwickelt. Die Applikationen und ihre Entwicklung wurden anhand verschiedener Kriterien evaluiert und verglichen. TypeScript wurde wegen seiner Verbreitung untersucht. Elm wurde untersucht, da die Architektur ihren Ursprung in dieser Sprache hat. Reason wurde als Mittelweg und wegen der guten React-Anbindung untersucht. Reason ist momentan noch zu wenig ausgereift, könnte aber in Zukunft eine gute Alternative zu TypeScript darstellen. Die Vorteile von TypeScript (grosses Ökosystem, Nähe zu JavaScript) überwiegen für langlebige Projekte gegenüber den Nachteilen der Sprache. Elm hat eine sehr stabile Laufzeitumgebung und bietet ideale Voraussetzungen für die Implementation dieser Architektur, hat aber im Bereich der Interaktion mit JavaScript und des Ökosystems starke Einschränkungen. Wenn man sich in einem neuen Projekt für die Elm-Architektur entscheidet, ist man in den meisten Fällen mit TypeScript am besten aufgehoben. Elm bietet Vorteile, man muss sich aber den Limitierungen der Sprache und des Ökosystems klar bewusst sein.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Lay-Summary

Ausgangslage Früher waren Webseiten viel einfacher. Mittlerweile werden diese aber immer komplexer und können grosse eigenständige Software-Projekte sein. Damit die Komplexität gut bewältigt werden kann, müssen die Projekte auf der richtigen Technologie basieren.

Momentan ist eine der populärsten Technologien, um Webseiten zu erstellen, TypeScript mit React und Redux (folgend TypeScript genannt). Bei TypeScript handelt es sich um eine Programmiersprache, wogegen React und Redux Werkzeuge sind. Redux hat konzeptionell seine Wurzeln in Elm. Elm ist auch eine Programmiersprache, mit der sich Webseiten erstellen lassen. Elm erschafft sich seine eigene Welt, in welcher die Konzepte von Elm gut umgesetzt werden können. Diese Welt wird nachher in eine Webseite umgewandelt. Nebst Elm und TypeScript vergleicht diese Studie auch noch Reason (mit den gleichen Werkzeugen Redux und React wie TypeScript). Reason wird auch in eine Webseite umgewandelt; der Unterschied ist aber kleiner als bei Elm und damit könnte Reason ein guter Mittelweg sein.

Ziel Diese Studie stellt sich nun die Frage: “Welche Technologie ist für den Start eines neuen Webseitenprojektes am besten geeignet?”

Vorgehen Um die Frage dieser Studie beantworten zu können, wird die gleiche Applikation in allen drei Varianten entwickelt und evaluiert. Einzelne Entwicklungsschritte werden parallel durchgeführt und verglichen.

Ergebnisse Reason ist noch nicht genug weit entwickelt, um produktiv verwendet werden zu können. Elm kann unter Umständen die bessere Lösung sein, man muss sich der Limitierungen von Elm aber klar bewusst sein. TypeScript ist zwar technologisch nicht die schönste Lösung, aber die externen Faktoren wie die weite Verbreitung überzeugen. Für den Start eines neuen Projektes lässt TypeScript am wenigsten Unsicherheiten offen und hat die absehbarste Zukunft.

2 Management Summary

Ausgangslage Web-Applikationen werden immer umfangreicher und komplexer und verlagern viel Logik vom Server in den Browser, welcher die Webseite anzeigt. Um diese Komplexität besser bewältigen zu können, ist eine gute Technologiewahl für die Zukunft der Applikation sehr entscheidend, da ein nachträglicher Wechsel mit grossem Aufwand verbunden ist.

Vorgehen und Technologien In dieser Studienarbeit werden folgende drei Technologie-Kombinationen auf die Frage hin untersucht, ob und in welchen neuen Projekten sie eingesetzt werden sollten.

- TypeScript mit React und Redux
- Reason mit ReasonReact und Reductive
- Elm

Dazu wird mit jeder der drei Varianten eine Beispiel-Applikation entwickelt, welche möglichst viele technologische Herausforderungen realer Use-Cases abdeckt. Anhand der gesammelten Erfahrungen und erhobenen Daten werden die Optionen miteinander verglichen und es wird eine daraus folgende Empfehlung bezüglich Technologiewahl für neue Projekte abgegeben.

Ergebnisse Die Programmiersprache Reason ist zum Zeitpunkt dieser Studie noch nicht ausgereift und sollte deshalb noch nicht für neue Projekte eingesetzt werden. Sofern eine dieser drei Technologien eingesetzt werden soll, ist in den meisten Projekten TypeScript zu wählen. Der Grund liegt dafür in erster Linie in der weiten Verbreitung und der damit verbundenen erhöhten Zukunftssicherheit. Elm kann in Projekten, welche auf eine äusserst hohe Stabilität und Testbarkeit angewiesen sind, eine bessere Alternative zu TypeScript darstellen. Diese Technologie bringt aber eine tiefere Zukunftssicherheit aufgrund der intransparenten Sprachentwicklung mit sich.

Ausblick Aufgrund der schnellen Veränderungen von Web-Technologien müssen solche Vergleiche in absehbarer Zeit wiederholt und revidiert werden. Es ist damit zu rechnen, dass Reason in Zukunft eine deutliche bessere Position einnehmen wird und TypeScript eventuell an einigen Stellen ablösen kann.

3 Einleitung

3.1 Ausgangssituation

Mit der rasanten Vergrösserung des Internets und der Ausweitung seiner Nutzung werden Web-Applikationen immer wichtiger. Um die Nutzerbedürfnisse abzudecken, werden auch die Frontends von Web-Applikationen immer grösser und komplexer. Diese Komplexität kann mittels zweier gängiger Wege bewältigt werden: Frameworks/Libraries oder Sprachen. Bei den Frameworks sind beispielsweise React, Angular und Vue weit verbreitet, welche alle auf JavaScript basieren. Die andere Option ist es, eigenständige Sprachen zu verwenden, welche zu JavaScript (und in Zukunft zu WebAssembly) transpiliert werden. Diese können zusätzliche Möglichkeiten und Sicherheit schaffen, was durch die reine Verwendung von JavaScript-Libraries nicht möglich wäre. So ist es z. B. möglich, ein statisches Typsystem einzuführen, was es so in JavaScript nicht gibt.

Die funktionale Programmierung befindet sich momentan im Aufschwung. Zeichen dafür sind insbesondere die Einführung von funktionalen Ansätzen in etablierten nicht-funktionalen Sprachen wie Java, C#, JavaScript etc. Dies ist unter anderem der Fall aufgrund der Versprechen, welche funktionale Prinzipien machen, wie z. B. bessere Nachvollziehbarkeit, weniger bis keine Laufzeitfehler und mehr Fehlererkennung zur Kompilierzeit.

Die Auftraggeber der Studie glauben daran, dass mithilfe von funktionalen Prinzipien die Komplexität in Web-Frontend-Applikationen besser bewältigt werden kann. Um diese Prinzipien optimal anwenden zu können, wird die Elm-Architektur gewählt. Diese Architektur kann mit verschiedenen Sprachen und Frameworks umgesetzt werden.

Problematik Die Elm-Architektur entstammt der Sprache Elm, jedoch findet sie heute die meiste Verbreitung in der Umsetzung mit React in Kombination mit Redux. Das setzt wiederum die Verwendung von JavaScript bzw. TypeScript voraus. Diese beiden Sprachen unterstützen jedoch funktionale Prinzipien und Typsicherheit nur beschränkt.

Die Frage ist deshalb, in welchen Fällen andere Sprachen, welche die grundlegenden Prinzipien der Elm-Architektur besser unterstützen, als Alternative zu React und Redux mit TypeScript fungieren können. Da die Wahl einer guten Web-Frontend-Entwicklungslösung weitreichende Konsequenzen hat, soll diese Studienarbeit mithilfe eines Experiments Informationen für bessere Entscheidungen in dieser Hinsicht beschaffen.

3.2 Abgrenzung und Recherche

Eine Untersuchung aller möglichen Web-Frontend-Sprachen und -Frameworks ist unmöglich. Es entstehen schneller neue Lösungen, als diese von zwei Studierenden untersucht werden könnten. Deshalb muss die Aufgabenstellung so ausgearbeitet werden, dass sie die Anzahl der untersuchten Lösungen einschränkt und gleichzeitig so aufeinander abstimmt, dass ein Vergleich sinnvoll möglich ist.

Im folgenden Abschnitt werden die Resultate der Recherche für die Aufgabenstellung und die daraus folgenden Begründungen für die Lösungswahl dargelegt.

3.2.1 Elm-Architektur

Um eine funktionale Web-Frontend-Applikation zu erstellen, muss eine entsprechende Architektur gewählt werden, welche diese Prinzipien unterstützt und fördert. Konkret bedeutet dies, dass es sich um eine Architektur handelt, welche die Business-Logik soweit wie möglich mit Pure-Functions modelliert. Dafür sind unidirektionale Architekturen[1] besonders geeignet. Um einen Vergleich zu ermöglichen, werden die untersuchten Lösungen darauf eingeschränkt, dass sie die gleiche Architektur unterstützen.

Gewählt wurde die Elm-Architektur deshalb, weil sie einerseits mit ihrer unidirektionalen Natur den Einsatz von Pure-Functions fördert und andererseits ein Grundstein von Redux ist. Letzteres ist relevant, da diese Studienarbeit durch praktische Erfahrungen mit React und Redux angestossen wurde und diese Technologien sinnvollerweise in den Vergleich eingeschlossen werden.

Im Rahmen dieser Studienarbeit werden deshalb folgende Web-Frontend-Entwicklungslösungen untersucht:

- TypeScript mit React und Redux
- Elm
- Reason mit ReasonReact und Reductive

[1] Beispiele undirektionaler Architekturen: https://staltz.com/unidirectional-user-interface-architectures.html

Funktionsweise Das folgende Diagramm veranschaulicht die wichtigsten Elemente der Elm-Architektur.

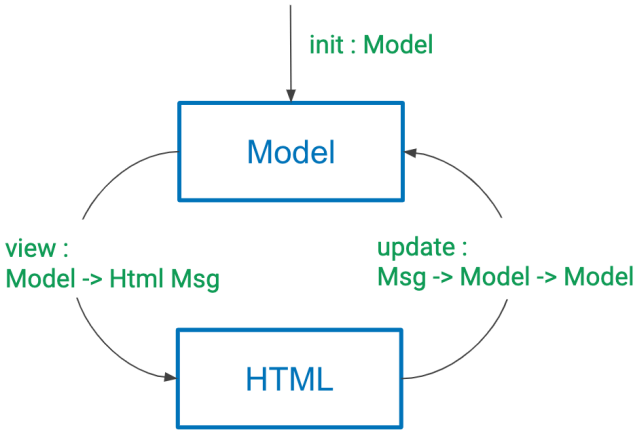

Abbildung 1: Das Modell der Elm-Architektur.

Der wichtigste Aspekt dieser Architektur ist die Kombination aus einem Runtime-System (blau markiert) und Pure-Functions (grün markiert). Der grösste Teil der Applikationslogik wird in den beiden Funktionen *view* und *update* abgebildet. Die *view*-Funktion bildet den aktuellen Applikationsstate (Model) auf eine Darstellung ab und die *update*-Funktion erzeugt aus Nachrichten (z. B. Browser-Events, meist ausgelöst durch den Benutzer) und dem letzten Applikationsstate einen neuen State. Dieser neue State wird dann wiederum mit *view* visualisiert.

Einschätzung Die Evaluation der Elm-Architektur im Vergleich zu anderen Architekturmustern ist kein Teil dieser Arbeit. Dennoch soll an dieser Stelle eine knappe Abschätzung der markantesten Vor- und Nachteile dieser Architektur dargelegt werden.

Vorteile:

- Explizit: Die Quelle der Daten ist immer sichtbar, es kann nichts “automagisch” passieren. Es gibt auch keinen internen State, über den Wissen benötigt wird.
- Nachvollziehbar: State-Änderungen sind nur basierend auf Nachrichten möglich. Der State vor und danach können verglichen und die Unterschiede nachvollzogen werden.
- Transparenz: Interaktionen mit der Aussenwelt (mittels Side-Effects) werden klar markiert. Dies führt dazu, dass mehr Klarheit über die Aufgabe bestehen muss. Beispielsweise muss beim Anzeigen des aktuellen Datums der Zeitpunkt bestimmt werden, wann das Datum ermittelt und wann es aktualisiert werden soll.

Nachteile:

- Kein interner State: Es ist nicht möglich, komplett unabhängige Komponenten zu erstellen, welche ihre gesamte Interaktion mit einem internen State abhandeln. Dies macht die Entwicklung und den Einsatz von Library-Komponenten aufwendiger.

- Verbosität: Alle genannten Vorteile und der obere Nachteile resultieren meist in mehr Daten, welche übergeben werden müssen. Es ist nicht möglich, etwas zu verstecken und somit muss der Entwickler über mehr Informationen verfügen.

3.2.2 React und Redux in TypeScript

React und Redux ist eine der beliebtesten Frontendlösungen.

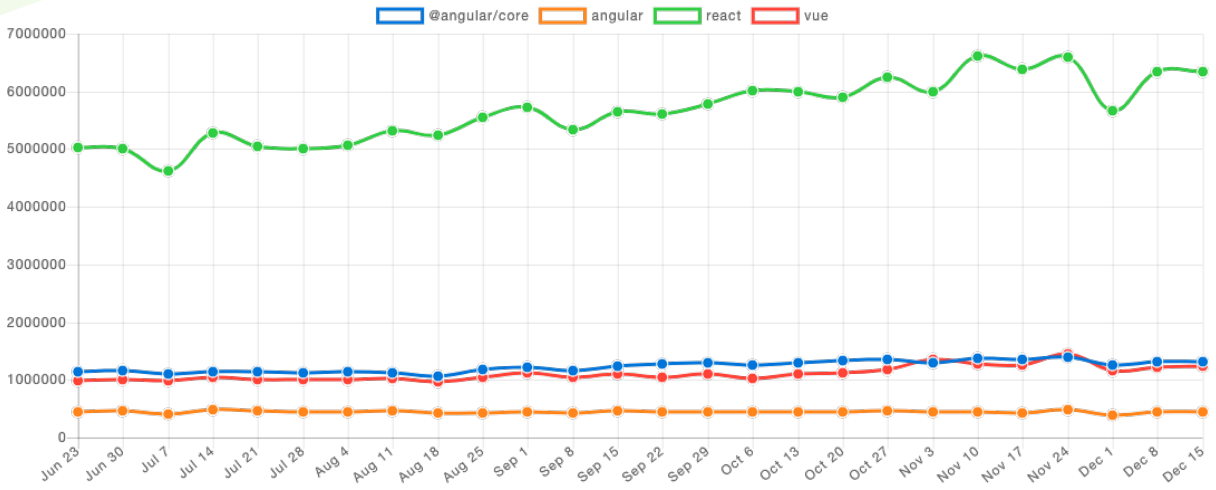

Abbildung 2: Anzahl NPM-Downloads der letzten 6 Monate wichtiger UI-Libraries.

Stand 17. Dezember 2019

Wie oben ersichtlich ist React momentan die UI-Library, welche am meisten von NPM heruntergeladen wird und Redux ist der beliebteste State-Manager, welcher meist in Kombination mit React verwendet wird. Zur Entwicklung wird TypeScript verwendet, welches gewisse Typsicherheit bietet. Da TypeScript ein Superset von JavaScript ist, kann grundsätzlich beliebiger JavaScript-Code innerhalb von TypeScript verwendet werden. Das bedeutet, dass einerseits bestehende Projekte schrittweise zu TypeScript gewandelt werden können, andererseits alle bereits bestehenden JavaScript-Libraries verwendet werden können.

Der Grund für die Wahl dieser Frontendlösung liegt in ihrer Beliebtheit in Kombination mit einer Architektur, welche auf funktionalen Prinzipien aufbaut. Ausserdem gab, wie oben erwähnt, diese Frontend-Lösung den Anstoss zu dieser Arbeit.

3.2.3 Elm

Elm ist eine eigenständige Sprache. Ihre Wurzeln hat sie in den “reineren” funktionalen Sprachen wie Haskell. Besonders interessant an Elm ist, dass die ganze JavaScript- und Browserumgebung abstrahiert wird. Es ist nur über von Elm bestimmte Wege möglich, mit dem Browser und dem Benutzer zu interagieren. Dies erlaubt Elm, striktere Paradigmen und Kontrollen durchzusetzen und mehr Garantien zu bieten.
Von Elm nicht bereitgestellte Interaktionen können nicht implementiert werden. So ist es beispielsweise nicht möglich, eine JavaScript-Library direkt zu verwenden.

Zur Untersuchung gewählt wurde sie aus folgenden Gründen:

- Redux, welches auch untersucht wird, hat Elm bzw. die Elm-Architektur als Inspirationsquelle [1].
- Elm bietet einen starken Kontrast zu JavaScript/TypeScript (Schnittmenge der gemeinsamen Features ist klein).
- Elm abstrahiert die Browserumgebung.
- Elm wird bei realen, grösseren Projekten eingesetzt. Beispielsweise von NoRedInk[2] für ihre Core-Business-Applikation [2].

3.2.4 Reason

Reason (manchmal auch ReasonML) ist eine Sprache, welche massgeblich von Facebook entwickelt wird (wie auch React). Reason basiert auf OCaml bzw. ist ein Dialekt von OCaml und wurde für den Web-Frontendbereich konzipiert. OCaml ist nicht so "rein" wie Elm und erlaubt auch nichtfunktionale Paradigmen.

Die Wahl zur Untersuchung erfolgte aus folgenden Gründen:

- Reason ist von der Striktheit/Sicherheit zwischen TypeScript und Elm angesiedelt und bietet somit eine weniger radikale Alternative zu Elm.
- Reason und ReasonReact werden von Facebook, den Entwicklern von React, entwickelt.
- Reason wird bei realen, grösseren Projekten eingesetzt. Beispielsweise setzt Facebook Reason beim Facebook Messenger ein [3].

3.2.5 Nicht untersuchte Frontendlösung

Es gibt natürlich weitere Alternativen zu den drei gewählten Frontendlösungen wie beispielsweise PureScript. Aus Gründen der begrenzten Ressourcen innerhalb einer Studienarbeit können diese nicht alle untersucht werden. Die nicht untersuchten Alternativen bieten eine weniger natürliche und gute Unterstützung der Elm-Architektur.

Im Folgenden einige Kategorien (nicht abschliessend), welche nicht untersucht werden.

Transpiler zu JavaScript von bestehenden Sprachen Es existieren Transpiler zu JavaScript von einer Vielzahl von etablierten Programmiersprachen. Die Interaktion mit JavaScript ist jedoch meistens kein Hauptaugenmerk dieser Sprachen, die Umwandlung zu JavaScript wird oft nur im Rahmen einer Machbarkeitsstudie oder eines kleineren Projekts entwickelt. Oft fehlt es an guter Dokumentation und einer Community.

Reason fällt aufgrund seiner OCaml-Wurzeln zu einem gewissen Teil in diese Kategorie, hat aber durch seinen Fokus auf JavaScript-Entwickler und seine React-Unterstützung den klaren Anspruch, in der Web-Frontend-Entwicklung verwendet zu werden. Mit Facebook im Rücken hat Reason das Potenzial, zu einer etablierten

[2] https://www.noredink.com/

Lösung zu werden.

Ein weiteres Beispiel dieser Kategorie ist Haste. Haste wird nicht als praktikable Lösung angesehen, da es nur wie eine Machbarkeitsstudie erscheint. Indizien dafür sind beispielsweise:

- Auf der offiziellen Seite erscheinen Fehler (die Demo kann sich nicht mit dem Server verbinden)
- Letzte Änderung am Compiler auf GitHub war vor Monaten
- Dokumentation ist spärlich

Frameworks / Libraries für JavaScript Von möglichen Frameworks und Libraries zu JavaScript wimmelt es geradezu. Bei den Frameworks besteht oft das Problem, dass diese eine Architektur vorgeben, welche nicht einfach mit der gewählten Architektur verglichen werden kann. Von den Libraries ist React die momentan etablierteste und meistverwendete und wird deshalb als Stellvertreter dieser Kategorie verwendet.

WebAssembly Eine neue Möglichkeit bietet WebAssembly. Die Interaktion mit WebAssembly ist momentan noch nicht ganz ausgereift und wird deshalb nicht genauer betrachtet.

3.2.6 Library-Versionen

Technologien entwickeln sich schnell weiter; damit die Aussagen dieses Berichts nachvollziehbar sind, werden hier die verwendeten Versionen der wichtigsten Libraries und Sprachen dokumentiert. *NPM-Name* referenziert den Namen, welcher das Package auf NPM verwendet.

Tabelle 1: Versionen der wichtigsten Libraries

Name	NPM-Name	Version
TypeScript	*typescript*	3.6.4
Elm	*elm*	0.19.0
Reason	*bs-platform*	5.2.1
ReasonReact	*reason-react*	0.7.0
Reductive	*reductive*	2.0.1
Redux	*redux*	4.0.4
React	*react*	16.9.0
React Scripts	*react-scripts*	3.2.0
Elm Test	*elm-test*	0.19.0
Jest	*jest*	24.9.0
Terser	*terser*	4.3.1
Bisect_ppx	*bisect_ppx*	2.0.0-dev.0

3.3 Untersuchung

Ziel Das Ziel ist, Hilfestellung zu folgender Frage zu bieten: *In welcher Situation, in welchem Projekt sollte man eine dieser drei Varianten verwenden? Welche Option ist für welche Art Projekt am geeignetsten und wo sind andere Varianten angebracht?* Zur Hilfestellung werden dieser Bericht sowie ein Handout erstellt.

Vorgehen Zur Evaluation wird in den drei untersuchten Frontendlösungen je ein HTTP-Client entwickelt.

Auf der Basis dieser drei Applikationen und der gesammelten Erfahrung bei der Entwicklung können die drei Varianten miteinander verglichen werden.

Ein HTTP-Client als Entwicklungsstück wird repräsentativ für reale Entwicklungsaufgaben aus folgenden Gründen gewählt:

- Kommunikation mit einem REST-Backend
- Umfang ist nicht gering, kann aber im zeitlichen Rahmen einer Studienarbeit erfüllt werden
- Darstellung verschiedener Elemente
- Single Page App
- Entwickler sind die Zielgruppe der Anwendung, somit wird kein externer Nutzer benötigt

Zielpublikum Das Zielpublikum sind Entwickler und Entscheidungsträger mit einem technischen Hintergrund.

4 Untersuchungsreport

In diesem Abschnitt werden jeweils Erkenntnisgewinne aus Entwicklungsschritten dokumentiert. Der Report orientiert sich bei der Aufzählung an der jeweiligen Reihenfolge, in der entwickelt wurde. In einem Entwicklungsschritt können viele Konstrukte (Programmierlogik, Designelemente) von einer Sprache in eine andere Sprache übernommen werden. Daher wird die Reihenfolge immer wieder verändert, damit sich keine Vorteile für eine Variante bilden.

Die folgende Abbildung gibt einen Eindruck, wie die entwickelte Beispiel-Applikation aussieht:

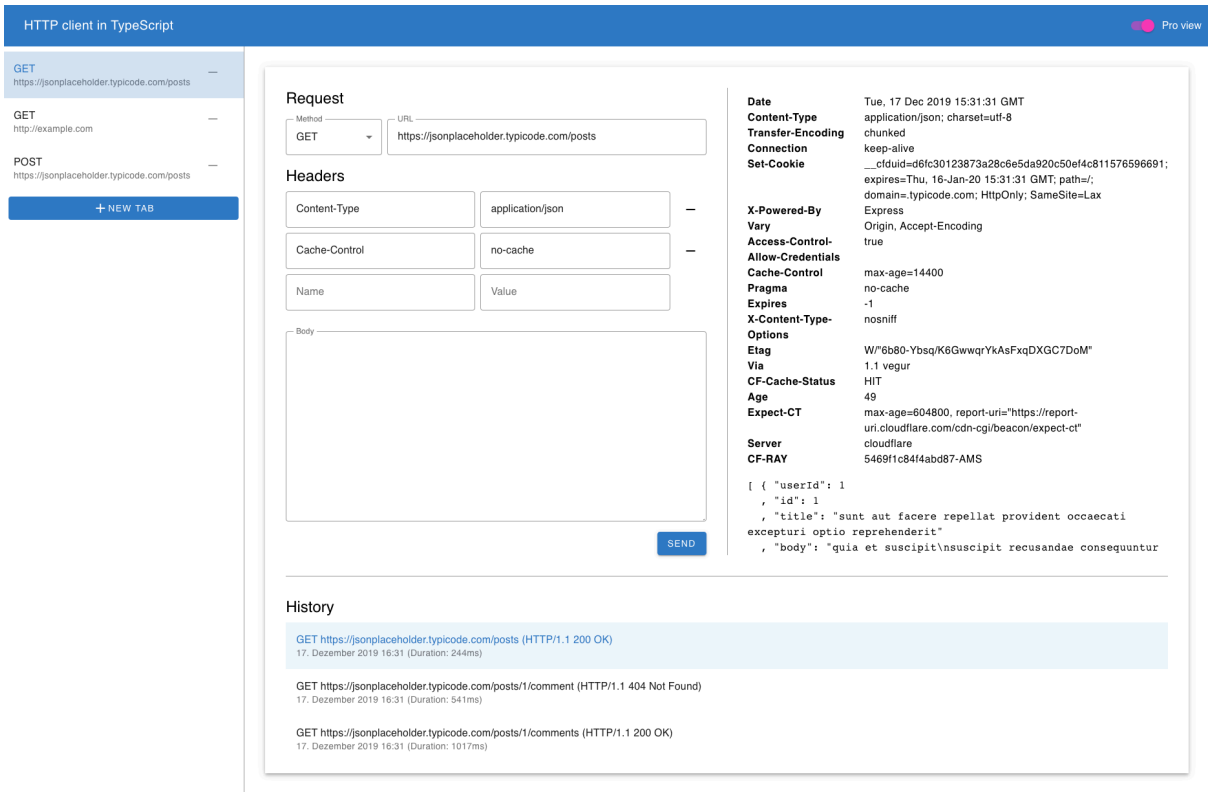

Abbildung 3: Die Beispiel-Applikation: Ein einfacher HTTP-Client.

4.1 Projektsetup und CI/CD

4.1.1 Gegenstand der Entwicklung

In diesem Abschnitt wird das initiale Aufsetzen des Projektes und die Integration in die CI/CD-Umgebung beschrieben.

4.1.2 Entwicklung Elm

Projektsetup Elm bietet nur ein sehr minimales Projektsetup, welches per Kommandozeile generiert werden kann. Um ein verbessertes Projektsetup zu erhalten, muss dieses entweder selber entwickelt oder von einer bestehenden Codebasis adaptiert werden. Insbesondere sind keine Tools für das Erstellen von Produktionsbuilds enthalten. In diesem Bereich gibt es einige offizielle Dokumentationen, welche aber etwas verstreut aufzufinden sind. Eine Integration eines Stores oder einer Library für das Handling von Side-Effects ist dafür nicht nötig, da die Sprache und die damit vorgegebene Architektur bereits gute Lösungen dafür beinhaltet.
Für das Testing, Linting und Code-Coverage müssen entsprechende Libraries hinzufügt werden. Da es in der Elm-Community dafür De-facto-Standards (oder zumindest keine realen Alternativen) gibt, fällt die Wahl auch nicht schwer und die Integration funktioniert auch entsprechend gut.

CI/CD Der CI/CD-Prozess ist bei Elm relativ einfach, da alle nötigen Abhängigkeiten als Node-Module verfügbar sind. Da nicht direkt ein optimierter Build verfügbar ist, muss dieser selber eingerichtet werden. Das Deployment ist gleich wie bei anderen statischen Dateien, welche ausgeliefert werden müssen, und hat keine Elm-spezifische Eigenheiten.

4.1.3 Entwicklung Reason

Projektsetup Das Projektsetup von Reason mit ReasonReact ist von BuckleScript bestimmt. BuckleScript ermöglicht die Erstellung eines Projekttemplates mit ReasonReact ohne zusätzliche Tools. Darin enthalten sind sowohl die nötigen Abhängigkeiten für ReasonReact als auch ein Setup für Entwicklungs- und Produktionsbuilds.
Reductive (die Reason-Alternative zu Redux) muss selber integriert werden. Die Dokumentation von Reductive ist eher schmal, reicht aber gut aus, um die Library in der Applikation zu integrieren. Für die Verarbeitung von Side-Effects gibt es keine etablierte eigenständige Library, allerdings hat Reductive einen Vorschlag für eine selber implementierbare Middleware, welche auf dem Konzept von Redux-Thunk basiert.
Für das Testing werden meist direkt JavaScript-Testing-Libraries wie Jest verwendet. Das bedeutet, dass der Reason-Code zuerst transpiliert und danach als JavaScript-Code getestet wird. Mit entsprechenden Reason-Bindings kann der Test-Code auch in Reason geschrieben werden und ist damit eine sehr gute Variante.
Linter sind praktisch nicht auffindbar für Reason. Der Compiler kann jedoch so konfiguriert werden, dass viele Dinge, welche in anderen Sprachen via Linter überprüft werden müssen, direkt vom Compiler geprüft werden.

Die Lage bei der Code-Coverage ist schwierig. Grundsätzlich wäre es natürlich möglich, hier mit Jest die Code-Coverage zu berechnen. Das Problem dabei ist, dass damit die Coverage des transpilierten Codes und nicht des originalen gemessen würde. Somit könnte damit nur eine ungefähre Grössenordnung der Testabdeckung

angegeben werden. Ein Mapping auf konkrete Funktionen und Zeilennummern wäre dagegen schwierig (bzw. müsste selber implementiert werden). Mit der Library Bisect_ppx kann die Code-Coverage des Reason-Codes zwar berechnet werden, dafür ist deren Setup sehr aufwendig. Dass diese Library nicht in einem Test-Runner integriert ist, erschwert die Verwendung zusätzlich. Somit müssen Tests z. B. mithilfe eines Zusatzfiles ausgeführt und so die Coverage berechnet werden. Direkt mit Jest ist das leider nicht möglich, da Jest die Tests in einem isolierten Kontext ausführt, aus dem Bisect_ppx nicht auf die Coverage-Daten zugreifen kann.

CI/CD Das Ökosystem von BuckleScript basiert auf Node-Modulen, welche entsprechend einfach in einer Pipeline verwendet werden können. Da direkt ein optimierter Build verfügbar ist, muss an dieser Stelle nichts getan werden. Das Deployment der statischen Dateien ist gleich wie bei Elm.
Die grösste Herausforderung für die Continous Integration von Reason ist das Berechnen der Code-Coverage. Bisect_ppx basiert auf auf esy[3], welches einen OCaml-Compilers benötigt. Die Erstellung des OCaml-Compiler dauert ungefähr fünf Minuten, daher sollte dieser Schritt in einem Docker-Image bereits im Vorfeld ausgeführt werden.

4.1.4 Entwicklung TypeScript

Projektsetup Das Projektsetup mit TypeScript, React und Redux ist sehr einfach, da Facebook und die React-Community mit Create React App ein gut gewartetes Tool zur Verfügung stellen, welches eine sehr gute Standard-Toolchain für das Entwickeln von React-Applikationen anbietet. Somit genügt für die Initialisierung der React-App mit TypeScript ein einziger Kommandozeilenaufruf. Dieses Setup beinhaltet auch bereits das Testingframework Jest (inkl. Linting und Code-Coverage) und die Möglichkeit, einen optimierten Produktionsbuild zu erstellen.
Redux muss danach selber integriert werden. Bei diesem Schritt stellt sich die Frage nach einer sinnvollen Ordnerstruktur, welche selber definiert werden muss (und natürlich anhand von Community-Vorschlägen adaptiert werden kann). Für die Integration von Redux in React gibt es sehr gut offizielle Anleitungen und unzählige Tutorials.
Um Side-Effects isoliert abarbeiten zu können, kann eine Library wie Redux-Saga oder Redux-Thunk[4] verwendet werden. Dies ist wie Redux selber nicht im initialen Projekttemplate enthalten und muss deshalb separat hinzugefügt werden. Auch hier existieren aber gute Anleitung, welche von den Libraries selbst zur Verfügung gestellt werden.

CI/CD Wie bei den anderen beiden Varianten basiert auch diese komplett auf Node-Modulen. Der CI-Prozess ist entsprechend einfach und *Create React App* hat sogar einige CI-spezifisch optimierte Funktionen (z. B. laufen die Tests automatisch nicht im Watch-Modus wie in der Entwicklungsumgebung). Das Deployment ist wie bei den anderen Varianten nicht spezifisch.

[3] https://esy.sh/
[4] https://github.com/reduxjs/redux-thunk

4.2 Backend ansprechen / Handhabung von Side-Effects

4.2.1 Gegenstand der Entwicklung

Folgende Punkte gab es zu erfüllen:

- Typen für HTTP-Anfrage und -Antwort zum Backend definieren
- HTTP-Anfragen an das Backend senden
- HTTP-Antworten des Backends parsen und temporär speichern
- Tests für diese Side-Effects schreiben

Mit *HTTP-Anfragen* und *HTTP-Antworten* sind die Entitäten wie in Swagger dokumentiert gemeint. Für die Umsetzung dieses Entwicklungsabschnittes musste definiert werden, wie mit Side-Effects umgegangen wird.

Remo Dörig führte diesen Entwicklungsschritt durch.

4.2.2 Entwicklung Elm

Funktionsweise In Elm kann eine HTTP-Anfrage ausgeführt werden, in dem der Elm-Runtime ein Befehl (*Cmd*) zurückgegeben wird, welcher die nötigen Daten enthält, um die Anfrage auszuführen. Beim Befehl wird der entsprechende Decoder für JSON mitgegeben.

Wie die Anfrage ausgeführt wird, entscheidet die Elm-Runtime. Für den Entwickler ist dies eine Black-Box.

Ein anderer Weg ist nicht möglich. Dies führt dazu, dass eine Entscheidung erspart bleibt.

Erstellung der Entitäten Die Entitäten von Swagger lassen sich einfach als Typen definieren. Das Encoden und Decoden für JSON muss selber definiert werden. In anderen Sprachen wie Haskell können Encoder und Decoder mit guten Defaultwerten automatisch generiert werden.

Strukturierung Die Strukturierung der Funktionalitäten sind von Elm klar definiert. Die Filestruktur ist jedoch nicht so klar definiert und kann frei gewählt werden.

Testing Da alle Funktionen keine Nebeneffekte haben, lassen sich einfach Tests schreiben. Der JSON-Decoder wird an Elm übergeben, wie Elm diesen dann genau anwendet ist nicht ganz klar, daher ist es auch nicht ganz klar, wie dieser getestet werden sollte.

Probleme bei der Entwicklung Das Elm-Tooling funktionierte auf Windows nicht auf Anhieb. Die Fehlermeldungen sind hilfreich, die meisten Fehler können schon bei *TYPE MISMATCH* entdeckt werden. Es braucht dafür je nach dem schon etwas Zeit, bis der Code kompilierbar ist mit den richtigen Typen. Am Ende ist der Code klein und überschaubar.

Die Dokumentation könnte besser sein. Oft findet man auf rudimentäre Fragen noch keine Antwort (beispielsweise auf Stackoverflow).

Debuggen funktioniert manchmal nicht so gut. Einerseits sind die Debug-Möglichkeiten für Personen aus einem imperativen Hintergrund nicht intuitiv. Andererseits sieht man in die internen Mechanismen von Elm nicht hinein. Es gab das Problem, dass die HTTP-Anfrage über Elm zuerst nicht funktionierte. Via nativem JavaScript mit *fetch* funktionierten die Aufrufe, aber die HTTP-Anfrage, welche von Elm gestellt wurde, veranlasste den Browser dazu, zuerst eine *OPTIONS*-Anfrage an den Server zu senden, welche mit einem Status-Code von *400* beantwortet wurde.

4.2.3 Entwicklung TypeScript

Funktionsweise In TypeScript wurde die HTTP-Anfrage über Redux-Saga durchgeführt. Eine Saga aktiviert sich, sobald ein bestimmter Befehl an den Redux-Store gesendet wird. Eine Saga ist eine Generator-Funktion und spricht über das Generator-Interface mit der Saga-Middleware, um asynchrone Szenarien abzuwickeln. Eine Saga kann dann wiederum einen Befehl an den Redux-Store senden.

Der eigentliche Aufruf wurde mit der von JavaScript gelieferten Funktion *fetch* ausgeführt. Das Encoden und Decoden funktioniert auch über JavaScript-Funktionen, welche keine Typsicherheit bieten.

Erstellung der Entitäten Die Entitäten konnten sehr unkompliziert definiert werden, da TypeScript sehr nahe an JavaScript ist. Die Default-Attribute sind für unseren Zweck nicht optimal gewählt, da jedes Property als *readonly* deklariert werden muss. Problematisch war ein Union-Typ, welcher aus zwei Subtypen besteht. Die Unterscheidung, welcher Typ der konkrete Wert hat, kann zur Laufzeit nicht über das Typsystem von TypeScript ermittelt werden, da zu diesem Zeitpunkt keinerlei Typinformationen vorhanden sind. Entweder muss ein zusätzliches Property (beispielsweise namens *type*) hinzugefügt werden, in welchem diese Informationen stehen, oder es wird anhand der anderen Properties des Objekts entschieden.

Strukturierung Die Strukturierung der Funktionalitäten ist durch die angewendeten Libraries gut definiert. Für die Filestrukturierung gibt es etablierte Standards. Generell ist der Entwickler frei in der Strukturierung, es gibt jedoch etablierte Best Practices, was Entscheidungen vereinfacht.

Testing Die funktionalen Komponenten können gut getestet werden. Das Testing der Sagas ist nicht ganz so simpel, aber es gibt Libraries, welche dies vereinfachen und ermöglichen.

Probleme bei der Entwicklung Es sind viele Libraries für ein kleines Problem. Manche kleine Fehler könnten mit einem strikteren Typsystem besser entdeckt werden. Die Funktionalität wird in vergleichsweise viele Einzelschritte aufgeteilt.

Für Redux gibt es einen guten Debugger als Browser-Extension, was das Debuggen sehr vereinfacht.

4.2.4 Entwicklung Reason

Vorwort Bei Reason gibt es keinen definierten Weg oder etablierte Best Practices für vieles, da die Sprache relativ jung ist. Für die Handhabung der Side-Effects benutzt das Beispiel von Reductive Thunks. Thunks sind Funktionen, welche Befehle an den Store übergeben können. Dies ermöglicht die Handhabung von Side-Effects.

Bei der Entwicklung musste zuerst eruiert werden, wie die Side-Effects programmiert werden. Nachträglich wurde dies von Thunks in eine selber geschriebene Middleware geändert.

Funktionsweise Für Reason konnte keine Library ähnlich wie Redux-Saga gefunden werden, daher wurde eine Middleware entwickelt. Diese löst Funktionen bei bestimmten Befehlen aus, ähnlich wie bei TypeScript, jedoch nicht über Generatoren.

Die Anfragen werden über eine von Reason gegebene *fetch*-Implementation gemacht, welche auf *Promises* basiert. Für das Encoden und Decoden mussten Funktionen ähnlich wie in Elm geschrieben werden.

Erstellung der Entitäten Entitäten können nicht direkt verschachtelt definiert werden. Jede Ebene muss einzeln definiert werden, was den Code aufbläst und es müssen Namen für jede Ebene definiert werden. Das Encoden und Decoden für JSON muss selber definiert werden. In anderen Sprachen wie Haskell können Encoder und Decoder mit guten Defaultwerten automatisch generiert werden.

Strukturierung Für die Strukturierung gibt es wenige offizielle Informationen und die Entwickler müssen selber entscheiden. Im Beispiel von Reductive wird eine hierarchische Unterteilung der Funktionen verwendet. Dies führt dazu, dass jeweils nur ein Teil des Stores die Befehle erhält.

Dies erwies sich als impraktikabel, weshalb alle möglichen Aktionen auf die gleiche Hierarchiestufe gehoben wurden. Dies führt aber wiederum dazu, dass das Typsystem nicht mehr überprüfen kann, ob alle Aktionen behandelt werden, da jeder Reducer selber entscheiden kann, welche dieser Aktionen er behandeln möchte.

Testing Die funktionalen Teile sind gut testbar. Die Middleware und der HTTP-Aufruf sind nur mit intensivem Mocking testbar.

Probleme bei der Entwicklung Es ist bemerkbar, dass Reason noch in den Kinderschuhen steht. Teilweise bricht der Compiler ab, ohne eine Fehlermeldung zu geben. Syntax-Fehler ergeben jeweils nur *Syntax Error* ohne eine Beschreibung, was falsch ist. Einige Fehlermeldungen sind dennoch hilfreich und der Compiler ist sehr schnell.

Bei der Entwicklung konnten nicht auf etablierte Standards oder Best Practices zurückgegriffen werden. Die Community ist noch klein, die Dokumentation ist vorhanden, könnte aber ausführlicher sein.

Die Sprache erlaubt keine zyklischen Referenzen. Man kann argumentieren, dass dies zu schönerem Code führt. Falls zyklische Referenzen auftreten, stirbt der Compiler teilweise ohne eine Fehlermeldung.

Wie mit JSON in Reason umgegangen wird, ist noch nicht definiert, was für eine Webprogrammiersprache ein wichtiger Bestandteil ist.

Es kam vor, dass der Compiler nicht valides JavaScript generierte. Dies geschah aber nur aufgrund von alten Daten im Cache und das Problem tritt auch nur unter Windows auf.

4.3 Body und Headers bearbeitbar machen

4.3.1 Gegenstand der Entwicklung

In diesem Entwicklungsschritt wurden der Body und die Headers der Anfrage bearbeitbar gemacht. Header können nun hinzugefügt, entfernt und bearbeitet werden. Beim Öffnen der Seite / eines Tabs sind bereits einige häufig verwendete Headers vorhanden.

Remo Dörig führte diesen Entwicklungsschritt durch.

Der damit entwickelte Teil sieht wie folgt aus:

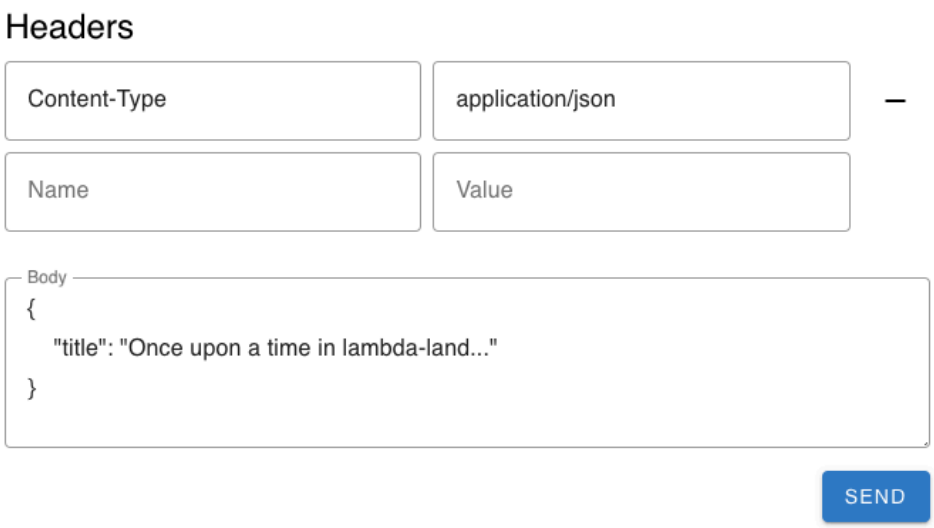

Abbildung 4: Die Steuerelemente für die Headers und den Body.

4.3.2 Allgemein

Die Entwicklung war alle drei mal sehr ähnlich und bot die gleichen Schwierigkeiten. In diesem Abschnitt werden die Sachen vorgestellt, welche in allen Versionen gleich waren.

Das Vorgehen, um den zu bearbeitenden Body zu implementieren, war relativ klar vorgegeben und es kam zu keinen Problemen der Programmierlogik.

In einem funktionalen Stil fühlt es sich falsch an, wenn man Indexe einer Liste verwendet, noch mehr als in anderen Sprachen. Für die klarere Bearbeitung der Headers und im Ausblick darauf, dass diese möglicherweise umsortiert werden könnten, wurde den Header-Elementen für die Ansicht jeweils eine ID vergeben. Bei der Generierung der ID stellte sich nun die Frage, was für IDs wie erstellt werden. IDs gleich dem Index setzen würde bei der Verschiebung der Elemente zu Verwirrung oder sogar Fehlern führen. Zuerst wurde es mit UUIDs implementiert. Die UUIDs wurden im Reducer generiert und hinzugefügt, im Reducer dürfen jedoch keine Side-Effects bestehen. Dass die UUIDs eigentlich Side-Effects sind wurde erst bei der Entwicklung von Elm klar. Die UUIDs wurden deshalb von aussen (Containers, main.js bei Elm) initialisiert, dies erschwert unnötigerweise die Verwendung der Aktionen/Befehle, welche UUIDs benötigen. Daher wurden die UUIDs durch deterministisch fortlaufend generierte IDs mit einem String-Präfix ersetzt.

4.3.3 Entwicklung Reason

In Reason war die Implementierung relativ klar. Probleme verursachten vor allem die Unerfahrenheit des Entwicklers (Remo Dörig) mit der Architektur und dem Compiler. Der Compiler ist nicht sehr informativ, wenn es darum geht, was mit dem Programm nicht korrekt ist. Durch die starke Typsicherheit ist es schwer, Arbeitsvorgänge nur teils zu programmieren, um diese Bruchstücke bereits im Betrieb analysieren zu können. Dies erschwert den Zugang für Personen, welche in der Umgebung noch nicht so erfahren sind.

4.3.4 Entwicklung Elm

Die Fehlermeldungen bei Elm sind bereits wesentlich hilfreicher als die von Reason. Refactoring des Codes dauert zwar wegen der suboptimalen IDE-Unterstützung etwas länger wie bei etablierten Sprachen, nach dem Refactoring ist aber durch das strikte Typsystem eine hohe Sicherheit in der Funktionsweise vorhanden. Wie in Reason ist es schwer, Arbeitsvorgänge nur teils fertig zu entwickeln und dann zu analysieren.

Die Unterteilung der Funktionsweisen ist bei den anderen beiden Sprachen durch De-facto-Standards klarer definiert.

4.3.5 Entwicklung TypeScript

Funktionsweise konnte fast eins zu eins von Reason übernommen werden. Auf Windows wurden File-Änderungen nicht immer korrekt erkannt. Bei der Entwicklung wurde aus Versehen Node aktualisiert, was zu Inkompatibilitäten mit dem Coverage-Runner für die Tests führte. Diese Inkompatibilitäten waren nicht klar ersichtlich.

4.4 Tabs

4.4.1 Gegenstand der Entwicklung

Es mussten neue Tabs hinzugefügt und entfernt werden können. Bei der Übersicht der Tabs musste ersichtlich sein, um welche Tabs es sich handelt, sprich ein Tab musste identifizierbar sein. Zu einem Tab gehören die Anfrage, die Antwort und die History. Durch diesen Entwicklungsabschnitt mussten diese Elemente und deren Handhabung neu in einen Tab verschachtelt werden.

Remo Dörig führte diesen Entwicklungsschritt primär durch. Joel Fisch veränderte insbesondere die Verschachtelung der Aktionen in TypeScript und Reason.

Die Tabs werden in der Beispiel-Applikation auf der linken Seite folgendermassen dargestellt.

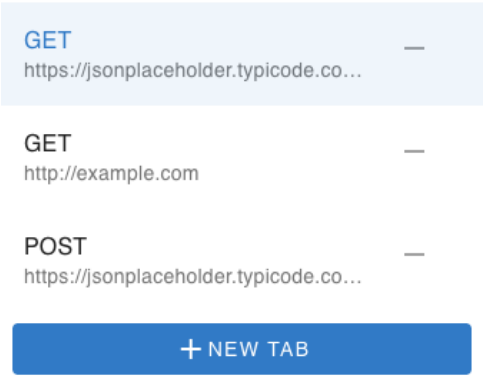

Abbildung 5: Die Tabs des HTTP-Clients.

4.4.2 Entwicklung Elm

Umstrukturierung für die Tabs Vor diesem Entwicklungsabschnitt war der Hauptteil von *update*, *view* und den Models in *Main.elm*. Diese Teile mussten in die Tabs verschachtelt werden, dies wurde gerade genutzt, um die einzelnen Bereiche in verschiedene Files zu verschieben. Mithilfe der *Html.map* Funktion können Befehlsgeneratoren verschachtelt werden. Somit konnte der bestehende *view*-Code immer noch die gleichen Befehle generieren, ohne dass dem Code überhaupt bewusst ist, dass er in einem Tab ist. Bei der Verarbeitung der Befehle in *update* konnten durch algebraische Datentypen und Pattern-Matching die verschachtelten Befehle einfach verarbeitet werden. Den Grossteil des bestehenden *update*-Codes konnte unverändert wiederverwendet werden.

Bearbeitung der Tabs Die eigentliche Bearbeitung der Tabs (Hinzufügen, Entfernen) funktionierte fast gleich wie bei den Headern des Requests.

Problematiken bei der Entwicklung Das Refactoring konnte schlecht in einzelne Teile aufgeteilt werden. Es mussten viele Bereiche gleichzeitig angepasst werden, damit das Programm wieder lief. Wie die Verschachte-

lung am besten strukturiert wird, war nicht so klar. Von der Community her gibt es keine wirklichen De-facto-Standards, an die sich die meisten halten würden. Bei der Funktion *Html.map*, welche für diese Umstrukturierung notwendig war, stand in der Dokumentation, man sollte diese sehr wahrscheinlich nicht benutzen müssen. Für die Begründung ist ein Link angegeben, welcher nicht mehr gültig ist. Bei den halboffiziellen Beispielen wird diese Funktion aber auch verwendet.

4.4.3 Vorwort zu TypeScript und Reason

TypeScript und Reason besitzen keine Funktionalität wie *Html.map* von Elm und haben in unserer Ausführung eine flache Aktionsstruktur. Anfangs machten wir den Fehler, dass wir die Aktionen für die Inhalte der Tabs jeweils auf den aktiven Tab anwendeten. Dies funktioniert meist, jedoch nicht bei Aktionen, welche durch Side-Effects ausgelöst werden (insbesondere asynchron, wie z. B. das Empfangen der Response). Zwischen Anfrage an den Server und Antwort des Servers konnte der Tab gewechselt werden und so wurde die Antwort im falschen Tab abgespeichert.

4.4.4 Entwicklung TypeScript

Umstrukturierung für die Tabs Da fast alles im Store mithilfe von Helperfunktionen von Libraries geregelt wird, ist die grundlegende Struktur nicht offensichtlich. Die Verschachtelung und Weiterleitungen der Aktionen an die "Subreducers" mussten manuell durchgeführt werden. Eigentlich ist dies ganz simpel, jedoch ergibt dies eine Irregularität gegenüber allen anderen Reducern, welche verwendet werden. Ob dieser manuelle Code mit den Helperfunktionen der Libraries funktioniert, musste zuerst abgeklärt und getestet werden.

Wie im Vorwort beschrieben konnten die Aktionen nicht einfach verschachtelt werden (möglich wäre es, es gäbe aber keine Sicherheit, dass nur korrekt verschachtelte Aktionen verwendet werden). Jede Aktion wurde um ein Feld für die Identifizierung des Tabs erweitert. Dies hat zur Folge, dass jeder Container, welcher Aktionen für die Tabinhalte generierte, angepasst werden musste. Diese Anpassungen sind zwar simpel, brauchen aber viel Code und vor allem viel redundanten Code.

Der Code konnte gut stückchenweise umstrukturiert werden und der Code kompilierte immer noch.

Bearbeitung der Tabs Die eigentliche Bearbeitung der Tabs (Hinzufügen, Entfernen) funktionierte fast gleich wie bei den Headern des Requests.

Problematiken bei der Entwicklung Für die Umstrukturierung musste fast der ganze Code des Stores angepasst werden. Wenn man bedenkt, dass die Tabs eventuell noch in eine Collection kommen könnten, welche selber nochmals verschachtelt wären, dann skaliert dieser Ansatz nicht so gut. Wenn man diese Verschachtelungen von Anfang an kennt, dann ist es weniger ein Problem.

Beim Kompilieren werden die Tests auch überprüft, was zur Folge hat, dass man die Tests frühzeitig an einen Entwicklungsstand anpassen muss, welcher noch nicht definitiv ist. Im Vergleich zu Reason und Elm gibt es des Öfteren Runtime-Fehler.

4.4.5 Entwicklung Reason

Umstrukturierung für die Tabs Die Umstrukturierung war ähnlich wie bei TypeScript, hat aber ein paar wesentliche Unterschiede. Die Ordnerstruktur spielt bei Reason keine Rolle, es kommt also nicht darauf an, ob alle Files im gleichen Ordner sind oder in einer tief verschachtelten Struktur abgelegt werden. Der Filename muss deshalb aber auch eindeutig sein. Dadurch lohnt es sich nicht, die Files in einen Unterordner zu verschachteln, so konnten einige Merge-Probleme präemptiv verhindert werden.

Im Gegensatz zu TypeScript lassen sich mit algebraischen Datentypen die Aktionen gut und natürlich verschachteln. Im Vergleich zu Elm müssen jedoch trotzdem alle Container, welche Aktionen dispatchen angepasst werden, jedoch mit weniger Aufwand als bei TypeScript.

Bearbeitung der Tabs Die eigentliche Bearbeitung der Tabs (Hinzufügen, Entfernen) funktionierte fast gleich wie bei den Headern des Requests.

Problematiken bei der Entwicklung Wie bei Elm muss bei Reason fast die ganze Umstrukturierung auf einmal geschehen, damit der Code wieder kompiliert wird. Die schlechten Fehlermeldungen des Compilers helfen hier sehr wenig.

Durch die momentan kleine Community gibt es noch wenig Erfahrungen, wie eine solche Verschachtelung gut gelöst werden kann.

Die Selektoren sind ein wenig anders aufgebaut als in TypeScript. Sie sind über parametrisierte Module gelöst, wodurch Typen explizit übergeben werden können. Dies ist gewöhnungsbedürftig und erforderte ein wenig Einarbeitung.

4.5 Persistenz

4.5.1 Gegenstand der Entwicklung

Im Entwicklungsschritt zur Persistenz wurde dafür gesorgt, dass Änderungen nach einem Page-Reload und dem Ende einer Browser-Session dem Benutzer noch zur Verfügung stehen. Dies betrifft sowohl ausgeführte Requests, deren Antworten als auch erstellte Tabs und Änderungen am aktuellen Request.

Dieser Entwicklungsschritt wurde von Joel Fisch durchgeführt.

4.5.2 Allgemein

Diese Aufgabe wurde durch die Elm-Architektur sehr vereinfacht. Da das Rendering der View immer auf einem serialisierbaren State basiert und Aktionen nur diesen State verändern, kann der aktuelle Zustand der Applikation relativ einfach abgelegt und beim Laden der Applikation wiederverwendet werden. Dies wäre in den meisten Architekturen nur mit sehr grossem Aufwand möglich.

Bereits zu Beginn wurde aufgrund der Anforderungen festgelegt, dass die Daten in der LocalStorage des Browsers abgelegt werden sollen. Das bedeutet, dass die Daten als String serialisiert gespeichert werden müssen.

4.5.3 Elm

In Elm gibt es zwei offensichtliche Möglichkeiten, um dieses Feature zu implementieren. Die erste Variante ist das Ablegen der Daten nach jeder Aktion mittels eines Elm-Moduls, welches das Speichern von Daten in der *LocalStorage* ermöglicht. Beim Starten der Applikation werden jeweils als erste Aktion die Daten wieder geladen und deserialisiert. Die zweite Variante ist die, dass der initiale State als Flag[5] der Applikation übergeben wird und nach jeder Aktion die Daten mit einem Port[6] an das aufrufende JavaScript zum Speichern übermittelt werden.

Für die Implementation wurde die zweite Variante gewählt, da so eine Abhängigkeit zu einem weiteren externen Modul vermieden wird und gleichzeitig erste Schritte in Bezug auf Interoperabilität mit JavaScript getätigt werden konnten.

Das Implementieren war relativ einfach und entsprach weitestgehend den Vorstellungen vor Beginn. Mit den JSON-Encodern und -Decodern, welche auch für HTTP-Requests und -Responses verwendet werden, war das Serialisieren und Deserialisieren verhältnismässig schnell implementiert. Wünschenswert wäre natürlich, dass Elm selber eine Möglichkeit anbieten würde, um beliebige Daten ohne weitere Definitionen zu serialisieren. Dies ist wohl insbesondere wegen den algebraischen Datentypen, welche keine direkte Abbildung in JavaScript-Objekten haben, bis jetzt nicht möglich. Das bedeutet natürlich, dass bei weiteren Änderungen im State, diese auch auf Stufe der Encoder und Decoder abgebildet werden müssen. Einen Vorteil, welchen die Encoder dafür bieten, ist, dass feingranular bestimmt werden kann, welche Daten tatsächlich abgelegt werden sollen. So ist

[5] https://guide.elm-lang.org/interop/flags.html
[6] https://guide.elm-lang.org/interop/ports.html

in dieser Entwicklungsaufgabe z. B. entschieden worden, dass der aktuelle State eines ausführenden Requests (beispielsweise ob er gerade ausgeführt wird) nicht gespeichert werden soll, da dieser State nach einem Reload nicht mehr der Realität entspricht (der Browser bricht den Request bei einem Reload ab). Durch die Decoder ist immer sichergestellt, dass nur Objekte, welche den Typ-Definitionen entsprechen, in die Applikation transportiert werden können.

4.5.4 TypeScript

Da TypeScript keine anderen Datentypen als JavaScript unterstützt, können TypeScript-Objekte exakt gleich mit JSON.stringify zu einem String serialisiert und mit JSON.parse[7] wieder zu einem JavaScript- bzw. TypeScript-Objekt umgewandelt werden. Da diese Variante sehr einfach und idiomatisch ist, wurde sie ohne erweiterte Variantensuche gewählt.

Da JSON.stringify und JSON.parse keine einfache Variante bieten, um feingranulare Anpassungen am Serialisierungsprozess zu machen, werden diese Funktionen direkt so verwendet. Das bedeutet aber, dass z. B. der Request-State auch gespeichert wird.

Wirkliche Probleme gibt es aber erst bei der Deserialisierung. Da nicht sichergestellt werden kann, dass die Daten ausserhalb der Applikation nicht verändert werden, muss damit gerechnet werden, dass sie nicht mehr den Typ-Definitionen entsprechen. TypeScript bietet keine einfache Möglichkeit, um dies sicherzustellen. Deshalb muss eine externe Library verwendet werden, welche z. B. mithilfe von einem Schema sicherstellt, dass Typ-Definitionen und Daten übereinstimmen. Mit einer solchen Library können auch die Daten, welche nicht abgelegt werden sollten, bei der Deserialisierung ignoriert werden. Der Trade-Off ist auch hier, dass dieses Schema bei einer Änderung im State angepasst werden muss. Im Gegensatz zu Elm betrifft dies aber nur die Deserialisierung.

4.5.5 Reason

In Reason hat man dank BuckleScript Zugriff auf die *LocalStorage* als auch JSON.stringify bzw. JSON.parse. Die Lösung ähnelt in dieser Hinsicht sehr der von TypeScript. Da auch hier algebraische Datentypen nicht einfach serialisiert und deserialisiert werden können (bzw. nur im Vertrauen darauf, dass BuckleScript deren Implementierung nicht verändert), wurde die gleiche JSON-Encode/-Decode-Library wie bei den HTTP-Requests verwendet. In dieser Hinsicht gleicht die Implementation somit der von Elm. Auch bei Reason können bei den Encodern Felder ignoriert werden und bei Decodern wird sichergestellt, dass die Daten den Typ-Definitionen entsprechen.

[7] https://developer.mozilla.org/de/docs/Web/JavaScript/Reference/Global_Objects/JSON

4.6 Formatierung des Bodys

4.6.1 Gegenstand der Entwicklung

Die Antwort eines Servers sollte formatiert werden versuchen, falls im Header *Content-Type: application/json* definiert war. Die Formatierung wird durch eine JavaScript-Library durchgeführt. Damit die Daten formatiert werden können müssen diese zuerst geparst werden. Eine Library wird verwendet, um die Interaktion der verschiedenen Sprachen mit der JavaScript-Welt auszutesten.

Remo Dörig führte diesen Entwicklungsschritt durch.

Die Formatierung sorgt dafür, dass alle JSON-Antworten gleich formatiert sind, wie unten zu sehen ist.

```
{
  "userId": 1,
  "id": 1,
  "title": "sunt aut facere repellat
provident occaecati excepturi optio
reprehenderit",
  "body": "quia et suscipit\nsuscipit
recusandae consequuntur expedita et
cum\nreprehenderit molestiae ut ut quas
totam\nnostrum rerum est autem sunt rem
eveniet architecto"
}
```

```
{ "userId": 1
, "id": 1
, "title": "sunt aut facere repellat
provident occaecati excepturi optio
reprehenderit"
, "body": "quia et suscipit\nsuscipit
recusandae consequuntur expedita et
cum\nreprehenderit molestiae ut ut quas
totam\nnostrum rerum est autem sunt rem
eveniet architecto"
}
```

Abbildung 6: Die Formatierung einer Beispiel-Antwort vorher (links) und nachher (rechts).

4.6.2 Entwicklung TypeScript

Recherche Nach einer kurzen Suche war bereits klar, wie eine normale JavaScript-Library eingebunden werden kann.

Umsetzung Das Parsen der Daten funktioniert normal über *JSON.parse* von TypeScript bzw. JavaScript selber. Die Library wird über *require* im TypeScript-File eingebunden, anstatt über *import*. Das ist nötig, da es für die verwendete Library keine TypeScript-Typ-Definitionen gibt, für viele Libraries existieren diese aber. JavaScript-Libraries können ganz normal über *import* verwendet werden, wenn für diese Libraries Typ-Definitionen für TypeScript vorhanden sind.

Probleme bei der Entwicklung Die Entwicklung bot fast keine Probleme. Der Ort der Einbindung musste noch entschieden werden, hatte aber keinen Einfluss.

Sicherheit Es besteht keine Sicherheit und nur wenige Informationen über die Daten, welche die externe Library dem TypeScript-Code übergibt. Es könnte ein Schnittstellen-Handler implementiert werden, dies ist aber nicht notwendig.

4.6.3 Entwicklung Reason

Recherche Es gibt verschiedene Wege, wie Reason mit der JavaScript-Welt kommuniziert. Es brauchte ein wenig Recherche, um den für diesen Fall idealen Weg zu finden. Es war nicht hilfreich, dass die Informationen über die Dokumentationen von Reason und BuckleScript verteilt sind.

Umsetzung Das Parsen der Daten ist in Reason so nicht einfach möglich, da der Inhalt nicht bekannt ist, daher wird das Parsen auch via JavaScript gemacht. Für die Interaktion mit JavaScript werden Compilerbefehle in speziellen Klammern definiert. Dem Compiler wird gesagt, welche Typen den Funktionen übergeben werden und welche zurückgegeben werden. Der Typ des Parsens ist ein abstrakter Typ, welcher dann direkt der Formatierungsfunktion übergeben wird. Dieser Typ wird nur von JavaScript verwendet. Das Laden des Moduls übernimmt Reason/BuckleScript direkt selber.

Probleme bei der Entwicklung Es gibt verschiedene Arten, wie es eingebunden werden kann. Der richtige Weg war nicht gerade offensichtlich. Die Funktion der Library und die Funktion von JavaScript selber werden verschieden eingebunden. Wie der Typ der Daten, welche geparst werden und dann dem Formatierer übergeben werden, definiert werden muss, benötigte eine kurze Recherche. Man muss einen Typ definieren ohne irgendwelche Werte. Dies nennt man in Reason einen abstrakten Typen. Diesen Typ kann man in Reason nur an andere externe Funktionen übergeben.

Sicherheit Es besteht keine Sicherheit über die Daten, welche die externe Library dem Reason-Code übergibt. Der Code weiss jedoch, um welche Typen es sich handeln sollte. Die Typen werden jedoch von Reason selber nicht überprüft.

4.6.4 Entwicklung Elm

Recherche In Elm gibt es über die verschiedenen Versionen verschiedene Wege mit der JavaScript-Welt zu interagieren. Um eine direkte Antwort von einer externen Funktion zu erhalten, konnten Kernel-/Native-Module verwendet werden. Das Problem ist, dass deren Dokumentation nicht sehr gut ist und dass ab Version 0.19, welche verwendet wird, diese Module nur noch vom Elm-Team selber erstellt werden können. Bei der Recherche ist viel Unmut über diesen Entscheid in der Community gefunden worden. Es gibt Umwege wie es trotzdem möglich wäre.

- Compiler forken und diese Überprüfung entfernen
- Ein von Elm entwickeltes Modul herunterladen und die gecachten Daten ändern

Beide Wege haben aber offensichtlich gravierende Nachteile.

Daher wurde der Plan, über Kernel-/Native-Module die JavaScript-Library anzusprechen, wieder verworfen. Für die Umsetzung wurden dann Ports verwendet. Über Ports können Nachrichten gesendet und Empfangen werden, jedoch kann keine Antwort auf eine Anfrage entgegengenommen werden. Ein Port empfängt entweder Nachrichten oder sendet Nachrichten, aber nicht beides. Es gibt Libraries, welche versuchen eine Antwort einer Anfrage zuzuordnen.

Umsetzung Die Umsetzung erwies sich als mühsam. Das Parsen der Daten kann nicht über Elm gemacht werden und wird daher auch direkt im JavaScript-Code umgesetzt. Die Umsetzung erfolgte über Ports. Da über Ports nicht direkt Anfrage und Antwort zugeordnet werden kann, gibt es nur eine einzige formatierte Antwort, immer die momentan selektierte Antwort. Man hätte den Anfragen IDs mitgeben können, dies wäre aber ein weiterer Overhead für nur eine Formatierung. Die Handhabung der Ports basiert auf oberster Ebene im Code. Der JavaScript-Teil muss in einem separaten File implementiert werden, Elm-Code und JavaScript-Code können nicht im gleichen File sein. Das Modul wird in JavaScript normal über die JavaScript-Funktionalitäten eingebunden.

Probleme bei der Entwicklung Die Recherche brauchte relativ viel Zeit. Die Implementierung ist im Vergleich zu den anderen zwei Sprachen kompliziert und fehleranfälliger. Das Resultat der Umsetzung ist auch schlechter als das der anderen Programmierungen, sowohl in Funktionalität, Erweiterbarkeit wie auch in Klarheit. Die formatierten Daten werden temporär abgelegt, anstatt nur angezeigt. Ein Caching, damit die Daten nur bei Änderungen der Daten geändert werden, musste manuell implementiert werden.

Sicherheit Falls der externe JavaScript-Code nicht das zurückgibt, was der Elm-Code erwartet, loggt die Elm-Runtime eine Fehlermeldung und nimmt die Daten nicht an. Dadurch ist man sich sicher, dass im Elm-Code immer die korrekten Typen verwendet werden. Handhabung der Fehler muss in diesem Fall von JavaScript übernommen werden.

4.7 Testing

4.7.1 Gegenstand der Entwicklung

Bei dieser Aufgabe war das Ziel, die automatisierten Tests in allen drei Applikationen zu überprüfen und wenn nötig zu erweitern. Bei allen drei Applikationen wurde die Request-Komponente (das Formular, welches zum Erstellen eines Requests verwendet wird) mit einem Komponenten-Test getestet.

Dieser Entwicklungsschritt wurde von Joel Fisch durchgeführt.

4.7.2 Allgemein

In diesem Report wird nur auf die Komponenten-Tests eingegangen, da bisher keine solchen Tests entwickelt wurden. Unit-Tests für die Pure-Functions des Stores wurden weitestgehend während der Entwicklung der jeweiligen Teile erstellt.

4.7.3 Entwicklung TypeScript

Es gibt einige etablierte React-Testing-Libraries (z. B. Enzyme[8], ReactTestUtils[9], react-testing-library[10]) [4]. Empfohlen wird von React die *react-testing-library* [5], welche für dieses Projekt verwendet wurde. Diese Library erlaubt es, Komponenten zu HTML zu rendern und darauf Annahmen zu treffen. Im Gegensatz zu Enzyme gibt

[8] https://airbnb.io/enzyme/
[9] https://reactjs.org/docs/test-utils.html
[10] https://testing-library.com/react

es keine Unterstützung von Methoden wie Shallow Rendering, da damit oft nur Implementationsdetails getestet werden (eine Komponente sollte Aufgaben in private Sub-Komponenten auslagern können, ohne die Tests fehlschlagen zu lassen). Auch in anderer Hinsicht versucht diese Library, möglichst Tests aus Benutzersicht zu bevorzugen. So kann beispielsweise ein Input-Feld anhand des assoziierten Labels ausgewählt werden.

Mit dieser Library konnte die Request-Komponente einfach getestet werden. Da Aktionen als reine Side-Effects funktionieren, müssen die übergebenen Event-Handler gemockt werden. Dies ist mit den Core-Funktionen von Jest aber ohne weiteres möglich.

4.7.4 Entwicklung Reason

Die gleiche Testing-Library (*react-testing-library*) wie für TypeScript wurde auch bei Reason eingesetzt, da es sich hier auch um React handelt und es bereits BuckleScript-Bindings für diese Library gab. Leider wurde schnell klar, dass diese Bindings sowohl unvollständig als auch schlecht dokumentiert sind. Gewisse Funktionalitäten konnten selber nachgerüstet werden, bei anderen wurde eine Alternative verwendet.

Die zweite Hürde war das Mocking von Funktionen in Reason mit Jest. Bei den Jest-Bindings für BuckleScript ist diese Funktionalität als experimentell eingestuft und auch entsprechend schlecht dokumentiert. Nach einigem Suchen und Ausprobieren klappte dies schliesslich aber auch.

4.7.5 Entwicklung Elm

In Elm bietet die etablierte Library *elm-test*, welche auch für die Unit-Tests der Funktionen verwendet wurde, Funktionalitäten für das Testen von Views an. Besonders komfortabel gelöst ist das Testen von Aktionen (Commands in Elm), welche vom Benutzer ausgelöst werden. Im Gegensatz zu den zwei auf React basierenden Ansätzen verwendet Elm hierfür keine Impure-Functions und deshalb wird auch kein Mocking benötigt. Entsprechend schnell konnten die Komponenten-Tests auch implementiert werden (siehe Kapitel 5.8.3).

4.8 Change-Request

4.8.1 Gegenstand der Entwicklung

Im Rahmen eines Change-Requests wurde eine Simple-View entwickelt, welche nicht so häufig benötigte Elemente ausblendet. So sind sowohl beim Request als auch bei der Response die Header nicht sichtbar. Weiter wird die History nicht mehr angezeigt, sondern nur durch einen Vor- und Zurück-Button repräsentiert. Schliesslich wird sowohl auf der Mobile- als auch der Desktop-Ansicht die Liste der Tabs als Drawer angezeigt, welcher über den Menu-Button in der App-Bar geöffnet werden kann. Unten ist die umgesetzte Simple-View zu sehen.

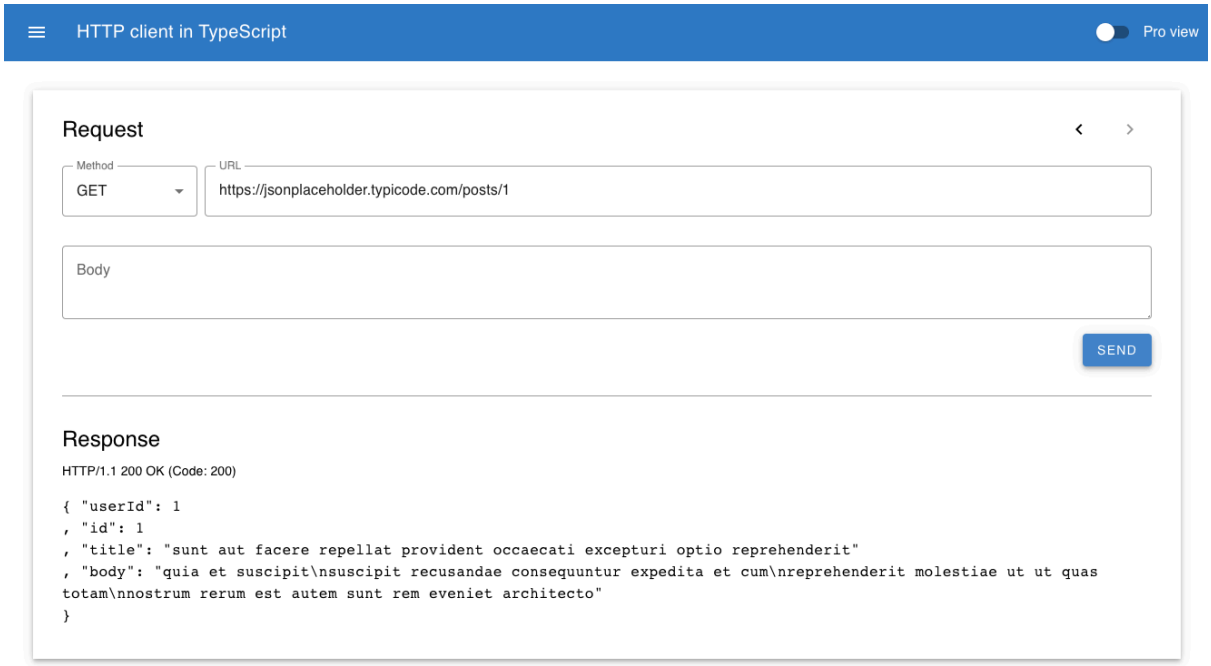

Abbildung 7: Der HTTP-Client in der Simple-View.

Das hauptsächliche Ziel dieser Entwicklungsaufgabe war aber nicht das effektive Umsetzen dieser Anforderungen, sondern das Testen, wie einfach und effizient Änderungen bei einer bestehenden Applikation mit den jeweiligen Technologien durchgeführt werden kann.

Joel Fisch führte diesen Entwicklungsschritt durch.

4.8.2 Allgemein

Da die Design-Änderungen bei allen drei Applikationen gleich waren, wurden diese im Vorfeld durchgeführt und sind in den effektiven Zeiten pro Sprache nicht enthalten.
Für diese Änderungen wurden ungefähr zwei Stunden benötigt.

4.8.3 Entwicklung Elm

Die Entwicklung von Elm dauerte ungefähr 1.5 Stunden. In dieser Zeit wurden sechs Source-Dateien und drei Test-Dateien verändert. Es wurde auf Root-State-Level ein neuer boolescher Wert eingeführt, der angibt, ob die Applikation sich im Pro- oder Simple-Modus befindet. Dieser kann über einen Switch-Button, welcher sich in der App-Bar befindet, verändert werden. Auf verschiedenen Stufen wurden Elemente (Header und History) aufgrund dieses Werts ausgeblendet.

Die grössten Änderungen waren bei der History nötig, da diese die zwei neuen Aktionen Vor und Zurück unterstützen muss. Da die History ein eigenes Modul ist, konnten lediglich die vier neuen Methoden (zwei für die Aktionen und zwei für die Abfrage, ob die Aktionen aktiviert sind oder nicht) implementiert und mit neuen Tests getestet werden. Bei diesen Änderungen zeigte sich, dass die Benennung der internen Datenstruktur der History nicht optimal gewählt worden war. Mittels Rename in der IDE konnte dies schnell angepasst werden.

Die grösste Zeitverzögerung wurde dadurch verursacht, dass eine Komponente in eine eigene Funktion ausgelagert wurde, diese aber noch gar nicht verwendet wurde. Dies ist ein klarer Fehler auf der Seite des Entwicklers, mittels Usage-Analyse und Warnungen von ungenutzten Funktionen hätte er hier aber unterstützt werden können.

4.8.4 Entwicklung TypeScript

Die Entwicklung von TypeScript dauerte ungefähr 1.75h. In dieser Zeit wurden 18 Source-Dateien und drei Test-Dateien verändert. Auch hier wurde auf Root-State-Level ein neuer Wert hinzugefügt. Da aber ein Hinzufügen direkt auf Root-Level sehr unpraktisch ist, wurde ein neuer State-Tree direkt unter dem Root-State angelegt, welcher die Toggle-Aktion und den entsprechenden Reducer enthält. Dieser Wert wurde mittels Selektors an die nötigen React-Komponenten angebunden, um die Elemente auszublenden.

Da die History ihren eigenen State-Tree hat, konnte die Logik für Vor und Zurück direkt dort implementiert werden. Diese war aufgrund der internen Repräsentation der Daten deutlich aufwendiger als bei Elm.

Eine grössere Verzögerung als erwartet verursachte die Logik zum Überschreiben des aktiv editierten Requests. Bei der Navigation in der History werden immer die Daten des ausgewählten History-Eintrags eingefüllt. Zusätzlich erhält der Benutzer die Option, diesen Vorgang rückgängig zu machen. Da diese Logik vom Resultat der History-Auswahl abhängig ist, muss der Request-View irgendwie Zugriff auf die History gestattet werden. Dies wurde deshalb in einer Saga implementiert, welche beim Verändern des History-Eintrags den aktuell ausgewählten Request in der Request-View durch eine weitere Action setzt.

4.8.5 Entwicklung Reason

Die Entwicklung von Reason dauerte ungefähr 1.6h. In dieser Zeit wurden 22 Source-Dateien und vier Test-Dateien verändert. Die Implementation war weitestgehend identisch mit der von TypeScript. Auch hier musste ein neuer State-Subtree für den booleschen Wert hinzugefügt werden und mittels Selektoren an die Komponenten gebunden werden.

Die Veränderung der History war gleich wie bei Elm, da sich dort die internen Datenstrukturen entsprachen. Gleich wie bei TypeScript entspricht die History einem eigenen Reducer inkl. Actions und Selektoren.

Die gleiche Problematik bei Zugriff der Request-View auf die History ergab sich auch bei Reason. Die Lösung wurde gleich wie bei TypeScript mittels der Middleware für Side-Effects gelöst.

5 Vergleich

Im folgenden Abschnitt werden einzelne Aspekte zwischen den Technologien basierend auf den Untersuchungsreporten verglichen.

5.1 Stabilität

5.1.1 Vergleichskriterien

Bei diesem Vergleichspunkt geht es um die Stabilität zur Laufzeit. Folgende Kriterien sind wichtig:

- Wie gut können Laufzeitfehler verhindert werden?
- Wie stabil ist die Laufzeitumgebung?

Dieser Vergleich berücksichtigt nicht nur die Sprachen, sondern auch die verwendeten Technologien für die Implementierung der Elm-Architektur. Deshalb werden im Folgenden die dafür benötigten Libraries zur Laufzeitumgebung gezählt.

5.1.2 Allgemeines

Die getesteten Laufzeitumgebungen sind generell stabil. Im Rahmen dieses Projektes sind keine Fehlverhalten bezüglich der Laufzeitumgebungen aufgefallen, daher werden für diesen Vergleich mögliche Fehlerquellen der Laufzeitumgebung im Vergleich zu JavaScript gewertet.

5.1.3 TypeScript

Die Laufzeitumgebung von TypeScript ist fast identisch wie die von JavaScript, auf welcher alle untersuchten Laufzeitumgebungen aufbauen. Folglich sind Fehler, welche vom TypeScript-Compiler zur Laufzeitumgebung hinzugefügt werden, am unwahrscheinlichsten.

Bei der Entwicklung war TypeScript die einzige Sprache, bei der es zu Laufzeitfehlern gekommen ist. Dies ist dadurch zu erklären, dass das Typsystem das unsicherste ist und somit Fehler nicht bereits beim Kompilieren gefunden werden.

Positives

- Laufzeitumgebung ist fast direkt JavaScript

Negatives

- Laufzeitfehler durch schwaches Typsystem

5.1.4 Reason

Wie bei TypeScript gibt es keine wirkliche Laufzeitumgebung zwischen Reason und JavaScript, die Umwandlung des Codes ist aber komplexer. Die Standard-Bibliotheken sind nicht gleich wie die von JavaScript und stellen daher eine weitere mögliche Quelle von Fehlverhalten dar, was sich im Rahmen dieses Projektes jedoch nicht auswirkte.

Durch das sichere Typsystem sind keine Laufzeitfehler aufgetreten während der Entwicklung. Glaubt man dem Blog von Reason, so konnte dank Reason die Laufzeitstabilität des Facebook Messengers deutlich verbessert werden [3].

Positives

- Keine Laufzeitfehler bei der Entwicklung

5.1.5 Elm

Bei Elm ist die Laufzeitumgebung am wenigsten direkt JavaScript, was eine mögliche Fehlerquelle darstellt. Die Elm-Umgebung verhindert bzw. erschwert durch eine deklarative Verwendung menschliche Fehler, welche sich zur Laufzeit auswirken. Die strikte Abgrenzung zu JavaScript verlagert einige Logik in den JavaScript-Teil, welcher instabiler ist.

Durch das sichere Typsystem sind keine Laufzeitfehler aufgetreten während der Entwicklung.

Positives

- Keine Laufzeitfehler bei der Entwicklung
- Elm-Umgebung beugt menschlichen Fehlern teils vor

Negatives

- Laufzeitumgebung ist mögliche Quelle von Fehlern

5.1.6 Fazit

Unterschiede Durch ein stabiles und intelligentes Typsystem verhindern Reason und Elm die meisten Laufzeitfehler. Die Standard-Bibliotheken verhalten sich bei allen drei Varianten stabil.

Elm weicht im Vergleich zu den anderen zwei Sprachen am meisten von der JavaScript-Umgebung ab.

Wertung Elm gewinnt dadurch, dass einerseits das Typsystem die meisten Laufzeitfehler verhindert, andererseits neigt die deklarative Art der Elm-Umgebung dazu, Laufzeitfehler zu verhindern. Reason ist an zweiter Stelle. Das Typsystem verhindert Laufzeitfehler gleich gut wie Elm. An letzter Stelle steht TypeScript, welches Laufzeitfehler am schlechtesten vorbeugt.

5.2 Testbarkeit

5.2.1 Vergleichskriterien

Die Testbarkeit einer Applikation hat eine immer höhere Relevanz, insbesondere wenn es sich um langlebige Produkte handelt. Da dies bei Web-Applikationen immer häufiger der Fall ist, müssen auch Web-Frontends einfach automatisiert getestet werden können. Durch automatisierte Tests kann die korrekte Funktionalität bei Refactorings und Bugfixes wie auch bei Erweiterungen leichter sichergestellt werden.

Für den Vergleich sind insbesondere folgende Punkte wichtig:

- Wie können Funktionen getestet werden?
- Wie können Komponenten getestet werden?
- Wie können Side-Effects getestet werden?
- Gibt es weitere nützliche Test-Methoden, welche ohne Zusatzaufwand unterstützt werden?
- Wie einfach erhält der Entwickler Feedback über fehlgeschlagene Tests?
- Gibt es Vorschläge der Frameworks/Sprachen, wie End-to-End-Tests und Integrationstests umgesetzt werden sollen?

5.2.2 Allgemein

Für alle drei Varianten gilt, dass sie die Elm-Architektur verwenden. Dies führt in gewissen Bereichen automatisch zu ähnlichen Charakteristiken, welche deshalb hier für alle gemeinsam erläutert werden. Die Elm-Architektur setzt durch ihre unidirektionale Natur stark auf den Einsatz von Pure-Functions. Da das Resultat einer Pure-Function nur von ihren Input-Werten abhängt, ist kein Wissen über die interne Implementation nötig, um eine solche Funktion zu testen. Damit stellt diese Kategorie von Funktionen die am einfachsten testbare dar. Das führt automatisch zu einer guten Testbarkeit aller drei Varianten im Bereich des Stores/State-Managements. Auch Komponenten werden damit einfacher testbar, da sie nur vom aktuellen State abhängen.

Die Geschwindigkeit, mit welcher Tests ausgeführt werden können, ist einerseits für die Entwicklung wichtig, um schnelles Feedback zu bekommen, andererseits werden die Tests typischerweise in einer Pipeline ausgeführt. Sollen Entwicklungsansätze wie TDD eingesetzt werden, muss sichergestellt werden, dass die Entwicklung nicht durch eine zu langsame Test-Ausführung behindert wird. Alle drei Varianten ermöglichen es, Tests im Watch-Modus (d. h., dass Datei-Änderungen automatisch getestet werden) und nur auf im VCS geänderten Dateien ausführen zu lassen.

Tabelle 2: Durchschnittliche Testausführzeit aller Tests

TypeScript	5.8s
Reason	3.5s
Elm	2.7s

5.2.3 TypeScript

TypeScript bietet innerhalb des Ökosystems eine Vielzahl an Libraries für die verschiedensten Bereiche des Testings an. Für Unit-Tests von Funktionen ist Jest, welches direkt mit Create-React-App geliefert wird, eine sehr gute Wahl. Da TypeScript keine Pureness von Funktionen sicherstellen kann, ist man hier weitestgehend auf die Einhaltung der Konventionen durch die Entwickler angewiesen.

Im Bereich von Komponenten-Tests muss eine zusätzliche Library verwendet werden, von denen es wiederum für jeden Geschmack eine Option gibt. Der Output von Komponenten lässt sich mit solchen einfach testen. Aktionen sind etwas aufwendiger zu testen, da dafür gemockte Funktionen an die Komponente übergeben werden müssen. Diese Funktionen müssen danach auf getätigte Aufrufe überprüft werden. Mocking ist eine Funktionalität, welche Jest anbietet und somit ohne weiteren Aufwand verfügbar ist.

Wenn Side-Effects mit Redux-Saga umgesetzt werden, wird idealerweise eine Test-Library für Redux-Saga verwendet. Es ist zwar grundsätzlich möglich, Sagas ohne Zusatz zu testen, da es sich dabei um Generator-Funktionen handelt. Davon ist aber abzuraten, da so viel über die interne Implementation der Saga bekannt sein muss. Mit einer entsprechenden Testing-Library kann eine bessere Abstraktion von Implementations-Details erreicht werden.

Jest bietet für die Überwachung der Tests ein sehr gutes CLI an. Die Tests werden bei Änderungen fortlaufend ausgeführt und die neuen Resultate sind in der Konsole sichtbar.

Ein zusätzliches Feature (nebst vielen anderen), welches Jest anbietet, ist das Snapshot-Testing. Dabei wird der Output einer geänderten Komponente mit ihrem früheren Output verglichen. Änderungen am Output müssen vom Entwickler manuell bestätigt werden (im CLI von Jest). Anhand des neuen Snapshots, welcher versioniert wird, sind die Änderungen auch im Code-Review sichtbar.

Für E2E-Tests bietet React keine speziellen Optionen an. In der Dokumentation [5] wird darauf verwiesen, dass E2E-Testing nichts direkt mit React-Komponenten zu tun hat und somit Out-of-Scope ist. Die meisten Tutorials führen die etablierten Optionen wie Selenium[11], Puppeteer[12] oder Cypress[13] auf.

[11] https://selenium.dev/
[12] https://github.com/puppeteer/puppeteer
[13] https://www.cypress.io/

Positives

- Viele etablierte Libraries für Testing (inkl. guter Dokumentation)
- Side-Effects können dank Sagas relativ gut getestet werden
- Jest: Gutes CLI

Negatives

- Pure-Functions können nicht garantiert werden
- Sehr viele verschiedene Meinungen und Varianten, wie getestet werden soll.

5.2.4 Reason

Im Reason-Ökosystem werden oft die gleichen Libraries wie in JavaScript verwendet (mit entsprechenden BuckleScript-Bindings). So ist auch hier Jest die typische Wahl als Testing-Library und bringt deshalb die gleiche Funktionalität mit (wie z. B. das CLI). Auch Reason kann keine Garantien bezüglich Pure-Functions abgeben, allerdings bietet es mehr Sicherheit im Bereich Immutability.

Komponenten-Tests werden wiederum mit den gleichen Libraries geschrieben wie bei TypeScript, meist aber mit weniger Funktionalität und schlechterer Dokumentation. Aktionen müssen auch hier gemockt werden, dies ist aber weniger einfach möglich.

Side-Effects sind nur sehr aufwendig testbar, da es für Reason bzw. Reductive keine Library wie Redux-Saga gibt. Um sie zu testen, müssten Module mit Jest gemockt werden.

Was für React in Bezug auf E2E-Tests gilt, gilt auch für ReasonReact. Somit gibt es von der Sprache und der Library her keine klaren Vorschläge.

Positives

- Gewisse Garantien bezüglich Immutability
- Jest: Gutes CLI

Negatives

- Pure-Functions können nicht garantiert werden
- Side-Effects können fast nicht getestet werden
- BuckleScript-Bindings für JavaScript-Testing-Libraries sind unvollständig und schlecht dokumentiert

5.2.5 Elm

Elm bietet eine einzige etablierte Library für das Testing an, welche sowohl für Unit-Tests von Funktionen als auch für Komponenten-Tests eingesetzt wird. Da Elm nur Pure-Functions kennt, hat man die entsprechenden Garantien. Auch View-Funktionen sind pure und können so ohne Mocking von Funktionen getestet werden.

Side-Effects sind in Elm nur durch die Elm-Runtime möglich. Diese Side-Effects sind deshalb innerhalb des Elm-Codes nur als Beschreibung in Form eines Datenobjekts vorhanden. Der eigentliche Side-Effect kann und muss somit nicht getestet werden, da die Elm-Runtime die korrekte Funktion sicherstellt. Eigene Side-Effects, welche mit Ports implementiert werden, können innerhalb von Elm nicht getestet werden. Das JavaScript auf der anderen Seite des Ports müsste somit mit einer anderen Library wie z. B. Jest getestet werden.

Die Test-Library von Elm bietet ebenfalls die Möglichkeit, Tests bei Änderungen innerhalb der Konsole auszuführen. So kann Feedback direkt bei der Entwicklung ohne weitere Handgriffe erhalten werden.

Die Library bietet ausserdem Fuzz-Testing an, welches beispielsweise für Property-Tests genutzt werden kann.

Elm bietet keine integrierte Möglichkeit für E2E-Testing an. In der Dokumentation [6] wird dabei auf etablierte E2E-Test-Tools wie Cypress verwiesen.

Positives

- Gute Testing-Library, welche viele Entscheidungen abnimmt
- Pure-Functions sind garantiert
- elm-test: Gutes CLI

Negatives

- Praktisch keine Alternativen bei der Wahl von Testing-Libraries
- Ports können nicht getestet werden (inkl. dem JavaScript)

5.2.6 Fazit

Unterschiede Elm hat eine einzige Testing-Library, TypeScript dagegen eine schier unüberschaubare Anzahl an Testing-Libraries. Reason steht mit einer eingeschränkten Variante der TypeScript-Libraries in der Mitte.

Reason und TypeScript ähneln sich bei den Komponententests sehr, da beide React verwenden. Elm unterscheidet sich in diesem Bereich, da seine Komponenten/View-Funktionen Pure-Functions sind.

Alle Varianten bringen einen Test-Runner auf der Konsole mit, welcher Dateiänderungen überwacht. Keine der Varianten bietet eine vorgegebene Variante für E2E-Tests.

Wertung Reason hat von allen dreien am wenigsten zu bieten, da die Libraries von TypeScript/JavaScript nur eingeschränkt verwendet werden können, aber alle Tests grundsätzlich gleich umgesetzt werden müssen. TypeScript und Elm haben sehr grosse Unterschiede und je ihre eigenen Vor- und Nachteile. Während Elm klare Vorgaben mit wenigen Optionen und viel Sicherheit und Klarheit bietet, hat TypeScript eine sehr grosse Auswahl und einen Weg für fast jedes Problem.

5.3 Interaktion mit JavaScript

5.3.1 Vergleichskriterien

Für die Web-Entwicklung ist es in vielen Projekten notwendig, mit der JavaScript-Welt zu interagieren. Es gibt eine Vielzahl von Libraries in JavaScript, welche genutzt werden können. Einige Funktionalitäten, welche der Browser zur Verfügung stellt, sind in der verwendeten Version einer der drei Sprachen eventuell noch nicht verfügbar.

Für den Vergleich sind insbesondere folgende Punkte wichtig:

- Wie einfach kann JavaScript-Code angesprochen werden?
- Wie sicher kann der JavaScript-Code angesprochen werden?
- Wie funktioniert die Kommunikation zwischen JavaScript und dem internen Code?
- Wie einfach können Libraries eingebunden werden?

5.3.2 TypeScript

TypeScript ist sehr nahe an JavaScript, da es sich um ein Superset von JavaScript handelt. Daher ist die Integration von JavaScript-Code sehr einfach. Der Unterschied zwischen verwendeten JavaScript-Libraries in TypeScript und TypeScript-Libraries selber ist sehr gering. Dies ist sowohl ein Vorteil als auch ein Nachteil, je nach Ansichtsweise.

Für viele beliebte JavaScript-Libraries bestehen TypeScript-Typ-Definitionen, welche die Integration vereinfachen. Diese Typ-Definitionen können auch manuell erstellt werden. Die normalen JavaScript-Funktionalitäten, welche vom Browser gegeben sind, sind sowieso alle bereits verfügbar.

Positives

- Sehr einfach
- Standard-JavaScript-Funktionen ohne Zusatzaufwand verfügbar
- Integrierter Code lässt sich gleich behandeln wie der TypeScript-Code

Negatives

- Es besteht keine klare Abgrenzung
- Fehler können sich entgegen den definierten Typen in die TypeScript-Welt fortpflanzen

5.3.3 Reason

Die Integration mit JavaScript ist eines der primären Ziele von Reason und BuckleScript. Die Integration funktioniert über verschiedene Compiler-Anweisungen. Diese sind klar als solche erkennbar. Die externen Funktionen können gleich wie Reason-Funktionen verwendet werden. Libraries können direkt über BuckleScript selber eingebunden werden. Wie “diszipliniert” JavaScript eingebunden wird, kann der Programmierer teils selber entscheiden. Nebst der Integration von JavaScript hat Reason den Vorteil, dass auch OCaml-Libraries verwendet werden können. Die definierten Typen werden nicht überprüft.

Positives

- Relativ einfach
- JavaScript- und Reason-Code können im selben File verwendet werden
- Integrierter Code lässt sich gleich behandeln wie der Reason-Code
- Es ist dem Programmierer relativ frei, wie strikt definiert er die Integration haben will
- Abgrenzung ist klar

Negatives

- Es müssen für jede externe Integration Bindings definiert werden
- Fehler können sich entgegen den definierten Typen in die Reason-Welt fortpflanzen

5.3.4 Elm

Das primäre Ziel von Elm bezüglich der Integration von JavaScript scheint zu sein, dass die Elm-Welt heil bleibt. Bei der Initialisierung von Elm können Flags von aussen mitgegeben werden. Bis Version 0.19 konnten Native-/Kernel-Module entwickelt werden, welche die Integration von JavaScript-Funktionalitäten erlaubte. Ab Version 0.19 ist dies nur noch für Module, welche vom Elm-Team selber herausgegeben werden, möglich. Die Community ist bezüglich der Integration von JavaScript geteilter Meinung.

Kommunikation mit JavaScript findet über Ports statt. Ports senden entweder Nachrichten in die JavaScript-Welt oder empfangen Nachrichten von der JavaScript-Welt. Synchrone bidirektionale Kommunikation ist nicht möglich. Die Ports werden in Elm auf oberster Ebene eingebunden. Elm stellt sicher, dass keine Daten mit falschen Typen in die Elm-Welt kommen. Elm loggt in diesem Fall eine Fehlermeldung und verarbeitet die Daten nicht weiter.

Typen können in Elm gleich wie JSON geparst werden, was eine Fehlerhandhabung seitens Elm ermöglicht. Abstrakte Typen (wie in Reason), welche nur weitergegeben werden können, sind in Elm nicht möglich. Hierfür muss ein generischer JavaScript-Typ verwendet werden, der keine Restriktionen aufweist.

Module werden im JavaScript-Code von JavaScript selber eingebunden.

Positives

- Fehler des JavaScript-Codes können sich nicht in den Elm-Code fortpflanzen
- Klare und strikte Abgrenzung

Negatives

- Implementation ist im Vergleich aufwendiger
- Synchrone bidirektionale Kommunikation ist nicht möglich
- Integration nur auf oberster Ebene

5.3.5 Fazit

Unterschiede Auf der einen Seite kann TypeScript JavaScript-Code verwenden, als wäre es TypeScript-Code, auf der anderen Seite steht Elm, welches die Kommunikation mit JavaScript über strikte Kanäle leitet. In der Mitte ist Reason, welches dem Programmierer in gewissem Masse die Striktheit der Kommunikation offenlässt. In Reason und TypeScript können Bindings definiert werden, müssen aber nicht. Diese Bindings werden aber bei beiden Sprachen nicht überprüft und sind nur Informationen für den Compiler.

Fehlverhalten seitens JavaScript kann sich in Reason und TypeScript fortpflanzen. Bei Elm wird jegliches Fehlverhalten, welches durch das Typsystem erkannt werden kann, direkt abgeblockt. In Reason und TypeScript kann JavaScript-Code gleich aufgerufen werden wie eigener Code, in Elm nicht.

In Reason und TypeScript kann im selben File JavaScript-Code und eigener Code vorhanden sein. In Reason werden Module über BuckleScript eingebunden, in TypeScript und Elm werden Module über JavaScript-Funktionalitäten eingebunden. Reason erzwingt zwar die Bindings nicht, empfiehlt diese aber stark.

Wertung Bei TypeScript ist die Verwendung von JavaScript-Code ganz klar am einfachsten. Reason bietet, obwohl es kein Superset von JavaScript ist, gute Möglichkeiten, den JavaScript-Code einzubinden. Reason lässt dem Entwickler Freiheiten bei der Integration. Bei Elm ist die Integration von JavaScript-Code klar am aufwendigsten. Die Leitung der Kommunikation über strikte Kanäle macht das Programmieren aufwendiger. Interaktionen müssen von oben nach unten im Code weitergeleitet werden. Eine synchrone, bidirektionale Kommunikation ist gar unmöglich, wodurch für eine einfache Transformation von Daten grosser Programmieraufwand nötig ist. Die Typen, welche in Elm verwendet werden, sind immer sichergestellt. Typenfehler können über die gleiche Methodik wie das Parsen von JSON erkannt werden.

Am einfachsten ist die Interaktion mit JavaScript mit TypeScript. Reason unterscheidet sich sehr wenig von TypeScript bezüglich der Funktionalität der Integration. TypeScript hat aber den Vorteil, dass der Unterschied des eigenen Codes zu JavaScript sehr gering ist. Elm zwingt den Programmierer über vorgeschriebene Schnittstellen mit JavaScript zu interagieren, was die Implementation erschwert und nicht wirklich klarer macht.

5.4 Compiler / Transpiler

5.4.1 Vergleichskriterien

In diesem Abschnitt werden die Compiler / Transpiler der Sprachen verglichen. Während der Entwicklung ist das Feedback (Fehlermeldungen) wichtig, welches man vom Compiler bekommt. Für das fertige Produkt ist wichtig, wie gut der entstandene Code ist.

Für diesen Vergleich sind folgende Punkte wichtig:

- Wie gut helfen Fehlermeldungen weiter?
- Wie verhält sich der generierte Code bzw. wie schnell ist er?
- Wie viele Fehler können bereits beim Kompilieren erkannt werden?
- Muss manchmal aktiv gegen den Compiler gearbeitet werden?

5.4.2 Chrome Audits

Für die Messung der Performance wurde das Tool *Lighthouse*[14], besser bekannt als Chrome Audits, verwendet. Diese Applikation ist Open Source und kommt von Google. Es verwendet den Chrome-Browser, um Messungen durchzuführen und ist daher sehr nahe an den realen Zeiten, welche ein Benutzer erlebt. Nebst der Performanceauswertung erstellt das Tool auch noch andere Metriken, wie SEO und Best Practices.

Im Anhang dieser Arbeit sind die Kurzauszüge aller sechs Audits (pro Sprache einmal für Mobile und einmal für Desktop) zu finden.

5.4.3 Allgemeines

Beim Entwickeln war die Zeit zum Kompilieren jeweils irrelevant, da andere Faktoren wie Webpack wesentlich mehr Zeit benötigten. Bei allen drei Applikationen ist die Performance der gebuildeten Clients sehr gut. Die Performanceauswertungen wurden jeweils auf einem lokalen Rechner durchgeführt, um die Varianz des Servers von *gitlab.com* zu vermeiden. Beim Audit von Chrome gab es nur bei der Performance relevante Unterschiede, *Best Practices* ergab in allen Fällen 93% und *SEO* ergab 90% bei Mobile und 88% beim Desktop.

5.4.4 TypeScript

TypeScript ist ein Superset von JavaScript. Dies hat zur Folge, dass sich zwischen dem generierten JavaScript-Code und dem geschriebenen TypeScript-Code keine grossen Unterschiede bilden. Die Hauptarbeit des Compilers ist das Überprüfen der Typen.

[14] https://developers.google.com/web/tools/lighthouse

Die Fehlermeldungen sind meistens gut. Sie leiten den Entwickler an die richtige Stelle und beschreiben, was falsch ist und was erwartet wird. Die Fehlermeldungen sind teils unübersichtlich lange, wenn Typen nicht übereinstimmen. Einerseits wird die Verschachtelung der Typen nicht schön aufgeschlüsselt (soll ab Version 3.7 verbessert sein), andererseits können die impliziten Typ-Definitionen sehr lange werden mit Union-Typen.

Im Vergleich zu JavaScript werden sehr viele Fehler bereits beim Kompilieren entdeckt. Die Typsicherheit zieht sich aber nicht durch den ganzen TypeScript-Code. Manchmal muss der Compiler dem Entwickler einfach glauben, dass er schon den richtigen Typ verwendet. Bei der Entwicklung mussten manchmal allgemeine Typ-Definitionen (wie z.B. *any*) verwendet werden, um den Compiler zufriedenzustellen.

Build Der Build für die Produktion dauerte auf der Testmaschine durchschnittlich 3.6 Sekunden und ist somit deutlich der langsamste Build der verglichenen Sprachen. Die resultierenden Files, welche vom Browser heruntergeladen werden, sind 488kb gross und sind somit rund 10-mal grösser als die von Elm generierten Files. Diese Grösse kommt hauptsächlich durch die verwendeten Libraries zustande.

Tabelle 3: Audit von Chrome für TypeScript

Performance Mobile	97%
Performance Desktop	97%

Positives

- Gute Fehlermeldungen
- Generierter Code ist nahe am geschriebenen Code

Negatives

- Vergleichsweise werden wenig Fehler während dem Kompilieren erkannt

5.4.5 Reason

Die Fehlermeldungen des Reason Compilers sind oft nicht hilfreich. Bei einem Syntax-Fehler wird oft gar nicht beschrieben, was für ein Syntax-Fehler es ist und die Zeile, die angegeben wird, ist oft weit entfernt vom Ursprung des Fehlers. Auch die anderen Fehlermeldungen sind nicht immer hilfreich. Es lässt sich einfach konfigurieren, welche Warnungen der Compiler als Fehler zu behandeln hat. Für einige Warnungen gibt es aber keine sauberen Alternativen; diese wurden deshalb abgeschaltet.

Bei der Entwicklung gab es keine Laufzeitfehler, der Compiler erkennt viele Fehler beim Kompilieren. Manchmal muss ein Typ explizit angegeben werden, obwohl der Compiler dies selber herausfinden könnte. Die getroffene Wahl kann dann aber im Gegensatz zu TypeScript in jedem Fall vom Compiler geprüft werden. Der Compiler gibt bei der Interaktion mit JavaScript die Möglichkeit, die Typsicherheit zu umgehen, was aber innerhalb des Projektes nie nötig war.

Zyklische Referenzen werden in Reason auf Modulebene nicht zugelassen, leider gibt es auch hierfür in vielen Fällen keine zielführende Compiler-Nachricht.

Build Der Build ist mit 1.4 Sekunden etwa in der Mitte zwischen Elm und TypeScript. Der Build kann aber sehr gut gecacht werden und wenn dies der Fall ist, braucht er nur noch durchschnittlich 0.26 Sekunden, was noch schneller ist als Elm. Die generierten Files sind mit 752KB im Falle dieses Projekts mit Abstand die grössten.

Der Performance Audit von Chrome ergibt sehr ähnliche Resultate wie bei TypeScript, was wohl an den sehr ähnlichen Libraries liegt. Wenn die Details betrachtet werden, ergeben sich dennoch kleine Differenzen (siehe Anhang).

Tabelle 4: Audit von Chrome für Reason

Performance Mobile	98%
Performance Desktop	97%

Positives

- Mithilfe von Caching schnelles Feedback beim Entwickeln

Negatives

- Fehlermeldungen sind nicht hilfreich
- Generierte Files sind gross

5.4.6 Elm

Der Compiler von Elm gibt generell zielführende Fehlermeldungen. Die Fehlermeldungen sind in einer Art geschrieben, als würde der Compiler mit dem Entwickler reden. Er gibt Auskunft darüber, was erwartet wird und erkennt meistens die korrekte Stelle. Der Compiler erkennt die Typen automatisch, ohne dass in gewissen Fällen trotzdem Typ-Annotationen nötig wären.

Die meisten Fehler können frühzeitig erkannt werden und es traten keine Laufzeitfehler auf.

Build Der Build ist mit durchschnittlich 0.3 Sekunden sehr schnell. Der Browser muss bei der Elm-Version nur 48KB herunterladen, was deutlich am wenigsten ist: 10-mal weniger als bei TypeScript und 15-mal weniger als bei Reason.

Die Performance von Elm ist besser als die von Reason und TypeScript.

Tabelle 5: Audit von Chrome für Elm

Performance Mobile	99%
Performance Desktop	99%

Positives

- Schnelle Kompilierzeit
- Kleine generierte Files
- Sehr gute Performance des generierten Codes

5.4.7 Fazit

Unterschiede Die Fehlermeldungen der Compiler unterscheiden sich stark. Bei Elm ist die Fehlermeldung wie in einem Dialog mit dem Entwickler geführt. Bei Reason sind die Fehlermeldungen noch nicht ausgereift.

Alle drei Compiler sind schnell, der Compiler von Elm ein wenig schneller als die anderen zwei. Die generierten Files sind bei Elm deutlich kleiner, dies liegt auch an den verwendeten Libraries.

Die statischeren Varianten Elm und Reason erkennen mehr Fehler bereits beim Kompilieren. Reason speichert die generierten JavaScript-Dateien direkt bei den Source-Dateien, was dem Compiler ein Caching ermöglicht und ihn bei der Entwicklung sehr schnell macht.

Wertung In dieser Kategorie verliert Reason momentan ganz klar, insbesondere aufgrund der schlechten Fehlermeldungen. Der generierte Code performant zwar gut, ist aber wesentlich grösser als der von TypeScript oder Elm.

Elm gewinnt in dieser Kategorie. Der generierte Code wird am schnellsten erstellt, braucht am wenigsten Platz und ist am schnellsten. Die Fehlermeldungen sind die hilfreichsten und in einer sympathischen Art geschrieben.

TypeScript ist hier in der Mitte, aber näher bei Elm als bei Reason. Die Fehlermeldungen sind etwa gleich hilfreich wie die von Elm, aber der generierte Code braucht mehr Platz und Zeit.

5.5 Projektsetup und CI/CD

In diesem Abschnitt werden das initiale Aufsetzen des Projektes und die Integration in die CI/CD-Umgebung verglichen.

5.5.1 Vergleichskriterien

Beim Projektsetup wird verglichen, wie einfach und schnell ein neues Projekt aufgesetzt werden kann. Dies betrifft insbesondere Bereiche wie Testing, Optimization etc., welche in einem realen Projekt benötigt werden, jedoch nicht in jedem Fall direkt verfügbar sind.

Bei CI und CD wird betrachtet, wie einfach eine solche automatisierte Umgebung eingerichtet werden kann. Dabei wird insbesondere beachtet, wie Tests ausgeführt, Code-Coverage ausgewertet und die Applikation deployt werden kann.

5.5.2 TypeScript

Bei TypeScript ist das Setup und die Integration in CI/CD sehr einfach, da mit *Create React App* ein gut unterhaltenes Tool existiert, welches Testing, Build, Code-Coverage, Linting etc. inklusive guten Standardvorgaben mit sich bringt. Da React auch ohne Redux verwendet werden kann, muss letzteres manuell integriert werden.

Positives

- Einfaches Setup
- Test-Framework enthalten
- Code-Coverage enthalten
- Optimierter Build enthalten
- Linting enthalten
- Einfache Integration von SCSS
- Gute Dokumentation und Anleitungen

Negatives

- Für Buildanpassungen ist Opt-out von Updates von *Create React App* nötig
- Redux und Redux-Saga müssen manuell integriert werden

5.5.3 Reason

Das Setup von Reason ist durch BuckleScript bestimmt, welches ein vernünftiges initiales Setup mitbringt. Weitere Komponenten wie die Test-Library müssen selber integriert werden. Die grössten Knackpunkte stellen Code-Coverage und Linting dar. Reductive muss wie bei TypeScript selber integriert werden. Für das Handling

der Side-Effects stehen keine bestehenden Libraries zur Verfügung und somit muss entweder der Vorschlag von Reductive übernommen oder eine eigene Variante implementiert werden.

Positives

- Einfaches Setup
- Optimierter Build enthalten
- Compiler kann stark konfiguriert und so als Linter verwendet werden
- Einfache Anpassung der Webpack-Konfiguration möglich
- Einfache Integration von Jest

Negatives

- Code-Coverage ist sehr aufwendig zu integrieren
- Keine Linter verfügbar
- Manuelle Integration von Reductive
- Eigene Middleware für Side-Effects

5.5.4 Elm

Bei Elm ist nur ein minimales Setup vorhanden. Alles weitere muss selber integriert werden, was durch Community-Module aber einfach möglich ist. Da Elm mit der Elm-Architektur auch das Handling von Side-Effects und des Stores direkt implementiert, muss in dieser Hinsicht nichts Zusätzliches integriert werden.

Positives

- Community-De-facto-Standards für Testing, Linting, Coverage etc.
- Webpack-Loader für Elm
- Keine Integration von Redux o. ä. nötig
- Anleitung für Build-Optimierung

Negatives

- Kein vollumfängliches Setup, somit eigenes Setup nötig
- Wenige Linting-Rules verfügbar

5.5.5 Fazit

Unterschiede Bei TypeScript ist ein ausgereiftes und einfaches Projektsetup verfügbar, welches praktisch keine Wünsche offenlässt. Bei Elm ist zwar kein Setup vorgegeben, allerdings können dank etablierten Community-Modulen die erwünschten Funktionalitäten relativ schnell umgesetzt werden. Bei Reason gibt es ein gutes Setup, leider sind Dinge wie Linting und Code-Coverage schwieriger als erwartet.

Wertung In dieser Kategorie hat TypeScript einen klaren Vorsprung gegenüber den anderen zwei Kandidaten. Werden weder spezielles Linting noch Code-Coverage-Berechnung benötigt, sind Elm und Reason vergleichbar. Sobald diese Funktionalitäten aber benötigt werden, liegt Reason hinten.

5.6 Ökosystem

5.6.1 Vergleichskriterien

Für viele Probleme werden von den Communities einer Sprache oder Technologie Libraries zu deren Lösung zur Verfügung gestellt. Sprachen mit einer grösseren Community haben folglich meist mehr Libraries als andere Sprachen.

Ein belebtes Ökosystem hilft nicht nur bei der Anzahl an Libraries. Es gibt online mehr Hilfe, zu Fragen werden schneller Antworten gefunden, die Sprache ist besser getestet etc. Systeme mit einem grossen Ökosystem haben ausserdem viel höhere Chancen, langfristig gut einsetzbar zu sein.

In diesem Vergleich wird einerseits betrachtet, wie viele Libraries es gibt und in welchem Zustand sie sind, andererseits wie aktiv die gesamte Community der Sprache ist.

5.6.2 Datengrundlage

Nebst den persönlichen Erfahrungen beim Entwickeln sind für diesen Vergleich die nachstehenden Daten gesammelt worden. Ihre Erfassung fand am 26.11.2019 statt.

NPM-Downloads Wie oft wurde ein Paket in dieser Woche heruntergeladen?
Die Anzahl Downloads gibt Hinweise darauf, wie oft ein Paket eingesetzt wird. Die Daten kommen von npmjs.com selber. Nebst den Compilern für die Sprachen sind noch einige andere wichtige Libraries aufgeführt.

Tabelle 6: Wöchentliche Downloads auf npmjs.com

TypeScript	8’232’810
Elm	13’653
bs-platform (Reason)	12’550
React	6’608’399
Redux	3’771’275
ReasonReact	8’474
Reductive	41

Als Diagramm auf NPM-Trends werden die grossen Unterschiede besonders deutlich:

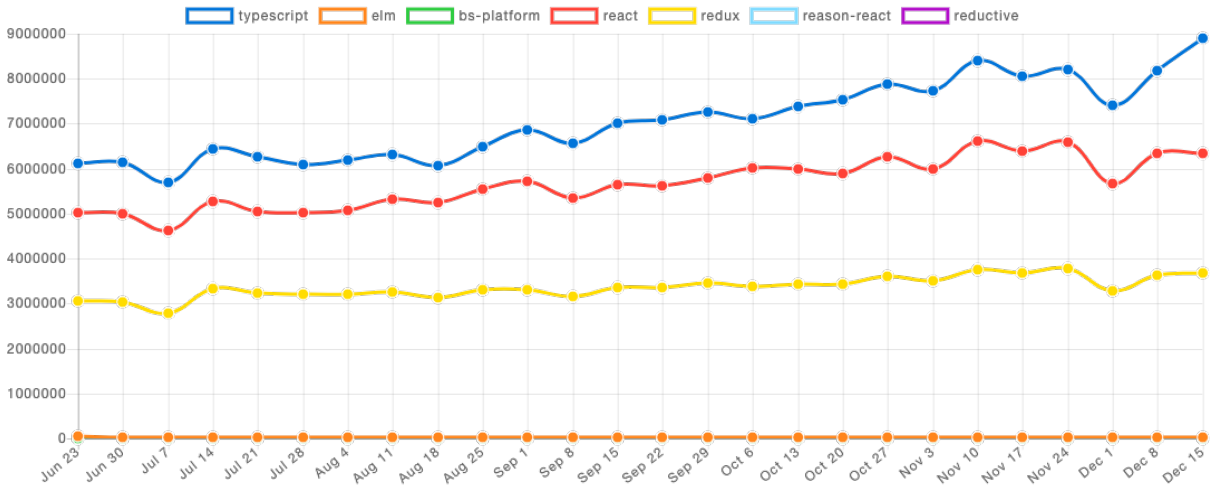

Abbildung 8: Anzahl NPM-Downloads der letzten 6 Monate der verwendeten Libraries.

Stand 17. Dezember 2019

In der oberen Grafik sind aufgrund des TypeScript-Ökosystems die Unterschiede der anderen beiden Ökosysteme nicht ersichtlich. Nachfolgendes Bild zeigt die Unterschiede ohne TypeScript, React und Redux.

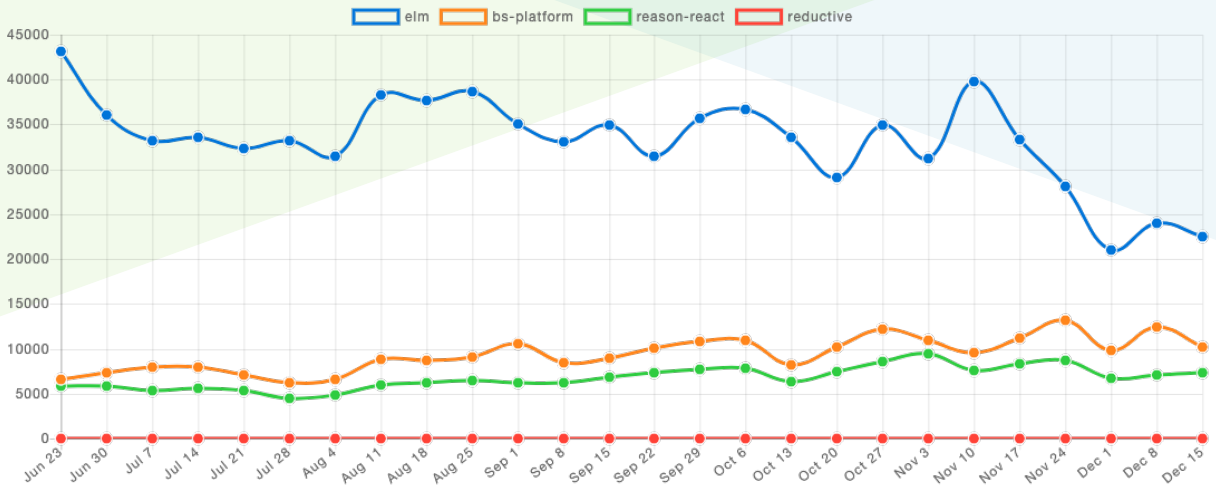

Abbildung 9: Anzahl NPM-Downloads der letzten 6 Monate ohne TypeScript, React und Redux.

Stand 17. Dezember 2019

Repositories auf github.com Wie viele Repositories werden auf github.com/search zu einer Sprache gefunden?
Die Daten kommen von github.com/search selber. Die Anzahl Repositories gibt unter anderem einen Hinweis darauf, wie viele Libraries von der Community entwickelt werden. Für TypeScript wurden 164'769 Repositories gefunden, zu Elm 6'953. Reason wird von GitHub nicht korrekt als eigene Sprache erkannt, sondern wird als OCaml identifiziert. Für OCaml wurden 9'932 Repositories gefunden. Auf gitlab.com steht eine Suche nach Sprache nicht zur Verfügung.

Die drei Sprachen haben alle ihre Compiler auf GitHub. TypeScript hat 56k Sterne, Elm 5.6k Sterne und Reason 8.2k Sterne.

Fragen auf Stackoverflow Wie viele Fragen wurden heute, diese Woche oder diesen Monat auf stackoverflow.com gestellt?
Die Quelle der Daten ist stackoverflow.com selber. Diese Anzahl gibt einen Hinweis darauf, wie viele Leute diese Sprache verwenden.

Tabelle 7: Fragen auf stackoverflow.com

TypeScript	97'521 (127 heute)
Elm	1'580 (17 diesen Monat)
Reason	266 (10 diesen Monat)
Redux	21'783 (20 heute)
React	171'809 (278 heute)
ReasonReact	79 (0 diesen Monat)

5.6.3 TypeScript

TypeScript hat ganz klar das grösste Ökosystem der drei Sprachen. Die Ökosysteme von TypeScript und JavaScript sind zudem sehr nahe beieinander. In allen erfassten Daten für diesen Vergleich hat TypeScript einen deutlichen Vorsprung gegenüber Reason und Elm.

Bei der Entwicklung wird dieses grosse Ökosystem auch spürbar. Für viele Probleme stehen etablierte Libraries zur Verfügung. Einige Libraries der Sprache (beispielsweise React) werden mehr genutzt als Reason und Elm zusammen. Die Sprache steht unter der Führung von Microsoft, was sowohl ein Vorteil als auch ein Nachteil sein kann.

Positives

- Mit Abstand das grösste Ökosystem
- Mit Abstand die meist verwendete Sprache

5.6.4 Reason

Reaon gibt es erst seit 2016, was nicht einmal halb so lange ist wie TypeScript oder Elm. Reason hat gemessen an den Daten das kleinste Ökosystem.

Für viele rudimentäre Probleme stehen noch keine spracheigenen Libraries zur Verfügung (beispielsweise JSON). Zu einigen Problemen gibt es auch keine von der Community bereitgestellte Libraries (beispielsweise Sagas). Die Dokumentation der beim Entwickeln eingesetzten Libraries ist oft nicht gut. Vielfach muss der Source-Code der Library zu Rate gezogen werden, um überhaupt genug Informationen zur Verwendung zu bekommen. Die Libraries sind auch noch nicht so stabil und ändern sich immer wieder. Die Sprache ist momentan noch im Wandel, aber da Reason eigentlich nur ein Dialekt von OCaml ist, ändert sich an den Fundamenten der Sprache relativ wenig.

Reason hat den Vorteil, dass relative einfach TypeScript und JavaScript Libraries verwendet werden können. Reason wird von Facebook entwickelt.

Positives

- Interaktion mit dem JavaScript- und TypeScript-Ökosystem einfach
- Interaktion mit dem OCaml-Ökosystem
- Noch viel Potential

Negatives

- Kleinstes Ökosystem
- Sprache selber ist noch nicht stabil

5.6.5 Elm

Elm ist knapp die älteste Sprache, die verglichen wird. Elm hat zwar ein grösseres Ökosystem als Reason, im Vergleich zu TypeScript ist es aber immer noch sehr klein.

Für die wichtigsten Probleme existieren Libraries. Das Ökosystem von Elm grenzt sich klar von JavaScript, TypeScript und Reason ab.

Im Vergleich zu den anderen Sprachen schränkt Elm den Entwickler auch bei der Wahl der Architektur stark ein. Dies führt dazu, dass die ganze Community in der gleichen Architektur arbeitet und die Libraries jeweils zu dieser Architektur passen. Generell fühlt sich die Community homogener an.

Positives

- Zu den wichtigsten Problemen gibt es etablierte Libraries
- Das ganze Ökosystem ist relativ einheitlich

Negatives

- Ökosystem ist relativ klein
- Ökosystem grenzt sich stark ab

5.6.6 Fazit

Unterschiede TypeScript hebt sich von den anderen beiden Sprachen durch die schiere Grösse des Ökosystems ab. Reason hat eine etwas kleinere Community als Elm, ist aber auch die jüngste Sprache. Reason kann Libraries von JavaScript einfach verwenden. Dadurch, dass Elm die Architektur vorgibt, gibt es dafür relativ viele Libraries.

Wertung In diesem Vergleich siegt TypeScript ganz klar.

Reason und Elm haben vergleichsweise kleine Ökosysteme. Elm hat für den vorgegebenen Weg von Elm gute und genügend viele Libraries. Das Ökosystem von Reason ist nicht so geeint wie das von Elm, die Interaktion mit dem JavaScript-Ökosystem ist dafür viel einfacher. Solange man nicht an die Grenzen des Ökosystems von Elm kommt, ist es besser als das von Reason.

5.7 IDE-Unterstützung

5.7.1 Vergleichskriterien

Eine Entwicklungsumgebung kann die Entwicklung in vielerlei Hinsicht deutlich erleichtern. Dies reicht vom Einfärben von Schlüsselwörtern bis zu komplizierten Refactorings, welche sehr sprachspezifisch sind. Die Präferenzen bezüglich der Unterstützung durch den Editor sind sehr unterschiedlich, weshalb hier auf einige typische Bereiche eingegangen wird, ohne Anspruch auf Vollständigkeit zu erheben.

- Welche passiven Funktionen werden unterstützt (Farbgebung, Fehlererkennung, Hilfestellung bei Fehlern, Quick-Documentation)?
- Welche Unterstützung wird beim Schreiben geboten (Auto-Completion, Context-Sensitivity, automatische/vereinfachte Imports)?
- Welche Möglichkeiten stehen für Refactoring zur Verfügung?
- Welche Navigationsmöglichkeiten werden unterstützt (Jump-to-Definition auch zu fremdem Code, Verwendungsorte anzeigen)?

Je nach Technologie-Stack könnte hier auch die Unterstützung von Builds und Tests direkt in der IDE verglichen werden. Da diese Aufgaben in allen drei Ökosystemen normalerweise direkt auf der Konsole erledigt werden, hat ein solcher Vergleich hier wenig Relevanz.

Zu allen hier dargestellten Erkenntnissen ist anzumerken, dass Editorunterstützungen ein sich sehr schnell veränderndes Feld darstellen. Es kann gut sein, dass ein Community-Plugin in sehr kurzer Zeit viele neue Features dazu erhält oder aufgrund einer neuen Sprache-Version nicht mehr mithalten kann.

5.7.2 Allgemein

Für den Vergleich werden fast ausschliesslich IntelliJ IDEA und entsprechende Plugins dafür betrachtet. Der Grund dafür ist, dass während des Projekts mit diesem Setup gearbeitet wurde. Bei Reason wird zusätzlich wenn nötig das Visual-Studio-Code-Plugin evaluiert, da dies die offizielle Empfehlung ist.

Es gibt für alle drei Sprachen viele verschiedene Editoren bzw. Editor-Plugins. Bei TypeScript ist die Auswahl so gross, dass es keine offizielle Seite mit einer Auflistung gibt. Bei Elm und Reason gibt es das: https://github.com/elm/editor-plugins und https://reasonml.github.io/docs/en/editor-plugins. Es handelt sich bei diesen meist um Community-Plugins. Im Gegensatz dazu werden für TypeScript viele kommerzielle Entwicklungsumgebungen angeboten (wie z. B. IntelliJ IDEA von Jetbrains).

5.7.3 TypeScript

Für TypeScript gibt es breite Unterstützung in allen gängigen Entwicklungsumgebungen, welche für Web-Applikationen verwendet werden. Dies gilt auch für die Produktfamilie von Jetbrains (IntelliJ IDEA, Webstorm usw.). Das Plugin namens *JavaScript and TypeScript* wird von Jetbrains selber entwickelt und kann in jeder beliebigen

Jetbrains-Entwicklungsumgebung nachinstalliert werden, falls es nicht direkt mitgeliefert wird. Bei IntelliJ IDEA in der Ultimate-Version ist es ohne weitere Schritte verfügbar.

Die Unterstützung ist dementsprechend gut und es werden alle in den Vergleichskriterien genannten Features unterstützt. Für Refactorings stehen insbesondere Renaming und Extractions (Method, Field, Variable etc.) zur Verfügung. Auch die Navigationsmöglichkeiten sind sehr umfangreich und vergleichbar mit denen von IntelliJ für andere weitverbreitete Sprachen. Auch die Navigation zu Modulen ausserhalb des eigenen Codes funktioniert sehr gut. Meist führt sie aber nur zu den Typ-Definitionen und nicht zum TypeScript-Source-Code. Letzteres ist meist gar nicht möglich, da der Source-Code in den Node-Modulen nicht mehr als TypeScript vorliegt, sondern als kompiliertes JavaScript und den zugehörigen, getrennt davon abgelegten Typ-Definitionen.

Anzumerken ist, dass viele dieser Funktionen für TypeScript aufgrund der stärkeren Typisierung besser funktionieren als für JavaScript. Gleichzeitig liegt TypeScript in dieser Hinsicht trotzdem noch hinter stärker typisierten Sprachen wie C#, Java oder Kotlin. Dies zeigt sich z. B. beim Renaming, bei dem es Fälle gibt, in denen nicht alle Vorkommnisse gefunden werden.

Positives

- Keine zusätzlichen Plugins nötig
- Unterstützung wird kommerziell weiterentwickelt
- Umfangreiche Refactoring-Möglichkeiten
- Umfangreiche Navigations-Möglichkeiten

Negatives

- TypeScript-Source-Code von importierten Modulen nicht einfach verfügbar
- Durch eine weniger starke Typisierung ist die Unterstützung nicht immer perfekt

5.7.4 Reason

Bei Reason wurde in IntelliJ das Plugin *ReasonML*[15] verwendet. Es handelt sich dabei um ein Community-Plugin. Auf der Reason-Seite wird das Plugin für VS-Code empfohlen. Aus diesem Grund wird bei Funktionen, welche vom IntelliJ-Plugin nicht unterstützt werden, jeweils die Unterstützung in VS-Code geprüft.

Das Syntax-Highlighting in IntelliJ beinhaltet nur wenige Keywords (z. B. let, module, switch, true, false), Strings und teilweise Typ-Konstruktoren (in einigen Fällen werden diese nicht markiert, z. B. bei eigenen Typ-Konstruktoren ohne Argumente). Auch werden diese Elemente je nach Ort fälschlicherweise markiert (z. B. im HTML-Tag *<option>* wird vermeintlich der Daten-Typ *option* erkannt). Dies ist in VS-Code besser umgesetzt. Dort gibt es mehr Farbkategorien, welche zur Unterscheidung von Typen, Funktionen, Modulen und JSX-Elementen helfen.

[15] https://github.com/giraud/reasonml-idea-plugin

Fehlermeldungen des Compilers werden innerhalb der IDE angezeigt (inkl. der Angabe von Zeilen- und Spaltennummern). Leider sind die Compiler-Meldungen selber so schlecht, dass sie auch in der IDE nicht viel nützen. Dennoch kann ohne einen Wechsel in die Konsole erkannt werden, dass z. B. ein Syntax-Error vorliegt. Dasselbe gilt auch für VS-Code.

Quick-Documentations gibt es keine, dafür werden nach erfolgreicher Kompilation Typ-Informationen inline angezeigt. Diese Typinformationen erweisen sich während der Entwicklung immer wieder als nützlich.

Die Auto-Completion ist selten brauchbar. Variablen- und Parameternamen werden oft nicht erkannt, dafür werden Vorschläge von Core-Modulen geliefert, welche fast nie verwendet werden. Das grösste Problem dabei ist die Inkonsistenz, die dazu führt, dass die Auto-Completion oft gar nicht mehr verwendet wird, auch wenn sie einmal ein nützliches Resultat liefern würde. Dies scheint in VS-Code besser umgesetzt zu sein, aber auch hier werden teilweise fragwürdige Resultate geliefert. Import-Unterstützung gibt es weder in IntelliJ noch in VS-Code.

Möglichkeiten, um gewisse Code-Stellen zu extrahieren, gibt es in beiden Editoren nicht. Renaming scheint dagegen zur Verfügung zu stehen, leider funktioniert es in keinem der beiden Editoren korrekt und ist somit nicht brauchbar.

Die Navigationsmöglichkeiten sind vergleichbar schlecht. Es gibt zwar Situationen, in denen Definitionen bzw. Usages gefunden werden, sobald aber Sub-Module oder andere Hürden ins Spiel kommen, sind diese Funktionen nicht mehr verfügbar. In VS-Code ist die Lage diesbezüglich noch deutlich schlechter als in IntelliJ. Zu externen Modulen kann in keinem Fall navigiert werden.

Positives

- Grundlegende Unterstützung vorhanden (Code-Highlighting, Ansätze von Auto-Completion)
- Fehlermeldungen des Compilers werden verwendet (könnte in Zukunft besser werden)

Negatives

- Keine direkte Unterstützung durch Jetbrains
- Fast alle erweiterten Features sind entweder nicht vorhanden oder funktionieren selten bis gar nicht

5.7.5 Elm

Bei Elm wird die IntelliJ-Unterstützung durch das Plugin *IntelliJ Elm*[16] bereitgestellt. Dieses Community-Plugin kann in IntelliJ ohne weiteren Aufwand aus dem offiziellen Repository installiert werden.

[16] https://github.com/klazuka/intellij-elm

Grundlegende Funktionen wie Farbschema und Fehlererkennung in der IDE werden unterstützt. Auch Fehlermeldung werden angezeigt, auch wenn diese im Vergleich zu den Meldungen des Compilers nicht sehr informativ sind. Z. B. wird ein Funktionsname, bei dem ein Zeichen vergessen ging, rot markiert und mit der Meldung *Unresolved Reference* klassifiziert. Der Compiler gibt dagegen gleich Vorschläge, was die richtige Schreibweise sein könnte. In der Quick-Documentation eines Werts wird dessen Typ (auch bei Funktionen) und Implementierungsort korrekt angezeigt. Falls vorhanden wird die Inline-Dokumentation angezeigt.

Die Auto-Completion ist rudimentär, aber nicht komplett context-insensitive. So werden bei Methoden-Signaturen nur Typ-Vorschläge gemacht und keine Variablennamen angeboten. Bei der Schreibweise *Modul.* werden sogar nur die Inhalte dieses Moduls vorgeschlagen. Wurde ein Modul, welches verwendet wird, noch nicht importiert, kann es mithilfe der IDE importiert werden. Es werden aber keine Module automatisch importiert.

Die Refactoring-Möglichkeiten sind spärlich. Es gibt nur die Extract-Option *Variable*, welche eine Expression in einen let-Block auslagert. Das Renaming funktioniert gut, auch über Modulgrenzen hinweg.

Usages werden gefunden und auch das Springen zur Implementation funktioniert. Dabei ist anzumerken, dass auch in die Definitionen von Core-Modulen (und anderen installierten Modulen) navigiert werden kann. Dort ist jeweils der gesamte Elm-Code inklusive Implementation vorhanden.

Positives

- Sehr gutes Code-Highlighting
- Navigation und Usage-Finden funktionieren gut
- Renaming über Modulgrenzen hinweg

Negatives

- Keine direkte Unterstützung durch Jetbrains
- Fehlermeldungen des Compilers werden nicht verwendet
- Keine umfangreicheren Refactorings

5.7.6 Fazit

Unterschiede TypeScript wird fast überall unterstützt, Elm und Reason nur mit entsprechenden Plugins. Bei TypeScript sind alle State-of-the-Art-Funktionen verfügbar, bei Elm etwas weniger und bei Reason bedenklich wenig.

Die folgenden Abbildungen geben einen Eindruck der visuellen Unterschiede in IntelliJ bezüglich Code-Highlighting.

Abbildung 10: Code-Highlighting von TypeScript.

Abbildung 11: Code-Highlighting von Reason.

Abbildung 12: Code-Highlighting von Elm.

Wertung In dieser Kategorie ist Reason der klare Ausreisser gegenüber den anderen beiden Kandidaten, da Reason auf fast keine Unterstützung bei der Entwicklung zählen kann. Elm und TypeScript sind nicht auf dem gleichen Stand, insbesondere aufgrund der professionell entwickelten (und auch professionell eingesetzten) Tools für TypeScript. Dennoch kann Elm in vielen Bereichen der Editorunterstützung mithalten. Damit stellt die IDE-Unterstützung von Elm im Gegensatz zu der von Reason kein Hindernis dar.

5.8 Entwicklungsgeschwindigkeit

5.8.1 Vergleichskriterien

Wie schnell etwas entwickelt werden kann, ist einer der wichtigsten Faktoren für den Entscheid einer Sprache oder eines Frameworks. Nicht nur kann ein Produkt schneller entwickelt werden, sondern es kann schneller auf Änderungen reagiert werden und geplante Änderungen können einfacher als Prototyp eruiert werden. Für diesen Vergleich wird die beanspruchte Zeit für die Entwicklung angeschaut. Diese Zeiten beinhalten unter anderem die Zeit für die Einarbeitung, die Zeit für die eigentliche Entwicklung und die Zeit für die Entwicklung der Tests. Zeitlicher Aufwand für Wartungen werden nicht berücksichtigt, da es faktisch keine solchen Aufwände innerhalb dieser Arbeit gab.

Die Datengrundlage wurde mithilfe von GitLab Time Tracking erfasst. Für jeden Entwicklungsschritt wurde für jede Entwicklungssprache ein eigenes Issue erstellt. Auf dieses Issue wurde die Zeit gebucht, welche benötigt wurde, um das jeweilige Feature zu entwickeln. Arbeiten, welche für alle drei Sprachen gleichermassen galten (beispielsweise CSS-Styling, Backend implementieren), wurden auf separate Issues gebucht oder gleichmässig auf die entsprechenden drei Tickets aufgeteilt. Die Reihenfolge der Entwicklung wurde immer wieder verändert, so dass keine Sprache dadurch benachteiligt ist.

In diesem Vergleichspunkt wird nicht auf die IDE-Unterstützung eingegangen, obwohl diese natürlich einen erheblichen Einfluss auf die Entwicklungsgeschwindigkeit haben kann, da dieser Punkt in einem eigenen Kapitel behandelt wird.

5.8.2 Gesamter Entwicklungszeitaufwand

Die folgenden Tabellen zeigen den Entwicklungsaufwand einmal kumuliert und danach aufgeschlüsselt nach Personen.

Tabelle 8: Gesamter Entwicklungszeitaufwand

Reason	45.17h
Elm	39.92h
TypeScript	33.75h

Tabelle 9: Gesamter Entwicklungszeitaufwand von Remo Dörig

Reason	20.50h
Elm	19.50h
TypeScript	17.00h

Tabelle 10: Gesamter Entwicklungszeitaufwand von Joel Fisch

Reason	24.67h
Elm	20.42h
TypeScript	16.75h

Gesamthaft hat Reason am meisten Zeit für die Entwicklung beansprucht und TypeScript am wenigsten. TypeScript hat im Vergleich zu den anderen Sprachen den Vorteil, dass Joel Fisch vor dem Start dieses Projektes bereits über weitreichende TypeScript-Erfahrung verfügte (auch mit den verwendeten Libraries). Dieselbe Rangfolge der Entwicklungszeit zeigt sich jedoch auch bei Remo Dörig, welcher vor diesem Projekt nur wenige Zeilen TypeScript geschrieben hat.

5.8.3 Vergleich signifikanter Entwicklungsschritte

Um die Unterschiede und deren Gründe in der Entwicklungszeit genauer betrachten zu können, werden in diesem Abschnitt Entwicklungsschritte genauer beleuchtet, welche signifikante Zeitunterschiede zwischen den Sprachen aufweisen.

HTTP-Call implementieren In diesem Entwicklungsschritt dauerte die Entwicklung in Reason fast doppelt so lange wie die Entwicklung in den anderen Sprachen. Bei Reason war es nicht klar, wie die Side-Effects gehandhabt werden sollten, so wurde zuerst eine Variante implementiert, welche später durch eine neue ersetzt wurde.

Tabelle 11: Entwicklungszeitaufwand HTTP-Call implementieren

Reason	9.83h
Elm	5.08h
TypeScript	5.00h

Formatierung der Response Für die Formatierung wurde eine JavaScript-Library verwendet. In Elm benötigte die Verwendung externer JavaScript-Libraries vergleichsweise viel Zeit.

Tabelle 12: Entwicklungszeitaufwand HTTP-Call implementieren

Reason	1.75h
Elm	3.33h
TypeScript	1.08h

Basis-UI und History Anhand der Entwicklung der ersten UI-Version und der History können die markanten Unterschiede, welche insbesondere durch die Erfahrung entstehen, aufgezeigt werden. Diese Features wurden durch Joel Fisch entwickelt und weisen bei Reason und Elm eine ca. doppelt so lange Entwicklungszeit auf.

Tabelle 13: Entwicklungszeitaufwand UI für Request und Response

Reason	5.33h
Elm	4h
TypeScript	2.25h

Tabelle 14: Entwicklungszeitaufwand History von Calls

Reason	6.5h
Elm	7.5h
TypeScript	3h

Testing Im Entwicklungsschritt des Testings wurde die Abdeckung der Tests überprüft und angeglichen. Zusätzlich wurden für eine bestimmte Komponente Komponententests für alle drei Sprachen erstellt. Auffallend ist dabei, dass Elm in diesem Bereich markant weniger Zeitaufwand benötigte. Dies ist insbesondere darauf zurückzuführen, dass in Elm die Standard-Library für Testing (elm-test) gute Möglichkeiten für Komponententests (auch für das Testen von Aktionen) bietet. In Reason verzögerte das Mocking von Funktionen und die schlechte Dokumentation und Unvollständigkeit der BuckleScript-Binding von den Test-Libraries die Entwicklung noch zusätzlich.

Tabelle 15: Entwicklungszeitaufwand Testing

Reason	4.33h
Elm	1.5h
TypeScript	3.5h

Change-Request Der nach dem Ende der geplanten Entwicklung durchgeführte Change-Request wird hier nicht aufgrund der grossen Zeitunterschiede aufgegriffen, sondern um die Unterschiede bei der Umsetzung von nicht geplanten Features zu zeigen. Wie der Tabelle unten zu entnehmen ist, waren die Unterschiede nicht gross. Anzumerken ist, dass bei diesem Entwicklungsschritt die gemeinsamen Vorarbeiten (Konzept und Styling) unabhängig von den untenstehenden Zeitmessungen durchgeführt wurden. Es kann somit festgehalten werden, dass die Entwicklungszeiten in bestehenden Applikationen nicht grundsätzlich stark divergieren. Davon auszuschliessen sind Features, welche mit einer der drei Varianten ausserordentlich aufwendig umzusetzen sind (z. B. Einbindung einer JavaScript-Library in Elm).

Tabelle 16: Entwicklungszeitaufwand Change-Request

Elm	1.5h
TypeScript	1.75h
Reason	1.6h

5.8.4 TypeScript

TypeScript profitiert von einer grossen Community. Für die meisten Probleme, auf die man stösst, findet sich schnell eine Antwort im Internet. Der Compiler/Transpiler ist reif und gibt entsprechend gute Rückmeldung. Über die meist verwendeten Konzepte existieren gute Erläuterungen sowie Tutorials mit Beispielen. Für Probleme gibt es meist mehrere bekannte Lösungen und oft ist es der Fall, dass eine Lösung von der Community deutlich bevorzugt wird. Manchmal fällt diese Entscheidung aber auch schwer.

Die Lösung in TypeScript verwendet eine Vielzahl an externen Libraries. Die Dokumentation dieser Libraries ist meistens nicht auf dem gleich guten Stand wie die der Sprache und der Haupt-Libraries (z. B. React). Die verwendeten Libraries benötigen jeweils eine gewisse Einarbeitungszeit.

Bei der Entwicklung können schnell Sachen ausprobiert werden, Feedback über Veränderungen sind in der laufenden Applikation schnell ersichtlich. Dafür treten vergleichsweise viele Fehler erst bei der Ausführung auf. Die Fehlermeldungen sind meist hilfreich und geben Aufschluss über die Ursache des Problems. Es muss relativ viel “Boilerplate”-Code geschrieben werden (Code ohne eigene Logik, welcher z. B. generiert werden könnte).

Um in einem funktionalen Stil zu programmieren, bietet TypeScript verhältnismässig wenig Unterstützung. Die Hauptpunkte sind hierbei mangelhafte Immutability, nur sehr verbose Varianten von algebraischen Datentypen, mässig gute Typ-Inferenz und frühe Evaluierung von generischen Typinformationen. Der letzte Punkt führt dazu, dass oftmals zusätzliche Typinformationen angegeben werden müssen, wo in den anderen Sprachen der Compiler die Typen ohne Probleme herausfinden kann.

Positives

- Grosse Community
- Gute Dokumentation
- Schnelles Feedback bei Veränderungen

Negatives

- Viel “Boilerplate”-Code
- Laufzeitfehler können aufwendig zu finden sein
- Mangelhafte Unterstützung von funktionalen Elementen

5.8.5 Reason

Reason ist eine relativ junge Sprache [7]. Die Community ist noch nicht so gross. Für viele Probleme findet man die Antwort nicht direkt online oder man findet eine Lösung für OCaml, welche man für Reason anpassen muss. Erläuterungen zu den verwendeten Konzepten gibt es nicht viele. Die Dokumentation ist vergleichsweise klein.

Viel Zeit wird für die Suche online gebraucht und für die undeutlichen Fehlermeldungen des Compilers. Der Compiler erkennt viele Fehler bereits beim Kompilieren. Oft führen die Fehlermeldungen des Compilers den Programmierer jedoch in eine falsche Richtung.

Es existieren momentan noch wenige Libraries, welche direkt verwendet werden können, dafür ist aber die Interaktion mit JavaScript gut.

Die Typ-Inferenz ist weitestgehend gut, lediglich bei der Verwendung von Records mit überlappenden Feldnamen müssen zusätzliche Typinformationen angegeben werden. Auch Immutability, algebraische Datentypen (inklusive Pattern-Matching) und Infix-Operatoren werden von Reason in einem guten Umfang unterstützt.

Positives

- Viele Fehler werden beim Kompilieren bereits erkannt
- Gute Unterstützung von funktionalen Elementen

Negatives

- Dokumentation könnte wesentlich besser sein
- Kleine Community
- Schlechte Fehlermeldungen des Compilers

5.8.6 Elm

Elm wird zwar nicht so viel verwendet wie TypeScript, jedoch verfügt Elm auch über eine gute Community. Zu vielen Problemen findet man online eine Lösung. Es existieren relativ viele Libraries mit angemessener bis sehr guter Dokumentation. Meistens existiert eine Library pro Thema und nicht mehrere, was die Entscheidung vereinfacht.

Der Compiler erkennt viele Fehler bereits beim Kompilieren und gibt hilfreiche Fehlermeldungen. Es muss wenig “Boilerplate”-Code geschrieben werden.

Online werden viele Konzepte gut erläutert. Für Personen, welche von der JavaScript-Welt kommen, benötigt der Umstieg auf Elm Zeit. Es gibt hilfreiche Tutorials, welche den Einstieg in die Sprache vereinfachen. Die Dokumentation ist weitestgehend gut.

Die Typ-Inferenz von Elm ist sehr gut, sodass keine Typinformationen zwingend notwendig sind. Algebraische Datentypen, Pattern-Matching und Infix-Operatoren werden auch unterstützt. Immutability ist bei Elm auf dem höchst möglichen Grad, da innerhalb der Sprache Mutationen auf keine Weise möglich sind.

Positives

- Aktive Community
- Gute Dokumentation
- Hilfreiche Fehlermeldungen
- Wenig “Boilerplate”-Code
- Gute Unterstützung von funktionalen Elementen

Negatives

- Viel Einarbeitungszeit für die meisten JavaScript-Programmierer

5.8.7 Fazit

Unterschiede Die Sprachen unterscheiden sich stark in der Grösse und Aktivität der Community. Dies führt zu einem Unterschied bei der Dokumentation, Libraries und Suchresultaten auf Probleme.

TypeScript braucht für die verwendete Struktur am meisten externe Libraries, in welche sich ein Programmierer zuerst einarbeiten muss. Ein JavaScript-Entwickler findet in TypeScript am meisten Ähnlichkeiten und kommt schnell in der Umgebung zurecht. Gleichzeitig findet potenziell weniger schnell ein Umdenken zur Elm-Architektur und funktionalen Prinzipien statt, da in TypeScript wie bis anhin entwickelt werden kann.

Für TypeScript gibt es am meisten externe Libraries, Reason kann diese Libraries auch relativ einfach benutzen. Für Elm gibt es weniger Libraries, dafür meist nur eine etablierte Variante. Die Dokumentation dieser Libraries ist oft gut und die Konzepte der Library werden gut erklärt. Die Herangehensweise von Reason und TypeScript ähneln sich in vielen Bereichen.

Unterschiede der Sprache selber ohne Community und Dokumentation Reason und Elm erkennen frühzeitig Fehler, wo hingegen TypeScript es dem Programmierer schneller erlaubt, unfertigen Code auszutesten. Die Fehlermeldungen des Reason-Compilers sind nicht sehr hilfreich. TypeScript benötigt am meisten “Boilerplate”-Code, insbesondere wegen der fehlenden Unterstützung von funktionalen Prinzipien.

Wertung Die Entwicklung hat bei Reason am längsten gedauert. Die Dokumentation ist noch am wenigsten ausgereift und die Fehlermeldungen des Compilers sind nicht zielführend. Diese Makel können aber in Zukunft ausgebessert werden. Reason nur als Sprache betrachtet bietet dagegen einige Dinge wie z. B., dass JavaScript-Libraries einfach eingebunden werden können und wenig Code benötigt wird, welcher keinen direkten Nutzen hat.

Elm benötigt für die meisten Entwickler, welche aus der JavaScript-Welt kommen, am meisten Einarbeitungszeit. Die Entwicklung in Elm hat im Falle dieser Arbeit etwas länger gedauert als in TypeScript. Wenn der Entwickler eingearbeitet ist und keine externen JavaScript-Libraries benötigt werden, ist die Entwicklung in Elm etwa gleich schnell oder sogar ein wenig schneller als in TypeScript.

Im Rahmen dieses Projektes waren externe Faktoren der Sprache (Community, Dokumentation, Compilerunterstützung, existierende Libraries) für die Entwicklungszeit signifikanter als die Faktoren der Sprache selber.

5.9 Arbeit in grossen Teams

5.9.1 Vergleichskriterien

Um grosse Applikationen mit einem hohe Business-Wert zu entwickeln, müssen zwangsläufig viele Entwickler an der gleichen Code-Base entwickeln. Dies kann durch Features vereinfacht werden, welche dafür sorgen, dass die einzelnen Teammitglieder möglichst unabhängig voneinander entwickeln können.

Deshalb werden in diesem Bereich folgende Punkte verglichen:

- Splitting: Kann die Applikation einfach in mehrere Sub-Applikationen aufgeteilt werden? Wie können diese Sub-Applikationen miteinander kommunizieren?
- Independent Compilation: Können Code-Stücke unabhängig voneinander (abgesehen von Typ-Definitionen) kompiliert werden?
- Information-Hiding: Ist es möglich, komplexe Datenstrukturen in einem Modul oder einer Klasse zu verstecken, sodass diese Datenstrukturen intern verändert werden können, ohne dass dies zu Breaking-Changes an den Verwendungsorten führt?

Anzumerken ist, dass innerhalb der Entwicklung in diesem Projekt nur mit zwei Personen gearbeitet werden konnte, was bedeutet, dass die folgenden Aussagen zum grössten Teil nicht auf den Erfahrungen dieses Projekts basieren. Es wird stattdessen davon ausgegangen, dass die obengenannten Kriterien einen positiven Einfluss auf die Arbeit in grossen Teams hat und sich somit die Sprache und deren Ökosystem besser für Entwicklungen in solchen Teams eignen.

5.9.2 Allgemein

Zwei Kriterien, welche eine verbesserte Entwicklungssituation für grosse Teams bereiten können, welche sich nicht auf der oberen Liste befinden, sind automatische Code-Formatierung und dass eine Sprache Merge-Konflikte durch deren Aufbau vermindert. Da diese Punkte für alle drei Varianten gleich sind, werden sie hier gemeinsam behandelt.

Automatische Code-Formatierung führt dazu, dass der Code von allen Personen gleich formatiert ist. Dadurch werden Konflikte (sowohl auf Ebene des persönlichen Geschmacks als auch beim Mergen) vermieden. Solche Tools existieren für alle drei Sprachen und funktionieren auch bei allen sehr gut. Der Hauptunterschied liegt

dort vor allem in der Aggressivität, in der umformatiert wird. Während der *elm-format* hauptsächlich Einrückungen einheitlich gestaltet, wandelt *refmt* von Reason Code-Konstrukte um, um noch mehr Einheitlichkeit zu erreichen.

Während des Projektes wurden einige grössere Veränderungen durchgeführt, welche parallel die gleichen Code-Bereiche betrafen. Die dadurch erzeugten Merge-Konflikte waren vergleichbar gross und auch entsprechend ähnlich aufwendig zu lösen. Keine der drei Varianten verfolgt an dieser Stelle ein Konzept, um dieses Problem konkret zu adressieren.

5.9.3 TypeScript

Splitting React-Komponenten können sehr modular verwendet werden und erlauben es so, dass jede React-Komponente als unabhängige Sub-Applikation mit eigenem State verwendet werden kann. Redux bietet selber keine solche Möglichkeit, jedoch können mehrere Redux-Instanzen auf der gleichen Seite eingesetzt werden. Somit ist es möglich, auf einem höheren Level eine Root-Applikationskomponente zu haben, welche die Sub-Applikationen anzeigt und gleichzeitig als Message-Broker zwischen den Sub-Applikationen fungiert. Ein weiterer Ansatz wäre die Verwendung von Web-Components. Jede React-Applikation kann als Web-Component gerendert werden, welche von beliebigen anderen Applikationen verwendet werden kann. Die Kommunikation läuft dann auf DOM-Ebene ab, was bedeutet, dass Nachrichten über DOM-Events ausgetauscht werden.

Independent Compilation TypeScript erlaubt die Erstellung von TypeScript-Definition-Dateien (*.d.ts*), welche nur die Typ-Definitionen beinhalten. Dazu wird der zu JavaScript kompilierte (und meist schon minifizierte) Source-Code mitgeliefert. Somit muss das so vorkompilierte Modul nicht mehr neu umgewandelt werden, sondern nur noch der eigene Code. Mit dieser Technik können sehr grosse Projekte in verschiedene Module unterteilt werden, welche unabhängig voneinander kompiliert werden können.

TypeScript-Module, welche von NPM verwendet werden, setzen zum grössten Teil dieses Pattern ein.

Information-Hiding TypeScript kann in Klassen Membervariablen als privat deklarieren und damit stellt der Compiler sicher, dass keine Zugriffe von aussen darauf gemacht werden. Es gibt aber keine Möglichkeit, um dies ohne Klassen zu erreichen. Klassen sind aber verboser in der Verwendung als die in einem funktionalen Stil verwendeten Objekt-Literale. Ausserdem ist es nicht möglich, Member als Module-private o. ä. zu deklarieren, was dazu führt, dass alle Funktionen, welche auf den entsprechenden Daten operieren sollen, als Methoden innerhalb der Klasse definiert sein müssen. Diese Methoden können nun nicht mehr mit den typischen Function-Composition-Patterns verwendet werden. Sollen somit private Datenstrukturen verwendet werden, kann dies nur mit der Einführung einer anderen Kategorie von syntaktischen Elementen erreicht werden.

Positives

- Viel Flexibilität bei Code-Splitting (React-intern als auch mit Web-Components)
- Solide, weit verbreitete Independent Compilation
- Verstecken von einzelnen Membervariablen ist möglich

Negatives

- Bruch mit dem funktionalen Stil bei der Verwendung von privaten Daten

5.9.4 Reason

Splitting Da auch bei Reason die Basis React ist, bieten sich hier die gleichen Möglichkeiten wie bei TypeScript.

Independent Compilation BuckleScript unterstützt keine Independent Compilation für Reason als solches. Allerdings kann BuckleScript vorkompilierte Dateien verwenden, um den Build-Prozess zu beschleunigen, was schliesslich zu einem ähnlichen Resultat führt. Der Source-Code muss aber in jedem Fall vorhanden sein.

Information-Hiding In Reason werden Typen innerhalb von Modulen definiert und werden normalerweise auch immer exportiert. Mit sogenannten Module-Interfaces kann definiert werden, was aus diesem Modul exportiert werden soll. Damit ist es möglich, nur gewisse Funktionen und Typ-Definitionen zu exportieren. Beim Exportieren von Typ-Definitionen kann darüber entschieden werden, ob die Definition einschliesslich der inneren Werte exportiert werden soll oder nur der Typ als Black-Box. Soll der Typ mitsamt allen inneren Informationen exportiert werden, so muss die ganze Typ-Definition wiederholt werden.

Modul-Namenskollisionen Dies ist ein Punkt, welcher nur Reason betrifft. In Reason ist jede Datei, egal wo es abgelegt ist, ein Modul mit dem gleichen Namen wie die Datei. Das führt dazu, dass auf keiner Stufe der Ordnerstruktur eine Datei mit dem gleichen Namen existieren darf. Das kann in grossen Teams wiederum leicht zu Namenskonflikten führen.

Positives

- Viel Flexibilität bei Code-Splitting (React-intern als auch mit Web-Components)
- Beliebige Typen können Modul-intern gehalten werden
- Wiederverwendung von vorkompilierten Source-Dateien

Negatives

- Keine Independent Compilation
- Keine Strukturierung für Module

5.9.5 Elm

Splitting Da in Elm jede Applikation nur auf dem Root-Level einen State haben kann, können Elm-Applikationen nicht ohne weiteres ineinander verschachtelt werden. Auch eine Elm-Applikation kann mit Web-Components verwendet werden und so können sowohl Elm-Applikationen als auch beliebige andere Applikationen ineinander verschachtelt werden. Wie bei React wird die Kommunikation auf dieser Ebene über das DOM abgehandelt.

Independent Compilation Da eine Elm-Datei nicht direkt in JavaScript übersetzt werden kann, sondern nur inklusive der Elm-Runtime und allen anderen nötigen Elm-Dateien, ist eine Independent Compilation als solches nicht möglich. Am nächsten bei Independent Compilation liegen bei Elm die vorhergenannten Code-Splitting-Ansätze.

Information-Hiding Dies ist bei Elm mit eigenen Datentypen möglich. Bei den Export-Angaben eines Moduls kann entschieden werden, ob nur der Typ oder auch seine Konstruktoren exportiert werden sollen. Werden keine Konstruktoren exportiert, so ist es für aufrufende Module nicht möglich, den Inhalt des Typs zu sehen. Dies ist mit einem Typ-Alias nicht möglich, weswegen Typ-Alias dazu in einen eigenen Typ mit nur einem Konstruktor verschachtelt werden, um einen direkten Zugriff darauf zu unterbinden.

Positives

- Web-Components (inklusive Custom-Events und Attributen) werden unterstützt
- Eigene Typen können Modul-intern gehalten werden

Negatives

- Elm-Applikationen lassen sich nicht direkt verschachteln
- Keine Independent Compilation
- Type-Alias kann nicht direkt als Modul-intern deklariert werden

5.9.6 Fazit

Unterschiede Die markantesten Unterschiede beim Code-Splitting ist die Tatsache, dass bei Elm nur die gesamte Applikation einen State haben kann, wo dagegen bei React jede Komponente stateful sein kann. Dies führt dazu, dass bei Elm immer ganze eigenständige Elm-Applikationen verschachtelt werden müssen, bei React dies aber sowohl auf der Stufe von Komponenten als auch für ganze Applikationen möglich ist.

Independent Compilation ist ein Feature, welches so nur von TypeScript unterstützt wird. Dafür hat TypeScript keine Unterstützung von Modul-Level Information-Hiding, welches sowohl von Elm als auch von Reason unterstützt wird. Der Grund dafür liegt wohl in den Wurzeln, welche bei TypeScript in der objekt-orientierten Welt liegen.

Wertung Die Nützlichkeit der aufgeführten Features ist sehr projektabhängig, wie z. B. ob die verschiedenen Applikationsteile viel gemeinsamen State haben oder nicht. Deshalb ist es nicht möglich, eine Sprache hier als Gewinner dieser Kategorie zu definieren. Bemerkenswert ist allerdings, dass in keiner Sprache alle Bereiche vollständig unterstützt werden.

5.10 Wartbarkeit

5.10.1 Vergleichskriterien

Nach der initialen Entwicklung eines Produktes muss dieses gewartet werden. Einige Produkte werden immer weiterentwickelt, andere müssen sich nur an die verändernde Welt anpassen. Für diesen Vergleichspunkt wird nur die Wartung ohne Weiterentwicklung des Produkts betrachtet.

Die Wartung soll möglichst wenig Zeit und Aufwand in Anspruch nehmen. Für diesen Vergleich ist insbesondere wichtig:

- Wie rückwärtskompatibel ist der Compiler / sind die Libraries?
- Ist das Packagingsystem stabil und sicher?
- Wie einfach verständlich ist entwickelter Code für bestehende oder neue Entwickler?
- Haben die Libraries / der Compiler eine stabile Zukunft?

Frontends werden in der Regel schneller ausgewechselt als Backends oder Datenbanken. In diesem Vergleich wurde die Wartbarkeit auf etwa 5 Jahre angeschaut. Daher wird z. B. der Aspekt ignoriert, ob sich noch eine Maschine zum Kompilieren findet.

5.10.2 Allgemeines

Die Zeitdauer dieses Projektes genügt nicht, um anhand von Erfahrungswerten eine klare Aussage zu machen. Die Erkenntnisse bezüglich Wartbarkeit kommen hauptsächlich aus den jahrelangen Erfahrungen der Entwickler und Recherchen. Der Bewertungspunkt, wie schnell ein Entwickler den Code wieder versteht, ist sehr subjektiv. Er ist aber so wichtig, dass er trotzdem aufgeführt wird. Für die Wartbarkeit ist die zukünftige Entwicklung der Sprache wichtig, diese kann aber nur anhand von Indizien prognostiziert werden.

Das Package-System basiert bei allen untersuchten Varianten zumindest teils auf *NPM*, bei welchem allgemeine Sicherheitsrisiken bekannt sind.

5.10.3 TypeScript

Rückwärtsinkompatibilitäten gibt es bei TypeScript nicht viele, einige entstehen durch die Nähe an JavaScript. Liste der Rückwärtsinkompatibilitäten:
https://github.com/microsoft/TypeScript/wiki/Breaking-Changes

Die in diesem Projekt entwickelte Applikation verwendet relativ viele Libraries (25). Diese Libraries entwickeln

sich rasanter weiter als die Sprache selber und es muss für die Zukunft mehr angepasst werden. Auch die grossen Libraries wie React haben immer wieder neue Rückwärtsinkompatibilitäten.

Subjektiv betrachtet braucht TypeScript am meisten Zeit, um sich neu einzuarbeiten, einerseits wegen den Libraries, andererseits ist es am einfachsten, unbewusst Fehler zu machen. Es gibt dafür aber auch mehr TypeScript-Entwickler auf dem Markt als Reason- oder Elm-Entwickler.

Durch die grosse Community ist eine Weiterführung der Sprache und der wichtigsten Libraries fast sicher.

Positives

- Sprache alleine hat wenig Rückwärtsinkompatibilitäten
- Sichere Zukunft

Negatives

- Viele Libraries verwendete
- Viel Einarbeitungszeit

5.10.4 Reason

Die Sprache OCaml, auf welcher Reason basiert, ist vergleichsweise alt (1996) und erprobt. OCaml ist eine Implementation von Caml, welches seit 1987 existiert. OCaml ist stabil und hat sehr wenige Rückwärtsinkompatibilitäten. An der Sprache selber sind daher keine grossen Änderungen zu erwarten, obwohl Reason noch sehr jung ist.

Bezüglich der Interaktionen mit JavaScript und generell den BuckleScript-Bindings werden aber Rückwärtsinkompatibilitäten erwartet. Viele der Standard-Libraries werden wahrscheinlich auch noch ändern, einige müssen zuerst noch geschrieben werden. Das ganze Library-Umfeld ist noch in keinem stabilen Zustand.

Subjektiv gesehen lassen sich die meisten Sachen von Reason schnell erlenen. Dadurch, dass Reason vieles frei lässt, besteht die Gefahr, dass ein Wildwuchs im Code entsteht. Damit ist gemeint, dass an den einen Stellen die Probleme auf eine andere Art gelöst werden, als dies an einer anderen Stelle der Fall ist. Die Typsicherheit gibt gewisse Sicherheiten bei nachträglichen Änderungen. Auf dem Markt gibt es aber noch nicht so viele Reason-Entwickler.

Dadurch, dass die Sprache noch jung ist und vergleichsweise wenig in Gebrauch ist, ist die Zukunft ungewiss.

Positives

- Grundsprache vergleichsweise stabil
- Starke Typsicherheit

Negatives

- Reason ist noch zu jung, um gute Zukunftsprognosen bilden zu können

5.10.5 Elm

Elm hat keinen zeitlich definierten Plan für die Weiterentwicklung und somit ist nicht klar, wann Elm die Version 1.0 erreicht [8]. In neuen Elm-Versionen werden immer wieder Rückwärtsinkompatibilitäten eingeführt, welche auch nicht immer klar kommuniziert werden; dies verärgert Teile der Community [9]. Trotz der kurzen Zeit dieser Studie wurde dies sogar ein wenig spürbar. Native-Module wurden in der Version 0.19 abgeschafft, Dokumentation, Blog-Einträge und dergleichen existierten dazu aber noch.

In der Version 0.19 gab es Rückwärtsinkompatibilitäten auf fast jedem Level (Konfiguration, Tooling, Libraries, Sprach-Konstrukte, Modularisierung). Rückwärtsinkompatibilitäten werden in den offiziellen Release-Notes nicht komplett adressiert.

Die Applikation in Elm brauchte primär Libraries von Elm selber. Die Libraries von Elm haben eine gute Chance gleich lange zu leben wie die Sprache selber.

Die Typsicherheit von Elm hilft bei nachträglichen Änderungen. Elm-Entwickler gibt es momentan nicht viele auf dem Markt, die Sprache ist aber relativ schnell erlernt.

Bei der Weiterführung der Sprache stellt sich bei Elm die Frage, ob es einen Fork geben wird, der die Sprache spaltet. Dies wäre aufgrund der engen Führung der Sprach-Core-Entwicklung denkbar, welche die Zukunft der Sprache sehr direkt steuert.

Elm hat bei den Package-Systemen zwei Vorteile gegenüber den anderen Sprachen. Dadurch, dass Side-Effects explizit eingeführt werden, ist es einer Library nicht einfach möglich, externe Systeme aufzurufen. Dies verhindert einige Sicherheitslücken, wie zum Beispiel, dass eine Library heimlich Informationen sammelt und an Dritte weiterleitet. Semantische Versionierung wird bei den Packages auf Typ-Level durchgesetzt. Ein Patch-Update kann daher keine Änderungen an den Typ-Signaturen durchführen. Dies gibt Patch-Updates die Sicherheit, dass sie den Build einer bestehenden Applikation nicht fehlschlagen lassen können.

Positives

- Kurze Einarbeitungszeit
- Starke Typsicherheit

Negatives

- Sprache hat noch keine stabile Version
- Ungewisse Zukunft aufgrund des Core-Teams

5.10.6 Fazit

Unterschiede Reason als jüngste Sprache basiert auf der ältesten der genannten Sprachen (sofern Caml miteinrechnet wird, ansonsten ist JavaScript älter als OCaml). TypeScript ist am direktesten von der Zukunft von JavaScript abhängig. Elm hat für sein Alter noch relativ viele Rückwärtsinkompatibilitäten, dies könnte sich aber mit der ersten stabilen Version ändern.

Elm und Reason bieten mit der strikteren Typisierung mehr Sicherheit beim Verändern des Codes, dafür ist bei beiden Sprachen die Zukunft nicht so gewiss wie bei TypeScript. Die TypeScript-Applikation dieses Projekts braucht subjektiv gesehen am meisten Einarbeitungszeit, dafür lassen sich wohl am einfachsten Entwickler dafür finden.

Wertung TypeScript hat die sicherste Zukunft und am wenigsten zu erwartende Rückwärtsinkompatibilitäten bei der Sprache selber. Die verwendeten Libraries bei TypeScript sind aber im schnellen Wandel und Code-Änderungen können mit am wenigsten Sicherheit durchgeführt werden.

Das Projekt Reason ist noch zu jung, um es für ein langlebiges Projekt einzusetzen. Elm ist von Rückwärtsinkompatibilitäten und Unruhen in der Community geplagt, aber Entwickler können die Sprache schnell erlernen und finden sich in neuen Projekten schnell zurecht. Code-Änderungen können mit guten Sicherheiten vollzogen werden.

Daher fällt in dieser Kategorie eine Wertung sehr schwer. Für Projekte, welche eine Lebensdauer von über 2 Jahren haben, ist TypeScript aufgrund des tiefsten Risikos zu empfehlen. Für kurzlebigere Projekte kann Elm gebraucht werden, da es in Punkten Code-Änderungen etwas besser ist wie TypeScript.

Im Bereich Wartbarkeit hat Reason durch die ungewisse Zukunft und aktuelle Instabilität verloren.

5.11 Subjektiver Vergleich

Im folgenden Abschnitt legen die Autoren ihre persönliche Bewertung dar, in der sie die für sie wichtigsten Argumente erläutern. Zusätzlich wird im Hintergrund beschrieben, welche persönlichen Erfahrungen zu dieser Bewertung führen.

5.11.1 Joel

Hintergrund Ich arbeite seit ungefähr einem Jahr intensiv mit React, Redux und TypeScript. In dieser Zeit habe ich den Wert der Elm-Architektur für die Anwendung von funktionalen Prinzipien sehr schätzen gelernt. Insbesondere der Bereich der Testability durch die konsequente Anwendung von Pure-Functions eröffnete mir neue Horizonte, sodass sich automatisierte Tests plötzlich nicht mehr aufgezwängt, sondern natürlich anfühlten. Auch Test-Driven-Development wurde durch diese Architektur zu einer (zumindest teilweise) sehr gut umzusetzenden Praxis.

Durch die Adaption eines funktionaleren Stils wurden die Limitierungen von TypeScript aber immer sichtbarer. Es gibt im Arbeitsalltag immer wieder Situationen, in denen TypeScript nicht mehr herausfinden kann, welcher Typ an einer Stelle erwartet werden müsste, obwohl dies noch eindeutig bestimmbar wäre (Elm und Reason sind dazu in der Lage). Das führt dazu, dass viele Typ-Annotationen selber geschrieben werden müssen, obwohl dies nicht nötig wäre. Der grösste Negativpunkt dabei ist aber, dass TypeScript diese Typ-Annotation nicht einmal überprüfen kann und man somit darauf angewiesen ist, dass der Typ tatsächlich dem entsprechende Wert entspricht. Dies führt zusammen mit der Tatsache, dass TypeScript durch die enge Interoperabilität mit JavaScript zur Laufzeit nur sehr bedingte Typsicherheit geben kann, dazu, dass das Vertrauen in das Typsystem von TypeScript sinkt. Dies lässt wiederum die Vorteile, welche sich aus einer Typisierung ergeben, schwächer werden.

Aus diesem Grund bin ich auf der Suche nach Möglichkeiten, diese Schwächen von TypeScript auszubessern, sei dies mit Mitteln innerhalb des TypeScript-Ökosystems oder mit anderen Sprachen, wie dies in dieser Studie der Fall ist.

TypeScript Wie bereits im Hintergrund dargelegt, hat TypeScript in meiner Praxiserfahrungen einige markante Schwächen. Gleichzeitig bietet dieser Technologie-Stack die Möglichkeit, funktionale Prinzipien anzuwenden, ohne dafür in eine exotische Sprache wechseln zu müssen, welche bei Kundenprojekten aufgrund von Bekanntheit und Verbreitung niemals in Frage käme. Was die tägliche Arbeit erleichtern würde, was aber für mich nicht sehr grosses Gewicht hat, sind die fehlenden funktionalen Konstrukte wie (leicht zu verwendende) ADTs oder Pattern-Matching. Dies könnte sich aber aufgrund der JavaScript-Welt in den nächsten Jahren ändern [10].

Positives

- Grosses Ökosystem
- Weite Verbreitung, trotz funktionaler Architektur

Negatives

- Unsicherheit über Laufzeittypen
- Vergleichsweise schlechte Typ-Inferenz
- Fehlende funktionale Tools wie ADTs, Pattern-Matching, Infix-Operatoren

Reason Reason löst viele Probleme des TypeScript-Technologie-Stacks. So ist die Typsicherheit zur Laufzeit viel besser und die Hürde, um ungültige Typen aus JavaScript zu übernehmen ist um einiges höher. Dies geht Hand-in-Hand mit einer guten Typ-Inferenz, welche alle Typen bis auf überlappende Record-Typen herausfinden kann. Falls sie dies an einer Stelle nicht kann, muss der Programmierer zwingend eine Entscheidung treffen, ist aber dabei nie ganz auf sich gestellt, was bedeutet, dass sich auch an der Stelle keine falschen Typen einschleichen können. Durch die an JavaScript angepasste Syntax und die Unterstützung von JSX und React ist der Sprung zu Reason mit ReasonReact von TypeScript mit React nicht sehr gross.

Die Stelle, an der Reason aus meiner Sicht leider in die Unbrauchbarkeit abdriftet, ist das Tooling. Der Compiler ist zwar unglaublich schnell, aber dafür liefert er oft fast unbrauchbare Fehlermeldungen. Dazu ist die IDE-Unterstützung so schlecht, dass vielfach genauso gut in einem einfachen Texteditor gearbeitet werden könnte. Auch weitere Tooling-Aspekte sind mager, wie die Integration von Code-Coverage-Messungen gezeigt hat.

Positives

- Viel Ähnlichkeit mit JavaScript/TypeScript
- Unterstützung von vielen funktionalen Konstrukten
- Gute Typ-Inferenz
- React als weit verbreitete UI-Library verwendbar

Negatives

- Schlechte Fehlermeldungen des Compilers
- Schlechte IDE-Unterstützung
- JSX ist syntaktisch verboser als in TypeScript

Elm Elm kommt von einer ganz anderen Sichtweise her. Diese Sprache und ihre Architektur zeigen, wie eine Web-Applikation komplett *purely functional* umgesetzt werden kann. Aufgrund der Wurzeln in Haskell sind viele Dinge wie Typ-Inferenz und die Unterstützung von funktionalen Konstrukten sehr gut. Gleichzeitig fehlen mir persönlich einige “Goodies” von Haskell wie das direkte Pattern-Matching auf Funktionsargumenten. Für den Einsatz von Elm in einem Business-Kontext finde ich die sehr eng geführte Sprachentwicklung problematisch, da es dadurch gut denkbar ist, dass ein missionskritisches Feature niemals integriert werden kann. Gerade für sehr grosse, langlebige Applikationen ist dies für mich ein schwerwiegender Punkt.

Positives

- Vollständig reine Sprache
- Unterstützung von vielen funktionalen Konstrukten
- Sehr gute Typ-Inferenz
- Viele gute Informationen (in Dokumentationen, bei Compilermeldungen)

Negatives

- Eng geführte Sprachentwicklung
- Fehlende Goodies im Vergleich zu Haskell

Fazit Wenn ich zum aktuellen Zeitpunkt ein grosses, langlebiges Projekt starten müsste, würde ich trotz meiner Vorbehalte TypeScript wählen. Wenn es sich dagegen um eine kleinere Arbeit handeln würde, käme Elm für mich als reale Option in Frage. Ich würde grundsätzlich jedem React + Redux + TypeScript-Entwickler empfehlen, zumindest eine kleine Applikation mit Elm zu implementieren, da sich die Sinne für die Elm-Architektur dadurch schärfen werden. Viele dieser Erfahrungen werden dann auch in TypeScript übernommen werden können.

Reason ist für mich der ambivalenteste Kandidat. Einerseits wäre diese Sprache grundsätzlich eine hervorragende Option, um TypeScript ohne grundlegende Brüche abzulösen. Gleichzeitig ist das Ökosystem und somit auch das Tooling noch weit davon entfernt, dass ich das Gesamtpaket in einem produktiven Projekt einsetzen würde. Bei dieser Sprache habe ich die grösste Hoffnung, dass sie in den nächsten Jahren deutlich besser werden wird und vielleicht sogar besser für Business-Anwendungen als Elm und TypeScript.

5.11.2 Remo

Hintergrund Ich arbeite primär im Backend-Bereich mit PHP. Für die Frontendentwicklung nehme ich (durch den Arbeitgeber bedingt) im Normalfall nur JavaScript ohne grosse Strukturierungen. Bei der Arbeit entwickeln wir ein Produkt weiter, welches 1998 gestartet wurde. Es gibt viele Altlasten und ich musste schmerzlich die Erfahrungen machen, was es bedeutet, wenn ohne Struktur ein grosses Projekt langlebig durchgeführt wird. Mittlerweile hat sich aber einiges geändert und alle neuen Sachen sind primär objektorientiert durchgesetzt und es gibt ein halbwegs gutes Modularisierungssystem.

Fast der ganze Technologie-Stack dieser Studie war für mich zuvor grösstenteils unbekanntes Gebiet. TypeScript und React hatte ich in der Schule mal kurz. Redux war mir ein Begriff, ich weiss aber nicht mehr, ob ich das mal in der Schule hatte oder nicht. Von Reason hatte ich zuvor noch nichts gehört. Elm habe ich mir aus Interesse mal angeschaut, aber noch nie etwas darin programmiert. Was mir bereits vertraut war, war PHP für das Backend und GitLab für das Hosting eines Repositories. Die GitLab CI/CD-Fähigkeiten lernte ich aber erst in diesem Projekt richtig kennen.

Auf die funktionale Programmierwelt bin ich früh in der Lehre aufmerksam geworden. Ich habe mich schon früh für Programmiersprachen interessiert und habe darum Haskell gelernt. Mir gefielen die Konzepte von Haskell sehr. Die Vorteile einer reinen Funktion finde ich super, es erstaunt mich gewissermassen immer noch, dass es bei den meisten Mainstream-Sprachen keine Möglichkeit gibt, eine Funktion als rein zu definieren. Ich merkte, dass sich die Mainstream-Sprachen ein Teil in Richtung funktionale Programmierung weiterentwickelten. Beispielsweise bekam PHP Closures, Java bekam Optional und Methoden wie *map*.

Die funktionale Welt funktioniert für mich gut im Kleinen, ich weiss jedoch nicht, wie gut mittlere bis grössere Projekte konzeptioniert werden können. Momentan finde ich, können Strukturierungen im Grossen über Objektorientierung besser gelöst werden, als über funktionale Programmierung. Diese Studie ist daher interessant für mich, da ich ein reales Projekt auch über funktionale Konstrukte strukturieren kann. Des Weiteren lerne ich natürlich durch all die neuen Technologien viel dazu.

TypeScript Das grosse Ökosystem und die Nähe zu JavaScript sind für mich die grössten Vorteile von TypeScript. Die Nähe zu JavaScript finde ich super, so kann man TypeScript einfach partiell in bestehende Projekte einführen und das ganze Ökosystem von JavaScript kann einfach benutzt werden. Die Nähe zu JavaScript ist aber auch ein Nachteil, da mir JavaScript als Sprache nicht gefällt. Das Prototype basierte System gefällt mir nicht, die Standard-Library ist nicht konsistent.

Im Rahmen dieser Architektur mit React + Redux arbeitet man oft um die Sprachlimitationen herum, daher verwendeten wir relativ viele Libraries. Diese Libraries musste ich zuerst kennenlernen. Durch die Libraries wird die Core-Logik verschleiert, die Libraries kann man zwar halbwegs schnell anwenden, dafür weiss man nicht mehr, was im Hintergrund passiert. Ich bin der Meinung, man hat etwas erst wirklich verstanden, wenn man es selber programmieren könnte.

Funktionale Konstrukte wären hier sicherlich wünschenswert. Hinzu kommt, dass der objektorientierte Teil von TypeScript nicht wirklich gut ist. Im Vergleich zu JavaScript finde ich TypeScript wesentlich besser. Aufgrund dieser Studie werde ich einen Vorschlag bei einem Arbeitgeber machen, dass wir TypeScript einführen. Wenn ich wetten müsste, welche Sprache in 10 Jahren am erfolgreichsten ist, im Sinne von am meisten verwendet, würde ich sicherlich auf TypeScript wetten.

Bei TypeScript musste ich am meisten Boilerplate-Code schreiben. Die Typerkennung ist nicht so gut, wie sie sein könnte.

Positives

- Grosses Ökosystem
- Nähe zu JavaScript
- Sichere Zukunft

Negatives

- Typsystem ist nicht sicher
- Man entwickelt viel gegen die Sprache

Reason Reason verfügt im Vergleich zu TypeScript über bessere funktionale Konstrukte (ADT, Pattern-Matching). Der Code von Reason finde ich am leserlichsten. Bei der Entwicklung kam es fast nie vor, dass ich ein Feature der Sprache selber vermisste. Reason gibt dem Entwickler genügend Freiheit, damit er sich die Restriktionen grösstenteils selber definieren kann. Die Side-Effects lassen sich nicht so gut isolieren wie bei Elm. Das Modulsystem von Reason finde ich etwas gewöhnungsbedürftig.

Leider ist Reason noch nicht reif genug, um benutzt zu werden. Die Fehlermeldungen des Compilers sind so mühsam, dass ich einen richtigen Hass auf den Compiler entwickelte. Es fehlt an fundamentalen Libraries. Die Dokumentation ist nicht gut und verstreut über mehrere Seiten.

Da Reason noch so jung ist, ist die Zukunft der Sprache noch recht ungewiss. Reason ist sehr nahe mit React verknüpft. Ich finde die Syntax von Reason weniger schön wie die von OCaml.

Positives

- Die Sprache ermöglicht dem Entwickler viel
- Gutes Typsystem
- Interaktion mit JavaScript einfach

Negatives

- Unreifer Compiler
- Unreifes Ökosystem
- Typen könnten besser automatisch erkannt werden

Elm Dadurch, dass von Elm sehr vieles vorgegeben ist im Vergleich zu den anderen Sprachen, passt vieles gut zusammen. Elm ist bewusst simpel gehalten. Ich habe Typeclasses und dergleichen bei der Entwicklung nie sehr vermisst, aber es fühlt sich einfach falsch an, verschiedene *map*-Funktionen für verschiedene Daten zu verwenden.

Ich finde die Syntax von Elm unnötig verbose. Die Compilermeldungen helfen einem gut. Die Sprache ist schnell erlernt.

Einige simple Sachen (wie JSON-Encoder oder verschachtelte Record-Updates) generieren unnötigen Aufwand.

Elm ist unter der Führung einer Person, diese Person entscheidet was in die Sprache kommt und was nicht. Diese Person ist eher restriktive. Die Sprache ist zwar vergleichsweise alt, hat aber immer wieder neue Rückwärtsinkompatibilitäten. Die Community ist sehr gespalten und man liest immer wieder mal von einem möglichen Fork der Sprache, da die Leute so unzufrieden sind.

Die Sprache lässt dem Entwickler wenig Freiheiten, so lange man auf gleicher Wellenlänge ist wie die Sprache, funktioniert dies gut. Für grössere Projekte lässt Elm dem Entwickler zu wenig Freiheiten, so lange das Projekt klein bleibt funktioniert es gut. Die Interaktion mit JavaScript ist mühsam.

Positives

- Hilfreicher Compiler
- Schnell erlernt

Negatives

- Sprache verändert sich noch zu oft
- Ein-Mann-Führung der Sprache
- Für eine rein funktionale Sprache ein schwaches Typsystem
- Sprache verfügt über wenige Features

Fazit Von der Sprache selber fände ich Reason die beste Variante. Aber im jetzigen Zustand ist Reason nicht brauchbar. Die funktionalen Features von Elm vereinfachen im Rahmen dieser Architektur vieles. Ein kleines Projekt ist in Elm am schnellsten entwickelt. Die Laufzeitstabilität von Elm ist besser wie die von TypeScript. Elm arbeitet in einem relativ eng gesteckten Bereich, dieser Bereich beherrscht Elm aber gut.

Die Führung von Elm bereitet mir am meisten Sorgen. Es gibt zu viele Rückwärtsinkompatibilitäten. Für eine rein funktionale Sprache ist das Typsystem (und andere Features) nicht für grössere Projekte ausgelegt.

TypeScript ist zwar nicht schön und hat nicht die besten funktionalen Features, aber für ein neues Projekt würde ich TypeScript verwenden. Für ein bestehendes Projekt würde ich sowieso TypeScript nehmen. Was bei TypeScript klar schlechter ist, ist die Laufzeitstabilität.

6 Ergebnis

Nachfolgend werden die Ergebnisse dieser Studie dargelegt. Dies wird in vier Schritten vollzogen, indem zuerst in einer Vergleichsübersicht die Resultate aus den detaillierten Vergleichen aufgelistet werden. Danach werden aufgrund der Erfahrungen Bereiche in jeder Sprache aufgezeigt, welche verbesserungswürdig sind. Schliesslich werden diese Resultate diskutiert und die Empfehlung der Autoren erläutert.

6.1 Vergleichsübersicht

Um einen Überblick über die wesentlichsten Merkmale der verglichenen Sprachen zu gewinnen, werden bei allen drei Varianten je die zwei positivsten und negativsten Eigenschaften erläutert. Die Klassifizierung der wichtigsten Bereiche basiert auf den Erfahrungen dieser Studie und würde vermutlich von anderen Personen anders gewichtet werden.

Sowohl die Reihenfolge der Merkmale als auch der Sprachen beinhalten keine Informationen zur Relevanz oder eine Bewertung.

6.1.1 TypeScript: Markanteste Merkmale

Positives

- Weite Verbreitung und entsprechend grosses Ökosystem: Dies resultiert in einem tiefen Projektrisiko, da damit gerechnet werden kann, dass sowohl Hilfe bei Problemen als auch weitere Entwickler gefunden werden können.
- Sehr nahe an JavaScript: JavaScript-Entwickler können sehr schnell produktiv eingesetzt werden und die Interaktion mit JavaScript ist einfach und natürlich.

Negatives

- Der durch die Architektur vorgegebene funktionale Stil wird nur bedingt unterstützt und dadurch entstehen unnötig verbose Lösungen.
- Laufzeitfehler können aufgrund des Typsystems nicht bzw. nur teilweise ausgeschlossen werden.

6.1.2 Reason: Markanteste Merkmale

Positives

- Die funktionalen Elemente der Architektur können sehr gut durch entsprechende Sprachelemente abgebildet werden.
- Nähe zu React: Für React-Entwickler verringert sich die Einarbeitungszeit aufgrund der sehr ähnlichen Elemente markant. Zusätzlich kann damit gerechnet werden, dass auch zukünftige React-Features unterstützt werden.

Negatives

- Das Tooling (Compiler, IDE, Code-Coverage-Berechnung usw.) ist noch nicht ausgereift.
- Kleines Ökosystem: In vielen Bereichen stehen keine ausgereiften und gut dokumentierten Libraries zur Verfügung.

6.1.3 Elm: Markanteste Merkmale

Positives

- Die Sprache unterstützt die Elm-Architektur ideal und entsprechend natürlich ist die Arbeit in dieser Architektur.
- Pure-Functions: Es können nur Pure-Functions implementiert werden, was die Laufzeitstabilität und Testbarkeit markant erhöht.

Negatives

- Core-Team: Die Sprachentwicklung wird sehr strikt und nur eingeschränkt transparent geführt. Das führt zu einem erhöhten Risiko für langfristige Projekte, welche auf Updates über längere Zeit angewiesen sind.
- Aufwendige Interaktion mit JavaScript: Es ist in den meisten Fällen sehr umständlich, eine bestehende JavaScript-Library oder -API zu integrieren.

6.1.4 Übersicht

Hier werden die Resultate aus den Einzelvergleichen in einer tabellarischen Übersicht zusammengefasst. Mehr Informationen zu den einzelnen Punkten sind in den entsprechenden Kapiteln zu finden.

Bereich	**TypeScript**	**Reason**	**Elm**
Laufzeitstabilität	0	+	++
Testbarkeit			
Funktionen	+	+	++
Komponenten	+	0	+
Side-Effects	+	-	N/A [1]
Interaktion mit JavaScript			
Einfache Interaktion	++	+	- -
Abgrenzung	0	+	++
Compiler			
Rückmeldungen	+	-	++
Geschwindigkeit	-	0/++ [2]	+
Grösse des Resultats	0	-	++
Projektsetup			
Simplizität	++	+	0
Funktionalitätsumfang	++	0 [3]	+
Ökosystem	++	-	0
IDE-Unterstützung	++	0	+
Weitere			
Einarbeitungsaufwand für JavaScript-Entwickler [4]	++	+	0
Zukunftsrisiko [5]	++	0	-
Skalierbarkeit	+	0	0
Unterstützung der Elm-Architektur [6]	0	+	++

Tabelle 17: Vergleichsübersicht

[1] Side-Effects sind innerhalb von Elm nicht möglich und somit nicht testbar. JavaScript-Code, welcher über Ports angesprochen wird, kann nicht ohne weiteres getestet werden.

[2] Bei komplett neuen Builds ist Null angemessen, bei inkrementellen Builds zwei Plus.

[3] Reason hätte ein Plus, wenn die Integration der Code-Coverage nicht so schlecht wäre.

[4] Positiv bedeutet wenig Einarbeitungszeit.

[5] Positiv bedeutet ein tiefes Risiko, dass in Zukunft die Sprache nicht mehr oder mit grossen Breaking-Changes unterstützt wird.

[6] Dies beinhaltet Elemente wie ADTs, Pattern-Matching, Pure-Functions, Side-Effect-Handling etc.

Die Bereiche *Entwicklungsgeschwindigkeit*, *Arbeit in grossen Teams*, *Wartbarkeit* und die subjektive Wertung werden in der Übersicht nicht aufgeführt, da diese Bereiche für eine generelle Wertung mit diesem Projekt als Grundlage nur ungenügend geeignet sind.
Einzelne Aspekte aus diesen Bereichen werden unter dem Punkt *Weitere* zusammengefasst.

6.2 Verbesserungspotential

Im Folgenden sind Veränderungen an den Technologie-Stacks dargelegt, welche deren Wertung in dieser Studie verbessert hätte.

6.2.1 TypeScript

Algebraische Datentypen Durch die Einführung von ADTs und Pattern-Matching auf die ADTs würden die Reducer übersichtlicher und kürzer. Zudem wäre der Compiler durch die Struktur der ADTs in der Lage zu überprüfen, ob alle Möglichkeiten berücksichtigt worden sind. Letzteres ist bei gewissen Datenstrukturen bereits jetzt möglich, allerdings in einer sehr verbosen Art.

Pattern-Matching Abgesehen von den ADTs würde Pattern-Matching auf Records (Objekt-Literalen) auch schon viel Code übersichtlicher und kürzer machen. Es besteht bereits ein Proposal für JavaScript:
https://github.com/tc39/proposal-pattern-matching

Fehlermeldungen verbessern Die Fehlermeldungen des Compilers bei einem Type-Mismatch sind je nach Situation sehr lange, obwohl die Unterschiede der Types nur klein sind. Dieses Problem wurde während der Studie von TypeScript teilweise bereits behoben, da ab Version 3.7 die Fehlermeldungen diesbezüglich verbessert worden sind.

Einfachere Container-Komponenten Container-Komponenten werden in Redux verwendet, um eine React-Komponente an den Store zu binden. Aktuell führt der Aufbau dieser (im Prinzip sehr einfachen) Konnektoren zu sehr viel Boilerplate-Code, insbesondere aufgrund der TypeScript-Typ-Definitionen. Es könnte eine TypeScript-Utility-Library entwickelt werden, welche diesen Typing-Aufwand mittels Typ-Inferenz verringert.

6.2.2 Reason

Bei Reason besteht das grösste Verbesserungspotential bei den Fehlermeldungen des Compilers und der Unterstützung durch Entwicklungsumgebungen (bessere Navigation, Refactoring). Weiter könnten die Dokumentation verbessert und mehr Standardlibraries entwickelt werden.

Saga Bei Reason mit ReasonReact und Reductive gibt es momentan noch keine gute Library, welche das Handling und das Testen von Side-Effects mit Reductive vereinfacht. Es wäre eine Library wie Redux-Saga wünschenswert, welche das Saga-Pattern implementiert.

React.string Beim Erstellen der React-Komponenten ist die Verwendung von *React.string* verbos, da alle String-Ausgaben in JSX damit getätigt werden müssen. Dieses Problem ist auf eine Limitierung der ADTs im Vergleich zu Union-Typen zurückzuführen [11].
Da JSX sowieso schon ein Makro ist, wäre es aus Anwendersicht empfehlenswert, JSX so zu erweitern, dass es normale Strings direkt zu Elementen umwandelt.

Test-Coverage mit Test-Runner ermitteln Der häufigste Test-Runner Jest ist nicht mit dem Coverage-Modul von Reason kompatibel. Es würde die Coverage-Ermittlung erheblich vereinfachen, wenn die Coverage-Ermittlung gleichzeitig mit den Tests durch den Test-Runner ermittelt werden würde.
Zusätzlich wäre beim Coverage-Modul wünschenswert, dass alle Dateien für die Coverage berücksichtigt würden und nicht nur durch die Tests tatsächlich geladene.

6.2.3 Elm

In Elm ist es am schwierigsten, realistische Verbesserungsvorschläge zu präsentieren. Der Grund dafür liegt darin, dass die meisten Punkte, welche in dieser Arbeit kritisiert werden, auf Sprachebene liegen. Die Sprache kann aber praktisch nicht von ausserhalb des Core-Teams verändert werden und dieses Team möchte die Sprache so einfach wie möglich halten.
Das führt aber wiederum dazu, dass gewisse Features, welche aus Gründen der Einfachheit weggelassen werden, zu einer aufwendigeren Entwicklungsarbeit führen. Das kann aber wie erwähnt durch die Community faktisch nicht gelöst werden.

Folgendes Beispiel unterstreicht diesen Grundton: Hinsichtlich einer möglichen Folgearbeit wurden Veränderungen an Elm vorgeschlagen wurden. Zwecks dessen wurde eine Anfrage an den Ersteller von Elm (Evan Czaplicki) über Slack mit entsprechenden Vorschlägen gestellt. Diese wurden alle mit der Begründung abgelehnt, dass es einen Grund dafür gebe, dass diese Features nicht in Elm vorhanden sind.

Verschachtelte Record-Updates Die Syntax, um verschachtelte Records zu verändern, ist sehr verbos[17]. Es gab bereits Bestrebungen, diese Syntax zu ändern [12], was aber aus verschiedenen Gründen abgelehnt wurde. Hauptgrund ist, dass Elm eine möglichst einfache Syntax beibehalten will. Aus Sicht der Autoren ist aber anzunehmen, dass dies auch für Anfänger einen positiven Effekt hätte.

Case mit Records Das Case-Sprach-Konstrukt erlaubt Pattern-Matching in Elm. Leider lassen sich damit aber keine Records direkt verarbeiten. Dies hätte einige Code-Stellen im Beispiel-Projekt vereinfacht.

Code-Erstellung zur Compile-Time Eine andere invasivere Veränderung wäre, dass ein Konstrukt für Code-Erstellung zur Compile-Time erstellt werden würde. Es würde sich z. B. etwas wie *derive* in Haskell anbieten. Dieser Mechanismus könnte auch für eine einfachere Implementation von Encodern und Decodern verwendet werden.

[17] http://faq.elm-community.org/#how-can-i-pattern-match-a-record-and-its-values-at-the-same-time

6.3 Diskussion und Ausblick

Vorbemerkung zur Elm-Architektur Wie in der Einleitung erwähnt ist die Evaluation der Elm-Architektur im Vergleich zu anderen Architektur-Ansätzen kein Teil dieser Arbeit. Dennoch kann angemerkt werden, dass alle Anforderungen der Beispiel-Applikation innerhalb der Elm-Architektur sehr gut abgebildet werden konnten.

Basierend auf den vorliegenden Vergleichen kann klar gesagt werden, dass es keine Lösung gibt, welche in jedem Bereich vollständig überzeugt. Deshalb muss je nach Fall abgewogen werden, welche Kriterien am meisten Gewicht haben.

Für lange und grosse Projekte ist meist einer der wichtigsten Faktoren, dass die verwendeten Technologien für eine absehbare Zeit unterstützt und weiterentwickelt werden. Ausserdem muss abschätzbar sein, dass keine so grundlegenden Änderungen gemacht werden, dass die ganze Applikation neu geschrieben werden muss. Insbesondere aus diesen Gründen empfehlen die Autoren, sofern die Elm-Architektur eingesetzt werden soll, in diesem Fall TypeScript mit React und Redux zu verwenden.
Dies gilt auch für Applikationen, welche auf einige externe JavaScript-Libraries und somit eine gute Interaktion mit JavaScript angewiesen sind.

Für Projekte, welche klein sind oder auf eine äusserst hohe Laufzeitstabilität angewiesen sind, kann Elm eine reale Alternative darstellen. Elm hat sehr markante Stärken, aber genauso signifikante Schwächen. Wenn sich ein Entscheidungsträger dieser Schwächen bewusst ist und sie für nicht ausschlaggebend in seinem Projekt befindet, kann Elm auch zum jetzigen Zeitpunkt eingesetzt werden.

Reason befindet sich an einer ganz anderen Stelle als die anderen zwei Kandidaten. Viele sprachliche Elemente zeichnen eine ideale Option für die Ablösung von TypeScript in React-Projekten ab, jedoch sind die Aspekte ausserhalb der Sprache wie das Tooling und das Ökosystem noch nicht soweit, dass diese Ablösung zum jetzigen Zeitpunkt empfohlen werden kann. Wenn sich diese Kritikpunkte in Zukunft verbessern, was eine realistische Möglichkeit ist, dann könnte Reason zu einer besseren Option als TypeScript für lange und grosse Projekte werden.

Literatur

[1] "**Prior Art - Redux**." https://redux.js.org/introduction/prior-art#elm.
Informationen über die Inspirationsquellen von Redux, letzter Zugriff am 18.12.2019.

[2] "**Can you explain the structure of the main NoRedInk Elm app?**."
https://dev.to/joshhornby/comment/1ip.
Erklärung über die Verwendung von Elm bei NoRedInk, letzter Zugriff am 18.12.2019.

[3] "**Messenger.com Now 50% Converted to Reason**."
https://reasonml.github.io/blog/2017/09/08/messenger-50-reason.
Erklärung über die Verwendung von Reason bei Facebook Messenger, letzter Zugriff am 18.12.2019.

[4] "**Revisiting React Testing in 2019**."
https://codeburst.io/revisiting-react-testing-in-2019-ee72bb5346f4.
Blog-Artikel über die Möglichkeiten für das Testen von React-Komponenten, letzter Zugriff am 18.12.2019.

[5] "**Testing Overview - React**." https://reactjs.org/docs/testing.html.
Test-Empfehlungen von React, letzter Zugriff am 18.12.2019.

[6] "**elm test**." https://github.com/elm-explorations/test.
Dokumentation von elm-test, letzter Zugriff am 18.12.2019.

[7] D. A. Rauschmayer, "**Exploring ReasonML, What is ReasonML?**."
http://reasonmlhub.com/exploring-reasonml/ch_about-reasonml.html.
Buch über ReasonML, letzter Zugriff am 18.12.2019.

[8] "**Elm Timeline**." https://github.com/elm/projects/blob/master/roadmap.md#what-is-the-timeline.
Zeitlicher Zukunftsplan von Elm, letzter Zugriff am 18.12.2019.

[9] "**Elm 0.19 Broke Us**." https://dev.to/kspeakman/elm-019-broke-us--khn.
Beispiel von Rückwärtsinkompatibilitäten von Elm, letzter Zugriff am 18.12.2019.

[10] "**ECMAScript Pattern Matching**." https://github.com/tc39/proposal-pattern-matching.
Vorschlag für die Einführung von Pattern Matching in JavaScript, letzter Zugriff am 18.12.2019.

[11] "**Having to use stringToElement() is awkward**."
https://github.com/reasonml/reason-react/issues/138.
Diskussion über die Verwendung von Strings in Reason JSX, letzter Zugriff am 18.12.2019.

[12] "**Proposal: Field update/map syntax**." https://github.com/elm/compiler/issues/984.
Diskussion über eine mögliche Syntax-Änderung von Record-Updates in Elm, letzter Zugriff am 18.12.2019.

Bild-Quellen

Alle Bilder, deren Quellen hier nicht aufgeführt sind, stammen von den Autoren.

Logo Reason auf der Titelseite: https://reasonml.github.io/
Logo TypeScript auf der Titelseite: https://github.com/remojansen/logo.ts
Logo Elm auf der Titelseite: https://elm-lang.org/
Abbildung 2 auf Seite 14: https://www.npmtrends.com/@angular/core-vs-angular-vs-react-vs-vue
Abbildung 8 auf Seite 54:
https://www.npmtrends.com/typescript-vs-elm-vs-bs-platform-vs-react-vs-redux-vs-reason-react-vs-reductive
Abbildung 9 auf Seite 55: https://www.npmtrends.com/elm-vs-bs-platform-vs-reason-react-vs-reductive

Glossar

Bisect_ppx Code-Coverage-Tool für OCaml und Reason, unterstützt BuckleScript und Native
https://github.com/aantron/bisect_ppx

BuckleScript Compiler für Reason und OCaml zu JavaScript
https://bucklescript.github.io/

Create React App Create React App ist ein Tool aus dem React-Ökosystem, welches ein einfaches Setup von React-Applikationen erlaubt.
https://create-react-app.dev/

DOM DOM steht für Document Object Model und beschreibt die HTML-Struktur einer Webseite als Modell. Damit wird ermöglicht, mit JavaScript Veränderungen über das DOM an der Webseite vorzunehmen.
https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model

Elm Funktionale Programmiersprache, welche für Web-Frontendapplikationen entwickelt wurde.
https://elm-lang.org/

Elm-Architektur Die Elm-Architektur ist eine unidirektionale Architektur, welche den Einsatz von Pure-Functions für die Business-Logik erleichtert. Sie stammt aus der Programmiersprache Elm, in der sie die einzig mögliche Architektur darstellt.
https://guide.elm-lang.org/architecture/

Fuzz-Testing Fuzz-Testing ist eine automatisierte Testing-Technik, welche Funktionen mit zufällig generierten Daten testet.
https://en.wikipedia.org/wiki/Fuzzing

Haskell Funktionale Programmiersprache, die Elm stark beeinflusst hat.
https://www.haskell.org/

Haste Haste ist ein Projekt, um Haskell zu JavaScript zu kompilieren.
https://haste-lang.org/

Immutability Immutability beschreibt den Zustand von Daten, welche nicht verändert werden können. Dies kann sowohl zur Laufzeit als auch mit einem Typsystem sichergestellt werden.

Jest JavaScript-Testing-Framework, welches von Facebook entwickelt und mit Create-React-App ausgeliefert wird.
https://jestjs.io/

Linting Linting ist eine Technik der automatisierten Qualitätssicherung, welche Source-Code auf bestimmte Kriterien überprüft. Dies können z. B. Naming-Kriterien oder Verbote für gewisse Konstrukte sein.

LocalStorage Die LocalStorage ist eine Browser-API, welche das Speichern von Daten über eine einzelne Browser-Sitzung hinaus ermöglicht.
https://developer.mozilla.org/en-US/docs/Web/API/Web_Storage_API

NPM Der Node Package Manager ist sowohl ein Tool als auch eine Plattform, über die Node-Module (JavaScript) mit bestimmten Versionen und Abhängigkeiten verwendet werden können.
https://www.npmjs.com/

OCaml Funktionale Programmiersprache, welche auch imperative und objektorientierte Paradigmen erlaubt.
https://ocaml.org/

Pure-Function Pure-Functions sind Funktionen, welche keine Side-Effects beinhalten. Das führt dazu, dass die Output-Werte nur von den Input-Werten abhängen. Mathematische Funktionen sind auch Pure-Functions.

PureScript Funktionale Programmiersprache, welche zu JavaScript kompiliert wird und sehr viele Ähnlichkeiten mit Haskell hat.
http://www.purescript.org/

React UI-Library für JavaScript, um wiederverwendbare UI-Komponenten zu erstellen.
https://reactjs.org/

Reason Funktionale Programmiersprache, welche auf OCaml basiert. Die Syntax ist an JavaScript angenähert worden, um sie für Webentwickler leichter verständlich zu machen.
https://reasonml.github.io/

ReasonReact React-Support für Reason
https://reasonml.github.io/reason-react/

Reductive Redux-Support für Reason
https://github.com/reasonml-community/reductive

Redux State-Management-Library für JavaScript, welche sich gut mit React zusammen verwenden lässt.
https://redux.js.org/

Redux-Saga Middleware-Library für Redux, welche eine kontrollierte Verarbeitung von Side-Effects zulässt.
https://github.com/redux-saga/redux-saga

Shallow Rendering Shallow Rendering ist der Name einer Technik beim Testen von React-Komponenten (oder Komponenten einer anderen komponenten-basierten Frontend-Library), bei der eine Komponente ohne ihre Subkomponenten gerendert wird. Dadurch werden die Abhängigkeiten zu anderen Komponenten gemockt und die Komponente wird in Isolation getestet.
https://airbnb.io/enzyme/docs/api/shallow.html

TDD TDD steht für Test-Driven-Development und ist eine Entwicklungsmethode, bei der zuerst automatisierte Tests erstellt werden und erst danach die getesteten Funktionen/Komponenten etc. implementiert werden.

Transpiler Ein Transpiler, auch Trans-Compiler oder Source-to-Source-Compiler genannt, ist eine spezielle Art eines Compilers, der Code nicht zu Maschinencode umwandelt, sondern in eine andere Programmiersprache übersetzt. Im Web werden Transpiler oft eingesetzt, da Browser lange Zeit nur JavaScript ausführen konnten. https://en.wikipedia.org/wiki/Source-to-source_compiler

TypeScript Typisiertes Superset von JavaScript
http://www.typescriptlang.org/

Web-Component Web-Components ist ein Sammelbegriff für Browser-APIs, welche das Erstellen von eigenen HTML-Elementen ermöglichen.
https://developer.mozilla.org/de/docs/Web/Web_Components

WebAssembly WebAssembly ist eine neue Art von Code, welcher neben JavaScript vom Browser ausgeführt werden kann. Durch die Maschinen-Nähe können auch Sprachen wie C und C++ verwendet werden.
https://developer.mozilla.org/en-US/docs/WebAssembly