Nie wieder Laufzeitfehler! - Fehlerfreie Web-UIs mit Elm

von Martin Grotz, redheads Ltd., Schillerstraße 14, 90409 Nürnberg

Einleitung

Zum Vortrag

Zuerst einmal: Was ist dieser Vortrag nicht? Er ist keine strukturierte Einführung in die Programmiersprache Elm. Ich zeige immer mal wieder kleine Schnipsel von Elm-Code und erkläre diese auch, aber es gibt keine Step-By-Step-Einführung in Syntax&Co.

Es werden aber Vorteile (und Nachteile) von Elm gegenüber anderen gängigen Frontend-Programmiersprachen und Frameworks anhand von allgemeinen Problemstellungen, die mir als Frontend-Entwickler so im Alltag begegnen, aufgezeigt.

Was ist Elm?

Elm ist eine vollständig funktionale Programmiersprache, die aus Sicht des Programmierers frei von Seiteneffekten ist und außerdem einige starke Garantien abgibt, zum Beispiel das völlige Fehlen von Laufzeitfehlern.

Elm selbst ist noch relativ jung und aktuell in der Version 0.19 verfügbar. Es ist außerdem die einzige Programmiersprache, die ich kenne, bei der von Version zu Version auch mal Funktionen und Sprachelemente entfernt werden, weil sie sich in der Praxis nicht bewährt haben.

Die Elm-Syntax sieht, wenn man zum Beispiel von Java kommt, erstmal gewöhnungsbedürftig aus, da viel mit Whitespace gearbeitet wird und wenig mit Klammern.

Die Typsignatur lügt

In letzter Zeit, auch bedingt durch Angular, bekommt Typescript immer mehr Verbreitung. Und im Vergleich zu Javascript ist Typescript wirklich ein großer Schritt nach vorne. Aber auch bei Typescript stimmen Typsignatur bei Funktionen bzw. Methoden nicht unbedingt mit dem tatsächlichen Verhalten der Funktion überein.

Beispielsweise kann ich bei Typescript in der Standardeinstellung bei einer Funktion sagen, dass sie einen "string" zurückgibt, dann aber in der Funktion selbst einfach "return null" schreiben und der Compiler ist trotzdem glücklich. Die Typsignatur enthält also nicht die ganze Wahrheit, und daher kann man sich nicht auf diese verlassen.

Versuche ich dann anschließend auf den Rückgabewert zuzugreifen, bekomme ich den allseits bekannten Laufzeitfehler "Cannot read property XYZ of null". Hier empfiehlt es sich also auf jeden Fall, den Typescript-Compiler so zu konfigurieren, dass er streng auf "null" prüft, um diese Art von Fehlern vorab verhindern zu können.

In Elm hingegen ist die Grundidee, mögliche Fehler explizit zu machen. Hierfür kennt Elm zwei Datentypen, die im folgenden kurz vorgestellt werden:

Einmal ist das "Maybe", welches dazu dient, das mögliche Nicht-Vorhandensein eines Werts auszudrücken. Das wirklich spannende daran ist aber, dass Elm uns beim späteren Verarbeiten des Werts dazu zwingt, immer beide Fälle zu beachten. Man kann also niemals vergessen, den Nichts-Vorhanden-Fall auch explizit zu bearbeiten. Hat man eine Funktion, die explizit, oder noch schlimmer nur implizit, auch "null" zurückgeben kann, vergisst man leicht mal an irgendeiner Stelle darauf zu prüfen und schon hat man einen Fehler, der erst zur Laufzeit auftaucht.

Noch weiter geht in Elm der "Result" Datentyp: Mit diesem kann man nicht nur sagen, dass etwas schief gelaufen ist, sondern auch direkt, was. Auch hier ist es wieder so, dass der Aufrufer immer sowohl den Gut- als auch den Schlecht-Fall behandeln muss. Da führt kein Weg dran vorbei. Diesen Datentyp kann man zum Beispiel gut bei Anfragen an einen Server verwenden, die ja immer mal fehlschlagen können, oder beim Parsen von JSON, welches ja durchaus auch mal invalide sein kann, gerade wenn man die Gegenstelle nicht unter eigener Kontrolle hat.

Elm hilft einem also gegen diese Probleme, indem es mögliche Fehlerzustände explizit macht und einen zur Behandlung zwingt. Das klingt erstmal nervig, hilft aber im tatsächlichen Betrieb dann ungemein.

Zustand ist kompliziert

Das Verwalten des Zustands der Applikation ist oft ein Problem. Vor allem, wenn die Änderungen aus verschiedenen Quellen kommen, zum Beispiel durch Benutzer-Interaktion, Timer und Websocket-Nachrichten. Wenn dann noch der Applikations-Zustand über verschiedene Komponenten verteilt ist, die alle kreuz und quer miteinander verbunden sind, verliert man schnell die Übersicht. Dann rauszufinden, warum ein bestimmter Zustand entstanden ist, der dann zu einem Fehler geführt hat, ist oft nahezu unmöglich.

Elm wirkt dem entgegen, da es eine Architektur mit unidirektionalem Datenfluss erzwingt. Außerdem wird der gesamte Zustand an einer zentralen Stelle verwaltet. Dieses Vorgehen hat mittlerweile dank React Redux einigermaßen Verbreitung gefunden. Dazu kommt dann noch, dass alle Daten in Elm unveränderlich sind ("immutable"), so dass nirgends "aus Versehen" etwas geändert werden kann.

Die Elm-Architektur folgt dem "Model-View-Update"-Muster. Beim Start der Applikation werden Initialdaten ins Model geschrieben. Dieses wird dann an eine View-Funktion übergeben, die daraus die aktuelle Darstellung erzeugt. Aus der View

heraus können dann Nachrichten ausgelöst werden, um Beispiel bei einem Button-Klick. Diese werden zusammen mit dem aktuellen Model an eine Update-Funktion übergeben. Diese unterscheidet dann anhand von Nachricht und Modell, was zu tun ist. Sie kann einerseits eine veränderte Kopie des Models erzeugen, andererseits aber auch Aktionen auslösen, die Seiteneffekte haben können. Einer der Punkte, der mir am Anfang Schwierigkeiten bereitet hat, ist, dass Elm Befehle mit Seiteneffekten nicht sofort ausführt. Stattdessen übergibt man diese im Rückgabewert der Update-Funktion an die Elm-Runtime, die diese dann ausführt und nach dem Abschluss des Befehls dann wiederum eine Nachricht an die Update-Funktion übergibt.

Der Datenfluss ist also immer: Model an View übergeben und darstellen. View löst Nachrichten aus, die von der Update-Funktion verarbeitet werden. Danach gibt es ggf. ein verändertes Model, welches wieder an die View übergeben wird. Und immer so weiter. Im Endeffekt ist jedes Elm-Programm eine Endlos-Schleife aus Model-View-Update.

Es gibt dann noch als zusätzliche Nachrichten-Quelle sogenannte Subscriptions, zum Beispiel Timer oder Interaktion mit externem Javascript oder die Ankunft von Daten über eine Websocket-Verbindung.

Diese unidirektionale Datenarchitektur ermöglicht es einem, in Elm mit dem Debugger durch die Zeit zu reisen. Genau wie in den Redux-Dev-Tools kann man die Änderungen im Model nach jeder Nachricht nachverfolgen und die Applikation auch an jeden Punkt der Vergangenheit zurückspulen. Die ausgelösten Nachrichten und Model-Änderungen können auch exportiert werden, so dass Tester einfach einen Export schicken, ich importiere den dann in den Debugger und kann mir genau anschauen, welche Reihenfolge von Nachrichten zum Problem geführt hat.

Diese mächtige Funktion braucht man auch, da Elm vom Elm-Compiler in Javascript übersetzt wird. Der generierte Code ist aber nicht gerade gut menschenlesbar, d.h. das Debuggen im generierten Code ist sehr schwierig. Bisher musste ich das aber noch nie machen, da ich alle Denkfehler mit Elm-Compiler+Debugger zusammen mit der Elm-Architektur gefunden habe.

Änderungen sind fehleranfällig

In jedem Projekt kommt irgendwann der Moment, in dem eine Änderung gemacht werden muss - sei es ein Refactoring, eine Erweiterung oder ein Bugfix. Bei verwobenem Code mit offensichtlichen und versteckten Seiteneffekten kann das schwierig werden. Besonders, wenn man keine oder zumindest keine gute Tool-Unterstützung hat. Die modernen IDEs sind da schon sehr gut geworden, aber gerade bei Programmiersprachen oder Frameworks im Web gibt es oft einen "Medienbruch", wenn man zum Beispiel die Logik im Typescript schreibt, das Template aber in HTML, und die beiden dann von der IDE zusammengebracht werden müssen. In Elm sind alle Teile der Anwendung in der gleichen Programmiersprache geschrieben, und bei allen Teilen kann der Elm-Compiler hilfreich unterstützen.

Das ausdrucksstarke Typsystem und vor allem das Pattern Matching - sowas wie ein Switch-Case, nur viel besser - helfen in Verbindung mit Union Types sehr, gleich ganze Fehlerklassen auszuschließen. Und wenn man dann mal was ändert, unterstützt einen der Compiler mit echt guten Fehlermeldungen. So kann man bei Änderungen an einer Stelle, zum Beispiel der View, anfangen, und sich von da ausgehend vom Compiler leiten lassen. Gelegentlich spricht man dabei halb-scherzhaft auch vom "Compiler Driven Development".

Meine schönste Erfahrung in diesem Zusammenhang war es, als der Kunde bei meinem letzten Elm-Projekt mit einem dringenden Änderungswunsch kam, und ich ganz entspannt "Okay, kein Problem" sagen konnte. Die Änderung habe ich dann angefangen, und als am Ende der Compiler wieder glücklich war, lief die ganze Anwendung direkt wieder problemlos.

Außerdem hilft es enorm, dass alle Funktionen "pur", also ohne Seiteneffekte sein müssen. Dadurch kann man Logikfehler - denn vor diesen schützt einen Elm dann leider auch nicht - oft schon durch Hinschauen und Nachdenken erkennen und lösen. Man muss als nicht erst langwierig den Debugger rausholen und lange durch den ausgeführten Code springen, um rauszufinden, wo jetzt das Problem liegt.

Tests sind zu anstrengend

Je nach Framework ist das Schreiben von Tests richtig anstrengend oder nachgerade nervig. Ich finde zum Beispiel das Testen von Angular-Komponenten sehr schwierig, fehleranfällig und zeitraubend. Daher teste ich meistens nur die Services - aber jede Interaktion mit dem tatsächlichen HTML fällt damit unter den Tisch. Und hier verbergen sich dann oft die fiesen Fehler.

In Elm ist erstmal alles entweder eine Funktion oder ein unveränderlicher Datensatz. View, die von der Runtime zu HTML gerendert wird? Aus Programmierer-Sicht eine Seiteneffekt-freie Funktion. Update, in dem veränderte Kopien der Daten anhand von Nachrichten gemacht werden? Ebenfalls eine Seiteneffekt-freie Funktion. Daher ist beides nur anhand der Eingabedaten testbar. Aufwändige Setups und Mocks entfallen.

Will man doch mal Seiteneffekte haben, zum Beispiel weil man eine HTTP-Anfrage stellen muss, übergibt man diesen Wunsch als Datensatz an die Elm-Runtime. Diese führt den Befehl aus und meldet sich mit einer Nachricht, was wiederum in die gewohnte Update-Funktion eingespeist wird. Also kann man auch Seiteneffekte rein über Ein- und Ausgabedaten abtesten. Als Entwickler wird man in der kleinen heilen Elm-Welt komplett abgeschirmt und kann sich rein auf das Lösen der fachlichen Probleme konzentrieren.

Praktischerweise ist Elm schon direkt nach der Installation vollständig mit guten Tools ausgestattet. Es kommt also "Batteries included" daher. Der mitgelieferte Test-Runner ist zugleich das Test-Framework. Für das direktere Testen von HTML&Co. gibt es auch noch einige zusätzliche Bibliotheken. Bisher habe ich diese aber in der Praxis nicht gebraucht. Und alle Tools im Elm-Umfeld sind sprechend benannt. Der

Test-Runner heißt nicht Karma und führt das Framework Jasmine aus. Sondern er heißt elm-test. Der Compiler heißt elm-make. Der Auto-Formatter heißt elm-format, usw.

Praktische Erfahrungen

Das letzte Projekt, bei dem ich Elm einsetzen konnte, war ein Wirtschaftlichkeits-Rechner für den Einsatz von KWKK (Kraft-Wärme-Kälte-Kopplung) Anlagen. Die Server-Seite besteht hierbei hauptsächlich aus einem Rechenkern und ein bisschen Persistenz und Benutzerverwaltung und einer Administrations-Oberfläche. Der Server ist in F# geschrieben, der Client als Webapp in Elm.

Hier muss ich ehrlich sagen: Ich hatte schon lange nicht mehr so viel Spaß beim Programmieren wie mit F# und vor allem Elm. Zwar gibt es einige Sachen, die in anderen Frameworks sehr einfach sind, die in Elm echt schwierig sind, weil man dazu auf Ports für Javascript-Interop angewiesen ist, dafür waren Zustandsmanagement und Fehlerbehandlung sehr einfach. Eine Sache, in der Elm nicht besonders gut ist, sind große Formulare. Nicht etwa, weil die nicht gut funktionieren würden oder langsam wären, sondern weil man dann doch recht viel Boilerplate-Code braucht. Es gibt zwar in der Community schon verschiedene Ansätze, das besser zu machen, aber jeder davon hat verschiedene Nachteile. Auch viel Boilerplate-Code bekommt man durch die Vorgabe, dass auch das Senden und Empfangen von JSON als Grenze zur Elm-Welt gilt. Hier muss man Encoder und Decoder schreiben, oder sich generieren lassen. Bei großen Datentransfer-Strukturen kann das schon ausarten.

Praktischerweise lässt sich Elm auch sehr gut als Einstieg in die funktionale Programmierung nutzen, da es nur einen eingeschränkten Sprachumfang hat und einige kompliziertere Bausteine von FP gar nicht erst drin sind. Meine beiden Kollegen konnte ich daher über Elm an funktionale Programmierung "gewöhnen", um anschließend das doch etwas mächtigere und umfangreichere F# zu erklären. Dazu kommt auch noch, dass es für Elm einen eigenen Code-Formatter gibt, "elm-format". Dieser hat genau einen gültigen Regelsatz, welcher 100% durchgesetzt wird. Dadurch ist jeder Elm-Code identisch formatiert, egal wer ihn geschrieben hat. Das erleichtert zusammen mit der Tatsache, dass jedes Elm-Programm zwangsweise der Elm-Architektur folgen muss, das Verständnis von fremdem Code enorm.

Das Wichtigste ist aber, dass die Garantien von Elm zusammen mit dem vollständig funktionalen Backend in F# dazu geführt haben, dass wir bisher noch keine Applikations-internen Fehler hatten. Probleme gibt's höchstens immer dann, wenn man in F# das normale .NET-Framework nutzt und dann doch wieder mit Exceptions&Co. hantieren muss, weil das Framework leider kein Maybe und Result kennt. Vergisst man hier mal ein try-catch rumzumachen, kann die Applikation doch wieder crashen...

Abschluss

Als Fazit bleibt mir zu sagen, dass es sich sehr lohnt, Elm mal auszuprobieren. Gerade jetzt mit der letzten Version kamen nochmal viele kleine Performance-Verbesserungen dazu und der generierte Code ist nochmal deutlich kleiner geworden. Selbst wenn man Elm dann nicht in der alltäglichen Arbeit einsetzt hilft einem die Kenntnis von ein bisschen funktionaler Programmierung und von einer unidirektionalen Datenarchitektur immer mal wieder dabei, den Code in der Programmiersprache der Wahl ein bisschen besser zu schreiben.

Zumindest die Architektur kann man ja mittlerweile auch in anderen Sprachen und Frameworks nutzen, zum Beispiel in React mit Redux, Angular mit ngrx oder bei Vue mit Vuex. Auch mit F# gibt es Bibliothek, zum Beispiel zur Nutzung mit WPF oder Xamarin. Möchte man eher die funktionale Programmierung haben, nicht aber die Architektur, lohnt es sich auch sehr, sich mit PureScript zu beschäftigen. Da gibt's dann auch wieder Bindings zum Beispiel an React, so dass man die volle Packung FP hat, ohne hart an den Rest des Elm-Ökosystems gebunden zu sein. Möchte man weiter mit Typescript arbeiten, sollte man zumindest den Compiler und den Linter maximal scharf einstellen, damit diese möglichst viele Fehler erkennen, bevor diese erst zur Laufzeit gefunden werden...