Hochschule Bremen

Exposè

Media-science Bsc.

Roman Quistler

April 15, 2019

Contents

1 Introduction

2 Problem and solution

2.1 Problem

Today's applications seem to be increasing in complexity over the last few years. Especially in the field of frontend development, the range of functions has increased [1]. The web development has transitioned from server rendered, page-reloading websites, to modern so called single page applications or SPA's. But the same applies to the mobile world: Social networks, navigation, sharing and editing files together is a common desire by the user. Ein großer Teil der Applikationen steht somit auch im Austausch mit APIs, greift auf lokale und externe Datenbanken zu oder interagiert mit dem zugrundeliegenden System selbst (bswp. die periodische Erfassung des Standortes via GPS oder WiFi). Zusätzlich zu der Interaktion mit dem Nutzer und dessen Einfluss, muss die Anwendungen auf oben genannte Einflüsse reagieren können und vorrausgegangene Ereignisse und deren Auswirkung speichern. Es muss bekannt sein, welche Zustände bei welchen Ereignissen eintreten können und welche nicht. Dies ist nötig, um eine zubeverlässige Funktionsweise einer Appliaktionen zu garantieren. Folgendes Beispiel soll den Sachverhalt verdeutlichen: Der Nutzer befindet sich auf einem "Anmelden" Fenster. In diesem kann der Nutzer eine E-Mail-Adresse und ein Passwort eingeben. Darunter befindet sich ein herkömmlicher "Anmelden" Knopf. Des weiteren existiert die Möglichkeit auf "Passwort vergessen?" zu klicken. Folgende Zustände können hierbei eintreten:

- Die E-Mail-Adresse ist nicht zulässig und das Passwort Feld leer
  - Der "Anmelden" Knopf ist deaktiviert
- Die E-Mail-Adresse ist zulässig und das Passwort Feld leer
  - Der "Anmelden" Knopf ist deaktiviert
- Die E-Mail-Adresse ist zulässig und das Passwort Feld nicht leer
  - Der "Anmelden" Knopf ist aktiviert
- Der Nutzer betätigt den "Anmelden" Knopf
  - Der "Anmelden" Knopf ist aktiviert
- Die Email ist inkorrekt oder existiert nicht
- Die Email ist korrekt und das Passwort falsch
- Die Email ist korrekt und das Passwort richtig
  - Der Nutzer gelangt zum nächsten Fenster
- Eine dreimalige falsche Eingabe des Passworts führt zu einer temporären Sperrung
- Der Nutzer klickt auf "Passwort vergessen"

- Es gibt einen Netzwerkfehler
  - Es wird eine Fehlermeldung angezeigt
- Es besteht keine Verbindung zum Internet
  - Es wird eine Meldung angezeigt
- Der Nutzer dreht das Smartphone
  - Der Zustand muss gespeichert werden
- Die Appliaktionen wird vom System beendet
  - Der Zustand muss gespeichert werden

Anhand dessen wird sichtbar, wie viele Ereignisse und Zustände bereits aus einer simplen Funktion wie "Annmelden" hervorgehen. Die oben aufgezeigte Problematik ist ein Faktor, der dazu führen kann, dass die Implementierung einer Funktion (hier: Anmelden), nicht alle möglichen Zustände/Fälle abdeckt. Dies wiederum lässt etwas zu, das einem nichtdeterministisches Verhalten gleichkommt. Ein Ereignis, das nicht bedient wird, könnte unerwartete Komplikationen hervorrufen. Zusätzlich, wie in den meisten Systemen ohn Struktur, verleitet dies zu unleserlichem Quellcode. Die Schwiriegkeit die hierbei unter anderem entsteht, ist, die Fülle an möglichen Zuständen zu konkretisieren und formal als auch inhaltlich "ansprechend" darzustellen/zu beschreiben. Wie sollte die Struktur eines Programmes aufgebaut/organisiert werden, wenn man Zustandsorientiert arbeitet? Wirft der Entwickler einen Blick auf die Implementierung einer Funktion (wie das obige Beispiel des Vorgangs des Anmeldens), ist oft nicht gleich ersichtlich, welche Zustände insgemein existieren. Überhaupt ist es oft kompliziert, den Ablauf eines Programmes ohne größen Aufwand nachzuvollziehen. Dazu gesellt sich zwangsläufig die Frage, auf welche Art und Weise ein Zustand ünberhaupt definiert wird. Wird er dynamisch, anhand eines hinterlegten Regelwerks gebildet, oder ist er von Beginn an festgelegt? Nicht weniger unherheblich ist, wie der Übergang von einem Zustand in einen anderen Zustand erfolgt und was genau der Initiator eines Übergang ist. Ein weitere, nicht zu unterschätzende Problematik ist die bereits erwähnte asynchrone Kommunikation, bzw. das oft angewendete "Multi Threading" in heutigen Appliaktionen. Dies führt zu sogenannten "Side effects" und erschwert es, den Ablauf eines Programmes nachzuvollziehen. (Callback Hell)

2.2 Solution

In order to reduce the cognitive burden on the developer of managing the different conditions and states the application can end up in, a small and lightweight, but opinionated library will be developed in order to test this hypothesis.

The library tries favouring a declarative programming paradigm for its public api. The core of the library will consist of a "Mealy" state machine. Thus the output of this finite state machine is determined by both its current state and inputs. The input reflects an "event" that took place in the user facing layer. That "event" is flexible and

can be directly caused by the user (and his interactions) but also invoked by the system directly as part of an initialisation flow. Since a state machine is often defined purely in terms of mathematical functions with no reference to mutable state, the library will heavily rely on functional programming that embodies these concepts.

Internally pure functions will be used whenever it's possible as a methodology to adhere a state machines functionality.

In addition to that immutability will play a significant role. It makes data structures read only, limits them in their functionality and thus removes possible and unwanted side effects (Copy-on-write). Thereby it also makes multi-threading "safer" or "thread-safe". The "state" data structure will serve as the single source of truth.

Another important part is to handle exceptions effeciently. Often they are used to control the flow of the application, which can be viewed as an anti-pattern [2]. Although it will be up to the developer to decide what is considered a(n) exception/error/failure, the library will try to help to make it more convenient to handle. For this task, the library again will use a concept of functional programming called "monads". The exception handling is facilitated by the use of the so called 'Either' monad. This allows a function to return either a left value or a right value, but only one exclusively at any given time. By convention but not limited to, the left side is generally the negative case as in a failure or exception and the right side represents the positive case.

To address the problem of todays application being inherently asynchronous and (therefore/often) developed with the use of multi threading, another abstraction layer is offered. That layer is build upon reactive programming. It's a declarative programming paradigm build on top of functional programming.

Asynchronous programming can result in a so called "callback hell", meaning that the developer must explicitly handle each callback once a asynchronous task completes, which becomes cumbersome once the application and complexity grows and makes it harder to reason about. For this reason there exists a number of solutions in most programming languages, for example "async/await" from Javascript or "coroutines" in the Kotlin world. But they all suffer from the fact that the streams in these frameworks ends after one event.

It's better to have streams that can fire multiple events before closing since it fits better into the interactiveness of modern applications .

This is where the concept of Reactive programming comes in which takes a different approach. It wraps asynchronous tasks (inside of an observable) and treats everything as a stream of data. These streams can easily be combined and transformed by applying functions on this data.

This makes it possible to listen to events instead of a classic polling based techniques to check whether data is available. This behaviour is called the "Observer Pattern" and it follows the Hollywood Principle of "Don't call us, we'll call you".

The state of the application should be observable and the library should provide a solution for asynchronous actions (side effects) out of the box. The developer should work on streams, rather ...

https://gist.github.com/staltz/868e7e9bc2a7b8c1f754

As said by Andreas Staltz **??** on reactive programming:

”Reactive Programming raises the level of abstraction of your code so you can focus on the interdependence of events that define the business logic, rather than having to constantly fiddle with a large amount of implementation details.”

The library must be deterministic and follow the principles of an unidirectional data flow. It will force the developer to create something that comes close to a ”State transition table” and makes him think about every state the application can possibly be in.

* pattern matching unterschuchen * strom von events

* zustandsorientiere Modell -> inwieweit eignen sich reactive und functional? * wie lässt sich ein ereignis orientiertes Modell mit rfp umsetzen? * ist fp nützlich

List of Figures

References

[1] Kevin Ball. *The increasing nature of frontend complexity*. Jan. 30, 2018. url: https://blog.logrocket.com/the-increasing-nature-of-frontend-complexity-b73c784c09ae (visited on 02/04/2019).

[2] Multiple. *Dont Use Exceptions For Flow Control*. Feb. 18, 2019. url: http://wiki.c2.com/?DontUseExceptionsForFlowControl (visited on 02/18/2019).