Hochschule Bremen

Exposè

Media-science Bsc.

Roman Quistler

March 29, 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. The user

But the same applies to the mobile world. Social networks, navigation, sharing and editing files together. The functionality that a application includes today

More functionality does thus in some sense translate into more conditions the application can be in.

Considering a simple login screen feature inside an android application. A user can provide authentication credentials in order to login. It's possible that a user enters wrong credentials, which in turn can be handled by the app by displaying a short description of what went wrong, like an error message. If the user then provides the wrong credentials more than three times, his user account could get locked. This also in some way must be expressed by the application. He could also make use of the "Forgot Password" functionality. That could allow the user to navigate away from the application into a browser window instead in order to reset the password.

| State/Event | Logging in | Logging out | | Logged in | x | x | | Logged out | x | x |

Immediately it becomes clear that there are already four possible states in which the application can be in. A developer thus ends up with states * events possible states which must be considered and handled appropriately.

For example the application might be in the state of "logging in" and then a user tries to transition into the event of "logging in" again, which might not be desirable.

The complexity for such a "simple" or at least standard feature grows linearly with the amount of states and events that are introduced into the application.

With the increasing states and events it becomes harder and harder to have a general overview of all possible states.

Given that a project grows and more and more functionality is added to the codebase, the more important the state becomes and the harder it becomes to introduce state as an afterthought.

In most cases **??**, an application communicates with other interfaces, such as a backend or local database. On mobile devices it's also often necessary to know the GPS status, whether mobile data is enabled or not, etc.

A click of a user thus can cause many changes inside and outside of the application. A developer must be aware of all the possible outcomes, react to them and save what has happened.

By increased application complexity the necessity to make sure the application behaves exactly as expected in all cases increases. This property in software development is referred to a deterministic system, which states **??** that a deterministic model always

produce the same output from a given starting condition or initial state. This is a desirable property because it makes it simpler to predict and reason about events and states in which the application can be within.

The problem of opposite, nondeterministic software is thus that it makes the code less readable since it's impractical to gloss over the code to determine the potential output of a function call.

Code readability becomes a vital part in codebases when they start growing in complexity and size. But also when they are being worked on by more people than just the person who initially wrote it.

productive in writing code instead of deciphering other peoples codes it becomes essential.

A product of readability is another important property which is increased maintainability. Maintainability ties into the notion of not having to rewrite parts of the codebase that is not directly related to the functionality being added or modified.

The arguably most well known remedy for this problem is the SOLID principle, first proposed by **??** in his paper **??** published in the year 2000.

* Single responsibility principle * Open-closed principle * Liskov substitution principle * Interface segregation principle * Dependency inversion principle

Thus far it can be summaries the problem to be that of maintainability, flexibility and readability. These are basic concepts that unless respected have a domino-like side-effects

And core to this problem is managing the applications state in a way that it can survive platform restrictions like for example unexpected process death or in android specifically a configuration change ie device rotation.

Once this basic login feature is implemented another problem emerges which is to add more functionality like for example a 'protected view' which requires the user to be logged in this view should not be reachable from the 'logged out' state without transition into 'logged in' state first.

Side-problems solved by solution to main problem

* Nondeterministic * Closed for extension, open for modification * Unreadable code * Unmaintainable code

The developer must thus manage the state. The problem is to find a good design pattern, which solves the state problem by not treating it as an afterthought.

in such a way that it respects SOLID/DRY etc

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.

mention reactive programming

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.

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).