Lappeenranta University of Technology
School of Engineering Science
Software Engineering
Master's Programme in Software Engineering and Digital Transformation

Esa Kuparinen

MANAGING APPLICATION STATE AND CONTROL FLOW USING REDUX AND REDUX-SAGA IN A WEB APPLICATION

Examiners: Professor Ajantha Dahanayake
M.Phil. Jarno Tenni

Supervisors: Professor Ajantha Dahanayake
M.Phil. Jarno Tenni

TIIVISTELMÄ

Lappeenrannan teknillinen yliopisto
School of Engineering Science
Tietotekniikan koulutusohjelma
Master's Programme in Software Engineering and Digital Transformation

Esa Kuparinen

Sovelluksen tilan ja kontrollivirtauksen hallinta käyttäen Reduxia ja Redux-Sagaa web-sovelluksessa

Diplomityö

2019

62 sivua, 4 kuvaa, 17 koodikatkelmaa

Työn tarkastajat: Professori Ajantha Dahanayake
FM Jarno Tenni

Hakusanat: Redux, Redux-Saga, sovelluksen tila, kontrollivirtaus, ohjelmistoarkkitehtuuri
Keywords: Redux, Redux-Saga, application state, control flow, software architecture

Redux on JavaScript-kirjasto sovelluksen tilan hallintaan. Redux-Saga on Redux-väliohjelmakirjasto, jonka tarkoituksena on tehdä sovelluksen sivuvaikutusten hallinnasta ja testaamisesta helpompaa. Tämän työn tavoitteena on suunnitella ja toteuttaa skaalautuva arkkitehtuuriratkaisu käyttäen näitä kahta kirjastoa. Arkkitehtuurimalli kuvaa kontrollin virtauksen järjestelmässä. Modulaarisen rakenteen avulla saavutetaan eri huolenaiheiden erottelu. Järjestelmän käyttötapausten mallintaminen sagoina tekee niistä luettavampia ja tukee automaattista testaamista. Arkkitehtuurin käytännön toteutus kuvataan koodikatkelmien avulla.

ABSTRACT

Lappeenranta University of Technology
School of Engineering Science
Software Engineering
Master's Programme in Software Engineering and Digital Transformation

Esa Kuparinen

Managing application state and control flow using Redux and Redux-Saga in a web application

Master’s Thesis

2019

62 pages, 4 figures, 17 listings

Examiners: Professor Ajantha Dahanayake
M.Phil. Jarno Tenni

Keywords: Redux, Redux-Saga, application state, control flow, software architecture

Redux is a JavaScript library for application state management. Redux-Saga is a Redux middleware library that aims to make application side effects easier to manage and test. The goal of this thesis is to design and implement a scalable software architecture solution using these two libraries. The architectural model describes the control flow in the system. Separation of concerns is achieved through a modular structure. Modeling the use cases of the system as sagas makes them more readable and supports automated testing. The practical implementation of the architecture is described through code listings.

ACKNOWLEDGEMENTS

The practical side of the work related to this master's thesis has been done for LeadDesk Plc. I want to thank everybody who has supported me during the project and helped me finish the thesis. Thanks to my colleagues and supervisors at LeadDesk for their ideas and feedback. Especially the peer support of Timo Kiviharju has been invaluable during this thesis project.

Special thanks go to Henna for her loving support over the past months.

Lappeenranta 18.10.2019

Esa Kuparinen

TABLE OF CONTENTS

LIST OF SYMBOLS AND ABBREVIATIONS

API	Application Programming Interface
DBMS	Database Management System
HTTP	Hypertext Transfer Protocol
JSON	JavaScript Object Notation
IDE	Integrated Development Environment
LLT	Long Lived Transaction
SQL	Structured Query Language
UI	User Interface

1 INTRODUCTION

1.1 Background

One of the most important attributes of a software codebase is the readability and understandability of the code [1]. Readability can be defined as consisting of a clear project structure as well as a clear structure in each individual code file. When we are talking about a software project that lasts for many years and has a large amount of developers working on it the readability of the code will have a substantial effect on the speed and quality of the development effort. It has been estimated that the ratio of time spent reading versus writing can be over 10:1 [2]. Developers are constantly reading old code when they are developing new code.

Managing the application state and control flow properly has a huge impact on the readability and by extension also on the perceived complexity of a piece of software [3]. If the codebase and the flow of control in the application is easy to follow and understand for the developers, then developing new features or fixing existing bugs will be faster and less error prone. Architectural complexity has detrimental effects on development productivity [4]. Reducing the complexity of the software will also definitely help alleviate the cognitive burden on the developers allowing them to be more productive as well as less frustrated in their work [5]. And of course, all the time saved on development work can be directly mapped to cost savings as well.

As agile software development methodologies are more widely applied and continuous delivery has become commonplace, there is an increasing need for continuous testing as well [6]. It is important to be able to automatically test as many parts of a system as possible. A proper architectural solution should facilitate and support testability as a primary concern.

New technologies, techniques and tools are constantly being introduced for handling common challenges in the field of software development. So, most of the time there will be a very limited amount of empirical research available, if you wish to start using the most modern libraries for JavaScript, for example. In the company where this thesis work was carried out, a part of their product was about to be re-written and the relatively new (first version introduced three years ago) Redux-Saga library was taken into use. Redux-saga

introduces a new kind of a pattern for handling application side-effects and flows. In this thesis a case study about the possibilities, benefits and drawbacks of the new technology will be done. Mostly the focus of the study will be on implementing the state management and application logic control flows.

1.2 Goals and delimitations

The goal of the thesis work is to find a scalable architecture solution for a modern Redux-based web application. Redux will be used for managing the application state changes and Redux-Saga will be used as the tool for modeling the application control flows. These two libraries will be the main building blocks of the architecture solution. The requirement for scalability comes from the experience of working with a previous implementation for an application in the same domain. It is already known that the application will become complex over time and that it will have a large number of different features. Therefore, a solution that facilitates the easy addition of new features must be found. In other words, the application codebase must remain maintainable even when the functional complexity of the application inevitably increases.

One delimitation for the work is that it has already been decided that the system under development will be using Redux and Redux-Saga for handling the application state management and the related side-effects respectively. Within the boundaries of these technological choices a clean architectural design for the system will be proposed. The thesis will shortly introduce the basic theory behind the libraries and paradigms used in the practical implementation of the architecture proposal. Example code snippets are presented in the later chapters, but the thesis does not teach everything required to write code using the libraries efficiently just based on the information provided by the thesis. That is outside the scope of the thesis and is left for the reader to pursue as a separate venture if desired.

As the thesis work is very limited in scope, there was no possibility to actually get to see how the new architecture would fare when the application gets more and more complex. During the making of this thesis the application is still relatively simple, since only a fraction of the planned full functionality has yet been implemented. So, there is a clear need for further research on the matter.

1.3 Research methodology

In the thesis the following research questions will be answered:

- What kind of a Redux-based architecture would scale well?
- Can Redux-Saga be used for modeling the application level logic and use cases?
- How can the architecture proposal be effectively implemented in practice?
- Can the architecture support automated testing of the system use cases?

The research methods used are literature review and a practical case study. Literature and online sources will be used for studying Redux and Redux-Saga related development and design ideas. Generally applicable good software architecture properties and paradigms will be studied. After formulating a suitable architectural solution, the practical case study will be carried out by implementing the application based on the formulated architectural guidelines.

1.4 Structure of the thesis

Section 2 contains the general background theory and introductions to the libraries used in the practical side of the work. Section 3 describes the main process behind coming up with the architectural design and doing the practical project implementation. Section 4 presents the results: the architecture proposal and an example module from the system with code snippets to illustrate part of the practical implementation. Section 5 contains the conclusions and discussion about the results and possible future research.

2 PRINCIPLES, PATTERNS AND LIBRARIES

2.1 Software architecture

2.1.1 Definition

There is no clear-cut definition for the term software architecture. Often it is used to describe the different structures that exist inside a software system and the principles and patterns used to create and maintain those systems. For example, the components that make up the system and the relationships between those components [7]. Sometimes the definition is broader and includes things such as the product requirements, the computing environment and the post development processes [8]. In this thesis the focus is solely on the design of the inner structure of the application and exclude all other aspects of architecture from the scope of the thesis. From this point of view the peripheral concerns such as the exact problem domain of the application or the UI (User Interface) library or the database server being used are just details that are not really important in the architectural context of the application that is address here.

In this thesis the high-level application components and their relationships will be considered as the main parts of the proposed architectural solution, but also conventions on the lower-level coding practices will be introduced.

2.1.2 Properties of clean architecture

A successful software project must focus on the needs of a customer [9]. There are different ways of modelling out the domain to make sure you are truly developing something that produces value for the customer. For example, domain-driven design focuses on designing and maintaining an evolving model of the domain that is then used as a basis for the software implementation [10]. Good architecture should be one that enables the use of different domain models and supports the developers in implementing the required features and use cases. In other words, the architecture should provide a proper way for handling pieces of the application logic and use case workflows.

But getting a piece of software working is not the hard part. “Getting software right is hard.” This quote is from the book Clean Architecture written by Robert C. Martin. The principles of clean architecture presented in the book are highly useful and universal. According to the book, a good architecture will support the development, deployment, operation and maintenance of a software system. By this definition software architecture actually has very little to do with whether the system works properly. You do not need a good architecture to get a system working. You need it to support the life cycle of the system. Good architecture makes the system easy to develop, easy to maintain and easy to understand. Ultimately minimizing the lifetime cost of the system and maximizing developer productivity. This thesis focuses mostly on the ongoing development and maintenance aspects of the system lifecycle. [11]

Figure 1 displays a diagram that portrays the main idea behind the clean architecture pattern. The circles represent different levels of the software – the innermost circle being the highest level. The outermost circle is the lowest level of the software system which, by definition, manages the inputs and outputs of the system. In other words, the farther a policy is from the inputs and outputs, the higher its level. At the heart of the pattern is the dependency rule. Dependencies must always point inward, toward the higher levels. An inner circle must never know anything at all about the contents of the outer circles. At code level this means that the name of a software entity declared in an outer circle must never be mentioned in the code of an inner circle. Any data that passes the layer boundaries must not communicate any details about the outer circles inwards. The passed data must be in the format that is most convenient for the inner circle, so the format should always be defined by the inner circle. [11]

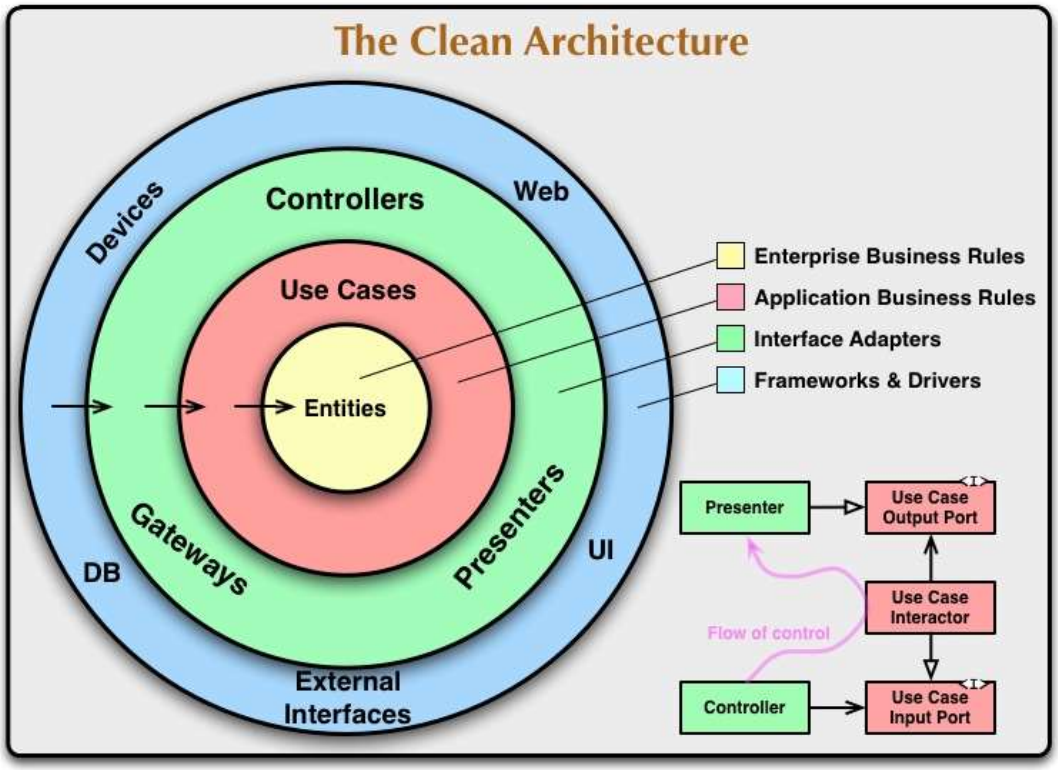

Figure 1. The clean architecture. *[12]*

Entities at the highest level describe critical business rules that are applicable to the whole enterprise. They are the high-level policies that usually are key to the business and make money for the company. Entities are the most stable part of the software system and are the least likely to change because something external changes. Many different applications can use the same entities. [11]

The use cases layer implements all the use cases of the system. They are application-specific business rules that describe how an automated system is used. Use cases specify the inputs they receive, the outputs that they produce, and the processing steps required in producing those outputs. Use cases handle the interaction between the users of the system and the entities. But use cases do not describe what the user interface is or how it looks like. They operate on data coming in from the other levels of the system, but should have no idea how that data is actually gathered from the user or how the output will be presented to the user. The input and output data models should be defined by the use cases themselves so that they remain independent. Changes in this layer should not affect the entities. And by the same

token, changes in the outer circles of the architecture (like the database or the UI) should not directly affect the use cases. Only changes in the entities or in the use cases themselves should prompt modifications to the code in this layer. [11]

The interface adapters layer consists of a set of adapters that convert data from the format most convenient for the entities and the use cases, to the format used by the outer layer. And vice versa when the layer is receiving data from the lower level. For example, if the system is using an SQL (Structured Query Language) database as the persistence framework, then all the SQL should be restricted to this layer. The code in the inner circles should know nothing at all about the database being used. And the same goes for all the other details in the outer layer. [11]

The frameworks and drivers layer is the outermost layer of the software system. This layer contains concrete details like the database, the user interface and the web framework. Usually this layer will be quite thin and should not contain a lot of code other than the code absolutely necessary for communicating to the next circle inward. Because none of the inner circles know anything about these details, the drivers and frameworks can be easily changed without causing modifications to cascade to the higher levels of the system. That is important because most of the time the lowest level is also the most volatile in the sense that is has many reasons to change in the lifetime of the software system. [11]

In a real-world software system, you might end up having more than just the four layers described here. The number is not important, but the dependency rule must always apply. Inner circles must never depend on anything in the outer circles. Adhering to the clean architecture will also keep the software testable, which can be said as being characteristic of a good architecture. Different levels of the system remain separated and can be tested in isolation. [11]

One important principle of good architecture is always leaving as many options open as possible. That way you can delay making certain decisions until you have more information available on what to base those decisions. High-level policies should be the most essential part of the system and the lower-level details should be kept irrelevant to them. Then you can defer decisions about those details for a long time. Keeping options open will also help

with the inevitable changes in the constraints and requirements of the system by making the system easier to be modified, since you are not committed to too many details. [11]

2.2 Application state management

Proper state management is vital for keeping track of all the data in your application. Failing to do so will most likely result in some kind of problems during the development of the software. Some of the most common problems are issues with duplicate and out-of-sync data. Letting the application to get into that kind of a state will increase the effort it takes to maintain the software and it will most likely also introduce bugs along the way. It is hard work to try to manage multiple instances of a single piece of data and developers – being human – tend to miss some of the instances when making modifications to the codebase. Worst case scenario is that you might end up with presenting or storing wrong data if there are multiple instances of the supposedly same piece of data. You can keep your application much simpler if you take care and make sure there is a single source of truth for the data. Luckily there are patterns and libraries for doing just that. [13]

2.2.1 Flux

Flux is a design pattern that introduces the idea of one-way data flow for managing application state [13]. ***Figure 2*** displays an example data flow utilizing the Flux pattern and introduces the different entities and components that are part of the pattern.

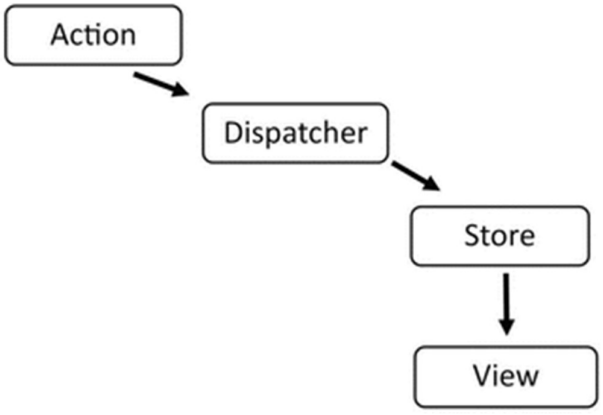

Figure 2. Basic data flow example in Flux. *[14]*

Actions are pieces of data that originate from the internal layers of the system or from the view layer that corresponds to the user interface of the application. They contain the data that is used to update the application state. In the Flux pattern, the only way to change the application state is to create an action. Usually the actions are objects that have a type and a payload. The data contained within an action could be something as trivial as indicating a certain button has been clicked on the user interface. In that case you don't generally even need a separate payload property in the action object since the type of the action already tells what happened. Or the action could contain some other data input by the user or an internal event from the system, in which case you could put the necessary information in the action payload property. [14]

After creation the actions are passed on to the dispatcher, which is a singleton object that makes sure all the actions are handled synchronously in the order that they have arrived. This prevents all kinds of race conditions from happening. The dispatcher is responsible for relaying the action to all the stores that have registered themselves with the dispatcher and are listening for actions of the particular type. [14]

The store layer can contain multiple stores in Flux. Usually those would represent different subdomains within the application. The stores process the received actions and update their internal state according to the application logic rules contained internally within the stores. So, the stores actually contain the state of a Flux application. And when that state changes the stores broadcast an event so that the views can then fetch the new state and re-render the application UI as needed. [14]

2.2.2 Redux

Redux is a JavaScript library that implements the basic ideas of the Flux pattern, while also keeping the structure simple by enforcing the use of a single store for all the data and having the dispatch functionality integrated into the store itself. There are three fundamental principles that Redux is based on: [15] [16]

- "Single source of truth" means that there is a single store where the state of your entire application is stored as an object tree. This approach has many benefits,

including easier debugging and tracking of the application state, since it is centralized in one place. You can also for example serialize and save the application state easily and then restore the same state later. [15] [16]

- "State is read-only" refers to the fact that there is no way to modify the state by directly writing to it. The only way to change the state is to dispatch an action. Like already mentioned in the chapter about Flux, actions are objects that express an intent to mutate the state. Enforcing the use of actions makes monitoring the state changes easier. A certain state can always be deterministically reproduced by dispatching the same sequence of actions given a known initial state. All the actions can be logged, stored and replayed to achieve "time traveling" powers in a sense where you can move back and forward in the timeline of the dispatched actions. This opens a lot of possibilities for creating useful development and testing tools. [15] [16]
- "Changes are made with pure functions." Redux introduces a new concept to the Flux pattern and that is the reducer function. The state tree contained inside the store is only ever changed by executing the available reducers. Usually each branch in the state tree is handled by a separate reducer function. The reducers are just functions that take the previous state and an action object as arguments. Based on those arguments the reducers return the new state value that will be stored in the store. So, in Redux reducers actually contain the application logic rules that determine what the application state will look like after an action has been processed. The reducer functions should not mutate the previous state they receive as an argument but should always return a new state object instead. Reducers must be pure functions in the sense that they do not have any side effects and will always return the same value when given the same parameters. That makes it easy to write tests for the reducer functions as well. [15] [16]

Figure 3 displays the components that are commonly associated with a Redux controlled data flow. In short, action creators are functions that are used to create the action objects which are then dispatched to the store. In the store you can have middleware installed that will process the action before it reaches the reducers. And the reducers use the action object to produce a new state for the application. Finally, the store will inform all registered listeners about the change in the state. Please note that many times the component that initially called the action creator will also be among the registered listeners that will receive

notification about the updated state in the end. That way it can refresh itself according to the new application state. [16]

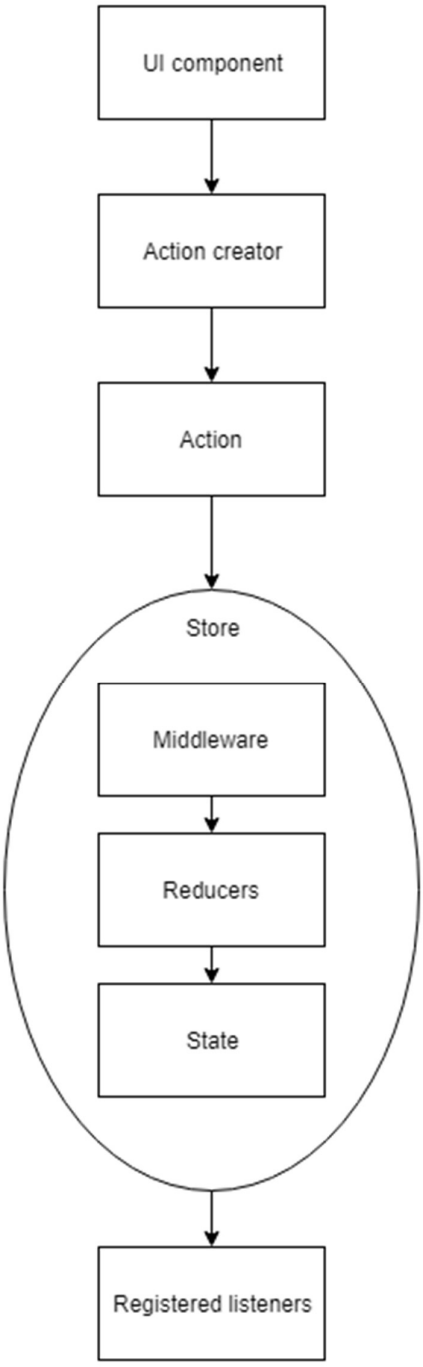

Figure 3. Redux data flow example.

2.3 Sagas

The term “saga” and the general idea behind sagas was first introduced by Hector Garcia-Molina and Kenneth Salem in their conference proceeding paper from 1987 [17]. They proposed sagas as a control mechanism for handling LLTs (Long Lived Transactions) in the context of a DBMS (Database Management System). Traditionally LLTs caused serious performance issues, since they had to be atomic actions and would cause the DBMS to lock the objects accessed by the transaction until the whole transaction would be completed. That meant the other transactions waiting to access the same objects would get delayed substantially and there would be drastically more potential for deadlocks and abortions. [17]

However, in many applications this kind of a rigid control is not necessary and there the saga pattern comes into the picture. The term saga refers to an LLT that can be split into a sequence of sub-transactions that can be interleaved with other transactions. Each sub-transaction is treated like a proper atomic transaction and it will leave the database is a consistent state. The sub-transactions in a saga are closely related to each other, but the saga itself should be executed as a non-atomic unit. Partial executions of a saga are undesirable but will eventually happen and there must be a mechanism for amending partial executions. Therefore, each saga transaction should have a compensating transaction defined that will be executed if that particular transaction needs to be undone. That scenario could occur when the whole saga is aborted or when just that single transaction is aborted. If the whole saga gets aborted, all the already executed transactions must be compensated for by running the corresponding compensating transactions in reverse order. [17]

The saga pattern also supports the idea of parallel transactions. Often some of the transactions in a saga can be executed concurrently. Then there will be a parent saga that creates new child processes to be executed in parallel. The failure and abort recovery mechanisms are a bit more complicated in that case but they are still very well manageable. [17]

2.3.1 JavaScript generator functions

Generator functions were introduced in the ECMAScript 2015 (ES6) specification. They are a mechanism for asynchronous programming in JavaScript code. Other popular techniques are promises and the async/await pattern. They all try to solve the issue of writing asynchronous code in JavaScript. Traditionally, asynchronicity is handled by using callback functions but that quickly leads to something often referred to as "callback hell", since code with a lot of callbacks is difficult write and understand. Also handling errors in a long callback chain is quite tricky. [18]

JavaScript generator functions are declared using a special syntax that is just slightly different from regular function declarations. While a normal function can be declared using the keyword **function**, generators are declared by appending an asterisk to the end like so: **function***. ***Listing 1*** shows an example of a simple generator function declaration and how it can be used. [18]

Listing 1. Example of a generator function. *[18]*

```
function* generatorSequence() {
    yield 'first';
    yield 'second';
    yield 'third';
}

let generatorObject = generatorSequence();

generatorObject.next().value; // the value is 'first'
generatorObject.next().value; // the value is 'second'
generatorObject.next().value; // the value is 'third'
generatorObject.next().value; // the value is undefined
```

When you call the generator function it will return a generator object. That object is an iterator that has a function called **next** for getting the next value in the generator sequence. Calling the **next** function will result in the generator function's body to be executed until the first **yield** expression. The **next** function will return an object that has a value property that will contain the value of that **yield** expression. The interesting thing here is that now the generator function's execution has halted, and it will not continue until you call the

next function again. Then the execution continues from the point where it last ended (in this case, after the first **yield** expression). So that is why the second call to the **next** function on the generator object will return an object with the value of "second" and so on. Until we reach the end of the sequence defined in the generator function that is. In the listing above you can see that the fourth call made to the **next** function will actually return an object with the value of undefined. To be exact, the object returned looks like this: {value: undefined, done: true}. This is related to the iterator protocol that the generator object implements. Every time you have not yet reached the end of the sequence the returned object will look something like this: {value: 'first', done: false}. So the **done** property is there to signal that the generator function has finished and will not yield any more values. [18] [19]

2.3.2 Redux-Saga

Redux-Saga is a Redux middleware library for handling application side effects (e.g. asynchronous data fetching). Since Redux works completely synchronously there is usually need for a middleware that can manage all the asynchronous things in your Redux application. The mental model behind Redux-Saga is somewhat similar to the sagas discussed earlier, but in this case the sagas are executed in the context of a Redux application. Therefore, they mainly communicate with the application through Redux actions and also have access to the application state stored in the Redux store. The Redux-Saga library uses generator functions to manage the asynchronous flows of the sagas. This results in the sagas being easy to write and read, since the code looks almost like standard synchronous code. The sagas provide a clean way to handle logical control flows inside your application and also act as a failure management system, since you can declare all the necessary steps to recover from a failure or a cancellation right there is your sagas. Another huge benefit of using this middleware is that the sagas are actually very easy to test. [20]

Listing 2 shows an example of a simple saga to illustrate some of the basic concepts. There would also have to be some boilerplate code that would apply the Redux-Saga middleware to the Redux store and run the root saga, but those are omitted for the sake of simplicity in this chapter. Let us just assume that the root saga is running the **watchFetchProducts** saga when the application starts. This saga is responsible for fetching a list of products from an API endpoint. What usually triggers the execution of a saga is a Redux action being

dispatched. In this example you can see the watcher saga called **watchFetchProducts** is waiting for an action that has the type "PRODUCTS_REQUESTED". When this type of an action is dispatched as a result of some user activities on the UI for example, the saga will come to life. The **takeEvery** helper function provided by the middleware will catch that action and invoke the **fetchProducts** worker saga provided as an argument. In the **fetchProducts** saga the execution will reach the **yield** statement that makes an asynchronous call to the API using another helper function called **call**. After that the saga just waits for a response in the background. When the API call is finished the saga continues execution and the response will be stored into the **products** constant and then logged to the console. [20]

Listing 2. Simple saga example. *[20]*

```
import { call, takeEvery } from 'redux-saga/effects'
import Api from './path/to/api'

function* watchFetchProducts() {
  yield takeEvery('PRODUCTS_REQUESTED', fetchProducts)
}

function* fetchProducts() {
  const products = yield call(Api.fetch, '/products')
  console.log(products)
}
```

Listing 3 shows how easy it is to write tests for the sagas. What makes so simple testing possible, is the fact that in Redux-Saga the generator functions actually just yield plain JavaScript objects whenever you use the provided helper functions such as **call**. Those objects are called "effects" in the Redux-Saga terminology. The middleware will then take care of handling the yielded effects, perform any necessary actions and give the results back to the generator function. This way the code actually becomes declarative and that makes testing easy. You can in fact re-use the same helper functions in your test code and just do equality assertions to the effects (plain objects) that they return. In this example we are testing that the saga makes the appropriate API call, but still we don't need to resort to mocking the API or anything. If you would be using some other middleware such as Redux-Thunk, you might probably have to mock the API to be able to test you code and that would make testing more difficult and the tests less reliable. [20]

Listing 3. Testing a saga. *[20]*

```
const iterator = fetchProducts()

// expects a call instruction
assert.deepEqual(
  iterator.next().value,
  call(Api.fetch, '/products'),
  "fetchProducts should yield an Effect call(Api.fetch, './products')"
)
```

There are also more advanced functionalities you get when using Redux-Saga. Pulling future actions is a really powerful one. The **takeEvery** helper function you saw earlier is very naïve since it will trigger a function every time the particular action is dispatched. Usually you will want more control over your sagas and make them react to actions only when it is appropriate. Another benefit of pulling actions is that you can declare the whole control flow in one saga, and it will be obvious how different steps in the flow are related to each other. ***Listing 4*** demonstrates how to use the **take** helper function to pull actions and declare the expected action sequence when considering a login flow found in many applications. The **take** function will block the generator function until the proper action is dispatched and then it will resume execution. The saga clearly shows the order in which the actions must occur and it is easy to understand how these actions and steps in the flow are related to each other. [20]

Listing 4. Pulling actions to form a clean flow. *[20]*

```
function* loginFlow() {
  while (true) {
    yield take('LOGIN')
    // ... perform the login logic
    yield take('LOGOUT')
    // ... perform the logout logic
  }
}
```

Other advanced concepts include executing parallel sagas, composing sagas and cancelling tasks. Redux-Saga provides helper functions for forking sagas and also cancelling the forked tasks when needed. There are helpers for starting multiple sagas simultaneously and continuing the parent saga execution when the first child saga is completed (**race**) or when all of them are finished (**all**). You can detect the cancellation of a saga using the appropriate helper functions and declare the operations to perform upon cancellation right there is the

saga body. So, all in all, there are also powerful tools for handling concurrency in Redux-Saga. [20]

2.4 TypeScript

TypeScript is a superset of the JavaScript language. That means that regular JavaScript code is also valid TypeScript code. TypeScript just adds new features on top of JavaScript to facilitate more disciplined coding practices. Those will help in eliminating certain types of bugs in the software, such as type related mistakes that are a common cause of errors, since JavaScript uses dynamic typing instead of static typing. [21]

Some of the most useful features of the TypeScript language are the type annotations and the type inference system. Describing the actual TypeScript syntax is outside of the scope of this thesis so I will just be explaining the features on a more general level and skipping more advanced concepts like type unions and type guards. Type annotations allow the developer to explicitly define what the type of a variable is. For the simplest cases that is usually not even necessary, since the type inference system can actually infer the type from usage. For example, if you are defining a variable and initializing it with a “string” type value, then type inference can do its thing and infer that the variable will be of type “string”. [21]

Variable types are not the only place where TypeScript will help you. Types can also be defined for constants, function arguments, function return values, class members and interfaces, for example [21]. Interfaces in particular are a very useful tool. One of the most common issues you run into while developing is accessing JavaScript object properties. Many APIs (Application Programming Interface) use JSON (JavaScript Object Notation) as the data structure for transmitting data over HTTP (Hypertext Transfer Protocol) requests and Redux uses action objects extensively, so it is really helpful to have a tool that assists the developer when working with objects. TypeScript interfaces and type definitions allow you to explicitly define the properties and the data types of the properties that a particular object will have. If the type-safety is properly implemented the developer cannot mistakenly use a non-existing property name anymore. Or even use a wrong data type when assigning values to the object properties. When you combine that with the intelligent auto-complete

features available in the moderns IDEs (Integrated Development Environment) you get a significant boost in the developer proficiency to write code.

Before you can execute the application written in TypeScript, you must run the TypeScript compiler which compiles the code into plain JavaScript that can be executed by the browsers. This has a couple of important implications. First, none of the TypeScript code that you write will end up in the JavaScript bundle that will be deployed and downloaded by the end users' browsers. So, using TypeScript in your will have no adverse impact on the run-time performance of your code or the bundle size. For all intents and purposes, TypeScript will only exist in the development phase to help the developers and then it will gracefully bow out of the way when it is time to deploy the application.

Secondly, the TypeScript compiler will allow the developers to catch many of the errors they make without having to even actually run the application. If you would be developing using vanilla JavaScript only, then you would need to run the code and test it in practice to catch any errors it might contain. But using the TypeScript compiler you will catch the errors even sooner and save a lot of time and effort. Modern IDEs like Microsoft's Visual Studio Code support installing plugins for TypeScript that will bring this functionality straight into the IDE. The developer will instantly be notified of any possible type errors on the UI which speeds up the process even more.

In our project we use TypeScript to add type-safety to our JavaScript code. I consider it an integral part of the application architecture, since it helps developers greatly in writing maintainable code and avoiding mistakes. All in all, it just enables enforcing a better structure and a higher quality in the code that we write.

3 IMPLEMENTING THE ARCHITECTURE

In this section the practical and empirical side of the research will be presented. The implementation process will be described phase by phase as it happened. The process was an iterative one so this section will also include the refactoring steps necessary to reach the final state of affairs in the scope of this thesis.

3.1 Main design ideas

First the Redux-Saga library and the saga way of thinking had been studied thoroughly. After that the idea was born that the main application logic could be expressed as sagas by using the Redux-Saga middleware for Redux. There were a couple of online articles [22], [23] found what were describing a somewhat similar architecture, but it seems this is not a very widely adopted approach. It just seemed intuitive because sagas are naturally able to describe flows and therefore, they would be well suited for expressing the use cases of the system. After all, a use case is generally described as a sequence of steps to produce the desired outcome.

Using sagas to model the use cases of the system yield also other benefits. Code readability and by extension also the maintainability increases if the use cases can be expressed as seemingly synchronic sequences of actions and contained in a single module or function. Asynchronic code is notoriously difficult for developers to understand when using traditional asynchronic programming techniques like callbacks. Even though the generator functions are actually working asynchronously the code looks like it is synchronous, and it is generally easier for the developers to read. Also when the whole use case is contained in a single saga, it becomes a lot easier to see everything that the use case does by just looking at that saga and the code contained within. There are many software systems in which the use cases are not clearly modeled in any reasonable way and are split into many different components in the system. Then it becomes very laborious to even figure out how the particular use case works in the system and what it does. Let alone go in and modify it in some way. Tracking the impact of any change becomes a chore.

Not only do the sagas make it possible to encapsulate application-specific business rules and use cases, but in doing so they also allow for keeping the user interface at the lowest level of the system very lean. Also using Redux to store and handle the application state frees the UI components from having to maintain the state themselves. You don't need to have a lot of code in the user interface layer if you have a higher-level layer that contains the application logic and handles the state. This is a good thing because the user interface is also very volatile in the sense that the code related to that will change relatively often and for many different reasons. Now the impact of those changes will be contained only to the presentational aspects of the system and they cannot affect the application logic in other ways. It becomes easier to change the user interface safely. The user interface can focus on the presentation and input concerns and those remain completely isolated in the UI components.

Testability is another big reason for using sagas. If the use cases are implemented in a way that they are not fully contained or encapsulated in any particular module or piece of code in the system, then it effectively becomes impossible to fully test them automatically. The Redux-Saga middleware has been designed with testability in mind and there are many test libraries for it. So, if the use cases are expressed fully using sagas then they can also be automatically tested. That has a huge impact on the ability to modify and refactor the software as you can catch errors very quickly during development. And every developer who has worked in the industry for more than a couple of years will surely know that the only constant thing is change. The requirements of the system will change and refactoring of the code will happen frequently.

3.2 Initial project structure

The implementation of the first version of the system was preceded by a study into the best practices of Redux application design. After reading through a set of online articles it was decided to go with a pretty simple setup for the first version as not succumb to over-engineering. It was clear that the process would be iterative, and the architecture would have to support refactoring, so there was no need to make any binding decisions for the long term.

Here is what the folder structure of the project looked like (simplified not to contain all the actual use cases):

- components/
    - Dashboard.tsx
    - Settings.tsx
- redux/
    - actions.ts
    - reducers.ts
    - sagas.ts
    - selectors.ts
- storage/
    - users.ts
    - session.ts
- types/
    - users.ts
    - session.ts

The structure clearly covers all the basic parts of a Redux application. There is a place for the action object definitions and action creator function definitions in the redux/actions.ts file. All the reducer functions are defined in the redux/reducers.ts file and sagas in the redux/sagas.ts file. The storage/ folder has APIs for accessing the storage layer implemented by the backend server. And the types/ folder contains the TypeScript types related to different classes and objects in the system. Purposefully no attention is being payed to the components/ folder, because it contains the UI components and those are not relevant for the architecture proposal presented here. The folder is there just for illustration purposes.

The redux/selectors.ts file concerns something we have not discussed before. Selectors are not really mentioned in the Redux documentation and are missing from many other sources as well but are still widely [24], [25] considered as one of the key concepts of the Redux application architecture. Selector functions are used to access the state data in the store. Selectors allow you to keep the data in the store in a normalized form and still have easy-to-use getter functions for accessing the data in the UI components, for example. Another big

benefit you get from using selectors is that they decouple the components using the state data from the actual shape of the state. This means that even if you refactor the state structure in the store, you only have to make changes to the selector functions. All the components using the selector functions can remain untouched. This gives you a lot of freedom to improve the state shape to suit the needs of the application along the way. And if you experience performance issues with selector functions that get called very frequently, there are third party libraries for implementing simple cache functionalities for the selectors. One example of such a library is "reselect". [26]

After having worked with this structure for a while and having implemented a couple of new features in the system, it quickly became apparent that the structure did not scale well at all. The files containing the actions, reducers and sagas started to grow rapidly. Also, they were containing code from completely different use cases and domains that were not at all closely related to each other. Every change could have impact on other totally unrelated use cases as well. And so, the first refactoring needed to happen. The folder structure was changed to this:

- components/
    - Dashboard.tsx
    - Settings.tsx
- redux/
    - actions/
        - users.ts
        - session.ts
    - reducers/
        - users.ts
        - session.ts
    - sagas/
        - users.ts
        - session.ts
    - selectors/
        - users.ts
        - session.ts

- storage/
    - users.ts
    - session.ts
- types/
    - users.ts
    - session.ts

Now the large single files containing all the actions, reducers, sagas and selectors had been split into separate files - one for each use case or domain in the system. This change made the code files more manageable by drastically limiting their size. The structure became more scalable since we could now scale horizontally instead of vertically by adding new folders when new use cases were implemented in the system.

However, after a while it came time for another iteration in the structuring of our application. While the structure was already pretty scalable, there were issues that made working with it a bit tedious. For example, if you were modifying some functionality related to the session use cases, as a developer you would probably need to change one file in the actions folder, another one in the reducers folder and yet another one in the sagas folder. As the folders grew to contain more and more files, soon it became difficult to find all the places you needed to modify even though you were just working on one single use case.

Also, if you just take a look at the file names in each of the folders, you quickly start to see a pattern (which is even more apparent in the simplified listings that do not contain all the use cases our application actually had at the time). Clearly there is another way to structure the project that would keep related files closer to each other.

3.3 Refactor into modular structure

The modular structure was inspired by the “ducks” pattern that is often proposed to be used with Redux [27]. The pattern aims to separate the modules by domain and contain everything related to a specific domain or use case in the same module. It results in a structure that is more logical and easier to maintain, since typically as a developer you are only working on a single use case at a time and can find all the related components in the same module.

This is the kind of folder structure that was selected for the application:

- components/
    - Dashboard.tsx
    - Settings.tsx
- redux/
    - modules/
        - users/
            - index.ts
            - actions.ts
            - reducers.ts
            - sagas.ts
            - storage.ts
            - types.ts
        - session/
            - index.ts
            - actions.ts
            - reducers.ts
            - sagas.ts
            - storage.ts
            - types.ts

With all the details related to a certain use case neatly tucked inside a single folder, the structure becomes easy to follow and it is descriptive of the application we are developing. Just by gazing at the directory tree, one can quickly see what kind of use cases our system contains and get a good idea of what the application is all about. This is called “screaming architecture” in the Clean Architecture book. The structure is also very nicely following the component cohesion principles related to the clean architecture pattern. The book presents a “common closure principle” for software components that can be summarized as: “Gather together those things that change at the same times and for the same reasons. Separate those things that change at different times or for different reasons.” [11]

Changes to use cases are now easier to manage, since often the developer just needs to operate inside a single folder. And when the changes are confined to a single module, the impact of those changes is greatly reduced. Many times, only that one module is affected, but also in the worst case the impact can be narrowed down to only those other modules that have dependencies to the modified module.

At the code level this structure also creates a namespace for everything inside the module. So, you can just use simple names when working strictly inside the module, which should be the case most of the time. And if you need to access the module internals from another module or any other outside file, it is always possible to use aliases to avoid any naming conflicts. This modular structure brings a bonus benefit of documenting the dependencies between the modules in the form of the JavaScript import statements. You can always clearly see what modules a certain component is using and that might help you in finding hidden correlations between the modules or just spot errors in the dependencies. That information can be used to improve the project structure further.

Like most things in software development, designing the modules is an iterative process. Most likely you will not get everything right on the first try, so you must not be afraid of fixing things if you see reasons to do so. Hopefully the architecture will support you along the way.

4 RESULTS

In this section the final proposal for the architecture will be presented. First, the created architectural model will be described and then practical examples of the way it was implemented in the system at the code level will be showcased. Also, the automatic testing and the related benefits achieved by using the architecture will be described.

4.1 Architecture overview

Figure 4 presents the high-level architecture of the system and the control flow between the different components. At the highest level the system can be divided into three distinct layers that have their own separate policies and rules. The user interface logic layer should only be responsible for the presentational logic and otherwise the UI components should be very lean. The knowledge about the application business rules should not be leaking into the UI layer. So, the UI components may contain any logic that is necessary for presenting the application data to the user, but in general they should be stateless and not store their own state. Everything rendered on the screen should be based on the contents of the application state that is kept in the Redux store. That way it is possible to achieve deterministic view renders, which means that the UI will always be rendered similarly, given the same state. For example, it makes things simpler regarding testing, since you can trust the UI components to always behave similarly when the state is the same. There is no need to worry about race conditions that might disrupt the view rendering based on some asynchronous requests completing in different orders. [28]

To be exact, the user interface acts both as the input and the output for the users of the system. I covered the output aspect in the previous paragraph, but of course we must also acknowledge the input concerns. This side of the UI components will have very little code or logic. Actually, the only thing the UI components should do regarding user input, is to call action creator functions to dispatch actions to the Redux store. No other logic is really needed in the UI layer. In this architecture the application logic layer will contain all the policies and rules required to handle those user interface originated actions.

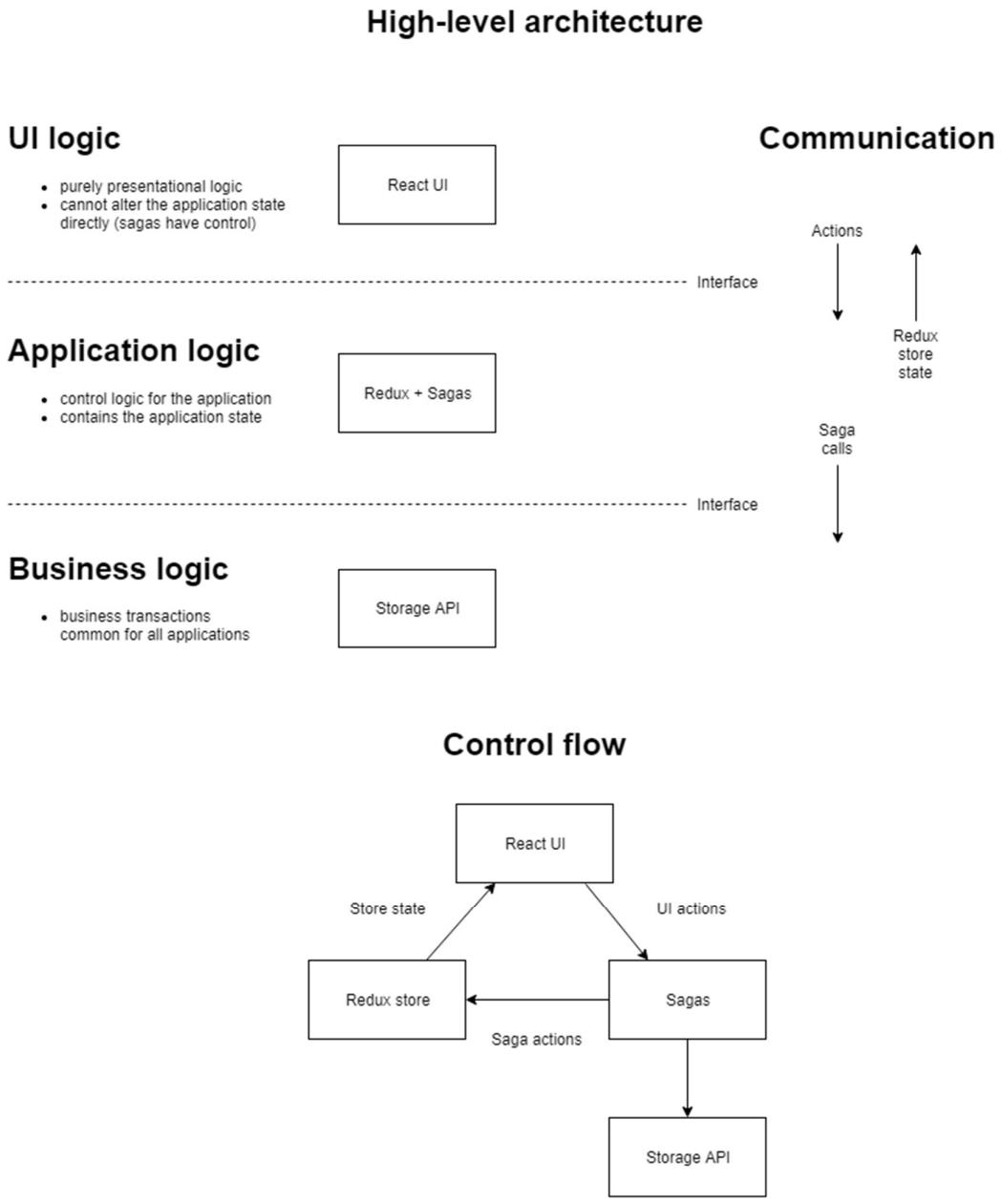

Figure 4. High-level architecture and control flow.

The application logic layer contains all the application level business rules and maintains the application state. In our case this layer is implemented by using Redux for managing the application state and Redux-Saga for the use cases and control flows. All the regular Redux related things are defined and used here: actions, action creators, reducers and selectors. The

reducer functions are the smartest part of the Redux state management as they control how each action modifies the application state. But in this architecture the sagas do most of the heavy lifting and contain most of the application logic. The use cases of the system are modeled as sagas, so they are embodiments of the application business rules and policies.

But there is also another very important responsibility that the sagas have. They are always in control of the application. By that I mean that nothing really happens inside the application unless a saga says so. In other words, the application state will never change unless a saga dispatches an action to change it. This concept is key in achieving true encapsulation and testability for the use cases of the system. If you make sure that the control always remains in the sagas, you will have a single place where you can find all the logic related to a particular use case. It is nicely contained in the sagas. It is encapsulated. It is easy for developers to find, read and understand. It is easy to test in isolation. It is possible to effectively and continuously verify that the whole use case works by using automated unit and integration tests.

There are systems where the application logic and control flow jump around from UI components to libraries and from there to asynchronous callbacks following API requests and then back to the UI components and so on… In that kind of an architecture it will become next to impossible to track the control flow and understand what is happening or what will happen next in the application. Also, the application state will most likely be a combination of the UI component states and a random collection of other variables in the system. In that case it is very difficult to determine the state that the application is in. And it will be impossible to test any single use case as a whole if the related logic is split into different components in the system.

Then how can you make sure the sagas are always in control? The answer lies in the control flow diagram in ***Figure 4***. Notice the unidirectional arrows in the diagram. The control flow is actually pretty similar to the data flow in Redux. Using this kind of a unidirectional flow keeps the responsibilities of the different system components and their relationships simple and clear. In this architecture you will have two fundamentally different types of actions. There are user interface originated actions and saga originated actions. And both have completely different purposes. The UI actions are dispatched to signify some event

originated from the user interface that might be interesting to the sagas. The sagas then process the UI actions and act according to the application business rules and policies modeled in the sagas. The sagas may do some processing, do requests to the storage APIs or dispatch actions of their own. Notice that the UI actions are only processed by the sagas and never the Redux store (i.e. the reducers). As a result, the user interface originated actions cannot directly change the application state. Only the actions dispatched by the sagas will be processed by the reducers and can affect the application state managed in the Redux store. This way the sagas retain control. The application state cannot be changed by anyone else. No processing will happen unless the sagas initiate it. Even though the initial signal to start some workflow may come from the user interface in the form of a UI action, it is always the sagas who decide whether that action will cause any kind of reaction in the application.

In the architecture the sagas are higher level components and treat the UI components as a bit untrustworthy. This reasoning boils down to the fact that the user interface is the one of the most volatile components of the system. Also, the user inputs cannot really be trusted, and they must be sanitized somewhere. In light of this knowledge it is beneficial to keep the user interface as simple as possible and not have the input sanitization logic in there. Then the UI components are as easy to change as possible. And in the end, the sagas should be responsible of the input sanitization and that means not blindly following every action received from the UI components, but deciding on a case by case basis how to react to each action based on the current application state. This can mean ignoring some user interface originated actions completely in some cases if the application is not in the right state to handle those. Or it can mean throttling the action handling i.e. only allowing one action of a certain type to be processed in a certain amount of time and ignoring other similar actions dispatched during that timeframe. Redux-Saga has ready-made helper functions for throttling and debouncing actions, for example.

The business logic layer in ***Figure 4*** is actually at the highest level in the architecture. The enterprise business rules are contained there. Those are policies that are reusable and applicable to all the applications in the enterprise. The storage API implements the business transactions that are common for all applications.

You may have also noticed some other details in the figure, such as “React”. Yes, React is being used for implementing the user interface in the application. But purposefully no attention has been paid to that fact, because it is just a detail that is not really relevant to the architecture being described here. This architecture aims to remain independent of and indifferent to the user interface library or framework used. That detail is not important to the operation of the system and because of that it can also be easily changed as needed. This provides great flexibility and always allows for choosing the proper tools for the job.

4.2 Implementation details

In this chapter a single module from the system will be described in more detail. That should give the reader real examples of what the proposed architecture would look like when implemented in practice. This information will be especially useful for developers or architects who wish to implement a similar architecture in their projects.

4.2.1 Example module implementation

This example module is called “ProgressiveCalling”. It handles the logic of automatically calling through a list of contact details stored in the system. This functionality allows for contacting a large amount of people effectively by utilizing phone calls. In the following paragraphs and listings every single file from this module will be explained thoroughly.

Each module will contain an index.ts file. It is used as the single point of access for the module. The index.ts file allows the other modules or components access to the module internals by just importing from the module folder directly. ***Listing 5*** displays a typical index.ts file for a module. The module internals are exported there and usually there will not be much else going on.

Listing 5. Example index.ts file.

```
export * from './actions';
export * from './sagas';
export * from './reducer';
export * from './types';
export * from './storage';
```

Listing 6 shows what the types.ts file looks like in this example module. The first line is immediately a very interesting one, since it presents a very useful third-party library [29] you can use to make sure you don't accidentally mutate the state object in your reducer functions. In particular, you should import the **DeepReadonly** type from that library and always wrap the type definition of your state object using that type. That will cause the TypeScript compiler to warn you if your reducer functions are trying to mutate the state object in any way. Mutating the state object would be against the rules of Redux.

Listing 6. Example types.ts file.

```
import { DeepReadonly } from 'utility-types';

export type ProgressiveCallingState = DeepReadonly<{
    mode: Mode;
    started: boolean;
}>;

export const enum Mode {
    Auto = 'auto',
    Semi = 'semi',
}
```

Continuing on with ***Listing 6*** reveals the TypeScript type definitions for the module. The **ProgressiveCallingState** type defines the structure for the state object related to this module. In this case it is just a mode identifier and a boolean flag telling if the progressive calling has been started or not. The different modes available are defined as an enumeration below.

Listing 7 shows the reducer.ts file for the module. The **defaultState** constant defines the initial state for the module store. That is applied when Redux first creates the store and calls all the reducers for the first time to initialize the store state. After that, only the dispatched actions will be able to change the state.

Listing 7. Example reducer.ts file.

```
import { SagaAction, ProgressiveCallingState, Mode, Action } from '.';
import { RootState } from '../../reducers';

const defaultState: ProgressiveCallingState = {
    mode: Mode.Auto,
    started: false,
};

export const progressiveCallingReducer =
    (state = defaultState, action: Action): ProgressiveCallingState => {
    switch (action.type) {
        case SagaAction.Start:
            return {
                ...state,
                started: true,
            };

        case SagaAction.Stop:
            return {
                ...state,
                started: false,
            };

        case SagaAction.SetMode:
            return {
                ...state,
                mode: action.payload.mode,
            };

        default:
            return state;
    }
};

export const getProgressiveCallingMode = (state: RootState) =>
    state.progressiveCalling.mode;
export const getProgressiveCallingStarted = (state: RootState) =>
    state.progressiveCalling.started;
```

The second argument provided for the reducer function is the **action** and it has been defined to be of the type **Action**. That type is a union type of all the different actions that there are in this module. Defining the **Action** type is a convention used in the modules and it helps especially in making the reducers simpler. This is because of TypeScript and how it can use

the switch-case statement as a type guard. For example, in this case the **action** argument has the union type **Action**, but actually in each of the different case branches the type will be resolved to a particular subtype of **Action**. Modern IDEs will be able to display that information to the developer in real-time when working with the code and also provide intelligent auto-complete options when typing out the properties of the **action** object.

At the bottom of the file you will find the selector functions of the module. These are used for accessing the data in the store that is related to this module. These examples are pretty simple, but the important thing here is to remember that strictly using selector functions as the reading API for your store will decouple all the other components of the system from the store structure. It will allow for refactoring the store structure later without prompting changes in the other components. The **RootState** type used here refers to the structure of the whole Redux store which is a combination of all the state trees defined by each of the modules.

Listing 8 presents the first part of the actions.ts file of the "ProgressiveCalling" module. In this file the **ContactT** type is imported from the contacts module and used here, since this module is also working with the contacts stored in the system. While it would be preferable to not have dependencies to other modules, in practice it seems unavoidable if the modules are small enough and you must implement complex workflows using them. At least the import statements are explicit and provide a nice documentation about the dependencies that a module or any other component in the system has.

Listing 8. Example actions.ts file (part 1/3).

```
import { Mode } from '.';
import { ContactT } from '../contacts';

// ACTION TYPE ENUMS

export const enum SagaAction {
    Start = 'SAGA_START_PROGRESSIVE_CALLING',
    Stop = 'SAGA_STOP_PROGRESSIVE_CALLING',
    SetMode = 'SAGA_SET_PROGRESSIVE_CALLING_MODE',
    ContactFetchComplete = 'SAGA_PROGRESSIVE_CALLING_CONTACT_FETCH_COMPLETE',
    ContactFetchFailed = 'SAGA_PROGRESSIVE_CALLING_CONTACT_FETCH_FAILED',
}

export const enum UiAction {
    Start = 'UI_START_PROGRESSIVE_CALLING',
    Stop = 'UI_STOP_PROGRESSIVE_CALLING',
    SetMode = 'UI_SET_PROGRESSIVE_CALLING_MODE',
    NextContact = 'UI_PROGRESSIVE_CALLING_NEXT_CONTACT',
    NextAndRemove = 'UI_PROGRESSIVE_CALLING_NEXT_AND_REMOVE',
}
```

The other things visible in ***Listing 8*** are the action type enumerations. The action types are separated into saga originated actions and UI originated actions, since both have their own distinct purposes in this architecture. This way the separation is very clear also on the code level. The different action types provide a pretty good view into what functionalities the module has and especially what controls the UI will have.

In ***Listing 9*** the actions.ts file is continued. This listing displays all the different action object definitions done using TypeScript type definitions. First there is the union type **Action** that was previously referred to regarding the reducer.ts file. It is just a union of all the available action object types in the module. And after the union type there are the actual type definitions for every single one of the action object types.

Listing 9. Example actions.ts file continued (part 2/3).

```
export type Action =
    | SagaStartProgressiveCallingAction
    | SagaStopProgressiveCallingAction
    | SagaSetProgressiveCallingModeAction
    | SagaProgressiveCallingContactFetchCompleteAction
    | SagaProgressiveCallingContactFetchFailedAction
    | UiStartProgressiveCallingAction
    | UiStopProgressiveCallingAction
    | UiSetProgressiveCallingModeAction;

interface SagaStartProgressiveCallingAction {
    type: SagaAction.Start;
}

interface SagaStopProgressiveCallingAction {
    type: SagaAction.Stop;
}

interface SagaSetProgressiveCallingModeAction {
    type: SagaAction.SetMode;
    payload: { mode: Mode };
}

interface SagaProgressiveCallingContactFetchCompleteAction {
    type: SagaAction.ContactFetchComplete;
    payload: { contact: ContactT; phoneNumberId: number };
}

interface SagaProgressiveCallingContactFetchFailedAction {
    type: SagaAction.ContactFetchFailed;
    payload: { error: string; description: string };
}

interface UiStartProgressiveCallingAction {
    type: UiAction.Start;
}

interface UiStopProgressiveCallingAction {
    type: UiAction.Stop;
}

interface UiSetProgressiveCallingModeAction {
    type: UiAction.SetMode;
    payload: { mode: Mode };
}
```

The **type** property is always one of the previously defined saga action or UI action enumeration values. Since every action object has a different **type** property, it can be used as a discriminating property by the TypeScript type guards such as the switch-case statement. This usage was demonstrated earlier when describing the reducer.ts file where is it particularly helpful. The **payload** property type definitions are also very useful during development, since they allow the IDE to help with auto-completing the property names contained within. And the TypeScript compiler will catch errors if the developer is trying to use a property that is not actually there in the payload. All that is very beneficial in a complicated application where the amount of different actions is large. It quickly becomes difficult to remember the all payload types off the top of one's head.

Listing 10 shows the last thing defined in the actions.ts file – the action creators. Those are a very basic piece of the Redux puzzle. In practice they are nothing more than functions that return action objects. In some architectures the action creators contain part of the application logic and even perform asynchronous tasks. But in the architecture proposed in this thesis the action creators remain very simple and as a result are stable and trustworthy. Especially when you meticulously define all the argument types and the function return types, it in turn makes using the action creators safe and easy all around the components of your application.

Listing 10. Example actions.ts file continued (part 3/3).

```
export const sagaStartProgressiveCalling = ():
SagaStartProgressiveCallingAction => ({
    type: SagaAction.Start,
});

export const sagaStopProgressiveCalling = ():
SagaStopProgressiveCallingAction => ({
    type: SagaAction.Stop,
});

export const sagaSetProgressiveCallingMode = (mode: Mode):
SagaSetProgressiveCallingModeAction => ({
    type: SagaAction.SetMode,
    payload: { mode },
});

export const sagaProgressiveCallingContactFetchComplete = (
    contact: ContactT,
    phoneNumberId: number
): SagaProgressiveCallingContactFetchCompleteAction => ({
    type: SagaAction.ContactFetchComplete,
    payload: { contact, phoneNumberId },
});

export const sagaProgressiveCallingContactFetchFailed = (error: any):
SagaProgressiveCallingContactFetchFailedAction => ({
    type: SagaAction.ContactFetchFailed,
    payload: error,
});

export const uiStartProgressiveCalling = ():
UiStartProgressiveCallingAction => ({
    type: UiAction.Start,
});

export const uiStopProgressiveCalling = ():
UiStopProgressiveCallingAction => ({
    type: UiAction.Stop,
});

export const uiSetProgressiveCallingMode = (mode: Mode):
UiSetProgressiveCallingModeAction => ({
    type: UiAction.SetMode,
    payload: { mode },
});
```

Listing 11 presents the first part of the sagas.ts file in the example module. The import statements have been omitted to make the code snippet more compact. Here a common pattern with an infinite loop has been used to keep the saga running continuously while still allowing the saga to pull the actions that it is interested in. The first **take** effect will suspend the saga until a **UiAction.Start** action or a **UiAction.SetMode** action is dispatched by the related UI component. The **UiAction.SetMode** branch of the code is very simple, it will just dispatch a saga action to instruct the reducer to update the selected mode in the Redux store. And the **continue** keyword there is just regular JavaScript code which causes the while loop to continue from the top.

Listing 11. Example sagas.ts file (part 1/3).

```
function* progressiveCallingSaga() {
    while (true) {
        let action: Action = yield take([
            UiAction.Start,
            UiAction.SetMode
        ]);
        if (action.type === UiAction.SetMode) {
            yield put(sagaSetProgressiveCallingMode(action.payload.mode));
            continue;
        }
        const progressiveMode = yield select(getProgressiveCallingMode);
        yield put(sagaStartProgressiveCalling());
        while (true) {
            const fetchContactTask = yield fork(fetchNextContact);
            action = yield take([
                SagaAction.ContactFetchComplete,
                SagaAction.ContactFetchFailed,
                UiAction.Stop
            ]);
            if (action.type === UiAction.Stop) {
                yield cancel(fetchContactTask);
                yield put(sagaStopProgressiveCalling());
                break;
            } else if (action.type === SagaAction.ContactFetchFailed) {
                yield put(sagaStopProgressiveCalling());
                if (action.payload.error === 'no_contacts_left') {
                    yield put(sagaShowError(action.payload.description));
                }
                break;
            }
```

Most of the code in this saga is only accessed if the **UiAction.Start** action is dispatched. In that case the actual progressive calling functionality is started, and a saga action is dispatched to update the store with that information. Then the saga will kick off another infinite while loop which then handles the internal logic of running the progressive calling. The first thing there is to fetch a contact to be called. ***Listing 13*** contains the definition for the **fetchNextContact** generator function that is defined later in this same sagas.ts file. The fetching task is forked so that it will be processed in parallel while this saga still continues to monitor for the next set of actions that might get dispatched.

There are three actions that this saga is listening for at this point. The contact fetching task is an asynchronous task that might complete or fail at any time. And then there is also the possibility that the user wants to stop the progressive calling while the contact fetching is still ongoing. This is a good example of a typical situation in concurrent programming [30]. There are multiple things that can happen in an undetermined order and at unknown times and somehow the program must handle those events properly. Sagas provide a descriptive way of writing this as a piece of code. After the take effect catches an action, a simple if statement can be used to determine the action type and process the action accordingly.

If the user tries to stop the progressive calling, the **UiAction.Stop** action is dispatched. In that case the contact fetching task will be cancelled, and a saga action is dispatched to update the progressive calling state in the store. Also, the **break** keyword is used to break out of the inner while loop and once again end up at the very beginning of the outer while loop and waiting for the progressive calling to be started again.

If the contact fetching fails, then the progressive calling is also stopped. There is special handling logic for a specific kind of an error (‘no_contacts_left’) that will trigger an error notification to be displayed for the user if the contact list runs out of contacts. Also, other kinds of errors could be handled here, but at this point those are just ignored.

Listing 12 describes what happens if the third kind of action (contact fetch complete) is dispatched. In that case the progressive calling has not been stopped and a contact has been successfully fetched. At that point the phone number of the contact is available in the Redux store and accessible through the selector function **getCurrentPhoneNumber**. In this

example, only the **Auto** mode of progressive calling is considered, so the conditional checking the **progressiveMode** variable in the if statement will be true.

Listing 12. Example sagas.ts file continued (part 2/3).

```
            const phoneNumber = yield select(getCurrentPhoneNumber);
            if (progressiveMode === Mode.Auto) {
                yield put(sagaCall(phoneNumber.number, CallType.Auto));
                yield fork(phoneNumbersStorage.bounce, phoneNumber.id);
                yield put(sagaOpenModal(ModalType.ContactDialog));
                action = yield take([
                    CallsUiAction.FinalizeCall,
                    UiAction.Stop,
                    UiAction.NextContact,
                    UiAction.NextAndRemove,
                ]);
                yield put(sagaCloseModal(ModalType.ContactDialog));
                if (action.type === UiAction.Stop) {
                    yield put(sagaStopProgressiveCalling());
                    yield fork(hangup);
                    break;
                }
                if (action.type === UiAction.NextContact) {
                    yield fork(hangup);
                    continue;
                }
                if (action.type === UiAction.NextAndRemove) {
                    yield fork(hangup);
                    const currentCall = yield select(getCurrentCall);
                    yield fork(callStorage.invalidate, currentCall.id);
                    continue;
                }
            }
        }
    }
}
```

The **sagaCall** action creator will be used to create an action that will trigger another module in the system to make a phone call. And the **phoneNumbersStorage.bounce** storage API method is used to move the called number to the end of the contact list so that it will not be called again right away. The **sagaOpenModal** action creator is used for creating an action for opening a modal dialog. In this case the “contact dialog” will be opened to handle the ongoing phone call and manage the contact details.

Once that is done, another **take** effect will suspend the saga until the phone call is ended or another UI action is dispatched. Any of those actions will prompt the closing of the "contact dialog" and depending on the action type some other logic as well. If the **CallsUiAction.FinalizeCall** action is dispatched from the "calls" module, then no further processing is needed here, since in that case the call hang up logic has already been handled by the "calls" module sagas.

The progressive calling module controls three different buttons in the "contact dialog" and the actions initiated by the user clicking on those buttons are handled here. The **UiAction.Stop** action will stop the progressive calling and hang up the ongoing phone call. The **break** keyword will break out of the inner loop and suspend the saga to wait for a **Start** action. The **UiAction.NextContact** action will hang up the ongoing phone call, but it will not break out of the inner loop. Instead, it is using the **continue** keyword to continue execution from the beginning of the inner loop, effectively fetching the next contact on the contact list and calling that contact. The **UiAction.NextAndRemove** action behaves in a similar fashion, but it removes the current contact from the contact list altogether before proceeding to fetching the next contact.

Listing 13 shows the **fetchNextContact** generator function declaration. This is an example of how sagas can be split into smaller functions and how an asynchronous task can be contained in its own generator function that can communicate with the other sagas using Redux actions. The main progressive calling saga uses the **fork** effect to start this task in the background so that it can keep listening for other actions constantly.

The implementation of **fetchNextContact** is pretty simple and the main thing to understand is that the **storage.nextContact** is calling an API endpoint over a network so the request will take some time to complete and may also fail for a variety of different reasons. The "try-catch-finally" construct handles the possible failures and cancellations. If the request is successful, the **SagaAction.ContactFetchComplete** action is dispatched with the contact information. If the request fails, the "catch" block is executed and as a result the **SagaAction.ContactFetchFailed** action will get dispatched with the related error message. The "finally" block is always executed after the "try" and "catch" blocks and that is where

the saga cancellation logic could be implemented. In this case there is no need for any actual cancellation handling, but the construct is left in the code as an example. The last line in ***Listing 13*** shows how the sagas are exported from the module so that the root saga can start them when the application is started. The **fetchNextContact** generator function is not included in the exported sagas, since it is only intended to be invoked by the **progressiveCallingSaga** and should not be started at application startup.

Listing 13. Example sagas.ts file continued (part 3/3).

```
function* fetchNextContact() {
    try {
        const currentCampaign = yield select(getCurrentCampaign);
        const nextContact = yield call(storage.nextContact,
            { campaign_id: currentCampaign.id });
        yield put(sagaProgressiveCallingContactFetchComplete(
            nextContact.contact,
            nextContact.phone_number_id));
    } catch (e) {
        yield put(sagaProgressiveCallingContactFetchFailed(e.response.data));
    } finally {
        if (yield cancelled()) {
            // ... put special cancellation handling code here
        }
    }
}

export const progressiveCallingSagas = [progressiveCallingSaga];
```

Listing 14 displays the storage.ts file of the module. It is responsible for providing an API for the module to access some permanent storage. The concrete API implementation is done elsewhere, and this file is just an abstraction, so there should be no direct references to any network or database technologies in here. In this module there is only the **nextContact** function in the storage. Also, the response and request data types are defined here, since they are so closely related to the storage function.

Listing 14. Example storage.ts file.

```
import { api } from '../../../storage';
import { ContactT } from '../contacts';

interface NextResponse {
    phone_number_id: number;
    contact: ContactT;
}

interface NextRequest {
    campaign_id: number;
    contact_list_id?: number;
}

class ProgressiveModeStore {
    /**
     * Get next contact information for progressive calling from campaign
     *
     * @params {NextRequest} request parameters
     * @return {Promise<NextResponse>}
     */
    nextContact(params: NextRequest): Promise<NextResponse> {
        return api.get('ProgressiveMode', 'nextContact', params);
    }
}

export const storage = new ProgressiveModeStore();
```

4.2.2 Combining all the modules of the application together

Listing 15 shows the reducers.ts file of the application. That is used for combining all the different module reducers together to produce the full Redux store structure for the application. Redux provides a **combineReducers** function for doing that easily. The full store structure object will be passed in as an argument for the function. And also, the root state type can be defined for type-safety. All the module reducers and their related state type definitions just need to be imported here and used appropriately.

Listing 15. Example reducers.ts file.

```
import { combineReducers } from 'redux';

import { callsReducer, CallsState } from './modules/calls';
import { campaignsReducer, CampaignsState } from './modules/campaigns';
import { callManagerReducer, CallManagerState } from './modules/callManager';
import { sessionReducer, SessionState } from './modules/session';
import { modalsReducer, ModalsState } from './modules/modals';
import { progressiveCallingReducer, ProgressiveCallingState } from
    './modules/progressiveCalling';
import { errorsReducer, ErrorsState } from './modules/errors';
import { contactsReducer, ContactsState } from './modules/contacts';
import { viewsReducer, ViewsState } from './modules/views';

export interface RootState {
    callManager: CallManagerState;
    calls: CallsState;
    campaigns: CampaignsState;
    contacts: ContactsState;
    errors: ErrorsState;
    modals: ModalsState;
    progressiveCalling: ProgressiveCallingState;
    session: SessionState;
    views: ViewsState;
}

export default combineReducers<RootState>({
    callManager: callManagerReducer,
    calls: callsReducer,
    campaigns: campaignsReducer,
    contacts: contactsReducer,
    errors: errorsReducer,
    modals: modalsReducer,
    progressiveCalling: progressiveCallingReducer,
    session: sessionReducer,
    views: viewsReducer,
});
```

Listing 16 displays the sagas.ts file of the application. Here the root saga of the application is defined. The root saga is what the Redux store will execute when the application is started. The official Redux-Saga documentation [31] provides multiple different examples for root saga patterns and this is a slight modification of one of those examples. Basically, all the sagas imported from the modules will be looped through and started one by one using the

spawn effect. The module sagas are also wrapped in a generator function that will restart them if an error occurs and logs that error in the console for easier debugging.

Listing 16. Example sagas.ts file.

```
import { call, all, spawn } from 'redux-saga/effects';

import { callsSagas } from './modules/calls';
import { campaignsSagas } from './modules/campaigns';
import { progressiveCallingSagas } from './modules/progressiveCalling';
import { callManagerSagas } from './modules/callManager';
import { sessionSagas } from './modules/session';
import { errorsSagas } from './modules/errors';
import { initSagas } from './modules/init';
import { contactsSagas } from './modules/contacts';
import { viewsSagas } from './modules/views';

export default function* rootSaga() {
    const sagas = [
        ...callsSagas,
        ...initSagas,
        ...sessionSagas,
        ...callManagerSagas,
        ...campaignsSagas,
        ...progressiveCallingSagas,
        ...errorsSagas,
        ...contactsSagas,
        ...viewsSagas,
    ];

    yield all(
        sagas.map(saga =>
            spawn(function*() {
                while (true) {
                    try {
                        yield call(saga);
                        break;
                    } catch (e) {
                        console.error('Error in saga ' + saga.name + ': ' + e);
                    }
                }
            })
        )
    );
}
```

4.2.3 Automated testing of the sagas

Listing 17 shows two examples of automatic tests implemented for the **fetchNextContact** saga that was presented earlier in ***Listing 13***. The tests are written using the Jest testing framework for JavaScript [32], but they should work with other frameworks as well by adjusting the syntax to match the API of that particular framework. Moreover, the "redux-saga-test-plan" library [33] is utilized to help structure the test cases. It provides means to write unit tests and integration tests using the same library, so it is a bit more versatile than the other testing libraries available for testing sagas. The official Redux-Saga documentation contains a brief overview of the different testing libraries and the basics of testing sagas, so that is a great place to start when considering different testing approaches [34].

Looking at ***Listing 17*** from the top, the first thing used there is the **describe** function that is part of the Jest framework. It is used for declaring a test suite and in this case the test suite is about testing the **fetchNextContact** saga and it is also named accordingly. The **nextContact** constant is providing test data for the test suite, in this case a dummy contact object that will be used later in the test cases. The **it** function is yet again part of the Jest framework and is used for declaring individual test cases.

First there is an example of a unit test. That test case verifies that the **fetchNextContact** generator function actually goes through all the steps it is supposed to. The **testSaga** function is imported from the "redux-saga-test-plan" library and it provides an API for writing unit tests. The API allows chaining all the related helper function calls together to produce a clear and easily readable test case structure. The **next** function is used for advancing the saga execution to the next **yield** statement in the code. So, the API is similar to the common JavaScript generator function syntax in that sense. Then the **select** function asserts that the selector **getCurrentCampaignId** gets called by the saga. After that, the **next** function is once again used to advance the saga and to simulate a response value from the **yield** statement which in this case was the **select** effect. So, in this example the number 42 is the simulated response value from the yielded **select** effect. The logic is similar for the other effects as well. In the end, the **isDone** function is used for asserting that the generator function has finished.

Listing 17. Automated tests for a saga.

```
describe('fetchNextContact', () => {
    const nextContact = {
        phone_number_id: 1,
        contact: {
            id: 1,
            first_name: 'Test Contact',
        },
    };
    it('unit test', () => {
        testSaga(fetchNextContact)
            .next()
            .select(getCurrentCampaignId)
            .next(42)
            .call(storage.nextContact, { campaign_id: 42 })
            .next(nextContact)
            .put(sagaProgressiveCallingContactFetchComplete(
                nextContact.contact,
                nextContact.phone_number_id
            ))
            .next()
            .cancelled()
            .next()
            .isDone();
    });
    it('integration test with contactsReducer', () => {
        return expectSaga(fetchNextContact)
            .withReducer(contactsReducer)
            .provide([
                [select(getCurrentCampaignId), 42],
                [call(storage.nextContact, { campaign_id: 42 }), nextContact]
            ])
            .hasFinalState({
                current: {
                    id: nextContact.contact.id,
                    phoneNumberId: nextContact.phone_number_id,
                    history: [],
                },
                contactsById: {
                    [nextContact.contact.id]: nextContact.contact,
                },
            })
            .run();
    });
});
```

The second test case is an example of an integration test. The “redux-saga-test-plan” library provides the **expectSaga** function for that purpose. One interesting capability of this library is being able to write integration tests that also the test the related reducer. This way it is possible to verify that the part of the Redux store managed by that reducer ends up in the expected state. In integration tests it is not necessary to verify and test all the steps in the sequence the saga will perform, but instead make sure the saga integrates properly with the rest of the system.

In this example test case the **fetchNextContact** saga is tested with the **contactsReducer** that is the reducer function of the “contacts” module in the system. The reasoning for that is that the **fetchNextContact** saga does not cause any changes in the state managed by the reducer of the “progressiveCalling” module, but it in fact updates the current contact information of the system that is managed by the **contactsReducer**. The **expectSaga** function is called with the tested the saga as a parameter and the **withReducer** function sets the reducer function to test. The **provide** function of the test API is used for providing mocked response data for the effects yielded by the saga. It takes an array of tuples as an argument. The first value in the tuple is the effect matcher (e.g. select(getCurrentCampaignId)) and the second value is the response value for that effect (e.g. 42, respectively). The **hasFinalState** function is then used for verifying the final Redux store state produced by the tested reducer to make sure that the integration between the saga and the reducer worked as expected.

5 DISCUSSION AND CONCLUSIONS

The previous section described the results of the thesis work. A theoretical architecture model was formulated, and concrete implementation examples were presented. The model was validated empirically by applying it in practice in a real software project to produce a working application.

Separation of concerns [35] is an important principle that aims to make software development and design simpler by employing a "divide and conquer" type of strategy. Modularization of code is good way to achieve separation of concerns in practice. Grouping logically related elements together gives the project a structure that supports developability and maintainability of the system. A module with high cohesion and low coupling allows for easier modifications by containing the impact of the change within the module. Inversely, concern scattering [36] has a detrimental effect on the programmer productivity, because locating all the related code fragments that implement a single concern or feature is more difficult. And basically, no proper modification to the code is possible before all the relevant code is located, so this is a very important factor in developer efficiency. The modular project structure proposed in this thesis works well in this regard. And on the other hand, using sagas to model the use cases of the system achieves similar benefits. When a use case is contained in a saga, the related code is easy to locate and understand by studying the flow of the saga.

One of the main ideas behind the architecture was introducing different levels of Redux actions. By separating the UI originated actions and the saga originated actions it became possible to keep the sagas always in control. It enables fully containing the use cases related logic in the sagas. While helping with program comprehension, it also facilitates for automated testing of the use cases of the system. A quality that is ever so valuable in the modern agile software development industry.

Using Redux makes the code very explicit in many places and increases the readability. The developer reading the code does not need to guess what is going on, so the perceived complexity of the code decreases. Future research on the topic could include decreasing the boilerplate code. There is quite a bit of code required for declaring all the action object types and action creators, for example. While code like that is explicit and easy to understand in

itself, writing a lot of similar code will get tedious over time and might cause developers to lose focus and make mistakes in the process. Some sources like [37] provide guidelines for reducing the boilerplate code while keeping the code type-safe, so that could be something worth studying in the future. Also, when the project grows and the amount of different modules and actions in the system increases, it might become important to add namespacing like this [38] to the action type constants to avoid possible name collisions in the future.

There are many different ways to do asynchronous programming in JavaScript. Traditionally callback functions have been used to handle asynchrony, but they quickly result in complex code that is difficult to read. Promises are another mechanism that, while more usable than the callbacks, still often results in hard to understand nested and chained structures in code. The async/await method is providing a syntax that is similar to writing synchronous code and that seems to be the easiest for developers to grasp. The Redux-Saga library leverages generator functions to keep the code structures looking synchronous while providing fully asynchronous functionalities. After you get used to the syntax it becomes a powerful tool for asynchronous programming.

One thing that came up multiple times during this project was that the module boundaries are not easy to define. It is hard to say what part of the software should be contained in its own module and what to include in that module. That will just probably be a question that there is not always a clear answer for. And sometimes you might need to go back on your decisions and split or combine modules if it becomes apparent that the current structure is causing issues. Once again, this too is an iterative process. The most important thing is to start doing things and not be afraid that the first decision might not be the best one right off the bat. The modules can always be refactored later if needed.

Also, it was noticed that at least in this project the modules will need to use other modules pretty frequently and import types and functionalities from other modules. While it is not optimal to have the modules depend on each other and it would be better if they could be completely self-reliant, there seems to be no way around it in practice. Doing that has also not caused any issues so far, so you should not be too afraid of doing it. Still it should be better to have a lot of small and crisply defined modules that might use other modules than to have large overly complex modules containing too many features.

The results achieved provide answers to the research questions. The architecture model seems scalable and supports automated testing of the system use cases. Redux-Saga suits well for modeling the application logic and use cases. The presented implementation example of a single module in the system describes an effective way of implementing the architectural model in practice.

6 SUMMARY

There was a need to formulate a description of a software architecture that would provide scalability to support the long-term development of a complex web application built on Redux and Redux-Saga. Code readability, separation of concerns in the code level and the ability to write automated tests for the system use cases were considered key values to aim for in the design. By imposing certain rules on the way that Redux actions are utilized in the system the application level business rules can be modeled using Redux-Saga and the use cases can be contained in the sagas.

In the scope of this thesis work the designed architectural solution seemed able to meet the requirements, even though the application itself had not yet grown very complex at this time. So, there would be need for further research to properly validate the design in a larger and more complex application. The practical implementation at the code level could also be improved by doing more research on available libraries and patterns for streamlining the boilerplate code associated with Redux. But even as it is the presented implementation utilizing TypeScript will provide a solid foundation to build on.

REFERENCES

[1] D. Anselmo and H. Ledgard, "Measuring Productivity in the Software Industry," *Communications of the ACM,* vol. 46, no. 11, 2003.

[2] R. C. Martin, Clean Code: A Handbook of Agile Software Craftsmanship, USA: Prentice Hall, 2009.

[3] D. Alawad, M. Panta, M. Zibran and M. R. Islam, "An Empirical Study of the Relationships between Code Readability and Software Complexity," in *International Conference on Software Engineering and Data Engineering (SEDE)*, New Orleans, Louisiana, USA, 2018.

[4] A. Trendowicz and J. Münch, "Factors Influencing Software Development Productivity - State-of-the-Art and Industrial Experiences," *Advances in Computers,* no. 77, 2009.

[5] S. Fakhoury, Y. Ma, V. Arnaoudova and O. Adesope, "The Effect of Poor Source Code Lexicon and Readability on Developers' Cognitive Load," in *ICPC '18*, 2018.

[6] W. Ariola and C. Dunlop, "DevOps: Are You Pushing Bugs to Your Clients Faster?," in *Pacific NW Software Quality Conference*, 2015.

[7] I. Gorton, Essential Software Architecture, Berlin, Heidelberg: Springer Berlin Heidelberg, 2011.

[8] R. Schmidt, Software engineering: architecture-driven software development, Waltham, MA: Morgan Kaufmann, an imprint of Elsevier, 2013.

[9] T. Stober and U. Hansmann, Agile Software Development: Best Practices for Large Software Development Projects, Berlin, Heidelberg: Springer Berlin Heidelberg, 2010.

[10] E. Evans, Domain-Driven Design: Tackling Complexity in the Heart of Software, USA: Addison-Wesley, 2004.

[11] R. C. Martin, Clean Architecture: A Craftsman's Guide to Software Structure and Design, USA: Prentice Hall, 2018.

[12] R. C. Martin, "The Clean Code Blog," [Online]. Available: https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html. [Accessed 23 August 2019].

[13] O. Farhi, Reactive Programming with Angular and ngrx: Learn to Harness the Power of Reactive Programming with RxJS and ngrx Extensions, Berkeley, CA: Apress, 2017.

[14] C. Gackenheimer, Introduction to React, Berkeley, CA: Apress, 2015.

[15] F. Zammetti, Practical React Native: Build Two Full Projects and One Full Game using React Native, Berkeley, CA: Apress, 2018.

[16] "Redux Docs," [Online]. Available: https://redux.js.org. [Accessed 24 July 2019].

[17] H. Garcia-Molina and K. Salem, "Sagas," in *SIGMOD '87 Proceedings of the 1987 ACM SIGMOD international conference on Management of data*, Princeton University, Princeton, NJ, 1987.

[18] A. Aravinth and S. Machiraju, Beginning Functional JavaScript: Uncover the Concepts of Functional Programming with EcmaScript 8, Berkeley, CA: Apress, 2018.

[19] "MDN web docs: function*," [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/function*. [Accessed 25 July 2019].

[20] "Redux-Saga docs," [Online]. Available: https://redux-saga.js.org/. [Accessed 25 July 2019].

[21] S. Fenton, Pro TypeScript: Application-Scale JavaScript Development, Berkeley, CA: Apress, 2018.

[22] A. Van, "Redux-Saga in Action(s)," [Online]. Available: https://medium.com/@totaldis/redux-saga-in-action-s-f7d11cffa35a. [Accessed 24 August 2019].

[23] M. Schulze, "Using React (-Native) with Redux and Redux-Saga. A new proposal?," [Online]. Available: https://medium.com/@marcelschulze/using-react-native-with-redux-and-redux-saga-a-new-proposal-ba71f151546f. [Accessed 24 August 2019].

[24] K. Poole, "Redux Best Practices," [Online]. Available: https://medium.com/@kylpo/redux-best-practices-eef55a20cc72. [Accessed 30 September 2019].

[25] "Redux Selector Pattern," [Online]. Available: https://gist.github.com/abhiaiyer91/aaf6e325cf7fc5fd5ebc70192a1fa170. [Accessed 30 September 2019].

[26] E. Ong, "So you've screwed up your Redux store — or, why Redux makes refactoring easy," [Online]. Available: https://engineering.haus.com/so-youve-screwed-up-your-redux-store-or-why-redux-makes-refactoring-easy-400e19606c71. [Accessed 25 August 2019].

[27] E. Rasmussen, "Ducks: Redux Reducer Bundles," [Online]. Available: https://github.com/erikras/ducks-modular-redux. [Accessed 25 August 2019].

[28] E. Elliott, "10 Tips for Better Redux Architecture," [Online]. Available: https://medium.com/javascript-scene/10-tips-for-better-redux-architecture-69250425af44. [Accessed 31 August 2019].

[29] P. Witek, "GitHub repository: piotrwitek/utility-types," [Online]. Available: https://github.com/piotrwitek/utility-types. [Accessed 27 September 2019].

[30] F. B. Schneider, On Concurrent Programming, Ithaca, NY, USA: Springer, 1997.

[31] "Redux-Saga Docs: Root Saga Patterns," [Online]. Available: https://redux-saga.js.org/docs/advanced/RootSaga.html. [Accessed 29 September 2019].

[32] "Jest - Delightful JavaScript Testing," [Online]. Available: https://jestjs.io/. [Accessed 29 September 2019].

[33] "GitHub repository: jfairbank/redux-saga-test-plan," [Online]. Available: https://github.com/jfairbank/redux-saga-test-plan. [Accessed 29 September 2019].

[34] "Redux-Saga Docs: Testing," [Online]. Available: https://redux-saga.js.org/docs/advanced/Testing.html. [Accessed 29 September 2019].

[35] P. A. Laplante, What Every Engineer Should Know About Software Engineering, Boca Raton, FL, USA: CRC Press, 2007.

[36] J. Juhár and L. Vokorokos, "Separation of concerns and concern granularity in source code," in *2015 IEEE 13th International Scientific Conference on Informatics*, Poprad, Slovakia, 2015.

[37] P. Witek, "GitHub repository: piotrwitek/react-redux-typescript-guide," [Online]. Available: https://github.com/piotrwitek/react-redux-typescript-guide. [Accessed 30 September 2019].

[38] S. C. Barrus, "The Ducks File Structure for Redux," [Online]. Available: https://medium.com/@scbarrus/the-ducks-file-structure-for-redux-d63c41b7035c. [Accessed 1 October 2019].