

Nguyen Huu Ngoc Chi

Functional Reactive Programming in React Application

Metropolia University of Applied Sciences

Bachelor of Engineering

Information Technology

Thesis

13 April 2018

Author Title	Nguyen Huu Ngoc Chi Functional Reactive Programming in React Application
Number of Pages Date	30 pages 13 April 2017
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Professional Major	Software Engineering
Instructors	Janne Salonen, Head of Department (ICT)
<p>Tsampo - is a new Finnish startup company with strong desire to create an application that connects researchers and investors. The company plans to renovate funding procedure by replacing all manual processes and converting paper works completely to electrical forms. Instead of collecting resources manually, the application can help dispatching fund request automatically from a group of scientists to subscribed list of interested investors with matching fields and criteria through connections from organization and universities.</p> <p>However, those functionalities and requirements of the application have put the development of application in many challenges. User experienced slow response interface, the development process is stuck with the performance problems, unmaintainable codebase.</p> <p>Therefore, the objective of this thesis is to propose software solutions for current Tsampo's user interface while keeping development process easier in the future.</p> <p>For long-term development, React is introduced to make the layout easy to maintain and understand. The interface is built with highly reusable, scalable React components. Main issues related to slow interface from complex user's interaction and data handling can be resolved by applying functional reactive programming (FRP) as a suggestion. Functional reactive programming paradigm help abstracting sequences of user's data updates, asynchronous requests, and events into data streams, which make managing and updating application's state easily, dynamically, increase overall performance and reduce bugs.</p> <p>In conclusion, introducing FRP's benefits along with React have proven to resolve Tsampo's problems. Pulling multiple files from different sources is handle without affecting application's performance. FRP makes React component functional and reacts dynamically towards complex state changes.</p>	
Keywords	React, Reactive Programming, Functional Programming, Observable, Web Development, RxJS, Mobx

Contents

1	Introduction	1
2	Theoretical background	2
2.1	React	2
2.1.1	Virtual DOM	3
2.1.2	React Component	5
2.1.3	Component Lifecycle	6
2.2	Programming Paradigms	9
2.2.1	Imperative Programming	9
2.2.2	Functional Programming	9
2.2.3	Reactive Programming	11
2.2.4	Functional Reactive Programming	12
2.3	Observer Pattern	13
2.3.1	Definition	13
2.3.2	Observer Pattern and Functional Reactive Programming	14
3	Implementation	15
3.1	Section Overview	15
3.2	Building Scalable User Interface with React	15
3.2.1	Structure Tsampo Application in React's components	15
3.2.2	Stateless Functional Component	17
3.2.3	Reactive stateful component	19
3.2.4	Higher Order Component	21
3.3	Performance Improvement	23
3.3.1	Optimize performance with React component's lifecycle.	24
3.3.2	State management with Mobx	25
3.3.3	Handle concurrent HTTP requests.	26
3.3.4	Throttle click events and user input	28
4	Conclusion	30
	References	31

List of Abbreviations

FRP	Functional Reactive Programming.
DOM	Document Object Model.
HTML	Hypertext Markup Language.
DIV	HTML Division tag.
UI	User interface.
HTTP	Hypertext Transfer Protocol.
URL	Uniform Resource Locator.
UX	User Experience.
SVG	Scalable Vector Graphics.
ES6	ECMA Script 2015. One version of ECMAScript standard
HOC	Higher Order Component.

1 Introduction

By the time of this project was initially written, there are around 5000 publications every day [1] and not all of them were used effectively. Tsampo company aims to create a revolution on how science is delivered by creating a workspace where questions and answers can resolve each other. The sources of the answers come from Tsampo's publications database, which was gathered from universities and scientific websites and institutions.

Moreover, instead of manually sending multiple grant requests and emails to multiple investors to get funding, the platform will help scientists and researchers simplify most of those human work by automatically transfer documents to investors, along with the ability to get funding requests tracked and notified.

The company's application comes with many challenges from handling a large amount of science data sources and user's interaction. User experienced slow responses, missing data and service interruptions from fetching a variety of journals, documents, scientific research in different universities and educational institutes at the same time. Multiple users with different access roles and live-editing events reduce overall website's performance, interaction performs slowly and unstable from numerous state updates.

Therefore, the goal of this project is focused on using benefits of FRP along with React to resolve problems that developers have faced during the implementation of Tsampo's user interface and making the development process easier in the future.

The project was developed for web platform, using React to create a scalable user interface and functional reactive programming to handle all complex cases of user interactive and asynchronous events.

2 Theoretical background

This chapter is divided into four subchapters to present and focus on explaining the theoretical background aspects from the title of this thesis. In 2.1, the overview introduction of React - main frontend library that entirely used to develop the user interface in this project. In 2.2, the general understanding about common programming paradigms and focus on FRP that heavily affect the main concept of this thesis. In 2.3, the presentation of design pattern which was chosen to use with FRP. Finally, in 2.4 is an explanation about the main motivation behind using new architectures and design pattern.

2.1 React

React is a Javascript library that is currently one of best designated for building user interfaces [2]. It was first invented and developed for internal usage of Facebook to build their own products [3]. Later then, published as open-source to the community and quickly become one of the most popular libraries around the world, among many other UI frameworks.

React is blazing fast in performance, unidirectional data flow [4], easy to learn and understand, quick to setup [5]. Developers who interested in creating React application not only learn how to create the code but also change their way to think in React - Think in component [6]. In order to enhance the ability of reusable code, React introduce encapsulated components, that can be shared and reused between application codebase [6]. Therefore, React helps to increase the consistency of the code, easy to test, reduce duplicated work, saving time and effort spent on the implementation process. Every website, application, however big it is, can also be split into small components. [6]

However, unlike other frontend frameworks, for example, AngularJS or VueJS, React is only a Javascript view library [5], which basically do the same job as building user interfaces [5]. React is lightweight in package size but depend on many other libraries to be used effectively. React component can be used not only for creating the user interface for web client application, also mobile apps (React Native), and render on a server by using Node.

Finally, in order to understand React deeply, a programmer must know keys that make React success toward others in both performance and simplicity: Virtual DOM, React component and lifecycle.

2.1.1 Virtual DOM

What is DOM? DOM is known as the standard object model and programming interfaces for HTML documents [7]. The user interface in our web browser nowadays is structured as an HTML DOM tree which contains multiple HTML elements as tree nodes. The figure below is an example of an HTML DOM tree:

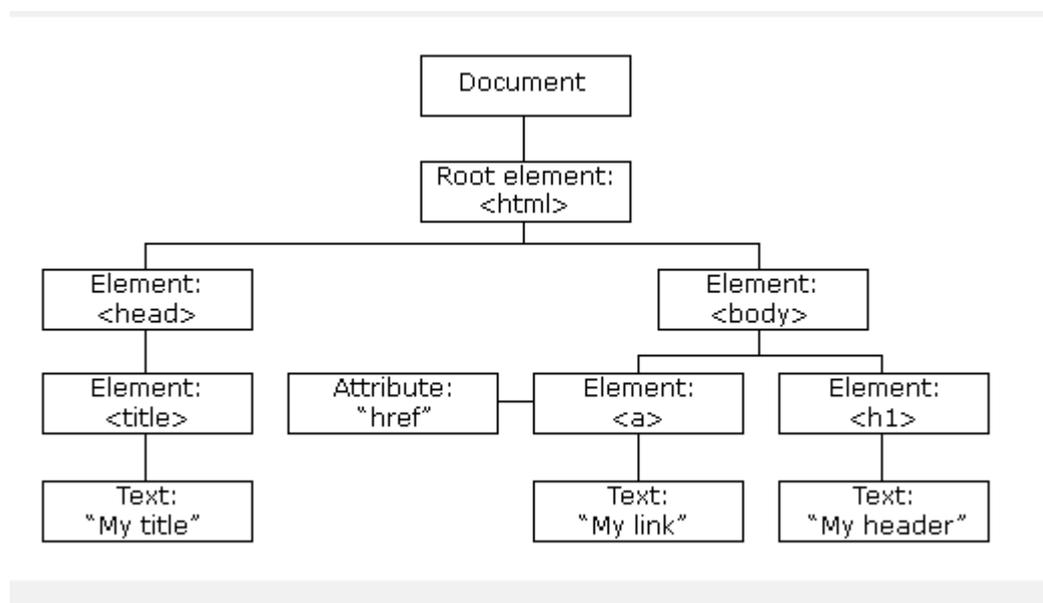


Figure 1: HTML DOM Tree. Screenshot from [7]

Website getting more and more complicated and dynamic. As a consequent, the DOM tree gets huge by day. Each tree node is bind with multiple event handlers, such as: click, submit, hover, mouse interactions... Consider accessing a website with thousands of div like Facebook – the process of traversing, finding the element to update UI state based on user or server event actions is not easy and obviously slow, because of a large number of the node. React did not invent Virtual DOM but offer using it as a solution to solve those performance problems.

Virtual DOM as how the name hardly described is a fast representation of real DOM [8, p.19]. Virtual DOM is separate copy, simplified and lightweight version of DOM in Javascript instead of HTML structure [9] Therefore, operations going through the whole structure will be faster and easier. Without touching the real DOM tree and manipulate them, React uses an advanced algorithm call **reconciliation** [10] to find the element that need to be updated in real DOM by comparing between old and new virtual DOM.

Reconciliation compares each tree's root elements by **type** and **key**. Key is a unique attribute that React generate and add to element when the virtual DOM is rendered [10]. Key attribute makes the comparison process faster caused by unique value. If the roots have same types and same attributes, move to the children and repeat the process. If there is any difference is found, fully rebuild the DOM tree from that node position without going further, saving time and resources cost.

Re-render only happen with the selected DOM elements without affecting the whole HTML hierarchy, change the overall application's state. The process below can describe that process of updating UI state in React:

- An existing Virtual DOM representation get rendered with UI
- When there is a change in UI state, a new Virtual DOM representation will be rendered and compare to the old one. The differences between those two will be filtered and waiting to be applied to real DOM to show the changes.
- React apply the filtered changes to real DOM

A visual explanation of how the process worked is displayed in the figure below:

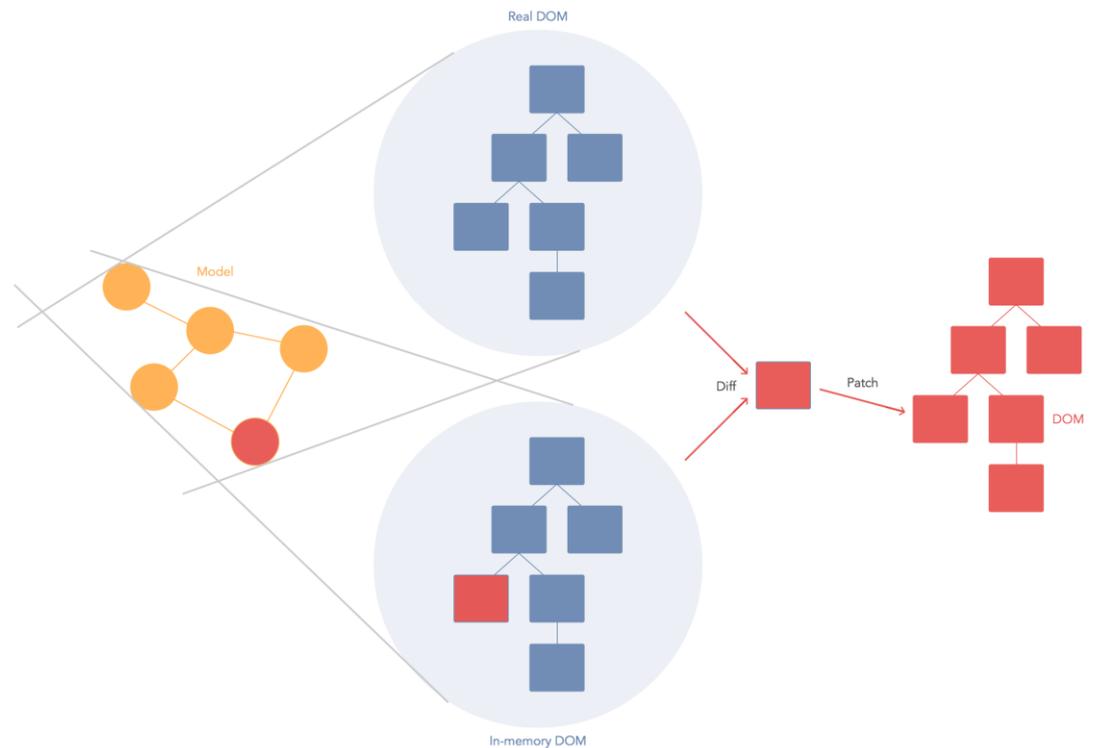


Figure 2: React Virtual DOM process of finding and update DOM. Copied from [11]

2.1.2 React Component

“Thinking in React” [5] term start with the idea of breaking any UI into small components. The whole application can be described as a hierarchy of React components. Those components are supposed to be reusable, highly independent and easy to customize.

Theoretically, React component should look like a Javascript function. A function in mathematics take input, process it and return output related to that input. React component also behave the same by taking arbitrary inputs (call “props”) as the parameter, process it and return React elements [12]. React is a declarative library [13], which the code, application’s state, and flow explain its own purpose instead giving instructions. React’s component which is a function should also explain the goal properly the same way.

Props are set by parent [14] and the representation of component's properties. By using props, a component can be reused and fit into different place in the application with small changes in the properties [14].

Below is an example of simple React component display greetings to the user based on their name as input props.

```
function Greetings(props) {  
  return <h1>Hello, welcome {props.name}</h1>;  
}
```

Figure 3: React component Greetings.

React component is controlled not only by props, the component itself can also have internal state. React component will re-render based on state changes by calling **setState()**. State enable the customization of the component by giving the ability to trigger changes remotely, from inside or outside of the component. The state makes React component divided into two kinds: stateless and stateful component.

A stateless component is React component which doesn't contain any internal state. The stateless component only updating toward changes from props, which was obtained from parent/wrapper component.

A stateful component is React component with internal state. Stateful component update/re-render either on internal state changes or props changes.

2.1.3 Component Lifecycle

The most important method that a React component must always have is **render()**. The method described how a component should look like. But having only render is not enough. Along with development process, a developer might not just expect the component to be rendered immediately after loaded. Some tasks should be done before rendering such as data fetching, validation, authentication... and sometimes no render is needed at all. In order to intercept various stages of React component rendering, React

component itself have a set of methods call lifecycle to control before, after, a update of the rendering process.

React lifecycle methods can be grouped into three main set as the figure below:

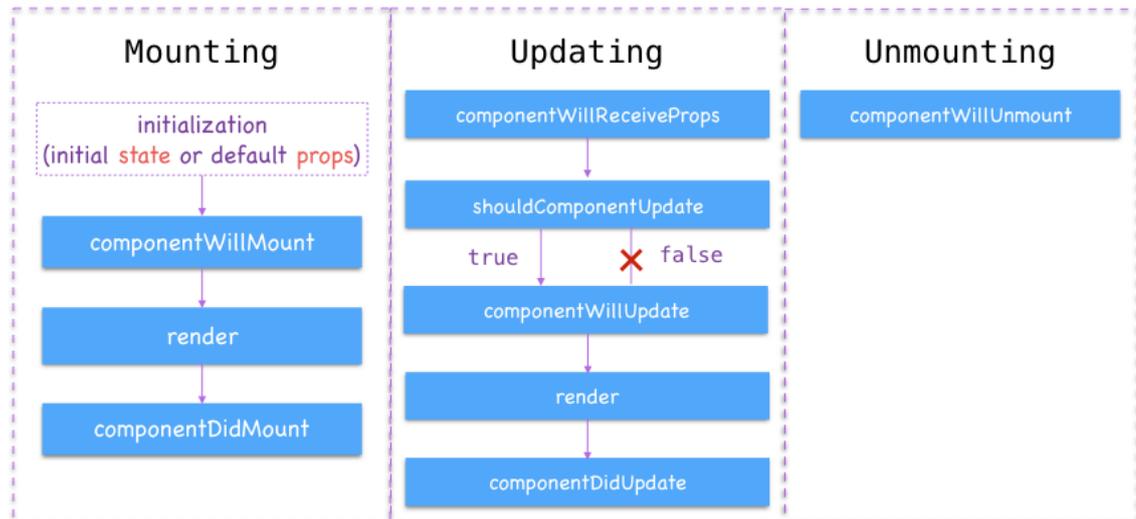


Figure 4: React component's lifecycle. Copied from [15]

- **Mounting:** This phase happens when the component is added to the DOM [8, p.65], including before and after insertion process. **ComponentWillMount** controls the stage before mounting to DOM, usually used for gather resources, initial values for first rendering. After the component is mounted to DOM, **componentDidMount** is called. Data fetching and side effects handling usually start to happen at this stage instead of `componentWillMount` to make sure component still render fine in earlier step with initial values.
- **Updating:** This phase happens when the component is already added to the DOM, component props or state is changed and required to decide to be re-render or not. **ComponentWillReceiveProps** is the stage when component receives state or props changes. **shouldComponentUpdate** give decisions whether the component should re-render based on changes or not, the default is true. If component decides to re-render, the stage before re-render can be controlled in **componentWillUpdate**. After the re-render, the stage goes to **componentDidUpdate**.

- Unmounting: This phase happens when the component is being removed out of DOM, handled by **componentWillUnmount**. This stage is usually designated for cleaning listeners, set timeout or interval function to prevent memory leak.

By using React component lifecycle method reactively, a developer can control different stages of the component and the re-rendering process smartly, reduce unpredicted state or props changes that need to be refresh. Hence, increasing overall application's performance.

Below is an example of using React component lifecycle to reduce unnecessary re-render:

```
class CounterButton extends React.Component {
  constructor(props) {
    super(props);
    this.state = {count: 1};
  }

  shouldComponentUpdate(nextProps, nextState) {
    if (this.props.color !== nextProps.color) {
      return true;
    }
    if (this.state.count !== nextState.count) {
      return true;
    }
    return false;
  }

  render() {
    return (
      <button
        color={this.props.color}
        onClick={() => this.setState(state => ({count: state.count + 1})}>
        Count: {this.state.count}
      </button>
    );
  }
}
```

Figure 5: CounterButton component. Screenshotted from [16].

From the figure above, the **shouldComponentUpdate** lifecycle is used to decide the re-render of button component. The re-render will only happen when the color props and count state is changed.

2.2 Programming Paradigms

Javascript is a multi-paradigm and dynamic programming language. It supports more than one programming paradigm and different software design pattern, including object-oriented, imperative, functional, reactive programming and many others. Even those not originally designed to support it from the beginning like other languages like Java or Ruby [17, 333]

2.2.1 Imperative Programming

Imperative programming is a programming paradigm that uses a sequence of instructions to complete a certain objective. Each instruction which is known as statement change program's state every time it was executed following orders. Imperative programming is the most natural way of programming for a majority of first taught programmers.

```
var total = 0;
var a = 5;
var b = 10;
total = a + b;
```

Figure 6: Example of imperative programming.

As described in the figure above, first three set of statements assign a value to each variable. By the end, final statement instructs the program to take the sum of a and b and assign the result value to the total variable.

2.2.2 Functional Programming

Functional programming is a declarative programming paradigm that builds software by composing computation process as mathematical functions, manipulates but avoid mutate data and shared programming states. To be short, the output of a programming function should behave like a mathematical function, will only depend on changing the value of inputs/arguments no matter how many time got executed. The output data must stay same if the input is the same. The outcome of operation should also be predictable.

By increasing immutability of state, data and avoid side effects, the result of functional programming is more concise output, readable code, easy to understand, test and debug.

Below is a simple example of functional programming:

```
// value changes if timing changed, make execute order changed

var x = {
  val: 2
};

var x1 = () => x.val += 1;

var x2 = () => x.val *= 2;

x1();
x2();

console.log(x.val); // 6

x2();
x1();

console.log(x.val); // 5

=====

// In functional programming, timing independent
var x = {
  val: 2
};

var x1 = x => Object.assign({}, x, { val: x.val + 1});

var x2 = x => Object.assign({}, x, { val: x.val * 2});
// compose two function
console.log(x1(x2(x)).val); // 5
```

Figure 7: Different in execute order changed result of functions. Modified from [18]

The figure above explained the difference when writing code using functional programming and other programming paradigms without immutable data. In FP, the state/properties are not changed during the computation process, or the order of execution, the value of x is always equal 2.

2.2.3 Reactive Programming

Reactive programming is a programming paradigm that designed around data flows, event-based and continuously respond to external input changes [19, p.1] To be in short, reactive programming is programming with asynchronous data streams [20]

A stream is a list of continuous events happening in an amount of time. In reactive programming, everything such as events, actions... occurring over a certain amount of time can be stream-able.

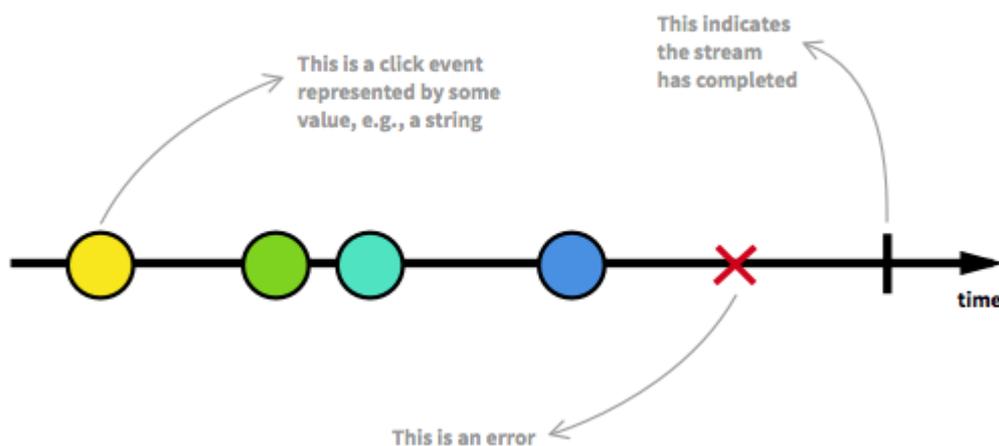


Figure 8: A stream of events. Copied from [20]

In programming, event stream can be various, such as mouse input events, sequence of HTTP requests or server actions. There is always a connection between event emitter and event listeners. When there is event happened in a moment of time, all event listener will reactively response to that change.

All events after being converted will behave as one and single stream, a source of continuous data. For practical use, for many specific usages, stream should be transformed

into multiple streams, merged or filtered into a single stream with the desired output. Therefore, the result will enable the ability to perform complicated tasks on multiple streams at once.

2.2.4 Functional Reactive Programming

Functional Reactive Programming is a combination of reactive and functional programming. To be exact, FRP is reactive programming build on top of functional building blocks [19, p.8]. Reactive programming introduces the idea of combining sequences of events into a single continuous stream. Functional programming provides functions to manipulate those streams. Functional programming provides the higher order functions: map, reduce, filter, merge... to transform data stream.

This figure below demonstrates the usage of functional programming on transforming data streams:

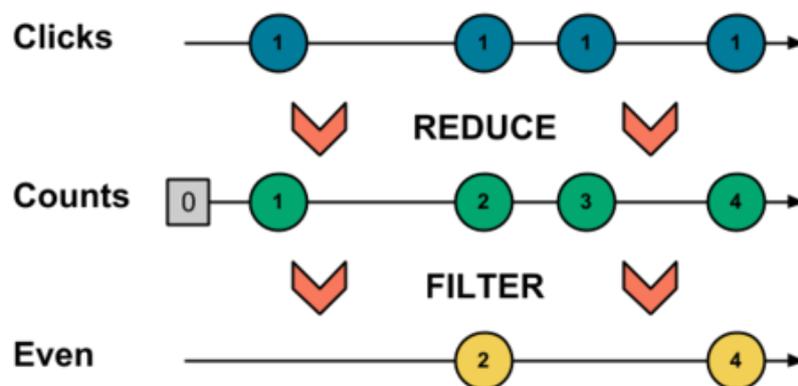


Figure 9: Data stream manipulations. Copied from [21]

2.3 Observer Pattern

2.3.1 Definition

Observer software pattern defines a one to many corresponding relationships between one object called **subject** and list of many other objects called **Observers**. Observers depend on the subject. Every time there is changes in states of the subject, all the list of dependents will get notified and respond to that changes by synchronizing its state with the state of the subject. [22, p.326]

The action of listening to changes from Observer to subject call **publish-subscribe** where the subject was known as notification publisher. The subject does not know how many subscriber (Observer) that listen to its change. Whenever there is changes, subject automatically dispatch notification to all subscribers. [22, p.327]

Below is a diagram shortly explain Observer Pattern

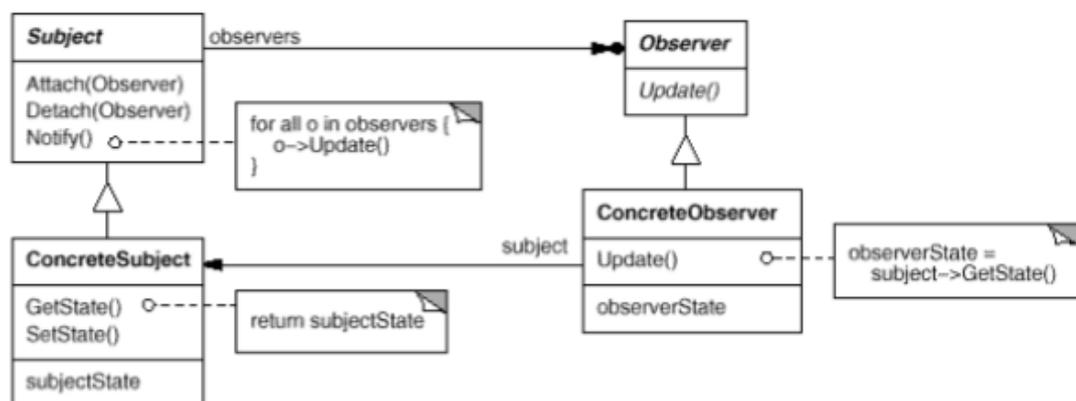


Figure 10: Structure implementation of Observer Pattern [12, p.328]

2.3.2 Observer Pattern and Functional Reactive Programming

FRP's concept of propagating changes during program execution time has principle similarity with Observer Pattern. Both FRP and OP are different in general, one is programming paradigm, one is design pattern. But both share a same concept, except FRP related to streams while OP concern about changes of the object.

By using both, functional reactive programming should have observable streams of events call Observable Streams. [19, p.39]

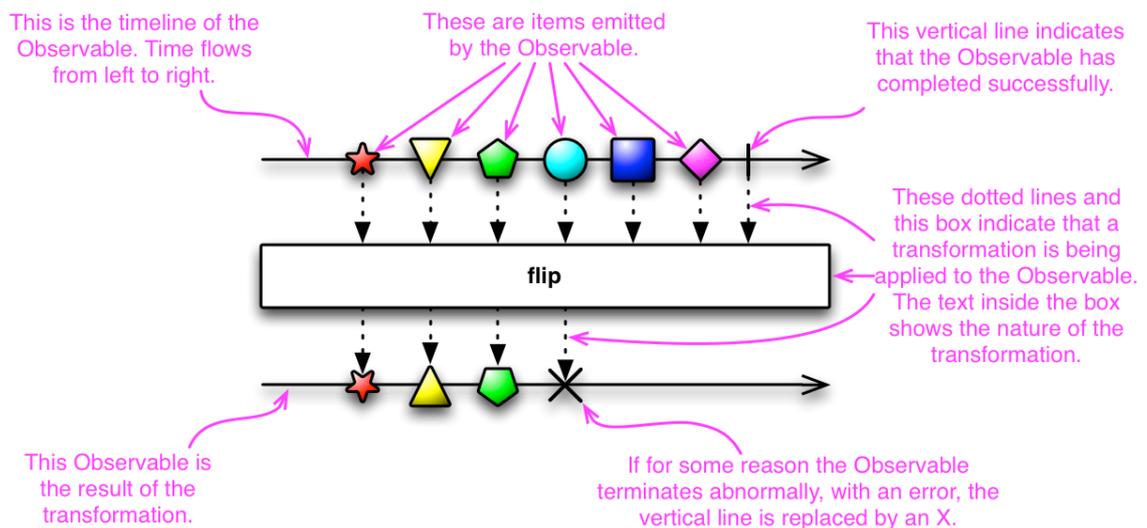


Figure 11: Diagram of Observables transformation. Copied from [23]

The diagram above is an example of transforming one Observable to another. Instead of defining a list of instructions, chaining and execute them in order, waiting to handle the response to achieve demanding purpose to perform next action, instructions can be all constructed as one Observable. Events can be fired in order without knowing their side effects. Outputs were captured by a list of event listeners – known as separated Observers. Whenever each of the processes is finished, notifications were sent to their observers, by calling its method [23]. Therefore, every event is independent and can execute parallel.

Observable have all properties which were inherited from the stream. Observable can contain one or sequence of many events happened in a certain amount of time. Same with stream, Observable can also be combined as one or split into multiple Observable based on user's needs using many query operators.

3 Implementation

3.1 Section Overview

As explained from the introduction, the objective of this thesis is proposing software solution for Tsampo's challenges. This section will be divided into two big section, focus on applying the benefits of both React and FRP to resolve most of Tsampo's problems.

The first part is restructuring Tsampo's interface in a scalable way with React components: plan of splitting UX design into multiple components, turn stateless, reusable component into functional component, stateful component into reactive stateful component.

The second part is restating current performance issues that Tsampo is facing and propose solutions based on using help from external libraries written with FRP concepts.

3.2 Building Scalable User Interface with React

3.2.1 Structure Tsampo Application in React's components

The first point to think when starting to build React application and structure the user interface is the ability to divide the application into multiple components. Instead of creating a big component that only works for one purpose, thinking of splitting into multiple small parts. Well-splitting component increases chances to reuse and repeat components in the application, reduce code duplication and also better component's unit testing.

The figure below is an example of separating front page of application's UX design into components

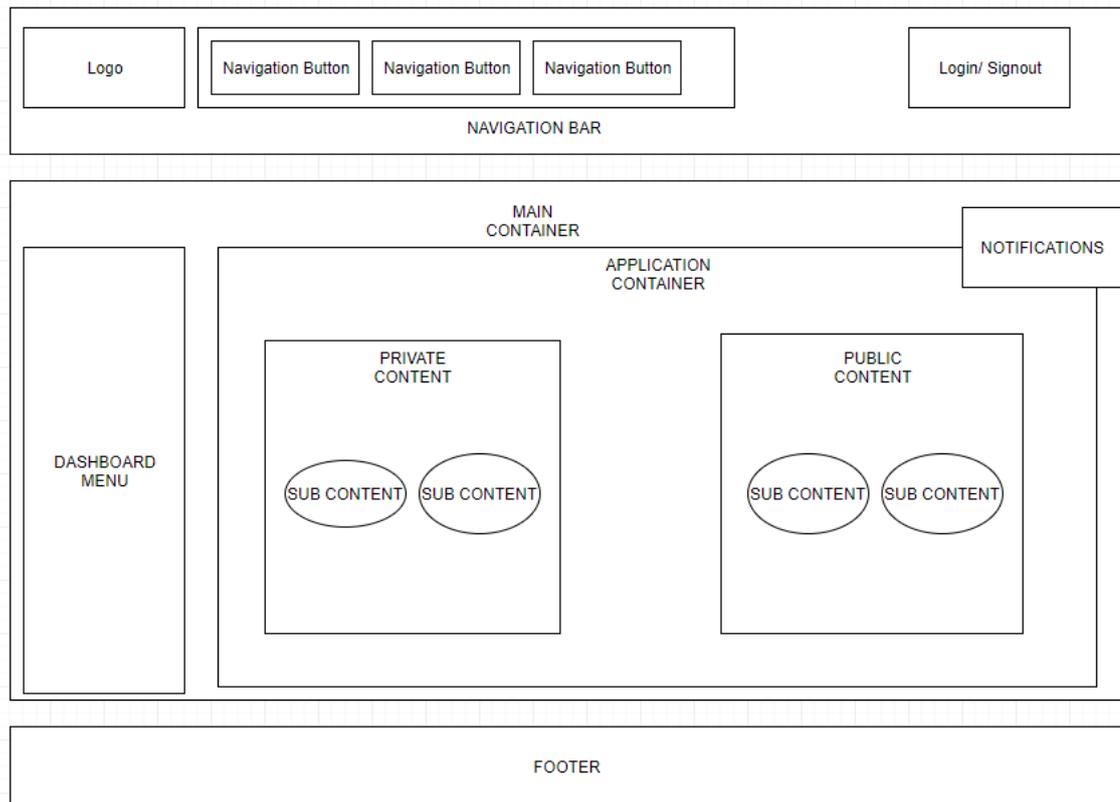


Figure 12: Thinking in component in Tsampo's Frontpage.

By default, React application always appeared with the biggest component as the first container, usually named **App**, wrap all other component and attached to DOM **root** node as the start.

From the figure above, App contains all child components: application's header, main container, notification, and footer. Those components are mandatory parts of every view, and will not change the position, but the content will react and render differently base on each view's purpose.

As one example: **Notification** is a global bubble, always stay invisible until there are needs of displaying user's notifications. For example request errors, request success, login status, actions...etc.

React application's interface is a hierarchy of React components, which can be easily described using a graphical tool: React Developer Tools Extension for Chrome, developed by Facebook React team.

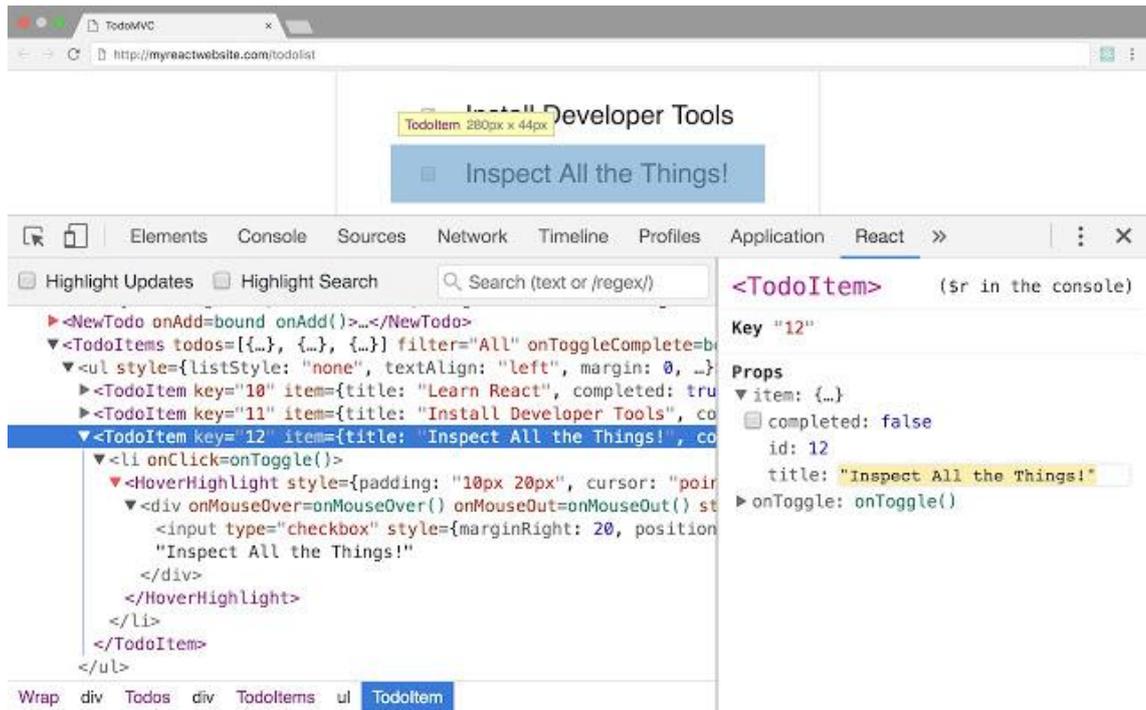


Figure 13: Chrome React Developer Tool snapshot. Copied from [24].

The tool helps visualize React component's structure, component's properties, props, and state. As a consequent, debugging and component's inspection is done easier without putting a breakpoint or console log.

3.2.2 Stateless Functional Component

Instead of defining a React component by extending **React.Component** class, which is not friendly to read and understand. With the help of ES6 functionalities, React component's declaration can be done in a functional way using arrow function, which is a new syntax for writing function expressions.

By using this way of transformation, React stateless component is simplified and functional. The component can be seen as a pure function, taking props as input, processing and return predictable results while keeping the props unchanged.

The figure below is React stateless component written in a functional way.

```

1 import React from 'react';
2
3 class HelloWorld extends React.Component {
4   constructor(props) {
5     super(props);
6   }
7
8   sayHi(event) {
9     alert(`Hi ${this.props.name}`);
10  }
11
12  render() {
13    return (
14      <div>
15        <a
16          href="#"
17          onClick={this.sayHi.bind(this)}>Say Hi</a>
18      </div>
19    );
20  }
21 }
22
23 HelloWorld.propTypes = {
24   name: React.PropTypes.string.isRequired
25 };
26
27 export default HelloWorld;

```

```

1 import React from 'react';
2
3 const HelloWorld = ({name}) => {
4   const sayHi = (event) => {
5     alert(`Hi ${name}`);
6   };
7
8   return (
9     <div>
10      <a
11        href="#"
12        onClick={sayHi}>Say Hi</a>
13    </div>
14  );
15 };
16
17 HelloWorld.propTypes = {
18   name: React.PropTypes.string.isRequired
19 };
20
21 export default HelloWorld;

```

Figure 14: Difference between functional and non-functional component. Copied from [25]

Functional component and non-functional component are basically the same when compiled. But functional component is by far easy to understand, shorter, easier to test and higher performance because of no non-necessary lifecycle methods.

From the example above, **HelloWorld** component's general purpose is outputting a **Hi** message to whatever input come in **name** variable as props. Component's expression in the functional way on the right side is shorter in code, showing clearer component as a function with the **name** as an argument.

Most of stateless functional component can be reused across application is small components, as an essential part of building bigger components. For example: Buttons, Messages, Input, Links...etc. Those components can be customized for specific purposes through props, state editing and values, creating from scratch of importing from many React libraries.

The figure below is an example of creating one reusable React component call SvgIcon.

```
const SvgIcon = (props) => {
  const {children, className, height, viewBox, width} = props
  return (
    <svg className={classnames('icon', className)}
      viewBox={viewBox || '0 0 36 36'}
      width={width || 36}
      height={height || 36}>
      {children}
    </svg>
  )
}
```

Figure 15: SvgIcon component.

SvgIcon is a component to create SVG responsive icons with options to customize the class name, width, height, content by changing the content of children component, usually SVG position.

Using those kinds of the component by far increasing chance of reusing codes and making the process of maintaining and developing Tsampo's application easier in the future.

3.2.3 Reactive stateful component

As normal, a stateful React component is usually written in this following way.

```

class Timer extends React.Component {
  constructor(props) {
    super(props)
    this.state = { timesClicked: 0 }
  }

  handleClick() {
    this.setState((prevState) => ({
      timesClicked: prevState.timesClicked + 1
    })))
  }

  render() {
    return (
      <div>
        <div>Times Clicked: {this.state.timesClicked}</div>
        <button onClick={this.handleClick.bind(this)}>Click Me</button>
      </div>
    )
  }
}

```

■ Logic (behavior)
■ Structure (presentation)

Figure 16: Classical React component. Copied from [26].

From the figure above, this way of expression makes React component's declaration mix between the logic and the structure, reduce the code readability and functional unit testing. Update/re-render is made when **setState()** is invoked manually every time **handleClick()** function is called. The component does not know when to update.

```

@observer
export class Timer extends Component {
  @observable timeClicked = 0

  render() {
    return (
      <div >
        <div>Times Clicked: {this.timeClicked}</div>
        <button onClick={this.timeClicked++}>Click Me</button>
      </div>
    )
  }
}

export default Timer

```

Figure 19: Reactive stateful component Timer.

By using FRP observable, React component is reactive. Component's declaration is shorter, clearer, easier to understand without **handleClicked()** function and **setState()**. The component as an observer subscribes for changes from **timeClicked** observable, automatically update/re-render if observable is changed.

3.2.4 Higher Order Component

HOC is based on the concept of higher-order functions, which is a benefit of bringing functional programming to the process of building a user interface. Higher-order functions take a function, process and return a function. HOC behave the same way, taking a component as input, do some process and return a component. Taking HOC into practice can be seen through benefits of container wrapper and authenticate HOC.

Container wrapper is a normal component, act as the parent of others component. Container wrapper takes responsibilities to connect and fetch data from application's state store or outside application and deliver it to child component through props. By using this, component's logic part of pulling resources such as: fetching customer's data, fund request, access roles, journal, documentation... can be reused and customized easily outside of the child component's scope. Therefore, keeping the child component simple, easier to understand and test.

Below is a simple diagram explaining the logic of component wrapper.

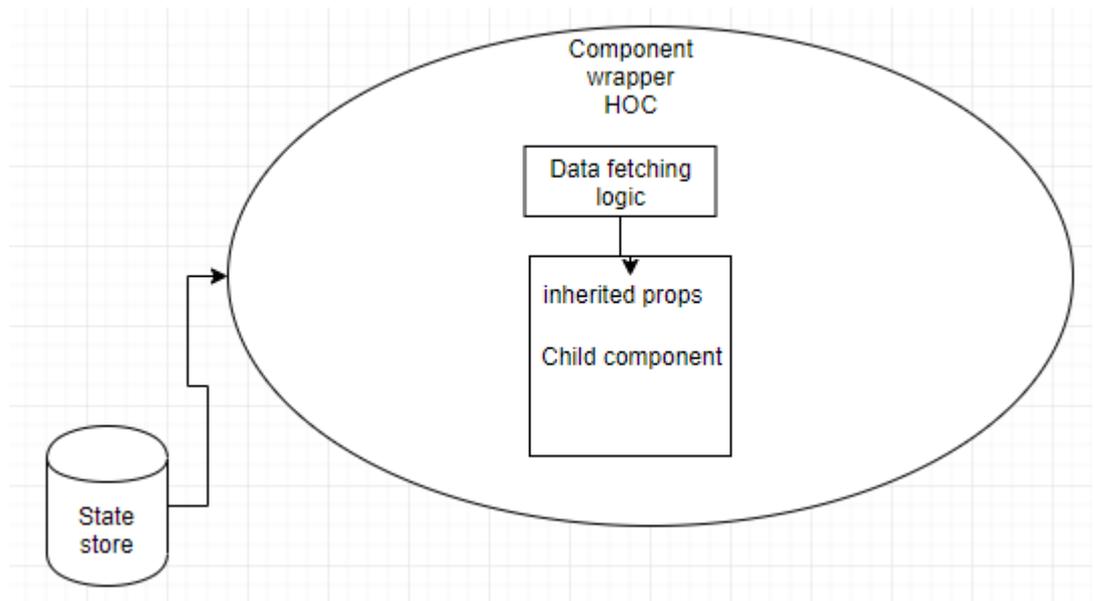


Figure 17: Component wrapper diagram.

HOC can be inspired, also be applied to implement Tsampo's authenticate flow. Inside application, there are some specific routes that require user credentials or user logged in before going forward, some route only open for certain kind of user with specific permissions...etc. Those routes are called, for example **ProtectedRoute**. HOC for authenticate purpose is simply checking the condition, if the condition is met, return the route. Otherwise, deny access, redirect user to login page.

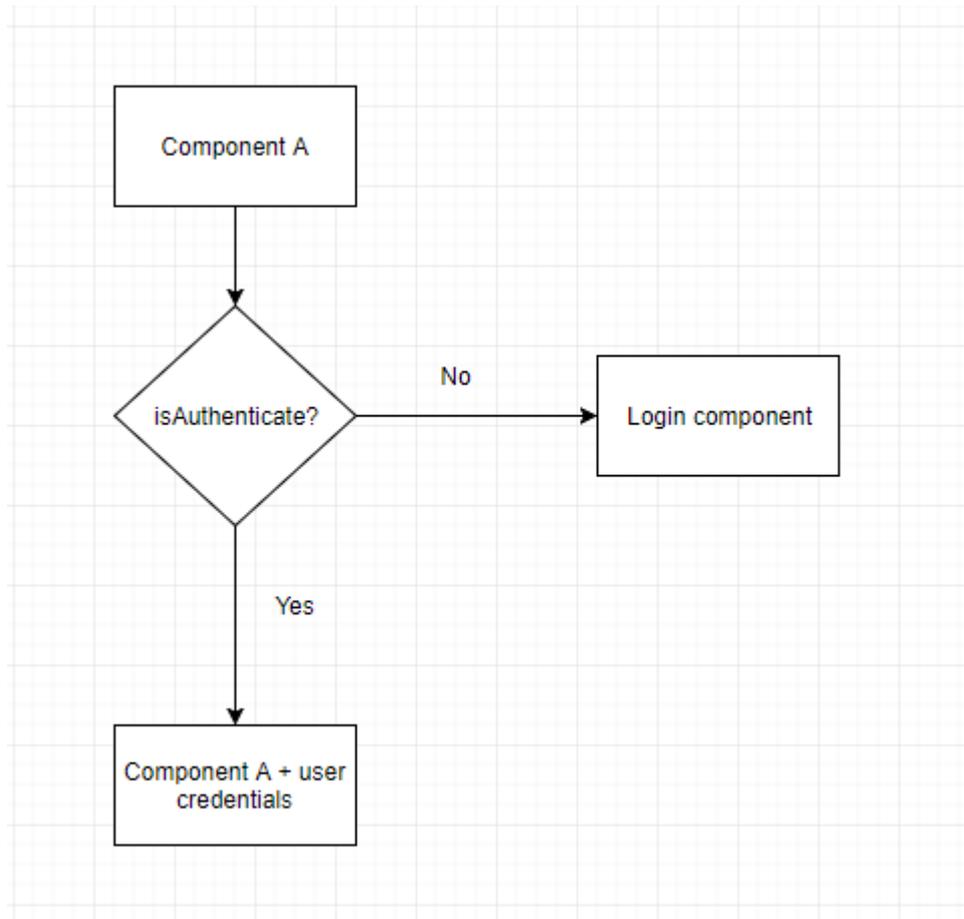


Figure 18: User Authentication using HOC.

3.3 Performance Improvement

As explained from overview section. This section focuses on improving Tsampo's performance by using benefits from two main libraries: RxJS and Mobx.

RxJS is a library of operators, utilities using for transforming streams. RxJS is written based on reactive programming's concept, using Observables [27].

Mobx is one of state management library for React. Mobx's core idea is also based on FRP, but different from RxJS. Mobx took care of React state and component's rendering instead of providing functionalities to compose streams.

3.3.1 Optimize performance with React component's lifecycle.

React component react to changes by doing re-render in case of internal state changes or props changes. But not all those changes are necessary and need to perform a re-render. A re-render is a costly action because the component not only re-render itself, wrapper container might contain children, and its child components have to be re-render as consequent.

As an example of optimizing performance, Tsampo application form is designed with a mechanism to dispatch save user's form value action temporarily in case the user is idle for longer than one minute (60 seconds).

An internal counter is started when component is mounted and destroy on unmount by using **componentWillMount()** and **componentWillUnmount()** lifecycle's method, to prevent memory leak by not destroying the repeated action.

The counter starts counting action after mounted, causing state changes in every minute. In order to reduce the risk of performance by massive re-rendering, **shouldComponentUpdate()** method was used, and only allow to make re-render when the current timer is equal 60 seconds as shown in the figure below.

```
public shouldComponentUpdate(nextProps: ApplicationFormProps, nextState: ApplicationFormState) {
  if (nextProps.currentTime === 60) {
    this.submitChanges()
  }
  if (nextProps.currentTime !== this.props.currentTime) {
    return false
  }
  // prevent unnecessary re-render from counter
  return true
}
```

Figure 19: Using lifecycle method to improve performance in Tsampo's application form.

Therefore, the component will only re-render once when the timer is equal the limit from now on.

3.3.2 State management with Mobx

From the theoretical background, React is fast with **reconciliation** process - finding the state updates through Virtual DOM and apply those changes to real DOM. Reconciliation is triggered whenever component **render()** method is called on the component. The process work not only on the component's level but also propagate to component's child and children of that child recursively to find the updates. This is fine for a small application with small codebase and structure, there is almost no difference can be seen from the user interface. But for a large scale application with heavily nested DOM structure like Tsampo, the process can be really expensive, causing performance issues.

Taking examples of displaying user's application list, the state changes on one application should only trigger reconciliation process on itself or the children, instead of the whole list.

One of React's lifecycle method **shouldComponentUpdate()**, which is used in the last chapter is one of the solution to reduce the cascade of rendering. **shouldComponentUpdate()** basically check the component's old props and state and compare to the new one to decide if the render should happen or not. If the data is not immutable data, which is not mutable from beginning, finding the differences can be challenging. The time processing for **shouldComponentUpdate()** depends on the data structure of props and state, the algorithm to find the differences, multiply by the number of the component that needed the calculation as well.

Mobx is state management library for React, based on FRP's implementation. Mobx introducing using observable in React. The idea behind Mobx is using state as observable, the state can be stored in one or multiple store. Store is the place to keep application's state. React component is **observer** which render, observe and react to that state from store. Whenever the state is changed, all component that shares the same state will automatically update rendering base on that change.

Mobx **@observer** decorator replace **shouldComponentUpdate()** [28], control the rendering of the component by only allow re-render to happen when observer notified with changes from observable. By far reducing un-necessary re-rendering of the component and all sub-component as child.

tsampo.com

tsampo Grant applications Profile Share knowledge Logout

Unfinished applications

Group application: "The effect of xyzyzy when applied in foobar" **Complete, waiting to be sent** Open
created 25.8.2017

Personal application: "Study of deviousness of various furry pets" **Incomplete** Open
created 25.8.2017

Create a new grant application

Sent applications

Personal application: "Orbital properties of slingshotted birds" Retract View
created 22.7.2017

Archived applications

Group application: "Coffee sweetening solutions" Create new from this View
created 1.1.2016

Figure 20: Tsampo application list.

The figure above is application list of Tsampo application, the status, title of the application, number of application is derived from application's state. The update came will affect only part of this UI. For example, edit on group application's name will only make that application re-render. Because that is the only place the group data is used, even the component is a part of the list. The reconciliation process is limited to each component, without cascading to child component, which is not displayed in this view.

3.3.3 Handle concurrent HTTP requests.

From the introduction part, one of most challenging factor Tsampo application facing is getting various types of data from different sources. Customer's data is fetched from universities, institutes.. and other trusted sources of data sciences.

HTTP requests are an asynchronous event and hard to predict. During the development process, Tsampo's interface needs to display many user data as one, which means a lot of requests has to be made. A large amount of request is fired concurrently consume a

large amount of computer's resources. Old devices with less processing power and low network conditions have worst user experience, the application is frozen and slow.

Moreover, all request URL is stored and treated equally as a part of an Array, there is no error handler specialize for each of the requests.

By using FRP, the issue can be resolved by transforming a long array of document's URL into a stream, fetch one by one after another using helps from RxJS operators.

Below is a step-by-step explanation for resolving concurrent requests:

- RxJS **from()** method transform sources data from Array to Observable stream, pass URL from the source stream one by one to **concatMap()** method. **concatMap()** contains **fromPromise()** operator inside.
- **concatMap()** ensure the request is done one by one, not concurrently. Because **concatMap()** only subscribe to next observable until the previous one is totally finished. [29]
- Later, each URL is piped to **fromPromise()**, which contain **fetch()** method inside to execute HTTP request
- HTTP GET request is executed using Javascript **fetch()** method, return the response as a Promise. The outcome of **fromPromise()** is an Observable.
- The output stream is supposed to be Observable of Observables but the result is flattened down to Observable of responses with **concatMap()** helps
- During the process, if one of the requests gets failed, **concatMap()** will not subscribe to next observable. The error can be catch and display to the interface.

After all, the end results are similar: a collection of URL data. But all the request is done in a different way: one after another in a waterfall shape, without stacking up the

memory and reduce performance. The error is also handled and displayed correctly to each request.



Figure 21: Waterfall of HTTP requests one after another. Screenshot from [29]

3.3.4 Throttle click events and user input

Multiple click events and input changes from multiple users on a live-editing document of Tsampo gave system ton of actions during short about of time. Stacks of actions without debouncing or throttling is queued and have to be processed by the system. The outcome is long processing time and collections of update that the system can not decide which and when to display.

The idea is only to focus on click event only after certain amount of time. By this way, amount of click event is reduced. Therefore, less action has to be processed and updated in the future. Stress is reduced on the system and performance is improved.

Take an example of reducing non-necessary search request. Instead of sending search request with search query every time search input box is changed, the change event can be debounced by 1000ms. The debounce time to make sure the request is sent when user has finished typing or stopped to see the results.

The figure below is a visual explanation of how debouncing is done using RxJS **debounce()** operator

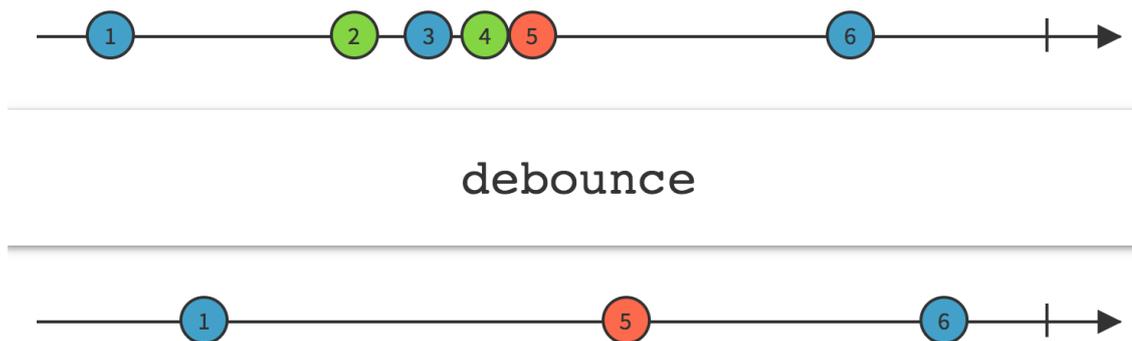


Figure 22: Visualize stream transformation of debouncing. Copied from [31]

Below is the implementation of the solution:

- Pipe all input-change events into a stream of events. Using RxJS **fromEvent()** operator and select input box by **#search-box** id, with event as **keydown**.
- Apply a **debounce()** with value as 1000ms to the source stream
- Get output stream, make request with **fetch()**, **fromPromise()** and **concatMap()** as explained from last chapter.
- Get search result and display the updates to the interface.

Minimizing number of non-necessary action, update and HTTP requests means a lot with performance improvement, especially for a multi-user environment like Tsampo.

4 Conclusion

Despite many difficulties during implementation and planning process, the final objective of this thesis was met. A prototype version of Tsampo application has been successfully created using React and applied good architecture from many aspects of functional reactive programming. The challenges coming from a large amount of data sources and complex user's behaviors are also resolved. The codebase is also easy to understand, tested, and scalable in the future by Tsampo's new developers, also their upcoming projects related. The company now able to use the application to gather information from investors and scientists who interested in joining the first pilot test.

However, the application plan is heavily affected by technical decisions that delay deployment time of the company. The prototype was shipped to the user one month later than the initial plan. The vision of applying functional reactive programming increase the scalability of the project is needed but might not be put as the first priority. FRP cost a big amount of time to study, investigate and refactor the codebase to reach the desired purpose.

In order to reach the early stage of deployment, the quality of the code, also usability have been put behind the timeline. For production use in the future of completed application, further development should consider user-friendly as mandatory, also enhance in code review and writing tests between developers.

References

- 1 Tsampo. Tsampo Video Campaign for Raikaisu 100. [online]. URL: https://youtu.be/EB_gsZeZRzk. Accessed 14 December 2017.
- 2 Korotya E. 5 Best JavaScript Frameworks in 2017. [online]. Hackernoon. 19 January 2017. URL: <https://hackernoon.com/5-best-javascript-frameworks-in-2017-7a63b3870282>. Accessed on 14 December 2017.
- 3 Occhino T. React Native: Bringing modern web techniques to mobile. [online]. Facebook. 26 March 2015. URL: <https://code.facebook.com/posts/1014532261909640/react-native-bringing-modern-web-techniques-to-mobile/>. Accessed on 14 December 2017.
- 4 Harrington C. React vs AngularJS vs KnockoutJS: a Performance Comparison. [online]. Codementor. 13 January 2015. URL: <https://www.codementor.io/chrisharrington/react-vs-angularjs-vs-knockoutjs-a-performance-comparison-85hwzepbz>. Accessed on 14 December 2017.
- 5 Facebook. React Homepage. [online]. React. URL: <https://React.org/>. Accessed on 14 December 2017.
- 6 Facebook. Thinking in React. [online]. React. URL: <https://React.org/docs/thinking-in-react.html>. Accessed on 14 December 2017.
- 7 Javascript HTML DOM. [online]. W3schools. URL: https://www.w3schools.com/js/js_htmlDOM.asp. Accessed on 14 December 2017.
- 8 Artemij F. React.js Essentials. Packt Publishing Ltd; 2015.
- 9 Krajka B. The difference between Virtual DOM and DOM. [online]. React Kungfu. 12 October 2015. URL: <http://reactkungfu.com/2015/10/the-difference-between-virtual-dom-and-dom/>. Accessed 22 October 2017.
- 10 Facebook. Reconciliation. [online]. React. URL: <https://React.org/docs/reconciliation.html>. Accessed on 21 December 2017.
- 11 Peyrott S. React Virtual DOM vs Incremental DOM vs Ember's Glimmer Fight. [online]. Auth0. 20 November 2015. URL: <https://auth0.com/blog/face-off-virtual-dom-vs-incremental-dom-vs-glimmer/>. Accessed 22 October 2017.
- 12 Components and Props. [online]. React. URL: <https://React.org/docs/components-and-props.html>. Accessed 22 October 2017.

- 13 Facebook. Why did we build React?. [online]. React. URL: <https://React.org/blog/2013/06/05/why-react.html>. Accessed on 14 December 2017.
- 14 Facebook. Components, Props and State. [online]. React. URL: <https://facebook.github.io/react-vr/docs/components-props-and-state.html>. Accessed on 8th April 2018.
- 15 React.js Tutorial: React Component Lifecycle. [online]. Codevoila. URL: <https://www.codevoila.com/post/57/React-tutorial-react-component-lifecycle>. Accessed 22 October 2017.
- 16 Facebook. Optimizing Performance. [online]. React. URL: <https://React.org/docs/optimizing-performance.html>. Accessed on 21 December 2017.
- 17 Hayward J. Reactive Programming with JavaScript. Packt Publishing Ltd; 2015.
- 18 Elliott E. Master the JavaScript Interview: What is Functional Programming? [online]. Medium. 4 January 2017. URL: <https://medium.com/javascript-scene/master-the-javascript-interview-what-is-functional-programming-7f218c68b3a0>. Accessed 22 October 2017.
- 19 Doglio F. Reactive Programming with Node.js. Apress; 2016.
- 20 The introduction to Reactive Programming you've been missing. [online]. Github. URL: <https://gist.github.com/staltz/868e7e9bc2a7b8c1f754>. Accessed 22 October 2017.
- 21 Epping S. Functional Reactive Programming (FRP). [online]. Zweitag. 05 March 2015. URL: <https://www.zweitag.de/en/blog/technology/functional-reactive-programming-frp>. Accessed 22 October 2017.
- 22 Gamma E, Helm R, Vlissides J, Johnson R. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley; 1994.
- 23 Observable. [online]. ReactiveX. URL: <http://reactivex.io/documentation/observable.html>. Accessed 22 October 2017.
- 24 React Developer Tools. [online]. Chrome Google Extension. URL: <https://chrome.google.com/webstore/detail/react-developer-tools/>. Accessed on 11 November 2017.
- 25 House C. React Stateless Functional Components: Nine Wins You Might Have Overlooked. [online]. Hackernoon. 29 March 2016. URL: <https://hackernoon.com/react-stateless-functional-components-nine-wins-you-might-have-overlooked-997b0d933dbc>. Accessed on 11 November 2017.

- 26 Dom K. What happens when you use RxJS in React? [online]. Hackernoon. 30 January 2017. URL: <https://hackernoon.com/what-happens-when-you-use-rxjs-in-react-11ae5163fc0a>. Accessed on 8th April 2018.
- 27 Reactive.io. RxJS. [online]. Reactive.io. URL: <http://reactivex.io/rxjs/>. Accessed on 12 April 2018.
- 28 Sharon K. React Performance and Mobx. [online]. Medium. 21 February 2018. URL: <https://medium.com/workday-engineering/react-performance-and-mobx-b038085ecb72>. Accessed on 12 April 2018.
- 29 Learnrxjs.io. concatMap. [online]. Learnrxjs. URL: <https://www.learnrxjs.io/operators/transformation/concatmap.html?q=>. Accessed on 10 April 2018.
- 30 James S. Understanding Marble Diagrams for Reactive Streams. [online]. Medium. 29 December 2017. URL: <https://medium.com/@jshvarts/read-marble-diagrams-like-a-pro-3d72934d3ef5>. Accessed on 10 April 2018.
- 31 Tomas T. Practical RxJS In The Wild. [online] Hackernoon. 19 December 2017. URL: <https://blog.angularindepth.com/practical-rxjs-in-the-wild-requests-with-concatmap-vs-mergemap-vs-forkjoin-11e5b2efe293>. Accessed on 10 April 2018.