Bachelor's thesis

Degree programme in Information and Communications Technology

2020

Anh Tu Le

Developing a web application for task management with REACTJS

TURKU AMK
TURKU UNIVERSITY OF
APPLIED SCIENCES

BACHELOR'S THESIS | ABSTRACT

TURKU UNIVERSITY OF APPLIED SCIENCES

Degree programme in Information and Communications TechnologyDegree programme in Information and Communications Technology

2020 | 35 pages

Author: Anh Tu Le

DEVELOPING A WEB APPLICATION FOR TASK MANAGEMENT WITH REACTJSDEVELOPING A WEB APPLICATION FOR TASK MANAGEMENT WITH REACTJS

The commissioner of the thesis is Vertics which is a consultant software company. Vertics offers business services from turnkey projects to flexible resourcing assistance and all aspects of software development. Their client is an art production company,which currently works with about ten organizations in terms of Art. The client has much experience in ticket solution, marketing strategy, they would like to make a Subscription Builder web application. The application is based on timeline to show tasks that belong to events of marketing strategy to help their customer to achieve the purpose of increasing loyal subscribers for their events. Thus, the first objective of this thesis was to develop a time-line based web UI and flow for task management.

Simultaneously another objective of this thesis was to research the fundamentals of ReactJS to support the development of the website. ReactJS was chosen as main technology for the project because it is a popular Javascript library for front end development. The ReactJS library was originally created by a developer in Facebook to support for their team. After proving its strengths and high performance, ReactJS has been open-sourced and has received more powerful contributions from the ReactJS community.

The final result of this thesis project met the requirements of Vertics, which were to obtain an optimal web application with a user-friendly layout. The application is now processed for further development with other functions. Throughout the study and software development, the project helps the reader gain a deep understanding of React JS and how ReactJS works in practice.

KEYWORDS:

ReactJS, Components, Redux, JSX, Javascript, Front-end, Virtual DOM,HTML.

CONTENTS

FIGURES

PICTURES

LIST OF ABBREVIATIONS

DOM	Document Object Model
JS	Javascript
MVC	Model - View – Controller
SPA	Single Page Application

1 INTRODUCTION

The client of Vertics is a company that works intensely with approximately ten performing arts organizations annually. Their goal is to help their customers to reach larger audiences, increase their earned income, stabilize operations, and pursue greater artistic freedom. Over the years they have worked with database and ticket solutions, programming, marketing strategy and brand development, as well as strategic organizational and leadership development. They want to build a Subscription Builder website, with which they aim to dramatically increase the base of traditional fixed-seat, fixed-performance subscribers for their partners. The website is basically a management system for customer of Vertics's client to handle tasks as marketing strategy to grow the customer base of loyal subscribers and take control of the capacity of their halls. The website must be high speed and have a user-friendly layout that artists can easily use and make changes to.

ReactJS is applied in this project to create a Single Page Application that meets all the requirements from the art production company. ReactJS is a popular JS library to build a website that interacts with users by dynamically rewriting the current page rather than loading entire new pages from the server. It is used by a wide range of developers. A research study implemented by Stateofjs reveals that two third of developers who are asked about using front-end development tools, choose ReactJS and would use again.[1] Additionaly, ReactJS applications have best SEO practices to boost ranking companies in top positions in a certain search engine. With the advantage of the capability to render on server side using Nodejs, ReactJS helps search engine crawlers see the web application in its final form, make it handy for them to index it correctly.[2]

ReactJS library helps to build a quick response and high performance website, create more productivity than other frameworks in the same field and also reduce the maintenance costs for the system.

2 REACT CORE CONCEPT

ReactJS is a JavaScript library for building user interfaces in single page application. It creates a virtual DOM and allows to create reusable components[3]. It was first published by Facebook for their Facebook ads. Gradually, because of its strengths, Instagram, Netflix, Airbnb have used ReactJS to resolve their UI problems. ReactJS helps developer to work with UI inside JS code . The integration between HTML and Javascript into a file, which is a component in ReactJS, helps the structure of a core element to be clear to understand.

A ReactJS web is a compilation of distinct components, each of which defines a separate view on the web structure. The benefit of this solution is to be easier to reuse code lines for the same view since the component works independently. Additionally, data can be changed in the component without conflicting entire system. Applications created by ReactJS help business enterprise to gain more user interactions and benefits.

2.1 Virtual DOM

DOM is a programming interface for HTML and XML documents. It represents the page that the program can change the document structure, style, and content. Figure 1 shows how DOM is applied in web application. The DOM shows documents as nodes and objects. It contributes a way for JS to deal with any node in the nodes tree and manipulate it. When working with a web application, DOM manipulation is important to render HTML to browser.

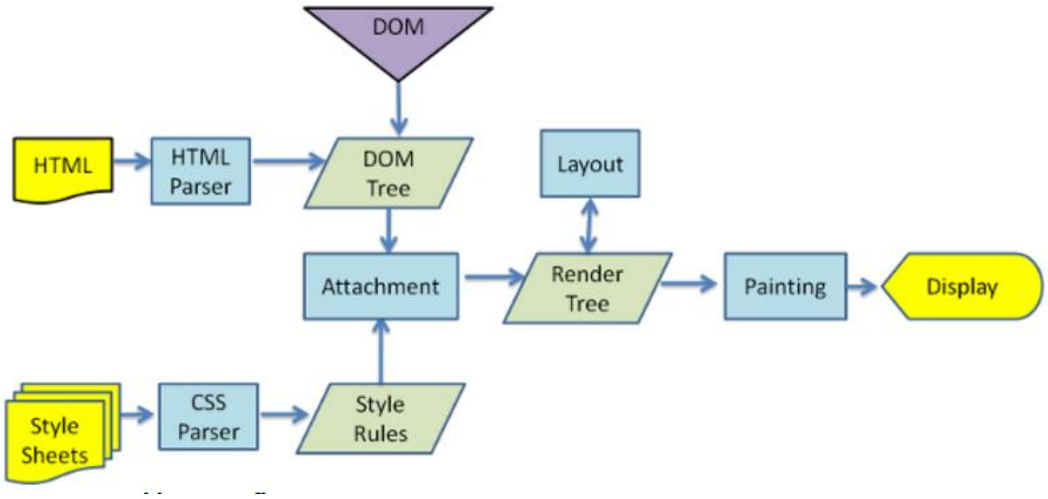


Figure 1. DOM manipulation in modern web application.[4]

The contents of a requested document are read by a parse engine. All the elements in HTML file become nodes in nodes tree, which is known as the DOM tree or "content tree". Next , styling is parsed to merge with DOM tree to be an attachment, which is used to create a

render tree. As soon as the render tree exists, a layout process begins to construct elements by giving each node a coordinate or address on the web. After that, this render tree traverses the painting process to help every element transform with corresponding styling.[4]

ReactJS

The main reason that makes the DOM expensive and reduces performance is that most JS frameworks update the DOM much more than necessary. For example, let us assume that there is a list of 100 items and the developer wants to update the first item. The result is that the entire list is re-rendered by the DOM, which is 100 times more than necessary, which is a waste. Moreover, modern browsers use the v8 engine, which compiles and executes JS code. Since it is a single threaded execution engine, 100 times of updating DOM at the same time make the browser work more slowly and reduce the execution result. Although it does not have real power as a real DOM, virtual DOM manipulation helps ReactJS to skip the real DOM manipulation and boost the process much more faster.[5]

ReactJS uses virtual DOM to find the node that changes and updates it on the real DOM as in Figure 2, because implementing in virtual DOM is simple, fast and inexpensive, same as updating an object. Every ReactJS stateful component is included with the render method to create a virtual DOM. After the component is updated, the new virtual DOM is created and the previous version become a snapshot. ReactJS implements a method to efficiently update the DOM base on both virtual DOM versions.[6]

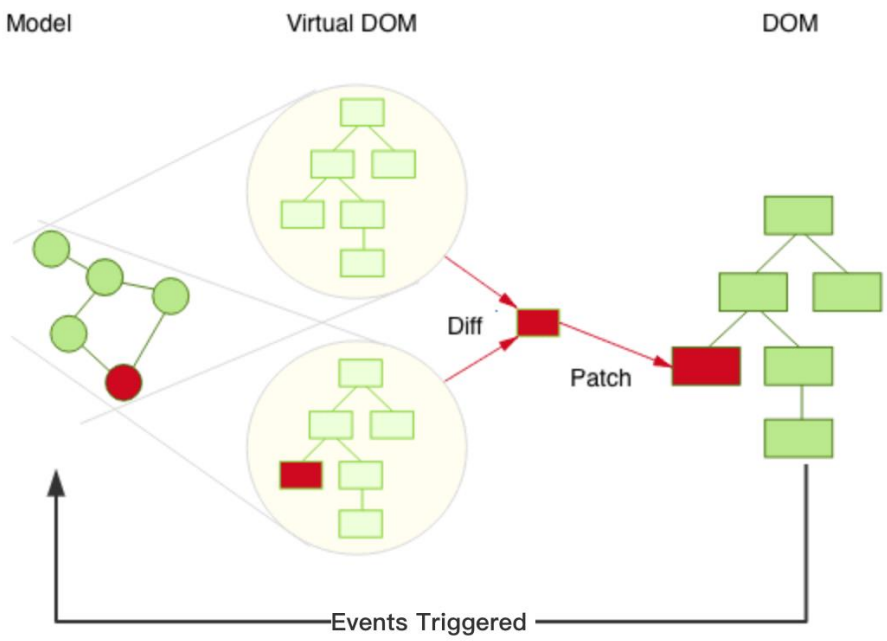
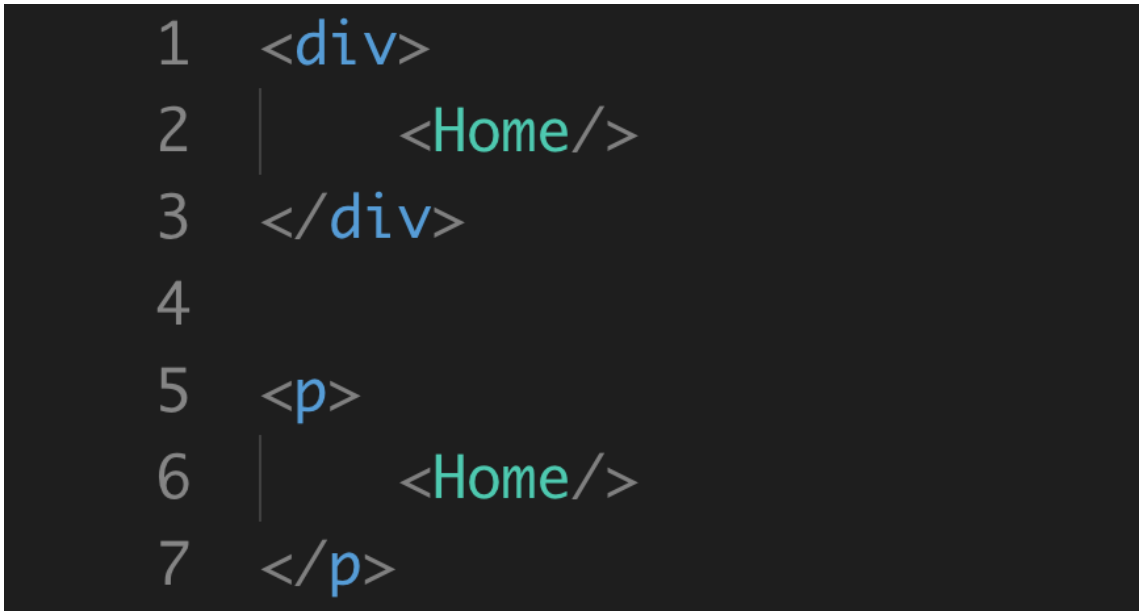


Figure 2. How React deals with virtual DOM.[6]

Diffing algorithm

ReactJS compares the current and the snapshot virtual DOM to find the minimum operations that changed between the 2 trees and update the corresponding objects in the real DOM. In a generic situation, changes in trees are expensive to find out in tree algorithm. It usually takes $O(n^3)$ operations to obtain the result. The real case, if there are 100 items, takes 1000000 comparisions. This is inefficient. In ReactJS, the Diffing algorithm, applies heuristics and reduces the calculations to O(n) which is competent in speeding issue.[7] The diff algorithm is based on 2 assumptions:

- Elements of different type will produce different trees. Picture 1 reveals that when the DOM updates from a div tag to a p tag, the entire subtree is re-rendered. ReactJS does not compare the sub tree if the sub nodes have been changed. Therefore, it replaces the old subtree with the new one.



Picture 1. ReactJS builds a new subtree in the case root.

- Child components are identified with key property. This assumption is very helpful to let ReactJS recognize which children components exist. Picture 2 shows that it is challenging for ReactJS to know what codes are changed in the UI. ReactJS updates all children without knowing that it can keep <Item>Samsung</Item> and <Item>Apple</Item>. This situation leads to worse performance.

```
<div>
    <Item>Samsung</Item>
    <Item>Apple</Item>
</div>

<div>
    <Item>Nokia</Item>
    <Item>SamSung</Item>
    <Item>Apple</Item>
</div>
```

Picture 2. Components without identity.

With key identity, ReactJS knows that Item with key number 1 and 2 exist, and Item with key number 3 is a new child component, as shown in Picture 3.

```
<div>
    <Item key="1">Samsung</Item>
    <Item key="2">Apple</Item>
</div>

<div>
    <Item key="3">Nokia</Item>
    <Item key="1">Samsung</Item>
    <Item key="2">Apple</Item>
</div>
```

Picture 3. Components with unique identity.

2.2 Props and State

A ReactJS applications are structured by nested components. The component itself is built as a function: it takes data through arguments, which are known as Props and transfer data by returning that value. **Props** is a special keyword in **React**, which stands for properties and is being used for passing data from one component to another. Furthermore, **props** data is read-only, which means that data coming from the parent should not be changed by child components.[8] Picture 4 shows clearly how props are passed to children.

```
4  class Person extends React.Component{
5      render(){
6          return <div>{this.props.name}</div>
7      }
8  }
9  const Person=({name})=><div>{name}</div>
10
11 <Name name="Tu"/>
```

Picture 4. How props are passed to children.

The props are passed as a reference to the children.Therefore, it is not changed. The following example illustrates the concept of props. For example, a school gives students a book as document to research about Finnish culture. All they have to do with the book can be: read the book, draw in the pages of the book,..but the students can not change it except in the case when the school gives them another book.

There is also a case that props are not received from a parent component. This means that the props are set as default in the beginning before the component is renderred. These special props are known as state, which is created inside the component. State is used when a value of data need to be changed inside the component. Same as props, state stores the data of the component but the way it is handled is different and more complicated. Normally a component that does not have state is called stateless component which is displayed in Picture 5

```
const Item=({})=>{
    return <span>Book</span>
}
```

Picture 5. Stateless component.

However, to keep track of information or data, state is needed in this case. For example, a form with inputs of text. If the inputs are changed by users, the state of input fields are changed. Picture 6 shows that the input takes the hard-coded state to user for the initial value.

```
class MyForm extends React.Component{
    constructor(props){
        super(props)
        this.state={
            name:'Tu'
        }
    }
    handleOnChange=e=>{
        const {name,value}=e.target
        this.setState({name:value})
    }
    render(){
        return <input name="name" value={this.state.name} onChange={this.handleOnChange}/>
    }
}
```

Picture 6. Stateful component.

As input can be changed all the time, state is mutable to update the current value for the UI. The difficulty is that when the value of input is changed by a callback function, the current value is changed but the UI still keeps the lastest version since nothing toggled the page to re-render for updating new UI. ReactJS does that part by creating the setState() function to set s new value to state object and automatically re-render the component to obtain the lastest UI with the new value. This part is taken care fast by ReactJS that the developers do not need to deal with the UI re-rendering problem .

2.3 JSX – Javascript XML

JSX is HTML-like code which describes the DOM nodes in logic part of the app, Javascript files. Unlike in the past, when developers had to use Javascript code in HTML file, JSX lets developers put HTML in Javascript files. It is a ReactJS extension to help write Javascript code that looks like HTML.[9] It is not required to use JSX in ReactJS application but JSX make

ReactJS easier to handle UI and logic in same file. For not using JSX as line 5 in Picture 7, it is complicated to render a short element in DOM node by Javascript.

```
3  import React from 'react'
4
5  React.createElement("div", { className: "red" }, "I love React");
6  <div className="red">I love React</div>
```

Picture 7. Example of using JSX.

In line 6 of Picture 7, JSX is applied to make the structure clean and easy to read. JSX can help to build the structure with familiar knowledge about rendering UI such as HTML. As JSX is a JS extension, it is compiled in Javascript code by Babel, which is a Javascript parse engine, to make sure the browser understands the structure of the ReactJS app.

Writing JSX is exactly the same as HTML. The only difference is that JSX can not use the “class” and “for” words, which are reserved keywords in Javascript. Therefore, JSX changes it to “className” and “htmlFor”, which is shown in Picture 8:

```
3  <div className="red">I love React</div>
4  <label htmlFor="text">This is text</label>
```

Picture 8. Two different points of JSX in comparison with HTML structure.

JSX is much clearer and easier for reading and writing HTML in a complicated structure. Moreover, the ReactJS team recommend to use JSX for a simplicity and easy for code structure in ReactJS applications.[10]

2.4 Component life cycle

A component in ReactJS has 3 main phases in its whole lifecycle: mounting, updating and unmounting until it completely disappeared in the DOM (Figure 3). ReactJS has created many predefined functions for a better interaction with the props and state during each events.[11]

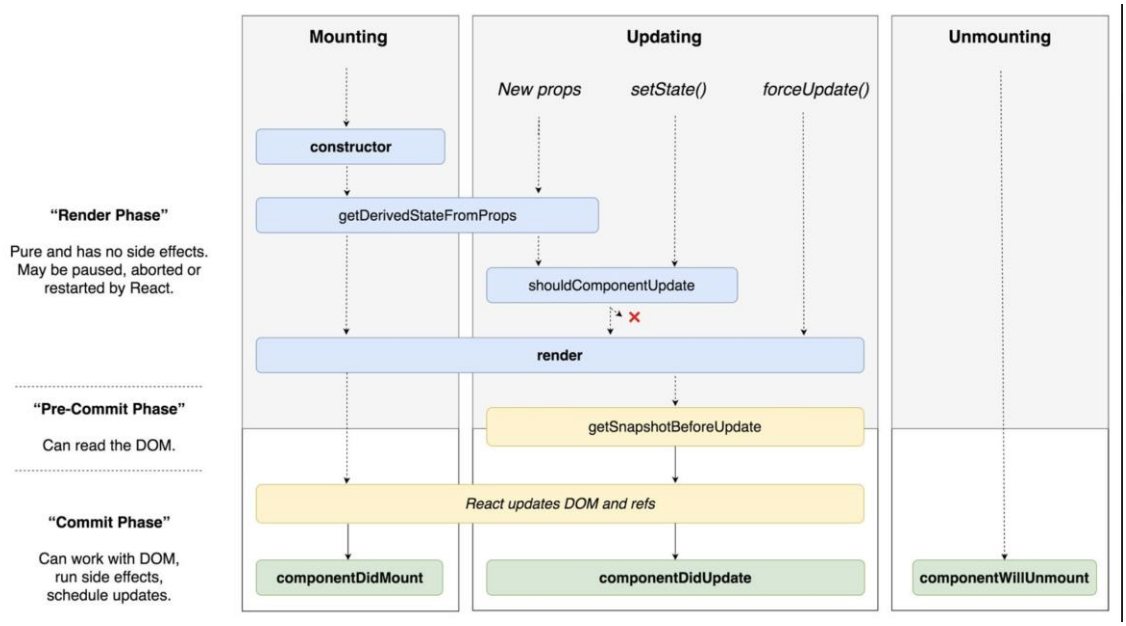


Figure 3.Component life cycle in React.[11]

Mounting phase

A ReactJS component receives props and state to work as its own data. The initialization of these data occurs when the component is about to be alive. Mounting phase is when the component is about to be rendered in the web application. At that time, a series of events is triggered to load and modify the component data. In this phase, events are called in this order: constructor(), getDerivedStateFromProps(), render(),componentDidmount().[11] Picture 9 describes clearly structure of mouting phase code:

```
4  import React from "react";
5  class Person extends React.Component {
6    constructor(props) {
7      super(props);
8      this.state = {};
9    }
10   render() {
11     return <div className="person-container"></div>;
12   }
13 }
```

Picture 9. Mounting phase in ReactJS component.

In Picture 9, constructor() is the first event to be called to setup data of the component. Normally, props and state are created at this point. Props from the parents are recognized by calling super(props) to initiate constructor method and let the component to inherit all the props from outside.

The next step is the stage for getDerivedStateFromProps().It takes state as argument to return the object which changes the state.

After the data are initialized, HTML elements are assigned to the DOM. This is implemented in the render() function. This event is required to render the UI in the webpage for every class component. JSX is used in this function to construct HTML elements in the component. Render() is the most important event in the lifecycle. It should be pure and avoid using any API fetchings or setState() functions.

ComponentDidMount() is the last event in the phase when HTML has already been mounted to the DOM. With this method, the elements are catch to restyle or change the content, or even interacting with API to retrieve? new data.

Updating phase

When the component is alive and exists in the DOM, any actions that lead to the change in props or state bring the component to new life, which is the updating phase. For this phase, the component is equipped with 5 different events: getDerivedStateFormProps(), shouldComponentUpdate(), render(), getSnapshotBeforeUpdate(), componentDidUpdate(). [11]

At this point of lifecycle, changes in data make the UI updated. To avoid unnecessary re-rendering, ReactJS gives a special event that can manage when the view of the component is updated by using shouldComponentUpdate(). This event returns the boolean value as true or false to determine whether ReactJS should keep going with rendering or not.

The important and common period of this phase is data manipulation in componentDidUpdate(). After all the changes are updated in the DOM, setState() is used to update for new view for the component. The component should be wrapped in a condition to check for changes in compare with the previous props and state. Incorrect usage of setState() leads to infinite loop and the app will be broken.

Unmounting phase

The only event reserved for this phase is componentWillUnmount(), which is fired just before HTML is removed from DOM and component is destroyed. This is an event for ReactJS to erase all the data of the component. setState() should not be used because it is useless when the component is disappeared. [11]

3 DATA MANAGEMENT – REDUX

Redux is a state management tool for JS application to help with a better solution for storing the state of the whole application in just one object, which is store. It helps to control data that is rendered in the UI and how to respond to actions coming from client side.

ReactJS has its own way to manage state of the component by tranfering data from parent to children component. It is simple when applying this principle in a small application. In reality, growth of application leads to the fact that data of the app goes huge and make it difficult to manage. Redux comes to help storing all the data in a global object with ability to share the benefit to all components in the application

3.1 Reason for choosing Redux

All JS library and frameworks such as ReactJS, Angular, VueJS,..are build by components with their own state. Therefore, components can manage their own data easily in a simple application. When the application goes big, transferring data is very difficult to developers.

In Figure 4, it is an application whose nodes are components. If there is an action in node d3 and is triggered to change on state of c3 and d4:

- How data is tranfered d3 to c3: d3-c2-b1-a-b2-c3
- How data is tranfered d3 to d4: d3-c2-b1-a-b2-c4-d4

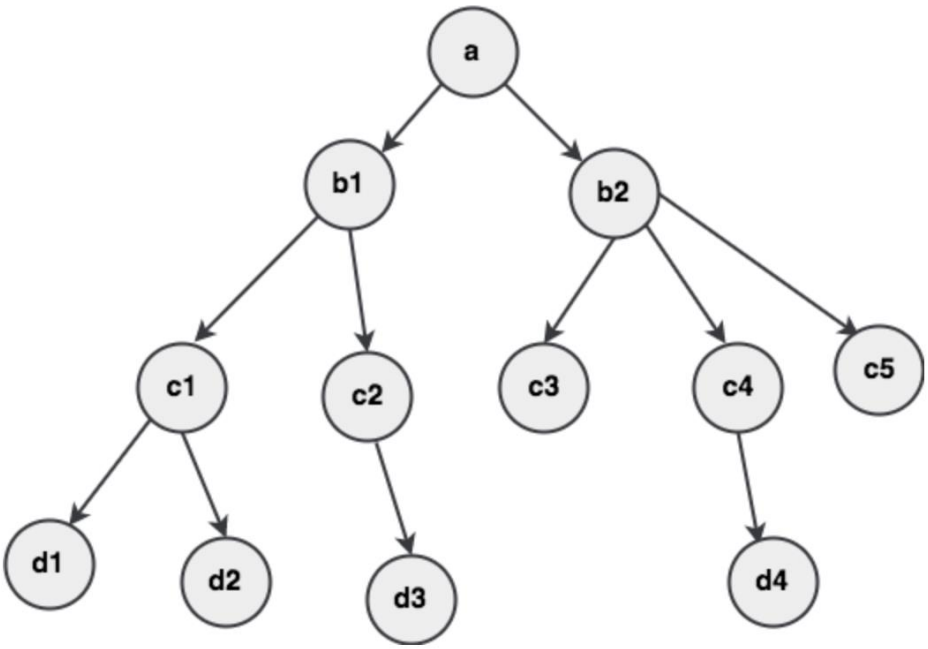


Figure 4. Data management by React.[12]

By using Redux, a store, as a global object, holds state of all the components in one place, then update the data automatically without moving data in a complicated way. In Figure 5, action in d3 is sent to store. Node d4 and c3 need to connect to store to update the change in data and it is all done.

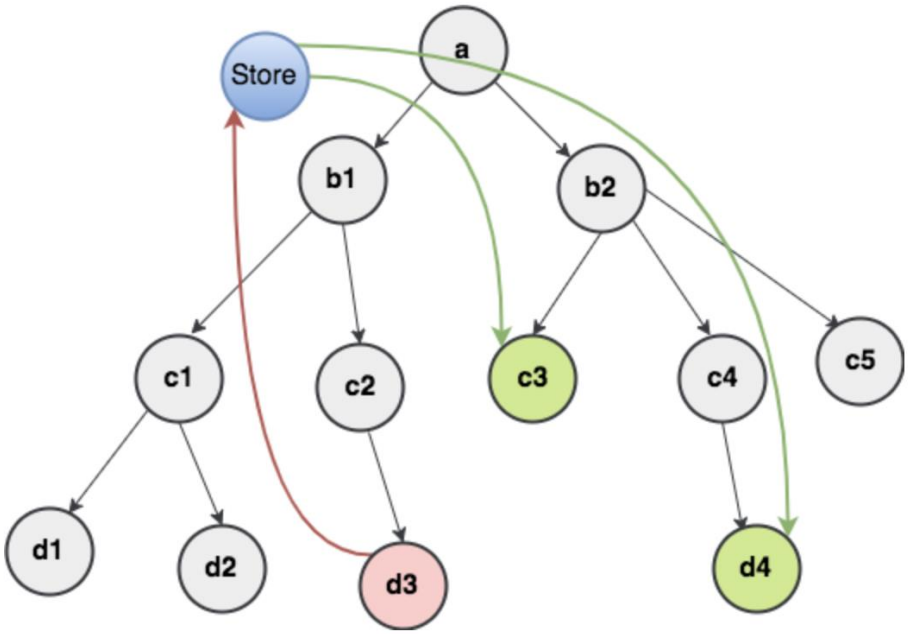


Figure 5. Data management by Redux. [12]

3.2 Principle of Redux

Single source of truth [13]

This principle means that data is taken from the only source, which is store, the global object that keep data of application. In another word, it is one app – one store – one state. Storing data in one place also helps it easier for data debug and inspection.

State is read-only [13]

In another word, the state is immutable. Redux is strict to actions that interact with the data. The only thing to make changes on data is to emit an object as an action to describe what happens to the state. This technique is to avoid direct changes from server fetching to data in store. Picture 10 illustrates structure of object emitted to reducer.

```
17  const login = data => ({
18      type: types.login,
19      payload: {
20          request: {
21              method: 'POST',
22              url: `/login`,
23              data
24          }
25      }
26  })
```

Picture 10. Object as an action to modify data in store.

Changes are made with pure functions [13]

When there are changes on state object, Redux does not let us to modify the state directly. A new object is made and modified depend on how the action describes in reducer. The state is changed by a new updated object. Redux keeps old state intact.

Reducer are pure funtions which take old state and action as arguments, and return next new state (Picture 11). Action is filtered in reducers to guide the reducer what to do with the old state.

```
const initialState = {
    hlrkey: null,
    email: '',
    password: '',
    loading: false,
    error: null
}
export const loginReducer = (state = initialState, action) => {
    switch (action.type) {
        case types.changeField:
            return {
                ...state,
                [action.name]: action.value
            }
```

Picture 11. Pure function as a reducer in Redux.

3.3 Understanding how Redux works

The way how Redux works is very simple. It has a Store where to keep the state of the application. Each component can have a connection with the Store to get the right to directly access the state instead of sending data throught components.

Redux has 3 main parts: Actions,Reducers and Store

Actions

An action is just a plain JS object as in Picture 12 that describes what happen in changing of state. It is created by an action creator, which is a pure function that return an object as an action.

```
const changeField = (name, value) => {
    return {
        type: types.changeField,
        name,
        value
    }
}
```

Picture 12. An action in Redux.

When an action is triggered in a component, it means that a plain JS object is dispatched, to Store and the rest are handled by Reducer. In the actions, the only required field is the type of the action. This type defines what kind of action dispatched to Store. The type should be constant. Other fields are optional in case there is a need to send data to the Store for further manipulation of state.

Reducer

Picture 13 reveals structure of a reducer. Reducer is a pure JS function which takes current state of application to deal with an action and return a new application state. This function basicly define how the state changes. Whenever an action has been dispatched to the Store, it goes through the Reducer. Reducer receives previous state of the app and action as arguments. The Reducer depends on the type of the action to calculate the change in next state in compare with the old state.

```
26  export const tasksReducer = (state = initialTasksState, action) => {
27      switch (action.type) {
28          case types.changeField:
29              return {
30                  ...state,
31                  [action.name]: action.value
32              }
```

Picture 13. A reducer in Redux.

Store

Store is where to keep the application state. Althought it is theoratically to create multiple store, but it will against Redux principle as one source of truth, there should be one store in an application. Connecting the component to Store helps to subscribe and listen to the changes, which can lead to updating in the component that need to be re-rendered.

Store is also a JS object and is known as “state tree”. It is possible to add unlimited fields that need to be used for our application

Data flow

Figure 6 displays data flow diagram in Redux.[14] Although Redux is included with many concepts, data flow in Redux is simple.

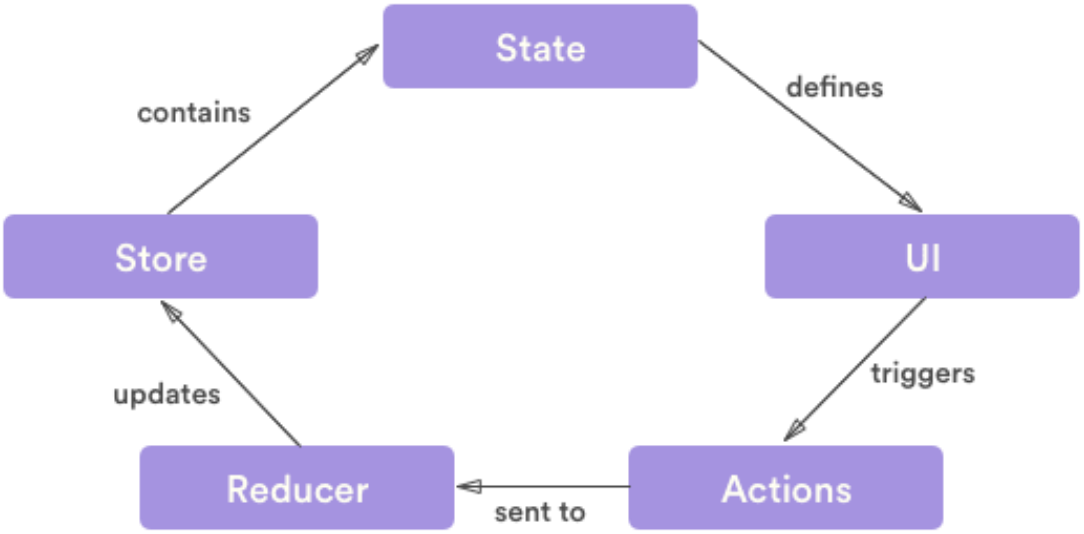


Figure 6. Data flow in Redux.[14]

When an action is called in a component, it is dispatched to reducer of the store. In the reducer, data that attached in actions can be used for state manipulation. A new state object is returned to Store. Since the application subscribes to Store, the UI is updated accordingly to what the action defined.

This data flow is called unidirectional data flow. The flow makes the application easier for maintenance and debugging. It has a clear structure so that developers can keep track of where the data come from, also they do not have to deal with multiple independent copies of data that are unaware of another.

4 IMPLEMENTATION

4.1 Setup the project

Choosing editor

There are some code editors that are designed specifically for writing and editing code lines in front-end development such as Visual Studio Code, Atom, Sublime,.. Each of them has its props and cons. Visual Studio code are free,light-weight and have a nice user interface. Atom and Sublime provides a lot of coding themes but packages need to be downloaded to support the best of coding experiences.[15]

In this thesis, Visual Studio Code is chosen to implement the Subsciption Builder application. With its build-in Git support and terminal, fast extension installation, and well-formatted with Prettier extension, Visual Studio Code is a good tool to enhance the speed of development progress.

Package management

Node package management (NPM) is a tool to create and manage a JS library. In Javascript community, developers share thousand of libraries or simply a component that can be reused for a specific UI or functionality. It helps new project to avoid rewriting a basic component, library or even a framework such as: ReactJS library, Express,… Npm helps developer to easily run the command, download a library and use it, or to publish a package to share with the community.[16]

This thesis uses Yarn, a package management developed by Facebook. Facebook creates Yarn with ability to use it offline and creating node_modules libraries by same structure of package installation in every machine, where npm is complicated and different.

4.2 Timeline

Requirement

Timeline is the place to show all the activities with specific dates that the plans happen during the year. Those activities are pre-defined to be part of marketing campaign to help customers for getting more subscribers to their concert. Essentially, timeline is a schedule with events that are built by experiences from Vertics's client that bring the best strategies for customers to growth their business from selling more tickets.

Vertics requires this timeline need to follow these criterias:

- Activities in timeline need to show finished and pending activities by different colors. Status of activities will depends on deadline of tasks of the activity
- Activities that are in same phase should be separated with other activities from another phase.
- Each phase should have a status fo define if it is completed or in active or pending
- Each activity should show end date and order of it in the Timeline

Data Flow

The flow of Timeline development is shown on Figure 7:

Figure 7. Data flow of Timeline.

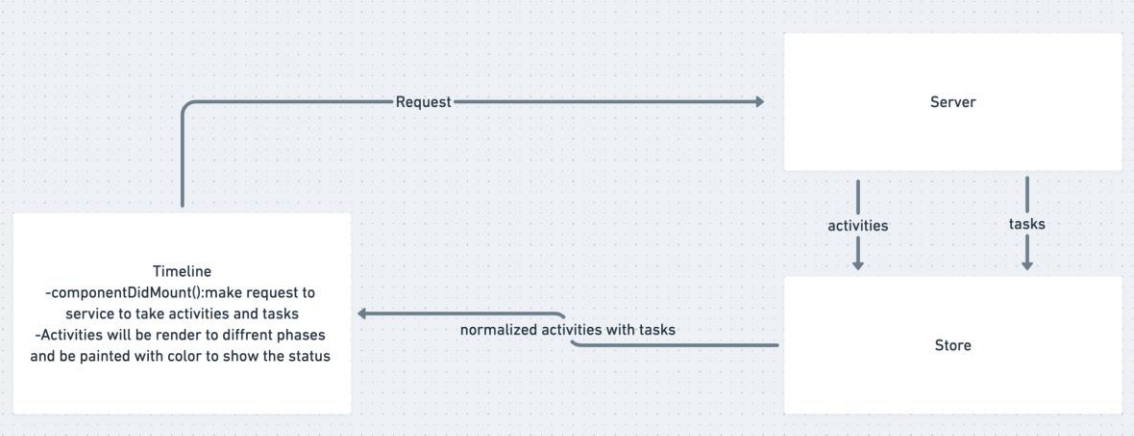


Figure 7. Data flow of Timeline.

When Timeline component is mounted, componentDidMount() makes a request to inform server that the application needs activities and tasks as data to render the UI. After the server responds, data is saved to Store for further usage.

In this case, Timeline is included with mixed data of tasks and activities. Therefore, data sent to Timeline component need to be normalized, which is attaching correct tasks to its activity.

After activities are changed with tasks, they are sent to component to render exact UI follow the needs of Vertics.

Components

- Timeline: timeline code-based structure is shows on Figure 8

```
23  class Timeline extends Component {
24      componentDidMount=()=>{…
26      }
27      componentDidUpdate(prevProps) {…
39      }
40      onActivitySelect = activity => {…
55      }
56      renderContent = () => {…
101     }
102     isCurrentTimePhase = activities => {…
110     }
111     overdueTasks = activities => {…
125     }
126     renderItems = () => {…
173     }
174
175     render() {
176         return <div>{this.renderContent()}</div>
177     }
178 }
179 const mapStateToProps = state => ({…
186 })
187 export default connect(mapStateToProps)(withRouter(Timeline))
```

Figure 8. Structure of Timeline component.

When Timeline is mounted, a request is made to send to the server. The data needed for this component are taken from Store through mapStateToProps() function as props.

In the beginning, when there is no request to get new data, UI is null. After new data arrive, props activities of this component will changed, render() function is triggered again to render new UI with new data taken from Store.

After that, renderItems() function on line 126 of Figure 8 is implemented to render each phases of the timeline through Phase component, which is one of children of Timeline component and is the component to hold all tasks in same session. Figure 9 reveals all the properties that needed for Phase component to render in the webpage. There are 4 main phases which are predefined by the client of Vertics: Preparation and Planning, Internal Campaign, Public Campaign and Customer care. The most important property is isCurrentTimePhase, which includes a progress bar to the phase that currently running by employees.

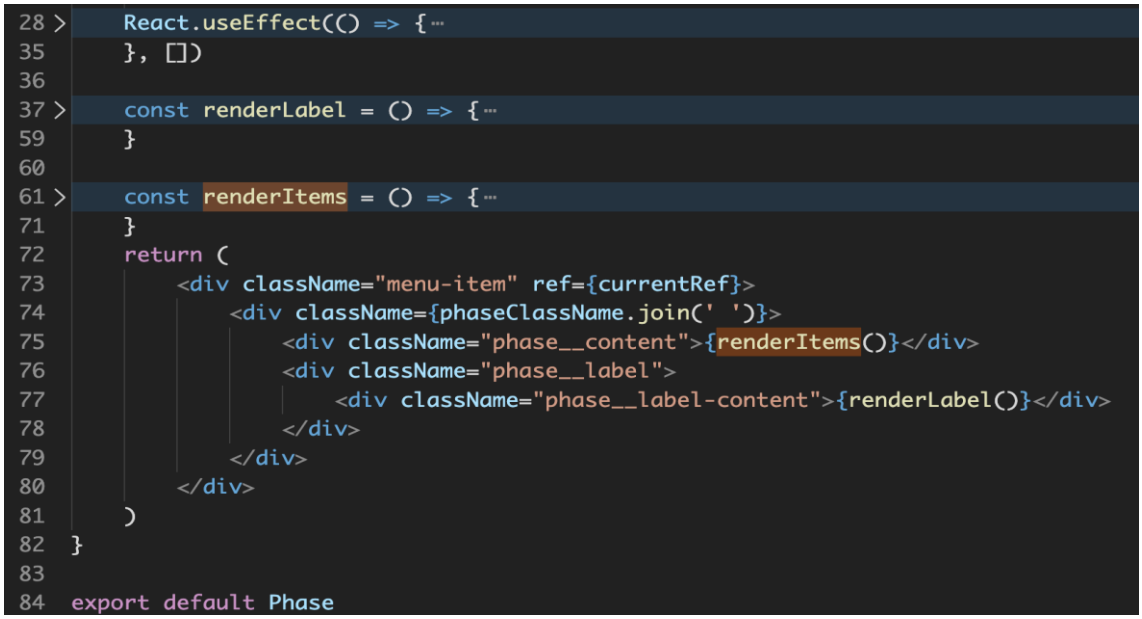
```
126     renderItems = () => {
127         const { selectedActivity, activities } = this.props
128         const normalizeTimelineData = normalizeTimeline(activities)
129         return normalizeTimelineData.map((item, key) => {
130             let doneTask = 0
131 >           let statusArr = item.activities.map(activity => {…
149             })
150
151 >           statusArr.map(item => {…
155             })
156             return (
157                 <Phase
158                     key={key}
159                     name={
160                         key === normalizeTimelineData.length - 1 ? 'I. Prep' : item.phase
161                     }
162                     data={item.activities}
163                     onActivitySelect={this.onActivitySelect}
164                     selectedActivity={selectedActivity}
165                     isPhaseDone={doneTask === statusArr.length}
166                     doneTask={doneTask}
167                     allTasks={statusArr.length}
168                     isCurrentTimePhase={this.isCurrentTimePhase(item.activities)}
169                     overdueTasks={this.overdueTasks(item.activities)}
170                 />
```

Figure 9. Rendering UI of Timeline with activities.

- Phase

Phase component on Figure 10 is basically a wrapper for Activity component , where render information of an activity from the data.

Phase is where stores the logic to define how an activity should be rendered, props for Activity component



```
28 >    React.useEffect(() => {…
35      }, [])
36
37 >    const renderLabel = () => {…
59      }
60
61 >    const renderItems = () => {…
71      }
72      return (
73          <div className="menu-item" ref={currentRef}>
74              <div className={phaseClassName.join(' ')}>
75                  <div className="phase__content">{renderItems()}</div>
76                  <div className="phase__label">
77                      <div className="phase__label-content">{renderLabel()}</div>
78                  </div>
79              </div>
80          </div>
81      )
82  }
83
84  export default Phase
```

Figure 10. Structure of Phase component.

- Activity

Figure 11 shows Activity component, one of basic and reusable component in the project. This component receives props from Phase which are: activity object to get detail information of the activity, onActivitySelect to define the event function from Phase component, seletedActivity id to define which Activity component will be active in UI by different color, and all the tasks of that activity so that this component will validate corresponding UI with correct status.

```
9  const Activity = ({activity,onActivitySelect,selectedActivity,allTasks}) => {
10     if (!activity) {…
12     }
13     const className = ['activity']
14     let tasks = allTasks.filter(
15         task => task.relatedTo && task.relatedTo._id === activity._id
16     )
17     let isLate = false
18     let isDone = true
19     tasks.map(task => {…
30     })
31
32     return (
33         <div
34             className={className.join(' ')}
35             onClick={() => _onActivitySelect(activity)}
36         >
37             <span className="order">
38                 {activity.fields['5dd54490f7f0c47a2af5675f'].value}
39             </span>
40             {renderContent()}
41             {activity.highlight === 'Yes' && (
42                 <div className="notification">
43                     <img src={timeLineIcon} alt="icon" />
44                 </div>
45             )}
```

Figure 11. Structure of Activity component.

Results

The completed UI of timeline component is illustrated generally in Figure 12 and in detail in Figure 13.

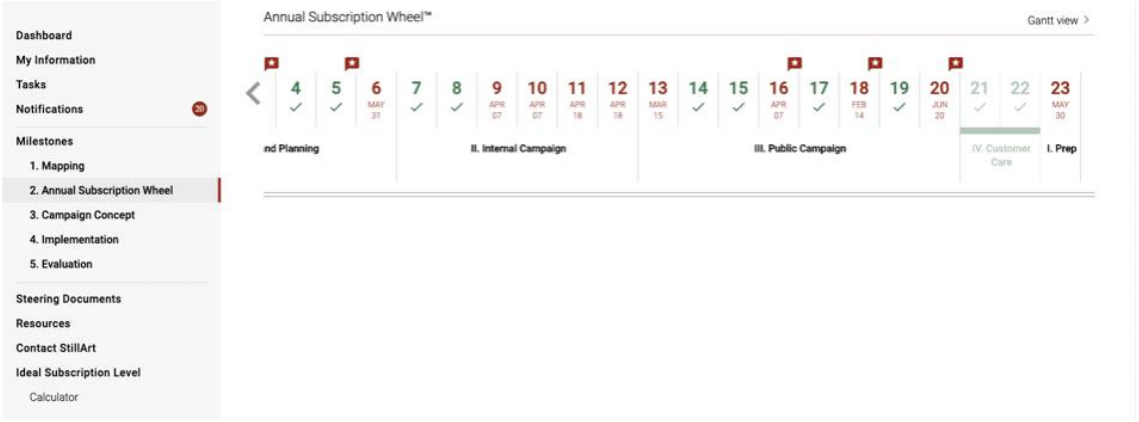


Figure 12. Completed UI of Timeline.

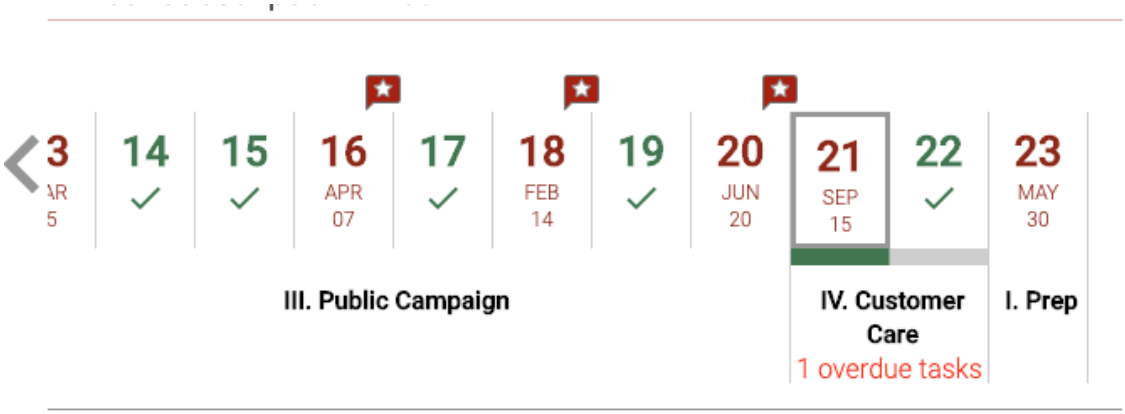


Figure 13. Timeline with indicator of unfinished Phase.

The completed UI for Timeline of the Subscription Builder is colorful to recognize the status of each activities. The green activity shows that all the tasks of that activity have been resolved. The Timeline has four main phases which are : Preparation and Planning, Internal Campainge, Public Campainge, Customer care. Each phase is clearly separate by a light longer verical line in compare with border of activities.

Goals that are reached by this section and make commissioner satisfied:

- Activities has been shown correctly on a scrollable Timeline.
- Activities' status shown by correct colors, which are red for unfinished and green for finish activities.
- Each activity has an icon to show if it has highlight note from the admin.
- Activities are divided by phases nicely and have information of each phase below the activities.

4.3 Task management

Requirements

After activities have been defined to support for marketing campaign, tasks are created to make the activity processable. Each activity is included by tasks that related to. Status of activity depends on status of all the tasks it has. Basicly, user has to finish all the tasks to make activity done and move to the next activity.

Requirements:

- Task details should be shown in popup modal for a better user experiences
- Task modal should have discussion about the workloads, questions.
- Task status can be changed from:Not started, Progress, Waiting for confirmation and Completed.

- Users can be added to the task to follow the process.

Data flow

Timeline, Tasks and Modal components are in same root, which is Board component. Although they are in same component, each component works independently with each others. Triggering events from one component to change data in another component need the existence of same state, which is Store on the right side of Figure 14.

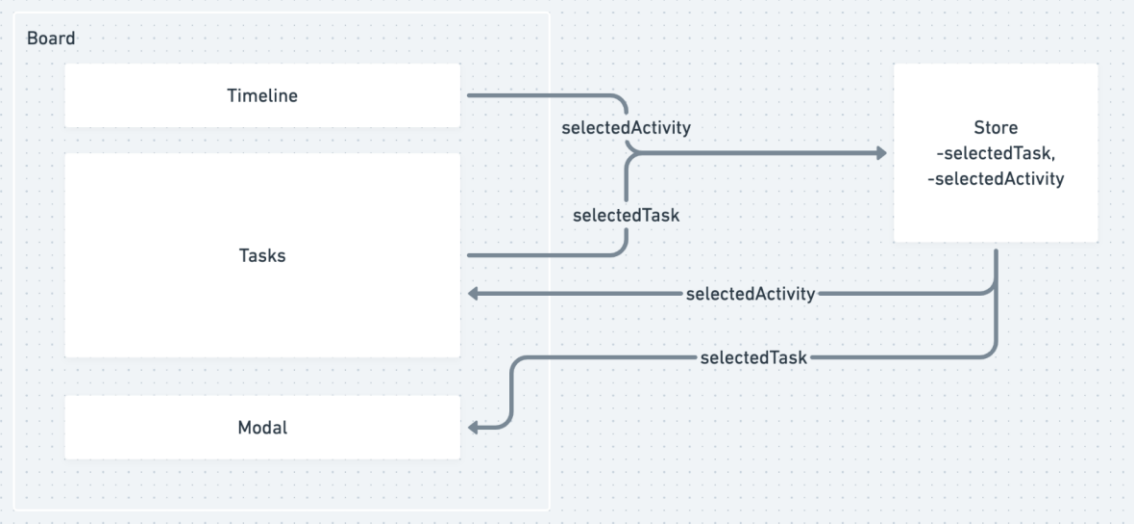


Figure 14. Data flow of managing task.

selectedActivity is transfered from Timeline to Store to keep a specific activity object. This changes make the acitivity props change in Tasks component, which was empty initially. The existance of selectedActivity provides enough information for Tasks component to render all the tasks of selected activity in Timeline.

Same as selectedActivity, when an event is triggered in tasks to send a selectedTask Object to Store, this change will make props in Modal component change and new UI of task details is rendered after that changing.

Components

- Tasks: Figure 15 shows how to render all the tasks with TaskItem component.

```
51    renderTasks = () => {
52        const { selectedActivity, taskPhases } = this.props
53        if (!selectedActivity) {
54            return null
55        }
56        if (!selectedActivity.tasks || selectedActivity.tasks.length === 0) {
57            return <span>There is no task yet</span>
58        }
59        return selectedActivity.tasks.map((item, key) => (
60            <TaskItem
61                key={key}
62                title={item.name}
63                status={normalizeTaskPhase(item.currentPhase, taskPhases)}
64                statusUpdatedTime=""
65                endDate={item.endDate}
66                task={item}
67                onTaskSelect={this.onTaskSelect}
68            />
69        ))
```

Figure 15.TaskItem is renderred in tasks wrapper.

In this functional component, tasks are rendered by TaskItem components. TaskItem represents for the wrapper of a task with props to tell the component what it should do with a task,what to render and what logic should be done.

- key: to create identity for each task to help Virtual DOM works faster.
- title: to show title of the task.
- status: to get status of the task so that TaskItem render correct
- endDate: to get endDate of the task to deal with the logic inside the component.
- onTaskSelect: this is an event and is triggered when user select or click on the task UI. This send selectedTask object to Store to let the Modal component know the existence of a task props of the Modal.

- Modal

Modal component is used to render the popup modal which is creating new UI on top of the website's UI. This is to save space and make the UI more user-friendly. As default, Modal component is not shown, as a explaination for showing props of this component in line 81 of Figure 16.

```
81        <Modal show={selectedTask} title="modal" onModalClose={onModalClose}>
82            {infoData && (
83                <InfoModal item={infoData} isFooterShown={true} page="task" />
84            )}
85        </Modal>
```

Figure 16. Task Information Modal.

Initially the selectedTask is null. There for show props is equal to false. When a selectedTask is sent to Store by onTaskSelect of TaskItem component in Figure 15, selectedTask is updated immediately and show props is equal to true, which to toggle the modal to be shown.

In the content of this Modal, information about the task are rendered through the props of InfoModal component. InforModal component receives item as a props to get the data of the component, isFooter shown is to decide if footer is rendered or not; page props is the define which page is this InfoModal using.

- InfoModal

The main purpose of using InfoModal component is to render discussion and status of the task. Figure 17 shows the structure of Modal component with needed children to render all the information required by Vertics: title, discussion, deadline and current status of the task

Figure 17. Component structure of InfoModal.

InfoModalDiscussion component shows all the messages that users discuss about stories of the task. It receives props as discussion to get array of all messages, followers to render people

who are in the discussion, onEdittor function to be triggered when the chat input is focused. This InfoModalDiscussion is rendered separately with another component in case there is some error from another place and messages are still rendered.

InfoModalStatus helps user to update status of the current task. The purpose of this component is to make the task proceed and users can keep track of what is going on, what they need to do next. The component simply render the status of the current task and have a selection to toggle the task to new status. It receives status props to render current status, onTaskStatusToggle to trigger the function to handle the event when user update status of task

Results

On Figure 18,activity details are shown below the timeline. Tasks list is rendered through task item, which have information about status, title, avatar of user who has main responsibility for the task. Description of the activity is also revealed on the bottom to instruct the participants about the task and note for it.

Information of a task is demonstrated in the modal when it is chosen as in Figure 19.

Header of the modal shows basic information about the task which is name, which activities the task belongs to and the control panel to help user toggling between files, discussions and close the modal. Content of the modal is the most important part where renders discussions or files of the task. Deadline of the task and its status are display at the bottom of the modal. Figure 20 shows clearly about the status of current task. The status is the selection so that user can change the phase of the task by choose options, which are Not Started, In Progress, Waiting Confirmation, Completed.

Goals that are reached by this section and make commissioner satisfied:

- Task detail popup fast and nicely render all the basic information about the task.
- Layout of discussions and files in attachment is fine. Chat shows information of users and times.
- Task status works well.
- Users can be added to the task easily.

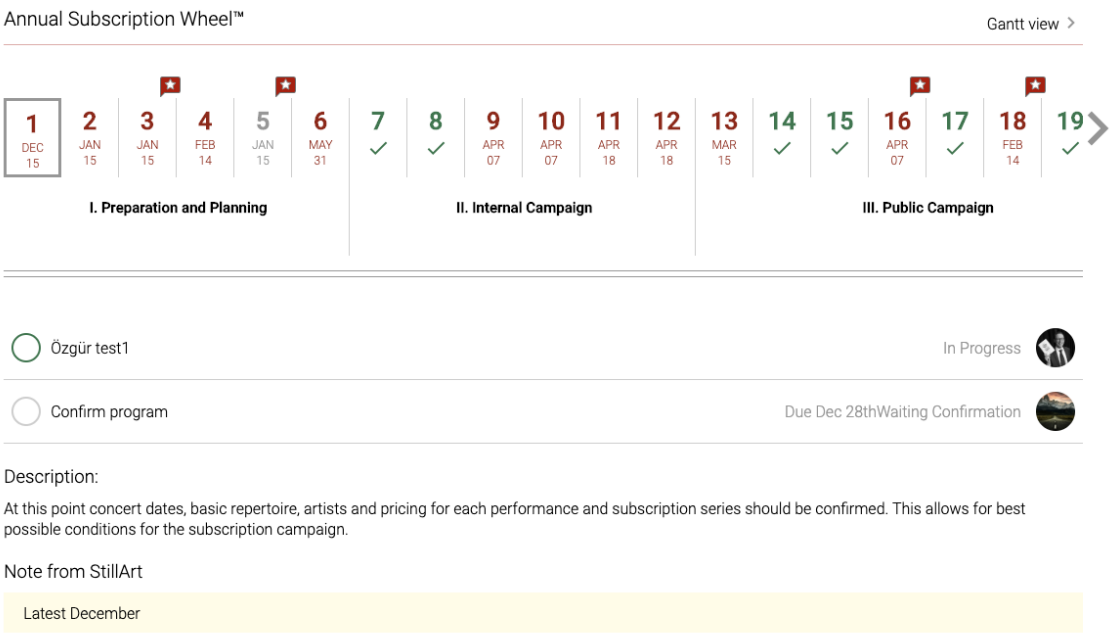


Figure 18. Completed UI of task management system when an activity is selected.

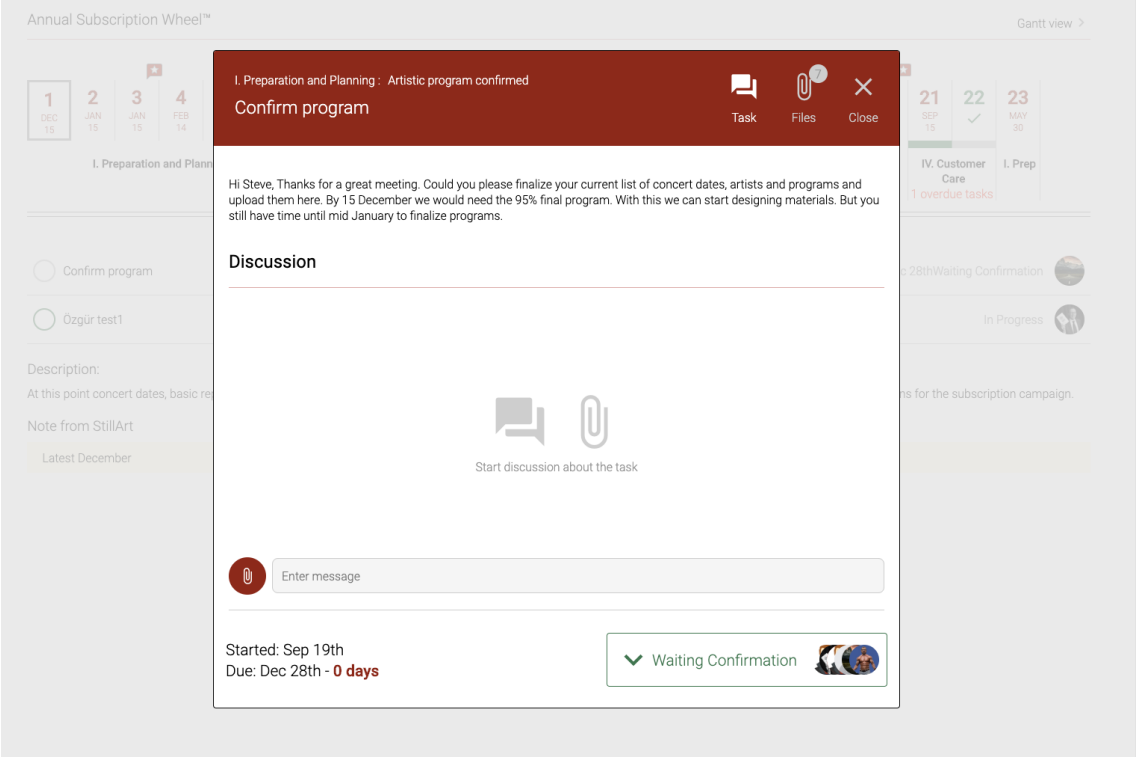


Figure 19. Completed UI of task detail.

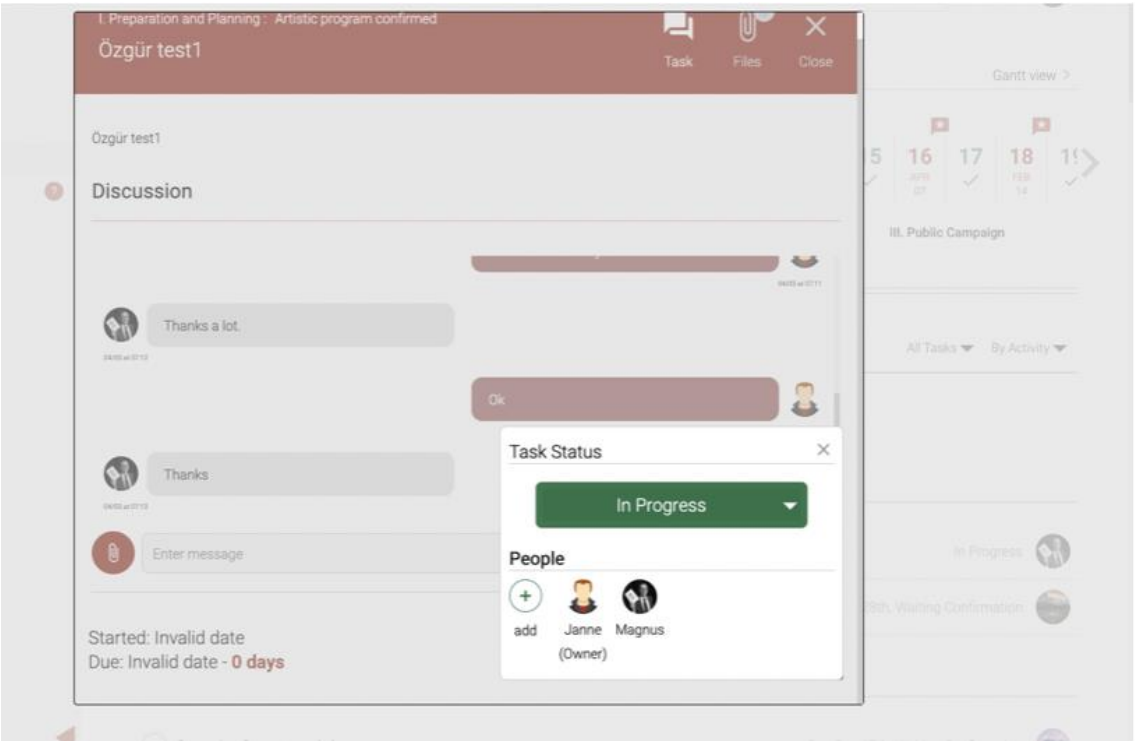


Figure 20. Completed UI of updating status of a task.

5 CONCLUSION

The objective of this thesis was to contribute a timeline-based website for client of Vertics company. ReactJS theory is researched to apply on the application to make a website with high speed and user-friendly.

The time-line based web application is only focus on the development of timeline, which aimed to show how to apply fundamentals of ReactJS on a real application. The thesis did not include advanced features of the products since it will be discussed in further phases with the client. To sum up, in the beginning, the work began quite well in UI wise. The challenges in this project occurred mostly when working with data from the back-end. It was necessary to rearrange all the data before using it locally in the project. Finally, the project was managed successfully and the goals of the thesis were achieved. The project manager in Vertics has checked and had some comments about the website. The feedback was all about the styling of the UI, which has been finished in the new version of this web. The results show that all the functions as the commissioner required and discussed, work perfectly and soon the current website will be replaced with the new version.

The website basically helps art companies to get more audiences, increase annual income by letting their employees to follow a standard marketing strategy, which is built in the web application. The product helps customer to be easier in managing tasks and made the organizations bigger in the future with the support of ReactJS which improve the performance of the application.

REFERENCES

1. Stateofjs. (2019). Front-end Frameworks – Overview. [online] Available at https://2018.stateofjs.com/front-end-frameworks/overview/ [Accessed on 1 Aug. 2019]
2. React. (2019). ReactDOMServer. [online] Available at https://reactjs.org/docs/react-dom-server.html [Accessed on 5 Aug. 2019]
3. React. (2019). ReactJS definition. [online] Available at https://www.w3schools.com/REACT/default.asp [Accessed on 5 Aug. 2019]
4. Medium. (2019). Lets better know about the famous DOM. [online] Available at https://medium.com/geekabyte/lets-better-know-the-famous-vdom-a21faf9e9157 [Accessed on 10 Aug. 2019]
5. Svelte. Virtual DOM is pure overhead. [online] Available at https://svelte.dev/blog/virtual-dom-is-pure-overhead. [Accessed on 14 Aug. 2019]
6. Hackernoon. (2019). Virtual DOM in ReactJS. [online] Available at https://hackernoon.com/virtual-dom-in-reactjs-43a3fdb1d130 [Accessed on 15 Aug. 2019]
7. React. (2019). Reconciliation. [online] Available at https://reactjs.org/docs/reconciliation.html [Accessed on 22 Aug. 2019]
8. React. (2019). Components and Props. [online] Available at https://reactjs.org/docs/components-and-props.html [Accessed on 22 Aug. 2019]
9. Newline. What is JSX? ES6?. [online] Available at https://www.newline.co/fullstack-react/30-days-of-react/day-2/ [Accessed on 25 Aug. 2019]
10. React. (2019). Introducing JSX. [online] Available at https://reactjs.org/docs/introducing-jsx.html [Accessed on 15 Sep. 2019]
11. Programing with Mosh. (2019). React lifecycle method. [online] Available at https://programmingwithmosh.com/javascript/react-lifecycle-methods/ [Accessed on 30 Sep. 2019]
12. CSS trick. (2019). Leveling up with React:Redux. [online] Available at https://css-tricks.com/learning-react-redux/ [Accessed on 15 Oct. 2019]
13. Redux. (2019). Three Principle. [online] Available at https://redux.js.org/introduction/three-principles [Accessed on 20 Oct. 2019]
14. Dev. (2019). An Intro to Redux that you can understand. [online] Available at https://dev.to/macmacky/an-intro-to-redux-that-you-can-understand-55k8 [Accessed on 17 Nov. 2019]
15. Medium. (2019). How to choose a text editor for Javascript. [online] Available at https://medium.com/free-code-camp/how-to-choose-a-text-editor-for-javascript-ec75c2cee6b8 [Accessed on 31 May. 2020]

16. Medium. (2019). Which node package manager should I use. [online] Available at https://medium.com/@timothy.kaing/which-node-package-manager-should-i-use-2019-ac3d324a83e5 [Accessed on 31 May. 2020]