SOFWERX

Integrating Finite State Models with React and Redux

Jim Ladd

June 29, 2020

Contents

Introduction

Software development is not easy but it should be straightforward. Anytime an activity within this process wanders from the straight path forward, an opportunity for improvement emerges. Such is the case with designing client applications with the React/Redux technologies. After evaluating past projects delivered by vendors, reviewing online tutorials and examples, and talking with different developers, there is potential to improve the design of React/Redux-based software.

This paper presents how Finite State Models (FSM) can be utilized to increase the knowledge and specification of the problem domain and drive the design of the software in the solution domain. A non-trivial problem involving multiple countdown timers is described, the design rules are explained, and a working demo is created. The complete source code is available via a cloud-based repository.

Background

React is a very popular library for building user interfaces [1]. React can be used as a base for development but it is focused on rendering data to the DOM. Creating React-based applications usually requires the use of additional libraries for state management and routing [2]. Redux is also a popular JavaScript library and is used for the management of application state. Typically, when an application is designed with React, it is also using Redux [3].

Just because a library is popular, doesn't necessarily mean that it is straightforward to use. Several developers have divulged challenges when learning React/Redux [4] [5]. I have seen a wide range of designs, patterns, styles, etc. with applications built using React/Redux. Even in the same application written by a single vendor, I observed a wide range of implementations of React/Redux that made timely and effective maintenance very difficult.

These bumps in the learning curve can be attributed to 1) a lack of consistency in the tutorials and examples, 2) unfamiliarity with a state-based paradigm, and 3) the ES6 notation in which minimal typing seems to rule over readability. While the third factor can be addressed with adequate code comments, the first two require more serious considerations. When a project involves state-oriented subject matter, finite state model (FSM) technology can assist with the understanding of the problem space and facilitate the creation of the solution space. This paper demonstrates how FSM can be applied to a state rich problem to increase the knowledge of that problem and then be integrated with React/Redux in a consistent fashion to guide the design of the solution.

Finite State Models

Finite state models (FSM) have been used for a number of years in a wide spectrum of industries. Domains using finite state model technology include communication systems, automobiles, avionics systems, and man-machine interfaces [6]. These problem domains share common characteristics: they are usually large in size, high in complexity, and reactive in nature. A primary challenge of these domains is the difficulty of describing reactive behavior in ways that are clear, concise, and complete while at the same time formal and rigorous.

Finite state models provide a way to describe large, complex systems. FSMs view these systems as a set of inbound and outbound events, conditions, and actions integrated together to understand, model, and manage the change of state within the application. Some FSM technologies also provides a set of rules for translating the problem artifacts or specifications into source code. The partitioning of the problem

into the events, conditions, and actions, the structured processing environment, and the ease of expressing the processing logic are the foremost strengths of FSMs.

Several different FSM patterns and archetypes have been researched and developed over the years. From the simplistic approach of a multi-value system variable to extremely sophisticated frameworks incorporating concurrency, hierarchies, persistency, and historical information. The pattern chosen for this project has been used successfully on past commercial systems. It is a straightforward but capable model. The fundamental components of our finite state model include:

- ***State*** represents the "mode of being" the object at any given time. An instance of the object being modeled may change state via Transitions. A Transition may be traversed when a specific event occurs and a set of associated conditions are met.

- ***Transition*** describes a single pathway from a given state to another state. The set of all transitions describe all possible paths among the defined states. A transition contains an event that it is subscribing to a condition and an action. A transition also contains the state from which the transition is exiting (i.e. the "from" state) and the state to which the transition is entering (i.e. the "to" state).

- ***Event*** is the mechanism that the system uses to interact with external systems and with itself.

- ***Condition*** represents a set of logic that evaluates to a Boolean result. It is used to determine if a transition is to be activated.

- ***Action*** is the processing to be performed with a transition is activated.

A diagram of the major components of the FSM is shown below:

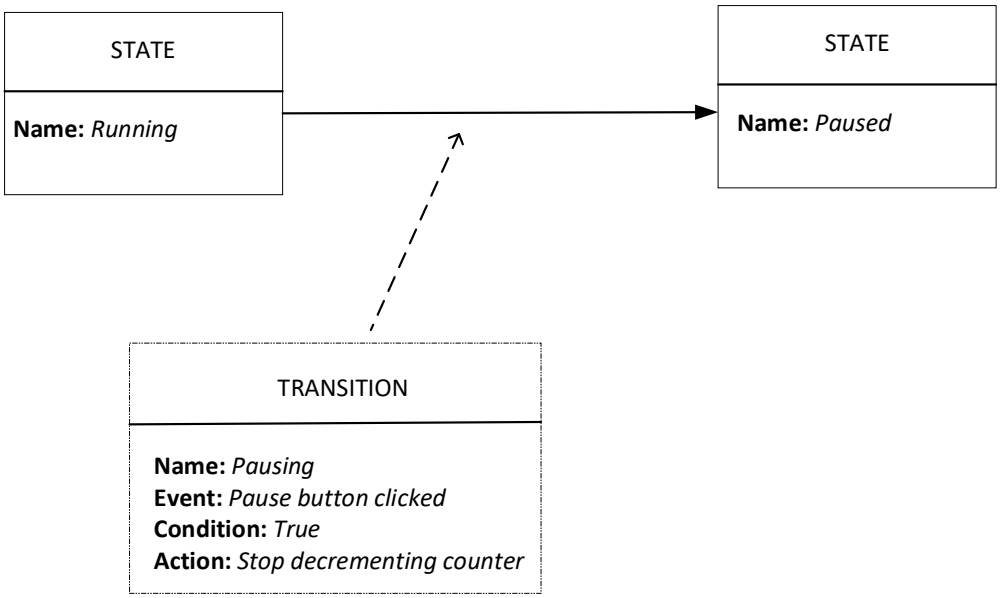

Figure 1 – Key elements of the finite state model

One of the more puzzling topics covered during the learning process of finite state models concerns events. What exactly is an event? How are they created? What's *their* lifecycle? Events can be considered as instances of trigger mechanisms that initiate code execution. In most problem domains, event generation can originate from four sources:

- ***User Input*** – Events can be created when a human interacts with a user interface. Mouse clicks, key strokes, etc., can all generate events that triggers processing.
- ***External Systems*** – Messages, responses, signals, etc. from external systems can also create events that may be relevant to the finite state models.
- ***Passage of Time*** - The simple act of time passing can generate events. The FSM may represent the passage of time in various ways. A common technique is shown in our code example.
- ***Change of Internal State*** – The occurrence of a FSM instance achieving or "entering" a new state may warrant a new event to be generated.

Problem Domain

A major challenge in constructing a tutorial is selecting an appropriate problem to be solved. A problem that is too trivial allows the readers to learn some concepts quickly but they may not be helpful when the applied to a real world scenario. A problem that is too complex can overwhelm the readers and raise more questions than answers. The problem domain chosen for this effort was 1) based on a real world application, 2) has non-trivial state information, and 3) contains widely known subject matter.

Our goal in this project is to build a countdown timer application. First, let's review the high level requirements:

- The user can create one or more countdown timers in a web application.
- The user may enter a time (in seconds) and then start the timer.
- The user may stop a clock which terminates the count down immediately. The timer shows an indicator of being stopped for 3 seconds.
- The user may pause a timer indefinitely. A visual indicator will show that the timer is paused. The time that the user paused the timer will be displayed.
- The user may resume a paused timer. The timer will continue the countdown from the time that the paused occurred.
- When a timer completes a countdown, it will show an indicator of being completed for 3 seconds.

Instead of immediately jumping to code, first create a finite state model to facilitate the understanding of the problem domain. Based on the problem statement, the following finite state model diagram is created:

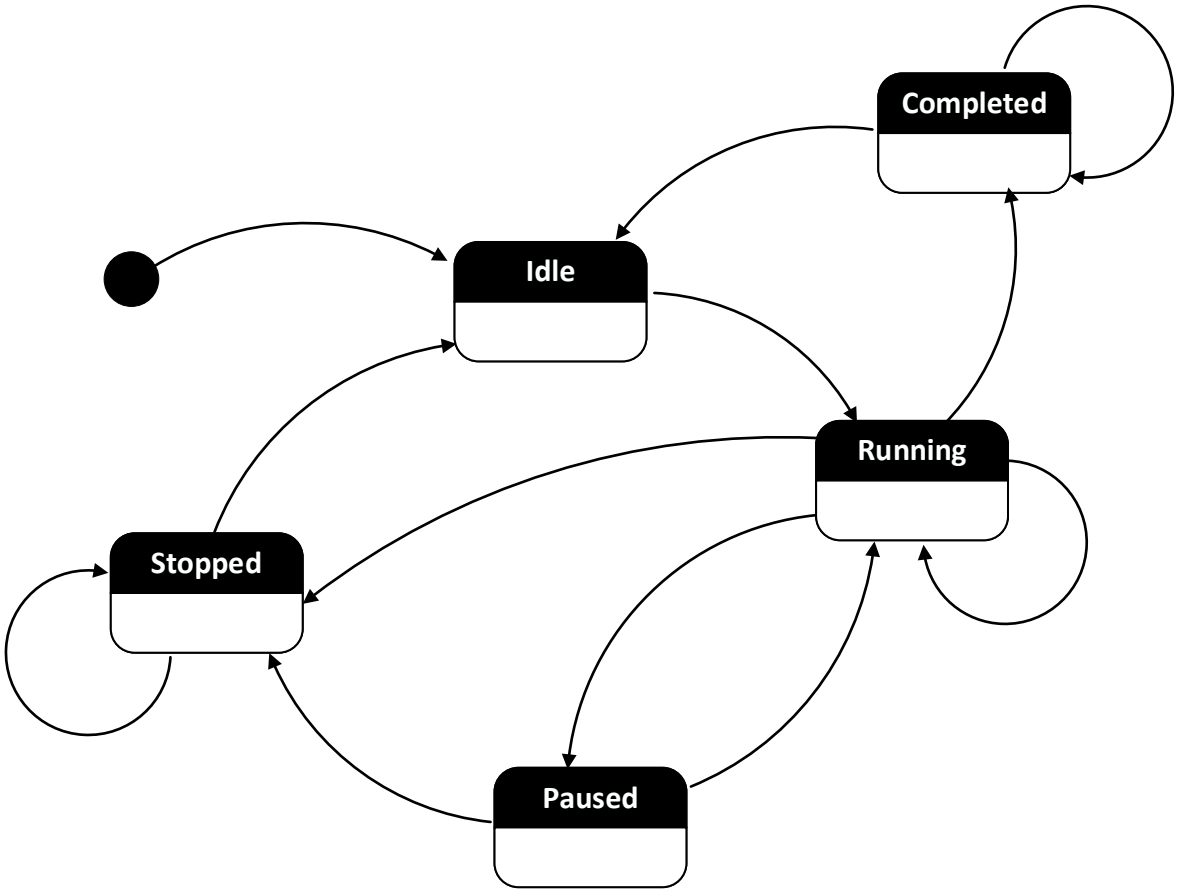

Figure 2 – The graphical finite state model for the countdown timer.

One of the features with FSM is that the same information can be shown in a graphical form as well as a tabular format. Both formats can be helpful in the analysis phase of the process. The tabular format of the "happy path" processing (i.e. *Idle -> Running -> Completed -> Idle*) is shown below.

FROM STATE	TO STATE	EVENT	CONDITION	ACTIONS
Start	Idle	CreateTimer	True	1. Create a new timer instance with the name of the entered value.
Idle	Running	StartTimer	True	1. Set the entered value to the timer's value. 2. Start the timer.
Running	Running	TimerTick	If time has not expired	1. Decrement the timer's value.
Running	Completed	TimerTick	If time has expired	1. Stop the timer. 2. Set a new time for 3 seconds. 3. Start the timer (*i.e. to show the "Completed" state for 3 seconds*)

Completed	Completed	TimerTick	If time has not expired	1. Decrement the timer's value.
Completed	Idle	TimerTick	If time has expired	1. Stop the timer.

Table 1 - The tabular format of the path though the FSM

Solution Domain

So far, our time has been spent entirely in the problem domain. Even though we should have a deep understanding of the problem, the work to this point is independent of the solution space. We could easily design and implement our countdown timers in a variety of technologies and frameworks. However, this paper promised to show how FSM can assist with React/Redux development so let's begin those design activities.

To translate the FSM into a code, the following guidelines are followed:

1. ***Core objects are captured as "Models"***
   The concept of a "Model" originates from the Model-View-Controller (MVC) pattern. In this paradigm, a Model maintains data and the logic for manipulating that data. A Model is independent of views or display artifacts. This concept of Models is not within the traditional structure of React/Redux designs but it is still very valuable since it aids in the partitioning of responsibility.

2. ***React Components are responsible for the display of the Models***
   Components are responsible for the display of the Model. Depending on the overall user interface design, zero or more Components can be constructed for each Model.

3. ***Redux Actions are simple mechanisms to represent FSM events***
   There is an overloading of the "actions" term between the Redux and FSM realms. To minimize the confusion, we are keeping the Redux Actions structure and how it operates in the Redux pattern. We are using Redux Actions to create the FSM events. The FSM "actions" will be delegated to the Redux Reducers structure.

4. ***Redux Reducers are responsible for event processing and state transitions.***
   Reducers are responsible for the bulk of the FSM processing. However, by partitioning the design into Models and Components, the overall burden of this processing is reduced and very focused. Reducers perform the event processing, the evaluation of the conditions related to the incoming events, the execution of actions associated with the transition, and finally, the migration of the model instance to its new state.

Models

To apply the guidelines to our problem domain, we first create a Timer model. The states of its lifecycle are defined in this class and can be referenced by the other code in the project. A simple constructor is created and sets the initial state to IDLE and a time interval of 1 second. A "time" attribute is responsible for maintaining the remaining time in the count down activity. The "timerId" attribute is a unique identifier for the instances of the Timer class. This code is shown below:

```
import {store} from '../App'

export const IDLE          = "IDLE"
export const RUNNING       = "RUNNING"
export const STOPPED       = "STOPPED"
export const PAUSED        = "PAUSED"
export const COMPLETED     = "COMPLETED"

class Timer {

    constructor(name, timerId) {
        this.name = name;
        this.timerId = timerId;
        this.time = 0;
        this.lifecycleState = IDLE;
        this.intervalLength = 1000;
    }

    setTime( value ) {
        this.timer = value;
    }

    isExpired() {
        if ( this.time <= 0 ) {
            return true;
        }
        else {
            return false;
        }
    }
```

A design decision was made to have the Timer class encapsulate the actual time keeping required for the countdown process. The built in *setInterval()* function is used to bind a class function so when the timer interval is reached, the class function, *tick(),* will be invoked. The *tick()* function will then dispatch a Redux Action with the value of the *timerId* so this instance can be referenced by other code. The remainder of the Timer class is shown below:

```
    startTimer() {
        this.interval = setInterval( this.tick.bind(this),
                                     this.intervalLength);
    }

    tick() {
        store.dispatch({type: 'TIMER_TICK',
                        payload: { timerId: this.timerId } } );
    }

    decrement() {
        this.time = this.time - ( this.intervalLength / 1000 );
    }

    stopTimer() {
        clearInterval(this.interval);
    }
```

```
} // end class

export default Timer
```

Components

There are three Components to support the display of the Timer model. The NewTimer component is used to accept the name of the timer from the user and then invokes the *createTimerEvent* function. The TimersList component is responsible for iterating over the list of timer instances and renders the TimeView component for each one. Lastly, the TimerView component renders the display elements of a single timer instance.

The TimerView component is a traditional React/Redux component but it is streamlined since responsibility for the time keeping processing has been delegated away. The imports and constructor of the TimerView is shown below:

```
import React, { Component } from 'react'
import { connect } from 'react-redux'

import { startTimerEvent,
         stopTimerEvent,
         pauseTimerEvent,
         resumeTimerEvent } from '../actions/TimerActions'

// Import the lifecycle states
import { IDLE, PAUSED, RUNNING, STOPPED, COMPLETED } from
'../models/Timer';

class TimerView extends Component {

    constructor(props) {
        super(props)

        this.state = {
            inputValue: ''
        };
    }

// invoke the startTimerEvent and then clear the input value
onClickStartButton = () => {
    this.props.startTimerEvent(this.props.index,
this.state.inputValue);
    this.state.inputValue = "";
};
```

The *onClickStartButton()* function is needed so that the *inputValue* can be cleared of the user's characters once the Start button is pressed. The *render()* function is typical Redux code that extracts the instance of the Timer class and displays its values along with a Start, Pause, Resume, and Stop buttons. The rest of the TimerView code is located in the */src/components/TimerView.js* file.

Actions

The Redux Actions are used to define the event types. The reducers for the Timer model consist of a set of functions that return a type attribute and a payload. The first part of the */src/actions/TimerActions.js* file is shown below:

```
export const NEW_TIMER = "NEW_TIMER"
export const START_TIMER = "START_TIMER"
export const STOP_TIMER = "STOP_TIMER"
export const TIMER_TICK = "TIMER_TICK"
export const PAUSE_TIMER = "PAUSE_TIMER"
export const RESUME_TIMER = "RESUME_TIMER"

export const createTimerEvent = (name) => {
    return {
        type: NEW_TIMER,
        payload: { name }
    }
}

export const startTimerEvent = (index, inputTime) => {
    return {
        type: START_TIMER,
        payload: { index, inputTime }
    }
}
```

Reducers

The Redux Reducers perform the bulk of the event processing. The structure of the TimerReducer code is traditional Redux but its internal code is focused on evaluating the inbound event, locating the appropriate Timer instance, checking any conditions, executing the FSM actions, and then transitioning to the next state where applicable. The processing for the NEW_TIMER and START_TIMER event types are shown below:

```
import { NEW_TIMER, START_TIMER, TIMER_TICK, STOP_TIMER, PAUSE_TIMER,
RESUME_TIMER } from '../actions/TimerActions';
import Timer from '../models/Timer';
import { IDLE, PAUSED, RUNNING, STOPPED, COMPLETED } from
'../models/Timer';

const defaultState = {
    timers: [],
    isNewTransition: false
};

export default (state = defaultState, action) => {
    let newState = state;

    switch (action.type) {
        case NEW_TIMER:
            // Add a new timer, return a copy of state
            const name = action.payload.name ? action.payload.name :
`Timer ${state.length}`
            const timerId = state.timers.length;
```

```
            newState = { ...state,
                         timers: [...state.timers, new Timer(name,
timerId)] };

            return newState;

        case START_TIMER:
            // Find the timer and set the time to the input
            state.timers.map((timer, index) => {
                if (action.payload.index === index) {
                    if ( timer.lifecycleState === IDLE ) {
                        timer.time = action.payload.inputTime;
                        timer.startTimer();
                        timer.lifecycleState = RUNNING;

                        // a transition happened so update the list of
timers view
                        newState = { ...state, isNewTransition: true };
                    }
                }
                return null;  // must return something or a warning
will be generated
            })
            return newState;
```

Note that when code determines that a transition should occur, the *isNewTransition* attribute is set to true in the *state* structure. This will cause the TimerView's *render()* function to be invoked so the user interface has the chance to be updated with the most current data.

Demonstration

The NewTimer component is responsible for receiving the user's desired name of the timer to be created and sending the event to create it. The web page that is generated is shown below:

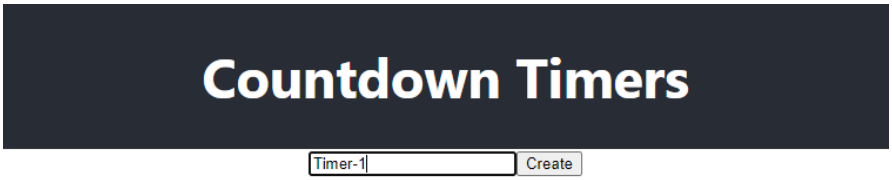

Figure 3 – The rendering by the NewTimer component.

The following is a rendering of the TimersList and TimerView components. The two timers are in the IDLE state.

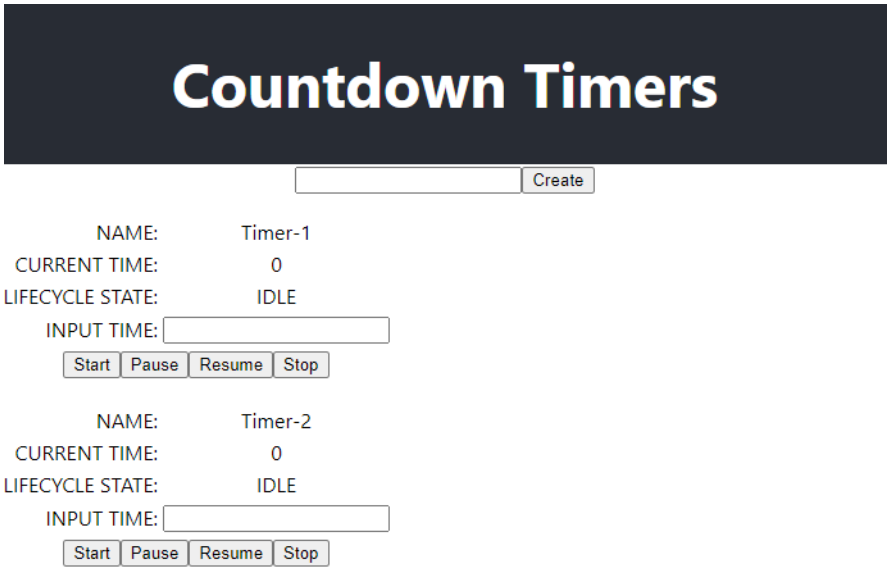

Figure 4 – Two timers that are in the IDLE state.

This last screen shot shows the two timers in different states.

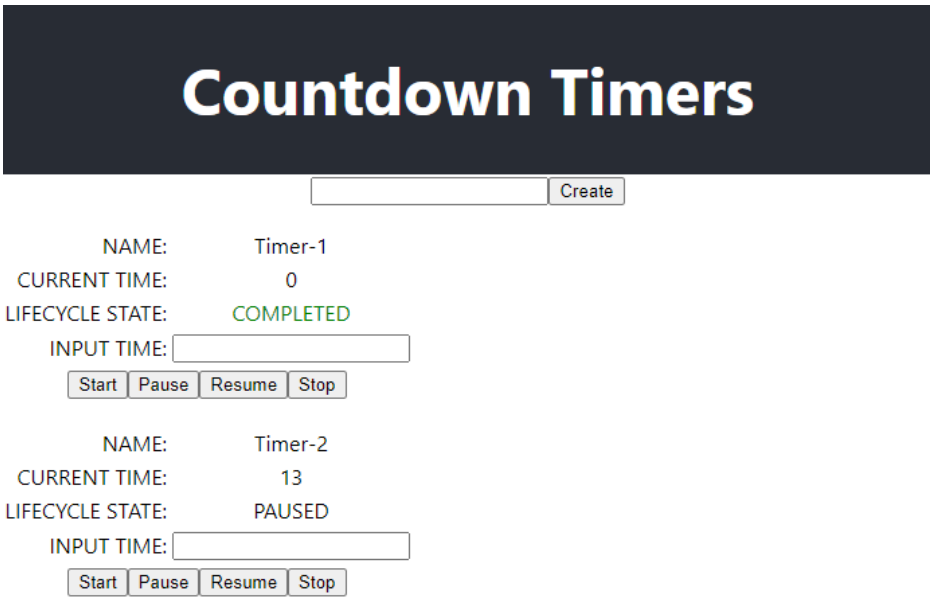

Figure 5 – Two timers in different states

Where to Find and How to Build

The link to the Git repository is:

https://github.com/sofwerx/react-redux-fsm

Once the repository is cloned, use the following command to install the required libraries:

```
npm install
```

To execute the project, use the following command:

```
npm start
```

A few warning messages will be issued during the startup phase. None of these warnings are serious.

Looking Back and Moving Forward

Finite state ***models*** were leveraged in this project to simplify the design of React/Redux applications. This effort stopped short of implementing a finite state ***engine***. The engine approach represents the lifecycle in an explicit model, typically casted in a JSON or XML-based specification, and has code that executes that model. Such engines are commonly embedded in service-side code for workflow, control-heavy, and process automation domains. To implement such an engine in a browser application would add too much complexity, especially considering the objective is to simplify the software development.

A natural next step in this project is to create a consistent approach to interacting with external web services via HTTP requests. This would address scenarios such as issuing GET and POST requests and then driving the state models to generate the appropriate display to the user.

Summary

The objective of this project was to simplify the software development process when using the React and Redux technologies. The approach applies Finite State Model technology to the problem space and maps the artifacts into the React/Redux code space in a consistent and straightforward manner. The process started with requirements involving count down timers and was completed with a working demo.

References

[1] S. Daityari, "Angular vs React vs Vue: Which Framework to Choose in 2020," [Online]. Available: https://www.codeinwp.com/blog/angular-vs-vue-vs-react/. [Accessed 29 June 2020].

[2] Wikipedia, "React (web framework)," [Online]. Available: https://en.wikipedia.org/wiki/React_(web_framework). [Accessed 30 June 2002].

[3] H. Stevanoski, "The only introduction to Redux (and React-Redux) you'll ever need," [Online]. Available: https://medium.com/javascript-in-plain-english/the-only-introduction-to-redux-and-react-redux-youll-ever-need-8ce5da9e53c6. [Accessed 29 June 2020].

[4] J. Snook, "Why Did I Have Difficulty Learning React?," [Online]. Available: https://snook.ca/archives/javascript/difficulty-with-react. [Accessed 29 June 2020].

[5] B. Frost, "my struggle to learn react," [Online]. Available: https://bradfrost.com/blog/post/my-struggle-to-learn-react/. [Accessed 30 June 2020].