Dominic Travis Kudiabor

STATE MANAGEMENT WITH REACT-REDUX

Thesis
CENTRIA UNIVERSITY OF APPLIED SCIENCES
Information Technology
December 2020

ABSTRACT

Centria University of Applied Sciences	**Date** December 2020	**Author** Dominic Travis Kudiabor
Degree program Information Technology		
Name of the thesis STATE MANAGEMENT WITH REACT-REDUX.		
Centria supervisor Jari Isohanni		**Pages** 25

State management is the most essential characteristic of developing scalable web or native applications. The objective of the thesis was to establish the significance and practicality of utilizing React-Redux as a state management library in React. The architecture of React-Redux can be divided into three main sections: store, actions, and reducers. The store encompasses the global state of the application. The actions are descriptive plain objects. The reducers contain the main business logic of updating the store. This thesis elaborates on the functionality of these three sections and how they are interconnected with each other.

The thesis further elucidates the flow of data within the application and the efficacy of React- Redux in managing the global state. The login feature of an application with React-Redux is utilized for the analytical explanation of the core concepts and the data flow within the application. The thesis aims to demystify the complexity of implementation and also to improve personal comprehension of the Redux library. The future of state management is discussed with suggestions for improvement. This thesis would enable React developers to optimize their applications by using global state management. Applications with several asynchronous actions would benefit significantly from state management.

Keywords

Component, ES6, Props, React, React-Redux, Redux, Scalability, State

CONCEPT DEFINITIONS

Component

A compact reusable piece of code that is essentially a JavaScript function or ES6 class that returns a React element to be rendered on the user interface.

ES6

ECMAScript 6 also referred to as ECMAScript 2015 is a JavaScript language specification standard that defines the functionality of its usage.

Props

Props is the term used to describe inputs being passed from a parent component to a child component. They are read-only.

React

A JavaScript library for developing interactive user interfaces.

React-Redux

The official React library for managing the global state of a React application.

Redux

A predictable state management library for managing the global state of a JavaScript application.

Scalability

The efficacy of an application to accommodate growth without disrupting the end-user.

State

A user-defined plain JavaScript object which holds information specific to a component in an application.

ABSTRACT
CONCEPT DEFINITIONS
CONTENTS

1 INTRODUCTION **1**

2 STRUCTURE AND DESIGN **2**
2.1 The MVC software design pattern **2**
2.2 The Flux software design pattern **3**
2.3 The Redux software design pattern **4**

3 DATA FLOW **6**
3.1 Store **7**
3.2 Actions **8**
3.3 Reducer **9**
3.3.1 Unit Reducer **10**
3.3.2 Root Reducer **10**

4 ASYNCHRONOUS ACTIONS **11**
4.1 Redux-Thunk **12**
4.2 Redux-Saga **13**

5 EVALUATION AND DISCUSSION **17**

6 CONCLUSION **24**

REFERENCES **27**

FIGURES
FIGURE 1. The MVC design pattern 3
FIGURE 2. The Flux design pattern 3
FIGURE 3. The Redux design pattern 4
FIGURE 4. The flow of data in an application using Redux 6
FIGURE 5. The Redux Store 7
FIGURE 6. The Store Provider 8
FIGURE 7. Redux Action process 9
FIGURE 8. JavaScript Runtime Architecture 11
FIGURE 9. A Redux Flow with Middleware 12

PICTURES
PICTURE 1. Redux Developer Tools 5
PICTURE 2. A Unit Reducer 10
PICTURE 3. A Redux-Thunk example 13
PICTURE 4. A Redux-Saga Generator function 14
PICTURE 5. A Google Sign-In Saga 15

Picture 6. The Sign-In Functional Component 17
Picture 7. The User Reducer Function Returning New State 18
Picture 8. The Sign-In Actions 18
Picture 9. The Fetch-User Service Function 19
Picture 10. The Email Sign-In Saga 20
Picture 11. The User Reducer 21
Picture 12. The ‘UseEffect’ Hook 22
Picture 13. The App State 23

1 INTRODUCTION

React applications consists of several isolated components, which by design can control their state. The concept of a global state combines the individual states of the components into a single large state. Data from this state is accessible to all components within the application. There are many libraries created specifically to manage the global state. Examples of such libraries are Redux, MobX, and RxJs The React team introduced React-Redux as the state management library exclusively for React applications. The structure and design chapter of the thesis explains the origins of the Redux concept and inspiration from its predecessors.

The global state is governed by three major principles. First, it is a single source of truth. Debugging the application is relatively easier and features such as persistence and hydration are easily implemented. Secondly, the global state is read-only. The immutability of the global state prevents unexpected errors. The only way to effect a change in the store is to dispatch an action. Actions, in Redux, are objects that describe the change that needs to occur. Lastly, changes occur only with pure functions. A reducer is required for a state update to occur. Reducers are pure functions that take a previous state object and return an entirely new state object. It is pure because it does not mutate the existing global state. These processes would be explained further in the data flow chapter of the thesis.

React- Redux is not without flaws. There are a couple of improvements which could enhance its ability. Overall, the library is extremely versatile, but it has a few limitations. Developers who are new to the concept of state management might have to consider these tradeoffs before opting to use the library. The conclusion of the thesis discusses these tradeoffs and the future iterations of the library. The figures and pictures used throughout the thesis originate from my personal educational projects therefore I own the rights to use them without providing references.

2 STRUCTURE AND DESIGN

The state of an application contains locally created data and responses from a remote server. As the application grows in complexity, the state must be managed efficiently because it changes intermittently. Having multiple states in an application can increase the difficulty associated with keeping track of the data contained in them. (Voorhees 2020, 141.) Various software design patterns were introduced to provide a solution to this problem. Before the launch of JavaScript libraries, most applications were developed using the MVC (Model, View, and Controller) software design pattern. As the popularity of JavaScript libraries increased, React developers invented the Flux design pattern to provide a modern solution to managing states. (Garreau & Faurot 2018, 170.)

2.1 The MVC software design pattern

In the MVC model, the application is split into three separate parts, which allows the programmer to organize code according to functionality. The model contains the data that the user sees on the front end, the view is the user interface, and the controller is the business logic that interprets data to be displayed in the view. (Voorhees 2020, 218.) There can be multiple MVCs in an application. In a large application, sharing data between controller requires developers to introduce complicated logic which can be tedious to comprehend. This design pattern can be very difficult to debug due to the multiple models present in the application. Voorhees (2020, 223) considers this pattern as unpredictable because changes made in a model by its controller does not reflect in the other models unless an event listener is set to listen for changes in that particular controller. FIGURE 1 describes the data flow logic of an MVC application. The model never interacts directly with the view, the user requests are routed to the controller which performs data manipulations on the model and returns a response to the view. The model contains the data pertaining to that specific view. Controllers can modify the same data that is displayed in separate views; however, this can cause errors in the application if the logic pertaining to the data manipulation is not synchronized.

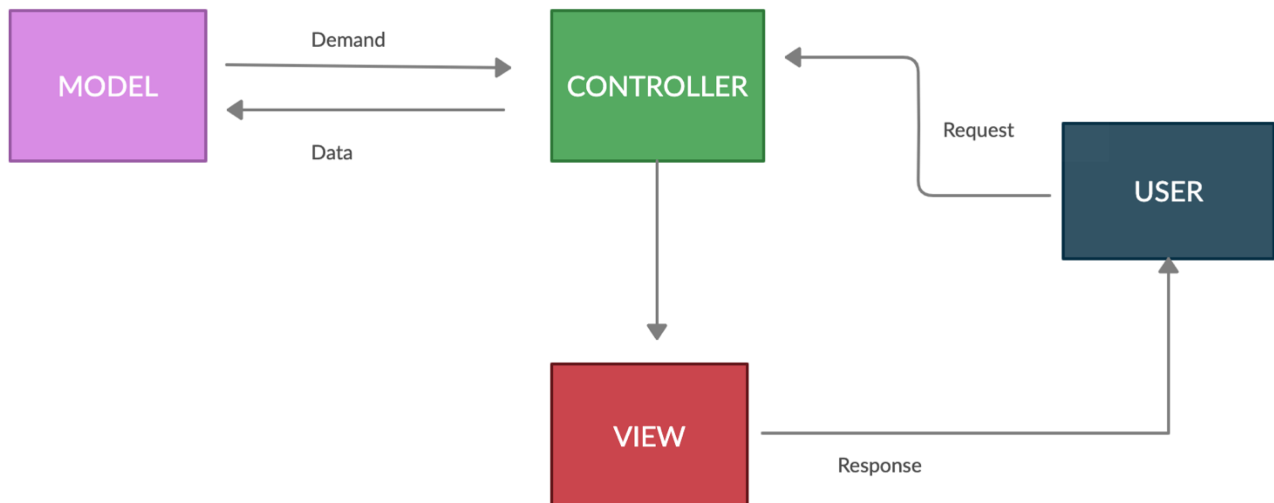


FIGURE 1. The MVC design pattern

2.2 The Flux software design pattern

The Flux software design pattern offers a much simpler and well-architected solution to the errors caused by controllers modifying the same data in different views. The Flux design pattern follows a unidirectional data flow approach as seen in FIGURE 2. The entire data flow procedure is repeated each time an action is executed by the user. (Garreau & Faurot 2018, 175.) The term "action" refers to the activity performed by a user of an application such as clicking a button, uploading a photo, and sending an email.

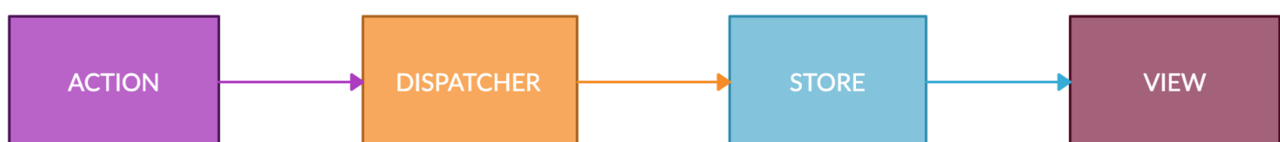


FIGURE 2. The Flux design pattern

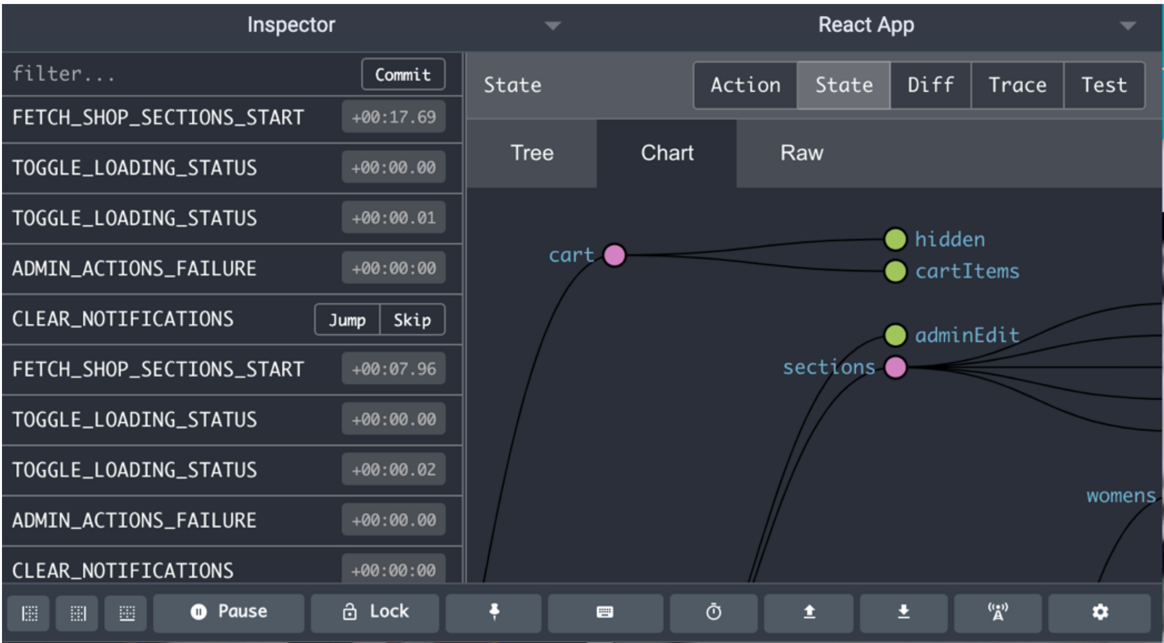
In the Flux pattern, an action is dispatched to the store by a dispatcher and the store performs the action and returns the response to the view. If the view gets updated, it creates an action that is dispatched to the store and then displayed on the view. (Garreau & Faurot 2018, 180.) This pattern was a revolutionary improvement to the MVC design. In React applications, a component by default can manage its local state. In a large application , these components might share data which can pose a problem. One solution would be moving the state to a higher parent component and passing data down to all the children components via props, however, this would result in prop-drilling. Prop-drilling refers to the passing down

of props from a parent component to a deeply nested child component. To avoid this situation entirely would require the data being moved to the root of the application and only fetched when needed (Garreau & Faurot 2018, 191). The Flux model centralizes data by creating a store from which the view update logic is regulated. This model prevented the code in the different parts of the application to directly mutate the store but rather describes the mutation as a plain object termed an action. The Flux model uses the concept of a dispatcher to execute these actions. (Garreau & Faurot 2018, 195.)

2.3 The Redux software design pattern

The architecture of Redux is inspired by the Flux design pattern. Its primary objective was to manage applications with a large global state and to facilitate the sharing of data between components. The global state of the application is located in the store (Lee, Wei & Mukhiya 2019, 21). When a user clicks on an action, it goes through a reducer which takes the previous global state and the action and returns the next global state. When the store is updated due to the reduce function, it triggers a re-render of the DOM (Document Object Model) which is simply a change in the user interface. The actions are descriptive objects that can be logged, serialized, and saved for debugging and testing purposes. Redux developer tools can replay action sequences to track the flow of data in the application as seen in PICTURE 1. The global state is altered by actions, this is to ensure that network callback functions can never directly mutate the global state (Lee, Wei & Mukhiya 2019, 30). The Redux architecture ensures all changes in the store occur one after the other in a strict sequential order as illustrated in FIGURE 3.

FIGURE 3. The Redux design pattern.



PICTURE 1. Redux Developer Tools

3 DATA FLOW

In a React application with Redux state management, the data flow is unidirectional and the components which require the same data retrieve props from the global state (Banks & Porcello 2017, 184). When a user logs in, the user credentials might be displayed in the header component. The user dashboard might also require access to the same credentials to display purchasing stats such as previous orders. When an action is dispatched from a component, the reducer listens for the type of action and informs the corresponding unit reducer that matches the action type. Multiple reducers can exist in an application, each handling a specific section of the application. These multiple reducers combine to form a large state in the root reducer. The individual states are referred to as slices of the global state. The Reducers are split up to ensure the immutability of the entire state upon the execution of an action. The unit reducers listen to every action. Actions are only executed by the reducers which match the exact type of action. The root reducer updates the store which in turn supplies data to components in the form of props. (Banks & Porcello 2017, 234.)

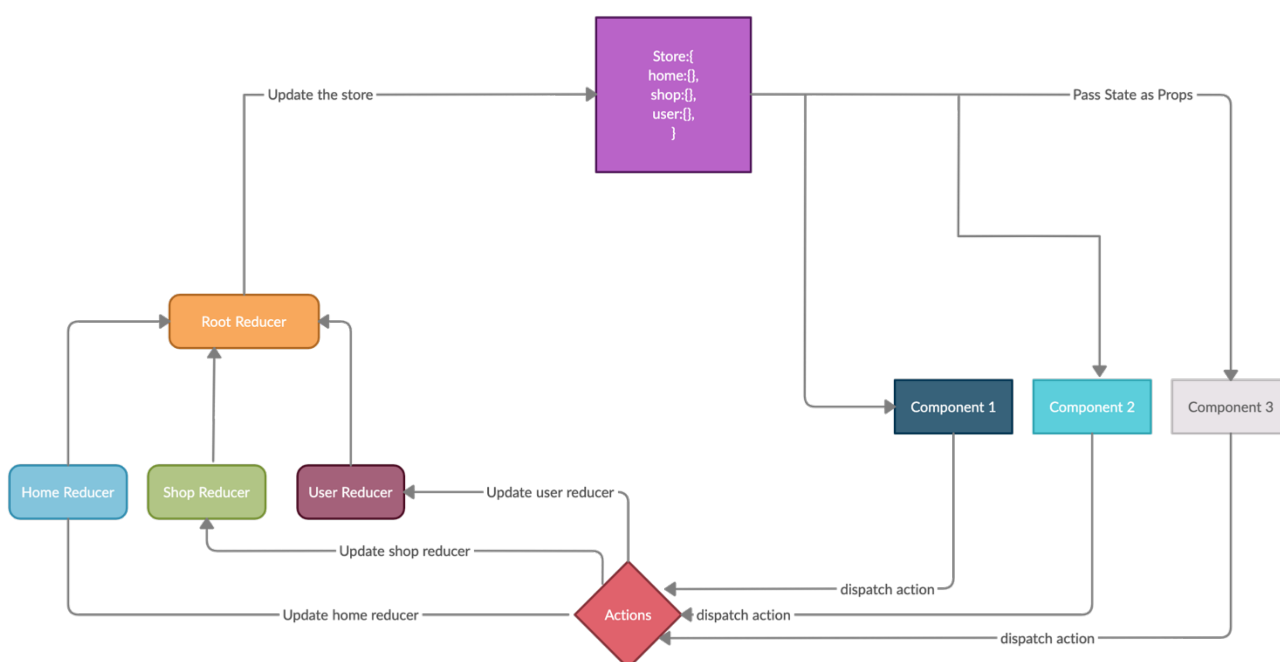


FIGURE 4. The flow of data in an application using Redux

In FIGURE 4, the application has three state slices present in the store: ‘userState’ , ‘shopState’, and homeState. Each state stores data pertaining to its description. Without Redux these states would exist in separate components. Sharing data between these components would be much easier if the states were

centralized (Banks & Porcello 2017, 180). With Redux, these states are moved up into the store and only updated if necessary. The home page can display data from the user state, such as the profile photo and login status. The shop page can display the contents of a shopping cart based on the user's login information. The shop reducer can only update the shop state and the same applies to every slice of state in the application. It is evident from FIGURE 4, how Redux makes sharing of data between components easier. The functionality of Redux is governed by three main terminologies: store, reducer, and actions. (Banks & Porcello 2017, 184.)

3.1 Store

The terms state and store are mutually exclusive. The global state lives in a store. Banks & Porcello (2017, 192) describe a store as a large object that encapsulates the global state tree as illustrated in FIGURE 5. There can be only one store in a React application. The store contains the individual states existing in the application. The components of the application must be nested inside a provider. The provider is a React element which enables components to access the store. Components which are not nested within a provider cannot access data from a store as illustrated in FIGURE 6. The provider is usually placed at the top level of the React application, with the entire app component tree nested within.

FIGURE 5. The Redux Store

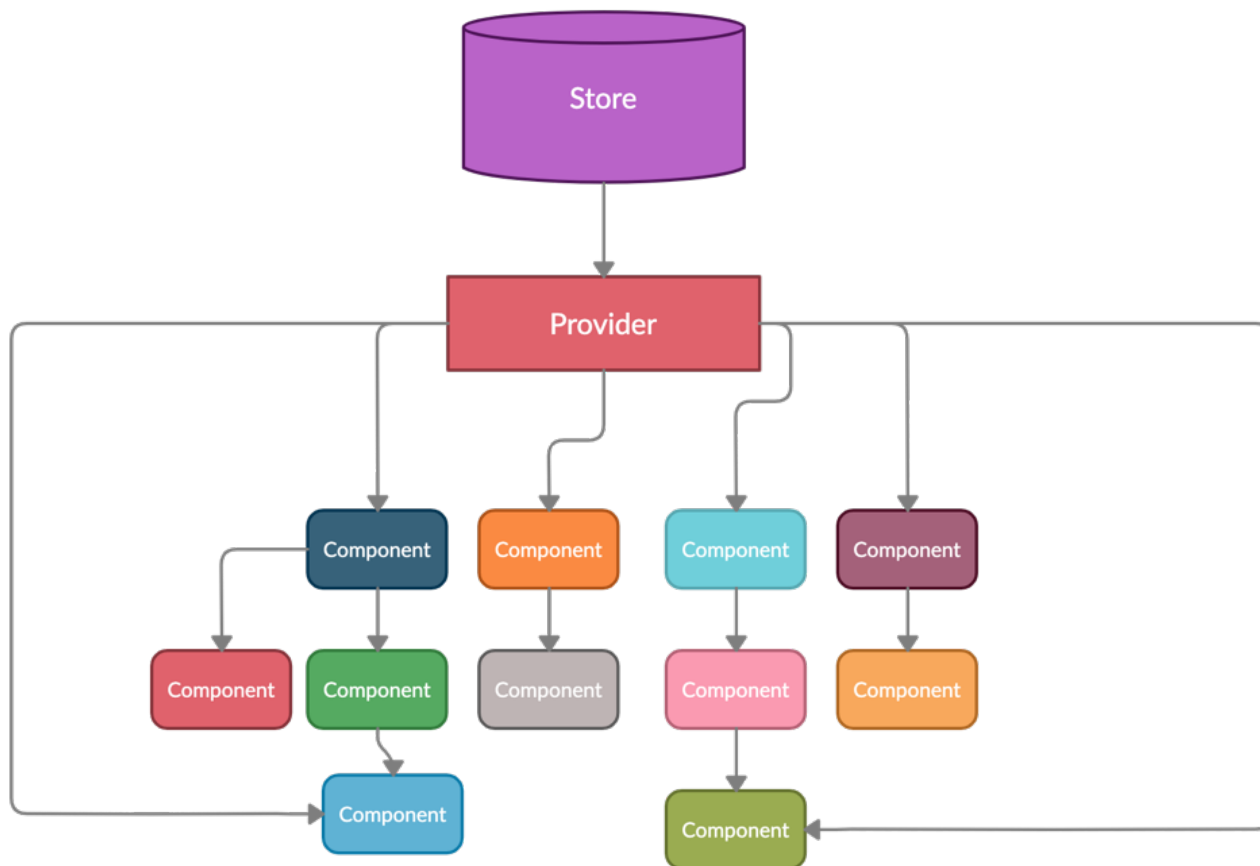


FIGURE 6. The Store Provider

3.2 Actions

An action is a plain JavaScript object which describes the objective of the change of state. Only actions can update a Redux store. They consist of two properties. The first is the "type", defined as a constant and usually imported from another module. The type is descriptive of the action being dispatched. They are defined as strings because they must be serializable. The second property is the optional payload. The payload contains the data that would be updated in the store as illustrated in FIGURE 7. A payload is always an object. The content can be of any type. (Banks & Porcello 2017, 205.)

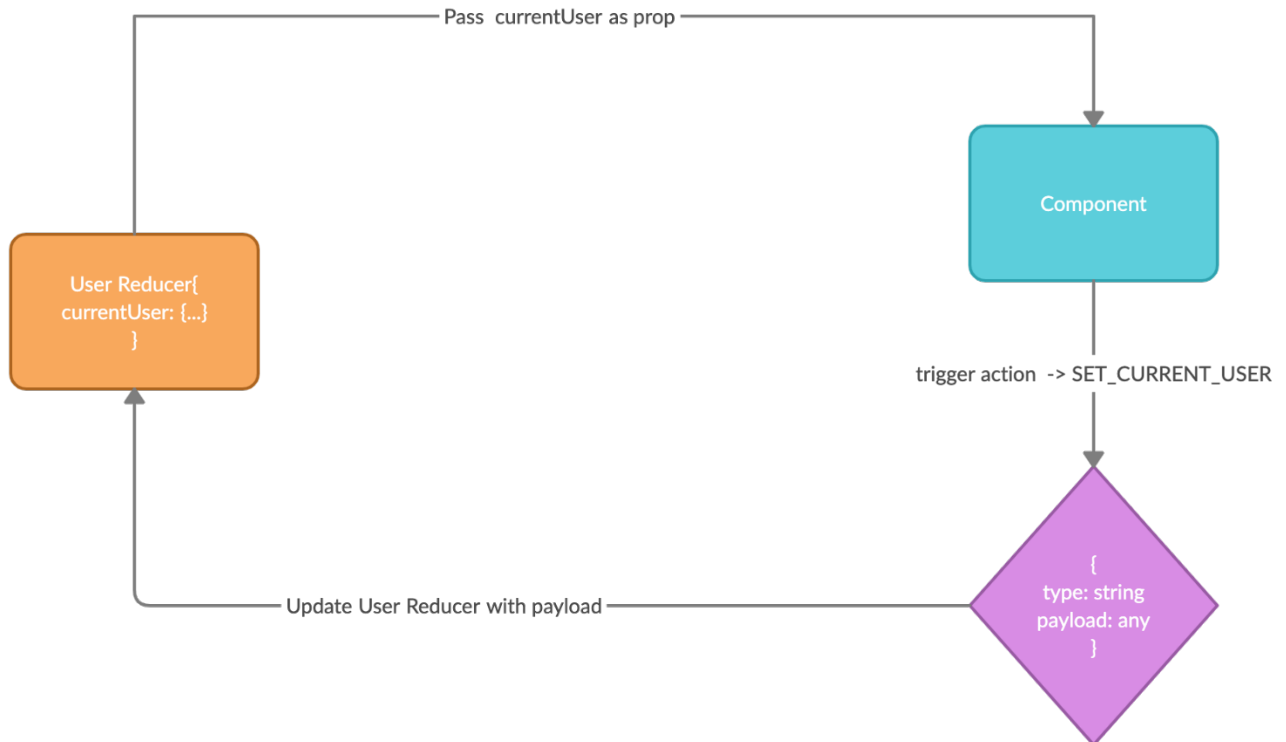


FIGURE 7. Redux Action process

3.3 Reducer

The Reducer is the most significant aspect of Redux. The functionality of a reducer is not unique to Redux. It is associated with the reducing method in JavaScript that accepts an accumulation and a value and returns a new accumulation. Reducing functions is a core concept of functional programming (Simpson 2016, 96.) In Redux, the accumulated value is the slice of state object and the value being accumulated is the actions. The reducers determine a new state based on the given action and its previous state. The reducer must be a pure function, free of side effects. Side effects are actions that change a variable from outside its scope (Garreau & Faurot 2018, 15.) It is imperative to note that reducers should never make API calls. Reducers have advanced features such as hot reloading and time travel. Hot reloading is a process of replacing pieces of code without restarting the entire application. (Banks & Porcello 2017, 220.) Time travel in Redux enables actions to be reversed or repeated without rewriting the logic that created the action. For instance, a list of items in the state can be removed by dispatching an action, this action can be reversed to recover the removed items and vice versa. (Geary 2018, 67.)

3.3.1 Unit Reducer

In large applications, the reducer is split up into multiple functions, each managing its slice of the global state. From FIGURE 4, three separate unit reducers are combined in a root reducer. Each unit reducer handles logic pertaining to the slice of state in question. For instance, the shop reducer only handles logic pertaining to shop actions such as add to cart, filter by a category, and pagination of shop content. The separation of the unit reducers prevents errors associated with updating the parts of the global state which do not need to be updated. Each unit reducer is independent and does not interfere with the login of the other unit reducers. It is worth to note the unit reducers are named after the slice of the state they manage. (Banks & Porcello 2017, 235.)

3.3.2 Root Reducer

The root reducer also known as the combine-reducer converts the unit reducing functions into a single function which returns the global state which is then passed to the store. Its primary function is to preserve the logical separation of the unit reducers. The core functionality of Redux revolves around the reducer and therefore there are two general strict rules to be adhered to when creating a root reducer. Firstly, for action to be executed, it must return the state given to the reducer function as a first argument. Secondly, the reducer must never return undefined (Banks & Porcello 2017, 240.)

```
import { FetchActions } from 'common/redux/actions/features'
import { FeaturesState, LOAD_SUCCESS } from 'common/redux/types'

export function features(state: FeaturesState = {}, action: FetchActions): FeaturesState {
  switch (action.type) {
    case LOAD_SUCCESS: {
      const { feature, data } = action.payload
      return {
        ...state,
        [feature]: {
          all: data,
        },
      }
    }

    default:
      return { ...state }
  }
}
```

PICTURE 2. A Unit Reducer

4 ASYNCHRONOUS ACTIONS

JavaScript is a single-threaded language, which essentially means it executes code in sequential order. It has a single call stack and memory heap, with the functionality to execute a single task before moving on to the next. The call stack performs the code operations, and the memory heap stores all the variables. (Simpson 2016, 124.) This explanation covers synchronous operations, however, asynchronous operation are executed differently Asynchronous actions such as an API call or a timer can take a while to complete. In the JavaScript engine, these operations first go to the call stack just like the synchronous actions but are redirected to WebAPI's to handle the task. The event loop moves the completed task to the call stack only if it is empty. The event loop runs continuously, checking to see if the call stack is empty before picking up a newly completed task in the callback queue. (Burnham 2012, 4.)

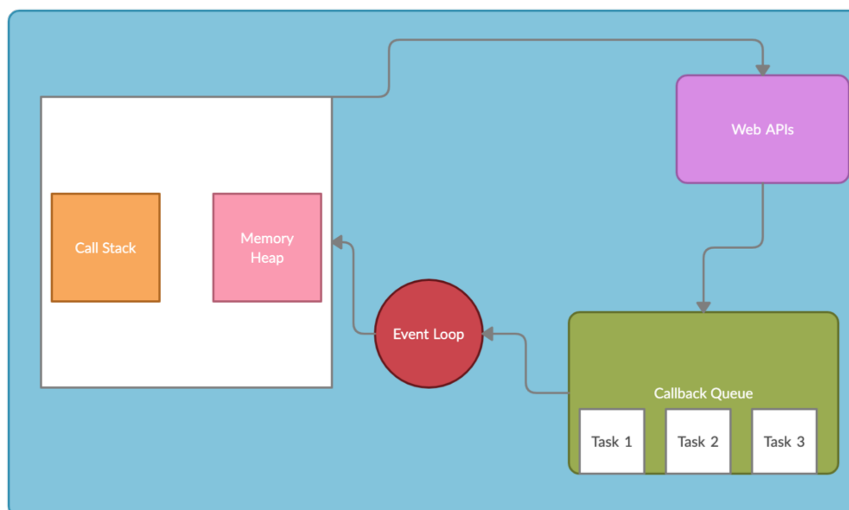


FIGURE 8. JavaScript Runtime Architecture

If a script takes a very long time to complete, it blocks the others. To ensure non-blocking code execution, the JavaScript engine has WebAPIs that handle multiple tasks concurrently. These WebAPIs are asynchronous, which essentially means they can run several tasks in the background and return a response once it is completed. Asynchronous code execution prevents the browser from freezing when a task takes a long time to complete. A callback function is always provided when making an asynchronous request. The function of the callback is to execute JavaScript code in the main thread once the WebAPIs completes its task. The event loop ensures only a single callback runs at any given time. The

subsequent callbacks in the queue would have to wait until the current one is completed. (Burnham 2012, 9.)

Upon execution of an asynchronous action, other synchronous tasks can occur simultaneously without affecting the user interface. When an asynchronous action such as an API call is executed, a change occurs in the application's state. The first change occurs immediately after the action is executed and when the second occurs when the task is complete. To trigger state changes, actions are dispatched to the reducers and they would be processed as synchronous requests. The first change informs the reducers that the action has begun. The second change notifies the reducers upon completion of a task. The reducers update the store depending on the success or failure of the task. To leverage asynchronous actions, middlewares are introduced in the Redux data flow to intercept every action before they reach the reducer. A middleware can be used for logging actions, dispatching new actions, reporting errors, and triggering subsequent asynchronous requests. There are two main middlewares used in Redux: Redux-Thunk and Redux-Saga. These middlewares are available as libraries that can be imported into any application with Redux functionality. (Dinkevich & Gelman 2017, 45.)

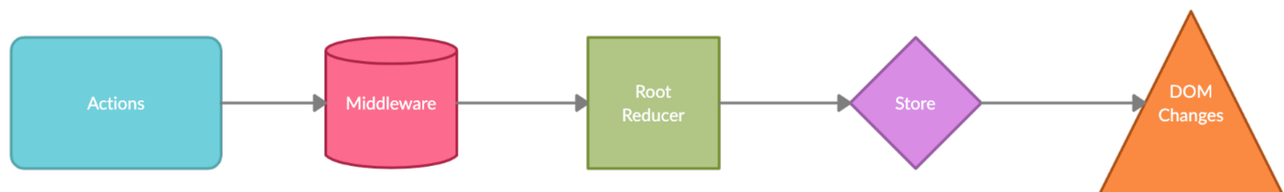


FIGURE 9. A Redux flow with Middleware

4.1 Redux-Thunk

This is a type of middleware that enables the developer to write Thunk action creators that return a function instead of an object as in the case of a regular Redux action. When an action is dispatched, Redux-Thunk can either delay this dispatch or cancel the dispatch if it does not meet certain requirements (Salcescu 2020, 52.) PICTURE 3 is a snapshot of a dispatch action. This action fetches the data of the products in the backend, and this could take some time depending on the size of the data. The action returns another dispatched action that activates the fetch and depending on the response, it dispatches a success action with a payload of the data received or an error action describing what might have gone wrong. It is worth noticing that fetchCollectionsStartAsync is a Thunk action creator, which returns a dispatch function. On the other hand, fetchCollectionsStart is an asynchronous action that dispatches an

API call. The fetchCollectionsSuccess/Failure actions are synchronous actions that are dispatched to the reducer to update the Redux store.

```
export const fetchCollectionsStartAsync = () => {
  return dispatch => {
    const collectionRef = firestore.collection('collections');
    dispatch(fetchCollectionsStart());

    collectionRef
      .get()
      .then(snapshot => {
        const collectionsMap = convertCollectionsSnapshotToMap(snapshot);
        dispatch(fetchCollectionsSuccess(collectionsMap));
      })
      .catch(error => dispatch(fetchCollectionsFailure(error.message)));
  };
};
```

PICTURE 3. A Redux-Thunk example

4.2 Redux-Saga

Asynchronous actions can cause side effects in the application. Side effects occur when a variable in a state is changed outside its scope of usage. When the application grows in complexity, the user actions increase exponentially. Moreover, if these actions were asynchronous the completion of each action must be intermittent. Redux-Thunk middleware utilizes callbacks, which makes the code hard to read. In some cases, these callbacks might also have callbacks, which adds to the complexity of data flow in the app. This peculiar situation is termed 'callback hell'. The Redux-Saga middleware eliminates callback hell and coordinates the data flow in a readable and systematic order. Testing and error handling are relatively simpler in Redux-Saga. (Garreau & Faurot 2018, 220.)

Redux-Saga's functionality is implemented using an ES6 feature called Generator functions or generators. They are based entirely on the concept of iteration. These functions can halt the execution of a task and resume execution from the exact point it was halted. It can return multiple values based on the

context of execution. The iterator has to be activated to fetch data from a generator. This is done by using a method called 'next'. When the 'next' function is called, the execution continues until it encounters the keyword 'yield'. 'Yield' halts the execution process. The output of a 'yield' can be passed to the 'next' function. If there are multiple functions to be executed, the 'next' method must be called. This renews the execution process until the next 'yield' is encountered. The 'next' method is called repeatedly until the desired outputs are obtained. Generators cannot be interrupted by an external function during task executions. It only halts mid-execution if a 'yield' is encountered. If there are no 'yields' present it continues execution until completion, similar to the execution of a regular function. (Garreau & Faurot 2018, 225.)

Implementing Redux-Saga requires special helper functions that, encapsulate generator functions to spawn new tasks when an action is dispatched. These functions are called effects. There are over 20 effects found in the Saga library. The 'takeEvery' effect, which is most commonly used allows multiple instances of the dispatched action to be executed. The 'call' effect, which provides a gateway for API requests to be executed, and the 'put' effect dispatches an action to the Redux store. The 'takeLatest' effect that only executes the latest dispatched action. (Garreau & Faurot 2018, 230.)

```
export function* loadOne() {
  yield takeEvery(LOAD_ONE, function* ({ payload: { query, id } }: FetchLoadOne) {
    try {
      const response = yield call(Services.loadData, query)
      if (response) {
        yield put(loadOneSuccess(id, response))
      }
    } catch (error) {
      yield put(loadFailure(error.errors[0].message))
    }
  })
}
```

PICTURE 4. A Redux-Saga Generator function

In reference to PICTURE 4, each time the 'FetchLoadOne' action is dispatched, it is intercepted by the 'loadOne' Saga and the API request is made using the 'call' Effect. If the request is successful, a new action is dispatched with a payload to update the store using the put Effect. Failed requests move to the catch block and a failure action is dispatched with the error payload also using the 'put' Effect.

The functionality of Saga to spawn multiple functions when a single action is dispatched makes this library very powerful and dominant over Redux-Thunk. Multiple asynchronous actions can occur without side effects in the application. A simple Saga function can execute multiple tasks in a single function, whereas achieving the same result without Sagas would require writing a significant amount of code. (Garreau & Faurot 2018, 238.)

The concept of implementation can be convoluted but the benefits are innumerous as discussed by Garreau & Faurot (2018, 250.) PICTURE 5 is an example of a Saga function which executes multiple tasks when the user signs in with Google. The Saga intercepts the sign-in action and sends a request to the backend to fetch existing user credentials or create a new one if it does not exist. If the response is successful, it is dispatched as a payload to the reducer. The user interface is refreshed with data from the new payload. The session of the user is activated and the token from the backend is saved in local storage and his token is checked periodically for validity. Requests made by the user require a valid token, if the server rejects the request the user is automatically logged out. When the sign-in is successful, the user interface displays a notification alerting the user of a successful sign-in. These notifications are cleared momentarily. All these processes occur in the same manner, each time another user logs into the application.

```
export function* onGoogleSignIn({ payload: { id_token } }: GoogleSignInStart) {
  try {
    const response = yield call(userActions.fetchGoogleUser, id_token);
    const googleResponse = response.data;
    yield put(signInSuccess(googleResponse));
    yield put(resetUiState());
    yield put(checkUserSessionStart(googleResponse.token));
    yield put(clearAllNotifications());
  } catch (error) {
    if (!error.response) {
      yield put(signInFailure('Cannot connect to network'));
      return yield put(clearAllNotifications());
    }
    const errorMessage = error.response.data.message;
    yield put(signInFailure(errorMessage));
    yield put(clearAllNotifications());
  }
}
```

PICTURE 5. A Google Sign-In Saga

Without using Sagas for this task, it would be difficult to track the data flow and debug errors if one of the tasks fails. In this instance, if the API request were to fail, the Saga moves the flow into the catch block. The catch block handles errors such as error server responses and dispatches a message to the reducer informing the user the server is unavailable. This advanced approach of state management can save a developer from encountering side effects in the application and helps other developers to understand the flow of data immediately. Garreau & Faurot (2018, 215) describe Sagas as a way to visualize asynchronous code as synchronous. Sagas enable developers to circumvent writing hardcoded complicated promises. It results in a cleaner and well-structured code. The only downside of using Sagas is the arduous learning curve that precedes proficiency.

5 EVALUATION AND DISCUSSION

To analyze the functionality of state management, the email login feature of an e-commerce store I built for educational purposes is examined. Access to the entire codebase is available on my personal GitHub account at https://github.com/dominickudiabor/clothing-store-frontend. The project was built using React with Typescript and React Hooks. In React, functional components can perform side effects using an ‘Effect Hook’. Hooks were introduced in React version 16.8. Local state in a component can be accessed using a ‘useState’ Hook. The local state can be updated using the ‘setState’ function. The sign-in component as seen in PICTURE 6 has an input for email and the password of the user. This input data is kept in the local state for validation purposes. The component also has redirected routes to the signup and password reset page. When the validation of the input is successful, the form submission dispatches an ‘emailSignInStart’ action with a payload of the email and the password. The ‘useDispatch’ Hook is used for dispatching actions.

```
const SignIn = () => {
  const [input, setInput] = useState({
    email: '',
    password: '',
  })

  const dispatch = useDispatch()
  const history = useHistory()

  const handleChange = (event: { target: { value: string; name: string } }) => {
    const { value, name } = event.target
    setInput({ ...input, [name]: value })
  }

  const handleToggle = () => {
    dispatch(toggleSignUpForm())
  }

  const handleReset = () => {
    history.push('/password/requestSignIn')
  }

  const handleEmailSubmit = async (event: { preventDefault: () => void }) => {
    const { email, password } = input
    event.preventDefault()
    if (email && password !== '') {
      dispatch(emailSignInStart(email, password))
      setInput({ email: '', password: '' })
    } else return
  }
```

PICTURE 6. The Sign-In Functional Component

The reducers are informed of the start of the action as seen in PICTURE 7. The reducers are notified when the action type matches the case type of the reducer. User reducers have multiple cases, each with a specific action type. This is to prevent the occurrence of mutating the wrong state. The reducers update the user state by toggling the loading status of the sign-in request. The loading state displays a spinner animation on the user interface.

```
case EMAIL_SIGN_IN_START: {
  return {
    ...state,
    isLoading: true,
    error: null,
    adminModification: null,
  };
}
```

PICTURE 7. The User Reducer Function Returning A New State

```
export function emailSignInStart(email: string, password: string) {
  return {
    type: EMAIL_SIGN_IN_START,
    payload: { email, password },
  };
}

export function signInSuccess(data: { token: string; user: NewUser }) {
  return {
    type: SIGN_IN_SUCCESS,
    payload: { data },
  };
}

export function signInFailure(error: string) {
  return { type: SIGN_IN_FAILURE, payload: { error } };
}

export function signOut() {
  return {
    type: SIGN_OUT,
  };
}
```

PICTURE 8. The Sign-In Actions

The actions are defined in a separate file. Each has a definitive type and is written in capital letters for debugging purposes. The payload is optional. The actions, responsible for the sign-in feature are seen in

PICTURE 8. The dispatched action is intercepted by the user Sagas. There are multiple user Sagas, each responsible for a specific action. The Saga responsible for email sign-in takes the payload from the form submission as an argument. The try-catch block is used because the action is asynchronous. In the try block, the call effect attempts an API request with the help of a service function as seen in PICTURE 9.

```
fetchEmailUser: async (email: string, password: string) => {
  const response = await axios.post(`${userUrl}login`, { email, password });
  return response;
},
```

PICTURE 9. The Fetch-User Service Function

A successful response from the API returns the user credentials as an object. This triggers the next action which takes the response as a payload and dispatches it to the reducer. The reducer updates the store, which triggers the next action in the Saga. This action re-renders the user interface to display the logged-in user's credentials. The next function in the Saga is an action that triggers the session duration of a user. Finally, all the notifications on the user interface are cleared. Errors with API requests are directed to the catch block. If the API is unresponsive it dispatches an action to the reducer with a payload of the network error. It also handles custom error responses from API requests by dispatching an action to the reducer with a payload of the error message. The Saga actions are illustrated in PICTURE 10.

```
export function* onEmailSignIn({
  payload: { email, password },
}: EmailSignInStart) {
  try {
    const response = yield call(userActions.fetchEmailUser, email, password);
    const emailResponse = response.data;
    yield put(signInSuccess(emailResponse));
    yield put(resetUiState());
    yield put(checkUserSessionStart(emailResponse.token));
    yield put(clearAllNotifications());
  } catch (error) {
    if (!error.response) {
      yield put(signInFailure('Cannot connect to network'));
      return yield put(clearAllNotifications());
    }
    const errorMessage = error.response.data.message;
    yield put(signInFailure(errorMessage));
    yield put(clearAllNotifications());
  }
}
```

PICTURE 10. The Email Sign-In Saga

The user reducer handles all state mutations concerning the user state. The user state contains properties associated with the user. Reducers take two parameters: the slice of state, in this case, the 'userState', and the action. It always returns a new state if there is an update, else it would return the previous state as illustrated in PICTURE 11. Since there are multiple user actions, a switch statement is often used to make the reducer logic more readable. For every case in the switch statement, the current state must always be returned in addition to the properties that have to be updated. The action-type determines which case must occur. The state must always be returned if there are no matching cases. The sign-in success action has a payload of the user credentials and the authentication token. The token is for identification when API requests are made, and it also contains the session time. In the user state, the loading state is toggled to false to deactivate the spinner animation. The current user object is updated with the newly logged-in user. And the token is also updated to begin the user session.

```
  state: UserState = {
    filteredUsers: [],
    notification: null,
    isLoading: false,
    currentUser: null,
    token: undefined,
    sessionExp: undefined,
    error: null,
    toggleNotifications: false,
    adminModification: null,
    adminUsers: [],
  },

  action: UserActions
): UserState {

  switch (action.type) {
    case EMAIL_SIGN_IN_START: {
      return {
        ...state,
        isLoading: true,
        error: null,
        adminModification: null,
      };
    }

    case SIGN_IN_SUCCESS: {
      const { token, user } = action.payload.data;

      return {
        ...state,
        notification: 'Sign in successfull',
        isLoading: false,
        currentUser: user,
        token: token,
      };
    }
```

PICTURE 11. The User Reducer

The’ useSelector’ Hook from React-Redux is used to access properties of the state in a functional component. The properties required from the state are destructured as a best practice. Destructuring assignment is a JavaScript syntax that enables properties of an Array or Object to be assigned to separate variables. It is worth noting that only the properties required are destructured. The’ useEffect’ Hook has a callback function as its first argument, this function is called each time the application renders. Rendering occurs when React performs a DOM update. The callback runs only after the update is complete. The second argument is an array that contains active subscriptions. When the subscription changes, the callback function is called. By default, this Hook runs after each render, but the subscription controls the frequency of its execution. If there are no subscriptions, then it runs each time there is a change in the React DOM. PICTURE 12 shows the ‘useEffect‘ Hook used for listening to changes in the session expiry time. The Hook checks the validity of the session by comparing the session expiry time to the current time. An expired session dispatches a sign-out action. The subscription is referred to as a dependency in the Hook. This is because the function only runs if the session expiry time exists.

```
export default function App() {
  const dispatch = useDispatch();

  const { sessionExp: sessionExpTime, token } = useSelector(
    (state: AppState) => state.user
  );

  axios.defaults.baseURL = "http://localhost:9000/api/v1/";
  token && (axios.defaults.headers.common["Authorization"] = `Bearer ${token}`);
  useEffect(() => {
    function activateSessionCheck(time: number) {
      if (Date.now() > time) {
        dispatch(signOut());
      }
    }
    sessionExpTime && activateSessionCheck(sessionExpTime);
  }, [dispatch, sessionExpTime]);

  return (
    <>
      <Routes />
    </>
  );
}
```

PICTURE 12. The ‘UseEffect ‘Hook

The ‘AppState’ also is known as the state tree represents the global state of the application. It is managed by the Redux store. By convention, it is a Map which is an object with key-value properties. This enables the data to be serializable. As a rule of thumb, data cannot be stored in the state object if it cannot be turned into JSON data. It is considered a best practice to label the keys as the corresponding slices of state. In PICTURE 13, there are four slices of state. The keys represent the name of the individual state slices. The values contain properties that store data accessible by the entire application. The slices of state are managed independently by unit reducers.

```
const initState: AppState = {
  cart: {
    hidden: true,
    cartItems: [],
  },
  product: {
    adminEdit: null,
    sections: [],
    shopData: {},
    adminProductData:[],
    filteredProducts:[],
    notification:null,
  },
  ui: {
    dialogOpen: {},
    toggleSignUp: false,
    verifiedAdmin: false,
    highlightName:undefined,
    adminView: true,
  },
  user: {
    notification: null,
    isLoading: false,
    toggleNotifications: false,
    currentUser: null,
    token: undefined,
    sessionExp: undefined,
    error: null,
    adminModification:null,
    adminUsers:[],
    filteredUsers:[],
```

PICTURE 13. The App State

6 CONCLUSION

The thesis focused on the structure and data flow process of React-Redux and provided a deeper understanding of state management. Applications are built primarily to control the flow of data from one component to the other. It is imperative to ensure the data flow is consistent to avoid unexpected errors in production. When applications are designed, the user experience has the highest priority. Large applications such as, an e-commerce store, display a colossal amount of data. This data is manipulated in multiple ways depending on the features and functionalities of the application. For instance, if the user is on the shop page, they can filter out options or use the sort and search functionality. If the user clicks to add a product to the cart, the cart items should be accessible on any page the user is currently on. State management enables data to be readily available to every component within an application. Shared data cannot be kept in a single view. Localizing the data is the most efficient way to provide access to all components in the application.

Many state management libraries can be used in a React application, such as MobX, apolloGraphQL, ContextAPI, Pullstate, and RxJs. Each has unique perspectives on state management; however, the most recommended library is React-Redux, which was built by React developers specifically for React applications. The terms React-Redux and Redux have been used interchangeably throughout the context of this thesis. Contrary to the similarity in terminology, they are not the same. To dispel the discrepancies, it is important to state that Redux is a library for managing application state but React-Redux is a library for managing React application state. Redux can be used in Angular, Vue, Ember, and vanilla JavaScript applications. It is not limited to only React; it manages state in all JavaScript-based applications. It follows the flux architecture. React- Redux on the other hand is designed specifically to work with React applications. It creates containers that listen to changes in the store and updates the components that display the updated data.

There are many tradeoffs associated with React-Redux, the most prominent is the boilerplate required for it’s setup is extremely complicated and intricate. Wiring a React application to use Redux might require a major refactoring of many components. The data flow logic must be reconstructed to fit the Redux model. The official documentation can seem overwhelming to a beginner. It is an advanced concept and the requisite for comprehension is based on the familiarity with writing code in React. There are three major limitations a developer might encounter when working with Redux. Firstly, it requires the application state to be described only as an object or array. If a developer chooses to store data

differently, then Redux might not be a suitable option. Secondly, changes that occur in the store can only be described as plain objects. And finally, the logic pertaining to handling these changes must be pure functions. There is no workaround to these limitations, they must be simply followed and strictly adhered to.

These limitations are not required when building a React application, they are only enforced when the developer decides to opt for React-Redux. These tradeoffs must be considered when designing an application because the entire workflow of the application depends on it. On the plus side, Redux can enhance user functionality and dramatically increase the performance of an application. It leverages local storage to persist data and it loads the app from the initial launch with this data. Circumstances, which require data refresh such as accidentally closing a browser or losing connection momentarily to the backend, can be resolved by using Redux persist, a great feature of Redux. Persistence enables Redux to retrieve previously saved store data from the browser and rehydrates the store upon launch of the application. There is no need to send API requests to the backend to retrieve data that has previously been loaded.

State management can occur in a component, this is referred to as local state management and it can co-exist with Redux. Implementing global state management does not necessitate the elimination of all local state. The local state can be useful for components that control their data and do not have to share the data with any other component. A good example would be the profile update component of the application. Updating a profile requires a form which has multiple inputs depending on the changes required. When the user enters information, there could be a data validation logic to check the input. The input is usually controlled by the local state. If the validation passes, the component dispatches an action with the form data which is intercepted by Redux middleware. The middleware sends a request to the backend to implement the change and the response updates the store with the new user credentials. The components that display user properties are automatically re-rendered to display the new data. This is an instance of how the power of Redux can be leveraged with the local state.

Redux is an incredibly useful library for React. Providing an easier way to manage the convoluted state within an application. React was built specifically for creating user interfaces, so it is advantageous if a separate library managed the data flow. This way, the logic of the application is easier to understand, and errors are easier to find. However, the most prevalent drawback faced by developers is the amount of boilerplate code required to set up Redux. With the introduction of Hooks, it cuts down the boilerplate code by a substantial percentage. React is preparing to roll out a package called the "Redux Toolkit" to enable developers to configure Redux without the complicated boilerplate code.

REFERENCES

Banks, A., Porcello, E. 2017. Learning React. Functional Web Development with React and Redux. California: O'Reilly Media.

Burnham, T. 2012. Async JavaScript: Build more responsive apps with less code. Pragmatic Bookshelf.

Desjardins, P. 2018. .Net Knowledge Book: Typescript, React, and Redux. Depot – Legal Bibliotheque et Archives national du Quebec.

Dinkevich, B., Gelman, I. 2017. The Complete Redux Book: Everything you need to build real projects with Redux. Second Edition. Self-published.

Garreau, M. Faurot, W. 2018. Redux in Action. First Edition. Manning Publications.

Geary, D. 2018. Building React.js Applications with Redux. First Edition. Pearson Technology Group Canada.

Lee, J., Wei, T., Mukhiya, SK. 2019. Redux Quick Start Guide: A beginner's guide to managing app state with Redux. Packt Publishing.

React Official Documentation 2020. Available: https://reactjs.org/. Accessed 17 October 2020.

React-Redux Official Documentation 2020. Available: https://react-redux.js.org/. Accessed 28 October 2020.

Redux Official Documentation 2020. Available: https://redux.js.org/. Accessed 20 October 2020.

Redux-Saga Official Documentation 2020. Available: https://redux-Saga.js.org/. Accessed 26 October 2020.

Salcescu, C. 2020. Functional Architecture with React and Redux. Amazon Digital Services LLC.

Simpson, K. 2016. You Don't Know JS. ES6 and Beyond. O'Reilly Media.

TypeScript Official Documentation 2020. Available: https://www.typescriptlang.org/. Accessed 12 October 2020.

Voorhees, D. 2020. Guide to Efficient Software Design. An MVC approach to concepts, structures, and models. Springer publishing.

Wieruch, R. 2020. The Road to React. Your journey to master plain yet pragmatic React.js. Self-published.