MAKERS ACADEMY, PORTFOLIO SUBMISSION

No. 5

Placement: Compare the Market
Apprentice: Krzysztof Kasprzak

Table of Contents

Redux state management task

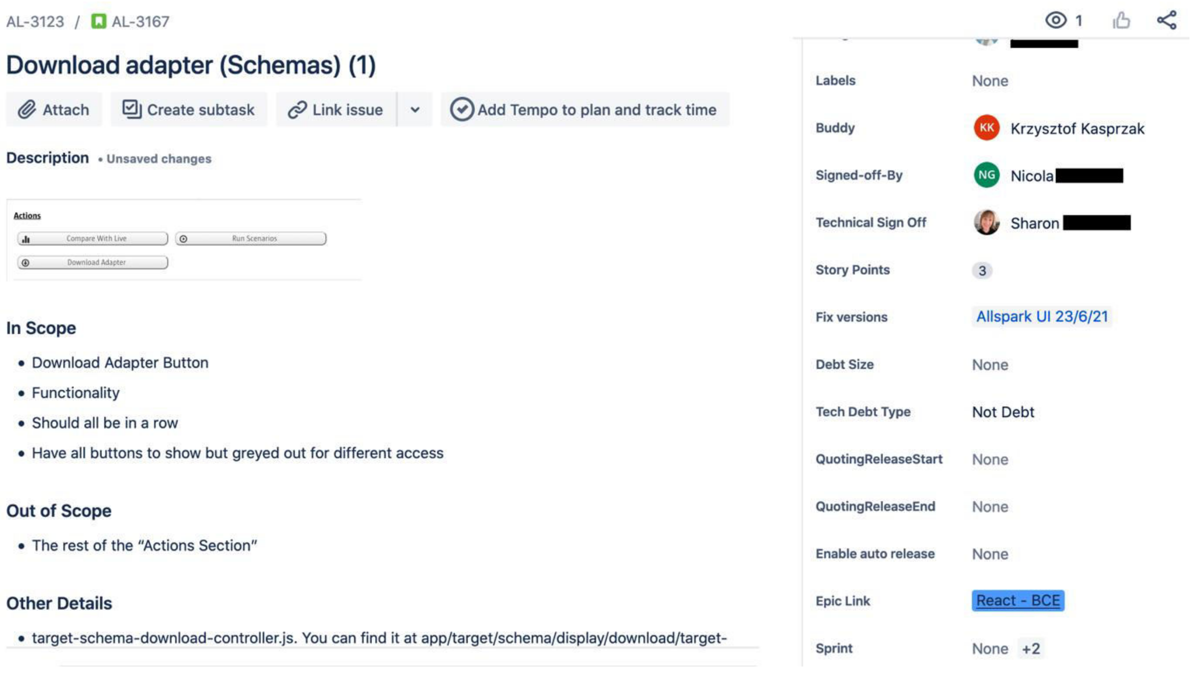
The task

Project, tools, and libraries I worked with

As per previous tickets, the task I am about to describe is contributing towards rewriting the Allspark UI (user interface) from Angular JS framework to React JS library project. As now usual, it required from me being able to understand the structure and syntax of the old project to be able to recreate its functionality in the new, front-end project using React library. This time however, I want to go into the details of how I incorporated and the usage of React-Redux library. It is a Java Script library that allows developers to devise and apply a client-side global storage for state management of properties (data) to be shared amongst react components.

The ticket, it's requirements and how I got involved

Ticket's serial number was AL-3167 (as seen on fig. 1.1 below) and it was another ticket tagged with an epic of `React-BCE` (ticket for developing a new UI).



AL-3123 / AL-3167

Download adapter (Schemas) (1)

Description • Unsaved changes

Actions

In Scope

- Download Adapter Button
- Functionality
- Should all be in a row
- Have all buttons to show but greyed out for different access

Out of Scope

- The rest of the "Actions Section"

Other Details

- target-schema-download-controller.js. You can find it at app/target/schema/display/download/target-

[redacted]

Labels	None
Buddy	Krzysztof Kasprzak
Signed-off-By	Nicola [redacted]
Technical Sign Off	Sharon [redacted]
Story Points	3
Fix versions	Allspark UI 23/6/21
Debt Size	None
Tech Debt Type	Not Debt
QuotingReleaseStart	None
QuotingReleaseEnd	None
Enable auto release	None
Epic Link	React - BCE
Sprint	None +2

Fig.1.1, screen capture of a ticket outline as presented in Jira.

The task was estimated to take 2 days to complete in the previous refinement session where all the team's engineers (including me) agreed on the estimate value.
As can be seen, I was a `Buddy`, that is a pair engineer to work on it; another assignee was Tim, Allspark's engineer who I work most often with.
The ticket's completion criteria were to develop a functional button on the provider details page (main hub for all the functionalities involved with working on given provider's details) which would allow project developers to download displayed insurance provider's adapter file.

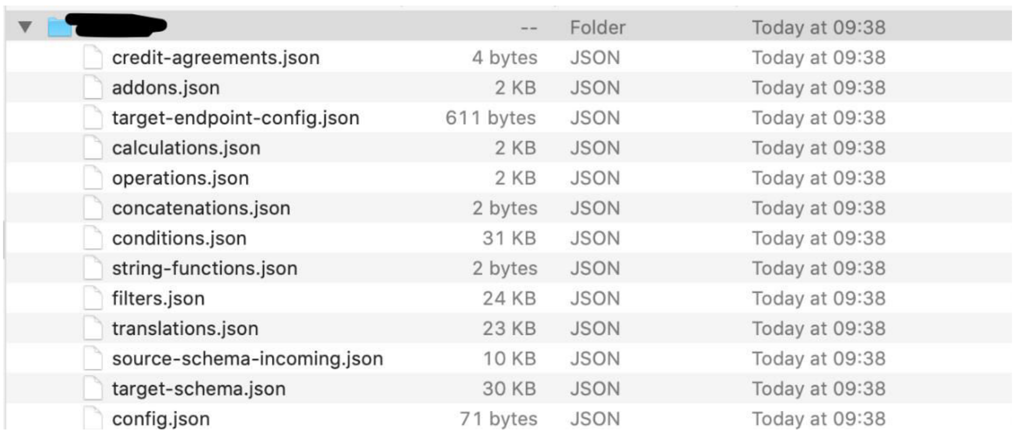
The ticket also included specifics about:

- The styling of the button, i.e. keeping it visible, but greyed out and not functional even for non-developer users.
- The positioning of the button, i.e. on the panel of buttons associated with `Actions` associated with a provider.

My participation in completion of the ticket was required, as out of the Allspark team, I was the most experienced in using React-Redux, a library in use for global, client-side state management. This library was needed to store information about logged in user being (or not) a developer to enable/disable the adapter downloading functionality.

What is adapter and why is it useful to be able to download it

The provider adapter file is a folder packed with .json files that contain information about given provider policy details (see an example of a file structure on the fig. 1.2 below).



[redacted]	--	Folder	Today at 09:38
credit-agreements.json	4 bytes	JSON	Today at 09:38
addons.json	2 KB	JSON	Today at 09:38
target-endpoint-config.json	611 bytes	JSON	Today at 09:38
calculations.json	2 KB	JSON	Today at 09:38
operations.json	2 KB	JSON	Today at 09:38
concatenations.json	2 bytes	JSON	Today at 09:38
conditions.json	31 KB	JSON	Today at 09:38
string-functions.json	2 bytes	JSON	Today at 09:38
filters.json	24 KB	JSON	Today at 09:38
translations.json	23 KB	JSON	Today at 09:38
source-schema-incoming.json	10 KB	JSON	Today at 09:38
target-schema.json	30 KB	JSON	Today at 09:38
config.json	71 bytes	JSON	Today at 09:38

Fig.1.2, shows a file structure of an exemplar adapter provider download.

As can be seen on the figure above, provider adapter includes .json files that outline details such as policy addons or credit agreements. It also holds more technical information, such as translation details (details of how the policy should be displayed on the customer-facing website), or configuration file for the customer-facing site to be able to correctly display provider data. Since the file holds the vital configuration and information details, it comes in handy when developing and/or debugging new or existing code. Knowing what the files hold, it is easy to manually verify if these are read and processed correctly.

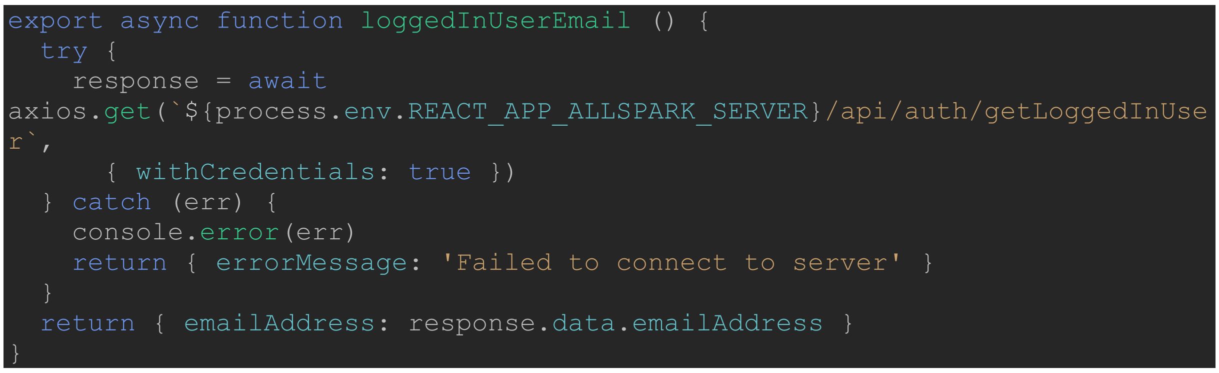
Reasons for developer only access to the feature

Since the adapter file is not being directly used by the end users of the UI, it does not have to be downloadable for them. By design, it is not a case of keeping details confidential, the users are internal, plus the details of data kept in the file or formatted and displayed by the UI.
Instead, it is a principle of least privilege pattern. This means that, since the users don't use the file, they don't need the access to it. The access design pattern of least privilege is a best practice approach that enforces app security by making sure that only specifically authorised users can use it. Narrowing down the pool of agents who can potentially exploit it.

Rationale behind data modelling and its client-side storage

Need for shared information across branched out components

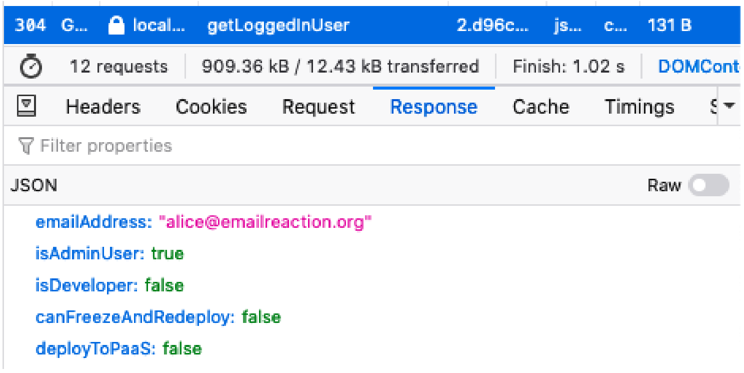
As part of this ticket, to only allow developers to be able to download the provider adapter file, it was required to call for (via API call) and store information details of a user on his/her logging in action. A call for user details has already been implemented in a log in/out button component, however the information about the user's role was not yet stored anywhere. At that stage (before any work on the described ticket took place), the only user detail being recorded was the user's email which then was being used to display his/her credentials in the header of the app (see fig. 1.3 below to find the snippet of the service call to fetch user email).



```
export async function loggedInUserEmail () {
  try {
    response = await
axios.get(`${process.env.REACT_APP_ALLSPARK_SERVER}/api/auth/getLoggedInUse
r`,
      { withCredentials: true })
  } catch (err) {
    console.error(err)
    return { errorMessage: 'Failed to connect to server' }
  }
  return { emailAddress: response.data.emailAddress }
}
```

Fig. 1.3, a snippet of a code I developed in some previous ticket. This method makes a call to relative URL path: `api/auth/getLoggedInUser`. Upon receiving the response, it specifically returns a user's email address.

As seen on the figure below (fig. 1.4), the response of the service call that is being made by a code snippet in the figure above, includes more than just an email address, amongst other properties, the response object also includes information about whether a user is or is not a developer (`isDeveloper`).



304 G... local... getLoggedInUser 2.d96c... js... c... 131 B
12 requests | 909.36 kB / 12.43 kB transferred | Finish: 1.02 s | DOMCont
Headers Cookies Request Response Cache Timings
Filter properties
JSON
Raw

```
emailAddress: "alice@emailreaction.org"
isAdminUser: true
isDeveloper: false
canFreezeAndRedeploy: false
deployToPaaS: false
```

Fig. 1.4 depicts a body of a response object that is being received upon using GET call in function `loggedInUserEmail` on the fig. 1.3.

The need to fetch and store a `isDeveloper` property across the components of the project stemmed from the fact, that in the future development, there will be several other components incorporated that will need be aware of developer role status of a user.

Minimising the amount of API calls

One possible solution to provide future React components with the needed user detail would be to allow them to connect to the server-side and fetch the required data. This, however, would be a bad practice as invoking unnecessary service calls would require time, making the app less responsive, leading to worse user experience. It would also put more undesired strain on the back-end. Making more calls to the API would not be efficient use of the server and could introduce slower response times if overloaded (further lowering the quality of user experience).

Thus, the solution of storing and passing relevant data and its state on the client-side was considered as a better solution.

Redux, what is it and why we implemented it

Redux, more specifically React-Redux is an official binding library from creators of Redux state management library for React projects. The main difference is that Redux on its own is a state management library that can be applied to any JavaScript framework/library project. Whereas React-Redux has been specifically created by the creators of Redux to support easier implementation of Redux data state management in React library projects.
The principal motivation behind applying React-Redux is that it is a tool that help to manage and define what props (data) is being passed down to what components at what events. It helps to control it and distribute it so that every component. It does it by uploading and updating data pieces to an application global state storage, allowing for an easy access to the required data.
With that, it does not only allow for better data management but also helps to pass props amongst the components no matter what their relationships are. The data flow for the Redux Store can be seen on the figure below (fig. 1.5).

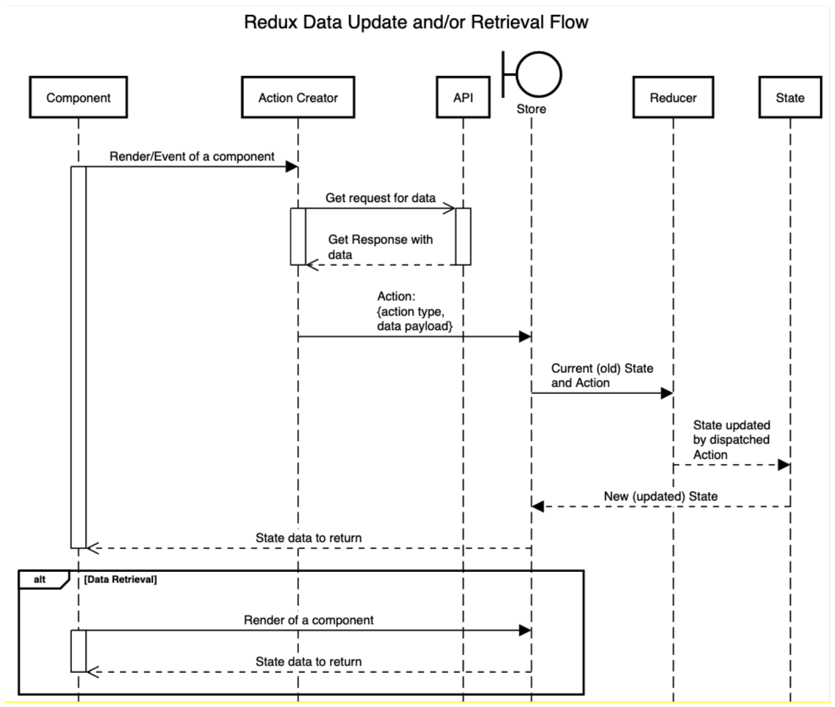


Fig. 1.5, sequence diagram I've created to represent the redux data flow. As seen on it, when the react component dispatches an action to update and retrieve data, redux actions (action creators) are called to fetch and update data. This fetched data is passed to reducer (the action type lets the root reducer know which sub-reducer must use) in a Redux store which takes the stale state and updates it by relevant data. Only then, the data that component requested is being passed back. In a case where there is no need for update, data is being passed to the component directly form the Redux store. This way data doesn't have to be passed between components and can be sourced straight from the global Redux store.

With the detailed control of data props and their state and the ability of Redux to pass them around the entire app, it was a decided to be applied. Additionally, Teletraan (Allspark's sister team) also uses React-Redux which was a significant factor in deciding how to manage the client-side state. With a sister team already having experience, skill transfer was possible. It also further enhanced the idea of interchangeability of engineers between the Teletraan and Allspark projects.

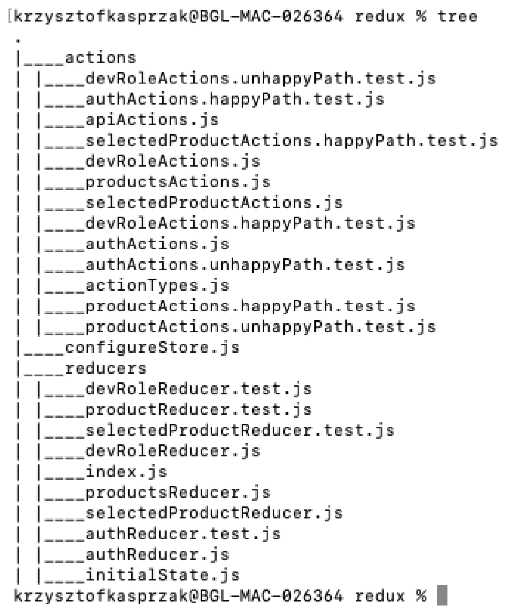
React-Redux versus Redux Toolkit

The React-Redux library is not the recommended library to use when it comes to adding Redux to a new project. For a long time, it was, yet users of the library did highlight its many problems were:
- Complicated configuration,
- Freedom of choice for the packages needed to initialise the functionality (which can be overwhelming),
- Code boilerplate.

In response for some time now, there exists the recommended library which is called Redux Toolkit.
Albeit the creators' recommendations, Tim and I decided to implement it because:
- We wanted the freedom of choosing the packages to add to the project, we were not afraid of it,
- We found the experience of writing our own redux logic (aforementioned boiler plate) much educational and clearer to understand,
- I had an experience in using it and defining from scratch thanks to my personal development studies,
- Teletraan also uses React-Redux, keeping technology and package stacks similar across the teams is very beneficial,
- As stated in the Redux recommendation, both React-Redux and Redux Toolkit will be continuedly developed and maintained, Redux Toolkit is only recommended due to its easier usage.

Steps to complete the task

Our file structure design and separation of actions and reducers



```
[krzysztofkasprzak@BGL-MAC-026364 redux % tree
 .
 |____actions
 | |____devRoleActions.unhappyPath.test.js
 | |____authActions.happyPath.test.js
 | |____apiActions.js
 | |____selectedProductActions.happyPath.test.js
 | |____devRoleActions.js
 | |____productsActions.js
 | |____selectedProductActions.js
 | |____devRoleActions.happyPath.test.js
 | |____authActions.js
 | |____authActions.unhappyPath.test.js
 | |____actionTypes.js
 | |____productActions.happyPath.test.js
 | |____productActions.unhappyPath.test.js
 |____configureStore.js
 |____reducers
 | |____devRoleReducer.test.js
 | |____productReducer.test.js
 | |____selectedProductReducer.test.js
 | |____devRoleReducer.js
 | |____index.js
 | |____productsReducer.js
 | |____selectedProductReducer.js
 | |____authReducer.test.js
 | |____authReducer.js
 | |____initialState.js
 krzysztofkasprzak@BGL-MAC-026364 redux %
```

Fig. 1.6, a file structure graph of a `redux` file within the project.

The file structure I introduced to Tim to keep the redux code ordered in efficient manner was rather simple. As fig. 1.6 (above) depicts it, the `redux` folder contains project's worth of code. It contains two sub-directories, one for actions the other for reducers. The `reducers` sub-directory also contains files that contain initial state for the client-side global state and index file. On the `redux` directory also contains store configuration file.

I have learned this structure by following an in depth React-Redux learning tutorial found on Pluralsight. The tutorial has advised to keep reducers and actions in different directories to keep the separation of concerns. To keep the cohesion of roles for each action-reducer pair, their always have the same, descriptive prefix. For example, as can be seen above, I made the action-reducer pair responsible for managing the state of `isDeveloper` Boolean to start with the prefix `devRole`.

Keeping Redux code separate from React components is coherent with creators of Redux libraries and with what Teletraan team adheres to.

At this point, it already is visible that usage of React-Redux library can create plenty of boiler plate code. However, Tim and I and Teletraan team agrees that it is much clearer to have plenty of small, single prop orientated action-reducers pairs, rather than one, big unwieldly action-reducer pair. It provides self-documented code (again, it also fits in with the library creators' recommendations).

Initial state, root reducer, and its configuration

To start using Redux's global storage, three essential files have to be created and set up:

1. Initial state file: as the name suggests, is a file that outlines the look of the initial state of props global storage when application renders. It is a simple file that exports a default object which includes props and their states. There can be more than one initial state files in the project, but all of those must be then combined when creating a store in the configuration file. In case of this project, I came forward with an idea to have one, centralised initial state file. The idea came from concluding that there will not be many props that the Redux store will hold. With that, it will only improve readability and ties in with yet another recommendation of Redux. Namely, creators of the library do advise to try and minimise number of variables stored in the state store. Redux should be used only when deemed useful and its store should not become a dump for all project's props. It would reduce efficiency, decrease readability, create more data noise, and make code maintenance more difficult. With that in mind, the initial state file I created and modified for the purpose of the ticket is simple (look at fig. 1.7 below):

```
export default {
  isLoggedIn: false,
  isDeveloper: false,
  products: [],
  selectedProduct: ''
}
```

Fig. 1.7, code snippet that outlines an initial state of Redux state store I created. As seen, I set `isDeveloper` Boolean value as false by default. It only makes sense, as before any update that might happen by receiving data about the user from the server-side, users should not be allowed to access unnecessary functionality.

2. Root reducer: reducers are functional elements that take in an initial state and based on any actions dispatched, it updates the props' state. What is important to know is that, although there can be many reducers in the project for many different action functions (to separate the concerns), they all have to be combined into so called root reducer. It is then being used in configuration file to initialise and manage the global state storage. What is essential to know, is that Redux library strongly implies functional programming and functional design, reducers do not mutate the state values. Instead, they just assign and update existing props. Functional programming and design are a software principal which favour creating usable modules out of functional components, not classes. It does so to achieve a goal of eliminating any side effects. Ultimate strive of functional design is to write pure functions (with no side effects). Hence, the need to keep the managed state immutable.

```
import { combineReducers } from 'redux'

import isLoggedIn from './authReducer'
import isDeveloper from './devRoleReducer'
import products from './productsReducer'
import selectedProduct from './selectedProductReducer'

const rootReducer = combineReducers({
  isLoggedIn,
  isDeveloper,
  products,
  selectedProduct
})

export default rootReducer
```

Fig. 1.8, a code written to create a `rootReducer`.

As the fig. 1.8 above depicts, root reducer imports all the sub-reducers and combines them into one using
Redux built-in function `combineReducers`. Worth mentioning is the naming convention. It is a common practice to name the sub0reducers passed in to the` rootReducer` by their corresponding props in the global state storage, this enhances readability of the code when comparing initial state and the reducer.

3. Configuration file: a place where the global state management gets created.

```
import { applyMiddleware, compose, createStore } from 'redux'
import reduxImmutableStateInvariant from 'redux-immutable-state-invariant'
import thunk from 'redux-thunk'

import rootReducer from './reducers/index'

export default function configureStore (initialState) {
  const composeEnhancers =
    window.__REDUX_DEVTOOLS_EXTENSION_COMPOSE__ || compose

  return createStore(
    rootReducer,
    initialState,
    composeEnhancers(applyMiddleware(thunk,
reduxImmutableStateInvariant()))
}
```

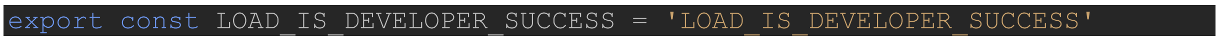
Fig. 1.9 is the body of a configuration file kept in the root of redux directory (see fig. 1.6 here).

The fig. 1.9 above includes quite a lot of not so obvious set up. Firstly, it imports Redux's functions: `applyMiddleware`, `compose` and `creatStore`. It imports a `redux-immutable-state-invariant` and `redux-thunk` libraries. `createStore` function, as its name suggests initialises the redux store ready to be used upon app render. It is being used by the defined and exported `configureStore` function which later gets invoked in the main `app.js` file to run the project. `compose` utility allows to invoke `applyMiddleware` which allows for extending the basic capabilities of Redux with additional libraries. In this case, `applyMiddleware` allows to extend the Redux functionality by `redux-immutable-state-invariant` which helps to detect and throws whenever any reducer attempts to mutate the state, and with `redux-thunk` which is a library that allows to dispatch functions as actions, not only objects as it is a case in the basic instalment of Redux.
Additionally, the line: ` window.__REDUX_DEVTOOLS_EXTENSION_COMPOSE__` allows the browser to use Redux DevTools Extension. A powerful browser extension that helps to track and debug Redux state in real time when using the application (more on the DevTools here).

Test driving the Redux functions

Having discussed the Redux set up and the file structure I incorporated, let's move on to how action-reducers pairs are created and how to test them.

Personally, I like implementing React-Redux in that manual way (rather than using Redux Toolkit), it gives the project a better, more predictable structure and development flow. Whenever I come to create a new action to dispatch and a reducer to handle it, I always start from creating a new action type. I keep them in the \`actionTypes.js\` file in the actions sub-directory (see fig. 1.6 here).



```
export const LOAD_IS_DEVELOPER_SUCCESS = 'LOAD_IS_DEVELOPER_SUCCESS'
```

Fig. 2.1, the only modification that had to be made to \`actionTypes.js\` file. Simple addition of a new action dispatch string.

\`actiontypes.js\` is a simple file which includes a list of all actions that can be dispatched to the root reducer. As seen on fig. 2.1 above, I only needed to add a new action type which would trigger a change of a \`isDeveloper\` prop in state storage upon successful loading of this user detail from API call.
Such simplistic change does not require any unit testing.

Next in line is test developing actions for loading and saving the \`isDeveloper\` prop. The pattern I follow is one I learned prior to working on this project and it is to have in the actions file (in this case it is \`devRoleActions.js\`) two functions:

1. \`getIsDeveloper\` which invokes a service call function to receive a data from API.
2. \`setIsDeveloperSuccess\` which takes as an argument the value of prop state to be updated. It is a function the \`getIsDeveloper\` invokes upon successful API call. This is the direct function that reducer (in this case \`devRoleReducer\`) takes in as an argument and acts on it updating the state.

Since, \`setIsDeveloper\` does not rely on any API call directly, it does not have to be tested against any error handling, only a happy path approach needs to be tested. Conversely, since \`getIsDeveloper\` relies on an API call, it needs to be able to handle network errors and refrain from potentially updating state with incorrect results. It needs to be tested along happy and unhappy paths. Please refer to figures 2.2 – 2.5 and their descriptions for evidence of code I produced in test driven manner to create appropriate Redux logic to model and update required piece of data.

```
import * as authorisation from '../../services/authorisation'
import * as types from './actionTypes'
import { apiCallError, beginApiCall } from './apiActions'

export function setIsDeveloperSuccess (isDeveloper) {
  return { type: types.LOAD_IS_DEVELOPER_SUCCESS, isDeveloper }
}

export function getIsDeveloper () {
  return async function (dispatch) {
    try {
      dispatch(beginApiCall())
      const response = await authorisation.isLoggedInUserDeveloper()
      dispatch(setIsDeveloperSuccess(response.isDeveloper))
      return response.isDeveloper
    } catch (error) {
      dispatch(apiCallError(error))
      throw error
    }
  }
}
```

Fig. 2.2, code written to create `devRoleActions`. The first, simpler function is a one that returns a dispatch call ('LOAD_IS_DEVELOPER_SUCCESS' in this case) and a passed in `isDeveloper` Boolean value. The second function is responsible of trying to make and catch a data response object (response object represented on fig. 1.4 here). Upon successful response it extracts only needed piece of data and models it in to be a state update by dispatching a `setIsDeveloperSuccess` function. When the API call fails, it does not attempt to update the state, and it only dispatches an error action to log that something went wrong.

```
import * as types from '../actions/actionTypes'
import initialState from './initialState'
export default function devRoleReducer (state = initialState.isDeveloper,
action) {
  switch (action.type) {
    case types.LOAD_IS_DEVELOPER_SUCCESS:
      return action.isDeveloper
    default:
      return state
  }
}
```

Fig. 2.3, a `devRoleReducer`. It is being triggered in the event of dispatching `setIsDeveloperSuccess` function and it updates the state of Redux `isDeveloper` prop by the value provided by aforementioned action function. In the unlikely event of the action function attempting to mutate the state, it defaults to initial state to avoid introducing any breaking errors.

```
mport * as actions from '../actions/devRoleActions'
import devRoleReducer from './devRoleReducer'
describe('When passed LOAD_IS_DEVELOPER_SUCCESS', () => {
  const initialState = {
    isDeveloper: true
  }
  let newState

  beforeEach( () => {
    const action = actions.setIsDeveloperSuccess(initialState)
    newState = devRoleReducer(initialState, action)
  })

  it('then it loads developer boolean', () => {
    expect(newState).toEqual(initialState)
  })
})
```

Fig. 2.4, simple test assertion for `devRoleReducer`. Since this reducer's job is only to make sure to update the global state storage with the latest `isDeveloper`, the only assertion to test here was to make sure it does that very thing. This rudimentary unit test uses jest testing library and nothing more. The structure of the assertion (with all the leading up actions defined in `beforeEach` wrapper) is team's standard.

```
mport * as types from './actionTypes'
import * as authorisation from '../../services/authorisation'
import * as devRoleActions from "./devRoleActions"
jest.mock('../../services/authorisation')
describe( 'When using getIsDeveloper successfully', () => {
    const isDeveloper = true
    let expectedOutcomes
    const mockedResponse = {
        isDeveloper
    }
    const dispatch = jest.fn()

    beforeEach(() => {
        expectedOutcomes = [
            { type: types.BEGIN_API_CALL },
            { type: types.LOAD_IS_DEVELOPER_SUCCESS, isDeveloper }
        ]
        authorisation.isLoggedInUserDeveloper.mockImplementation(() =>
Promise.resolve(mockedResponse))
        devRoleActions.getIsDeveloper()(dispatch)
    })

    it( 'then it creates BEGIN_API_CALL and
LOAD_IS_LOGGED_IN_DEVELOPER_SUCCESS', () => {
        expect(dispatch.mock.calls[0][0]).toEqual(expectedOutcomes[0])
        expect(dispatch.mock.calls[1][0]).toEqual(expectedOutcomes[1])
    })
})
```

Fig. 2.5, a snippet of happy path test suite for `devRoleActions`. It tests that the `getIsDeveloper` function correctly dispatches updates to Redux state upon retrieving a response from API call. Here, to make sure that I will only test the function, not the actual API call, I mocked the service call and made it return a promise of mocked responses to imitate its real-life workings. Also, to check that the function dispatches correct values, I also mocked the Redux's `dispatch` utility. Both of those actions ensured separation of concerns and stable behaviour of tests (not using API calls).

Abstraction of API call, and setter functions

Abstraction is an idea of hiding out unnecessary implementation details to focus on higher order problem. It is used in functional programming as much as it is in object orientated programming. While working on this task, I have used abstraction to increase modularity and readability of the code. As can be seem on the fig. 2.2 (under *‘Test driving the Redux functions`*) on page 10, the described `getIsDeveloper` function uses an abstracted away (imported from `authorisation.js`) API calling function `isLoggedInUserDeveloper` and separate `setIsDeveloperSuccess` to finally set the new Redux state. This way, the outer function ( `getIsDeveloper`) stays pure, by being concerned only about deciding what to do about received data supplied by `isLoggedInUserDeveloper` and invokes `setIsDeveloperSuccess` to handle the update.
The abstracted API calling function includes implementation details of setting up details of the call to be made (see the fig. 2.6 below). Tim and I have applied this kind of service call abstraction from the very start, to not only ensure the separation of concerns. Abstraction in functional programming enforces modularity of the code; hence, increasing its maintainability.

```
export async function isLoggedInUserDeveloper () {
  try {
    response = await
axios.get(`${process.env.REACT_APP_ALLSPARK_SERVER}/api/auth/getLoggedInUse
r`,
        { withCredentials: true })
  } catch (err) {
    console.error(err)
    return { errorMessage: 'Failed to connect to server' }
  }
  return { isDeveloper: response.data.isDeveloper }
}
```

Fig. 2.6, an example of the API calling function I have wrote to abstract away the details of setting up the URL code, its type, and its data to be sent with. It is worth to reflect that, as the application is going to grow, Tim and I are ware that for the benefit of even better maintainability, further abstraction can be investigated and potentially applied. For instance, all API calling functions can potentially be refactored to use more generic, get, post, put, delete helper functions.

Hooking React components to Redux state via connect

Finally, with the Redux logic tested and laid out, the only thing left to do is to connect relevant React components to Redux store to make them able to use state managed props. This ensures that the new Allspark UI project can base its behaviour on the prop state changes (in this case state change of developer role of a current user). It is an application behavioural state design, where written components render accordingly to given data state.

To connect any React component to Redux store, I use a `connect` Redux utility. It allows to wrap the component in such a way that it renders connected to Redux. In the example of allowing the download adapter button to be aware for Redux state of `isDeveloper` Boolean, following connection had to be implemented (see fig. 2.7 on the next page):

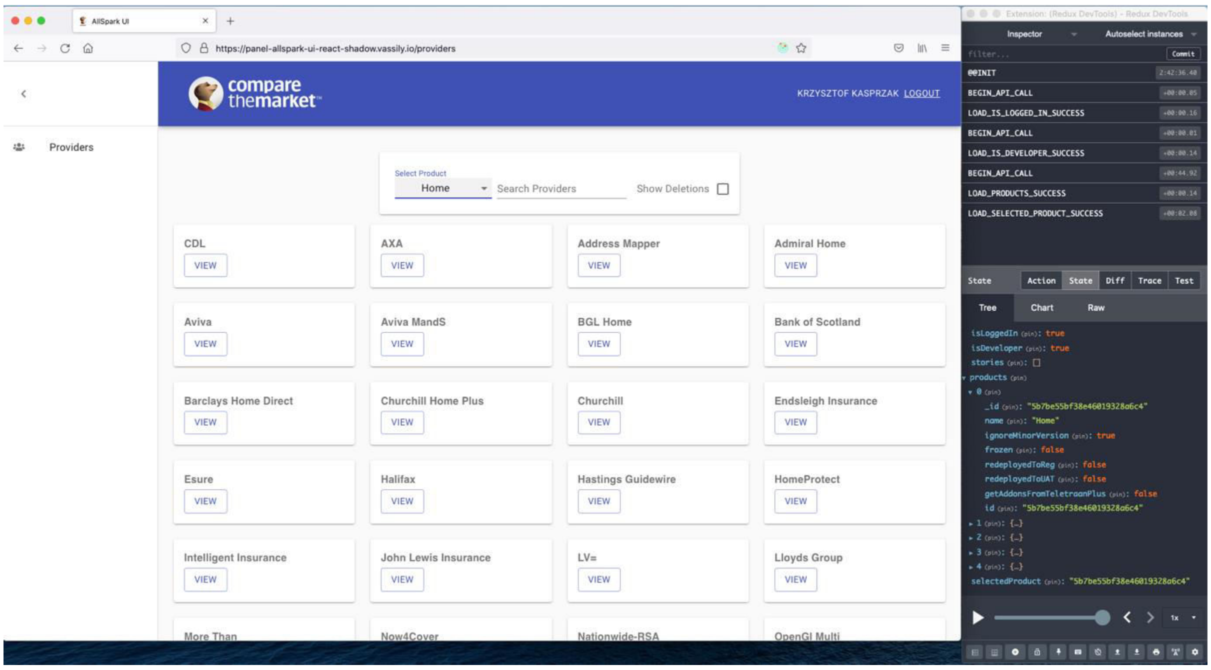
```
function mapStateToProps (state) {
  return {
    isDeveloper: state.isDeveloper
  }
}

export default connect(mapStateToProps)(ActionsPanel)
```

Fig. 2.7, code snippet that shows a typical way of connecting a React component to make it aware of specific global state store values. In this case, it is being made aware only of `isDeveloper` prop (with the usage of Redux utility function `mapStateToProps`). Then the `ActionPanel` component (which is a parent of download adapter button component) is being exported connected to given state, ready to be implicated in the main `app.js` file.

In this case, it was enough to connect the React component only to state storage props, however, when the need arises to make the component able to update the global state, then `mapDispatchToProps` utility function can be used. It works in the same way as `mapStateToProps`, but it returns Redux actions that can be invoked by the said component.

Redux state browser plug-in and its benefits

To increase my ability working with Redux, in my practice, I use Redux DevTools browser extension. This powerful browser extension enhances the browser's development tools by adding a specific suite of tools to track the state storage as the app is being used. It does not only display the actions dispatched, but also allows to track the state of the storage by displaying the current state, the changes to storage at each action dispatched and provides the function to playback/rewind actions' dispatches manually for better debugging/tracking functionality (see the fig. 2.8 and its description on the next page for an example of Redux DevTools usage).



```
@@INIT                              2:42:36.40
BEGIN_API_CALL                      +00:00.05
LOAD_IS_LOGGED_IN_SUCCESS           +00:00.16
BEGIN_API_CALL                      +00:00.01
LOAD_IS_DEVELOPER_SUCCESS           +00:00.14
BEGIN_API_CALL                      +00:44.92
LOAD_PRODUCTS_SUCCESS               +00:00.14
LOAD_SELECTED_PRODUCT_SUCCESS       +00:02.08
```

```
isLoggedIn (pin): true
isDeveloper (pin): true
stories (pin): □
▾ products (pin)
  ▾ 0 (pin)
      _id (pin): "5b7be55bf38e46019328a6c4"
      name (pin): "Home"
      ignoreMinorVersion (pin): true
      frozen (pin): false
      redeployedToReg (pin): false
      redeployedToUAT (pin): false
      getAddonsFromTeletraanPlus (pin): false
      id (pin): "5b7be55bf38e46019328a6c4"
  ▸ 1 (pin): {…}
  ▸ 2 (pin): {…}
  ▸ 3 (pin): {…}
  ▸ 4 (pin): {…}
selectedProduct (pin): "5b7be55bf38e46019328a6c4"
```

Fig. 2.8, the Redux DevTools extension picks up all the redux actions dispatched through the usage of app and displays them in chronological order (top list of the right-hand side browser extension). It also allows to track the latest storage state (bottom half of the browser extension). The storage state can be displayed in different ways, on the figure, it is being displayed in an easy to read, JSON-like format. The bottom holds an actions' player which allows to playback/rewind dispatches of actions. To display a change of state at each action, user needs to select dispatched action from the list and select `Diff` option form `State` toggle menu in the middle of the GUI of the extension.

In conclusion

Things I learned and improved upon during the work

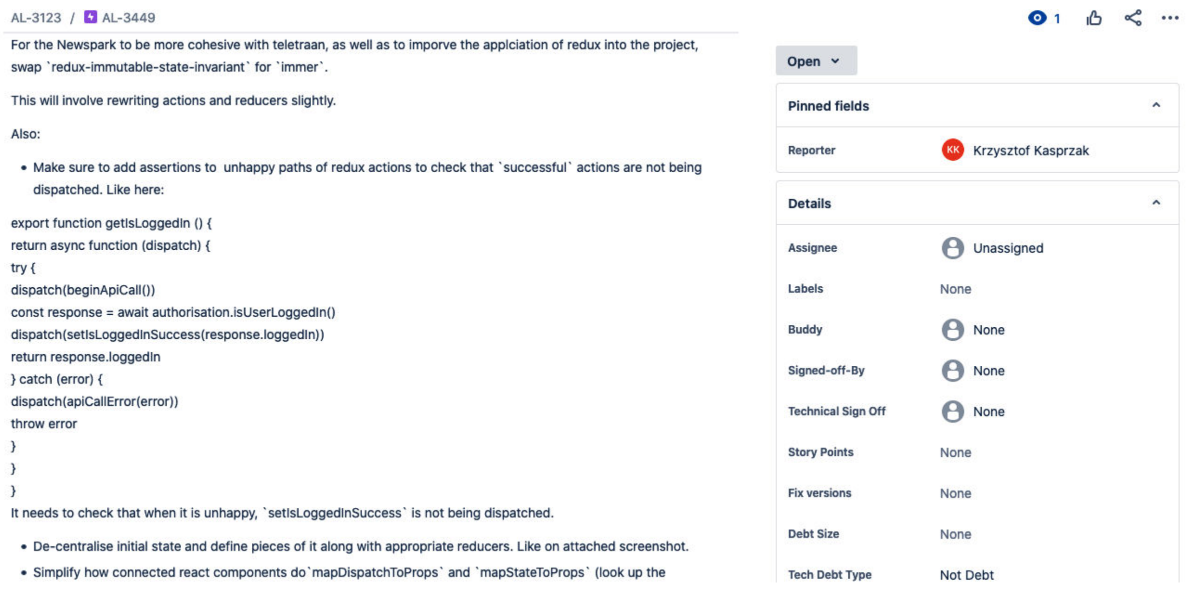
Since, it is best practice to include only essential data globally, I became better in deciding which pieces of data are necessary. It made me approach my development with more continues usage of least privilege principle when sharing data across components. I've also had to work with components that have their behaviour varied based on the state passed; I practiced and improved on applying behavioural state design.
Finally, by using Redux, I am continuously improving my ability to write code applying functional design.

Things I would do differently next time

While writing this very portfolio piece, I have thoroughly reflected on how I used Redux thus far. I realised that there are aspects of the code that can be improved. These are:

- Instead of using ` redux-immutable-state-invariant` package to keep the Redux state mutation free, I should swap it for `immer` more popular and robust library that serves the same purpose. This has been brought to my attention by having conversations with Teletraan team and also by taking a deeper dive into Redux style guide written by its creators.
- Instead of having a centralised initial state file, it actually is better to include pieces of initial state along with reducers. This way maintenance of reducers is easier, and it will also increase coherence of structure standard across Allspark and Teletraan.
- By having a code review session with Tim from Teletraan team, I realised that in my Redux actions' unhappy tests, I should also assert tests to make sure that the `successful` Redux actions are not being dispatched.

I have already taken an action to create a ticket to address those changes and will start to work on it when time will allow (see fig. 2.9 below for the ticket I created based on my practice reflections/code review with Tim).


AL-3123 / AL-3449

For the Newspark to be more cohesive with teletraan, as well as to imporve the applciation of redux into the project, swap `redux-immutable-state-invariant` for `immer`.

This will involve rewriting actions and reducers slightly.

Also:

- Make sure to add assertions to unhappy paths of redux actions to check that `successful` actions are not being dispatched. Like here:

```
export function getIsLoggedIn () {
return async function (dispatch) {
try {
dispatch(beginApiCall())
const response = await authorisation.isUserLoggedin()
dispatch(setIsLoggedInSuccess(response.loggedIn))
return response.loggedIn
} catch (error) {
dispatch(apiCallError(error))
throw error
}
}
}
```

It needs to check that when it is unhappy, `setIsLoggedInSuccess` is not being dispatched.

- De-centralise initial state and define pieces of it along with appropriate reducers. Like on attached screenshot.
- Simplify how connected react components do`mapDispatchToProps` and `mapStateToProps` (look up the

Open

Pinned fields

Reporter	Krzysztof Kasprzak

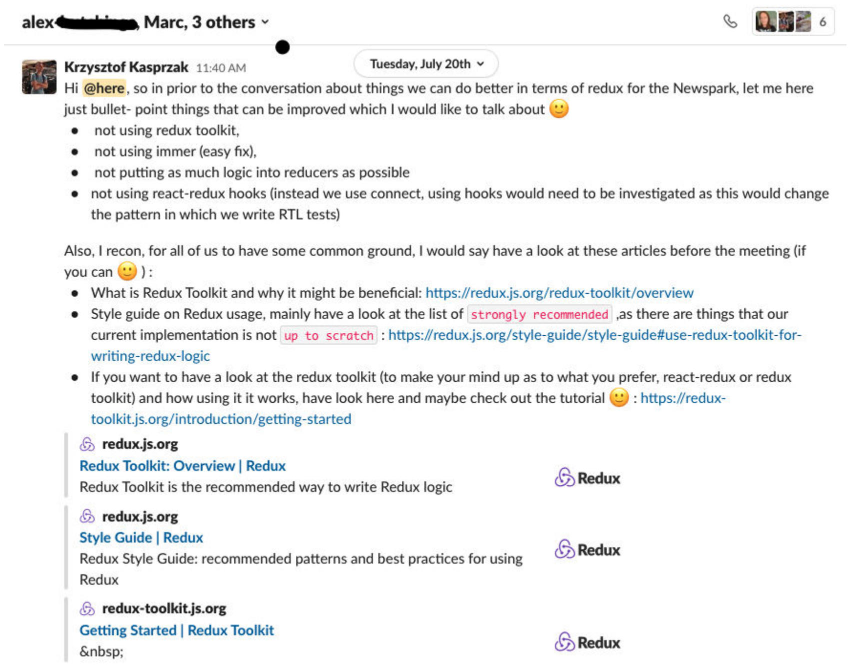
Details

Assignee	Unassigned
Labels	None
Buddy	None
Signed-off-By	None
Technical Sign Off	None
Story Points	None
Fix versions	None
Debt Size	None
Tech Debt Type	Not Debt

Fig. 2.9, screen capture of a ticket I created based on my reflections on my coding practice.

Value I brought and impact I made

Probably the biggest value I brought in by completing the ticket and consecutively writing up this piece was starting the cross-team conversation on generalising the standards of using Redux across Teletraan and Allspark projects (see fig. 3.1 below for an example of my cross-team communication on this topic). With that, I have also contributed to a better knowledge transfer between colleagues, which is specifically beneficial for my Allspark colleagues and I as it allows us to better ourselves with Redux usage.



alex[redacted], Marc, 3 others ⌄ 6

Tuesday, July 20th ⌄

Krzysztof Kasprzak 11:40 AM
Hi @here, so in prior to the conversation about things we can do better in terms of redux for the Newspark, let me here just bullet- point things that can be improved which I would like to talk about 🙂

- not using redux toolkit,
- not using immer (easy fix),
- not putting as much logic into reducers as possible
- not using react-redux hooks (instead we use connect, using hooks would need to be investigated as this would change the pattern in which we write RTL tests)

Also, I recon, for all of us to have some common ground, I would say have a look at these articles before the meeting (if you can 🙂) :

- What is Redux Toolkit and why it might be beneficial: https://redux.js.org/redux-toolkit/overview
- Style guide on Redux usage, mainly have a look at the list of `strongly recommended` ,as there are things that our current implementation is not `up to scratch` : https://redux.js.org/style-guide/style-guide#use-redux-toolkit-for-writing-redux-logic
- If you want to have a look at the redux toolkit (to make your mind up as to what you prefer, react-redux or redux toolkit) and how using it it works, have look here and maybe check out the tutorial 🙂 : https://redux-toolkit.js.org/introduction/getting-started

redux.js.org
Redux Toolkit: Overview | Redux
Redux Toolkit is the recommended way to write Redux logic

redux.js.org
Style Guide | Redux
Redux Style Guide: recommended patterns and best practices for using Redux

redux-toolkit.js.org
Getting Started | Redux Toolkit

Fig. 3.1, an example of how I took incitive to spark discussion about Redux and common standards across team.