Links

1. Completed app: https://autoura-san-b3e30.firebaseapp.com/
2. Repository: https://github.com/ralgr/a-better-autoura
3. Documentation on repository: https://github.com/ralgr/a-better-autoura/blob/master/documentation.md

Background

This project covers the creation of an interactive map that shows available locations around the United Kingdom related to food acquisition. The features included in the project are as follows:

- Interactive map - Functionality provided by Leaflet with map tiles coming from Thunderforest. The location data comes from the Autoura API, which contains various location data from different parts of the world. However this project will only make use of the location data related to places that vend food products around the United Kingdom.
- Authentication and Location Saving - Functionality made possible by Firebase as a complete back-end solution. Authentication encompasses account creation and account login, which is important for attributing the correct saved data to the correct user. Not having an account will not permit the user to make use of the saving feature, but will still be allowed access to the interactive map. In terms of user interface, some elements related to the saving feature will modify its visibility depending if a user is present.

This project and its features will be created using the Vue.js JavaScript framework as it simplifies the coding of complex features and decreases the time of development.

Methods

Planning and Prototyping using Adobe XD

Adobe XD was used to create a quick prototype for how the elements of the app will be laid out for different screen sizes. This step reduced the time spent on experimentations pertaining to the apps appearance, which resulted in having all the stated features above being polished to the point of working order at the least.

Protoype

Protoype

Protoype

Vue.js JavaScript Framework

As stated above, this project made use of the Vue.js framework. In terms of browser support, it is stated in the Vue GitHub Repository that Vue.js supports all browsers that are ES5-compliant. However, this means that users of Internet Explorer version 8 or lower will not be able to run the app.

Starting the development and creating the local folder for the app was done using the Vue CLI. This project makes use of manually selected features instead of the default settings to add more functionality to the app. In addition to Babel and Linter / Formatter defaults, the selected features are:

1. **Progressive Web App (PWA) support** - This option turns the app into a Progressive Web App without any further calibrations coming from developers. This is said to be achieved with the help of Workbox according to the Vue CLI PWA plugin repository. Being a PWA is ideal for any app that aims to provide a service uninterrupted by the unavailability of internet connectivity. As such, it is an obvious addition for this project as this app was intended to be an information provider for food

vendor locations. Losing an information source for locations as the network disappears would be a major inconvenience for any user.

2. **Vue Router** - This option allows the app to mimic the Multiple-page application (MPA) functionality of having numerous pages as a Single-page Application (SPA). However, navigating between pages in an SPA is done without page loading, unlike MPAs where it is normal behaviour.
3. **VueX** - This option allows the app to have a central store for data that can be used by any component regardless of their position in the app hierarchy, unlike when using only props and emits. Additionally, a development tool for major browsers that allows the developer to track and reverse the changes made to the store is available for use and makes it easy to troubleshoot problems regarding components making unintended changes to the store.

Vue.js Packages for Consuming Data and Displaying the Map

The following Vue.js packages were installed for the project to fulfil its intended purpose:

1. **Axios** - The location data for the app is taken from the Autoura API, as previously stated. To that end, Axios was installed to consume the API and make use of these location data.

```
// [[ AXIOS ACTIONS ]]
getStopsAction: ({ dispatch }, payload) => {
  // Get locations using Axios
  const autora = axios.create({ headers: { 'Authorization': 'Bearer ' + payload.key } })
  var url = `https://api.autoura.com/api/stops/search?group_context=${payload.context}&stop_types=food`
  autora.get(url)
  .then(r => {
    // Mapping new data on each result
    let response = r.data.response.map(response => {

      // Icon size data
      response.iconSize = payload.iconSize[0];
      // Z index data
      response.zIndex = payload.zIndex[0];
      // Highlight data
      response.isHighlighted = false;

      return response;
    })

    // Stops are sent to the store
    dispatch('setStopsAction', response)
  })
},
```

2. **Vue2Leaflet** - This package is used to display a map using the transport tiles from Thunderforest. The Leaflet marker feature from this package is what the app uses to display the locations on the map tiles. In addition, the popup feature allows the user to click these markers to display information on the clicked location.

```
<l-map :zoom="zoom"
       :center="center"
       @update:center="updateCenter"
       @update:zoom="updateZoom"
       style="z-index: 0"
       >
   <l-tile-layer :url="url"
                 :attribution="attribution"></l-tile-layer>
   <l-marker  v-for="(stop, index) in stopsGetter"
              :key="stop.stop_id"
              :lat-lng="createMarker(stop.location.geocode.lat, stop.location.geocode.lng)"
              :z-index-offset="stop.zIndex"
              @mouseover="toHighlight(index)"
              @mouseleave="toRemoveHighlight(index)">
      <l-popup>
        <span class="stop-visuals">
          <strong>{{ stop.name }}<br/></strong>
```

```
          </span>
          {{ stop.location.address }}<br/>
          <button type="button"
                  >More info</button>
        </l-popup>
        <l-icon :icon-size="stop.iconSize"
                :icon-url="icon">
        </l-icon>
    </l-marker>
</l-map>
```

Vuetify Material Design Component Framework

The Vuetify component framework was used in combination with Vue.js to create the user interface of the app. Additionally, Vuetify comes with its own grid system and display helpers that was used to the make sure that the app adheres to the responsive web design ethos in that, the app will adapt to the device that it is being viewed on. Additionally, this framework makes it easy to design a user interface suited for smaller devices by using the user interface components alongside the Material Design icons provided, which includes the mobile icon staples such as hamburger menus and directional chevrons among others.

Firebase Complete Back-end Solution

As mentioned, Firebase was used as complete back-end solution for the app. Among its array of different services, the app made use of its Firestore service for the storage of location data and its hosting service for when the app is deployed. The Firebase Firestore being a real-time database gives the app the advantage of instant updates to the app when data is added, modified, or removed without long and complicated code like what can be seen on AJAX based systems. Such an advantage is ideal for the saving feature of the app so that it can always immediately present up-to-date data to the user.

Additionally, the authentication feature available when integrating Firebase was also implemented. Authentication in this case, relates to the creation of accounts to do multiple things:

1. To **gain access** of the save feature.
2. As a **usability feature** to attribute the correct saved data to the correct user and only show these saved data.
3. As a **key** used in deciding whether to display or remove additional user interface elements pertaining to saving locations, which is a feature intended for users that are logged in only.

GitHub Workflow

This project also makes use of GitHub for version control. The ability to create branches separate from a main branch will enable the project to have a safe and efficient environment for experimentation without risk of breaking the application. The project workflow will be an iterative process that will mainly follow the pattern below for each feature to be included:

```
1. Create a experimentation branch.
2. Add and test experimentation code.
3. Does the experiment do what it is intended to do without breaking any elements?
    * If YES, proceed to step 4, otherwise go back to step 3.
4. Merge the experimentation branch to master then push to the repository.
```

This method ensures that there is always a working version of the app if an experimentation is a failure and that added features are thoroughly revised until they become functional without causing unintended consequences on other elements.

Manual Testing

Manual testing was also conducted throughout the development to ensure that each feature stated above functions as intended. This testing will generally follow the steps shown below:

```
1. Code a new feature.
2. List down the expected result.
4. Test the feature.
5. Does the feature function as stated in the expected results?
    * If YES, return to step 1, otherwise proceed to step 6.
6. Fix the feature then return to step 4.
```

Implementation (Technical)

Firebase Firestore Real-time Database

Firestore Rules

The Firestore database is set up so that only registered and logged in users are able to read and write data onto it by modifying the rules to check if they are authenticated as shown below.

```
service cloud.firestore {
  match /databases/{database}/documents {
    match /{document=**} {
      allow read, write: if request.auth.uid != null;
    }
  }
}
```

Saving Locations

Firestore is where the location data of any of the shown locations on the map that the user wants to save is stored. This is implemented in two steps:

1. To prevent data duplications, the app will first do a database call to check if the location that the user is trying to save is already present in the database. This check is done by performing compound queries for the location being saved and then using the `get()`
   method to look through the database once.
2. If the location being saved is present in the database, send out an error notification using the error alert component. Otherwise, proceed to save the data and send out a success notification using the success alert component. Before proceeding with the save, the location data, which is an object, is altered to include the username of the user trying to save the location for use in matching the correct location to the correct user later on as mentioned. Saving the data makes use of the `add()` method for convenience as it automatically creates a unique ID for the document.

```
// [[ FIRESTORE ACTIONS ]]
saveStopsAction: ({ getters, commit }) => {
  // Add user to the stop data for filtering on backend
  let stopToSave = getters.infoGetter
  stopToSave['user'] = getters.userGetter

  // Check for duplicates
  db.collection('saved-stops')
  .where("user", "==", `${getters.userGetter}`)
  .where("name", "==", `${getters.infoGetter.name}`)
  .get()
  .then(data => {
    // Add data if no duplicates
    if (data.empty == true) {
      db.collection('saved-stops')
      .add(stopToSave)
      .then(() => {
```

```
        // Clear previous alerts
        commit('clearSuccessAppAlert')

        // Stop loading animation
        commit('setAuthenticatingFalse')

        // Set alert message.
        const successMsg = {
          msg: 'Location saved',
          color: 'primary'
        }

        // Show alert Message
        commit('showSuccessAppAlert', successMsg)
      })
    }

    // Show alert if there are duplicates
    if (data.empty == false) {
      // Clear previous alerts
      commit('clearErrorAppAlert')

      // Stop loading animation
      commit('setAuthenticatingFalse')

      // Set alert message.
      const errorMsg = {
        msg: 'The selected location is already saved',
        color: 'error'
      }

      // Show alert Message
      commit('showErrorAppAlert', errorMsg)
    }
  })
},
```

Getting and Deleting the Saved Locations

At the same time, Firestore is also calibrated to provide the app w ith the information of the current contents stored in the database for a specific user. This information is show n in real-time using the `onSnapshot()` method w here inside, the location object is again modified to include its auto-generated unique ID given by Firebase before being set in the central store. This ID information w ill then be used as a reference to the saved data located in the Firestore that is to be deleted using the `delete()` method.

```
getSavesAction: ({commit, getters}) => {
  // Get saved locations from firebase
  db.collection('saved-stops')
  .where("user", "==", `${getters.userGetter}`)
  .onSnapshot(data => {
    let stopsFromFirebase = [];

    data.forEach(stop => {
      stopsFromFirebase.push({
        id: stop.id,
        ...stop.data()
      })
    })

    commit('getSaves', stopsFromFirebase)
  })
},
deleteSaveAction: ({commit}, payload) => {
  // Get saved locations from firebase
  db.collection("saved-stops").doc(payload).delete()
  .then(() => {
    // Set alert message.
    const successMsg = {
      msg: 'Location deleted',
```

```
      color: 'primary'
    }

    // Show alert Message
    commit('showSuccessAppAlert', successMsg)
    commit('openInfo')
  }).catch(error => {
    // Set alert message.
    const successMsg = {
      msg: error,
      color: 'error'
    }

    // Show alert Message
    commit('showSuccessAppAlert', successMsg)
  });
},
```

Mapping and Geolocation

The map component of the project made use of Leaflet markers and popups to create visual representations of the various location data taken from the Autoura API.

Leaflet Popup and Markers

The popups are implemented in a standard way in which it shows the location name and address for each location. On the other hand, this project modifies the default appearance of markers by importing `LIcon` from the Vue2Leaflet library in addition to popups and markers. The change is done by including a custom image on the project assets and is set in the `LIcon` component using props.

```
<l-icon :icon-size="stop.iconSize"
        :icon-url="icon">
</l-icon>
```

In addition to this, two features are implemented as a quality-of-life improvement for the user when using the app:

1. Feature to enlarge the marker upon hovering on a location listing.
2. Feature to highlight a listing when a marker is hovered.

This is implemented by adding custom data to the location object taken from the Autoura API in a process inside the Axios call. After the retrieval of the location object and before sending it to the central store, an array of custom values(normal and enlarged) for the z-index and icon size and a highlight data containing `false` is added using the JavaScript `.map()` method.

```
// Mapping new data on each result
let response = r.data.response.map(response => {

  // Icon size data
  response.iconSize = payload.iconSize[0];
  // Z index data
  response.zIndex = payload.zIndex[0];
  // Highlight data
  response.isHighlighted = false;

  return response;
```

The icon size and z-index will make use of the normal values initially and is changed to the large values upon hovering a location listing using methods. The highlight data is also changed to `true` upon hovering a marker to change the background colour of the listing related to it. Both features return to their normal values upon "un-hovering". The below code are Mutations that are

dispatched by Actions to follow the VueX coding best practice as stated on the tutorial by the Net Ninja.

```
// Icon enlarge for locations
biggify: (state, payload) => {
  state.stops[payload.index].iconSize = payload.iconSize;
  state.stops[payload.index].zIndex = payload.zIndex;
},
smallify: (state, payload) => {
  state.stops[payload.index].iconSize = payload.iconSize;
  state.stops[payload.index].zIndex = payload.zIndex;
},
// List highlight for locations
highlight: (state, payload) => {
  state.stops[payload].isHighlighted = true;
},
// List highlight for saves
unhighlight: (state, payload) => {
  state.stops[payload].isHighlighted = false;
},
```

Finding Selected Location on Map

Clicking an item in the location list shows an card element containing the specifics of the location built on the location data from Axios or the saved data from the Firestore. The card element offers multiple functions here depending on what list the user is on as can be seen below.

	Location List	Save List
Find location	✔	✔
Save location	✔	
Delete location		✔

The `Find location` function makes use of the latitude and longitude location data present in the location object to re-centre the map to show its exact coordinates as well as altering the zoom level. This is implemented by sending the latitude and longitude data along with a new zoom level in an object to be sent as a payload to be used by a receiving Action in the central store upon clicking the `Find location` button. The code below are the Mutations dispatched by a single Action.

```
findStop: (state, payload) => {
  state.latlng = payload
},

openInfo: state => {
  if (state.dialog == true) {
    state.dialog = false
  } else if (state.dialog == false) {
    state.dialog = true
  }
},
```

Progressive Web Application

As stated above, this app is a Progressive Web Application that is automatically set up using the Vue CLI. However, for the save data to be available offline from the Firestore database, an additional code to the Firebase configuration is necessary as shown below according to a tutorial by Hernández (2018).

```
db.enablePersistence({experimentalTabSynchronization:true})
```

Firebase Deployment

The app is currently deployed in Firebase and was done by following the Vue CLI tutorial on Firebase deployment. In summary, it required the installation of the Firebase CLI via `npm` and then initializing at the root directory of the app. This process is where the public directory is selected on which, it is set to be the `dist` folder as that is where the app will be stored upon creating a live build. The last important process is to confirm that the app is to be configured as a SPA. The rest of the process is set up with the defaults that Firebase has assigned. Afterwards, the app is then deployed using the code below with the option indicating that the app will only make use of the hosting feature for now.

```
firebase deploy --only hosting
```

Evaluation

Manual Testing

Test	Expected Results	Test results
Interactive Map (Leaflet Map)		
Leaflet marker appears on map for each location	Leaflet marker are visible	Leaflet marker are visible
Leaflet marker shows popup containing minimum details of the location when clicked	Leaflet marker shows popup on click	Leaflet marker shows popup on click
Hovering a marker highlights the location it is related to in the location list	Highlights the location it is related to in the location list when hovered	Highlights the location it is related to in the location list when hovered
Interactive Map (User Interface)		
Floating button opens and closes the UI element containing the location list	Floating button opens and closes UI element	Floating button opens and closes UI element
UI element containing the location list only shows locations when no user is logged in	Only shows location list when no user is logged in	Only shows location list when no user is logged in
UI element containing the location list shows tab changer buttons that changes what is viewed between locations available and saved locations when a user is logged in	Shows tab changer buttons when a user is logged in	Shows tab changer buttons when a user is logged in
Tab changer button for the location list tab shows the location list on click	Shows the location list tab on click	Shows the location list tab on click
Tab changer button for the saved locations list tab shows the saved locations list on click	Shows the saved locaiton list tab on click	Shows the saved locaiton list tab on click
Location list contains locations taken from the API	Location list contains locations	Location list contains locations

Saved location list contains no location when nothing is saved	Contains no location when nothing is saved	Contains no location when nothing is saved
Group context selector in the UI element changes the order of the list's locations depending on the user selection	Changes the order of the list's locations depending on the user selection	Changes the order of the list's locations depending on the user selection
Hovering a location on the location list enlarges the marker it is related to	Hovering enlarges the marker	Hovering enlarges the marker
Clicking a location on the location list opens a location card with more details about the selected location	Opens a location card	Opens a location card
Location card is responsive to the screen size	Is responsive	Is responsive
Location card contains different buttons depending if opened in the location list or the saved location list	Contains different buttons depending on where it was opened	Contains different buttons depending on where it was opened
Location card save button saves the location and shows in the saved locations list	Saves the location and shows in the saved locations list	Saves the location and shows in the saved locations list
Location card delete button deletes the location and is reflected in the saved locations list	Deletes the location and is reflected on the saved locations list	Deletes the location and is reflected on the saved locations list
Location card "find location" button re-centres the map to the selected location	Re-centres the map to the selected location	Re-centres the map to the selected location

Google Lighthouse Audit

Audit

As expected, the application scores well in the Best Practices, SEO and PWA areas of the audit as these are mostly automatically configured by the Vue.js framework. The use of Passive Listeners were recommended to further improve the score for Best Practices, while Meta Descriptions being unavailable prevents the SEO area from scoring higher. The Accessibility area is also relatively high but the score as is shown was due to unnamed buttons, which can result in screen readers just announcing the element as is without further context. Additionally, There were also problems with the contrast ratio of the background and foreground that may impact the visibility of elements. Finally, the Performance area where the app scored the lowest gave multiple reasons for the score with the most important being:

1. Due to large network payloads caused by making use of images in Cloudinary.
2. Due to not serving static assets attributed to both Cloudinary images and the map tiles from Thunderforest.
3. Due to the excessive DOM size. The recommended being ~ 1,500 DOM nodes, while the app has exactly 1,539 nodes.

Reflections

Below are the multiple most impactful problems and their solutions encountered during the creation of this project:

1. Saved location duplication - Initially, saving a location had no restrictions placed on it and a user could save the same location for as many times as they needed. However, having the same location saved more than once by accident and having to delete them would become tedious if done time and again and could potentially turn off the user from using the app. As such, the current system was put into place to check the database for the location being saved first before moving forward with

the saving process. With this in place, however, the app would then need an alert system to inform the user if the location is already saved or the location saving is a success.

2. Alert system - Initially, the alert system composed of only one component for informing the user of both success and failure of saving. This alert components takes an object from the central store containing the error message and the desired colour of the alert. The message object is created and passed to the central store inside Firebase Firestore `then` or `catch` calls. The problem with this system was that if there was a case where multiple alerts are sent one after the other, it will only show the latest one, which is not ideal for conveying the more important error alert information to the user. As such, the current system of having two separate alert components for error and success was put in place.

3. Route switching on login - Having transferred most of the database calls to the VueX central store, it was not possible to do a router push inside the `then` or `catch` calls as `$router` was not accessible. A workaround for this was done using JavaScript `promise` that resolves the VueX action for the database call, returning either `true` or `false` depending on how successful the task was, then pushing to the correct route inside a component's methods section.

4. Manual address entry allows the user to go to the Sign up and Sign in after logging in - This was solved by deploying navigation guards, the `beforeRouteEnter` specifically, on the said components. This would fist check if the a user is present in the central store before allowing the route change or denying it.