Keweenaw Time Travel Explorer Code Documentation

Robert Pastel, 9/10/2022

Application and User Description

The Keweenaw Time Travel Explorer is a web-based application which enables users to explore historical maps and records of the Keweenaw peninsula in Michigan. Users can perform database searches filtered by the categories "people", "places", and "stories", and by time regions between 1850 to the present. Historical records are located in both time and place. Keyword search results are displayed in a list and as icons on the map. The details of a specific database entry can be viewed by clicking on the list entry or the map icon. The historical interactive maps can be explored by selecting a time region in the timeline widget or by direct selection in the ESRI map widget.

The interactive design was targeted to the general public, but specific design efforts were made for users unfamiliar with GIS and efficient for archivists familiar with the app and GIS technology. Visiting the website, users are presented with a "landing" page that prompts them to enter a keyword search before viewing the map. In the map, a "I need help" button is prominently displayed which provides links to video tutorials. An archivist can explore the database and maps directly by clicking on the "Explore the map" button on the landing page.

Resources for Reading this Documentation

This document is intended for web app developers. It attempts to be an in depth description of the Keweenaw Time Travel (KeTT) Explore app implementation focusing on the frontend (user interface) implementation. As such the reader expected to be proficient in

- JavaScript (JS)
- CSS
- HTML
- PHP

MDN is the best site for learning and referencing details of the above technologies, https://developer.mozilla.org/en-US/

Development used Node, https://nodejs.org/en/, and npm https://docs.npmjs.com/.e

The app is implemented using the React frontend framework, the Redux state framework, and ESRI ArcGIS JS API. The application is large and complex, so this document cannot provide introduction to these tools and assumes the reader is familiar with these technologies. Good resources for learning about these technologies:

- React Tutorial, https://reactjs.org/tutorial/tutorial.html
- React Documentation, https://reactjs.org/docs/getting-started.html.
- The app uses React hooks, so read, https://reactjs.org/docs/hooks-intro.html
- Redux Tutorial, https://redux.js.org/tutorials/essentials/part-1-overview-concepts
- React-Redux Tutorial, https://react-redux.js.org/tutorials/quick-start
- The app uses Redux hooks, so read https://react-redux.js.org/api/hooks
- The code is constructed using Redux Toolkit (RTK), so read the usage guide, https://redux-toolkit.js.org/usage/usage-guide
- The app was initially constructed using the create-react-scrip, so you'll want to learn how to use the tool, https://create-react-app.dev/docs/getting-started
- The map uses ArcGIS API for JS, https://developers.arcgis.com/javascript/latest/
- You should read about the ArcGIS Map api, https://developers.arcgis.com/javascript/latest/maps-and-views/
- JQuery is also used to attach DOM listeners, https://api.jquery.com/

This document expects that the reader will follow the documentation with the app running and viewing the code.

The code is available at GitHub, https://github.com/Keweenaw-Time-Traveler/ktt-app. The app is live at https://kett.geospatialresearch.mtu.edu/, but you can and should build and run the app locally. You can clone the repository and build using npm or yarn. Instructions for building the app can be found on the GitHub repository. Note that I needed to use Node version 14.20 when using yarn version 3.2.2 to build.

Example User Interaction

So readers are familiar with the typical use of the app, below is a typical user interaction with the app and the app response. Later in this document, we will describe the details of the app response.

1. The user visits the Keweenaw Time Travel Explore site.
2. The app presents an animated landing page which terminates in a text input field. The user enters keywords in the text input field, e.g. "Peterson", and clicks the "search" icon.
3. The app then hides the landing page, makes a sequence of API and map tiles calls. After the response, the app displays the search results list and a grid representing icon density on the map.
4. The user can then filter the search results by year and category (people, place and story), or select directly from the search results list. Filtering the search by clicking on a category, e.g. people.
5. The app repeats the API calls and updates the list.
6. The user can select a search result by clicking on an entry in the search results.

7. The app will then make an API call, the timeline is updated showing the time region for the selection, the ESRI map makes tile calls and centers the map at the icon representing the entry, and the ESRI pop-up is shown.
8. The user can view the "Full details" by clicking the button or link in the pop-up.
9. The app makes API calls, the detail panel replaces the search list, centers the map again, and the ESRI map makes map tile calls.

App Architecture

Before diving into the details of app implementation and program flow, we should first study the app structure. This will facilitate navigating the app code and understanding the detailed implementation description.

The app is intended to be a React app using Redux for querying a geodatabase and navigating maps using the ArcGIS JavaScript API. The geodatabase API was hand built using PHP.

Major JavaScript Packages

The app implementation was initiated using create-react-app. Consequently the React packages are

- react, latest version, https://www.npmjs.com/package/react, for JSX expressions
- react-dom, latest version, https://www.npmjs.com/package/react-dom, for rendering on bowsers
- react-scripts, version 3.4.3, for running app during development. Uses webpack, https://webpack.js.org/ for packaging and development server.
- react-router-dom, major versions 5, https://v5.reactrouter.com/web/guides/quick-start, for URL page implementation with react.

Redux is used for state management, so major Redux packages are

- redux, major versions 4, https://www.npmjs.com/package/redux, for state management.
- react-redux, major versions 7, https://www.npmjs.com/package/react-redux, for redux integration with react.
- @reduxjs/toolkit, major versions1, https://www.npmjs.com/package/@reduxjs/toolkit, for development with redux
- @redux-devtools/extension, https://chrome.google.com/webstore/detail/redux-devtools/lmhkpmbekcpmknklioeibfkpmmfibljd?hl=en, browser extension for state management

The app uses ESRI ArcGIS so

- arcgis/core, major version 4, https://www.npmjs.com/package/@arcgis/core, ArcGIS JavaScript ES modules. <- I'm not sure used
- esri-loader, major version 3, for assistance loading AMD ArcGIS modulus

Other packages are used for API calls, styling and miscellaneous JavaScript coding

- axios, major versions 0, https://www.npmjs.com/package/axios, Promise based HTTP client for the browser and node.js
- node-sass, version 4.14.1, https://www.npmjs.com/package/node-sass, for importing and compiling sass to css
- breakpoint-sass, major versions 2, https://www.npmjs.com/package/breakpoint-sass, for writing media queries.
- jquery, major versions 3, https://www.npmjs.com/package/jquery

Directory Structure

At the top level the directory is divided into
- public/
- src/

The public/ directory contains index.html whose primary role is to provide the root div for the React app.

The src/ directory contains all the JavaScript code that will be processed by React Transform, bundled and loaded on the page by Webpack. The file src/index.js is the entry (starting point) for Webpack packaging.

The src directory structure mimics the old style directory structure for a React app, containing
- components/
- config/
- pages/
- redux/
- utill/

The file index.js in the src/ directory is the starting point and only wraps the App component with the react-redux Provider. The other JS file in src/ directory is App.js which defines the App component, and its role is to provide routing structure to React-Router.

The components/ directory contains all the React components except App.js which is in the src/ directory and the components used by react-router which are in the pages/ directory.

The pages/ directory contains all the top level React components used by react-router in App.js
- Login
- Main

- Signup

Only the Main component defined in src/pages/Main/index.js functions, and it is a wrapper for the Explorer React component.

The config/ contains data.js which are "constants" used by the app, specifically URL for the API and video tutorials.

I do not believe that the files in src/util are used by the app.

The redux directory contains store.js and the reducers/ subdirectory. The store.js just imports the slice reducers from the reducers/ subdirectory and then configures the reducer for the Redux store.

The reducers/ directory contains all the slice reducers:

- **dataSlice.js** - for storing and handling **FULL_DETAILS** data from PHP API POST calls to data URL. Defines getDetails thunk. The Redux Toolkit builder is used to construct the extra reducers, https://redux-toolkit.js.org/api/createAsyncThunk.
- **filtersSlice.js** - for storing and handling search data from PHP API GET calls. Defines Thunks get getPlaceName for PLACE_NAME.
- **historySlice.js** - for managing user search history. Stores the search keywords(?) in localStorage for persistent storage across sessions.
- **landingSlice.js** - for managing Landing page.
- **listSlice.js** - for storing, managing and displaying search results. Creates a thunk getList for LIST API.
- **mapSlice.js** - Manages displaying map and marks.
- **markersSlice.js** - stores markers
- **relatedSlice.js** - for storing and managing related data. Creates a thunk getRelated from RELATED_CONTENT data API.
- **submitSlice.js** - for managing and storing id for submitting a story**.**
- **timelineSlice.js** - for managing and storing the timeline widget. Creates a thunk getTimeline API POST call to DATA_PICKER.

All that is left is components/ directory. Its structure is

```
Explorer/
        Details/
                Data/
                Flag/
                images/
                Map/
                Related/
                        Data/
                        Group/
                Share/
                Source/
        List/
                Results/
                Section/
                        Item/

        Map/
                Chooser/
                Help/
                        Tabs/
                images/
                modules
                        mapPicker.js
        Submit/
                StoryForm/
FilterBar/
        Filters/
                RadioButton/
                Status/
                ToggleSwitch/
Landing/
        images/
        Intro/
                images/
        Options/
NavBar/
        hooks/
        Menu/
        Search/
        Timeline/
```

These are all the React components for searching the database and manipulating the map.The convention is that React components are defined in index.jsx files in the appropriate directory (with capital name). All index.jsx files have a default export. Typically, adjacent to the index.jsx file is a styles.scss for styling that particular component. The styles.scss and index.jsx are not listed above. Most styling is done through class selection in the scss.

This document will focus on these components.

At the top level, there are 4 major React Components

- Landing

- NavBar
- FilterBar
- Explorer

The Landing component controls the animation when the user first visits the Keweenaw Time Traveler (KeTT) Explorer app. The NavBar is the component at the top of the page. It contains a search field (Search), the timeline selector (Timeline) and a list of supplementary links (Menu). The FilterBar is just below the NavBar. It contains buttons for selecting categories for the search (Filter/RadioButton), filtering on stories with photos and showing map icons (Filters/ToggleSwitch) and a display of the current location time region (Status).

The Explorer is the major component of the app. It contains the search result list and the interactive components. It has major components

- Details
- List
- Map
- Submit

The List component contains the search results list. The Map component is the ESRI Map. The Details component responsible for displaying the "Full details" of a search item and will display over the list. The Submit component is a button on the map and handles submitting a new story to the database.

Example Program Flow

We will study the implementation of the KeTT Explorer app by following the program flow for the "Example User Interaction" described above.

The program flow is complex so to follow the flow you will need to use browser developer tools. You should watch the console, network panels and sometimes elements and React Components. In addition, you should install the Redux extension to watch the dispatching of actions and the change in state.

Note that for me to determine details of the program flow, I had to add console.log statements to the code. These log calls are prefixed with "RLP: ….".

User Visits KeTT Explorer

When the user first visits the website, they are presented with an animation that concludes with a text field for searching the geo-database.

The component App component controls the display of Landing component with the showLanding Redux selector and hiding the Navbar and FilterBar with the showMap selector.

The animation is controlled by the Landing component in an "initializing" useEffect with a sequence of setTimeout. Note I use the phase initializing useEffect for a useEffect with an empty dependency array, the useEffect second argument.

```
useEffect(() => {...}, [])
```

With an empty dependency array, the callback of the useEffect will run when the component is first mounted.

The Landing component uses React state variables, useState hook, to store and set the props for lower level components, Intro and Options. The Intro component is the banner that first shows announcing the purpose of the app. The Options component shows the input text field for a search and the "Explore the map" button.

Although the Map, and Timeline (in the Navbar component) are hidden from the view, they are mounted, so their initializing useEffects run during the animation. The Timeline initializing useEffect dispatches the timeline/getTimeline which is an async Thunk making the DATE_PICKER (https://geospatialresearch.mtu.edu/map_picker.php) API call which gets the values for setting up the timeline.

The Map initializing useEffect is very large, but it contains

```
watchUtils.whenTrue(view, 'stationary', function () {
…
}
```

The watchUtils.whenTrue (https://developers.arcgis.com/javascript/latest/api-reference/esri-core-watchUtils.html#whenTrue) will run when "view" (a MapView) is created because the view is stationary. The callback dispatches the filters/getPlaceName thunk making the PLACE_NAME (https://portal1-geo.sabu.mtu.edu/server/rest/services/KeweenawHSDI/cchsdi_placenames/FeatureServer/0/query) API call.

Also in the Map initializing useEffect after the view is created the function updateGrid is called

```
updateGrid(view, startingFilters);
```

The function updateGrid makes four asyncGrid calls corresponding to the different zoom level for the grids, eg:

```
asyncGrid(view, filters, '6').then((res) => {
```

The function asyncGrid

```
const asyncGrid = (view, filters, size) => { … }
```

makes the GRID (https://geospatialresearch.mtu.edu/grid.php) API call which sets up the grids for showing the icon density on the map. NOTE that grids are not controlled by Redux.

User Enters Search

As the user enters search the keywords, eg "Peterson" the onChange handler for the text input field dispatches filter.updateSearch actions which sets the state.filter.search to the value in the text field.

After the user clicks the search button, the handleSearchClick is called which just prevents the default action of the form and then calls the function without argument. The search function:

```
 const search = (hideList) => {
   dispatch(updateLandingView({ show: false, remove: true }));
   dispatch(updateMapView(true));
   dispatch(turnOnToolTips(false));
   if (!hideList) {
     dispatch(toggleList('show'));
   }
 }
```

Which dispatches landing/updateLandingView action that sets state.landing.showLanding to false. The App component showLanding selector then passes it back to the Landing component show prop which triggers the props useEffect and consequently hides the Landing component by setting the React useState and changing the class for the landing div.

Then the search function then dispatches the map/updateMapView which sets state.map.showMap to true. The App component then uses the selectShowMap selector to pass the value on to the show props for the Navbar and FilterBar, so they are visible.

The landing/turnOnTopTips action is dispatched. Although it updates the landing slice reducer, it is not used by any other components.

Finally the search function dispatches the list/toggleList with payload "show". The list slice sets the state.list.showList to true and the state.list.removeList to false. The Explorer (in Explorer/index.js and also called the ExplorerApp) uses the selectShowList to pass on to the List component show prop which is used by the List component to change the div class.

Because listOpen the selector value for selectShowList selector changes to true, the dependent useEffect runs. Because state.list.status initial value is "idle", the useEffect dispatches the list.getList thunk which makes the LIST (https://geospatialresearch.mtu.edu/list.php) API call. The successful response gives the list of search results in the JSON. The list slice reducer sets the state.list.entities to the search results. Note that entities is a JavaScript object with categories people, places, and stories.

The list variable of the List component is set by the selectAllList useSelector for state.list.entities. The List component then maps the people, places and stories component of the list to Section components' list prop.

Via the props, the Section components set up the title of the categories and map the categorical search results to Item components via the list prop. The data prop of Item holds the content of the search entity. The Item component sets up the div for the search result entity and also sets up the ToolTip (react-tooltip-lite) component for the entity.

The Map component set a DOM listener with

```
  $('body').on('click', '#intro-options-search-icon', function (e) {
              e.preventDefault();
              landingSearch();
    });
```

The Map's landingSearch function gets the search value from the DOM

```
const searchValue = $('#search-landing').val();
```

In Landing/Options/index.js, the input element is

```
 <input
       type="text"
       id="search-landing"
       name="search"
       placeholder="Search People, Places and Stories"
       value={filters.search}
       onKeyDown={handleKeyDown}
       onChange={(e) => dispatch(updateSearch(e.target.value))}
/>
```

It would have been more appropriate to get the value from the Redux store.

The landingSearch then centers the map view to a generic point (approximate center of the Keweenaw Peninsula) and then calls updateGrid. The updateGrid function calls asyncGrid for 4 zoom levels. Each call makes a GRID API call.

Filter Search Results

The user can filter the result by clicking the "people" radio button in the FilterBar/Filters component. The RadioButton component is primarily an radio input HTML element and a label HTML element. The styling in the adjacent styles.scss effectively hides the standard input element by specifying its height and width to be 1 px. Consequently, all the styling is done on the label element which contains a FontAwesomeIcon.

The Filters component passes the handleRadioChange function as the "changed" props to the RadioButton component. The changed prop is the onChange handler for the input element. The handleRadioChange first sets the Filters component's state variable "type" to the RadioButton value. The type component state is used in the "isSelected" RadioButton prop. Then the handler dispatches the filters/updateType action which sets the state.filters.type state variable to the selected category, in this case "people". Then the handler dispatches the list/getList thunk action which makes the LIST (https://geospatialresearch.mtu.edu/list.php) API call to get a new search result list filtered by the people category.

The Map component has the DOM listener

```
 $('.radio-button-input').on('click', function () {
           const type = $(this).val();
           typeRef.current = `${type}`;
           …
}
```

which then updates filterVal to the type for updating the grid. The updateGrid then makes 4 asyncGrid calls. Each asyncGrid makes a GRID API call for a zoom level.

The List component has a useEffect dependent on listStatus.

```
  useEffect(() => {
    if (listStatus === 'idle') {
      console.log("RLP: List useEffect")
      dispatch(getList({}));
    }
    //dispatch(getList({}));
  }, [listStatus, listOpen, dispatch]);
```

The component's listStatus is the value from the selectListStatus selector for state.list.listStatus. When the handleRadioChange dispatches the list/getList thunk action, the thunk changes the list.listStatus from "success" to "idle". Because the listStatus has changed the dependent useEffect is called, and because listStatus is now "idle", it dispatches the list/getList thunk which makes a second LIST API call. **This is a bug.** The second LIST call is not necessary.

Fortunately the second list/getList thunk does not change the state.list.listStatus or the app would be an endless loop.

This bug is due to the misuse of the useEffect dependency. Although the useEffect should be dependent on listOpen so it can dispatch list/getList after the landing page search, we do not want the useEffect to be called when the listStatus changes. In general, programmers should be careful when having a component dispatches an action that changes itself. It is better for another (generally higher level) component to dispatch the action. An exception is the initializing useEffect because the role of the effect is clear and will only be called once with the component mounts.

Select Search Entity

The Map component has a DOM listener when the user clicks an item (eg "Wm Peterson, 1917 Home") in the search result list.

```
$('.page-content').on('click', '.list-results-item', function () {
      listClick($(this));
  });
```

The function listClick(item) parses the data from the item and calls updateTimeline with mapyear (="1914-1924").

The function updateTimeline the uses the DOM to parse through the segments in the timeline

```
$('#date-range .segment').each(function () {
    …
    const min = $(this).data('min');
    const max = $(this).data('max');
     …
     if (mapyear >= min && mapyear <= max) {
       dispatches
            updateTimelineRange(`${min}-${max}')
            updateActiveSegment(`${id}`)
            updateActiveUrl(url)
            updateLeftPip(left)
            updateRightPip(right)
            updateDateRange(`${min}-${max}`)
            updateStartDate(`${min}`)
            updateEndDate(`${max}')
            updateReset(true)
          }
 });
```

The dispatches control the timeline appearance and the filter state variables, startDate and endDate.

In addition updateTimeline, calls

```
createTileLayer({ url, zoom: false })
```

The createTileLayer function creates an ESRI TitleLayer from the url of a historical map. It then adds the TileLayer to the map.

After calling updateTimeline, listClick then calls

```
gotoMarker(point, itemId, recnumber, markerid, loctype, type);
```

Note that the point parameter above is given x and y properties from the item's data.

The function **gotoMarker sets the active** references used by the map and then moves the map view to the point and opens a popup without content. It calls

```
asyncMarkerInfo(recnumber, markerid, loctype, type, filterVal).then(
    function (res) {
        view.popup.title = res.title;
        view.popup.content = res.body;
   }
);
```

To get the content for the popup.

The function asyncMarkerInfo sets up and makes the MARKER_INFO (https://geospatialresearch.mtu.edu/marker_info.php) API call. After a successful response, asyncMarkerInfo then creates the html for the content.

The watchUtils.whenTrue callback function runs when the map view is stationary. It dispatches the filter/getPlaceName thunk. It then calls asyncMarkers which calls the MARKERS (https://geospatialresearch.mtu.edu/markers.php) API call.

The onClick handler for the Item component, handleListItemClick, dispatches updateListItem (with empty strings payload), details/toggleDetails and related/toggleRelated actions with "hide" payload. It is not clear what the role of listItemId role is, but the role of the details/toggleDetails and related/toggleRelated actions hide the Details and Related components.

This program flow is very difficult to read. The cause of the difficulty is that the Map component uses a DOM listener to trigger an event to respond instead of dispatching an action to the Redux store. In addition the Map inspects the DOM to get state values rather than from the Redux store. This is contrary to a React Redux app. See the "Integrating ArcGIS with React and Redux" section for alternative program flow.

View Full Detail

The user can click the "Full details" button in the popup to view the details.

The map view registers a DOM listener to the full-details div

```
  $('.page-content').on('click', '.full-details', function () {...}
```

The listener inspects the DOM

```
const $active = $(this).closest('.map-popup');
```

to find the id and recnumber for search entry

```
const id = $active.find('li.active span.id').text();
const recnumber = $active.find('li.active span.recnumber').text();
```

and then calls

```
loadDetails(id, recnumber, loctype, title);
```

The loadDetails function dispatches

```
    updateListItem({ recnumber, loctype })
    toggleList('hide'))
    getDetails({ id, recnumber, loctype })
    toggleDetails('show'))
    toggleSubmit('hide'))
    toggleRelated('hide'))
    setActiveTab(''))
    updateSearch(title)
    getList({})
```

The list/updateListItems action sets the state.list.listItemId to the recnumber (= '14020|CD'). The details/getDetails thunk which calls the FULL_DETAILS ('https://geospatialresearch.mtu.edu/full_details.php) API call which responds with the content for the details panel. The fulfilled action processes the response and stores some of it in the DOM and some in the state.

```
  $('.page-content').on( 'click', '.map-popup.marker .data li', function () {
```

The filter/updateSearch sets the search result to the item selected for full detail. The list/getList thunk makes the LIST API call. The response returns to a single entry for the list.

The Detail component's useEffect runs when the status (from the selectDetailStatus selector) changes to "success". The useEffect dispatches the history/updateHistoryItems action. It stores the selected entry in state.history.historyItems and then stores the list in localStorage with key "kett_history" so it can be referenced following visits to the website.

The Related useEffects runs because the details/getDetails/fulfilled action changes the state.detial.id to '14020|CD'. The useEffect dispatches the related/getRelated thunk which makes the RELATED_CONTENT (https://geospatialresearch.mtu.edu/related_content.php) API call. The fulfill action of the thunk sets the state.related.people, state.related.places and state.related.stories state variables. By useSelectors these are passed to the data prop of The Data component.

Again this specific program flow mixes DOM listener callbacks and dispatch to the Redux reducers.

Integrating ArcGIS with React and Redux

Following the program flow is difficult because of the integration of ArcGIS with a React Redux app. The program mixes storing and querying the app state from the Redux store and the DOM. The program also responds to user interactions from either callbacks from DOM listeners or the Redux reducers. We should study alternative techniques for integrating ArcGIS with React and Redux.

There are two major challenges integrating Esri ArcGIS with React and Redux. One challenge is that traditionally web apps using ArcGIS used AMD modules while React apps used ES modules. The other major challenge is relaying changes in the React components to the Esri MapView.

Combining AMD and ES modules

The AMD (Asynchronous Module Definition) format uses a "require" statement and a third party loader to package the modules. ES modules (ECMAScript Modules) use an "import" statement. In 2016, ES modules became the de facto official standard for modularizing JavaScript, and all modern browsers implement ES modules. Obviously the AMD and ES modulus are not compatible. There are two solutions. One solution is to convert and package the ES modulus into AMD format and then use the ArcGIS modulus. This approach is demonstrated by Michael Van den Bergh

https://www.esri.com/arcgis-blog/products/3d-gis/3d-gis/react-redux-building-modern-web-apps-with-the-arcgis-js-api/

in his ArcGIS JS API 4.x / React / Redux Boilerplate example

https://github.com/Esri/react-redux-js4

Webpack and Babel package the React components into a library using AMD format

https://github.com/Esri/react-redux-js4/blob/master/webpack.config.babel.js

Then a Dojo configuration file specifies the library as dependency

https://github.com/Esri/react-redux-js4/blob/master/dojoConfig.js

And ArcGIS is loaded on the webpage by CDN

https://github.com/Esri/react-redux-js4/blob/master/index.html

In the program, the Esri ArcGIS modules are accessed using the "import" statement the Map or SceneView can be attached to the DOM element on the page.

https://github.com/Esri/react-redux-js4/blob/master/src/middleware/arcgis-sceneview.js

This is an elegant solution which makes the discrepancy between AMD and ES modulus invisible to the programmer. It has the disadvantage that webpack needs to be configured.

Developers using create-react-app

https://create-react-app.dev/

typically do not have access to webpack. They must eject the scripts configuring the build.

https://create-react-app.dev/docs/available-scripts#npm-run-eject

which is not something most developers would like to do.

The second solution is to load ArcGIS AMD modules into React ES modulus. Esri developed the esr-loader to do this.

https://github.com/Esri/esri-loader

The syntax and use of loadModules of esri-loader is very similar to the Dojo require that traditionally were used to build ArcGIS. In essence, the React Map component is a wrapper for the Esri Map module. Ahmed ElHussiny and Ryan Watkins introduced an ArcGIS app built on create-react-app https://mediaspace.esri.com/media/t/1_y9l5g3c9

A bootstrap app is available

https://github.com/Esri/esri-react-boot

The Keweenaw Time Traveler Explorer app uses this approach.

Currently, there is now a third choice. In 2020, Esri distributed their modulus via npm.

https://www.npmjs.com/package/@arcgis/core

So developers can directly import ArcGIS modulus into two of their project builds. This approach is the preferred technique.

Relaying User Interactions in React Components to the ESRI Map

React Redux Introduction

Before explaining the challenge of relaying user interactions in react components to the Esri Map, I will briefly introduce React and Redux. React is a popular frontend framework for developing single page applications. The main concept of React is that the DOM is hierarchically composed of React components using JSX.

https://reactjs.org/docs/introducing-jsx.html

React components can have state.

https://reactjs.org/docs/state-and-lifecycle.html

which can change and be passed as props (properties) to lower level components.

https://reactjs.org/docs/components-and-props.html

React will selectively rerender components when their props change. The notation is that the app is defined by its state and changes in state implies changes to the UI. So control of the UI becomes state management. The hierarchical structure of React components makes propagation of state down to subcomponent changes relatively easy to implement, but React does not offer any other tools for implementing and managing state.

Redux is a popular state management tool.

https://redux.js.org/

The main concept of the redux is that the app state is kept in a single store. The state should only change when actions are dispatched to reducers. In addition the reducers must be purer functions, meaning they do not produce side-effects. Redux is integrated with React using react-redux

https://react-redux.js.org/

Older applications used react-redux "context" function with functional arguments that mapStateToProps and mapDispatchToProps.

https://react-redux.js.org/using-react-redux/connect-mapstate
https://react-redux.js.org/using-react-redux/connect-mapdispatch

Modern react-redux apps use hooks.

https://react-redux.js.org/api/hooks

The Challenge

Using Redux requires adhering to three principles

https://redux.js.org/understanding/thinking-in-redux/three-principles

- There is a single source of truth, the store.
- State is read only, dispatch actions to change the state.
- Changes are made with pure functions, i.e. reducers are pure functions.

Pure reducers are important because dispatched actions should immediately and accurately change the app's state, so that React components can be updated. A common operation of web apps is to fetch data for the view from the server. The fetch is a side effect and in particular takes time, so one of two things happens when a reducer does the fetch. Either the state updates but the React components do not have data for the view or the app becomes unresponsive. The restriction that reducers are pure is a challenge using ArcGIS because almost any user interaction implies a side effect, a fetch to the map server.

There are two potential solutions to challenges using ArcGIS with React-Redux app:

- Use Redux Middleware
- Bypass Redux when Manipulating the Map

Redux Middleware

Using Redux Middleware is the preferred approach because the technique mimics the React-Redux architecture and keeps the state accurate. Also Developers know exactly how the actions are processed.

https://redux.js.org/understanding/thinking-in-redux/glossary#middleware

In essence a middleware is a layer around the reducer. Actions that invoke the middleware are called thunks.

https://redux.js.org/usage/writing-logic-thunks#thunk-overview

The thunk can perform operations and dispatch actions. Using the fetch as an example,

https://redux.js.org/tutorials/essentials/part-5-async-logic

the thunk for the fetch would first dispatch a "pending" action and make the fetch call. On successful receipt of the response, the thunk would then dispatch a "fulfilled" action.

ArcGIS JS API 4.x / React / Redux Boilerplate example

Michael Van den Bergh in his "ArcGIS JS API 4.x / React / Redux Boilerplate" example uses middleware to integrate the Map and SceneView.

https://www.esri.com/arcgis-blog/products/3d-gis/3d-gis/react-redux-building-modern-web-apps-with-the-arcgis-js-api/

https://github.com/Esri/react-redux-js4

The app code implements two Redux middlewares

https://github.com/Esri/react-redux-js4/tree/master/src/middleware

- arcgis-authentication
- arcgis-sceneview

Note that middlewares are integrated with the store in store.js.

https://github.com/Esri/react-redux-js4/blob/master/src/store/store.jsx

This is standard coding for adding middlewares to the store and not unique to an ArcGIS app.

We will study the arcgis-sceneview in more detail because it has subtitles for integrating the Esri Map or SceneView into a React-Redux app.

The WebSceneView (SceneView) is added to the view in app.js.

https://github.com/Esri/react-redux-js4/blob/master/src/components/app.jsx

This is also a typical React composition, but notice that the TimeNav and ShadowNav are third party React components and are mounted adjacent to the WebSceneView.

The WebSceneView component is defined in

https://github.com/Esri/react-redux-js4/blob/master/src/components/webscene.jsx

It renders only a div which is referenced. The div is where the SceneView will render, but it is not clear how. Note that the init prop is mapped to the initScene action creator, and the lifecycle method componentDidMount sets the init prop. So when the component is mounted, the initScene is dispatched with a reference to the component. In modern redux technique this convolution would be replaced with code in an initializing useEffect Redux hook.

The initScene action creator makes an action with type INIT_SCENE.

https://github.com/Esri/react-redux-js4/blob/master/src/reducer/webscene/actions.js

The arcgis-sceneview middleware intercept actions with type INIT_SCENE.

https://github.com/Esri/react-redux-js4/blob/master/src/middleware/arcgis-sceneview.js

Before switching on any types the middleware makes a global variable, arcgis

```
const arcgis = {};
window.arcgis = arcgis;
```

Then for actions with type INIT_SCENE, the middleware

1. Ensures that SceneView is a singleton
2. Stores a reference to the SceneView arcgis.scene. Note SceneView is a promise.
3. In a "then" clause, adds layers to the SceneView and dispatches the action
4. In a sequential "then" clause, makes initial setting for the other components and dispatches actions to update the components.
5. Finally returns the SceneView

It is important that a reference to the SceneView is global so that other thunks can reference the SceneView. See for example the arcgis-sceneview thunks for the action types SELECTION_SET, SELECTION_ADD, SELECTION_REMOVE, and SELECTION_RESET. It dispatches the action, next(action), and then calls updateHighlighs with arcgis.sceneView.

https://github.com/Esri/react-redux-js4/blob/master/src/middleware/arcgis-sceneview/highlights.js

esri-react-boot example

The esri-react-boot example uses a slightly different approach.

https://mediaspace.esri.com/media/t/1_y9l5g3c9
https://github.com/Esri/esri-react-boot

Rather than implementing the middleware directly, the bootstrap uses Redux-Saga.

https://github.com/redux-saga/redux-saga

Redux-Saga is a declarative framework for making Redux middleware.

If you are using the Redux Toolkit

https://redux-toolkit.js.org/

then you could use the createAsyncThunk to the thunks.

https://redux-toolkit.js.org/api/createAsyncThunk

In my opinion this might be a more clumsier approach than writing the middleware directly.

Bypassing Redux when Manipulating the Map

Another technique for handling map side effects is not to use Redux when manipulating the SceneView. There are several ways to do this:

- React components that manipulate the MapView have references to the MapView and use it to manipulate the view and then dispatch the action.
- The MapView component listens to events and manipulates the view.

Because most ArcGIS objects are created with promises, it is possible for the React component to use a global reference to the view and in the "then" or "when'" clauses dispatch the action to the Redux store. This technique directly associates the React component with the map manipulations. It has the disadvantage that it scatters the logic across multiple components. Redux best practice is that most of the business logic is in reducer or middleware.

The Keweenaw Time Traveler Explorer app uses the second approach of having the MapView component listen to events. There are several ways to create events and attach listeners. The React UI subcomponent can make and emit an event then the MapView component can listen and respond to the event. Another technique is to attach DOM listeners in the MapView then when the DOM changes the MapView can respond. Both the techniques do not require a global MapView reference, but the MapView becomes blotted with nearly all the business logic. Attaching DOM listeners also creates a challenge because it requires the developer to interpret the DOM selector and remember the naming of the DOM elements. Consequently, the code is very difficult to read and maintain. Another issue with using DOM listeners is that the app is using two sources of truth, the redux store and the DOM. When using the DOM listeners techniques, the developers must agree that their intent is that React-Redux will only be used to control the UI React components, and the MapView component will use the ArcGIS framework to control the map view. Not adhering to this rule will lead to bugs in the code and synchronization problems. One thing that should never be done is to use the DOM listeners to dispatch actions to the Redux store. This inverts the intent of the React-Redux architecture.

Component Descriptions

Pages

- Landing page is shown on initial navigation to the website
- Although React Router is used, coded in App.js with the main page, /, also /login and /signup, only the main page is implemented.

Landing Page

Upon visiting KeTT, App.js is loaded and the Landing page is shown because landing/showLanding is true. NavBar and FilterBar are not shown because map/showMap is false. These are the initial state of the state parameters.

App.js also specifies the pages using react-router-dom Router and Switch, but other than the main page these are not really implemented.

The Landing page is defined in src/Landing/index.jsx. The major components are:

- images (the background image of the Keweenaw, src/Landing/images/landing_bg.jpg, and a mini map, src/Landing/images/mini-map.jpg
- Intro which is a simple html components of the KeTT logo and text.
- Options (defined in Options/index.jsx with default export Search).

First the background image and Intro is shown 0.83 sec and then faded to show Options and the mini map in react.useEffect using setTimeouts and css fading.

The main divs of Options are the search input (intro-options-search-input) and a button, (Explore the map). Keystrokes in the are stored and updated by the redux action filterSlice.updateSearch. Keystrokes are first processed by handleKeyDown which watches for an 'Enter' keystroke which will call search(). Otherwise the input value is dispatched using filterSlice.updateSearch. The combined process initiates a search without the 'Enter'.

The function search, hides the landing view by distaching landingSlice.updateLandingView and showing the map by dispatching the mapSlice.updateMapView, and finally show the list of search results by dispatching listSlice.toggleList.

The handler for clicks on the "Explore the Map" button, handleExploreClick, hides the landing page by dispatching landingSlice.updateLandingView, listSlice.toggleList and mapSlice.updateMapView. In other words, it is the same as the search option without a search result list.

Main Page

The react component App.js provides a NavBar and FilterBar, but are hidden during the landing phase.They are shown when the user enters a search or clicks to explore the map.

NavBar

The NavBar is composed of Search (src/NavBar/Search/index.jsx), Timeline (src/NavBar/Timeline/index.jsx) and Menu (src/NavBar/Menu/index.jsx).

The mechanics of the Search component is very similar to the mechanics of the search in the landing page. It does have an additional feature. A history list of searches which is contained in a <nav> element. The visibility of the history list is controlled by css and handleBackClick which is the onClick callback for the search-nav containing a Font Awesome Icon faChevronLeft. The div history-list maps the historyList to the item list. The component also controls the visibility of the search result list by dispatching listSlice.toggleList action. The list is visible after the user starts a search by keying enter or clicking the search icon. The list is clear and hidden when the search text is empty or the user clicks the search-clear-icon div.

Timeline

The Timeline component is a very interesting widget. It is a user interface for users to choose dates for maps and search results. It is graphically represented by sequential line segments. The line segment lengths are retrieved by dispatching a timelineSlice.getTimeline thunk POST API call to DATE_PICKER in an initial useEffect. The fulfilled action of the thunk constructs the segment relative length by calculation on the date range for the maps. User interaction of the timeline is by clicking on the line segments. The callback for the segment divs, handleSegmentClick, gathers the date range from attributes of the click target and then dispatches the timelineSlice actions

- updateActiveSegment
- updateLeftPip
- uidateRightPip
- updateDateRange
- updateStartRange
- updateEndDate

- updateReset to true

If the list is not removed then a new search result list is generated by dispatching the thunk listSlice.getList.

The timeline also makes use of the Tooltip component from react-tooltip-lite, https://www.npmjs.com/package/react-tooltip-lite. The tooltips are mapped adjacent to the segment div. They display when the user hovers over the segments showing the date range for the segment.

If a date range is selected by the user then a button is displayed containing a fontawesome faclock. The handler, handleResetClick, resets the date range to 1880 to 2022 and clears the previously selected date range.

The Menu component, defined in src/component/Menu/index.jsx, is a simple list of links (anchor tags) to a variety of resources pages.

FilterBar

The FilterBar is a wrapper div for the Filters component (src/components/FilterBar/Filters/) to control and hide the fiters.

Filters

The Filters component, defined in src/components/FilterBar/Filters/index.jsx, is a user widget for filtering search results. It is composed of two parts. Four RadioButtons filter the results for 'everything', 'people', 'places' or 'stories'. The filtering is identified by the "id", "isSelected", and "value" RadioButton props. The handling of the user selecting the filter is defined by the handleRadioChange function which uses the target value to which first dispatch the filterSlice.updateType action using the target value and then dispatches the filterSlice.getList action.

The other part of Filters is two ToggleSwitch components. One ToggleSwitch filters search results to results with photos, and the other ToggleSwitch hides or shows the map icons. The ToggleSwitch props "id" and "checked" identify the ToggleSwitch role. Response to the user toggling switches is handled by the handleChange function which uses the ToggleSwitch's "id" to identify the ToggleSwitch. For id 'photos', the handler first dispatches filterSlice.updatePhotos action and then dispatches filterSlice.getList action. For id 'hide' (short for "hide map icons"), the handler dispatches a filterSlice.updateHide action.

In addition the Filters contain a Status (src/components/FiterBar/Filters/Status/index.jsx) which is a simple div showing the date range for search results and maps.

RadioButton

The RadioButton component, defined in /src/components/FilterBar/Filters/RadioButton/index.jsx is a simple input html element with type radio and label with a Font Awesome Icon. They represent the people, places, and stories categories for a search.

ToggleSwitch

The ToggleSwitch component, defined in /src/components/FiterBar/Filters/ToggleSwitch/index.jsx, consist of input html element type checkbox and a label containing alternative and additional labels depending on the media size. They allow the user to filter on if a search result has photos or if the map should show icons.

Explorer

The Explorer component, defined in src/components/Explorer/index.js, controls the display of the search result list (src/components/Explorer/List/index.jsx), details of selected search result items, (src/components/Explorer/Details/index.jsx, the component for submitting a story (src/components/Explorer/Submit/index.jsx), and the Map (src/components/Explorer/Map/index.jsx) using selectors from the listSlice, detailsSlice, submitSlice reducers.

List

The List component, defined in /src/components/Explorer/List/index.jsx, contains a Result component and 3 Section component. The Results component is displayed while waiting for search results or any error messages. The three Section components are containers for search results in the categories people, places and stories. The category is identified by the type Section prop. The list (= selectAllList from the listSlice) variable controls the display of Sections using the JavaScript convention guarding an evaluation with &&.

Section

The Section component, defined in /src/components/Explorer/Section/index.jsx, contains the category title text and Font Awesome Icon. In addition it maps list.results from the Section props to the Item component.

Item

The Item component, defined in /src/components/Explorer/Section/Item/index.jsx, displays the text for the item and associates the Tooltip content with the item. The data Item prop contains most of the information about the search result item including the recnumber (Record number equivalent to the database id) and the map year.

Submit

The Submit component, defined in /src/components/Explorer/Submit/index.jsx, manages the submitting a story by hiding and showing the phases of submitting story including instructions, closing and loading. The actual "Share a story" button is contained by the Map. It contains the wrapper divs and StoryForm and Loader.

SubmitForm

The SubmitForm component, defined in /src/components/Explorer/Submit/SubmitForm/index.jsx, is the form for submitting a new story. It uses Formik, https://formik.org/, to construct the form, and Yup, https://www.npmjs.com/package/yup, validates the field entries. The Thumb component (/src/component/Explorer/Submit/SubmitForm/Thumb/index.jsx) displays the attached files of the story. The aysnc function submitStory defines the submission process by constructing the record and then uses the JQuery trigger function selected on story-form element to send the "ktt:add-story" event. The URL for the submission is specified in the Map component (/src/components/Explorer/Map/index.js) by the FeatureLayer, storyPointLayer,applyEdits function.

Thumb

The Thumb component, defined in /src/components/Explorer/Submit/SubmitForm/Thumb/index.jsx, displays thumbnails for files attached to the story. It is an img tag followed by a "clear" span for clearing the attachment. A react useEffect hook reads the file as a data URL depending on the file attachment.

Map/KeTTMap

The Map components, defined in /src/components/Explorer/index.js, is the largest and most involved react component.

Challenges

- ArcGIS JS API traditionally uses CDN modules while React uses ES modules. The solution is to use esri-loader, https://www.npmjs.com/package/esri-loader. See the initializing useEffect.
- Integrating external React components controlling the map view. Potential solutions are
  - The technique used in this application is additional handlers/listeners of DOM elements defined in the Map component and then call ESRI supplied Map methods. The disadvantage of this approach is that developers must remember the DOM structure. The advantage is that it is a traditional technique.
  - Create and send an event from the external React component and listen for the event in the Map component. This technique is used once in the code to submit a story to the feature layer. This technique has the advantage that developers do not need to remember the specifics of DOM elements.
  - Use Redux selectors
- Map variables that should persist after rendering. The solution is to use React useRef hooks.

All of the below are defined in the useEffect
Loaded ESRI modules

- esri/Map
- esri/views/MapView
- esri/Basemap
- esri/layers/VectorTileLayer
- esri/layers/TileLayer
- esri/layers/FeatureLayer
- esri/Graphic
- esri/geometry/SpatialReference
- esri/core/watchUtils
- esri/geometry/Point
- esri/widgets/Slider
- esri/widgets/Expand
- esri/widgets/BasemapToggle

DOM Event Listeners

- $('#share-story') … on('click', function () {
- $('body').on('ktt:add-story', '#story-form', function() { <- JQuery ktt:add-story event
- $('#explorer-help') … .on('click', function () {});

The rest of DOM Event listeners are defined in view.when().then(…) ~ line 298 to 1142

- $('body').on('click', '.intro-options-explore', function (e) {

- $('.page-content').on('click', '.main-map-picker .overlay',function () {
- $('body').on('click', '.main-map-picker .basemap', function () {
- $('#time-chooser-select').on('change', function (e) {
- $('.segment').on('click', function (e) {
- $('.navbar-middle').on('click', '.timeline-reset', function () {
- $('body').on('click', '#intro-options-search-icon', function (e) {
- $('body').on('keydown', '#search-landing', function (e) {
- $('#search-icon').on('click', function (e) {
- $('#search').on('keydown', function (e) {
- $('body').on('click', '#search-clear', function () {
- $('.radio-button-input').on('click', function () {
- $('.toggle-switch-checkbox').on('click', function () {
- $('.page-content').on('click', '.list-results-item', function () {
- $('.page-content').on('click', '.related-data-item', function () {
- $('.page-content').on('click', '.map-popup-tabs .tab', function () {
- $('.page-content').on('click', '.map-popup.grid .data li', function () {
- $('.page-content').on('click', '.map-popup.marker .data li',function () {
- $('.page-content').on('click', '.timemachine .back', function () {
- $('.page-content').on('click','.timemachine .forward', function () {
- $('.page-content').on('click', '.data-actions .timemachine .tooltip span', function () {
- $('.page-content').on('click', '.full-details', function () {
- $('.page-content').on('click', '.map-popup-data li', function () {
- $('body').on('click', '.history-list-item', function () {
- $('.page-content').on('change', '#details-source', function () {

ESRI Widgets

- opacitySlider = new Slider
- basemapToggle = new BasemapToggle
- mapPickerExpand = new Expand
- view = new MapView

Function Declarations

- function loadDetails(id, recnumber, loctype, title) { **Line 805 to 816.** Dispatches
  - updateListItem({ recnumber, loctype })
  - toggleList('hide'));
  - getDetails({ id, recnumber, loctype })
  - toggleDetails('show')
  - toggleSubmit('hide')
  - toggleRelated('hide')
  - setActiveTab('')
  - updateSearch(title)
  - getList({})
- function listClick(item) {
- function hideMarkers(checked) {

- function resetMap() {
- function resetTimeline() {
- function landingSearch() {
- function mainSearch(clear) {
- function moveThroughTime(segmentId) {
- function dateChange(min, max, url) {
- function updateOpacity() {
- function toggleTiles(state) {
- function updateGrid(view, filters) {
- function generateGrid(view, cells, type, size, radius) {
- function createGridLayer(view, graphics, type, size) {
- function generateMarkers(view, markers) {
- function createActiveMarkerLayer(view, graphics) {
- function createInactiveMarkerLayer(view, graphics) {
- function createTileLayer(options) {
- function hideLayer(id, layers) {
- function addToView(layer) {
- function updateTimeline(mapyear) {
- function gotoMarker(point, itemId, recnumber, markerid, loctype, type) {

ESRI Listeners/Handlers
Now view is the MapView
- view.on('click', function (event) {
- view.hitTest(event).then(function (response) {
- watchUtils.whenTrue(view, 'stationary', function () {

In the component
Methods
- function attachmentifyPoint(pt) {

Asynchronous Functions
- asyncGridPopUp = (target) => {
- asyncMarkerInfo = (recnumber, markerid, loctype, type, filterVal) => { **//POPUP - When list item is clicked. line 2325 to 2433.** Makes axios post to MARKER_INFO. Makes popup DOM
- asyncMarkerPopUp = (target) => {
- asyncMarkerTitle = (target) => {
- asyncGrid = (view, filters, size) => {
- asyncMarkers = (view, filters, inactive, extent) => {

More functions
- getTabStatus = (tabCount) => {
- handleTimePeriod = () => {
- arcgisSafeString = (str) => {

The Return

```
 return (
  <div className={wrapperClasses}>
   <div className="webmap map" ref={mapRef} />
   <div id="share-help">
    <div id="share-story" className="share-story">
     <FontAwesomeIcon icon={faPencil} className="fa-icon" />
     <span>
      Share a
      <br />
      story
     </span>
    </div>
    <Help />
   </div>
   <Chooser show={showTimeChooser} update={handleTimePeriod} />
   {loadingMarkers && <Loader />}
  </div>
 );
```

Help

The Help component, defined in src/Explorer/Map/Help/index.jsx, is the container for the "I need help" help button and the material UI Dialog (https://mui.com/material-ui/react-dialog/), BootstrapDialog and BootstrapDialogTitle which is a modal. The components BootstrapDialog and BootstrapDialogTitle are defined in the title. BootstrapDialogs contains Tabs for video help.

Taps ie BasicTabs

BasicTabs component, defined in src/components/Explorer/Map/Help/Tabs/index.jsx, is a Material UX Box, https://mui.com/material-ui/react-box/#main-content, for a Material UI Taps (https://mui.com/material-ui/react-tabs/), and TapPanels (defined in BasicTaps) BasicTaps are iframe for videos.

Chooser

Chooser component, defined in src/components/Explorer/Map/Chooser/index.jsx, is a home made overlay of the map when the user zooms past the grid threshold and no time period shows. It is a html select composed from the timelineSlice.segments. The segments are mapped to the select options. The select handler, handleSelect, dispatches actions to the timelineSlice.

mapPicker.js

The function mapPickerList, defined in src/components/Explorer/Map/modules/mapPicker.js, generates the context of the ESRI Expand widget in the map, mapPickerExpand, from an axios PHP API POST, MAP_PICKER.

Details

The Details component, defined in src/components/Explorers/Detals/index.jsx, displays the panel with the details of a record and a ESRI Map showing the location of related content. The structure of the details:

- h1 html with content name which represents the title
- a select of sources mapped to a Source component (src/components/Explorers/Detials/Source/index.jsx) and handler handleSourceChange.
- followed by a "detailed-blocks" div which maps data to the Data component (src/components/Explorers/Data/Source/index.jsx)
- any attached photos in "detail-photos" div.
- The "detail-actions" div follows which contains buttons for "share related story" flagging the story (Flag), data resources, ask (an archivist) button, "share record" (Share).
- Related Content
- Finally a Map (src/components/Explorers/Detals/Map/index.jsx).

The state variable sources are from a useSelector of sources in the detailsSlice. The useEffect finds the selected source in sources and stores the index in selectedClient. The useEffect then dispatches historySlice.updateHistoryItems action.

A JQuery selector is used to set the closeDetails handler for the time-reset. The close details dispatches a sequence of detailsSlice actions to clear list items, and hide the Details and Related components and finally to show the List component.

The handler for the source select, handleSourceChange, dispatches the toggleRelated hiding the related, the listSlice.updateListItem action, the detailsSlice.getDetails action and finally the detailsSlice.getRelated action.

The action buttons "data resources" and "ask an archivist" open a new browser tab of www.keweenawhistory.com. Clicking on the "share related story" dispatches the submitSlice.toggleSubmit action which shows the Submit component.

Source

The Source component, defined in src/components/Explorer/Details/Source/index.jsx, is a select option html tag. The attributes for the option tags are provided by the components props. A JQuery selects in the Details components associated handlers.

Data

The Data component, defined in src/components/Explorer/Details/Data/index.jsx, constructs the detail-block by mapping the item fields passed to the component as a prop.

Flag

Flag component, defined in src/components/Explorer/Details/Flag/index.jsx, is a button and Material UI Dialog (a modal) to show after the flag has been POSTed. The handler for the button, handleClickOpen, POST to FLAG PHP API with the recordID.

Related

The Related component, defined in src/components/Explorer/Details/Related/index.jsx, is constructed with animated divs. The useEffect repeatedly dispatches relatedSlice/getRelated thunk action calling the RELATED_CONTENT API every half second. **(I think that the setInterval in the useEffect is a hack. I think that it is waiting on the details-source div to be mounted)** The detail-related-heading div handler, handleHeadingClick, dispatches the toggleRelated action to hide expanded related. The detail-related-tabs is a collection of tab divs. The handler for the tab divs, handleTabClick, is passed the related category and dispatches the setActiveTab with the category and dispatches the toggleRelated showing the related category list. There is a Data component for each category and will appear if their category matches the active tab.

Data

The Related component, defined in src/components/Explorer/Details/Related/Data/index.jsx, manages and constructs the related list composed of accordion-heading divs and Group components (src/components/Explorer/Details/Related/Data/Group/index.jsx). (Note that Data's data prop is an array; generally it has only one element, but sometimes more, eg "family" and "same location") Clicking on the heading opens the accordion showing a list of related entries.

Group

The Group component, defined in src/components/Explorer/Details/Related/Data/Group/index.jsx, maps the results (related content) to a list of related-data-item divs. The onClick handler, handleClick, dispatches getDetails action with the recnumber, toggleRelated hiding the related, updateListItem action with recnumber, and updateSearch actions and finally getList clearing search list. The final result shows a new Details component. Group component conditionally renders a footer div "Show On Map" and a ToggleSwitch. The Group component passes down to the ToggleSwitch prop onChange. The handler, handleChange, dispatches related/toggleRelatedMap. This will show a map with an icon indicating the related location.

ToggleSwitch

The ToggleSwitch component, defined in src/components/Explorer/Details/Related/Data/Group/ToggleSwitch/index.jsx, implements a general toggle switch. It requires props id, checked, and onChange props.

Map

The Map component, defined in src/components/Explorer/Details/index.jsx, is an ESRI map for showing related content locations. It is a map with limited interactions.

All of the below are defined the initializing useEffect
Loaded ESRI modules

- esri/Map
- esri/views/MapView
- esri/Basemap
- esri/layers/VectorTileLayer
- esri/layers/TileLayer
- esri/layers/FeatureLayer
- esri/Graphic
- esri/geometry/SpatialReference
- esri/core/watchUtils
- esri/geometry/Point
- esri/widgets/Slider
- esri/widgets/Expand
- esri/widgets/BasemapToggle

There is only a mapRef.

Only asyncMarkerContent functions

const relatedContentLayer = new FeatureLayer({ <<<< ?

In view.when().then(
DOM Listeners

- $('.page-content').on('change', '#details-source', function () {
- $('.page-content').on('click', '.detail-related .related-data-item', function () {
- $('.page-content').on('click', '.related-data-group-footer .toggle-switch-checkbox', function () {
- $('.page-content').on('click', '.detail-related.open .tab', function () {
- $('.page-content').on('click', '.detail-related.open .detail-related-heading', function () {
- $('.page-content').on('click', '.details-map-picker li', function () {

Helper functions

- function updateOpacity() {

- function gotoMarker(point)
- function addTileLayer(url) {
- function removeTileLayer() {
- function addSourceLayer(type, source, point) {
- function addRelatedLayer() {
- function addRelated(type, markers) {
- function removeRelated(markers) {
- function applyEditsToLayer(edits) {
- function removeLayer(layerId) {
- function getUrl(mapyear) {

The Return

```
return (
    <div className={`details-map-container ${show ? 'show' : 'hide'}`}>
      <div className="details-map" ref={mapRef} />
      {loadingMap && <Loader />}
    </div>
  );
```

Deploying

1. From the src folder in the project directory, run "yarn build". Make sure you are using node version 14.15.0 and yarn version 3.2.2 or it won't work.

2. Zip up the build folder that is generated in the project's root directory.

In KeTT's cPanel, under the public_html directory, upload the build zip and unzip it. Inside this folder, the live version of the app uses "montekett", and a backup of the previous version is called "montekett_old".

3. You will want to remove the "montekett_old" directory, rename the current live version to "montekett_old", and then rename the new build folder you uploaded to "montekett" to make it the current live version.

Notes:

- The site will be down for those few seconds you're renaming folders, so try to be quick!
- It will take a few seconds for the new build folder to be recognized on the live site. You might have to hard refresh, CTRL + F5, to see the changes.
- If the new build folder doesn't seem to be working, you can always rename the "montekett_old"folder back to "montekett" to restore the previous, working version.