Exploring Cultural Heritage Documentation with Glambulator

Michael E. Nelson

Cultural Heritage Informatics Initiative Graduate Fellowship

Michigan State University

Executive Summary

The project involved designing and developing a web application to afford the exploration of the CIDOC Conceptual Reference Model (CRM) for documenting cultural heritage phenomena. The project resulted in an application called Glambulator. Glambulator allows the user to browse the classes specified in the CRM, to query an online collection of cultural heritage objects, and to interact with query results as graphs at the instance level. This document describes the larger aims of this project, the product itself, the process by which it was designed and developed, lessons learned, and future directions. A list of references for further reading is also provided.

Background

Glambulator is about exploring the CIDOC Conceptual Reference Model (CIDOC-CRM) and phenomena that have been annotated by it. It's inspired by other applications that afford interaction with RDF resources at the instance level (LodLive) and/or at higher levels of abstraction (WebVOWL). Glambulator is also inspired by firms that propose to enhance supply-chain visibility in the fine art trade, to commercial fishing, or to fashion by committing transactions to a distributed transaction repository – a blockchain.

Information standards like controlled vocabularies, reference models, and ontologies facilitate predicating objects of subjects, making statements about phenomena in some domain. One problem that immediately comes to mind, though, is how to verify any such proposition, especially since attributes impart value. A certificate of authenticity, or some other kind of document issued by an authority serves to verify many statements that happen in everyday life. My state-issued driver's license, for instance, can undergird my purchase of a six-pack, showing that I have the attribute *is of drinking age*. Multiple actors can be involved in bringing a heritage object from one point to another, and blockchain technology itself, rather than any third party, has been proposed as a way to verify attributions.

The Application

The Application is a React.js app for exploring the CIDOC-CRM and collection objects annotated thereby, composed of a Redux.js store, Material-UI components, d3.js, and data returned from HTTP requests to the British Museum SPARQL and URI endpoints.

Startup

At startup, the Application configures and initializes store, and dispatches two actions to bootstrap the store: 1) setting the default SPARQL query for instances of E8_Acquisition, and 2) requesting the CIDOC-CRM document. The CIDOC-CRM document, an XML document, is parsed into a *ResourceCollection*, an array of instances of the custom *Resource* class. The `render` method of the React.js `ReactDOM` API (as distinct from the DOM-independent, isomorphic main React.js module) mounts `<Provider/>`, the component that provides access to the store; `<MuiThemeProvider/>`, the Material-UI wrapper component; and the three container components that constitute Glambulator: `<FeedbackIssuerContainer/>`, `<NavContainer/>`, and `<MainContainer/>`.

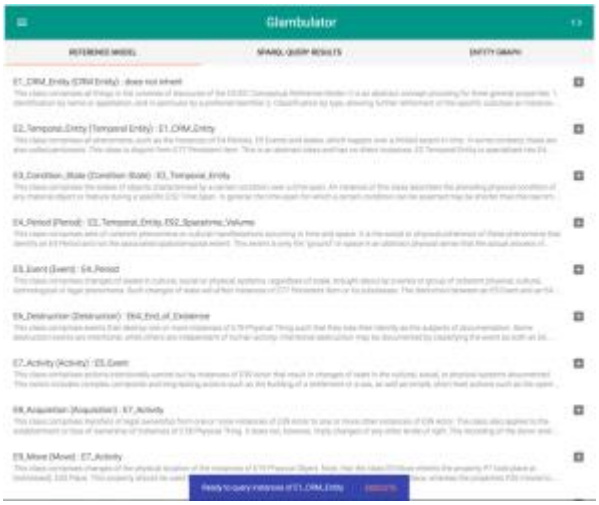

Figure 1: The snackbar provides feedback and (optionally) a follow-on action.

Figure 2: The loading drawer prevents user actions while the application is busy.

Feedback

The `<FeedbackIssuerContainer/>` wraps the `<FeedbackIssuer/>` component, providing access to the parts of the store that it needs to issue feedback to the user about their interactions with the application, and the consequences thereof. The `<FeedbackIssuer/>` controls the *Snackbar* and the *LinearProgress* Material-UI components. The former sends a message to the user (e.g., after receiving an empty HTTP response, after selecting a CIDOC-CRM class) and allows for follow-on actions (e.g., executing a SPARQL Query). These components (along with `<LoadingDrawer/>`) contribute to usability.

Navigation

The `<NavContainer/>` wraps the `<Nav/>` component, providing it access to the parts of the Redux store that it needs to afford navigation to the user of the application: `isNavDrawerOpen`, `isSparqlQueryDrawerOpen`, etc. The `<Nav/>` component mounts the application's `<ActionBar/>`, `<NavDrawer/>`, and `<SparqlQueryDrawer/>`.

Glambulation (sic)

The `<MainContainer/>` wraps the `<Main/>` component, providing it access to the parts of the Redux store that it needs to afford interactions with CIDOC-CRM classes in the *Reference Model* tab, interactions with SPARQL query results in the *SPARQL Query Results* tab, and interactions with single entities in the *Entity Graph* tab. The `<Main/>` component mounts containers for these modules: `<ReferenceModelContainer/>`, `<SparqlQueryContainer/>`, and `<EntityGraphContainer/>`.

Data

Unidirectional data flow in applications is a core principle of both React.js development. Designing a React.js application's data management solution, one decision to make is between a single data store and multiple data stores. For this project, the Redux.js library was selected as the way to provide data to React components and their children from a single, application-wide store (as opposed to component's managing their own state and their children's state, passed down with *props*). Redux was selected for the simple solutions it provides for a) persisting state to the browser's local storage is simply a matter of serializing the single state object and deserializing it, setting it as the initial state at the beginning of the application lifecycle, and for b) testing the application, since the pure functions encouraged by Redux and middleware like *redux-immutable-state-invariant*, in conjunction with the componentization afforded by React, makes for pleasant unit-testing. Except for the asynchronous action creators used in the entity, referenceModel, and sparqlQuery modules, there is no need for mocking.

Modeling Application Data

Within the single state object, the core objects contain data pertaining to CIDOC-CRM, the SPARQL-querying facility, the instances returned by the SPARQL endpoint, and the status of the various user

interface components (e.g., drawer toggled or not, feedback issuing or not). See Appendix A for an example of how the application's store might look at any given point.

Changing State

In keeping with the Redux store pattern, the only ways to change state is by dispatching an action. The actions used to update the data outlined above are outlined in Table 1. Application modules may contain multiple actions, all of which are consumed by a module's reducer. The reducer tells the store what to update and how reproduce itself in response to any given action. Modules' reducers are really about the convenience of separating logic according to modules, since they are ultimately combined into a single reducer.

Table 1: Actions through which application state is changed

ActionType	Reducer
APPLY_ENTITY_GROUP_FILTER	entity
CLEAR_FEEDBACK	common
GET_ENTITY	entity
GET_ENTITY_FAILURE	entity
GET_ENTITY_SUCCESS	entity
GET_REFERENCE_MODEL	referenceModel
GET_SPARQL_QUERY	sparqlQuery
GET_SPARQL_QUERY_FAILURE	sparqlQuery
GET_SPARQL_QUERY_SUCCESS	sparqlQuery
RECEIVE_ENTITY	entity
RECEIVE_REFERENCE_MODEL	referenceModel
RECEIVE_SPARQL_QUERY	sparqlQuery
SELECT_ENTITY	entity
SET_FEEDBACK	common
SET_SPARQL_QUERY	sparqlQuery
SET_TAB	common
TOGGLE_NAV_DRAWER	common
TOGGLE_SPARQL_QUERY_DRAWER	common
UPDATE_ENTITY_DATA	entity

Design

The application uses the Material-UI library of React.js components. This library consists of React components and miscellaneous resources (colors, icons, etc.) that follow the principles of material design. Using this library removed the burden of making low-level decisions about the Application's look and feel, going Google's Material Design guidelines instead. The Material-UI library provides components and other modules that implement, among other things, a color palette, a set of icons, animations like rippling buttons, and other recommendations for achieving the paper-like feel that Material Design strives for. Additional styles were written in Sass and compiled to CSS by webpack.

Development

Git was used for version control. The repository is available for browsing, pull requests, opening issues, etc. at https://github.com/michaelnetbiz/glambulator. Development tasks like building for distribution, serving the application locally, running unit tests, etc. are defined in the project's `package.json` file. Webpack is used to serve the application on a local Express.js development server. Webpack provides features like module bundling and hot module replacement. The value of using Webpack can be seen in the way the CIDOC-CRM RDFS-XML document is included in the Application. With Webpack, it's possible to import the CIDOC-CRM document as a module, leaving Webpack to handle loading it from the Application's assets folder (whether in development or in production) rather than hardcoding the CIDOC-CRM URL in the application. With Babel.js used for transpiling ECMAScript 2015 (ES2015) to browser-

runnable JavaScript, development of the Application can use the latest (or next to latest) APIs like spread operators and generators. ESLint was used for linting the application source during development, and Facebook's Flow static type checking system was used, as well.

Lessons Learned

Lesson 1: Avoid Trying to Reinvent the Wheel

At first, I wrote a wrapper for the vanilla JavaScript XMLHttpRequest API and an interface for the wrapper for each module that needed to access the network. I spent way too much time fine-tuning these services, and eventually decided to use the vanilla JavaScript *fetch* API. This principle could have been applied regarding the way I handled SPARQL queries, as well. As a newcomer to SPARQL and interacting with RDF stores, I wanted to handle it all myself, to better understand these technologies. But it might have been prudent to use libraries for working with SPARQL. The consequence of handling SPARQL myself is that the application is very light when it comes to SPARQL querying; all it can do, now, is request for instances of a class!

Lesson 2: Mobile-First Development is Easier

I knew that mobile-first is a best practice but I did not know that it was a practice that would make life easier for me. Not only should mobile viewports be prioritized by developers due to the ubiquity of internet-enabled mobile devices; also, it's easier to develop with this requirement in place from the very beginning. Providing for mobile viewports as an afterthought complicates development. It's a whole lot easier to provide for mobile viewports *first,* working your way out to larger viewports.

Future Directions

Writing documentation for each module in Glambulator will set the conditions for improving the structure of the application and will itself simplify project management. Writing unit-tests will make existing modules more robust by eliciting bugs, and will streamline the addition of modules, in the future. In keeping with the abovementioned lesson on not trying to reinvent the wheel, it would be worthwhile to look at the potential benefits of using the *d3-sparql* npm library, among other third-party libraries.

Make the Entity Graph More Informative

Currently, the entity graph is not very informative. Relationships are hardly discernible, and data at the vertex level of granularity are only accessible with recourse to *SubjectCards* and the tables of statements therein.

Add Zooming and Dragging Functionality to the Entity Graph

Making the entity graph zoomable and draggable is necessary to facilitate navigation given the Application's potential for expansive entity graphs. Many British Museum collection objects are associated with dozens, if not hundreds of statements. Currently, after the graph exceeds fifty entities, or so, it basically becomes unintelligible.

Add More SPARQL Endpoints and Queries

Experimenting with other cultural heritage-related SPARQL endpoints, and expanding upon the currently very limited SPARQL query functionality are two steps that both would make the Application more interesting.

Further Reading

CIDOC-CRM

CIDOC-CRM website

d3.js

d3.js website

Manipulating SVG with d3.js

Example forced-directed graph with d3.js

Example forced-directed graph with d3.js and React.js

Example directed graph with d3.js

Commentary on handling new nodes in a d3.js force-directed graph

ECMAScript 2015 (ES2015)

Mozilla Developer Network website

Blog posts on various ES2015 APIs

Inverted-Triangle CSS

Podcast interview about using ITCSS

Material-UI Documentation

Material-UI website

Material Design website

React.js (+ Redux.js + d3.js) Tutorials and Examples

Pluralsight course on building an a react-redux application

Exemplary react-redux application

Example d3 layouts with React.js

Commentary on working with React.js and d3.js

Example React.js components that use d3.js

SPARQL

SPARQL documentation

Appendix A

```
{
  common: {
    currentTab: 'entityGraph',
    feedbackAction: '',
```

```
  feedbackContent: '',
  isFeedbackIssuing: false,
  loadingColor: '#3949ab',
  isMobile: false,
  isNavDrawerOpen: false,
  isSparqlQueryDrawerOpen: false
},
entity: {
  entities: [
    [
      -202467184,
      {
        type: 'uri',
        value: 'http://collection.britishmuseum.org/id/object/23',
        isFocus: false,
        id: -202467184,
        className: 'britishMuseumCollectionObject',
        groupNumber: 0,
        abbreviatedValue: 'id/object/23',
        index: 0,
        x: 517.0668538912479,
        y: 496.24175125267817,
        vy: 15.751936818846984,
        vx: -7.000978332103126
      }
    ]
  ]
  entityGroupFilter: -1,
  entitySelection: {
    type: 'uri',
    value: 'http://collection.britishmuseum.org/id/object/23',
    isFocus: true,
    id: -202467184,
    className: 'britishMuseumCollectionObject',
    groupNumber: 0,
    abbreviatedValue: 'id/object/23'
  }
```

```
    isEntityLoading: false,
    statements: [
      [
        40757526,
        {
          subj: {
            type: 'uri',
            value: 'http://collection.britishmuseum.org/id/object/23',
            isFocus: false,
            id: -202467184,
            className: 'britishMuseumCollectionObject',
            groupNumber: 0,
            abbreviatedValue: 'id/object/23',
            index: 0,
            x: 517.0668538912479,
            y: 496.24175125267817,
            vy: 15.751936818846984,
            vx: -7.000978332103126
          },
          pred: {
            type: 'uri',
            value: 'http://erlangen-crm.org/current/P50_has_current_keeper',
            isFocus: false,
            id: -918561899,
            className: 'ontologyResource',
            groupNumber: 2,
            abbreviatedValue: 'current/P50_has_current_keeper',
            index: 3,
            x: 757.2252906268773,
            y: 356.55481201432383,
            vy: 12.156231433422803,
            vx: 7.547420643036565
          },
          obj: {
            type: 'uri',
            value:        'http://collection.britishmuseum.org/id/the-british-
museum',
```

```
            isFocus: false,
            id: -1279364364,
            className: 'britishMuseumCollectionObject',
            groupNumber: 0,
            abbreviatedValue: 'id/the-british-museum'
          }
        }
      ]
    ]
  }
  referenceModel: {
    isReferenceModelLoading: false,
    resources: [
      {
        id: 'E1_CRM_Entity',
        type: 'rdfs:Class',
        supers: [],
        name: 'CRM Entity',
        description: 'This class comprises all things in the universe of discourse of the CIDOC Conceptual Reference Model. \nIt is an abstract concept providing for three general properties:\n1.\tIdentification by name or appellation, and in particular by a preferred identifier\n2.\tClassification by type, allowing further refinement of the specific subclass an instance belongs to \n3.\tAttachment of free text for the expression of anything not captured by formal properties\nWith the exception of E59 Primitive Value, all other classes within the CRM are directly or indirectly specialisations of E1 CRM Entity. \n'
      }
    ]
    version: '6.2.1'
    sparqlQuery: {
      sparqlQueryDescription: 'Returns twenty instances of E1_CRM_Entity.',
      sparqlQueryExpression: {
        prefix: 'crm: <http://erlangen-crm.org/current/>',
        'select distinct': '?instance',
        where: '{ ?instance a crm:E1_CRM_Entity }',
        filter: false,
        order: false,
        limit: 20
      },
```

```
      isSparqlQueryLoading: false,
      sparqlQueryName: 'isa E1_CRM_Entity',
      sparqlQueryResults: [
        [
          -1279364364,
          {
            type: 'uri',
            value:        'http://collection.britishmuseum.org/id/the-british-
museum',
            isFocus: false,
            id: -1279364364,
            className: 'britishMuseumCollectionObject',
            groupNumber: 0,
            abbreviatedValue: 'id/the-british-museum'
          }
        ]
      ]
    }
  }
}
```