Walmart Internship Notes

Summer 2020

Contents

Walmart

Personal

Questions

- onboarding:
  - advantages of Hapi over Express?
  - any way to check if its working besides logging the cookies?
    - eg. which features will get toggled by setting cookies?
- cookies:
  - why not a single cookie that hooks into database?
    - ie. why so many Walmart cookies?
  - downside for only using `__Secure` prefixed cookies?
  - danger of cross origin requests?
- other:
  - reasons / goals for GLASS project?
  - does Walmart use a UI library, is it proprietary?
    - do frontend developers work on UI components, or do they source from a styled component library created by UI designers?
    - are they responsible for translating mockups from designers into code?
  - React excels as SPA, but Walmart website is too large, so composed into different applications?
  - scaling issues and solutions?
    - preparing far in advance for holidary rush
  - what metrics are measured for AB testing?
    - what to optimize for, user retention or how likely a user is to buy a product, etc.?
  - what tools are used for devops testing?
    - unit vs. end to end testing, etc.

Learning Goals

- what are concepts and skills I can learn at this internship that are *unique* to the opportunities presented here...

1. holistic view of the various software stack layers and technologies that are integrated together to compose a large application like Walmart's:
    - what, how, and why tools are used in the frontend, backend, devops, testing, deployment, etc.
2. *scaling* an application to provide for a large userbase:
    - the issues and solutions behind preparing a large eCommerce site like Walmart's for the holiday rush
    - is load balancing performed and how, handling timeout errors in case backend becomes overburdened with requests, etc.
3. devops related skills:
    - writing different kinds of tests (unit, E2E, etc.)
    - learning tools used for acquiring metrics
    - working with and analyzing testing data
4. user *experience* related optimizations:
    - mainly, optimizing performance and site responsiveness in concrete ways
    - practice writing complex client side code that is still efficient and seamless to users
5. security / backend
    - handling the many transactions, changes to accounts, database operations, etc. securely
    - what is encrypted, are forms sanitized, etc.

Log

- Thurs 6/26:
- what I did:
    - finished going through onboarding doc on Confluence about specific Walmart technologies
    - forked and cloned the PAC and cart repos
    - working through issues of building and running the apps
    - built cart and checkout apps (PAC gives me SSL error):
        * checkout app issues fixed by blocking the instart URL
    - downloaded Denys's extension and getting an understanding of what it does:
        * *intercepts* requests made to walmart websites and sets the corresponding cookies to match the environment
    - looked into the cart structure a little bit
        * standardized components from @walmart-components package

- what to do:
  - brainstorm overall plugin design
  - start working on plugin
    - finish easy features first, get them out of the way
    - functionality, then styling
- Wed 6/25:
- what I did:
  - started going through the onboarding doc on Confluence
  - learned more about Electrode
  - config stuff:
    - reinstalled Node, installed new tools, npmrc stuff
    - set up Github keys, etc.
- what to do:
  - finish up going through the doc:
    - Electrode section, more Walmart specific technologies, app introductions
  - locally setup CXO apps

Week 1

Cross-Browser Extensions

- WebExtension is a cross-browser system for developing browser add-ons:
  - promotes cross-browser compatibility
  - *already* supported and *mostly* compatible with Google Chrome's extension API
    - thus extensions written for Chrome can be easily ported to Firefox, Edge, etc.
  - JavaScript APIs for WebExtensions:
    - APIs allow interaction with cookies, browser storage, tabs, windows, etc.
    - can be used in any part of an extension eg. sidebars, popups, pages
  - every extension using the WebExtensions API must contain the `manifest.json` file:
    - specifies basic metadata and functionalities of the extension
    - different browsers support different manifest keys and functionalities, see documentation
      - most browsers support the major manifest keys
- however, there *are* some differences between the API for Chrome, Firefox, and edge:
  1. in Chrome, the JS APIs are accessed under the `chrome` namespace, instead of the `browser` namespace in Firefox and Edge
  2. in Chrome, async APIs are implemented using callbacks in Chrome and Edge, rather than the promises in Firefox
     - note that Firefox *does* support `chrome` and callbacks as a porting aid
       - *not* part of the WebExtensions standard
  3. different behavior between Chrome and Firefox for the notifications, proxy, tabs, webRequest, and windows APIs
  4. other incompatibilities such as relative vs. absolute URL resolution, missing APIs or manifest features
     - detailed in the documentation
  - in addition, each browser loads extensions in a different manner
- there are other tools to develop extensions:
  - the Add-on SDK and classic XUL/XPCOM extensions
  - most extensions now use WebExtensions, well standardized across browsers

- for Microsoft Edge:
  - the newer version of the Edge browser is built on Chromium, and thus natively supports Chrome extensions
    - with some differences, eg. no auth token API for payments
  - like Firefox, supported APIs and manifest keys are mostly similar to the ones supported by Chrome

Anatomy of an Extension

- the only file that is required in every extension is the `manifest.json` file:
  - contains basic metadata eg. name, versions, permissions, and provides pointers to the other files in the extension
- other types of extension files:
  - background scripts implement any long-running logic, independent of a particular page or window lifetime:
    - loaded as soon as extension is loaded, stay loaded until disabled or uninstalled
    - can specify scripts using the `background.scripts` key, or an HTML page with embedded scripts in the `background.page` key
  - sidebars, popups, option pages:
    - content for each is defined using an HTML page
    - sidebar is a pane displayed on the left-hand side
    - popup is a dialog that displays when the user clicks a toobar or address bar button
    - options page shown when accessing extension preferences
  - extension pages:
    - other HTML pagees that are not attached to some predefined user interface component
    - typically loaded using `window.create` or `tabs.create`
  - content scripts:
    - content scripts access and manipulate web pages, eg. can see and manipulate the page's DOM
      - note that content scripts can see the DOM, but not variables defined by page scripts
    - loaded into web pages and run in the context of that particular page
    - unlike ordinary page scripts, they can use some WebExtension APIs
    - to communicate with background scripts:
      - exchange one-time messages using `browser.runtime.sendMessage` and `browser.runtime.onMessage`
      - using the `runtime.Port` to establish a longer-lived connection
    - loaded at install time with the `context_scripts` manifest key, or at

runtime using the `contentScripts` API or `tabs.executeScript`
  - web accessible resources:
    * various resources that need to be accessible to scripts
- almost all of these extension scripts have access to the JS WebExtension APIs
  - content scripts access a smaller subset, eg. storage and messaging APIs
  - note that there is one global scope per extension, so different content scripts can access each other's variables, even if loaded at different times

Webpack

- **webpack** is a static module bundler for JavaScript applications:
  - when it processes an application, it internally builds a **dependency graph** that maps all the modules needed by the project, and generates **bundles**
  - modules can be ES5 modules, as well as CommonJS or AMD modules, CSS modules, etc.
- motivation:
  - natively, there are two ways to run JavaScript in a browser:
    * include a script for each functionality
    * use a single large JS file
    * *cons*:
      · network bottlenecks with multiple scripts, or problems with scope, size, readability, etc. with a single monolith file
  - people used IIFEs to solve scoping issues for large projects:
    * can concatenate multiple files into one without scope collision
    * developers used tools like Make, Gulp, Grunt
    * *cons*:
      · changing one file forces rebuilding entire project
      · no lazy-loading, eg. if a single function from a library is needed, entire library is added to source
  - module support:
    * Node.js ie. CommonJS modules motivated using modules for browser side JS
    * tools like Browserify, RequireJS, SystemJS allowed writing CommonJS modules in the browser
    * now, ES5 has native module support
  - handling dependencies and automatic dependency collection:
    * previously, tools such as task runners required *manual* declarations of all dependencies upfront
    * bundlers like webpack automaticaly infer the dependency graph based on imports and exports

    - webpack allows developers to write modules while still supporting any module format before ES5, as well as handle resources and assets such as images, fonts, and stylesheets at the same time
        * webpack also provides performance features such as prefetching and async chunk loading
            · eg. `non-initial` chunks can be loaded dynamically when needed
- key concepts and configuration parameters:
    - the **entry point** indicates which module webpack should use to build out from:
        * `./src/index.js` by default
    - the **output** property tells webpack where to emit the created bundles:
        * defaults to the `./dist` folder
    - different **loaders** allow webpack to process types of files other than JavaScript and JSON files, and convert them into valid modules:
        * each loader has a `test` property that indicates which files to transform, and the `use` property indicates which loader to use
    - **plugins** transform a wider range of tasks besides module transformation:
        * eg. bundle optimization, asset management, environment variable injection
    - a **mode** parameter can specify different environments, eg. production or devlopment

Application Performance

- improving performance for web applications:
    - important to have measured metrics to quantify performance improvements:
        * eg. time to interactive, time to first byte, time to fully loaded page
        * but must keep in mind user pain points, ie. user experience is the end goal
        * tools such as Chrome Dev Tools, WebPageTest, Lighthouse, Webpack Bundle analyzer can provide these metrics
- performance principles:
    1. do minimal work that is necessary in the browser
    2. minimze RTT
    3. optimize perceived performance
- minimizing work:
    - code splitting and lazy loading:
        * load *chunks* of code, organized by components, using **dynamic im-**

ports
    - however, aggressive splitting can cause duplication and reduced compression
  - slim down libraries:
    - eg. using smaller libraries and modern tool versions
  - differential serving:
    - transpiling and polyfills increases code size
    - thus serve ES6 scripts directly to supported clients instead of transpiling them unnecessarily
- minimizing RTT:
  - create shared bundles:
    - download *shared* library code only once across the entire site
    - eg. bundles can be made using Webpack's DLL plugin
  - shared components:
    - eg. shared header/footer are rendered in parallel for different pages during SSR
    - allows component caching
  - compression:
    - eg. the Brotli tool's dynamic compression is more efficient than Gzip
    - but dynamic compression has an overhead, so prebuild and host compressed assets from CDNs
- optimizing perceived performance:
  - leveraging priorities:
    - using async or deferred script tags to prevent blocking of HTML parsing:
      - JS async and defer
      - with async, the script gets downloaded asynchronously and then executed as soon as it's downloaded
      - with defer, the script gets downloaded asynchronously, but executed only when document parsing is completed
      - in addition, with defer, scripts execute in the same order they are called
    - or resource hints such as `rel="preload"` to prioritize assets
  - prefetching or preconnecting for faster DNS lookups:
    - use resource hint `rel="prefetch"` to prefetch assets (downloading in parallel), before they are needed
    - but this will consume more of the user's bandwidth

Cookies

- HTTP **cookies** remember stateful information for the inherently stateless HTTP protocol:
  - mainly used for session management, personalization, and tracking
  - for general client-side storage, other web storage APIs should be used instead
  - cookies are *visible* and can be *modified* by the end user
  - cookies vs. JSON webtokens:
    - for small to medium websites, session cookies are enough eg. log a user in and access details from the database
    - for an enterprise level site with many third party requests for APIs at different domains, JSON webtokens work better
    - JSON webtokens can encrypt any JSON that is in the payload, more flexible than cookies
- creating cookies in an HTTP response:
  - when an HTTP response includes the `Set-Cookie` header, it tells the client to store a cookie with the corresponding name and value
  - then, with every subsequent request to the *same* server, the browser sends back previously stored cookies in the `Cookie` header
- cookie **attributes**:
  - **session** cookies are deleted when the session ends, while **permanent** cookies last until an `Expires` attribute or time equal to a `Max-Age` attribute
  - cookies with the `Secure` attribute are only sent to the server on HTTPS connections
  - cookies with the `HttpOnly` attribute are inaccessible to the JS `Document.cookie` API
  - by default, the scope of hosts allowed to receive cookies is the same origin that set the cookie, excluding subdomains:
    - the `Domain` attribute specifies a domain *and* its subdomains as the scope
    - the `Path` attribute indiciates a URL path that must exist in the requested URL in order to send the cookie header
    - the `SameSite` attribute requires that a cookie isn't set with cross-origin requests:
      - with `Strict`, cookie is only sent to the same URL as the one that originated it
      - same for `Lax`, except when user navigates to a URL from an external site
      - `None` has no cross-site request restrictions
  - eg. `Set-Cookie: id=wasd; Secure; HttpOnly; SameSite=Strict`
- it is impossible for servers to confirm a cookie was set on a secure origin or to tell where a cookie was set:
  - using **cookie prefixes** asserts specific facts about a cookie, and acts as

a defense measure against a session fixation attack caused by stealing cookies
  - the `__Host-` prefix is accepted in a `Set-Cookie` header *only* if it marked with the `Secure` attribute, was sent from a secure origin, does not have a `Domain` attribute, and has the `Path` set to `/`
    * ie. essentially a domain-locked cookie
  - the `__Secure-` prefix is accepted in a `Set-Cookie` header *only* if it is `Secure` and sent from a secure origin
    * weaker than the host cookie prefix
  - thus on the server side, the application can check for the full cookie name with the prefix to confirm it was securely set
- to mitigate attacks involving cookies:
  - `HttpOnly` and `SameSite` attributes should be used
- cookies are associated with domains:
  - if the domain is the same as the domain of the page you are on, the cookie is a **first-party cookie**
  - if not, it is a **third-party cookie**
  - eg. ads will try and set third-party cookies that essentially track user's browsing histories
    * can be blocked by browser settings or extensions
    * limited by privacy regulations that restrict the use of cookies and trackers
- WebExtension cookies API:
  - `cookies.get cookies.getAll cookies.set cookies.remove` to get and set cookies
  - `cookies.onChanged` is an event handler that fires when a cookie is set or removed

Redux

- **Redux** helps manage **state** in React applications:
  - as a project grows, its state and state changes will grow as well
  - but *pure* frontend components should not need to know about the business *logic*
    * violates separation of concerns, want to separate UI from logic and behaviors
  - Redux helps with providing each frontend component the exact piece of state it needs
    * Redux can also hold business logic in its *own* layer called **middleware**
  - other solutions:

        * passing state through children props, and the React context API
    - sourced from Redux tutorial
- terminology:
    - the Redux **store** is a centralized data store that holds all of the application's state:
        * created with `createStore`, takes a reducer and an optional initial state
    - a Redux **reducer** is a JS function that handles state changes:
        * takes in the current state and an action, returns the *next* state of the application
        * like in React, state is immutable and thus should not change *in place*
            · thus methods like `concat` and `Object.assign`, or the spread operator should be used
        * thus the reducer must be a *pure* function
    - a Redux **action** is a signal that is sent to the store:
        * thus the only way to change the state is through *dispatching* an action
            · dispatching an action forwards the action to the reducer
        * the action is a JS object with two properties type and payload:
            · the type describes *how* the state should change, and the optional payload describes *what* should be changed
            · type is normally a string or string constant
            · eg. `{type: 'ADD_ARTICLE': payload: {...}}`
        * as a Redux best practice, actions are normally *wrapped* within a function, so that action creation is abstracted away
            · these wrapping functions are known as action creators
- important Redux methods:
    - `getState` reads the current state of the data store
    - `dispatch` dispatches an action
    - `subscribe` accepts a callback that fires whenever an action is dispatched
- connecting React and Redux:
    - Redux is actually framework *agnostic*, ie. can be used with vanilla JS, Angular, React, etc.
- for React, the `react-redux` library is used:
    - the `Provider` high order component takes in a Redux store wraps up a React application
    - the `connect` method then connects a React component with the store
    - `connect` takes a `mapStateToProps` and a `mapDispatchToProps` function:
        * `mapStateToProps` connects a part of Redux state to the props of a React component
        * `mapDispatchToProps` connects Redux actions to React props, given dispatch as a parameter

Middleware

- Redux **middleware** provides a way to check the action payload *before* the action gets to the reducer:
  - ie. a way to *tap* into the application's flow
  - eg. may want to verify or sanitize input before the input reaches the reducer:
    * but want to *avoid* any unnecessary business logic in our components!
    * middleware solves this issue
  - *pros*:
    * most logic can live outside the UI library
    * middleware becomes reusable pieces of logic, easy to reason about
    * middleware can be tested in isolation
  - middleware is a function returning a function
    * has access to `getState` and `dispatch`
  - middleware is *registered* using `applyMiddleware` and passed to `createStore`

Example middleware that filters out words in an action:

```
const forbidden = ['spam', 'nigerian prince'];

export function forbiddenWords({ getState, dispatch }) {
  return function(next) {
    return function(action) {
      if (action.type === 'ADD_ARTICLE') {
        const found = forbidden.filter(words =>
          action.payload.body.includes(words)
        );
        if (found.length) {
          // new, rerouted dispatch
          return dispatch({ type: 'FOUND_BAD_WORD' });
        }
      }
      return next(action); // lets next middleware run
    };
  };
}
```

Registering middleware:

```
import { createStore, applyMiddleware } from 'redux';
import rootReducer from '/reducers/index';
import { forbiddenWords } from '/middleware'
```

```
const store = createStore(rootReducer, applyMiddleware(forbiddenWords));

export default store;
```

- how should Redux deal with asynchronous functions?
  - previously, all data was handled *synchronously*
  - reducers should not call async functions, should remain simple and clean
  - actions must be plain objects, and so action creators cannot automatically call async functions
  - instead, a custom middleware called `redux-thunk` can be used that *allows* async work to be done in action creators
    - eg. dispatch other actions in response to AJAX calls
  - multiple middlewares can be integrated using `compose`

Composing and using `redux-thunk` :

```
import { createStore, applyMiddleware, copmose } from 'redux';
import thunk from 'redux-thunk';
...
const store = createStore(rootReducer,
  compose(applyMiddleware(forbiddenWords, thunk))
);

// in an action creator:
export function getData() {
  return function(dispatch) {
    return fetch(url)
      .then(res => res.json())
      .then(json => {
        dispatch({ type: 'DATA_LOADED', payload: json })
      });
  };
}
```

Redux Saga

- another alternative approach to handling async actions is `redux-saga` :
  - another Redux middleware that manages side effects eg. API calls, storage access, other impure actions
    - uses generator functions
  - here, all action creators will only dispatch plain actions
    - unlike with thunks, no async functionalities

  - a Redux saga consists of two main functions:
    * a *watcher* function that watches for specific actions
      · watcher will use `takeEvery` to watch for certain actions
    * in response to that action, the watcher will call a *worker* function that handles async side effects
      · worker can `call` async functions and then `put` ie. dispatch new actions upopn their completion
  - *pros*:
    * clearer separation between sync and async logic than thunks

Example Redux saga:

```
import { takeEvery, call, put } from 'redux-saga/effects';

export default function* watcherSaga() {
  yield takeEvery('DATA_REQUESTED', workerSaga);
}

function* workerSaga(action) {
  try {
    // can also access action.payload
    const payload = yield fetch(url).then(res => res.json());
    yield put({ type: 'DATA_LOADED', payload });
  } catch (err) {
    yield put({ type: 'API_ERROR', payload: err});
  }
}

// simpler, plain action creator:
export function getData() {
  return { type: 'DATA_REQUESTED' };
}
```

Redux Toolkit

- instead of using `createStore` alongside `combineReducers`, `compose`, and `applyMiddleware`:
  - `configureStore` from the Redux toolkit can be used instead
  - creates a store given a configuration object with `reducer` and `middleware` properties
  - no need to use unnecessary boilerplate methods
- `createAction` from the toolkit can be used instead of repeated action creator boilerplate:
  - returns an actual action creator ready to be called with an optional pay-

load
  - eg. `const login = createAction("LOGIN")`
- `createReducer` from the toolkit replaces the switch-case statement used to handle actions:
  - takes an initial state and a *mapping* object where properties are action types and values are the reducing function
  - *pros*:
    * less boilerplate bloat
    * uses `immer` in order to write mutative logic *without* altering the original object
    * no need to even return original state object

Using the Redux toolkit:

```
import { configureStore, getDefaultMiddleware,
         createAction, createReducer } from '@reduxjs/toolkit';

const loginSuccess = createAction("LOGIN_SUCCESS");
const loginFailed = createAction("LOGIN_FAILED");
const middleware = [...getDefaultMiddleware(), ...];
const initState = {...};

const authReducer = createReducer(initState, {
  [loginSuccess]: (state, action) => {
    state.token = action.payload;
  },
  [loginFailed]: (state, action) => {
    state.error = action.payload;
  }
});

const store = configureStore({
  reducer: { auth: authReducer },
  middleware
});
```

Using `createSlice` to simplify even further:

```
import { configureStore, getDefaultMiddleware, createSlice } from '@reduxjs/toolkit';

const middleware = [...getDefaultMiddleware(), ...];
const initState = {...};

const authSlice = createSlice({
```

```
  name: 'auth',
  initialState: initState,
  reducers: {
    loginSuccess: (state, action) => {
      state.token = action.payload;
    },
    loginFailed: (state, action) => {
      state.error = action.payload;
    }
  }
});

const { loginSuccess, loginFailed } = authSlice.actions;
const authReducer = authSlice.reducer;

const store = configureStore({
  reducer: { auth: authReducer },
  middleware
});
```

- finally, `createAsyncThunk` from the toolkit helps clean up async code even further:
  - automatically creates an action creator for each promise state, eg. pending, rejected, or fulfilled

SUIT CSS Naming Conventions

- **SUIT** is a naming convention for CSS that relies on structured class names and meaningful hyphens:
  - helps to encapsulate and declutter CSS names
  - divides CSS classes into **utilities** and **components**:
    - utilities are low-level structural and positional traits
    - components handle any component-specific styling
- utility names follow the syntax `u-<utilityName>` :
  - responsive utilities have the breakpoint patterns `u-sm u-md u-lg`
  - eg. `u-floatLeft` , `u-lg-block`
- component names follow the syntax `<namespace>-<ComponentName>` followed by optional `-<descendantName>--<modifierName>` :
  - this breakdown helps:
    - distinguish between the different parts of the classes
    - keep the specifity of selectors low

        * decouple presentation semantics from document semantics
    - note that the component name is the only name in the convention that uses pascal case
    - eg. `.twt-Button` , `nav-Button` for different namespaces
    - eg. `.Button--default` for a modifier that modifies the presentation of the base component
        * note that the modifier class should be present in *addition* to the base class
    - eg. `.Tweet-header` , `.Tweet-avatar` for desecendants, ie. classes that apply presentation to a descendant on behalf of a main component
    - `is-<stateName>` should be used to reflext changes to a component's state:
        * eg. `.Tweet.is-expanded`
- CSS variables have a syntax similar to copmonent names, but with a prepended `--` :
    - eg. `--ComponentName-descendant-onHover-backgroundColor`

HTML

- `targetElement.closest(selectors)` traverses the element and its parents until it finds a DOM node that matches the provided selector string

Data Attributes

- HTML **data attributes** are used for data that should be associated with a particular element but need not have any defined meaning:
    - ie. store extra information on standard HTML elements without other hacks such as non-standard attributes, extra properties on DOM, or `Node.setUserData`
- syntax:
    - any attribute on any element whose name starts with `data-` is a data attribute
    - eg. `<article id="electric-cars" data-columns="3" data-parent="cars">...`
- accessing data attributes:
    - from JS, can use `getAttribute` or simply access the `dataset` property
        * each property is a string and can be read or written
        * eg. `const article = document.querySelector('#electric-cars')` with `article.dataset.columns` or `article.dataset.parent`
    - from CSS, can use the `attr` function or attribute selectors:
        * eg. `content: attr(data-parent);` or `article[data-columns='3'] {...}`

Walmart Onboarding

Tools

- basic tools / technologies:
  - **NodeJS**:
    - with a proprietary npme (npm enterprise registry)
  - **fyn** as an alternative node package manager:
    - has some offline cached functionalities
    - can be run up to 30% faster than npm
  - **xclap** is a more powerful task runner similar to the npm scripts
- infrastructure:
  - Electrode
  - **Torbit** is an entry-level routing / optimization server:
    - routes incoming requests to the appropriate application server
      - eg. cart requests go to cart page server etc.
    - acts as a CDN for Walmart's resources and serves optimize content eg. lower size images for mobile devices
  - **Quimby** is an orchestration layer for Tempo modules:
    - **Tempo** modules are collections of custom curated data
    - Quimby fetches the module data and delivers it to the consumer apps
    - ie. a service to fetch data from Tempo and provide additional enhancement to the data
  - **Terra-Firma** is the data orchestration layer for the item page:
    - when item page frontend servers receive a request for an item, the item number is used to get all the item data from Tera-Firma
  - **Item Read Orchestration (IRO)** is the layer responsible for providing crucial item data like pricing and availability:
    - called by Terra-Firma to assemble item data for item page servers
    - one of the high traffic receiving services
  - **P13N** is the personalization service that returns various item recommendations
  - **Cloud Configuration Management (CCM)**:
    - where each application stores configurations that can likely be changed anytime in production and shouldn't require a production deployment
    - eg. feature flags, service endpoints
    - CCM are specific to the app and the environment where the appli-

    cation is hosted
    * in the consoles, CCM values are in `_wml.config.ccm`
  - **Shifu** is used as a mock server in order to mock the real backend routes when developing the frontend client:
    * can choose custom mocks to emulate different results from the backend routes
- build deployment tools:
  - **OneOps** is a cloud management and application lifecycle management platform:
    * devs can deploy a given version of code in production environments
    * used to deploy new versions to clouds
  - staging environments:
    * pre-production environments that can be accessed using certain cookie or header combinations
    * different teams use staging environments to perform QA and E2E validation
    * eg. the cookie `SENV=prodb` hits a pre-prod environment for the cart app
  - **Looper** is an automation / deployment tool for continuous integration / continuous deployment (CI/CD):
    * eg. runs unit tests when a pull request is made, automatically release to repo or deploy
    * CI jobs are configured in a Looper YAML file
    * also used to publish to the WalmartLabs (WML) npm
    * built over Jenkins
    * Conqueror is another alternative built over looper
- reporting and monitoring tools:
  - **Medusa** is the reporting platform where teams create dashboards to see their application health in production:
    * eg. health metrics like traffic, response times, error rate
    * production cluster has a specific job that reports these metrics to telemtry services that power the dashboards
  - **Splunk** is used for storing app server, web server, and event based logs being triggered from browser:
    * used for debugging issues by tracking user events
    * also provides dashboards for regular monitoring

Electrode

- **Electrode** is a platform for building scalable universal React and NodeJS applications:
  - features a standardized structure, best practices, and modern technolo-

gies built in
    * supports many frameworks out of the box, eg. React, PostCSS, PhantomJS, Mocha, Inferno
- focuses on performance, component reusability, and simple deployment
- follows the universal JavaScript stack, ie. JS for server client side code
- *pros*:
    * exposes a way to filter through different components and reuse them
    * optimized for **server-side rendering (SSR)**:
        · SSR is good for **search engine optimization (SEO)**, search engines can see and analyze webpage content
        · React is normally CSR, but can be set to be SSR
        · note that some sensitive payment information cannot be SSR, so the checkout page is CSR
    * can be optimized for fast **above the fold (ATF) rendering**:
        · ATF rendering refers to the first visible page to clients
        · while below the fold refers to anything *underneath* that is only visible on scroll
    * fast startup and deployment
- platform is broken into 3 main pillars:
    1. the Electrode Core provides a simple consistent structure that follows best practices
        * easily set up with generators that build boilerplates, and quickly deploys
        * eg. Electrode archetypes are npm modules that encapsulate boilerplates for configurations, workflows, dependencies
        * allows modules to share common scripts, configurations, etc.
    2. additional Electrode Modules add certain features:
        * independent of the core
        * eg. above the fold rendering, configuration management (with Confippet), stateless cross-site request forgery protection, etc.
    3. Electrode Tools help organize reusable components (with Electrode Explorer) and optimize large JS bundles (with Electrify)
        * can be consumed by other applications not built on Electrode
- electrode-ignite is used to quickly create an Electrode application
- Electrode also provides a module for CCM:
    * requires using the HapiJS framework (ie. Express alternative) to run NodeJS server
    * using electrode-ccm-client and electrode-quimby-client
    * takes in configuration for both CCM clients and Electrode CCM
- electrode-config manages application config based on the environment:
    * loads certain JSON files from a `config` directory
    * eg. `default.json`, `production.json`, `development.json`

    - electrode-logging for flexible logging:
        * can specify different transports, eg. stdout, a file, splunk, etc.
        * logging will contain metadata eg. id, request method, hostname, etc. depending on verbosity
        * data can be serialized
- Electrode solutions:
    - standardized Node and React application and component structure
    - isomorphic cookies and configuration support
    - Node server and modules for server
    - WML infrastructure support
    - build/CI/CD setup and support
    - cloud setup, deployment, and monitoring
    - application instrumentation and performance monitoring
- Electrode technology stack:
    - NodeJS
    - HapiJS: NodeJS webserver
    - React: UI framework
    - Redux: UI state management
    - Babel: JavaScript transpiler
    - Webpack: asset bundler
    - PostCSS: CSS with JS
    - ESLint: JS linter
    - Karma: test runner
    - Mocha: test framework

Walmart Apps

- **Post Add to Cart (PAC) Page**:
    - page reached after adding an item to cart from the item page
        * from here, customer can go to either cart or checkout page
    - first page in the CXO funnel
        * download shared code eg. DLLs and shared scripts here
    - *goals*:
        * add to cart action success
        * P13N for related items
- **Cart Page**:
    - the shopping cart page, shows all items in the active cart and in the **Saved For Later (SFL)**
    - customer may be logged in or a guest
- **Checkout Page**:
    - the checkout page, a SPA which allows you to:

    - select fulfillment options, eg. shipping or pickup
    - enter shipping / pickup details
    - select and enter payment method
    - review and then place order
- **Thank You Page**:
  - the post purchase page, with a thank you message and order details

Week 2

Virtual DOM and React Internals

- React terminology:
  - React **elements** are plain objects *describing* a component instance ie. DOM node and its desired properties:
    - has a `type` that can be a DOM node or a React component, and `props` for properties, children, etc.
      - `type` can be the function or class component itself or a string for host components (leaf nodes) eg. `div`
    - these React elements are easy to traverse and are much lighter than the *actual* DOM elements
    - make up the VDOM tree
    - created by `React.createElement()` or JSX
  - React **components** can be classes or functions that fundamentally take in props as their input, and return React elements as their output
  - a React **instance** is what is refered to as `this` in component classes:
    - useful for storing local state and reacting to lifecycle events
  - React implementation notes
- the **virtual DOM (VDOM)** is a programming concept where a *virtual* representation of a UI is kept in memory:
  - this VDOM is synced with the real DOM in a process called **reconciliation**
  - the biggest issue VDOM solves is the performance improvement on DOM manipulation
  - also allows for the declarative API of React where you can tell React what state you want the components to be in
    - abstracts out the manual DOM manipulation and event handling that would otherwise be used
  - React uses internal objects called **fibers** to hold additional information about the component tree

- details of reconciliation and *diffing*:
  - the `render` function creates a tree of React elements, and at the next state or props update `render` will return a different tree of React elements
  - ordinarilly, comparing two trees and generating the minimum operations to transform one tree into another has complexity $O(n^3)$
  - React implements a heuristic $O(n)$ algorithm based on two assumptions:
    1. two elements of different types produce different trees
    2. developer can hint at which child elements may be stable across different renders using the `key` prop
- the diffing algorithm of two trees:
  - first compares the two root elements:
    - if the root elements are of different types, React will tear down the old tree and build the new tree from scratch
      - when tearing down a tree old DOM nodes are destroyed and `componentWillUnmount` is called, and any associated state is lost
      - when building up a new tree, `componentWillMount` is called
      - any components below the root will also get unmounted
    - when the DOM elements are of the same type:
      - React keeps the same underlying DOM node, and only updates the changed attributes or changed style properties
      - when a component updates, the instance stays the same, so the state is maintained across renders, but the props may update
      - React then recurses on the children
    - handling child elements:
      - Reat normally just iterates over both lists of children and generates a mutation whenever there is a difference
      - however, thus the order of changes in the children may cause differing performance, eg. inserting at the beginning has worse performance as every child becomes mutated
      - this is solved through React's `key` attribute
- note that VDOM is different from **shadow DOM**:
  - shadow DOM doesn't represent the entire DOM and comes in smaller pieces
  - shadow DOM creates a *scoped* tree that is connected to a certain element ie. shadow host, but is *separated* from children elements:
    - thus everything added to the shadow DOM is *local*, even styles
    - solves the issues of scoping of variables and of CSS, since styles are isolated within the scope of the shadow DOM
    - also helps with performance, since browser can more accurately determine what needs to be updated
  - ie. *encapsulates* the implementation of web components:

        * eg. the `<input/>` tag has underlying HTML and CSS, but this subtree of DOM nodes is hidden from the main DOM to encapsulate its implementation

Fiber

- **React Fiber** is a new reconciliation engine in React 16:
  - main goal of enable incremental rendering of the VDOM
    * note however that reconciliation and rendering are independent phases:
      · the reconciler computes which parts of an element tree have changed, and the renderer then uses that information to actually update the rendered app
      · this renderer-agnostic design allows for React DOM and React Native to use their own renderers while sharing a React core reconciler
  - advantages of incremental rendering:
    * in a UI, it is not necessary to apply every update immediately
      · can be even wasteful, and drop frames and degrade UX
    * instead, different updates should have different priorities
      · take a *pull* rather than a *push* approach where computations can be delayed by the framework until necessary
    * to achieve this, cannot use purely a *call stack*:
      · need to break rendering work into incremental units
      · need to be able to interrupt the call stack and manipulate stack frames
    * ie. a *time-slicing* approach uses non-blocking rendering, priority-based rendering, and pre-rendering techniques
  - a **fiber**, in a way, represents a unit of work:
    * AKA represents a *virtual* stack frame that can be kept in memory and executed however and *whenever* needed
      · that also corresponds to an instance of a component
    * imagining UI as a pure function, where the output is the same for a given input of props, state, etc.
  - the fiber engine breaks up work into two phases:
    1. in the **render** phase, React applies updates to components by walking through the fiber tree and building a work-in-progress tree marked with any side-effects
       * this phase can be performed *asynchronously* and can be paused and continued
       * enables incremental rendering
    2. in the commit phase, the work that is described in the effects are applied *synchronously*

        - actual DOM updates occur here
    - Fiber overview
    - Walking through Fiber tree
- structure of a fiber:
    - a fiber is a JS object with information about a copmonent, its input, and its output
    - `type` property that is analagous to the type of a React element
    - `key` property used in reconciliation
    - `child` property corresponding to the value returned by a components `render` method
    - `sibling` property for the case where `render` returns multiple children
        - possible due to new React fragments
        - these children properties form a linked list of the fiber nodes
        - in functional analogy, similar to a tail-called function
    - `return` property is the fiber to which the program should return after processing the current one
        - ie. the parent fiber
        - in functional analogy, similar to the return address of a stack frame
    - `pendingProps` for a fiber are set at the beginning of its execution, while `memoizedProps` are set at the end
        - when the two fields are the same, the fiber's previous output can be reused
    - `pendingWorkPriority` indicates the priority of the work represented by the fiber
    - `alternate` property:
        - to *flush* a fiber is to render its output to the screen
        - a *work-in-progress* fiber has not yet completed, ie. a stack frame which has not yet returned
        - thus at any time, a component instance has at most two fibers corresponding to it, the flushed fiber and the work-in-progress-fiber
        - the alternate of the flushed fiber is the work-in-progress, and vice versa
    - `output` is the return value of the fiber, created entirely out of host component leaf nodes
    - note that unlike call stack reconciliation which traverses a tree of elements:
        - fiber reconciiation iterates through a *linear* list of fiber nodes (parent-first, depth-first)
        - whenever there are side effects that have to be applied to a fiber node, these nodes are linked together in a **effect list**
            - allows React to skip nodes that do not have effects to be applied
- root of the fiber tree:
    - every React application has one or more DOM elements acting as con-

tainers
- React creates a fiber root for each of those containers that can be accessed through a reference to the DOM element:
  - eg. `elem._reactRootContainer._internalRoot.current`
- the fiber root is a `HostRoot` with the special linking `fiberRoot.current.stateNode =`
- thus the fiber tree can be explored by accesing the topmost fiber root node, or getting an individual fiber node from a component instance eg. `instance._reactInternalFiber`