In this assignment you will partially implement and test the frontend of your web application. We start with our draft frontend application that contains three views:

- landing page with registration and login
- main page with articles, followers, and status message
- profile page for a user to upload a new profile picture and edit their data

Remember that when tackling any large task our best approach is to divide and conquer. For this assignment there are three major portions:

1. Using React+Redux to transform the three views into a single page application (SPA)
2. Writing unit tests of the desired functionality and mocking the dummy server endpoints
3. Implementing logic for our site and connecting to the dummy server

Additionally we want to perform test driven development as we implement our webapp. Therefore before we implement anything we will first write tests for our functionality. In this way the desired behavior will drive our implementation and design.

Perhaps the most important part of this assignment is the integration of your frontend webapp with a backend server that supplies data.

React and Redux

Using React for the views, refactor your three separate html files, index.html, main.html, and profile.html, into a SPA. This means index.html will contain a div for the application to mount and a script tag for the JavaScript application bundle. For navigating between the three views you may use React Router but a more simple approach is to have a "location" state variable and then selectively display a Component based on the value. For example,

```
if (location == MAIN_PAGE) {
        <Main .../>
} else if (location == PROFILE_PAGE) {
        <Profile .../>
} else {
        <Landing .../>
}
```

and "links" in your app will dispatch updates to the location state variable.

The file structure of your web app should have separate directories for each "section" of your application, see below for an example file structure. Remember to make your files small, ideally with only a single Component within, or at least a single concern.

As you being transforming your three pages into a React site, focus on the navigation of your site from page to page before implementing the details of each page. Keep the dummy data in your

views. After you have a working React site with static data, then you can implement the data flow using Redux.

Here is an example that should help you get started communicating with the dummy server.

Test Driven Development

We will exercise test driven development instead of writing the implementation of our web app first and testing later. Write Mocha unit tests for the desired behavior and execute the test suite as we develop.

Unit Tests

Every user interaction point should be validated. Most user interactions involve making an AJAX call to the server to update data. In our test environment we will not want to contact the real server, and therefore want to intercept the AJAX requests and provide our own mocked response.

For our first test we will simulate the user entering a new status message, which will be sent to the server for persistence. We completed similar task as part of an inclass exercise. Note that in the inclass exercise we were not using React. Therefore we had to insert values into the text inputs before executing the updateHeadline function. In your React+Redux application we test separately the React view and the Redux action.

We want to mock calls to fetch. If you are using Karma to run your tests, then webpack can alias isomorphic-fetch to mock-fetch. Otherwise if you use Mocha directly then we need to substitute in the runtime mock-fetch for the fetch coming from isomorphic-fetch, which is supplied by node-fetch. Here I provide you an example of doing this using mockery:

```
import { expect } from 'chai'
import mockery from 'mockery'
import fetch, { mock } from 'mock-fetch'

import * as profileActions from './profileActions'

...

let Action, actions
beforeEach(() => {
  if (mockery.enable) {
        mockery.enable({warnOnUnregistered: false, useCleanCache:true})
        mockery.registerMock('node-fetch', fetch)
        require('node-fetch')
  }
  Action = require('./actions').default
```

```
  actions = require('./actions')
})

afterEach(() => {
  if (mockery.enable) {
        mockery.deregisterMock('node-fetch')
        mockery.disable()
  }
})

it('should update the status message', (done) => {

  // the result from the mocked AJAX call
  const username = 'sep1test'
  const headline = 'A new headline!'

  mock(`${url}/headline`, {
        method: 'PUT',
        headers: {'Content-Type':'application/json'},
        json: { username, headline }
  })

  // review how complex actions work in Redux
  // updateHeadline returns a complex action
  // the complex action is called with dispatch as an argument
  // dispatch is then called with an action as an argument

  profileActions.updateHeadline('does not matter')(
        fn => fn(action => {
          expect(action).to.eql({
                headline, type: actions.UPDATE_PROFILE
          })
          done()
        }))

})
```

We use mockery to substitute mock-fetch in place of node-fetch. node-fetch is called by isomorphic-fetch. The "fetch" we use in resource comes from isomorphic-fetch.

Above we have mocked the resource calls to the dummy server and supply a hardcoded result to the client code. (The client code is updateHeadline, which is the code we are testing.)

See below for the list of unit tests that we require in this assignment and refer to the API for the return types for each service call. Instructions on getting started with the dummy server are at the top of the API page.

Behavior Implementation

After you have implemented all of the tests listed below, we need to implement the desired functionality so that the tests pass. In this way we also assure that all of the code we write is covered by our test cases. I.e., we should get high marks for code coverage with no extra effort and no need to later refactor our code so that it will be testable -- this again is a benefit of test driven development.

Once we are completed with the connections to the dummy server we can manually verify that our site works as expected because we already tested and implemented the status and followers functionality, the site should therefore be fully functional in these two areas. Perform live user testing by logging in with your account, changing your headline message, and then logging in with your test user account and verify that the headline message is indeed persistent.

For next time...

In the next assignment we will implement the missing functionality of updating data on the server along with uploading of images for articles, editing the user profile, and new user registation.

Requirements

Use Chrome as your standard supported browser. Whereas you can use any browser you like for development, your assignment will be accessed using Chrome by the grading staff and therefore it behooves you that it works.

Host your submission on Surge. Include the URL in a `README.json` file as before. The deployed version of your code on Surge may be used during grading. Therefore after you make your submission, please do not re-deploy to the same domain name until the next assignment.

Remember separation of concerns and write DRY modularized code.

Functionality

Below are the functional requirements for this assignment. All of the data mentioned below comes from the dummy server. There is no hardcoded data in your frontend. The currently logged in user's headline, profile picture, collection of articles, list of followers, and profile information all come from the dummy server.

- Landing view: User can log in with netID and three-word-passphrase, if successful then redirect to the Main view, otherwise inform user of incorrect login
- Main view: When a user logs out it redirects to the Landing view
- Main view: Link to profile page redirects to the Profile view

- Main view: headline for user is displayed
- Main view: ability to update headline for user using dummy server
- Main view: display user's profile picture
- Main view: sidebar shows followed users with profile pictures and headlines
- Main view: list of articles from server with newest article first
- Main view: each article has a list of comments displayed (you may want to show/hide them as in my solution)
- Main view: search bar filters displayed articles by author or body, but not date or post id
- Profile view: When a user logs out it redirects to the Landing view
- Profile view: Link back to main page redirects to the Main view
- Profile view: Display user's profile information including profile picture

Note that nothing else needs to work right now. I.e., we are only displaying the data from the server, the only data we have the ability to change is the user's headline. This means buttons for the other functionality still do not function.

Unit Test

Use Mocha and Karma to test your application. Note that in principle we only test "our" code and not "framework" code. Therefore we do not need to test that a Redux's dispatch functions as advertised, or that ReactDOM properly mounts Components. Instead we only want to test our specific business logic that we wrote.

- Validate actions (these are functions that dispatch actions)
    - resource should be a resource (i.e., mock a request)
    - resource should give me the http error
    - resource should be POSTable
    - should update error message (for displaying error mesage to user)
    - should update success message (for displaying success message to user)
    - should navigate (to profile, main, or landing)
- Validate Authentication (involves mocked requests)
    - should log in a user
    - should not log in an invalid user
    - should log out a user (state should be cleared)
- Validate Article actions
    - should fetch articles (mocked request)
    - should update the search keyword
- ArticlesView (component tests)
    - should render articles
    - should dispatch actions to create a new article
- Validate Profile actions (mocked requests)
    - should fetch the user's proile information
    - should update headline
- Validate reducer (no fetch requests here)

    - should initialize state
    - should state success (for displaying success message to user)
    - should state error (for displaying error message to user)
    - should set the articles
    - should set the search keyword
    - should filter displayed articles by the search keyword

Include your coverage results in your submission

Most of the above tests validate the logic code, i.e., your Redux code, not your React code. This implication is supported if we have simple React components for our view. The idea being that complex React components would require testing, and testing the virtual DOM is more difficult than testing simple functions. There are only a couple of component tests listed above, where you will want to validate that the component is properly rendered and the desired functionality is present.

What to submit

Be sure that you are not submitting or checking in node_modules, look for the .gitignore file in your repo, you can copy this one if you want.

Your submission *might* look like this:

```
.
|-- .babelrc
|-- .eslintrc
|-- .gitignore
|-- coverage
|   `-- ...
|--- dist
|    |-- bundle.js
|    |-- index.html
          `-- ....
|-- istanbul.yaml
|-- karma.conf.js
|-- mocha.opts
|-- package.json
|-- README.json
|-- src
|   |-- actions.js
|   |-- actions.spec.js
|   |-- components
|   |   |-- app.js
|   |   |-- article
|   |   |   |-- articleActions.js
|   |   |   |-- articleActions.spec.js
|   |   |   |-- article.js
```

```
|   |   |   |-- articlesView.js
|   |   |   |-- articlesView.spec.js
|   |   |   |-- comment.js
|   |   |   `-- newArticle.js
|   |   |-- auth
|   |   |   |-- authActions.js
|   |   |   |-- authActions.spec.js
|   |   |   |-- landing.js
|   |   |   |-- login.js
|   |   |   `-- register.js
|   |   |-- main
|   |   |   |-- followingActions.js
|   |   |   |-- following.js
|   |   |   |-- headline.js
|   |   |   |-- main.js
|   |   |   `-- nav.js
|   |   `-- profile
|   |       |-- avatar.js
|   |       |-- profileActions.js
|   |       |-- profileActions.spec.js
|   |       |-- profileForm.js
|   |       |-- profileForm.spec.js
|   |       `-- profile.js
|   |-- index.js
|   |-- reducers.js
|   `-- reducers.spec.js
|-- tests.webpack.js
`-- webpack.config.js
```

The important thing about the above structure is that each portion of the web app has it's own directory. We try to be modular in our design patterns. You are not required to follow this directory structure exactly, but you are required to have a modular layout. What is important is that there *is* structure and that the structure makes good extensible and maintainable sense.

Here is my example solution site https://ricebook.surge.sh