# Seed Contact Feature

## Introduction and prerequisites

To facilitate working in React, I’d like the team to consider working with me to make a Contacts Feature to get a feel for the end to end development process.

<http://ah-1064230-001.sdi.corp.bankofamerica.com:3000/zkraz53/react-16-seed-es6>

I’ll create a branch, dev-contacts

The koa backend has already been created and tested using Postman. The collection to import is here:

/doc, react-16-seed.postman\_collection.json

Note that after installing Postman, it will try to go online and it can’t. when you click the go online, an option to “work offline” will appear very briefly in the lower middle of the screen. You’ll need to be fast to click it.

A net-server backend is also being worked on.

Overall Epic:   
using the Todo sample in the seed as a template,

* Add the appropriate mocks to /store/mock; these will be referenced throughout the creation process and called with unit tests
* Create the redux store to handle the CRUDL responses from the backend for a Contact
* Create a Contacts component to display a datagrid of the List
  + Connect to the router and menu in app.jsx for navigation
* Create a Contact Component for Creating, Updating, and Deleting a single Contact
  + Connect to the router ~~and menu~~ in app.jsx for navigation
  + Add form validation via yup and react-hook-forms

## Json models

The following json models outline the shape of the data being called from the backend

Contact:

{

  "id": 2,

  "nameLast": "Smith",

  "nameFirst": "Will",

  "phone": "2234567890",

  "type": "2",

  "comments": "lorem ipsum orum"

}

Contacts:

[

{

  "id": 2,

  "nameLast": "Smith",

  "nameFirst": "Will",

  "phone": "2234567890",

  "type": "2",

   "comments": "lorem ipsum orum"

},

{

  "id": 3,

  "nameLast": "Doe",

  "nameFirst": "Jane",

  "phone": "3234567890",

  "type": "2",

   "comments": "lorem ipsum orum"

}

]

Options:

{

  "todo": "... omitted",

  "contact": {

      "type": [

          {

              "value": 1,

              "text": "Business"

          },

          {

              "value": 2,

              "text": "Personal"

          }

      ]

  }

}

# Needed objects and components

|  |  |  |
| --- | --- | --- |
| **Assigned**  **And status** | **Name** | **Description** |
|  | action-constants.js | /src/store/\_action-constants.js  Add the constant values for the actions.   An action consists of “type” and “payload”. The type needs to be a unique value so that the reducer can evaluate it and take appropriate action.   [*Note: Although you CAN add other properties to an action, 99% of the time you SHOULDN’T. this is an industry wide paradigm*] |
|  | contact\_act | /src/store/actions  Export each action individually. Unlike a component, these should individually set.  In our case, these are simple http requests that don’t require modification, so we can use redux-promise to simplify the flow of the store… when the request resolves it will be passed on to the reducer.  [*Note: also add to /actions/index.js. this simplifies the intellisense imports in the component section*] |
|  | contact\_act.test  (building unit tests is critical to maintaining the application | /src/store/actions  ideally, have the primary code displayed and the test in a pane on the right hand side. Makes it easier to work back and forth.  [*Note: this is difficult the first few times, but will pay HUGE dividends when refactoring or modifying the code. Don’t necessarily try to test EVERYTHING, especially in a component, but capture the critical functionality or add a test when a new modification is needed.*]   1. Describe the overall section 2. Use “it” to state what you’re about to add to the action 3. Create the test, it should fail 4. Write the primary code to make the test pass.   (note, for more complex items this can be a bit of back and forth 5. Lather, rinse, repeat  * You shouldn’t ever be trying to build more than one test section at a time. * Avoid creating the code without also working the test; otherwise you’re code can grow too quickly and become difficult to write the test. |
|  | contacts\_rdc | /srce/store/reducers  Uses \_rdc prefix, inner functions use  Note that this reducer handles the multiple Contact items. So both it and contact\_rdc BOTH share a bit of handling… CRUDS update the item in the array.   also note that the todo example CONVERTS the array so that the id becomes a key. This is arguably useful to avoid array syntax further on.  example:   |  |  | | --- | --- | | **Response** | **Changes to:** | | [    {      "id": 1,      "subject": "subject a",      "body": "body a",      "status": "0",      "result": "0"    },    {      "id": 2,      "subject": "subject b",      "body": "body b",      "status": 2,      "result": 2    },    {      "id": 3,      "subject": "subject c",      "body": "body c",      "status": 2,      "result": 2    }  ] | {    "items": {  **"1":** {        "id": 1,        "subject": "subject a",        "body": "body a",        "status": "0",        "result": "0"      },      "2": {        "id": 2,        "subject": "subject b",        "body": "body b",        "status": 2,        "result": 2      },      "3": {        "id": 3,        "subject": "subject c",        "body": "body c",        "status": 2,        "result": 2      }    } | |
|  | contacts\_rdc.test | Build the tests using the todos test as a template.  This test is CRITICAL. You want to be absolutely certain the reducer is working as expected before building components. |
|  | contact\_rdc | This reducer handles a single, selected item for CRUD operations |
|  | Contact\_rdc.test | Blah blah blah, I’m sure your tired of hearing me talk about unit testing |
|  | /store/reducers/index.js | Now we need to connect all this to the store, following the example.  Note the way the import uses   import foo from ‘./foo\_rdc.js  Instead of  Import {foo} from ‘./foo\_rdc.js’  The export default option in the \_rdc means we can change the naming however we like… in this case I change it so that we can take advantage of es6 shorthand when combining the reducers. In a json object, placing an object makes it do { objectname: object }  otherwise, I’d have to do this:  Import spinner\_rdc from ‘./spinner\_rdc’;  …  {    spinner: spinner\_rdc,  //etc, etc  } |
|  |  | **END OF STORE SECTION: Page Components**  Now we’re ready to build the components. I use /pages to differentiate between a general purpose component and one that is connected via the menu and router.  If there are child components that ONLY pertain to a page section rather than global, I prefer to add within the pages folder for better encapsulation. We can always refactor easily if need be:  **Example**: pages/Foo/components/Foobar.jsx  The stylesheet is always names style.css to avoid unnecessary extra naming convention; improves copy/paste of code  It’s important to build and reality check a small bit at the time.  I like to   * Create the component in the simplest fashion possible.   + Create the test for no initial data, verify it passed   + Dual pane the window in order to see the test and the code * Get the component working in menu/router (further described below) and verify it displays.   This means in real time you can observe changes * If you haven’t already, hook up the store using the boilerplate example from todo   + Use <textAreaDebug> to view the json data   + Create the test for retrieving data * Add the html code and methods, creating tests as you go. * Finally, refactor code and CLEAN UP ANY warning message in the console and browser console! Unused vars messages, etc. Keeping this clean creates a powerful tool to debug problems; multiple developers not cleaning up will result in so much noise that the console becomes useless.   Refactoring should be easy, because we have the tests to verify nothing gets broken |
|  | /app.jsx | This is a little out of order to the flow of creation, but you’ll want to hook up the component as early as possible so that it can be viewed as it is being created.  Do everything in small steps, saving and verifying constantly. There’s nothing worse than creating a large section of code and getting a million debug errors that are now unsolvable.  // this is the payoff for all the index.js work  import {    Home,   //note: JSX recognizes this is a component because of the uppercase first letter    Todos,    Todo,  } from './pages';  …            <Switch>              {/\* the order of the routes is important... /foo/:id wouldn't display if placed after /foo \*/}              <Route path="/todo/:id" component={Todo} />              <Route path="/todo" component={Todos} />              <Route path="/" component={Home} />            </Switch> |
|  | Contacts.jsx | /pages/contact/contacts.jsx  This is where we’ll look at the list of contacts in a grid view. Singular and plural “s” is used to differentiate in code and names between multiple items and a single item; I prefer not to clutter my screen with needless long naming.  Same goes in the state for mapstateToProps… I try to keep reusing item/items so that we can copy paste without needlessly modifying code for the sake of unnecessary naming.  \*.jsx is a naming convention for the IDE. Lets the IDE know that this page has react jsx on it and format appropriately.  The function uses the redux connect wrapper to set things up. It also wraps the action in a dispatch event so that you don’t have to manually handle with react hooks.  this also allows you to have an UNCONNECTED and CONNECTED component for testing:  export const Todo = (props) => {...} //unconnected  ...  // usually at bottom of page, examples use a different approach for clarity.  //connected  export **default** connect(mapStateToProps, {      //actTodo\_C,  //the backend is doing it all in update      actTodo\_R,      actTodo\_U,      //actTodo\_D, //TODO:Need to implement      actTodoClearSelected,  })(withRouter(Todo));  //understanding connect is very import, it cleans up code considerably  //and does away with manual dispatches... actions are wrapped in dispatch automatically  (explained further in /pages/contact/index.js) |
|  | Contacts.test.jsx | But seriously, this is where the rubber meets the road. Note that testing here DOES dispatch actions, but the mock store does NOT return the reducer. Each test is pretty much atomic and tests for a single condition or group of conditions, and passes the expected store state in.   therefore, as far as a unit test is concerned, you just verify an action was dispatched, a form validation was handled, a button clicked, etc. |
|  | /pages/contact/index.js | Set the index exports as discussed previously in contacts.jsx  EXAMPLE Pages/todo/Index.js  // a little sophistry to make the index use the connected component for components with store connections  import todos from './todos';  import todo from './todo';  //ensuring index uses the connected component instead of unconnected  export const Todos = todos;  export const Todo = todo;  /pages/index.js  // this approach with index ensures that the CONNECTED component is called  export \* from './home';  export \* from './todo';  //add the contacts components here  We’ll eventually add these into app.jsx for routing and that’s where the payoff of this approach occurs:  import {    Home,   //note: JSX recognizes this is a component because of the uppercase first letter    Todos,    Todo,  } from './pages';  … |
|  | contact.jsx | /pages/contact/contact.js  Using the todo example, create the display and edit page for a contact.  A good thing to note is that when the component is destroyed/unmounted, useEffect cleans up the redux store’s selected item by dispatching a clearSelected action:      useEffect(() => {          actTodo\_R(idItem);          // eslint-disable-next-line react-hooks/exhaustive-deps          return function cleanup() {              actTodoClearSelected();              // eslint-disable-next-line react-hooks/exhaustive-deps          }          // eslint-disable-next-line react-hooks/exhaustive-deps      }, []/\*this means it only runs once\*/); |
|  | Contact.test.jsx | I promise, you’ll thank yourself later. This page is probably where the majority of enhancement and modification tickets will be focused on.  Don’t drive yourself crazy checking and testing every little element, but at a minimum, test for:  it('should open with no items'…  it('should open form when data exists'  it('should validate the form'  it('should submit the form' //actionDispatched or button clicked… remember the mock store doesn’t return a refreshed reducer |

# Still here?

Awesome, you made it to the end and can get some coffee!!! Now let’s build it! We can do this training exercise in multiple ways; either assigning individual pieces to the group, or scheduling pair programming sessions as a group to build the pieces up.

If we do the group coding sessions, I’d like one person to actually do the code, while the rest of us follow along and build on their vdevs as we go. This stuff is not easily learned by just looking on… The learning curve isn’t just how to do it, it’s how to deal with all the niggling details when thinks don’t work as expected. It’s better to be in the drivers seat to get the firsthand experience.

Thank you!

