What Was Covered

That was a lot of information, but you made it! Hopefully you now have a better idea how some of the pieces fit together in a more realistic React+Redux application, with something bigger than yet another Todo list example.

Let's look back at some of the things that you should now be familiar with.

Project Planning and Setup

Create-React-App is a great tool for creating a new React project that has a good build configuration, and it keeps the build process abstracted away so you don't have to worry about it.

The Yarn package manager has an "offline mirror" feature. You can use that to save NPM package tarballs into a folder in your code repo, then commit that folder. That way, when someone else clones the repo, they get *exactly* the same package versions that you had, those packages can be installed faster, and the installations work both offline and across different OS platforms.

While CRA doesn't include Redux, we can easily set up Redux in a CRA project.

We can also use Webpack's "hot module replacement" feature to swap in new versions of files when we save changes, without having to refresh the entire page.

UI Layout and Project Structure

Semantic-UI-React is an excellent React-specific adaptation of the Semantic-UI UI toolkit. It won't be a good fit for every application, but if you're looking for an easy-to-use toolkit that looks good (or just want an alternative to Bootstrap), it's a great choice.

We can control what's being displayed in the UI by tracking "UI state"-type values like which tab is currently selected. By following the "container/presentational component" pattern, it becomes easier to change where data is coming from if you need to switch between component state, Redux, or another data store. The "presentational" component doesn't care, it just displays UI based on the props it's receiving from its parent.

There's no single "correct" way to organize folders and files in a project. However, I do prefer a "feature-first" style of structure. Ultimately, you should choose whatever approach best helps people understand where specific code lives in your project.

I'm also a fan of "absolute import" paths like `import TabBarContainer from "features/tabs/TabBarContainer"`, rather than `import TabBarContainer from "../../tabs/TabBarContainer"`. Fixing up relative paths is a pain, so I only use relative paths if they're one folder up at most.

It's fine if UI state like "active tab" is stored in a React component, but you can also put that data in Redux if you want.

Using Redux-ORM

The Redux team recommends that you store any nested or relational data in a "normalized" structure, which means using objects as lookup tables so you can easily find items by their types and IDs.

While it's fine to use "plain" JS code to manage normalized data, including utilities like Normalizr, I highly recommend using the Redux-ORM library as an abstraction layer for working with normalized data in your Redux store.

To use Redux-ORM, you extend its `Model` classes, and add declarations of both the plain data fields and how the model relates to other models, using database-style terminology. You then create a single `ORM` instance for your application, and use `orm.register()` to add Model classes. Finally, you add a reducer to act as the "database-like state" or "entities" portion of your Redux store.

Normal Redux-ORM usage involves creating a `Session` instance by calling `orm.session(state.entities)`, then accessing the `Model` classes available on the `Session`. `Model` instances act as facades over the plain JS data in the store, and the `Session` applies immutable updates internally. `QuerySets` are used to help filter collections and return the results from relational fields.

In my own app, I've written some memoized selector functions that cut down on the number of new `Session` instances created, as a small optimization. Also, it's possible to write generic "entity reducers" that work for any `Model` type, by using class names to look up the right `Model` class type and item IDs to retrieve the right instance.

I also add custom behavior to all my `Model` classes, including functions that let me serialize and deserialize plain JS data, generate new instances, cascade deletions so that related values are deleted, and copy values back and forth between different versions of the same `Model` instance.

Loading and Displaying Data

If you want to simplify the process of developing a client, you can write a mock API using functions that just return fake data in promises.

We can use the custom `parse()` deserialization functions to help load retrieved data into the Redux store. Ultimately, those just call `Model.create()` or `Model.upsert()`.

The Redux team *really* encourages the idea of independent "slice reducer" functions all responding to the same dispatched action, with each slice reducer updating its own slice of state appropriately.

We can use the Redux-ORM classes to act as "selectors", and look up the values we need from the store in a connected component's `mapState` function. That includes retrieving related model classes and including some of their values in the `mapState` result.

We can track which list item is selected by just storing the ID of the item, rather than the entire item. We then compare the selected ID with the item IDs as we're rendering the list, and highlight the selected item's row. That same value can be used for a details form to show the entire contents of the selected item.

Connected Lists, Forms, and Performance

You are encouraged to connect any components in your application, wherever it makes sense. A great example of this is connecting both a parent list component and the individual list item components. The best way to do this is have the parent list pass item IDs to the connected list item children. Each list item then uses the `ownProps` argument to `mapState` to help look up the right item from the store.

React+Redux performance is based on object reference comparisons. A connected component re-runs its `mapState` function each time an action is dispatched, and will re-render the “real” wrapped component if any of the fields returned from `mapState` are not `===` equal from what was returned last time. That means that things like using `map()`, which creates a new array reference, will cause the wrapped component to re-render each time. Also, using Immutable.js’s `toJS()` function is bad for performance, because it’s relatively expensive *and* creates new references.

You can optimize React+Redux performance by storing your data in a normalized form, using memoized selector functions, connecting more components, and making sure each component only reads out *just* the data it needs in its `mapState` function.

Controlled inputs are a very important concept to understand when using React. Using controlled inputs can take a bit more code, but it makes the data much easier to work with.

There’s several variations in how input change events are structured. It can be helpful to write a utility function that extracts the value and returns it in an object.

We can track state like “is editing” in the application, and use that to enable and disable form fields and buttons appropriately.

Form Change Handling, Data Editing, and Performance

Class components and functional components are both useful. I use functional components as a visual signal that “this component is just props in, UI out”. While you can define callback functions inside a functional component, needing to handle events is a good signal that it’s time to convert that component into a class.

When you connect forms to Redux, text fields and other similar inputs can cause too many actions to be dispatched (such as an action for every character typed). You can use a wrapper component to buffer those changes in its own component state, and only dispatch a Redux action after the user is done typing. React’s APIs like `React.Children` and `React.cloneElement()` make it possible for a component to return updated versions of the child components that were rendered inside of it.

While independent slice reducers is the most recommended approach, sometimes you need multiple “top-level” reducer functions that each work on the entire state tree rather than a specific slice. You can use a function like `reduceReducers` to call multiple reducer functions in sequence, or write your own custom logic instead. This can be useful for having multiple “feature reducers” that interact with the same slice of state, like a `state.entities` slice.

We can use Redux-ORM to create a generic “entity update” reducer that can update any model with new values based on its type and ID.

Form Draft Data Management

We can improve performance a bit by using a memoized selector to ensure that only a single Redux-ORM `Session` instance is created per update to the `state.entities` slice, rather than creating a separate `Session` instance every time any `mapState` function is re-run.

We can also add generic “delete a model” and “create a model” reducer functions, similra to the “update” reducer.

Even though those “entity CRUD actions” are generic, we can implement some specialized behavior in the other features by having reducers that listen for those same actions.

If you need to edit the attributes of some item, then it’s a good idea to make a copy of that item into a separate “draft / work-in-progress” section of your state. That way, you can edit the item while still showing the original values in another part of the UI. It also makes it easier to cancel or reset edits, or even use the work-in-progress values elsewhere.

We can implement generic logic for copying edited items from the “current entities” slice to the “editing entities” slice when we start editing, update the edited items as we go, and then clear them out when we stop editing. We can also reuse the existing low-level “entity CRUD” logic to create higher-level editing logic.

In order to save edits, we need to copy data from the “work-in-progress” slice back to the “current entities” slice. You can do that by creating separate Redux-ORM sessions for each slice, and implementing `Model` functions that

Redux-ORM sessions for each slice, and implementing `Model` functions that know how to copy values from one instance to another.

Managing Modals and Context Menus

Modal dialogs in React are normally implemented using a technique called a “portal”, which appends an extra element to the main page so that the modal floats over the rest of the UI. React 15 has an “unstable” API that implements portals, while React 16 adds an officially supported API. The `react-portal` library simplifies using portals.

The typical approach for controlling modals with Redux is to create descriptive objects that say what dialog component type should be displayed and what its props should be, and put those descriptions into the Redux store. Then, have a component that reads the descriptions, looks up the right modal components from a lookup table, and renders them using portals. You can use this technique to show multiple modals at once by storing an array of descriptions and mapping over the array to render an array of components.

Some modals need to “return” a value, like a color picker. Putting a non-serializable value like a function into the Redux store is discouraged. Instead, the code that asked for the dialog to be shown can include a pre-built “result action” as a prop for the modal, and the modal can dispatch that after it’s closed. There’s also a Redux middleware library that can return a promise when a modal is shown, and resolve the promise when the modal is closed.

Other UI components like context menus and popup toast notifications can be implemented using the exact same technique of descriptive objects in Redux + a React component to render them.

A context menu is really just a modal dialog that’s absolutely positioned on the screen. A wrapper component can be used to handle the absolute positioning of whatever context menu component is being shown.

When we show a context menu, we can use the mouse coordinates from the event to tell the context menu where it should position itself.