Front-End Development Code Flow using React-Redux-Saga for DevOps Platform Application

Structure you code so that application stays maintainable and scalable.

The first step to creating an awesome application is knowing where things are. If you created your project using the Toolbox or from a template, you will already have the folder structure created.

Want to start a new react application need to run a command ‘create-react-app’ in terminal...

Then this is the file structure created by create-react-app as of 2.1.5 :

```
my-app
 build
 node_modules
 public
  favicon.ico
  index.html
  manifest.json
 src
  App.css
  App.js
  App.test.js
  index.css
  index.js
  logo.svg
  serviceWorker.js
 .gitignore
 package.json
 README.md
```

- build folder is final, production-ready build. Running npm build or yarn build this build folder will be created.
- node_modules is where packages installed by NPM or Yarn will reside.
- public is where your static files reside. If the file is not imported by your JavaScript application *and* must maintain its file name, put it here. Files in the public directory will maintain the same file name in production, which typically means that they will be cached by your client and never downloaded again. If your file does not have a filename that matters — such as index.html, manifest.json, or robots.txt — you should put it in src instead.
- src is where your dynamic files reside.

One of the most important and agreed upon structures for a React project is to have a components directory for storing your Components. Some developers use two directories — one for stateful Components and one for stateless Components.

A single directory for Components. Keep related code as close as possible. That is where we’ll move App.js and its siblings.

```
src
 assets
 images
  logo.svg
 components
  app
  app.css
  app.js
  app.test.js
 index.css
 index.js
 service-worker.js
```

The project needs another important addition: a utilities directory.

Keep your code DRY (Don’t Repeat Yourself) by exporting repeated logic to a singular location and importing it where used. Parts of your application can now share logic without copy-pasting by placing shared logic in this utilities directory.

Now the repository looks something like this....

```
my-app
 build
 node_modules
 public
 favicon.ico
 index.html
 manifest.json
src
 assets
 images
  logo.svg
 components
  app
  app.css
  app.js
  app.test.js
 utils
 index.css
 index.js
 service-worker.js
.gitignore
package.json
README.md
```

- index.js is your entry point for importing your Component. It contains *nothing* but an export statement that points to the topmost Component at any point in time, because the topmost Component changes often during development.
- App.css is a straight-forward CSS file imported by your stateless view Component.

An additional directory I like to add is src/routes

Lastly, unit tests. You should store them alongside your tested files or keep them in separate folder for tests also fine. This allows you to easily navigate from your Component to its test file, import the tested files, and keep tests in sync with the files they are testing.

assets

This folder holds static assets for your app. They could be fonts, icon, SVG or JPEG images or any JSON file, etc. If you have too many file types, you probably want to create some subfolders to hold them.

services

This folder contains the API service files which helps your React app connects to the backend.

Now File Tree looks like this....

Src :

- Actions
- Components
- Containers
- Utilities
- Reducers
- Sagas
- Assets
- Index.js

Tests :

- Actions
- Components
- Containers
- Utilities
- Reducers
- Sagas
- AssetsTests – Jest and React-Testing-Library
- Index.spec.js

E2E Testing

Services

Store.js

Package.json Dependencies and DevDependencies

Production, Development, Test folders

Installing npm and yarn

Creating Code Coverage Reports

If you are making a Vapor application from scratch, this will show you exactly how to set it up for reusability and maintainability...

The folder and file structure creating like this.....

src /

- Component

- Action.js
- Container.js
- Comopnent.js
- Reducer.js
- Saga.js
- Test.js
- Style.css

- Utilities
- Assets
- Store

services

pakage.json

Coverage

Available Scripts

In the project directory, you can run:

npm start

Runs the app in the development mode. Open http://localhost:3000 to view it in the browser.

The page will reload if you make edits. You will also see any lint errors in the console.

npm test

Launches the test runner in the interactive watch mode. See the section about running tests for more information.

npm test -- --coverage

To check test coverage we have implemented using https://istanbul.js.org/ js library to see visually and by running this command it will create coverage folder with an index file. You will get the information from that index file on web view.

npm run build

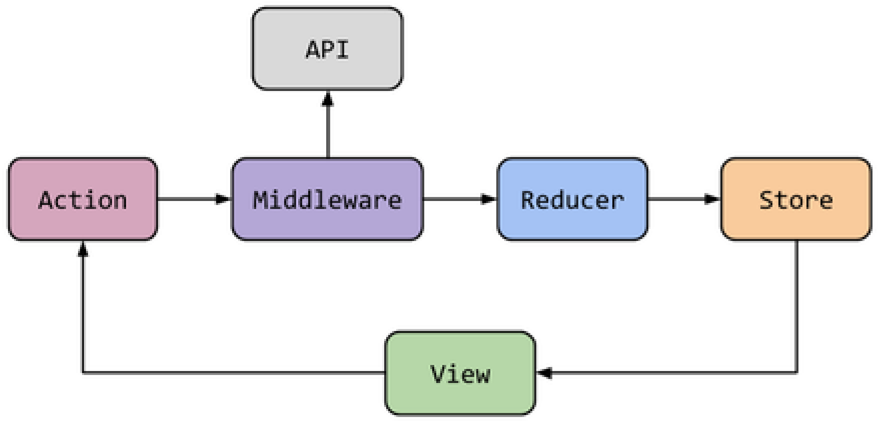
Builds the app for production to the build folder. It correctly bundles React in production mode and optimizes the build for the best performance.

The build is minified and the filenames include the hashes. See the production build section for more information.

Your app is ready to be deployed! See the section about deployment for more information about deploying your application to popular hosting providers.

Implementation of Redux Saga in React applications

Introduction



What you must understand about Redux are the following (I wont go into detail, ill keep it simple and basic, as well as relate it to your question):

1. **Actions:**These are as the name suggests, actions fired by the view/component. E.g. when you request to fetch thezipcode, this will be dispatched to the action.

```
componentDidMount(){

  this.props.fetchZipCode(85001);

}
```

action.js

```
export const fetchZipCode= (zip) => ({

  type: 'FETCH_ZIPCODE', zip,

});
```

1. **Reducers:**These handle the actions request. As described in Redux docs*'The reducer is a pure function that takes the previous state and an action, and returns the next state'*

reducer.js

```
const ZipcodeInfo = (state = 0, action) => {

switch (action.type) {
case 'REQUEST_ZIPCODE':
return {
...state
};
case 'RECEIVE_ZIPCODE':
return {
...state
zip: action.data.zipcode
};
case 'RECEIVE_ZIPCODE_FAILED':
return {
...state
}
default:
return state;
}
};
```

1. **Store:**This is what brings everything together. It holds application state, allows access to the store.

index.js (note: this is the root file of the application, so all states can be passed from the store to all components in your app)

```
import App from './App';
import { Provider } from 'react-redux';
import { createStore } from 'redux'
import ZipcodeInfo from './reducers'
```

```
const store = createStore(ZipcodeInfo)

ReactDOM.render(
<Provider store={store}>
<App />
</Provider>, document.getElementById('root'));
registerServiceWorker();
```

1. **Middleware:**Now the middleware is not necessary, but if you are making async calls you will need to use a middleware to pass information to the API etc... There a manymiddlewars to use for redux.

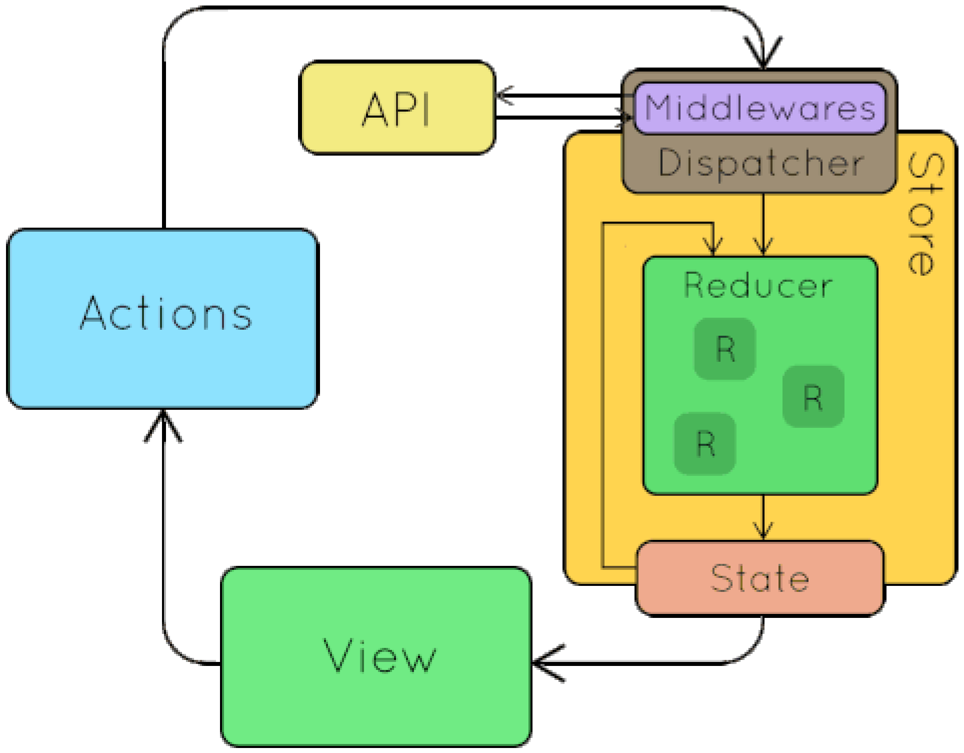
Redux-saga middleware example

```
import {call, put, takeEvery, fork} from 'redux-saga/effects';
import * as service from '../Services/api';
import * as actions from '../actions/zipcode';

//The worker: will handle the action and terminate
function* fetchZipcode(action){
try{
yield put(actions.requestZipcode());
const [zipcode] = yield [call(service.getZipcodeAPI, action.zip)];
yield put(actions.receiveZipcode(zipcode));
} catch(e){
yield put(actions.receiveZipcodeFailed(e));
}
}

//The watcher: will watch for dispatched actions and fork a worker on every
```

action

```
function* watchfetchZipcode(){
yield takeEvery('FETCH_ZIPCODE', fetchZipcode);
}

export default function* zipcodeSaga(){
yield fork(watchfetchZipcode);
}
```

I personally prefer Redux-Saga as a middleware, but it may be confusing as it uses generators(from es6) and more unknown terms.



redux-saga is a library that helps us to manage side effects (i.e. asynchronous things like ajax calls) by making it efficient to execute, simple to test and easier to handle exceptions.

It is used by attaching it as a redux middleware. So, it has access to all of redux application state and it can dispatch redux actions as well. And since it is a middleware itself, it can be triggered by dispatching normal redux actions.

Why use Redux Saga?

- It uses ES6 generators which makes asynchronous flow easier to write and understand.
- It is a better alternative to redux thunk (which, you should know if you've used before, can end up in callback hell).
- It is comparatively easier to test with little to no mocking.
- It enables serialization of logical operations.

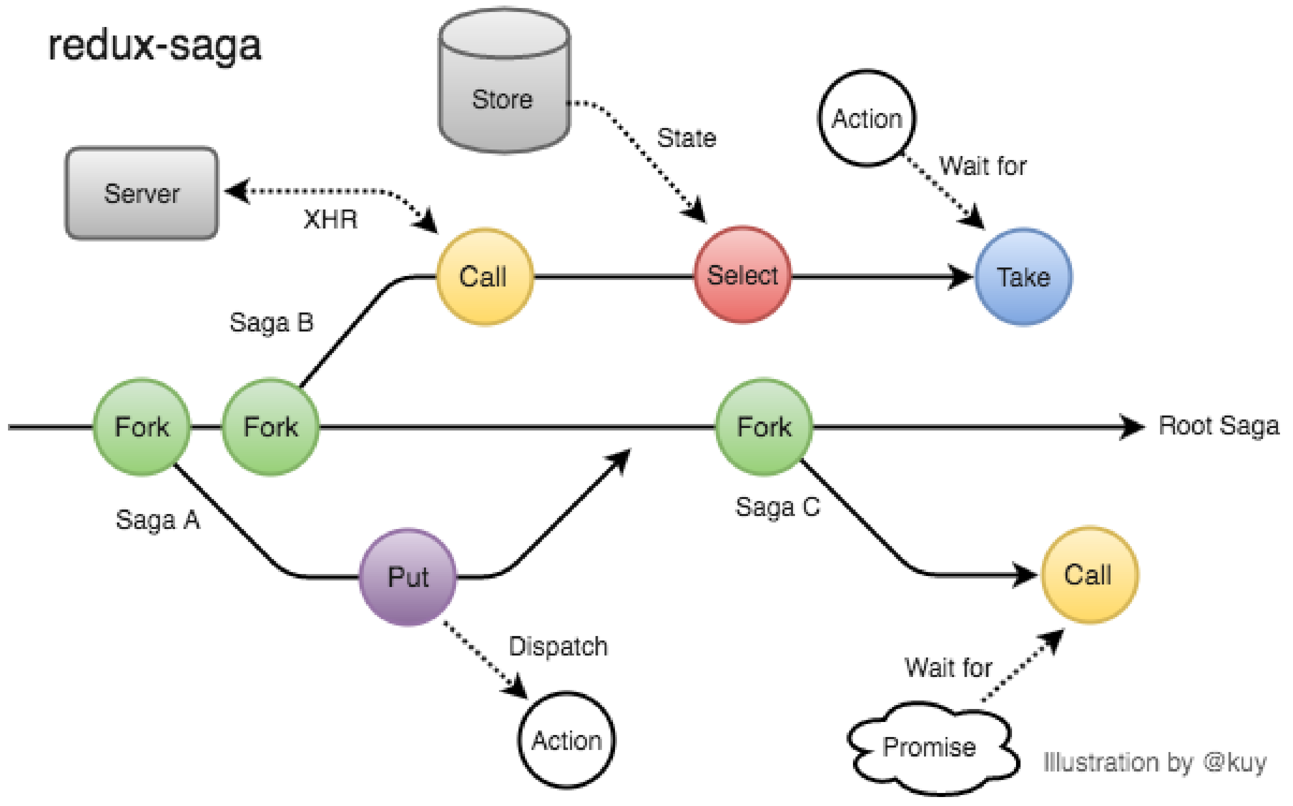
Installation

$ npm install --save redux-saga   or $ yarn add redux-saga

Usage

Add to redux as middleware

First step is to add redux-saga to reducer as a middleware. We need to import redux-saga in the file that creates the store for reducer (usually named as store.js or store/index.js). We then need to create a middleware for saga and add it to list of middlewares in the applyMiddleware() of redux. Finally, when creating the store, we need to wrap the now created middlewares inside compose() of redux and pass it as a parameter for cr eateStore().


redux-saga
Store
State
Action
Wait for
Server
XHR
Call
Select
Take
Saga B
Fork
Fork
Fork
Root Saga
Saga A
Saga C
Put
Call
Dispatch
Wait for
Action
Promise
Illustration by @kuy