React JS

Created by :
Sangeeta Joshi

Agenda

- Router
- Testing
- Redux

Router

- When we need to route between different parts of our React application, we'll probably need a router.

- It will be done with React Router.

What React Router is

- React Router helps you organise your applications based on (nested) URLs.
- This offers many advantages:

- Explicit views declarations: instantly understand what are your app views
- Restore any view + state with a simple URL
- Nested views: react-router handle nested views and their progressive resolution
- History: User can navigate backward/forward and restore state

Route components

There are 4 types of route components available with React Router:

- Route
- DefaultRoute
- NotFoundRoute
- RedirectRoute

Setting up the basics

- npm install react-router --save
- As for React components, we only need a simple high level container for our routes, say app.js.
- It only needs to contain some basic React boilerplate code.

defining routes

```
<Router>
    <Route path="/" component={Home}/>
    <Route path="/cars" component={Car}/>
    <Route path="/about"
component={About}/>
  </Router>,
```

Default route

- First load the React Core and React Router:
- Next to define what it should do when someone opens our page without adding any segments to the url.
- This is where the DefaultRoute comes in.

Parameters

<Route name="employeeDetail"
path="/employees/:empoyeeId"
component='EmpListComp')} />

</Route>

retrieve the passed employee ID using the
props.params.employeeId

Link to a Route

- React Router offers an abstraction over links called Link.
- Link allows you to leverage the routes you've defined, so you don't have to repeat them every time you want to link to a given page.
- Let's say we want to link to an employee with an ID of 1.
- This page URL looks like /employees/1.

Link to a Route

- <Route name="employeeDetail" path="/employees/:empoyeeId" component='EmpListComp ')} />

- <Link to="employeeDetail" params={employeeId: 1}>Go to employee #1</Link>

'Page Not Found' Route

```
var NotFoundRoute = Router.NotFoundRoute;
var routes = (
    <Route name="ourApp" path="/" component='------')}>
        .. your other routes ...
        <NotFoundRoute component='NotFound ')} />
    </Route>
)
```

Redirect Route

- var Redirect = Router.Redirect;

- <Redirect from='old-path' to='new-path' />

Testing with JEST

Set Up:

- Setup with Create React App –
  It is ready to use and ships with Jest!

- Setup without Create React App: if you have an existing application you'll need to install a few packages to make everything work together.
- use the babel-jest package and the react babel preset to transform code inside of the test environment.

Testing with JEST

Run:

npm install --save-dev jest babel-jest babel-preset-es2015 babel-preset-react react-test-renderer

Testing with JEST

```
// package.json
  "dependencies": {
    "react": "<current-version>",
    "react-dom": "<current-version>"
  },
  "devDependencies": {
    "babel-jest": "<current-version>",
    "babel-preset-es2015": "<current-version>",
    "babel-preset-react": "<current-version>",
    "jest": "<current-version>",
    "react-test-renderer": "<current-version>"
  },
  "scripts": {
    "test": "jest"
  }
// .babelrc
{
  "presets": ["es2015", "react"]
}
```

Testing with JEST

- Jest expects to find our tests in a __tests__ folder.
- Out of the box Jest also supports finding any .test.js and .spec.js files too

FirstTest.test.js :

```
describe('Addition', () => {
  it('knows that 2 and 2 make 4', () => {
    expect(2 + 2).toBe(4);
  });
});
```

Testing with JEST

- Jest lets us use ***describe*** and ***it*** to nest tests as we need to. (strings passed to describe and it read almost as a sentence)
- For making actual assertions, we wrap the thing we want to test within an ***expect()*** call, and then calling an assertion on it.
- In this case, we've used ***toBe. toBe*** checks that the given value matches the value under test, using === to do so.

Testing with JEST

- We can also use ***test***, which will often read better.

- ***test*** is just an alias to Jest's ***it*** function, but can sometimes make tests much easier to read and less nested.

- Demo :app_14Testing

Testing with JEST

Snapshot Testing:

Instead of rendering the graphical UI, which would require building the entire app, we can use a test renderer to quickly generate a serializable value for React tree.

Testing with JEST

Example test for a simple Link component:

```
import Link from '../Link.react';
import renderer from 'react-test-renderer';

it('renders correctly', () => {
  const tree = renderer.create(
    <Link page="http://www.facebook.com">Facebook</Link>
  ).toJSON();
  expect(tree).toMatchSnapshot();
});
```

Testing with JEST

The first time this test is run, Jest creates a snapshot file that looks like this:

```
exports[`renders correctly 1`] = `
<a
  className="normal"
  href="http://www.facebook.com"
  onMouseEnter={[Function]}
  onMouseLeave={[Function]}
>
  Facebook
</a>
`;
```

Testing with JEST

On subsequent test runs Jest will simply compare the rendered output with the previous snapshot.

If they match, the test will pass

If they don't match, either the test runner found a bug in your code that should be fixed, or the implementation has changed and the snapshot needs to be updated.

Testing with JEST

- Jest’s spy functionality to assert that functions are called with specific arguments.
- If we have the Todo component which is given two functions as properties, which it should call when the user clicks a button or performs an interaction:

In this test we’re going to assert that when the todo is clicked, the component will call the doneChange prop that it’s given.

- test('Todo calls doneChange when todo is clicked', () => {

});

Testing with JEST

- Mock functions make it easy to test the links between code by erasing the actual implementation of a function, capturing calls to the function (and the parameters passed in those calls), capturing instances of constructor functions when instantiated with new, and allowing test-time configuration of return values.

Redux-React

- You can write Redux apps with React, Angular, Ember, jQuery, or vanilla JavaScript.
- Redux works especially well with libraries like React and Deku because they let you describe UI as a function of state
- Redux emits state updates in response to actions.

Redux-React

Installing React Redux

```
npm install --save react-redux
```

Why Redux

- every single component has it’s own state
- and the only way to pass down data from parent components into child components are ***props***.
- This functions well, but it also gets extremely messy when for example we have two components that don’t have a direct parent-child relationship but they both depend on the same data.

Why Redux

- Suppose the data we need in both components comes from some API.
- To get the data to both components, we have options :
  - The two components have at least one mutual parent. We could move the API call into that component and just pass down the data as a prop until we get it into both of the components
  - We can ping the API from componentDidMount() in both components.

Redux

- React : attempt to solve problem in the view layer by removing direct DOM manipulation.
- However, ***managing the state of your data*** is left up to you.
- This is where Redux enters.
- Redux : attempts to make state mutations predictable by imposing certain restrictions on how and when updates can happen.
- These restrictions are reflected in the three principles of Redux.

Three Terms

- Store
- Actions
- Reducers

Store

- the store is a project-wide state
- Redux gives us access to this ***store***
- You can change the state from anywhere, and grab the state from anywhere.
- For a component to have access to the store though, you have to connect it.
- Every component that has access to the store becomes a “container” or a smart component.

Actions

- Actions are payloads of information that send data from your application to your store.
- They are ***the only source of information*** for the store.
- You send them to the store using store.dispatch().

Actions

- Actions are plain JavaScript objects.
- Actions must have a type property that indicates the type of action being performed.
- Types should typically be defined as string constants.

example action

(:adding a new todo item)

- { type: 'ADD_TODO', text: 'Go to swimming pool' }

More Examples :

- { type: 'TOGGLE_TODO', index: 1 }

- { type: 'SET_VISIBILITY_FILTER', filter: 'SHOW_ALL' }

Action

- Action Creators : functions that create actions.

```
function addTodo(text) {
  return {
    type: ADD_TODO,
    text
  }
}
```

- dispatch(addTodo(text))

Reducers

- Finally, to tie state and actions together, we write a function called a reducer:
- it’s just a function that takes state and action as arguments, and returns the next state of the app.

```
function visibilityFilter(state = 'SHOW_ALL', action) {
  if (action.type === 'SET_VISIBILITY_FILTER') {
    return action.filter;
  } else {
    return state;
  }
}
```

Reducers

- Reducers specify how the application's state changes in response to actions sent to the store.

(actions only describe what happened, but don't describe how the application's state changes)

Changes are made with pure functions

- To specify how the state tree is transformed by actions, you write pure reducers.
- Reducers are just pure functions that take the previous state and an action, and return the next state.
- Remember to return new state objects, instead of mutating the previous state.

Presentational and Container Components

React bindings for Redux embrace the idea of **separating presentational and container components**

Presentational Components

- Are concerned with how things look.
- usually have some DOM markup and styles of their own.
- Have no dependencies on the rest of the app, such as Flux actions or stores.
- Don’t specify how the data is loaded or mutated.
- Receive data and callbacks exclusively via props.
- Rarely have their own state (when they do, it’s UI state rather than data).
- Are written as functional components unless they need state, lifecycle hooks, or performance optimizations.

Container Components

- Are concerned with how things work.
- usually don’t have any DOM markup of their own except for some wrapping divs, and never have any styles.
- Provide the data and behavior to presentational or other container components.
- Call actions and provide these as callbacks to the presentational components.
- Are often stateful, as they tend to serve as data sources.

Three Principles

- Single source of truth:

  The State of your whole application is stored in an object tree within a single store

- State is Read Only

- Changes are made with pure functions

1.Single Source of Truth

```
console.log(store.getState())

/* Prints
{
  visibilityFilter: 'SHOW_ALL',
  todos: [
    {
      text: 'Consider using Redux',
      completed: true,
    },
    {
      text: 'Keep all state in a single tree',
      completed: false
    }
  ]
}
*/
```

2.State is Read Only

- The only way to change the state is to emit an action : an object describing what happened.
- This ensures that neither the views nor the network callbacks will ever write directly to the state.
- Instead, they express an intent to transform the state.

State

- This object is like a “model” except that there are no setters. This is so that different parts of the code can’t change the state arbitrarily, causing hard-to-reproduce bugs.

- To change something in the state, you need to dispatch an action.

Store & Actions

- Because all changes are centralized and happen one by one in a strict order, there are no subtle race conditions to watch out for.
- As actions are just plain objects, they can be logged, serialized, stored, and later replayed for debugging or testing purposes.

Actions

```
store.dispatch({
  type: 'COMPLETE_TODO',
  index: 1
})

store.dispatch({
  type: 'SET_VISIBILITY_FILTER',
  filter: 'SHOW_COMPLETED'
})
```

Reducers

```
function visibilityFilter(state = 'SHOW_ALL', action) {
  switch (action.type) {
    case 'SET_VISIBILITY_FILTER':
      return action.filter
    default:
      return state
  }
}
```

State

```
{
  todos: [{
    text: 'Eat food',
    completed: true
  }, {
    text: 'Exercise',
    completed: false
  }],
  visibilityFilter: 'SHOW_COMPLETED'
}
```

Actions

- { type: 'ADD_TODO', text: 'Go to swimming pool' }
- { type: 'TOGGLE_TODO', index: 1 }
- { type: 'SET_VISIBILITY_FILTER', filter: 'SHOW_ALL' }

Reducers

```
function todos(state = [], action) {
  switch (action.type) {
  case 'ADD_TODO':
    return state.concat([{ text: action.text, completed: false }]);
  case 'TOGGLE_TODO':
    return state.map((todo, index) =>
      action.index === index ?
        { text: todo.text, completed: !todo.completed } :
        todo
    )
  default:
    return state;
  }
}
```

Reducers

```
function todoApp(state = {}, action) {
  return {
    todos: todos(state.todos, action),
    visibilityFilter:
visibilityFilter(state.visibilityFilter, action)
  };
}
```