Table of Contents

Introduction

React is a very useful component when creating web applications. This book is an excellent guide for you to learn this JavaScript library, whether you are a beginner or an expert.

Chapter 1- Handling of Complex States

Consider a situation in which you have some projects displayed on a table. After clicking on any of the displayed projects, you should get a model which will make it possible for you to change the data of the model.

```
let t = new Baobab({
  projects: [{
    id: 0,
    title: ‘projectname’
  }, {
    id: 1,
    title: ‘bar’
  }],
  selectedProject: null
});
```

Consider the situation in which the user clicks on a project and then the data for the project is triggered:

```
import Baobab from ‘./t.js’;

let actions = {

  selectProject(index) {

    t.set(‘selectedProject’, t.get(‘projects’)[index]);

  }

};

export default actions;
```

Note that the project has been referenced as the first project in the projects array. This is a problem which might get us into trouble. This is because any change made to an object or an array will change the reference on the Baobab. With the Baobab, one can easily determine whether an object has changed or not. A clone can be created so as to solve this problem. This is shown below:

```
let Baobab = new Baobab ({

  projects: [{

    id: 0,
```

```
    title: ‘projectname’
  }, {
    id: 1,
    title: ‘bar’
  }],

  selectedProject: {
    id: 0,
    title: ‘projectname’
  }

});
```

At this point, we are having two instances of the same project. One is in the “*selectedProjects,*” and the other one in the “*projects.*” This means that changing the selected project will not be reflected in the list of projects, and the vice versa is true. A pretty and complex logic has to be created so that everything runs well.

Referencing in a List

In our above project, we have selected only one project. It is possible that you have multiple projects and there is a need for them to be shown in a table. This will mean that multiple projects will be cloned as shown below:

```
let t = new Baobab ({
  projects: [{
    id: 0,
    title: ‘projectname’
  }, {
    id: 1,
    title: ‘bar’
  }…1000],

  projectRows: [{
    id: 0,
    title: ‘projectrows’
  }, {
    id: 1,
```

title: ‘bar’

}…10]

});

We have a problem above just we had previously. Keeping in sync becomes a problem.

Reference within reference

In relational databases, referencing of any object is done by use of an ID. Suppose our projects have comments. These have to be referenced by use of an ID. This is shown below:

```
let t = new Baobab ({
  comments: [{
    id: 0,
    comment: ‘commentname’
  }, {
    id: 1,
    comment: ‘commentname’
  }],

  projects: [{
    id: 0,
    title: ‘projectname’,
    comments: [0]
  }, {
```

id: 1,

title: ‘bar’,

comments: [1]

}…1000],

projectRows: [{

id: 0,

title: ‘projectrows’,

comments: [{

id: 0,

comment: ‘commentname’

}]

}, {

id: 1,

title: ‘bar’,

comments: [{

id: 1,

comment: ‘commentname’

}]

}…10]

```
});
```

A more complex logic is needed at this point. This is because we do not need the ID of our comments, but we are in need of the comments themselves. This has to be handled during the time of cloning.

Baobab-react

A context is now being used for the purpose of passing a state to the components. This is a much friendlier way how this can be done.

The syntax itself is as follows:

```
import Baobab from ‘./Baobab.js’;
import {root, branch} from ‘Baobab-react/mixins’;
/* Let us set up the top component first. The component will need
   our “root” mixin. The root mixin will expose the state Baobab on
   our context */
let Application = React.createClass({
  mixins: [root],
  render() {
    return (
    <div>
    <Header/>
```

```
    </div>
    );
  }
});
/* Any of the child components in our application can attach the
    branch mixin. This mixin will allow you to attach cursors
    which will extract the state from our Baobab and attach it to
    our state object of our component */
let Hd = React.createClass({
  mixins: ,
  cursors: {
    foo: [‘bar’]
  },
  render() {
    return (
    <div>{this.state.name}</div>
    );
  }
});

React.render(<App Baobab={Baobab}/>, document.getElementById(‘application’))
```

Facets

Components should have the ability to use cursors for composing a description of the User Interface (UI). In this case, the facets should be used for composing the state. An example will best explain this as shown below:

```
import Baobab from ‘Baobab’;
/* A Facet is simply an object having two special properties, that is,

   “cursors” and “get” */
let selectedProject = {
  cursors: {
    id: [‘selectedProjectId’],
    projects: [‘projects’]
  },
  get: function (state) {
    return state.projects[state.id];
  }
};
/* We define our Baobab with some initial data and attach the facet */
```

```
let t = new Baobab({
  projects: {
    '0': {
    title: 'projectname'
    },
    '1': {
    title: 'bar'
    }
  },
  selectedProjectId: '0'
}, {
  facets: {
    selectedProject: selectedProject
  }
});
export default Baobab;
```

We have just begun by defining our facet which is an object having two properties.

The example given below shows how a facet can be used:

```
import React from ‘react’;

import {branch} from ‘Baobab-react/mixins’;

let P = React.createClass({

  mixins: ,

  facets: {

    p: ‘selectedProject’

  },

  render() {

    if (this.state.p) {

    return (

    <div>{this.state.p.title}</div>

    );

    } else {

    return null;

    }

  }

});
```

```
export default P;
```

As shown above, a property of facets can also be made to be available by use of the mixin. Its behavior will be similar to that of a cursor. It will grab the value which has been grabbed by our facet whenever the cursor has been updated.

Suppose an action defined as follows is triggered by a component:

```
import Baobab from ‘./t.js’;
let actions = {
  selectProject(projectId) {
    t.set(‘selectedProjectId’, projectId);
  }
};
export default actions;
```

Our facet will first react, then the project component will be notified, and the new state will be grabbed from the facet.

Solution

We need to display the projects list in a table.

The approach to this can be as shown below:

```
import Baobab from ‘t’;
let projectsList = {
  cursors: {
    ids: [‘projectsListIds’],
    projects: [‘projects’]
  },
  get: function (state) {
    return state.ids.map(function (id) {
    return state.projects[id];
    });
  }
};
let t = new Baobab({
```

```
  projects: {
    '0': {
    title: 'projectname'
    },
    '1': {
    title: 'bar'
    }
  },
  projectsListIds: []
}, {
  facets: {
    proList: proList
  }
});
export default t;
```

The component can use a facet similar to the one given below:

```
import React from 'react';
```

```
import {branch} from 'Baobab-react/mixins';

let ProList = React.createClass({

  mixins: ,

  facets: {

    projects: 'proList'

  },

  renderRow(p) {

    return (

    <tr key={p.id}>

    <td>{p.title}</td>

    </tr>

    );

  },

  render() {

    return (

    <table>

    <tbody>

    {this.state.p.map(this.renderRow)}

    </tbody>

    </table>

    );
```

```
  }
});
export default ProList;
```

An action can be implemented as follows:

```
import Baobab from ‘./t.js’;
let actions = {
  displayProjects(proIds) {
    t.set(‘proListIds’, proIds);
  }
};
export default actions;
```

We now have a complex mechanism how to handle the state and in case anything is changed, the update will be done automatically.

Reference within Reference

Suppose our users are responsible for creating the projects. The projects will only be responsible for storing the ID of the user on a property authorID. This is shown below:

```
import Baobab from 'Baobab';

let proList = {
  cursors: {
    ids: ['proListIds'],
    pro: ['projects'],
    users: ['users']
  },
  get(state) {
    return state.ids.map(function (id) {
    /* A new property is to be added to the project. The "author"
    property will have the user from our users map. We don't need

    to mutate our project object inside our Baobab, so a new
    object has to be created and hook the project object like a prototype to it. Any added properties will be part of our new object but not the actual
```

```
    project inside our Baobab, but any property can still be referenced

    from our original project object */
    let pro = Object.create(state.projects[id]);
    pro.author = state.users[project.authorId];
    return pro;
    });
  }
};
let t = new Baobab({
  pro: {},
  users: {},
  projectsListIds: []
}, {
  facets: {
    projectsList: projectsList
  }
});
export default Baobab;
```

Suppose that in our project, we have no data on the client side, and we have some project ids that we need to load. We can solve some of the challenges as follows:

```
import Baobab from ‘Baobab’;
import ProjectsListFacet from ‘./facets/ProjectsList’;
let t = new Baobab({
  projects: {},
  users: {}
  projectsListIds: [‘127’, ‘306’, ‘890’]
}, {
  facets: {
    projectsList: ProjectsListFacet
  }
});
export default t;
```

The facet can be defined as follows:

```
import projectLoader from ‘./../loaders/project.js’;

import userLoader from ‘./../loaders/user.js’;

let ProjectsList = {

  cursors: {

    ids: [‘projectsListIds’],

    pro: [‘projects’],

    users: [‘users’]

  },

  get(state) {

    return state.ids.map(function (id) {

    let pro = state.projects[id];

    if (pro) {

    proj = Object.create(project);

    } else {

    return projectLoader(id, this);

    }

    let author = state.users[project.authorId];

    if (author) {

    project.author = author;

    } else {

    project.author = userLoader(pro.author, this);
```

```
    }
    return pro;
    });
  }
};
export default ProjectsList;
```

The following is an example of a loader:

```
import ajax from ‘ajax’;
import batchCalls from ‘batchcalls’;
let getAndSetProject = batchCalls(function (ids, paths, Baobab) {
  ajax.get(‘/pro/?ids=’ + ids.join(‘,’))
    .success(function (pro) {
    ids.forEach(function (id, index) {
    t.set(paths[index], projects[id]);
    });
    })
    .error(function (err) {
```

```
    ids.forEach(function (id, index) {
    t.set(paths[index], {
    id: id,
    $err: err
    });
    });
    });
});
export default function (id, t) {
  let path = [‘projects’, id];
  let pro = Baobab.get(path) || {};
  pro.id = id;
  pro.$isLoading = true;
  getAndSetProject(pro.id, path, Baobab);
  return pro;
};
```

We need to use the facet so as to create a component. This can be done as shown below:

```
import React from ‘react’;

import {branch} from ‘Baobab-react/mixins’;

let Proj = React.createClass({

 mixins: ,

 facets: {

  proj: ‘projectsList’

 },

 /* After rendering a project, the UI states can be used for displaying what

  is happening to our data. Is loading being done, has an error been found etc.

  Expansion on the states can also be done using $notFound, $noAccess etc. */

 renderProject(proj) {

  if (proj.$error) {

  return (

  <li key={proj.id}>

  The project could not be loaded - {proj.$error}

  </li>

  );

  } else if (proj.$isLoading) {

  return (

  <li key={proj.id}>Loading the project…</li>
```

```
  );
  } else {
  return (
  <li key={proj.id}>
  {proj.id + ' - ' + proj.title}
  {this.renderAuthor(proj.author)}
  </li>
  );
  }
},

/* And we use the exact same approach with the author */
renderAuthor(author) {
  if (author.$error) {
  return (
  <small>Could not load author - {author.$error}</small>
  );
  } else if (author.$isLoading) {
  return (
  <small>Loading author…</small>
  );
```

```
    } else {
    return (
    <small>{author.name}</small>
    );
    }
  },
  render() {
    return (
    <ul>
    {this.state.projects.map(this.renderProject)}
    </ul>
    );
  }
});

export default Projects;
```

Note that with the Baobab-react, one can use the ES6 classes and ES7 decorators so as to allow one's components to grab a state. This is to be discussed in the following section.

ES6 classes

The class “*App.js*” should have the following code:

```
import {Component} from ‘react’;

import tree from ‘./t.js’;

import Projects from ‘./Pro.js’;

import {root} from ‘baobab-react/higher-order’;


class App extends Component {

  render() {

    return (

    <div>

    <Pro/>

    </div>

    );

  }

}
```

```
export default root(App, tree);
```

The class “*projects.js*” should have the following code:

```
import {Component} from ‘react’;
import {branch} from ‘baobab-react/higher-order’;
class Projects extends Component {
  renderProject(pro) {
    return (
    <li key={pro.id}>{pro.title}</li>
    );
  }
  render() {
    return (
    <ul>
    {this.props.pro.map(this.renderProject)}
    </ul>
    );
  }
```

```
}

export default branch(Pro, {

  facets: {

    pro: ‘projectsList’

  }

});
```

ES7 decorators

The class “*App.js*” should have the following code:

```
import {root} from ‘baobab-react/decorators’;
import tree from ‘./tree.js’;
import {Component} from ‘react’;
import Projects from ‘./Projects.js’;
@root(tree)
class App extends Component {
  render() {
    return (
    <div>
    <Projects/>
    </div>
    );
  }
}
export default App;
```

The class “*Projects.js*” should have the following code:

```
import {branch} from ‘baobab-react/decorators’;
import {Component} from ‘react’;
@branch({
  facets: {
    pro: ‘projectsList’
  }
})
class Proj extends Component {
  renderProject(proj) {
    return (
    <li key={proj.id}>{proj.title}</li>
    );
  }
  render() {
    return (
    <ul>
```

```
    {this.props.proj.map(this.renderProject)}
    </ul>
    );
  }
}
export default Proj;
```

So far, I hope that you are aware of how to use the Baobab-react so as to keep the state of an application. It offers a low-level solution compared to the rest of solutions. It has no assumptions on how to structure the state of our application. One is expected to create an abstraction on his own, since this is very different than the user interface itself.

In React, facets are very new components, and it is expected strategies will emerge which will take advantage of it.

Chapter 2- Creating a Scrollable Table In React

Let us begin by creating our container. This can be done using the code given below:

```
<div className=“fixed-scroll-element-container”>
  <div
    ref=“HeaderContainer”
    className=“fixed-scroll-element-header”/>
  <div
    ref=“Container”
    className=“fixed-scroll-element”
    style={containerStyle}/>
</div>
```

Our actual nodes can now give us the width, and the columns can be calculated. This is shown below:

```
// rowWidth: number — width of the row

// columnWidths: Array — array having the widths of the columns we are about to create, in which a real number will be in there if the width is available.


function getColumnWidths(rWidth, colWidths) {

  var comp = colWidths.reduce(function (agg, width) {

    if (typeof width === ‘number’) {

    agg.remainingWidth -= width;

    agg.customWidthColumns -= 1;

    }

    return agg;

  }, {

    autoSizeColumns: colWidths.length,

    remainingWidth: rWidth

  });

  var standardWidth = comp.remainingWidth / comp.autoSizeColumns;


  return colWidths.map(function (width) {

    if (width) {

    return width;

    } else {

    return standardWidth;

    }
```

```
  });
}
var contWidth = this.containerNode.offsetWidth;
var computedColumnWidths = getColumnWidths(contWidth, this.props.colWidths);
```

Now that we have configured our column widths, we can render what we are having on the screen. This is shown below:

```
dRender: function (colWidths, contWidth) {
  // this is the render for when the container has been rendered
  // and we know explicitly the container width
  var rows = this.state.visibleIndices.map((itIndex, kIndex) => {
    var top = itIndex * this.props.rowHeight;
    return this.props.rowGetter(itIndex, kIndex, top, colWidths, contWidth);

  });
  return (
    <table
    style={{height: this.props.rowCount * this.props.rowHeight}}>
    <tbody>
```

```
    {rows}
    </tbody>
    </table>
  );
},
var output = this.defRender(computedColumnWidths, contWidth);
React.render(output, this.containerNode);
```

At this point, we need to put all the above together. The scroll position has to be added inside the container. RxJS can be used for doing this as shown below:

```
setUpTable: function () {
  var contHeight = this.props.containerHeight;
  var rowHeight = this.props.rowHeight;
  var rowCount = this.props.rowCount;
  var visRows = Math.ceil(contHeight/ rowHeight);
  var gScrollTop = () => {
    return this.containerNode.scrollTop;
  };
```

```
var initScrollSubject = new Rx.ReplaySubject(1);

initScrollSubject.onNext(getScrollTop());

var scrollTopStream = initScrollSubject.merge(

  Rx.Observable.fromEvent(this.containerNode, 'scroll').map(getScrollTop)


);

var fVisibleRowStream = scrollTopStream.map(function (scrollTop) {


  return Math.floor(scrollTop / rowHeight);

}).distinctUntilChanged();

var visIndicesStream = fVisibleRowStream.map(function (firstRow) {


  var visIndices = [];

  var lRow = firstRow + visRows + 1;



  if (lRow > rowCount) {

  firstRow -= lRow - rowCount;

  }

  for (var j = 0; j <= visRows; j++) {

  visIndices.push(j + firstRow);

  }

  return visIndices;

});

this.visibleIndicesSubscription = visibleIndicesStream.subscribe((indices) => {
```

```
    this.setState({
    visIndices: indices
    });
  });
}
```

The visible indices have been calculated based on the height of your container and the number of rows which can fit in the container, and then the state of the component set depending on the number of indices to be displayed. The dynamic width also has to be taken care of. An update of the component has to be done on every window resize. This is shown in the code given below:

```
var windResizeStream = Rx.Observable.fromEvent(window, 'resize').debounce(50);
this.windowResizeSubscription = windResizeStream.subscribe(() => {
  this.forceUpdate();
});
```

At this point, the table should be ready to be used.

Chapter 3- Rendering on the Server-Side

Suppose we have a dynamic web page, and we want to render it in the form of HTML. We need to use the React-router and the Redux for doing this. The entry point for the Redux app should be as follows:

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

Our first page should have the following content:

```
import { connect } from 'react-redux';
import React, { Component, PropTypes } from 'react';
import _ from 'lodash';
import { loadQuestions } from 'actions/questions';
```

```
class MyPage extends Component {
  componentDidMount() {
    this.props.loadAnswers ();
  }
  render() {
    return (
    <p>
    <h2>Answer</h2>

    {
    _.map(this.props.answers, (k)=> {
    return (
    <p key={k.id}> { k.content }</p>
    );
    })
    }
    </p>
    );
  }
}
function mapStateToProps (state) {
```

```
  return { answers: state.questions };
}
export { Answer };
export default connect(mapStateToProps, { loadAnswers })(Answer);
```

Our second page should be as follows:

```
export const LOADED_ANSWERS = 'LOADED_ANSWERS';
export function loadAnswers() {
  return function(getState, dispatch) {
    request.get('http://localhost:3000/answers')
    .end(function(error, res) {
    if (!error) {
    dispatch({ type: LOADED_ANSWERS, response: res.body });
    }
    })
  }
}
```

We then want to use the Redux so as to render the content on the server-side. The code for the server-side should be as follows:

```
import createMemoryHistory from ‘history/lib/createMemoryHistory’;
import { RoutingContext, match } from ‘react-router’
import Express from ‘express’;
import Promise from ‘bluebird’;
let server = new Express();
server.get(‘*’, (request, resp)=> {
  let history = createMemoryHistory();
  let mystore = configureStore();
  let routes = crateRoutes(history);
  let location = createLocation(request.url)
  match({ routes, location }, (error, redLocation, renderProps) => {

    if (redirectLocation) {
    resp.redirect(301, redLocation.pathname + redLocation.search)

    } else if (error) {
    res.send(500, error.message)
    } else if (renderProps == null) {
```

```
    res.send(404, ‘Not found’)

    } else {

    let [ getCurrentUrl, unsubscribe ] = subscribeUrl();

    let requestUrl = location.pathname + location.search;

    getReduxPromise().then(()=> {

    let redState = escape(JSON.stringify(store.getState()));

    let html = ReactDOMServer.renderToString(

    <Provider store={store}>

    { <RoutingContext {…renderProps}/> }

    </Provider>

    );

    res.render(‘index’, { html, redState });

    });

    function getReduxPromise () {

    let { query, params } = renderProps;

    let computation = renderProps.components[renderProps.components.length -
1].WrappedComponent;


    let promise = computation.fetchData ?

    computation.fetchData({ query, parameters, store, history }) :


    Promise.resolve();

    return promise;
```

```
    }
    }
  });
  });
```

The view template for the server should be as follows:

```
!DOCTYPE html>
<html>
  <head>
    <title> A Redux example</title>
  </head>
  <body>
    <p id=“root”><%- html %></p>
    <script type=“text/javascript” charset=“utf-8”>
    window.__REDUX_STATE__ = ‘<%= reduxState %>’;
    </script>
    <script src=“http://localhost:3001/static/bundle.js”></script>
  </body>
```

</html>

We need to add a static method to our first page as shown below:

```
class MyPage extends Component {
  static fetchData({ store }) {
    // return the promise here
  }
  // …
}
```

Our middleware which is an application program interface (API) should be as follows:

```
import _ from ‘lodash’;
import { camelizeKeys } from ‘humps’;
import Promise from ‘bluebird’;
import superAgent from ‘superagent’;
```

```
export const CALL_API = Symbol('CALL_API');

export default store => next => action => {
  if ( ! action[CALL_API] ) {
    return next(action);
  }
  let request = action[CALL_API];
  let { getState } = store;
  let deferred = Promise.defer();
  // handle 401 and auth here
  let { method, url, successType } = request;
  superAgent[method](url)
    .end((error, resp)=> {
    if ( !error ) {
    next({
    type: successType,
    response: resp.body
    });
    if (_.isFunction(request.afterSuccess)) {
    request.afterSuccess({ getState });
    }
    }
```

```
    deferred.resolve();
    });
  return deferred.promise;
};
```

Now that we have the above, we can make a change to the original action in our first page as shown below:

```
export const LOADED_ANSWERS = 'LOADED_ANSWERS';
export function loadAnswers() {
  return {
    [CALL_API]: {
    method: 'get',
    url: 'http://localhost:3000/answers',
    successType: LOADED_ANSWERS
    }
  };
}
```

Our function in the first page will become as follows:

```
import React, { Component, PropTypes } from ‘react’;

import { connect } from ‘react-redux’;

import { loadQuestions } from ‘actions/answers’;

import _ from ‘lodash’;

class Answer extends Component {

  static fetchData({ store }) {

    return store.dispatch(loadAnswers());

  }

  //…

}
```

You will then be done. Loading of the web pages will be done faster.

Chapter 4- Creating a Chrome Extension

React is one of the best ways that one can implement the front end part for their apps. With it, each component of the UI can be implemented separately. This means that there will be a reduced complexity and understanding how the app works will be much easier.

We need to use the project structure given below for our project:

notes/

app/

coffee/

Johnlist_listing.coffee

scss/

Johnlist_listing.scss

res/

manifest.json

package.json

Brocfile.js

Package.json

This will be used for creating the project and for managing our external dependencies. Let us begin with a bare file and then add our dependencies:

{

“name”: “notes”,

“description”: “chrome extension”,

“version”: “0.0.1”,

“author”: “Nicholas”,

“license”: “ISC”,

“main”: ””,

“private”: true,

“scripts”: {},

“dependencies”: {}

}

We should begin by addition of the libraries which are to be used. React can first be installed as follows:

npm install —save-dev react react-dom

In the case of our React files, we will be using JSX and coffeeScript. There is then a need for us to come up with a mechanism on how these files will be converted into JavaScript so that the browser will be able to understand and then run them. Broccoli can be used for that purpose.

For this to be installed, navigate to your project directory and then execute the following two commands:

npm install -g broccoli-cli

npm install —save-dev broccoli

After the installation of broccoli, all of the needed libraries will have to be installed. This is the only way we can be able to process the SASS and coffeeScript files. The following command can be used:

npm install —save-dev broccoli-sass broccoli-fast-browserify broccoli-merge-trees broccoli-funnel coffee-reactify broccoli-timepiece

Manifest File

This will be the heart of our Chrome extension. It will be as follows:

{

“manifest_version”: 2,

“name”: “Johnlist Notes Extension”,

“description”: “This is the Extension to allow you to write notes about the different Johnlist listings and then view those notes on our page”,

“version”: “1.0”,

“permissions”: [

“activeTab”

],

“content_scripts”: [

{

“matches”: [“*://*.Johnlist.org/*.html”],

“css”: [“Johnlist_listing.css”],

“js”: [“Johnlist_listing.js”]

}

```
    ]
}
```

The property “*permissions”* is responsible for telling Chrome that our extension is responsible for running scripts on the Chrome tab which is currently active.

Using Broccoli to build the project

We need to make use of Brocfile.js for telling the broccoli how to build our project. Our SASS files will be compiled by the broccoli into CSS files. This is shown below:

```
// Importing some Broccoli plugins

var compSass = require('broccoli-sass');

var mergeTrees = require('broccoli-merge-trees');

var Funnel = require('broccoli-funnel');

var browser = require('broccoli-fast-browserify')

// Specifying our Sass and Coffeescript directories

var sassDirectory = 'app/scss';

var coffeeDirectory = 'app/coffee';

var manifest = 'manifest.json';

var resources = 'res'

// Telling the Broccoli how we need the assets to be compiled

var clListingStyle = compileSass(['sassDirectory], 'craigslist_listing.scss',
'craigslist_listing.css');


var scripts = browser (coffeeDirectory, {

    bundles: {
```

```
    “load_craigslist_listing.js”: {
    transform: [
    require(‘coffee-reactify’)
    ],
    entryPoints: [‘load_craigslist_listing.coffee’]
    },
    “load_craigslist.js”: {
    transform: [
    require(‘coffee-reactify’)
    ],
    entryPoints: [‘load_craigslist.coffee’]
    }
    }
});
var resourceFiles = new Funnel(resources, {
    //destDir: resources
});

// Merging the compiled styles and scripts into a single output directory.
module.exports = mergeTrees([clListingStyle, scripts, resourceFiles]);
```

Content Scripts

Our aim at this point is to create the SASS and coffeeScript files which will provide us with the actual functionality of the Chrome extension. This is shown below:

```
ReactDOM = require ‘react-dom’

React = require ‘react’

CLNotes = React.createClass({

  displayName: ‘MyNotes’

  getInitialState: ->

    notes: []

  render: ->

    <div>

    <NotesDisplay mynotes={@state.notes} />

    <NoteInput saveNote={@saveNote} />

    </div>

  saveNote: (mynote) ->

    mynotes = @state.notes

    notes.push(mynote)
```

```
    @setState
    notes: mynotes
})
NoteInput = React.createClass({
  displayName: 'NoteInput'
  getInitialState: ->
    note: ''
  render: ->
    <div>
    <input type='text' value={@state.note} onChange={@noteChanged} />
    <button onClick={@saveNote}>Save</button>
    </div>
  noteChanged: (event) ->
    mynote = event.target.value
    @setState
    note: mynote
  saveNote: ->
    @props.saveNote(@state.note)
    @setState
    note: ''
})
```

```
NotesDisplay = React.createClass({
  displayName: 'NotesDisplay'
  render: ->
    <div id='notesDisplay'>
    {
    @props.notes.map (mynote, j) ->
    <div key={j}>
    {mynote}
    </div>
    }
    </div>
})
# We are finding the 'mapAndAttr' div, and then insert our own div as its child,

# and then we will render the React component inside our new div

attributesDiv = window.document.getElementsByClassName('mapAndAttrs')?[0]

if attributesDiv
  notesDiv = document.createElement('div')
  notesDiv.id='clNotes'
  attributesDiv.appendChild(notesDiv)
  ReactDOM.render(
```

```
    <CLNotes />
    document.getElementById('clNotes')
  )
```

Note that we have started with our top level React component. This is the "CLNotes." This is then made up of two child components. One is to be used for placing new notes, and the other one for displaying the available notes.

We now need to create the file "*craigslist_listing.scss.*" It will be used for setting the size of our components, and for setting the background to grey. This is shown below:

```
#clNotes {
  width: 350px;
  height: 350px;
  background-color: lightgrey;
  padding:20px;
}
#notesDisplay {
  width:280px;
```

```
  height: 240px;
  background-color: white;
  overflow-y: scroll;
}
```

We now need to install and then run our Chrome extension. We have to use broccoli first, and then compile our files. Use the terminal inside your project directory to run the following command:

```
broccoli-timepiece dist/
```

With the above command, our project will be compiled to the directory "*/dist*" and this will automatically recompile the project in any case the project is changed in any way.

Once the execution of the above command is completed, open your Chrome browser and then navigate to the directory "chrome://extensions." Select "Developer mode" from the top right corner.

Chapter 5- Data Visualization in React.js

In this chapter, we will guide you on how to add a mouse-over interaction to a histogram. Our client-side component should be as follows:

```
‘use strict’;
var React = require(‘react/addons’);
var cx = React.addons.classSet;
var ws = require(‘ws’);
var Histogram = require(‘./transactions_by_date.jsx’);
var History = require(‘./history.jsx’);
var configuration = require(‘../transactions_by_date.js’).config;
var histogram = require(‘../transactions_by_date.js’).histogram;
module.exports = React.createClass({
  getInitialState: function() {
    return {
    histogram: hist(new ArrayBuffer(config.TOTAL_BYTES)),
    };
  },
componentDidMount: function() {
```

```
  var wsc = new ws('ws://localhost:8081');
  wsc.binaryType = 'arraybuffer';
  wsc.onmessage = function(message) {
  this.setState({ histogram: hist(message.data) });
  }.bind(this);
},
render: function() {
  return (
  <div className="container">
  <div className="row">
  <div className="col-xs-12">
  <h1> Visualization of Scalable Data </h1>
  <h2>Total Bitcoins per day </h2>
  </div>
  </div>
  <br />
  <div className="row">
  <div className="col-xs-12">
  <Histogram
  data={ this.state.histogram }
  />
```

```
    </div>
    <div className="col-xs-12">
    <History
    now={ this.state.histogram.domain[1] }
    />
    </div>
    </div>
    </div>
    );
  }
});
```

I have a web socket module and React, together with some React components. Our histogram is a live-updating bar chart, and it is a representation of the transactions by date.

Let us begin with an empty array of the right size for our histogram. This is shown below:

```
histogram: hist(new ArrayBuffer(config.TOTAL_BYTES)),
```

Consider the code given below:

```
wsc.onmessage = function(message) {
    this.setState({ histogram: hist(message.data) });
    }.bind(this);
```

With the above code, we will be in a position to set the histogram to whatever message which comes back. The server will be tasked with construction of an typed array representation of our data, and then it will be pushed to a web socket.

We are now in need of adding state to our component of the type “*hoverDate”*. This is shown below:

```
histogram: hist(new ArrayBuffer(config.TOTAL_BYTES)),
    hoverDate: null
```

Let us then add a function which will place a date on hover. This is shown below:

```
onHover: function(date) {
    this.setState({hoverDate: date});
  },
```

We want to be keeping track of the data which has been hovered on in our component. This should then be passed in as a property in the component as shown below:

```
<History
    now={ this.state.histogram.domain[1] }
    onHover={this.onHover}
    />
```

The above can then be used as follows:

```
‘use strict’;

var React = require(‘react/addons’);

var ReactCSSTransitionGroup = React.addons.CSSTransitionGroup;

var moment = require(‘moment’);

var events = [

  [new Date(2015, 0, 5), ‘Genesis block: 18:15:05 GMT’],

  [new Date(2015, 0, 7), ‘ Bitcoin v0.1 ‘],

  [new Date(2015, 0, 12), ‘ First Bitcoin transaction ‘],

];

events.reverse(); // for displaying backwards (newest on top)

module.exports = React.createClass({

  render: function() {

    return (

    <ul className=“list-group”>

    <ReactCSSTransitionGroup transName=“history-events”>

    {events.filter(function(event) {

    return evt[0].getTime() <= this.props.now;

    }.bind(this)).map(function(event) {

    var date = event[0];

    var text = event[1];
```

```
    return (
    <li className="list-group-item" key={text} >
    <span className="label label-primary">
    {moment(date).calendar()}
    </span>
    <span>{text}</span>
    </li>
    );
    }.bind(this))}
    </ReactCSSTransitionGroup>
    </ul>
    );
  }
});
```

We should now bind the on hover for our history item to the date. This is shown below:

```
<li
    className="list-group-item"
```

```
    key={text}
    onMouseOver={this.props.onHover.bind(null, date)}
>
```

The events should then be logged to ensure that everything runs well. This can be done as shown below:

```
onHover: function(date) {
    console.log(date);
    this.setState({hoverDate: date});
  },
```

After the hover, we want to select some items. We can set the color of the background. This is shown below:

```
<li
    className="list-group-item"
    key={text}
```

```
    onMouseOver={this.props.onHover.bind(null, date)}
style={{
    backgroundColor: this.props.hoverDate === date ? '#b0007f' : 'white',

    color: this.props.hoverDate === date ? 'white' : 'black'
    }}
    >
```

The hover date can then be passed as follows:

```
<History
    now={ this.state.histogram.domain[1] }
    onHover={this.onHover}
    hoverDate={this.state.hoverDate}
    />
```

The following can be used for setting the mouse hover event on the other file. This is shown below:

```
onMouseOver={this.props.onHover.bind(null, date)}
    onMouseOut={this.props.onHover.bind(null, null)}
```

We now need to implement the mechanism on how the histogram will be highlighted. This is shown below:

```
‘use strict’;
//our external deps
var React = require(‘react’);
var d3 = require(‘d3’);
var moment = require(‘moment’);
// our internal deps
var Axis = require(‘./axis.jsx’);
var Bars = require(‘./bars.jsx’);
module.exports = React.createClass({
  render: function() {
    var data = this.props.data;
    var values = data.getValues();
```

```
var width = 630;

var height = Math.floor(width/2);

if (data.domain[0] === 0) {

// we have no data

return (

<div>

<span>Loading… </span>

<span className=“glyphicon glyphicon-time” aria-hidden=“true”></span>


</div>

);

}

var scales = {

x: d3.scale.linear().domain([0, data.getBin(data.domain[1])]).range([0, width]),

y: d3.scale.linear().domain([0, d3.max(values)]).range([0, height])


};

return (

<div className={‘histogram ‘ + this.props.className}>

<svg

width={width + 100}

height={height + 100}
```

```
>

<Axis

scale={scales.x}

displayScale={

d3.time.scale.utc()

.domain([new Date(data.domain[0]), new Date(data.domain[1])])


.range([0, data.getBin(data.domain[1])])

}

tickFormatter={

function(date) { return moment(date).format('MMM YY'); }


}

x={100}

y={height+1}

axis='x'

/>

<Axis

scale={scales.y}

displayScale={

d3.scale.linear().domain([d3.max(values), 0]).range([0, d3.max(values)])


}
```

```
    tFormatter={
    function(t) { return t.toLocaleString(); }
    }
    x={99}
    y={0}
    axis='y'
    />
    <Bars
    data={data}
    width={width}
    height={height}
    scales={scales}
    zmIn={this.zoomIn}
    />
    </svg>
    </div>
    );
  }
});
```

We now have some axes and a set of bars in our component. Our aim is to highlight the bars:

```
<Bars
data={data}
width={width}
height={height}
scales={scales}
zmIn={this.zoomIn}
/>
```

The class for the bars is as follows:

```
‘use strict’;
// our external deps
var React = require(‘react’);
// the utility functions
function regularArray(tpdArray) {
  // TODO — figuring out how to map over the typed arrays cleanly
```

```
  // without the need to copy to the regular array
  var array = new Array(tpdArray.length);
  for (var j=0; j<typedArray.length; j++) {
    array[j] = tpdArray[j];
  }
  return array;
}
function translate(x, y) {
  return 'translate(' + x + 'px,' + y + 'px)';
}
module.exports = React.createClass({
  render: function() {
    var data = this.props.data;
    var values = regularArray(data.getValues());
    return (
    <g style={{transform: 'translateX(100px)'}}>
    {values.map(function(value, idx) {
    var click = null;
    if (data.bucket !== 0 &&
    idx === 0) {
    // making our first bar clickable, so as to dive into our results there
```

```
click = this.props.zmIn;
}
var scWidth = (this.props.width/(data.getBin(data.domain[1]))) + 0.5;

return (
<rect
fill='#0a8cc4'
key={idx}
x={0}
y={0}
width={1} /* sizing with the CSS transform for allow transitions */

height={1} /* sizing with the CSS transform so as to allow transitions */

style={{
cursor: click === null ? 'auto' : 'pointer',
transform: translate(
this.props.scales.x(idx),
this.props.height - this.props.scales.y(value)
) + ' scaleY(' + this.props.scales.y(value) + ')'
+ ' scaleX(' + scaleWidth + ')'
}}
onClick={click}
```

```
      />

      );

      }.bind(this))}

      </g>

      );

    }

});
```

Plumbing the data

Our debugger should be as follows:

var scales = {

x: d3.scale.linear().domain([0, data.getBin(data.domain[1])]).range([0, width]),

y: d3.scale.linear().domain([0, d3.max(values)]).range([0, height])

};

debugger;

return (

…

The highlight index should be as shown below:

<Histogram

data={ this.state.histogram }

highlightIdx={

```
    this.state.hoverDate === null ? null : (

    Math.floor((this.state.hoverDate - (new Date(this.state.histogram.domain[0]))) /
(1000 * 60 * 60 * 24))


    )

    }

    />
```

The “*highlightIdx*” should then be passed through our bars as follows:

```
var hl = this.props.highlightIdx !== null && Math.abs(this.props.highlightIdx - idx)
<= 1;


    <Bars

    data={data}

    width={width}

    height={height}

    scales={scales}

    zmIn={this.zoomIn}

    highlightIdx={this.props.highlightIdx}

    />
```

We now want to ensure that our bars are highlighted on hover. Just restart the server, and then reload your page. The content should be displayed on the window.

Chapter 6- Showing Exceptions

It is possible for us to show the exceptions from a flux dispatcher callback. Flux is simply a front end application architecture created by Facebook. The method *"register"* is used for accepting the callbacks. The callbacks will then be invoked whenever an action has been to it. The dispatcher usually eats exceptions which occur in a callback. This is good because a callback which fails cannot crash the whole application. However, it makes the process of debugging tough.

We are in need of showing our exceptions on the console. We should first define a function for logging errors of the function which is being passed. This should be done as follows:

```
var vm = function(f) {
  return function() {
    try {
    f.apply(this, arguments);
    } catch(ex) {
    console.error(ex.stack);
    }
```

```
  }
};
```

Note that we have used the console object, and this will fail in case you use older versions of Internet Explorer (IE). You should always have your developer console already opened. Let us now override the "*dispatch*" method so as to use the above class:

```
var Dispatcher = require('flux').Dispatcher;
var as = require('object-assign');
var ApplicationDispatcher = as(new Dispatcher(), {
  handleViewAction: function(action) {
    console.log(action);
    this.dispatch({
    source: 'VIEW_ACTION',
    action: action
    });
  },
  handleServerAction: function(action) {
    console.log(action);
    this.dispatch({
```

```
    source: 'SERVER_ACTION',
    action: action
    });
  },
  register: function(fn) {
    return Dispatcher.prototype.register.call(this, vm(fn));
  }
});
module.exports = ApplicationDispatcher;
```

Chapter 7- Testing React Components

Karma and webpack can be used for the purpose of testing React components. To do this, you have to begin by installing the necessary libraries; otherwise, it will not work effectively.

Configuration

Karma will read from the file "*karma.conf.js,*, so we should begin by setting it up. This is shown below:

```
var pack = require('webpack');
module.exports = function (config) {
  config.set({
    browsers: [ 'Chrome' ], //running in Chrome
    singleRun: true, //should be run once by default
    frameworks: [ 'mocha' ], //using the mocha test framework
    files: [
    'tests.webpack.js' // loading this file
```

```
    ],

    preprocessors: {

    'tests.webpack.js': [ 'webpack', 'sourcemap' ] //preprocessing with the webpack and the sourcemap loader


    },

    reporters: [ 'dots' ], //report results in this format

    webpack: { // copying of the webpack config

    devtool: 'inline-source-map', // doing inline source maps instead of default


    module: {

    loaders: [

    { test: /.js$/, loader: 'babel-loader' }

    ]

    }

    },

    webpackServer: {

    noInfo: true // avoid spamming the console if running in karma

    }

  });

};
```

Our next step should be to create a single file which will make use of the API webpack required for finding the files which are needed automatically. This is shown below:

```
var c = require.context('./src', true, /-test.js$/); // ensure that you have the directory and regex test being set correctly

c.keys().forEach(c);
```

We now need to write some tests. An example of this is shown below:

```
var React = require('react');

var TtUtils = require('react/lib/ReactTestUtils'); //It is good to use Test Utils, but one can use the DOM API instead.

var expect = require('expect');

var Root = require('../root'); // live testing is done in components/__tests__/, as it is what I need in my components

describe('root', function () {

  it('rendering with no problems', function () {

    var root = TtUtils.renderIntoDocument(<Root/>);

    expect(root).toExist();
```

```
  });

});
```

Chapter 8- Building a Sliding Menu

One can use React and LESS CSS to create a sliding menu. Our aim is for the menu to slide from either the left or the right based on an action of the external page.

In React, there are three classes which can be used to render the menus. These include the following:

- App
- Menu
- MenuItem

App

This can be used as shown below:

```
var Application  = React.createClass({
    showLeft: function() {
    this.refs.left.show();
    },
    showRight: function() {
    this.refs.right.show();
    },
    render: function() {
    return <div>
    <button onClick={this.showLeft}>View Left Menu!</button>

    <button onClick={this.showRight}>View Right Menu!</button>

    <Menu ref="left" alignment="left">
    <MenuItem hash="first-page"> Page 1 </MenuItem>
    <MenuItem hash="second-page"> Page 2 </MenuItem>
```

```
    <MenuItem hash="third-page"> Page 3 </MenuItem>
    </Menu>
    <Menu ref="right" alignment="right">
    <MenuItem hash="first-page"> Page 1 </MenuItem>
    <MenuItem hash="second-page"> Page 2 </MenuItem>
    <MenuItem hash="third-page"> Page 3 </MenuItem>
    </Menu>
    </div>;
    }
});
```

In the above example, we have added only menus to our app. One of these will slide it from the right and the other one from the left. The specification of these parameters has been done using the prop "*alignment.*"

We have created instances of the "*MenuItem*" class inside the "*Menus.*" The user will be able to click on these menu entries so as to perform a certain action. In our case, the action is most probably to navigate to a certain hash, and the specification of this is done using the "*hash*" prop on the declaration of the class.

Menu

This can be used as shown below:

```
var Menu = React.createClass({
    getInitialState: function() {
    return {
    visible: false
    };
    },
    show: function() {
    this.setState({ visible: true });
    document.addEventListener(“click”, this.hide.bind(this));
    },
    hide: function() {
    document.removeEventListener(“click”, this.hide.bind(this));

    this.setState({ visible: false });
    },
    render: function() {
```

```
    return <div className="menu">

    <div className={(this.state.visible ? "visible " : "") + this.props.alignment}>
{this.props.children}</div>


    </div>;

    }

});
```

The class "*Menu*" is responsible for rendering our actual menu and toggling of the visibility of our actual menu depending on how the methods "*hide*" and "*show*" are called. The method "*getInitialState*" is an indication that our menu is hidden by default. The method "*show*" can be used for setting the visibility of the menu to true.

MenuItem

This can be used as shown below:

```
var MnItem = React.createClass({

    navigate: function(hash) {

    window.location.hash = hash;

    },

    render: function() {

    return <div className=“menu-item” onClick={this.navigate.bind(this, this.props.hash)}>{this.props.children}</div>;


    }

});
```

The class “*MenuItem”* is responsible for rendering the item and then handling the action for click.

CSS

This can be used as shown below:

```
.menu {
    display:block;
    @menu-width:250px;
    >div {
    position:absolute;
    top:0;
      z-index:2;
    width:@menu-width;
    .border-box;
height:100%;
    .transition(-webkit-transform ease 250ms);
    .transition(transform ease 250ms);
    &.left {
    background:#284D7A;
    left:@menu-width*-1;
```

```
}
&.visible.left {
.transform(translate3d(@menu-width, 0, 0));
}
&.right {
right:@menu-width*-1;
background:#6B2828;
}
&.visible.right {
.transform(translate3d(@menu-width*-1, 0, 0));
}

>.menu-item {
float:left;
width:100%;
margin:0;
padding:15px 20px;
border-bottom:solid 1px #555;
cursor:pointer;
.border-box;
color:#F0B0B0;
```

```
    &:hover {
    color:#F0F0F0;
    }
    }
    }
}
```

We are in need of abstracting any vendor prefixes which are annoying. This is why we have used different LESS CSS mixins.

Chapter 9- Creating a Dropdown

We can use Font Awesome, LESS CSS, and React JS to create a dropdown. It uses the LESS CSS so as to provide a good and amazing animation.

React

This can be implemented as shown below:

```
var Dropdown = React.createClass({
    getInitialState: function() {
    return {
    listVisible: false
    };
    },
    select: function(item) {
    this.props.selected = item;
    },
    show: function() {
```

```
this.setState({ listVisible: true });

document.addEventListener("click", this.hide);

},



hide: function() {

this.setState({ listVisible: false });

document.removeEventListener("click", this.hide);

},

render: function() {

return <div className={"dropdown-container" + (this.state.listVisible ? " show" : "")}>


<div className={"dropdown-display" + (this.state.listVisible ? " clicked": "")} onClick={this.show}>


<span style={{ color: this.props.selected.hex }}>{this.props.selected.name}</span>


<i className="fa fa-angle-down"></i>

</div>

<div className="dropdown-list">

<div>

{this.renderListItems()}

</div>

</div>
```

```
    </div>;
    },

    renderListItems: function() {
    var items = [];
    for (var j = 0; j < this.props.list.length; j++) {
    var it = this.props.list[j];
    it.push(<div onClick={this.select.bind(null, it)}>
    <span style={{ color: it.hex }}>{it.name}</span>
    </div>);
    }
    return its;
    }
});

var colours = [{
    name: “Blue”,
    hex: “#1B66F2”
}, {
    name: “Red”,
    hex: “#F21B1B”
```

```
    }, {
    name: “Green”,
    hex: “#07BA16”
}];

React.render(<Dropdown list={colours} selected={colours[0]} />,
document.getElementById(“container”));
```

We have begun creation of a DropDown class by invocation of the class “createClass.” The function “getInitialState” is used for setting the state of our dropdown once it has been created, and it has taken two props,namely **list** and **selected.** The “*list”* specifies the items which are to be displayed once the user clicks on the drop down. The “*selected”* property specifies the item which is currently selected.

The React “*render”* method is then used for rendering the dropdown. Note that this will have the colors which we have specified.

LESS CSS

This can be used as shown below:

```
@height:50px;
@spacing:20px;
@select-colour:#2775C7;
@font-size:15px;
@border-colour:#DDD;
div.dropdown-container {
    &.show>div.dropdown-list {
    .transform(scale(1, 1));
    }
    @icon-width:15px;
    >div.dropdown-display {
    float:left;
    width:100%;
    background:white;
    height:@height;
```

```
cursor:pointer;
border:solid 2px @border-colour;
.border-box;
>* {
float:left;
height:100%;
.vertical-centre(@height);
}
>span {
font-size:@font-size;
width:100%;
position:relative;
.border-box;
padding-right:@icon-width+@spacing*2;
padding-left:@spacing;
}
>j {
position:relative;
width:@icon-width;
margin-left:(@spacing+@icon-width)*-1;
font-size:1.125em;
```

```
font-weight:bold;

padding-right:@spacing;

text-align:right;

}

}

>div.dropdown-list {

float:left;

width:100%;

position:relative;

width:100%;

.transform(scale(1, 0));

.transition(-webkit-transform ease 250ms);

.transition(transform ease 250ms);

>div {

position:absolute;

width:100%;

z-index:2;

cursor:pointer;

background:white;



>div {
```

```
float:left;

width:100%;

padding:0 @spacing;

font-size:@font-size;

.border-box;

border:solid 2px @border-colour;

border-top:none;


@icon-width:21px;


&:hover {

background:#F0F0F0;

}


&.selected {

background:@select-colour;

color:white;

}


>* {

.vertical-centre(@height);
```

```
    }


    >span {
    float:left;
    width:100%;
    }
    }
    }
    }
}
```

In the section for CSS, we have just set up how the dropdown will look and feel.

Chapter 10- React JS and Flux Architecture

The two can be combined together for the purpose of making a simple application. Flux provides you with a mechanism on how you can draw data from the server API. During this time, a strict decoupling of the components is kept on the client side. Separation of concerns is implemented, since a one-way communication protocol is made use of. React JS and Flux can be used together very well.

React JS

React usually functions off of classes. For some HTML to be rendered, you have to begin by creating a React class, and this will give a description of what to render. Consider a sample of React class which is given below:

```
var Hello = React.createClass({
   render: function() {
   return <div>Hello, there!</div>;
   }
});
React.render(new Hello (), document.body);
```

Note that we have defined a React class and given it the name "*Hello.*" We have then specified the "*render*" function in this class. In case we are in need of rendering our HTML inside, we will call this function, The JSX will help us in writing HTML inside JavaScript files without having to write them in separate templates. The last line in the above code shows that we are in need of rendering our class in the body of the document.

Props

Suppose we are in need of passing some data into our React classes. This one has not been provided in our previous class. The following class demonstrates how this can be done:

```
var Hello = React.createClass({
    render: function() {
    return <div>Hello, {this.props.name}</div>;
    }
});

React.render(new Hello ({ name: “John Joel” }), document.body);
```

The argument “*props”* has been introduced in our above class. This will allow us to pass data to the view of our class. Changes made to a prop variable will cause the appropriate portion of the view to be rendered. The changes can be from the parent view, the view itself or from any of the child views. With this, data can be passed within the views themselves and also kept in sync.

State

This can be used as shown below:

```
var Hello = React.createClass({
    getInitialState: function() {
    return {
    counter: 0
    };
    },

    increment: function() {
    this.setState({ counter: this.state.counter+1 });
    },

    render: function() {
    return <div>
    <div>{this.state.counter}</div>
    <button onClick={this.increment}>Increment</button>
```

```
    </div>;
    }
});

React.render(new Hello (), document.body);
```

A React class has a state, and this is what we have introduced above. This makes it possible for you to keep track of the changes made to your internal view. Whenever we are using an internal state, the function "*getInitialState*" will be required. It is an indication to the internal view of how our initial state should be. Even if you have an empty state, make sure that you make use of this function.

We make use of props when we want to pass data between our parent and child React classes, and in case a change is made to any prop, the view will be rendered again automatically. This includes both the child and the parent. If your data is only relevant to the view, make use of a state. In this case, any changes made will also be rendered in the view. If your data is pertinent to your class other than the view, feel free to make use of a private variable, but personally, I will use a state.

Nested Views

The concept of nesting views makes it possible and easy for us to make use of React. With this, we are able to render React classes from within the other React classes. An example of this is given below:

```
var FnButton = React.createClass({
    render: function() {
    return <button onClick={this.props.onClick}>
    <i className={“fa ” + this.props.icon}></i>
    <span>{this.props.text}</span>
    </button>
    }
});

var Hello = React.createClass({
    getInitialState: function() {
    return {
    counter: 0
    };
```

```
    },


    increment: function() {

    this.setState({ counter: this.state.counter++ });

    },


    render: function() {

    return <div>

    <div>{this.state.counter}</div>

    <FancyButton text=“Increment” icon=“fa-arrow-circle-o-up” onClick=
{this.increment} />


    </div>;

    }

});
```

In the above class, the button has been abstracted so as to increase the counter into a React class which is separate. That is how we can assign props via nested classes.

In Flux, we have to use a store for the purpose of retrieving, updating, and creating in case an action for doing so is received.

Consider the example given below which shows how flux works:

```
var Count = React.createClass({
    getInitialState: function() {
    return {
    items: []
    };
    },

    componentWillMount: function() {
    emitter.on("store-changed", function(its) {
    this.setState({ items: its });
    }.bind(this));
    },

    componentDidMount: function() {
    dispatcher.dispatch({ type: "get-all-items" });
    },
```

```
    render: function() {
    var its = this.state.its;
    return <div>{its.length}</div>;
    }
});

var Store = function() {
    dispatcher.register(function(pd) {
    switch (pd.type) {
    case: “get-all-items”:
    this._all();
    break;
    }
    }.bind(this));

    this._all = function() {
    $.get(“/some/url”, function(its) {
    this._notify(its);
    }.bind(this));
    }
```

```
    this._notify = function(its) {
    emitter.emit(“store-changed”, its);
    });
};

var ItemStore = new Store();
```

Our application has only a single page which is shown below:

```
“use strict”;
var Todo = React.createClass({
    getInitialState: function() {
    return {
    todos: []
    }
    },

    componentWillMount: function() {
```

```
    emitter.on(constants.changed, function(tds) {
    this.setState({ todos: tds });
    }.bind(this));
    },

    componentDidMount: function() {
    dispatcher.dispatch({ type: constants.all });
    },

    componentsWillUnmount: function() {
    emitter.off(constants.all);
    },

    create: function() {
    this.refs.create.show();
    },

    renderList: function(complete) {
    return <List tds={_.filter(this.state.tds, function(y) { return y.isComplete ===
complete; })} />;

    },
```

```
render: function() {

return <div className="container">

<div className="row">

<div className="col-md-8">

<h1>Todo List</h1>

</div>

<div className="col-md-4">

<button type="button" className="btn btn-primary pull-right spacing-top" onClick={this.create}>New Task</button>


</div>

</div>



<div className="row">

<div className="col-md-6">

<h2 className="spacing-bottom">Incomplete</h2>

{this.renderList(false)}

</div>

<div className="col-md-6">

<h2 className="spacing-bottom">Complete</h2>

{this.renderList(true)}
```

```
    </div>

    </div>


    <Modal ref="create" />

    </div>;

    }

});
```

List

This should be as follows:

```
var L = React.createClass({
    renderItems: function() {
    return _.map(this.props.todos, function(td) {
    return <Item todo={td} />;
    });
    },

    render: function() {
    return <ul className=“list-group”>
    {this.renderItems()}
    </ul>;
    }
});
```