React Notes

01 Intro

001 Why Should I Learn React

- Small learning curve
- Agile & fast
- JSX
- Great community

03 Getting Setup

006 Creating Your Web Server

```
npm init
npm install express@4 --save
# vi server.js
# vi public/index.html (Hello World)
node server.js
```

007 Hello React

Create application page structure:

```
<head>
  <meta charset="UTF-8"/>
  <script src="https://cdnjs.cloudflare.com/ajax/libs/babel-core/5.8.23/browser.min.js" />
  <script src="https://cdnjs.cloudflare.com/ajax/libs/react/15.3.2/react.js" />
  <script src="https://cdnjs.cloudflare.com/ajax/libs/react/15.3.2/react-dom.js" />
</head>
```

`babel-core/5.8.23/browser.min.js` is for just-in-time compilation for JSX & ES6 features. The following section will be translated by babel:

```
<script type="text/babel" src="app.jsx"></script>
```

009 Your First React Component

React components are the building blocks for your web app UI.

Boilterplate to create a component:

```
// React.createClass function defines Components. It's just a Class
definition
var ComponentName = React.createClass({ ... })
```

The `React.createClass` method accepts a option object, where you can override predefined component behaviors.

Example:

```
var Greeter = React.createClass({
  render: function () {
    return (
      <div>
        <h1>Hello React!</h1>
        <p>This is form a component!</p>
      </div>
    );
  }
});

ReactDOM.render(
  <Greeter/>, {/* Instance of the Greeter component. Similar to new
Greeter() in Java */}
  document.getElementById('app')
);
```

010 Learning JSX

Go to Babel for JSX translation.

011 Component Properties

`props`, which is short for properties, is a way to pass data into the component when it is started.

- Think them as parameters passed to constructor.
- From HTML element perspective, `props` are element attributes

```
var Greeter = React.createClass({
  render: function () {
    var name = this.props.name;

    return (
      <div>
        {/* Use {} to interpolate JavaScript code to JSX code */}
        <h1>Hello {name}!</h1>
      </div>
    );
  }
});

ReactDOM.render(
  <Greeter name='Andrew'/>,
  document.getElementById('app')
);
```

What if the `name` property to `Greeter` is left unspecified? To fix this, we have to provide default `props` using `getDefaultProps`.

```
var Greeter = React.createClass({
  getDefaultProps: function () {
    return {
      name: "React"
    }
  },
  render: function () {
    var name = this.props.name;

    return (
      <div>
        <h1>Hello {name}!</h1>
      </div>
    );
  }
});

ReactDOM.render(
  <Greeter />,
  document.getElementById('app')
);
```

012 User Events Callbacks

Understanding `ref`:

```
var Greeter = React.createClass({
  // ...
  onButtonClick: function (e) {
    // SPA idea!
    e.preventDefault();
    // Reference to the input HTML element named "name"
    var name = this.refs.name.value;
    alert(name);
  },
  render: function () {
    var name = this.props.name;
    var message = this.props.message;

    return (
      <div>
        <h1>Hello {name}!</h1>
        <p>{message + '!!'}</p>

        <form onSubmit={this.onButtonClick}>
          <input type="text" ref="name"/>
          <button>Set Name</button>
        </form>
      </div>
    );
  }
});
```

013 Component State

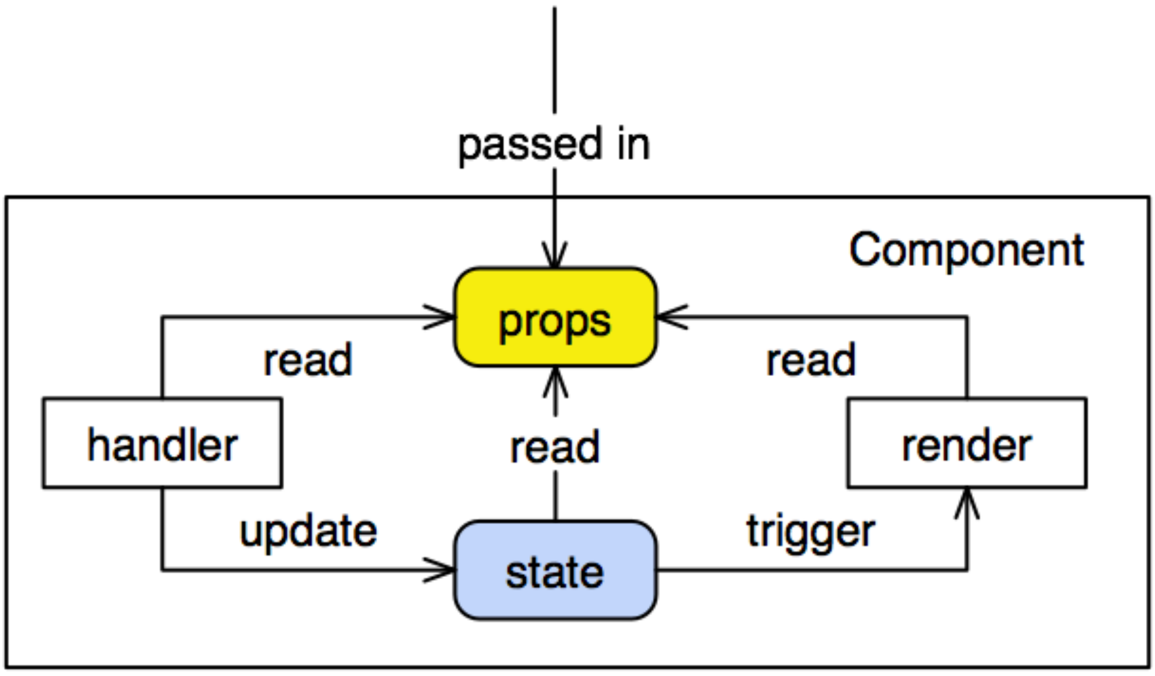
There are 2 types of data in a component:

- props: get passed into a component as you **initialize** it
- state: **internally** maintained and updated by the component

A Component shouldn't update its own props but is allowed to update its own state. To kick things off:

```
var Greeter = React.createClass({
  // ...

  // Similar to getDefaultProps function
  getInitialState: function () {
    return {
      name: this.props.name
    };
  },
  onButtonClick: function (e) {
    // ...
    if (typeof name === 'string' && name.length > 0) {
      // Update state, trigger render function
      this.setState({
        name: name
      });
      // The following line does NOT work, and will lead to unpredicable
result, you SHOULD'T do this!!!
      // this.state.name = name
    }
  },
  render: function () {
    // Get current state
    var name = this.state.name;
    // ...
  }
}
```

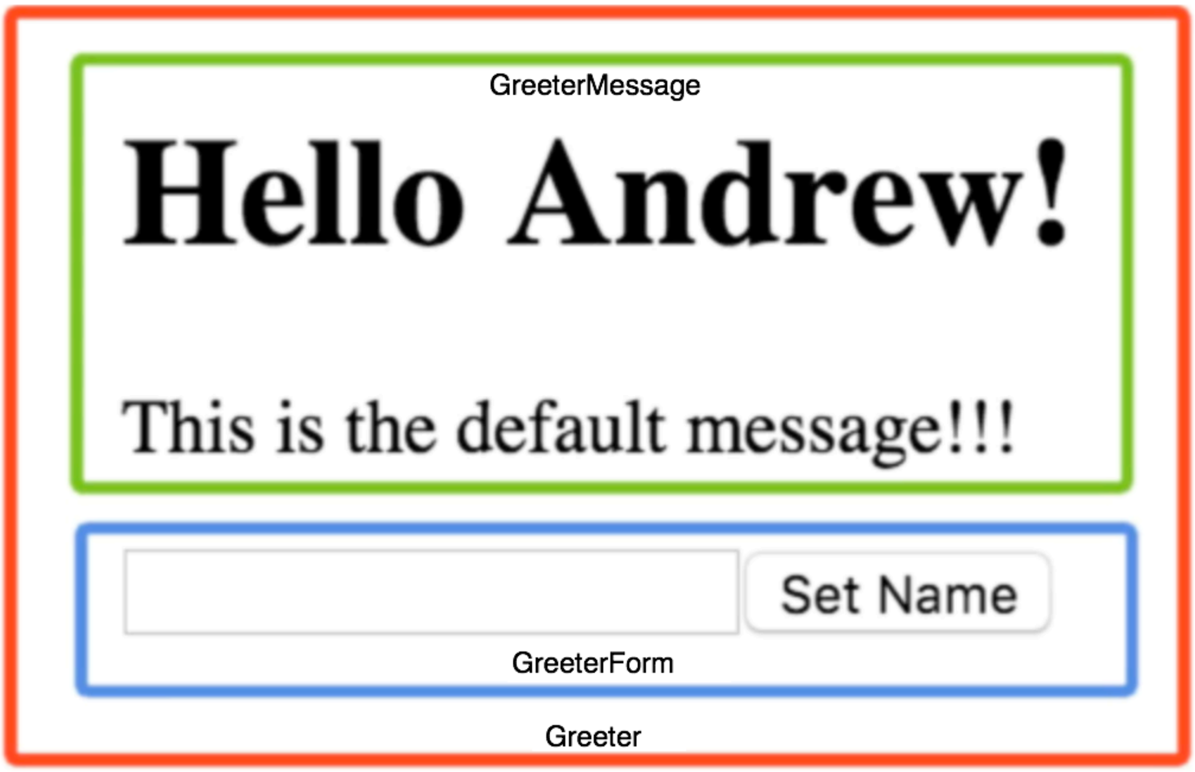


015 Nested Components Part 2

Every React component should be responsible for one thing and one thing only.

- **Container component**: A component that maintains state & renders child components. It should't do much rendering but render its children.
- **Presentational component**: A component that uses props to display information. It doesn't maintain state, kind of dummy component, but simply renders stuff to the browser

Single responsibility principle - By breaking things to smaller testable components, we can create better apps that have more functionality but without more complex code.



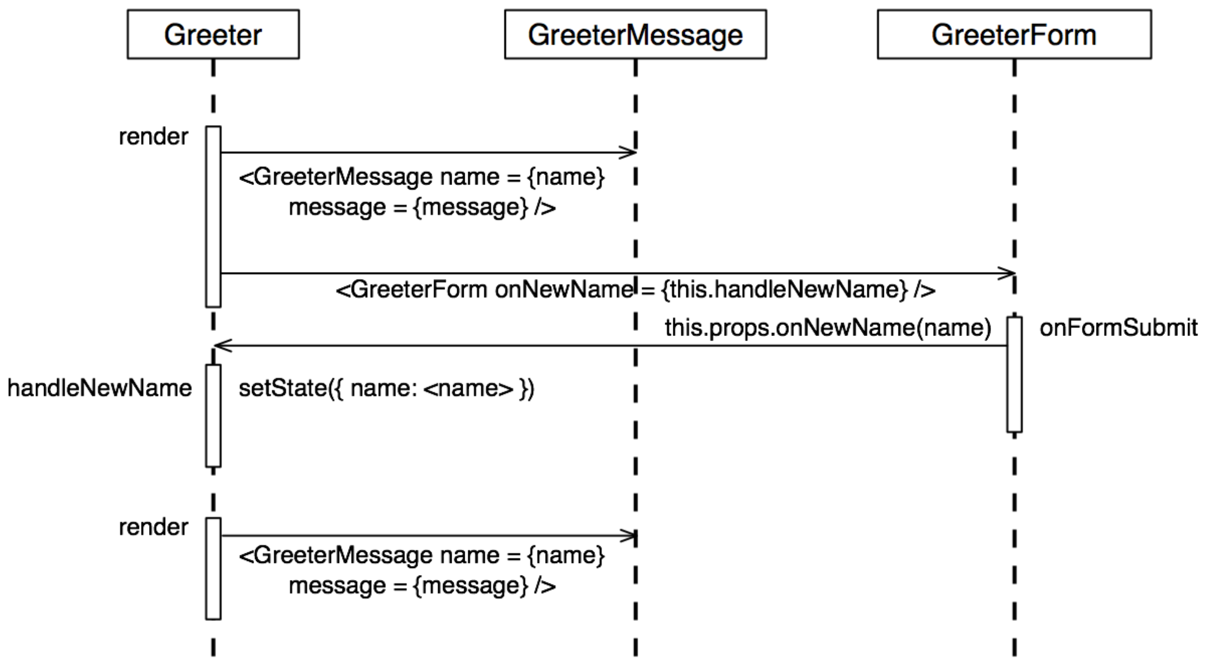
Presentational Component: `GreeterMessage` & `GreeterForm`

Container Component: `Greeter`

Naming convention

- `handleXXX`: Contianer/Parent component reacts to events from child component to update its state
- `onXXX`: Presentational/Child component react to the same event

Data flow


Greeter
GreeterMessage
GreeterForm
render
<GreeterMessage name = {name}
message = {message} />
<GreeterForm onNewName = {this.handleNewName} />
this.props.onNewName(name)
onFormSubmit
handleNewName
setState({ name: <name> })
render
<GreeterMessage name = {name}
message = {message} />

016 Aside Props State

`props` can never be changed, the following code will not work:

```
this.props.message = "something new"
```

04 A Better React Project

018 What is Webpack And Installing It

Current problem:

- 4 script tags will cause browser to fetch them using 4 seperate HTTP requests.
- The JSX code & ES6 features will cause the live convertion inside the browser, that could be very time consuming & expensive.
- All the react component should be in their seperate files for better test isolation and code maintenance.

Ideally, the code needs convertion would be converted on the server sie and manage all third party dependencies using npm and combine them into a single bundle using webpack.

In order to use webpack, install it globally as:

```
npm install -g webpack@1.12.13
```

019 Generating Our Bundle

Using `webpack`, `webpack` takes 2 arguments in the form of:

```
webpack <location-of-the-root-file> <location-of-the-bundle-file>
```

In the case of this episode, use the following command to trigger the bundle process:

```
webpack ./public/app.js ./public/bundle.js
```

020 The Webpack Config File

In order to support the JSX and ES6 convertion, we should config webpack using `webpack.config.js`

```
module.exports = {
  // location-of-the-root-file: the first arugment when using command line
  entry: './public/app.js',

  // location-of-the-bundle-file: the second argument when using command line
  output: {
    path: __dirname,
    filename: './public/bundle.js'
  },
  resolve: {
    extensions: ['', '.js', '.jsx'] // What file to convert? ES6 (.js) & JSX (.jsx)
  }
};
```

With `webpack.config.js`, we can run `webpack` without command line arguments:

```
webpack
```

Notice: up till now there is not any convertion, we have to config webpack for JSX & ES6 convertion, no magic here.

021 Adding Babel JSX Support

By default webpack doesn't know what to do with JSX files, if we change the `entry` In `webpack.config.js` as:

```
module.exports = {
  entry: './public/app.jsx',
  output: {
    path: __dirname,
    filename: './public/bundle.js'
  },
  resolve: {
    extensions: ['', '.js', '.jsx']
  }
};
```

Running webpack, we'll encounter the following error:

```
bash$ webpack
Hash: 2a82a67f90f9aa05ab4a
Version: webpack 1.12.13
Time: 34ms
   [0] ./public/app.jsx 0 bytes [built] [failed]

ERROR in ./public/app.jsx
Module parse failed: /Users/Song/Video/Selection/React/The Complete React
Web App Developer Course/04 A Better React Project/attached_files/021 Adding
Babel  JSX Support/react-4-4-babel-jsx/public/app.jsx Line 10: Unexpected
token <
You may need an appropriate loader to handle this file type.
|
|       return (
|         <div>
|           <h1>Hello {name}!</h1>
|           <p>{message}</p>
```

So, a **loader** is needed. Update `webpack.config.js`:

```
module.exports = {
  entry: './public/app.jsx',
  output: {
    path: __dirname,
    filename: './public/bundle.js'
  },
  resolve: {
    extensions: ['', '.js', '.jsx']
  },
  module: {
    loaders: [
      {
        loader: 'babel-loader', // devDependencies: "babel-loader": "^6.2.2"
        query: {
          presets: ['react', 'es2015']
        },
        test: /\.jsx?$/, // .js or .jsx
        // don't process files under these folders
        exclude: /(node_modules|bower_components)/
      }
    ]
  }
};
```

When using `webpack` every file is treated as its own island, that means if we want to use somthing as `React` we can't assume it is available globally, we have to export it explicitly.

```
var React = require('react');
var ReactDOM = require('react-dom');

// You can now use React & ReactDOM
// That' why we have them in package.json

// "dependencies": {
//   "express": "^4.13.4",
//   "react": "^0.14.7",
//   "react-dom": "^0.14.7"
// },
```

022 Refactoring Your Components

Load user defiend module using relative path:

```
// . means the current directory of current file
var Greeter = require('./components/Greeter');
```

Understanding load path:

```
// Remember #include <...> ?
var React = require('react');

// You don't specify the path, all set by package.json & npm install
var ReactDOM = require('react-dom');

// Remember #include "..." ?
var Greeter = require('./components/Greeter');
```

023 Webpack Custom Package Names

The goal: get rid of the relative path when loading user defined module.

Instead of:

```
var Greeter = require('./components/Greeter');
```

We can simply write:

```
var Greeter = require('Greeter');
```

Soution: in `webpack.config.js`:

```
module.exports = {
  entry: './public/app.jsx',
  output: {
    path: __dirname,
    filename: './public/bundle.js'
  },
  resolve: {
    // ------------------------------ START SOLUTIN ------------------------
--------
    root: __dirname,
    alias: {
      Greeter: 'public/components/Greeter.jsx',
      GreeterMessage: 'public/components/GreeterMessage.jsx',
      GreeterForm: 'public/components/GreeterForm.jsx'
    },
    // ------------------------------- END SOLUTION ------------------------
--------
    extensions: ['', '.js', '.jsx']
  },
  module: {
    loaders: [
      {
        loader: 'babel-loader',
        query: {
          presets: ['react', 'es2015']
        },
        test: /\.jsx?$/,
        exclude: /(node_modules|bower_components)/
      }
    ]
  }
};
```

Notice: Each time you create new module, you have to register it in `alias` field.

Launch `webpack` in watch mode:

```
# Listen for change in your file and will automatically re-bundle once a
file changes
webpack -w
```

024 Boilerplate Project

Project folder structure:

- `app` directory: your application logic (browser independent)
- `public` directory: filees used by the browser (`bundle.js` included)

Node.js project name restriction:

- No space
- No uppercase letters

025 Bonus Using Experimental JavaScript Features

In order to use JavaScript language experimental features, visit babel-plugin and following the instructions for setup.

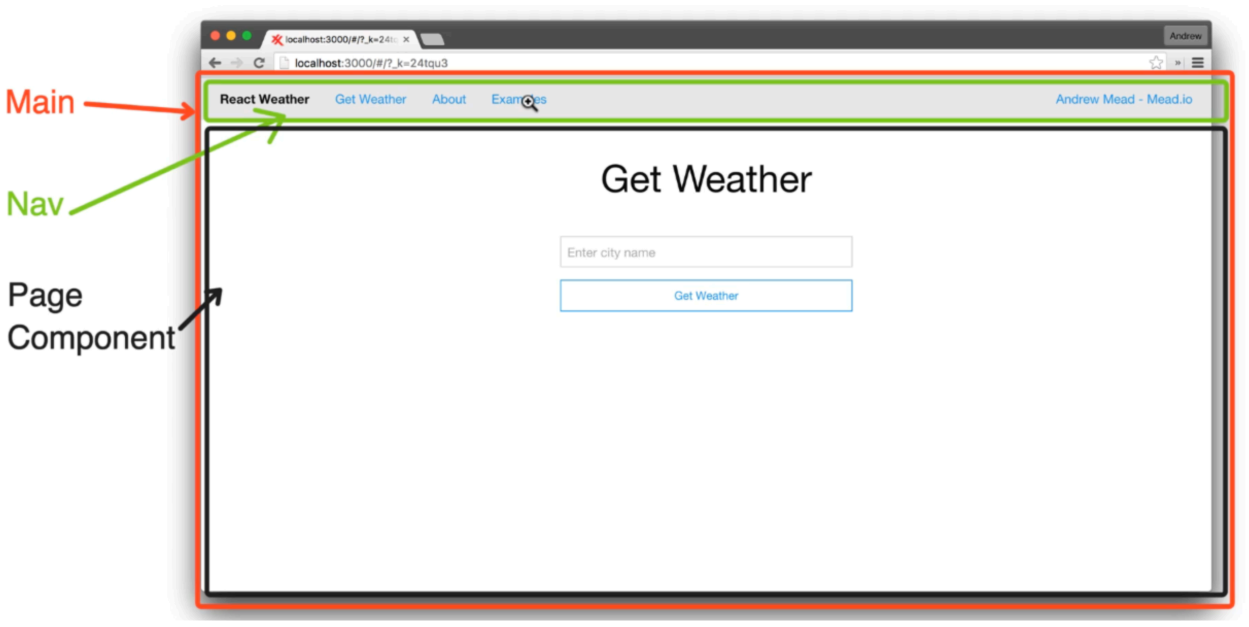
05 Routing Our Weather App

027 Adding React-Router

Why routing? Allow multiple react pages all inside the application on the frond end.

First steps when designing a React application

1. Dissect the page using component oriented thinking
2. Distinguish between container component and presentational component



027 Adding React-Router

Setup react router:

```
npm install react-router@2 --save
```

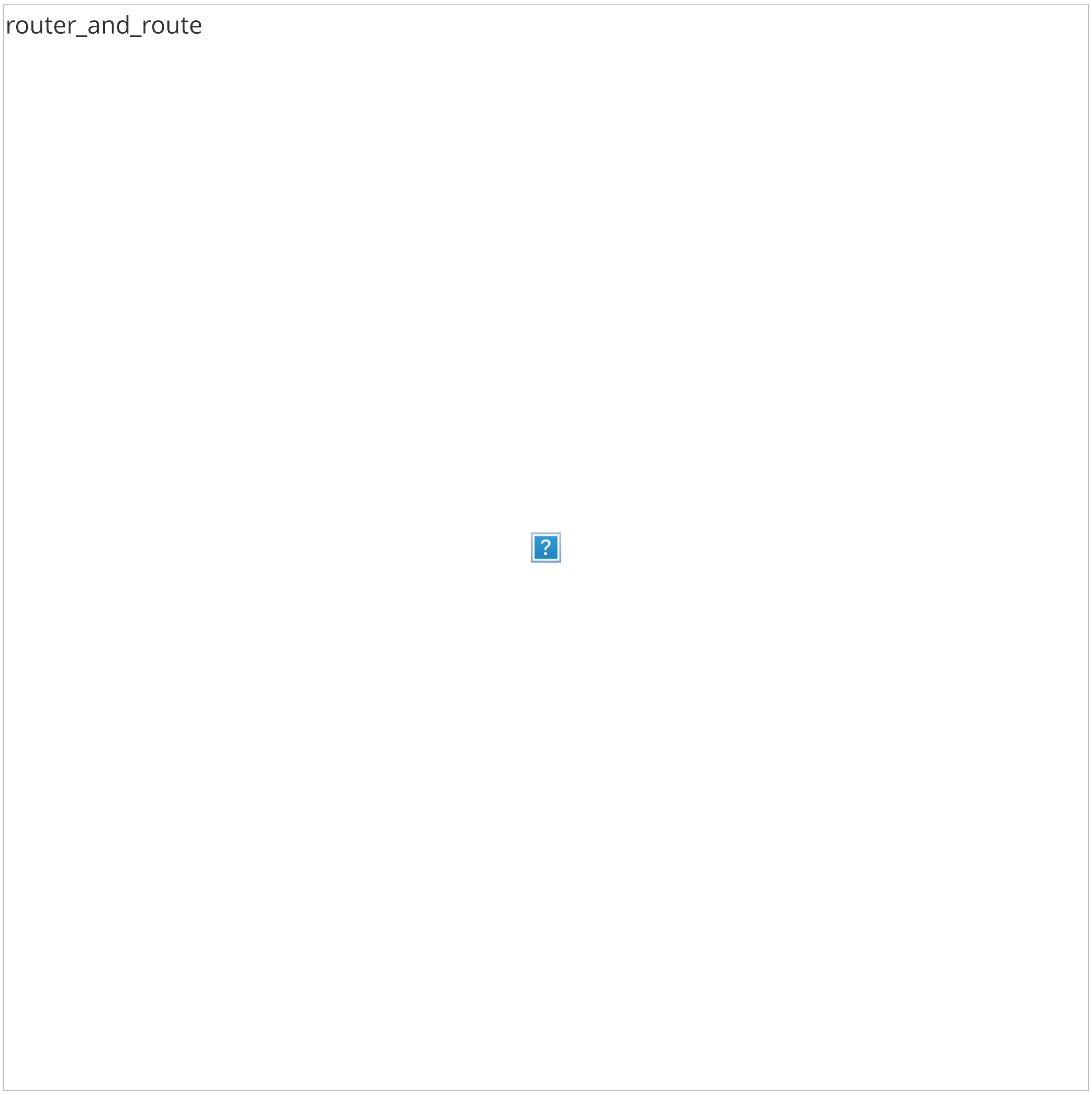
ES6 destructive syntax:

```
var { Route, Router, IndexRoute, hashHistory } = require("react-router");
```

The statement above is equivalent to:

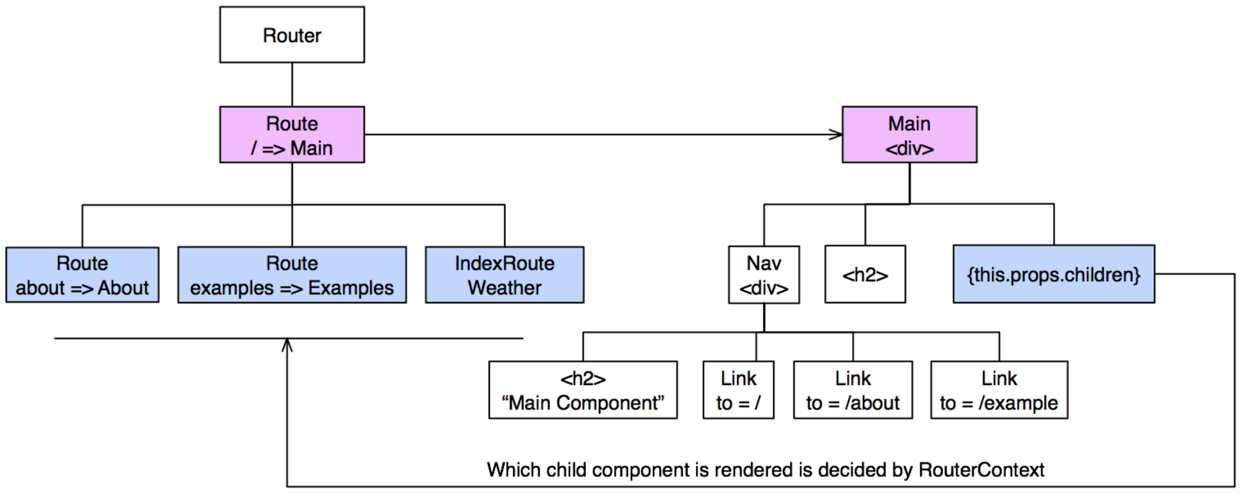
```
var Route = require("react-router").Route;
var Router = require("react-router").Router;
var IndexRoute = require("react-router").IndexRoute;
var hashHistory = require("react-router").hashHistory;
```

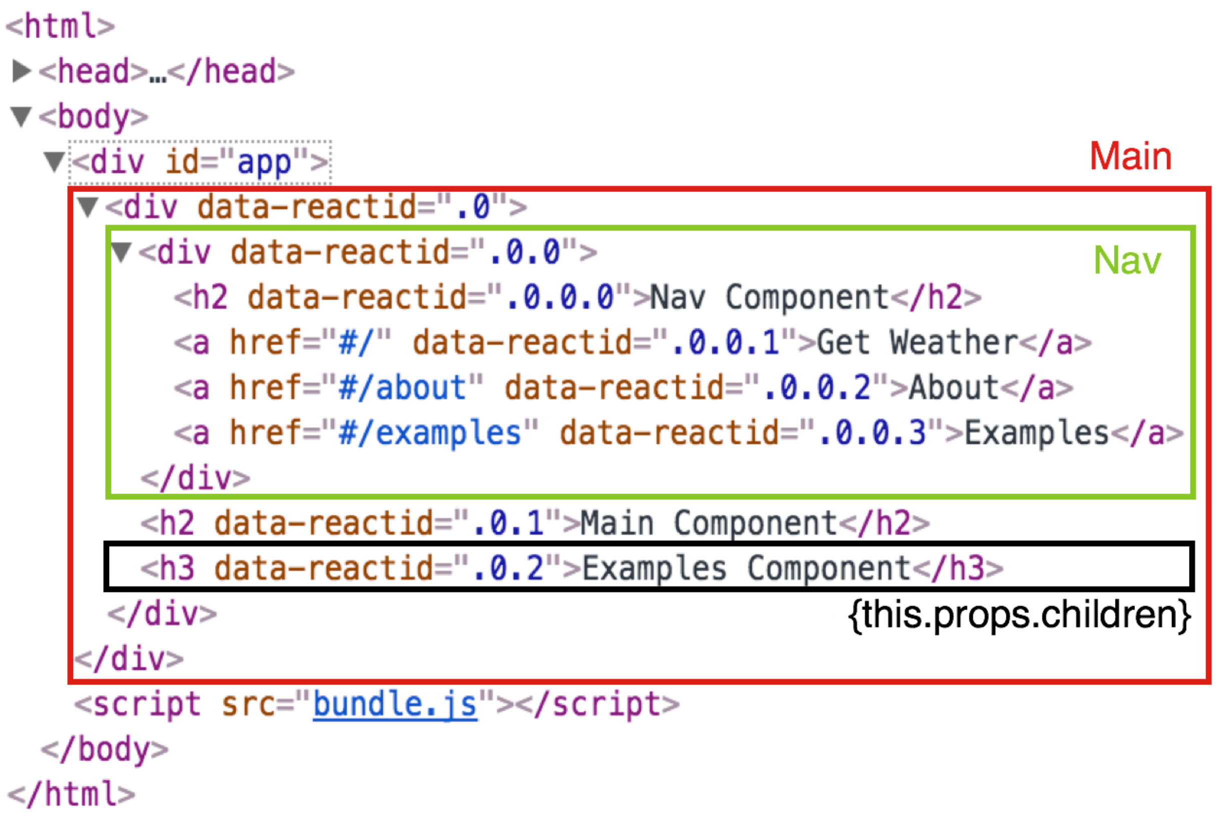
Understanding React router


router_and_route

Essentially, a React router is just a container component with additionaly features.

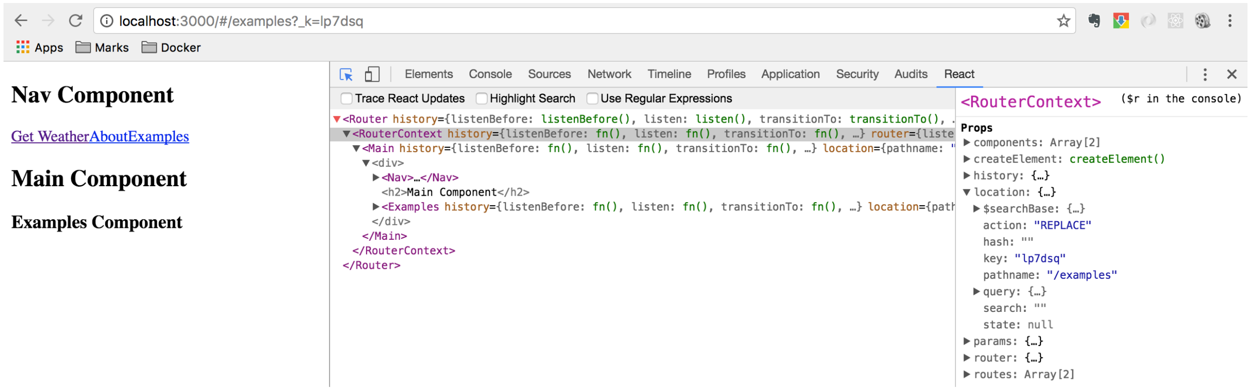
028 Creating Our Pages





```
<html>
▶<head>…</head>
▼<body>
  ▼<div id="app">
    ▼<div data-reactid=".0">
      ▼<div data-reactid=".0.0">
          <h2 data-reactid=".0.0.0">Nav Component</h2>
          <a href="#/" data-reactid=".0.0.1">Get Weather</a>
          <a href="#/about" data-reactid=".0.0.2">About</a>
          <a href="#/examples" data-reactid=".0.0.3">Examples</a>
        </div>
        <h2 data-reactid=".0.1">Main Component</h2>
        <h3 data-reactid=".0.2">Examples Component</h3>
      </div>
    </div>
    <script src="bundle.js"></script>
  </body>
</html>
```

Each `RouterContext` is just a stack frame. Click "Go back" & "Go forward" to view the stack change.



Nav Component

Get WeatherAboutExamples

Main Component

Examples Component

```
▼<Router history={listenBefore: listenBefore(), listen: listen(), transitionTo: transitionTo(), …
  ▼<RouterContext history={listenBefore: fn(), listen: fn(), transitionTo: fn(), …} router={liste
    ▼<Main history={listenBefore: fn(), listen: fn(), transitionTo: fn(), …} location={pathname: "
      ▼<div>
        ▶<Nav>…</Nav>
          <h2>Main Component</h2>
        ▶<Examples history={listenBefore: fn(), listen: fn(), transitionTo: fn(), …} location={path
        </div>
      </Main>
    </RouterContext>
  </Router>
```

```
<RouterContext>   ($r in the console)
Props
▶ components: Array[2]
▶ createElement: createElement()
▶ history: {…}
▼ location: {…}
  ▶ $searchBase: {…}
    action: "REPLACE"
    hash: ""
    key: "lp7dsq"
    pathname: "/examples"
  ▶ query: {…}
    search: ""
    state: null
▶ params: {…}
▶ router: {…}
▶ routes: Array[2]
```

Question: What if we delete the `IndexRoute`???

```
ReactDOM.render(
  <Router history={hashHistory}>
    <Route path="/" component={Main}>
      <Route path="about" component={About}/>
      <Route path="examples" component={Examples}/>
    </Route>
  </Router>,
  document.getElementById('app')
);
```

To conclude:

```
ReactDOM.render(
  <Router history={hashHistory}>
    <Route path="/" component={Main}>
      {/*
          All the components defined by the child Route will be the children
          of the component defined by the parent route.
          At runtime there will be at most one child component, decided by
RouterContext
      */}
      <Route path="about" component={About}/>
      <Route path="examples" component={Examples}/>
      <IndexRoute component={Weather}/>
    </Route>
  </Router>,
  document.getElementById('app')
);
```

Check out Leveling Up With React: React Router for all you should know about React router.

029 Why use Link

Why not just write:

```
<a href="#/about">Go To About</a>
```

To add custom styles and classes to the element of the current page.

```
<Link to="/about" activeClassName="active"  activeStyle={{fontWeight:
'bold'}}>About</Link>
```

Why change

```
<div>
  <h2>Nav Component</h2>
  <Link to="/" ...>Get Weather</IndexLink>
  <Link to="/about" ...>About</Link>
  <Link to="/examples" ...>Examples</Link>
</div>
```

to

```
<div>
  <h2>Nav Component</h2>
  <IndexLink to="/" ...>Get Weather</IndexLink>
  <Link to="/about" ...>About</Link>
  <Link to="/examples" ...>Examples</Link>
</div>
```

It's all about match! - When you nesting routes, React will try to match the URL (Link) to the routes defined in the code, when there is a match, React will add "active" class to the matched component. Before changing to `IndexLink` the "Get Weather" page is defintely a match.

030 Creating WeatherForm WeatherMessage

Quiz: Presentational/Container component analysis.

032 ES6 Promises

Checkut [MDN Refrence]() for ES6 Promise.

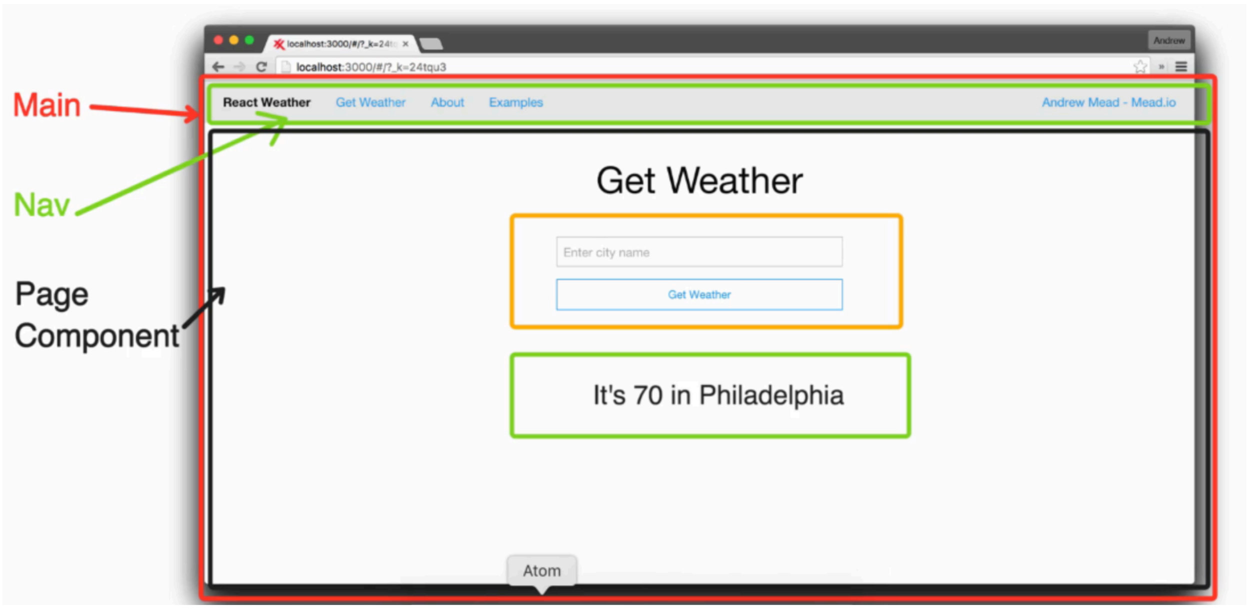
```
function addPromise (a, b) {
  return new Promise(function (resolve, reject) {
    if (typeof a === 'number' && typeof b === 'number') {
      resolve(a + b);
    } else {
      reject('A & b need to be numbers');
    }
  });
}

addPromise(2, 3).then(function (sum) {       // resolve
  console.log('success', sum);
}, function (err) {                          // reject
  console.log('error', err);
});

addPromise('andrew', 9).then(function (sum) {
  console.log('this should not show up');    // resolve
}, function (err) {
  console.log('This should appear', err);    // reject
});
```

033 Faking Our Call

Again, presentational vs. container



Again, the naming convention: `handleSearch` in `Weather` component and `onSearch` in `WhetherForm` component.

When developing React application, the key is to disset the system into small reusable pieses in for form of component. Taking `WhetherForm` as example, we can create `isValidLocation` for better test support.

Using destructive assignment syntaxt for passing container's state as props to the presetational.

```
render: function () {
  var {temp, location} = this.state;

  return (
    <div>
      <h3>Weather Component</h3>
      <WeatherForm onSearch={this.handleSearch}/>
      <WeatherMessage temp={temp} location={location}/>
    </div>
  )
}
```

Question: Which component should be responsible for the AJAX call? The container or the presentational? Why?

034 Making Our API Call

Project folder structure:

- `component` for UI
- `api` for library

Question, why the the component names start with uppercases, and the weather API library starts with a lowercase? Whether it is intended to be used as class or just a object.

035 Adding Loading Text

Using nested function in container component for conditional component rendering:

```
render: function () {
  var {isLoading, temp, location} = this.state;

  function renderMessage () {
    if (isLoading) {
      return <h3>Fetching weather...</h3>;
    } else if (temp && location) {
      return <WeatherMessage temp={temp} location={location}/>;
    }
  }

  return (
    <div>
      <h3>Weather Component</h3>
      <WeatherForm onSearch={this.handleSearch}/>
      {renderMessage()}
    </div>
  )
}
```

The true beauty of React: your application you see in the browser is always representing the state inside your components. Instead of rendering entire application, React is smart enough to figure out which part is changed and only re-renders that part.

039 ES6 Aside Arrow Functions

The major difference between arrow functions and anonymous functions: `this` binding.

```
var names = ['Andrew', 'Julie', 'Jen'];
var person = {
  name: 'Andrew',
  greetWithAnonymourFunction: function () {
    names.forEach(function (name) {
      // If you are using Node, `this` refers to the global node object
      console.log(this.name + ' says hi to ' + name)
    })
  },
  greetWithArrowFunction: function () {
    names.forEach((name) => {
      // `this` binds to the person
      console.log(this.name + ' says hi to ' + name)
    });
  }
};

person.greetWithAnonymourFunction();
person.greetWithArrowFunction();
```

040 Refactoring Stateless Functional Components

For presentational components, if the **ONLY** function defined in the component is `render`, you can use stateless functional components to simplify the definition:

```
// In order to write:
var About = React.createClass({
  render: function () {
    return (
      <h3>About Component</h3>
    )
  }
});

// Simply put: It's just a function
var About = (props) => {
  return (
    <h3>About Component</h3>
  )
};
```

If the `render` function refers `props`, pass `props` as function argument:

```
var Main = (props) => {
  return (
    <div>
      <Nav/>
      <h2>Main Component</h2>
      {props.children}
    </div>
  );
}
```

And you can apply destructive assignment to the function parameter:

```
var WeatherMessage = ({temp, location}) => {
  return (
    <h3>It's it {temp} in {location}.</h3>
  )
};
```

06 Deploying Your App To Production

SKIP

07 Styling Your App With Foundation

`class` attribute is the interface between JavaScript and CSS.

Page skeleton first, refactor using the target CSS library.

056 Adding A Modal For Errors Part 1

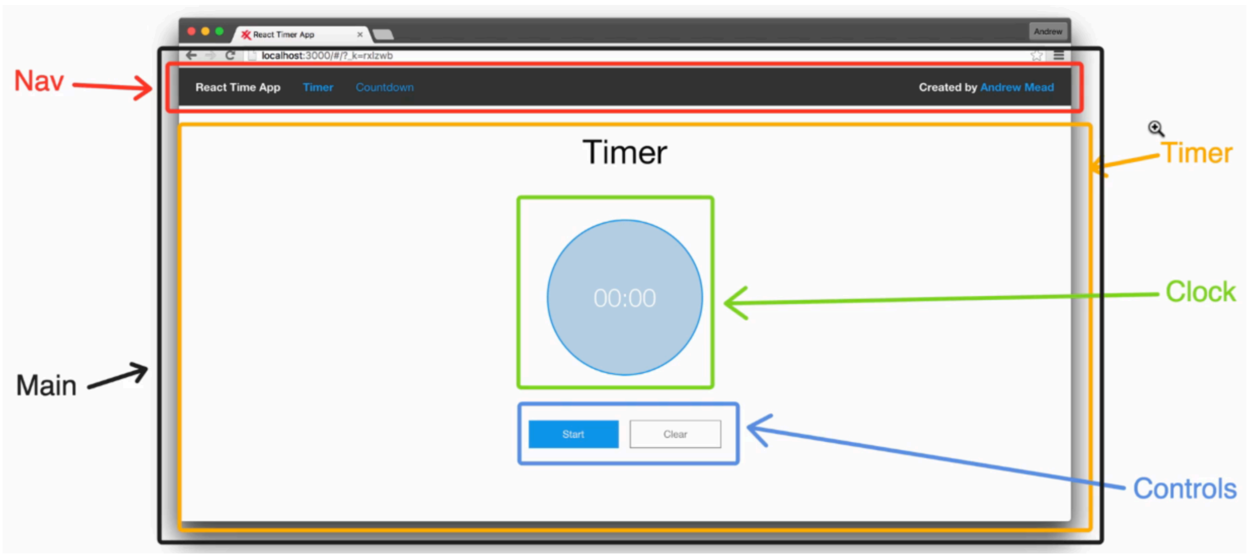
Understanding component lifecycle.

057 Adding A Modal For Errors Part 2

Property validation.

08 Testing Component Lifecycle

064 Component Breakdown



Reusable component: `Clock`, `Controls`

068 Configuring Tests With Webpack React

Libraries used:

Karma: Test runner

Mocha: Test framework (`describe` & `it`)

- `describe`: Group test topic into sections
- `it`: Represents the unit under test

Expect: Assertion Library

What is a test runner? Just have a look at `karma.conf.js`:

```
var webpackConfig = require('./webpack.config.js');

module.exports = function (config) {
  config.set({
    browsers: ['Chrome'],
    singleRun: true,
    frameworks: ['mocha'],
    files: ['app/tests/**/*.test.jsx'],
    preprocessors: {
      // webpack: import all the required libraries
      // sourcemap: when we get error messages in the test, they are not
using bundle.js file but the actual jsx files
      'app/tests/**/*.test.jsx': ['webpack', 'sourcemap']
    },
    reporters: ['mocha'],
    client: {
      mocha: {
        timeout: '5000'
      }
    },
    webpack: webpackConfig,
    webpackServer: {
      noInfo: true
    }
  });
};
```

Structure of a test case:

```
describe("<what>", () => {
  it("<when> & <then>", () => {

  })
})
```

070 Clock Component

Component oriented test folder structure.

```
bash$ tree app/tests/
app/tests/
├── app.test.jsx
└── components
    └── Clock.test.jsx

1 directory, 2 files
```

Applied test case structure:

```
describe('Clock', () => {
  it('should exist', () => {
    // ...
  });

  describe('formatSeconds', () => {
    it('should format seconds', () => {
      // ...
    });

    it('should format seconds when min/sec are less than 10', () => {
      // ...
    });
  });
})
```

Funtional testing: `Clock#formatSeconds` - It's a great TDD case.

To adopt "parameterized tests" with Mocha, check out
http://stackoverflow.com/questions/17569382/parametrized-tests-with-mocha

In order to create a component instance during test, use `react-addons-test-utils` library.

```
var TestUtils = require('react-addons-test-utils');

// ...
    it('should format seconds when min/sec are less than 10', () => {
      var clock = TestUtils.renderIntoDocument(<Clock/>);
      var seconds = 61;
      var expected = '01:01';
      var actual = clock.formatSeconds(seconds);

      expect(actual).toBe(expected);
    });
```

071 Clock Component Part 2

Rendering test - using jQuery w/ `react-addons-test-utils`:

```
describe('render', () => {
  it('should render clock to output', () => {
    var clock = TestUtils.renderIntoDocument(<Clock totalSeconds={62}/>);
    var $el = $(ReactDOM.findDOMNode(clock));
    var actualText = $el.find('.clock-text').text();

    expect(actualText).toBe('01:02');
  });
});
```