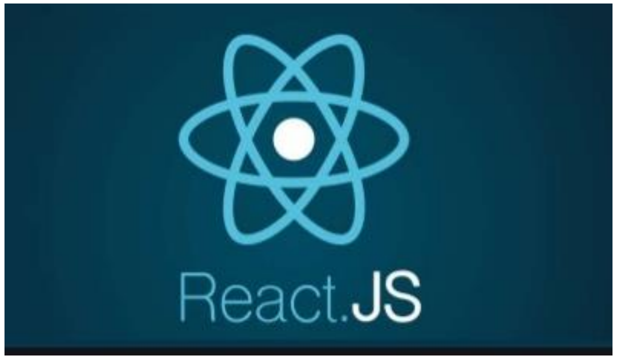


Presentation by Uplatz

Contact Us: https://training.uplatz.com/

Email: info@uplatz.com

Phone:+44 7836 212635

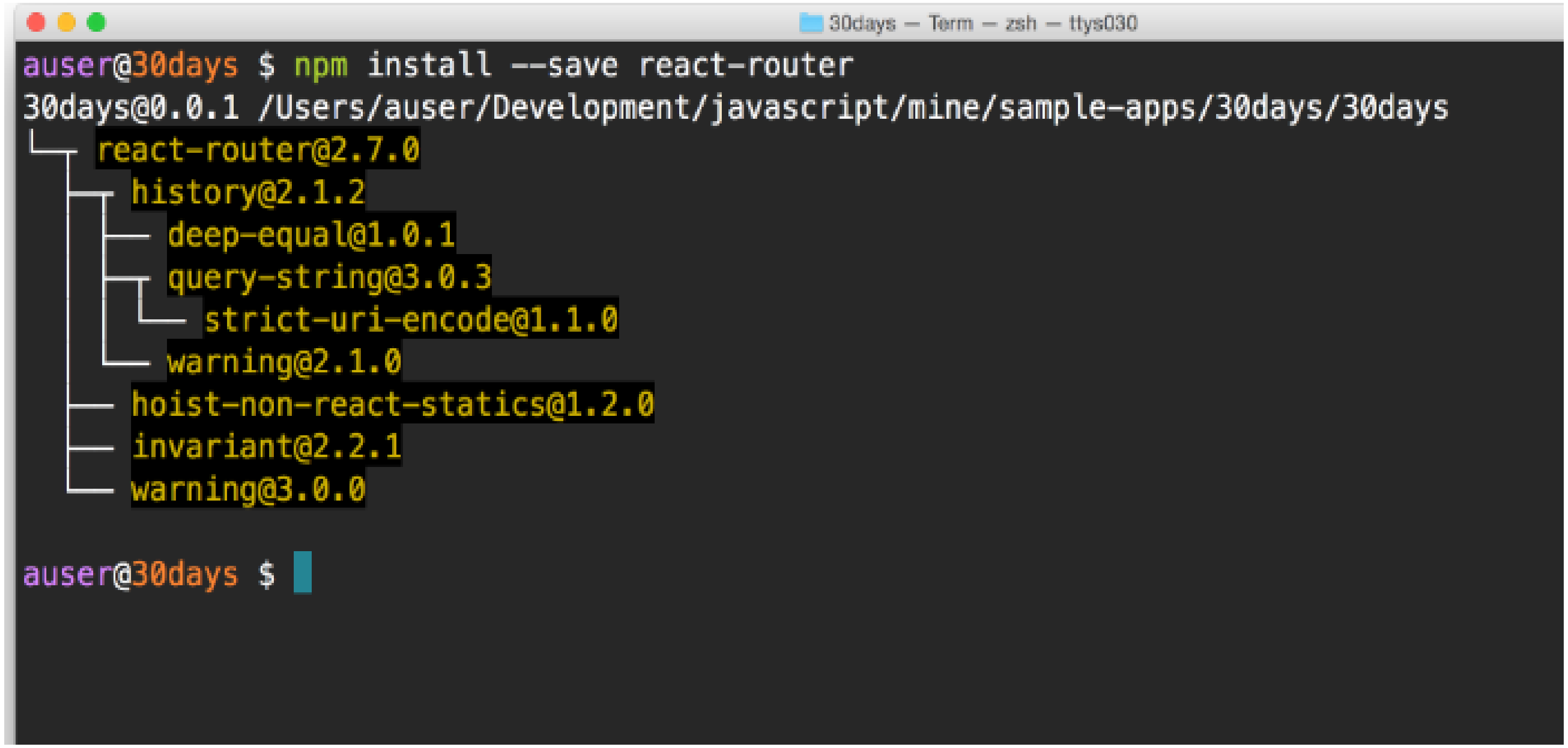
## Table Of Contents:

Uplatz

## Client-side Routing

- Right now, our app is limited to a single page. It's pretty rare to find any complex application that shows a single view.
- For instance, an application might have a login view where a user can log in or a search results page that shows a user a list of their search results.
- These are two different views with two different page structures.
- Let's see how we can change that with our app today.
- We'll use the very popular react-router library for handling different links.
- In order to use the react-router library, we'll need to install it using the npm package manager:

## npm install --save react-router-dom



```
auser@30days $ npm install --save react-router
30days@0.0.1 /Users/auser/Development/javascript/mine/sample-apps/30days/30days
└── react-router@2.7.0
    ├─┬ history@2.1.2
    │ ├── deep-equal@1.0.1
    │ ├─┬ query-string@3.0.3
    │ │ └── strict-uri-encode@1.1.0
    │ └── warning@2.1.0
    ├── hoist-non-react-statics@1.2.0
    ├── invariant@2.2.1
    └── warning@3.0.0

auser@30days $
```

- With react-router installed, we'll import a few packages from the library and update our app architecture.
- Before we make those updates, let's take a step back and from a high level look at how and why architect our application this way.

- Conceptually with React, we've seen how we can create tree structures using components and nested components.
- Using this perspective with a single page app with routes, we can think of the different parts of a page as children.
- Routing in a single page app from this perspective is the idea that we can take a part of a subtree and switch it out with another subtree.
- We can then dynamically switch out the different trees in the browser.
- In other words, we'll define a React component that acts as a root component of the routable elements.
- We can then tell React to change a view, which can just swap out an entire React component for another one as though it's a completely differe page rendered by a server.

- We'll take our App component and define all of the different routes we can make in our app in this App component.
- We'll need to pull some components from the react-router package.
- These components we'll use to set up this structure are as follows:

**<BrowserRouter /> / <Router />**

- This is the component we'll use to define the root or the routing tree.
- The **<BrowserRouter />** component is the component where React will replace it's children on a per-route basis.

**<Route />**

- We'll use the <Route /> component to create a route available at a specific location available

- The <Route /> component is mounted at page URLs that match a particular route set up in the route's configuration props.
- One older, compatible way of handling client-side navigation is to use the # (hash) mark denoting the application endpoint.
- We'll use this method. We'll need this object imported to tell the browser this is how we want to handle our navigation.
- From the app we created a few days ago's root directory, let's update our src/App.js to import these modules.
- We'll import the BrowserRouter using a different name syntax via ES6:

```
import React from "react";
import { BrowserRouter as Router, Route } from "react-router-dom";
export class App extends React.Component {
  render() {
    <Router>{/* routes will go here */}</Router>;
  }
}
```

- Now let's define our first route. To define a route, we'll use the <Route /> component export from react-router and pass it a few props:
- path - The path for the route to be active
- component - The component that defines the view of the route

- Let's define the a route at the root path / with a stateless component that just displays some static content:

```
const Home = () => (
  <div>
    <h1>Welcome home</h1>
  </div>
);// ...
class App extends React.Component {
  render() {
    return (
      <Router>
        <Route path="/" component={Home} />
      </Router>
    );} }
```

- Loading this page in the browser, we can see we get our single route at the root url.
- Not very exciting.
- Let's add a second route that shows an about page at the /about URL.

```
const Home = () => (
  <div>
    <h1>Welcome home</h1>
  </div>
);
// ...
class App extends React.Component {
  render() {
    return (
<Router>
    <div>
      <Route path="/" component={Home} />
      <Route path="/about" component={About} />
    </div>
  </Router>
 );
}}
```

- In our view we'll need to add a link (or an anchor tag -- <a />) to enable our users to travel freely between the two different routes.
- However, using the <a /> tag will tell the browser to treat the route like it's a server-side route.
- Instead, we'll need to use a different component (surprise) called: <Link />.

- The <Link /> component requires a prop called to to point to the client-side route where we want to render.
- Let's update our Home and About components to use the Link:

```
import { BrowserRouter as Router, Route, Link } from
"react-router-dom";
const Home = () => (
  <div>
    <h1>Welcome home</h1>
    <Link to="/about">Go to about</Link>
  </div>
);
const About = () => (
  <div>
<h1>About</h1>
    <Link to="/">Go home</Link>
  </div>
);
```

- This happens because the react router will render all content that matches the path (unless otherwise specified).
- For this case, react router supplies us with the Switch component.
- The <Switch /> component will only render the first matching route it finds.
- Let's update our component to use the Switch component.
- As react router will try to render both components, we'll need to specify that we only want an exa match on the root component.

```
import { BrowserRouter as Router, Route, Link, Switch }
from "react-router-dom";
// ...
const Home = () => (
  <div>
    <h1>Welcome home</h1>
    <Link to="/about">Go to about</Link>
  </div>
);
// ...
class App extends React.Component {
  render() {
    return (
      <Router>
<Switch>
        <Route path="/about" component={About} />
        <Route path="/" component={Home} />
      </Switch>
    </Router>
  );
 }}
```

**Showing views**

- ➢ Although this is a limited introduction, we could not leave the discussion of dealing with react router without talking about the different ways we can get subcomponents to render.
- ➢ We've already seen the simplest way possible, using the component prop, however there is a more powerful method using a prop called render.

- The render prop is expected to be a function that will be called with the match object along with the location and route configuration.
- The render prop allows us to render whatever we want in a subroute, which includes rendering other routes.

```
const Home = () => (
  <div>
    <h1>Welcome home</h1>
    <Link to="/about">Go to about</Link>
  </div>
);
const About = ({ name }) => (
  <div>
    <h1>About {name}</h1>
```

```
 </div>
);
// ...
class App extends React.Component {
  render() {
    return (
      <Router>
        <Switch>
          <Route
            path="/about"
            render={renderProps => (
              <div>
                <Link to="/about/ari">Ari</Link>
                <Link to="/about/nate">Nate</Link>
<Route  path="/about/:name"
              render={renderProps => (
                <div>
                  <About
name={renderProps.match.params.name} />
                  <Link to="/">Go home</Link>
                </div>
              )}
                />
              </div>
            )}
          />
          <Route
```

```
path="/"
          render={renderProps => (
            <div>
              Home is underneath me
              <Home {...this.props} {...renderProps} />
            </div>
          )}
        />
      </Switch>
    </Router>); } }
```

- Now we have multiple pages in our application.
- We've looked at how we can render these routes through nested components with just a few of the exports from react-router.

- **react-router provides so much more functionality that we don't have time to cover in our brisk intro to routing**

**React Routing**

**Example Routes.js file, followed by use of Router Link in component**

- Place a file like the following in your top level directory.
- It defines which components to render for which paths

```
import React from 'react';
import { Route, IndexRoute } from 'react-router';
import New from './containers/new-post';
import Show from './containers/show';
import Index from './containers/home';
import App from './components/app';
export default(
 <Route path="/" component={App}>
 <IndexRoute component={Index} />
 <Route path="posts/new" component={New} />
 <Route path="posts/:id" component={Show} />
 </Route>
);
```

- Now in your top level index.js that is your entry point to the app, you need only render this Router
- component like so:

```
import React from 'react';
import ReactDOM from 'react-dom';
```

**import { Router, browserHistory } from 'react-router';**

// import the routes component we created in routes.js

import routes from './routes';

// entry point

**ReactDOM.render(**

**<Router history={browserHistory} routes={routes} />**

**, document.getElementById('main'));**

- Now it is simply a matter of using Link instead of <a> tags throughout your application.
- Using Link will communicate with React Router to change the React Router route to the specified link, which will in turn render the correct component as defined in routes.js

**import React from 'react';**

```
import { Link } from 'react-router';
export default function PostButton(props) {
 return (
 <Link to={`posts/${props.postId}`}>
 <div className="post-button" >
 {props.title}
 <span>{props.tags}</span>
 </div>
 </Link>
 );
}
```

## React Routing Async

```
import React from 'react';
import { Route, IndexRoute } from 'react-router';
```

Uplatz

```
import Index from './containers/home';
import App from './components/app';
//for single Component lazy load use this
const ContactComponent = () => {
 return {
 getComponent: (location, callback)=> {
 require.ensure([], require => {
 callback(null,
require('./components/Contact')["default"]);
 }, 'Contact');
 }} };
//for multiple componnets
const groupedComponents = (pageName) => {
 return {
```

```
getComponent: (location, callback)=> {
 require.ensure([], require => {
 switch(pageName){
 case 'about' :
 callback(null, require( "./components/about"
)["default"]);
 break ;
 case 'tos' :
 callback(null, require( "./components/tos"
)["default"]);
 break ;
 }}, "groupedComponents");
 }}
};export default(
<Route path="/" component={App}>
 <IndexRoute component={Index} />
 <Route path="/contact" {...ContactComponent()} />
 <Route path="/about"
{...groupedComponents('about')} />
 <Route path="/tos" {...groupedComponents('tos')} />
 </Route>
);
```

## Communicate Between Components

**Communication between Stateless Functional Components**

- ➢ In this example we will make use of Redux and React Redux modules to handle our application state and for auto re-render of our functional components.,

- And ofcourse React and React Dom You can checkout the completed demo here
- In the example below we have three different components and one connected component
- **UserInputForm:** This component display an input field And when the field value changes, it calls
- **inputChange** method on props (which is provided by the parent component) and if the data is provided as well, it displays that in the input field.
- **UserDashboard:** This component displays a simple message and also nests UserInputForm component, It also passes inputChange method to UserInputForm component, UserInputForm component inturn makes use of this method to communicate with the parent component.

- UserDashboardConnected: This component just wraps the UserDashboard component using ReactRedux connect method.,
- This makes it easier for us to manage the component state and update the component when the state changes.

**App**: This component just renders the **UserDashboardConnected component.**

```
const UserInputForm = (props) => {
 let handleSubmit = (e) => {
 e.preventDefault();
 }return(
 <form action="" onSubmit={handleSubmit}>
 <label htmlFor="name">Please enter your
name</label><br />
```

```
<input type="text" id="name"
defaultValue={props.data.name || ''} onChange={
props.inputChange } />
 </form>
 )
}const UserDashboard = (props) => {
 let inputChangeHandler = (event) => {
 props.updateName(event.target.value);
 }
 return(
 <div>
 <h1>Hi { props.user.name || 'User' }</h1>
 <UserInputForm data={props.user}
inputChange={inputChangeHandler} />
 </div>
 )
}const mapStateToProps = (state) => {
 return {
 user: state
 };}
const mapDispatchToProps = (dispatch) => {
 return {
 updateName: (data) => dispatch(
Action.updateName(data) ),
 };};
const { connect, Provider } = ReactRedux;
const UserDashboardConnected = connect(
 mapStateToProps,
 mapDispatchToProps
```

```
)(UserDashboard);
const App = (props) => {
 return(
 <div>
 <h1>Communication between Stateless Functional
Components</h1>
 <UserDashboardConnected />
 </div>
 )}
const user = (state={name: 'John'}, action) => {
 switch (action.type) {
 case 'UPDATE_NAME':
 return Object.assign( {}, state, {name: action.payload}
);default:
```

```
return state;
 }
};
const { createStore } = Redux;
const store = createStore(user);
const Action = {
 updateName: (data) => {
 return { type : 'UPDATE_NAME', payload: data }
 },}ReactDOM.render(
 <Provider store={ store }>
 <App />
 </Provider>,
 document.getElementById('application')
);
```

## Static Routing

- If you’ve used Rails, Express, Ember, Angular etc. you’ve used static routing.
- In these frameworks, you declare your routes as part of your app’s initialization before any rendering takes place
- React Router pre-v4 was also static (mostly). Let’s take a look at how to configure routes in express:// Express Style routing:

```
app.get("/", handleIndex);
app.get("/invoices", handleInvoices);
app.get("/invoices/:id", handleInvoice);
app.get("/invoices/:id/edit", handleInvoiceEdit);
app.listen();
```

- Note how the routes are declared before the app listens.
- The client side routers we've used are similar.
- In Angular you declare your routes up front and then import them to the top-level AppModule before rendering:// Angular Style routing:

```
const appRoutes: Routes = [
 {
   path: "crisis-center",
   component: CrisisListComponent
 },
 {
   path: "hero/:id",
   component: HeroDetailComponent
 },
{
   path: "heroes",
   component: HeroListComponent,
   data: { title: "Heroes List" }
 },
 {
   path: "",
   redirectTo: "/heroes",
   pathMatch: "full"
 },
 {
   path: "**",
   component: PageNotFoundComponent
 }
];
```

```
e({
  imports: [RouterModule.forRoot(appRoutes)]
})
export class AppModule {}
```

- Ember has a conventional routes.js file that the build reads and imports into the application for you.
- Again, this happens before your app renders.// Ember Style Router:

```
Router.map(function() {
  this.route("about");
  this.route("contact");
  this.route("rentals", function() {
    this.route("show", { path: "/:rental_id" });
  });
});
```

```
export default Router;
```

## Dynamic Routing

- When we say dynamic routing, we mean routing that takes place as your app is rendering, not in a configuration or convention outside of a running app.
- That means almost everything is a component in React Router.

```
import { NativeRouter } from "react-router-native";
// react-dom (what we'll use here)
import { BrowserRouter } from "react-router-dom";
ReactDOM.render(
  <BrowserRouter>
    <App />
  </BrowserRouter>,
```

**el**

**);**

**Next, grab the link component to link to a new location:const App = () => (**

```
  <div>
    <nav>
      <Link to="/dashboard">Dashboard</Link>
    </nav>
  </div>
);
```

Finally, render a Route to show some UI when the user visits /dashboard.const App = () => (

```
  <div>
    <nav>
```

```
<Link to="/dashboard">Dashboard</Link>
    </nav>
    <div>
      <Route path="/dashboard"
component={Dashboard} />
    </div>
  </div>
);
```

- The Route will render <Dashboard {...props}/> where props are some router specific things that look like { match, location, history }.
- If the user is not at /dashboard then the Route will render null.

Nested Routes

## How to setup a basic webpack, react and babel environment

- How to build a pipeline for a customized "Hello
- world" with images

**Step 1: Install Node.js**

- The build pipeline you will be building is based in Node.js so you must ensure in the first instance that you have this installed

**Step 2:** Initialise your project as an node module

Open your project folder on the command line and use the following command:

**npm init**

- For the purposes of this example you can feel free to take the defaults or if you'd like more info on what all this means you can check out this SO doc on setting up package configuration.

**Step 3:** Install necessary npm packages

Run the following command on the command line to install the packages necessary for this example:

**npm install --save react react-dom**

- Then for the dev dependencies run this command:
- npm install --save-dev babel-core babel-preset-react babel-preset-es2015 webpack babel-loader cssloader style-loader file-loader image-webpack-loader
- Finally webpack and webpack-dev-server are things that are worth installing globally rather than as a dependency of your project, if you'd prefer to add it as a dependency then that will work to, I don't.
- Here is the command to run:

```
npm install --global webpack webpack-dev-server
```

**Step 3:** Add a .babelrc file to the root of your project

This will setup babel to use the presets you've just installed. Your .babelrc file should look like this:

```
{
 "presets": ["react", "es2015"]
}
```

**Step 4: Setup project directory structure**

Set yourself up a directory stucture that looks like the below in the root of your directory:

```
 |- node_modules
 |- src/
  |- components/
  |- images/
```

```
|- styles/
 |- index.html
|- index.jsx
|- .babelrc
|- package.json
```

**NOTE**: The node_modules, .babelrc and package.json should all have already been there from previous steps I just included them so you can see where they fit.

**Step 5:** Populate the project with the Hello World project files

- This isn't really important to the process of building a pipeline so I'll just give you the code for these and you can copy paste them in:

**src/components/HelloWorldComponent.jsx**

```
import React, { Component } from 'react';
```

```
class HelloWorldComponent extends Component {
 constructor(props) {
 super(props);
 this.state = {name: 'Student'};
 this.handleChange = this.handleChange.bind(this);
 }
 handleChange(e) {
 this.setState({name: e.target.value});
 }
 render() {
 return (
 <div>
 <div className="image-container">
 <img src="./images/myImage.gif" />
```

Uplatz

```
 </div>
 <div className="form">
 <input type="text" onChange={this.handleChange} />
 <div>
 My name is {this.state.name} and I'm a clever cloggs
because I built a React build
pipeline
 </div>
 </div>
 </div>
 ); }}
export default HelloWorldComponent;
src/images/myImage.gif
```

- Feel free to substitute this with any image you'd like it's simply there to prove the point that we can bundle up images as well.
- If you provide your own image and you name it something different then you'll have to update the
- HelloWorldComponent.jsx to reflect your changes. Equally if you choose an image with a different file extension then you need to modify the test property of the image loader in the webpack.config.js with appropriate regex to match your new file extension..

**src/styles/styles.css**

```
.form {
 margin: 25px;
padding: 25px;
 border: 1px solid #ddd;
 background-color: #eaeaea;
 border-radius: 10px;
}
.form div {
 padding-top: 25px;
}
.image-container {
 display: flex;
 justify-content: center;
}
```

**index.html**

```
<!DOCTYPE html>
```

```
<html lang="en">
<head>
 <meta charset="UTF-8">
 <title>Learning to build a react pipeline</title>
</head>
<body>
 <div id="content"></div>
 <script src="app.js"></script>
</body>
</html>
```

index.jsx

```
import React from 'react';
import { render } from 'react-dom';
import HelloWorldComponent from
'./components/HelloWorldComponent.jsx';
require('./images/myImage.gif');
require('./styles/styles.css');
require('./index.html');
render(<HelloWorldComponent />,
document.getElementById('content'));
```

**Step 6**: Create webpack configuration

- ➢ Create a file called webpack.config.js in the root of your project and copy this code into it:

webpack.config.js

```
var path = require('path');
var config = {
 context: path.resolve(__dirname + '/src'),
 entry: './index.jsx',
 output: {
 filename: 'app.js',
 path: path.resolve(__dirname + '/dist'),
 },
 devServer: {
 contentBase: path.join(__dirname + '/dist'),
 port: 3000,
 open: true,
},
 module: {
 loaders: [
 {
```

```
test: /\.(js|jsx)$/,
exclude: /node_modules/,
loader: 'babel-loader'
},
{
test: /\.css$/,
loader: "style!css"
},
{
test: /\.gif$/,
loaders: [
'file?name=[path][name].[ext]',
'image-webpack',
]
```

```
 },
 { test: /\.(html)$/,
 loader: "file?name=[path][name].[ext]"
 }
 ],
 },
};
module.exports = config;
```

**Step 7:** Create npm tasks for your pipeline

To do this you will need to add two properties to the scripts key of the JSON defined in the package.json file in the
root of your project. Make your scripts key look like this:

```
 "scripts": {
```

```
"start": "webpack-dev-server",
 "build": "webpack",
 "test": "echo \"Error: no test specified\" && exit 1"
 },
```

- The test script will have already been there and you can choose whether to keep it or not, it's not important to this example.

**Step 8:**

**Use the pipeline**

- From the command line, if you are in the project root directory you should now be able to run the command:

**npm run build**

This will bundle up the little application you've built and place it in the dist/ directory that it will create in the root of your project folder.

**If you run the command:**

**npm start**

- Then the application you've built will be served up in your default web browser inside of a webpack dev server instance.

## Summary:

- Client-side Routing
- Showing views
- React Routing
- Components
- Static Routing
- Dynamic Routing
- How to setup a basic webpackreact and babel environment

Thank You………

If you have any quries please write to info@uplatz.com".