React – Kraken

In this lab, you will create a Kraken app using React as the client UI.

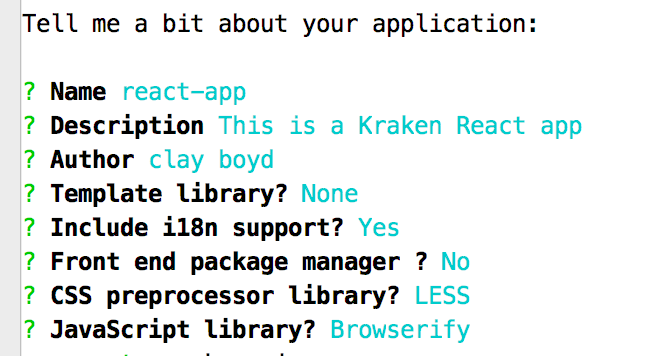
# Objectives

In this lab, you will

* Create a Kraken app named, ‘react-app’
* Add React configuration
* Add React pages
* Add React Routes
* Run the app

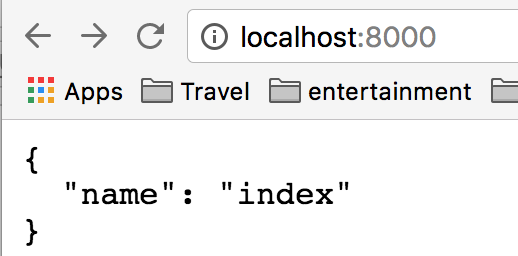
# Create a Kraken app

1. In the lab folder, create a Kraken app using yo kraken. Use the parameters as shown. NOTE: some of these are NOT the default.



# Run the Kraken app to verify correctness

1. Change to the react folder and run npm start.
2. Open the browser to localhost:8000 and the page should look like this:



1. The output comes from controllers/index.js as shown.



# Configure Kraken to use React

1. Configure config/config.json to use the react view engine. Add the following as the first element in the file.

**"express"**: {  
 **"view engine"**: **"jsx"**,  
 **"view"**: **"require:anemone-machina/lib/expressView"**,  
 **"view cache"**: **false**,  
 **"views"**: **"path:./public/views"**},  
**"view engines"**: {  
 **"jsx"**: {  
 **"module"**: **"anemone-machina/lib/server"**,  
 **"renderer"**: {  
 **"method"**: **"create"**,  
 **"arguments"**: [  
 {  
 **"routes"**: **"require:./routes.jsx"**,  
 **"routesFilePath"**: **"path:./routes.jsx"** }  
 ]  
 }  
 }  
},

1. Notice the views folder above is ./public/views, we must create this folder and insert our React components there.

cd public

mkdir views

1. Create the first file, public/views/layout.jsx, with the following content:

**'use strict'**;  
  
**var** React = require(**'react'**);  
  
module.exports = React.createClass({  
  
 render: **function** render() {  
  
 **return** (  
 <**html**>  
 <**head**>  
 <**meta charSet='utf-8'** />  
 <**title**>  
 {**this**.props.title}  
 </**title**>  
 </**head**>  
 <**body**>  
 Hello, World.  
 <**script src='/bundle.js'**></**script**>  
 </**body**>  
 </**html**>  
 );  
 }  
});

1. This is the page content. The component just renders the initial page with Hello World.
2. Create the routes file, ./routes.jsx, with the following content:

**'use strict'**;  
  
**var *React*** = *require*(**'react'**);  
**var *ReactRouter*** = *require*(**'react-router'**);  
**var *Router*** = ***ReactRouter***.**Router**;  
**var *Route*** = ***ReactRouter***.**Route**;  
**var *IndexRoute*** = ***ReactRouter***.**IndexRoute**;  
  
**var *Layout*** = *require*(**'./public/views/layout.jsx'**);  
  
**var *routes*** = ***module***.exports = (  
 <**Router**>  
 <**Route path='/' component=**{***Layout***}>  
 </**Route**>  
 </**Router**>  
);

1. The routes file maps the path, / , to the component, layout.jsx, defined above.

# Install the npm modules

1. Install the npm modules for react using the following commands:

npm install anemone-machina react react-router --save

1. Since we are using JSX, we need to install a compiler to convert it to JS. Use the following npm command:

npm install babel-register babel-preset-react --save

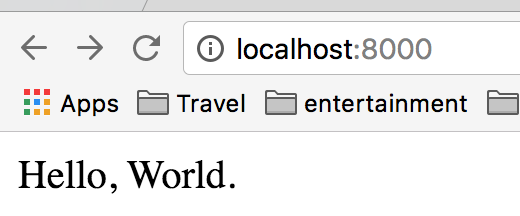
1. The above installs babel, now change the ./index.js file to require() it with:

**'use strict'**;  
  
*require*(**'babel-register'**)({  
 **presets**: [**'react'**]  
});  
  
  
**var *express*** = *require*(**'express'**);  
**var *kraken*** = *require*(**'kraken-js'**);

1. Edit the ./controllers/index.js to render a new React page. Add the res.render() call:

router.get(**'/'**, **function** (req, res) {  
   
*// res.send('<code><pre>' + JSON.stringify(model, null, 2) + '</pre></code>');* res.render( req.**url**, model );  
   
 });

1. Start the app with npm start and open the browser to <http://localhost:8000> You should see the image below:



1. Now that we have configured Kraken, we manage the React app by adding views and modifying the routers.jsx file.

# Build the ./.build/bundle.js file

When you examine the HTML generated by the /public/views/layout.jsx file, you will notice that it attempts loading the JavaScript file, /bundle.js. This file contains all the client side code required to run the React app. Without this file, the app uses server side rendering instead of client side rendering.

1. Examine the package.json file. Notice the build script refers to the command, grunt build.

**"scripts"**: {  
 **"test"**: **"grunt test"**,  
 **"build"**: **"grunt build"**,  
 **"all"**: **"npm run build && npm run test"**},

1. From the Gruntfile.js file, it appears that grunt invokes several tasks to perform the build. There’s a bug in that it invokes the task, esLint, twice!!!

*// Register group tasks*grunt.registerTask(**'build'**, [**'eslint'**, **'eslint'**, **'less'**, **'browserify'**, **'copyto'**]);  
  
grunt.registerTask(**'test'**, [ **'eslint'**, **'mochacli'** ]);

1. Notice the /tasks/browserify.js does NOT create the bundle.js file during the build.

*// Options***return** {  
 **build**: {  
 **files**: {  
 **'.build/js/app.js'**: [**'public/js/app.js'**]  
 },  
 **options**: {}  
 }  
};

1. Change this file to the following:

**'use strict'**;  
  
***module***.exports = **function** *browserify*(grunt) {  
 *// Load task* grunt.loadNpmTasks(**'grunt-browserify'**);  
  
 *// Options* **return** {  
 **build**: {  
 **src**: **'./public/main.js'**,  
 **dest**: **'./public/bundle.js'**,  
 **options**: {  
 **transform**: [**'reactify'**, **'require-globify'**]  
 }  
 }  
 };  
};

1. Using the src property above, browserify starts with ./public/main.js to determine the browser dependencies from the require() function. Create the ./public/main.js file with the following contents:

**'use strict'**;  
  
**var *Routes*** = *require*(**'../routes.jsx'**);  
**var *Client*** = *require*(**'anemone-machina/lib/client'**);  
  
*// Include all view files. Browserify doesn't do  
// this automatically as it can only operate on  
// static require statements.  
require*(**'./views/\*\*/\*.jsx'**, {**glob**: **true**});  
  
*// boot options***var *options*** = {  
 **routes**: ***Routes***,  
 *// supply a function that can be called  
 // to resolve the file that was rendered.* viewResolver: **function**(viewName) {  
 **return** *require*(**'./views/'** + viewName);  
 }  
};  
  
**document**.addEventListener(**'DOMContentLoaded'**, **function** *onLoad*() {  
 ***Client***.boot(***options***);  
});

1. The main.js file require()s the browser dependencies which browserify uses to find all the client frameworks. Note that it must also require() the view files or browserify will not load them into the bundle. This file is the last file catenated in the bundle.js. Therefore, it will be the first one executed AFTER loading all the framework files.
2. This will create the bundle.js and requires a couple of plugins to help.

npm install reactify require-globify --save-dev

npm install history --save

1. Now build the bundle with:

grunt build

1. And restart the application with:

npm start

1. This worked so well that we should change the way npm start works. Modify the scripts in package.json to add the start script.

**"scripts"**: {  
 **"start"**: **"npm run build && node server.js"**,  
 **"test"**: **"grunt test"**,  
 **"build"**: **"grunt build"**,  
 **"all"**: **"npm run build && npm run test"**},

1. Now when we run, npm start, it will first do a build, processing our JSX files and bundling all the client side JavaScript and then start the application.

npm start

Congratulations. You have completed this lab.