19.3 Lesson Plan - #LearnToMERN

Instructor Notes:

In this class, we’ll be showing students how to integrate their React applications with a Node, Express, and MongoDB back-end. Towards the end of the course, we will further introduce students to the new concepts of ES6.

Instructor Priorities:

- Students should firmly understand how to build a complete MERN (Mongo, Express, React, and Node) application.
- Students should understand both the concept and utility of React Router
- Students should gain a cursory knowledge of ES6.

Instructor Notes:

- Be expecting today’s class to be a fairly challenging one. The transition from React to MERN involves significantly more complex applications as students’ simple React applications will now need to make use of an Node Server, Express API, and MongoDB database. For many students, “getting started” and “getting organized” will present a huge opportunity for confusion.
- Despite, the challenge these concepts ARE important. Students will need to understand both routing and MERN integration to complete

the recommended homework assignment.

- Instructors/TAs: Spend the appropriate amount of time outside of class familiarizing yourself with the code – particularly the `AddressMERN` example. Students will work on this activity for close to an hour of class, it's important that they feel supported throughout that time.
- Towards the latter-end of class, students will also be exposed to ES6 concepts. This exposure serves a dual-purpose: 1) Many job openings make mention of ES6 as being a desired trait and 2) Many of the tutorials for React Native utilize ES6 syntax.
- Have your TAs reference 19.3-TimeTracker to help keep track of time during class.

Class Objectives

- To obtain a firm grasp of the MERN stack (MongoDB, Express, React, and Node).
- To understand the concept of front-end routing with React Router
- To gain an initial exposure to ES6 Javascript.

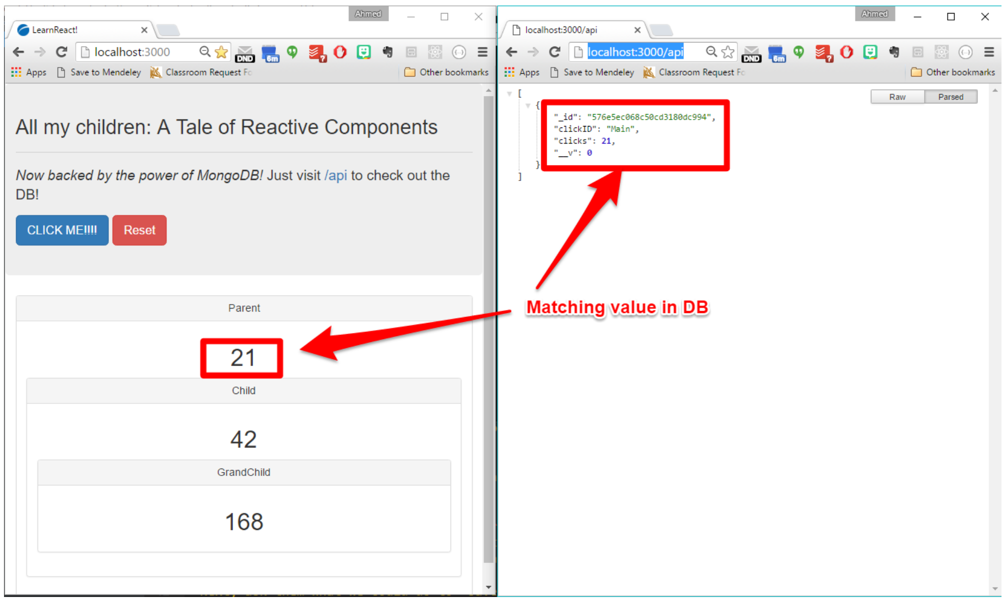
---

1. Instructor Do: Intro to MERN (0:15)

- Welcome students back to class and proceed to explain to students that today they will be learning about the MERN stack. Explain to them what each letter of the acronym means (MongoDB, Express, React, Node) and what a "stack" means in the first place (a stack is a set of complimentary technologies that can be used to build full stack applications).
- Describe how we'll be using ReactJS to handle the front-end of our application and Mongo, Express, and Node will handle the backend. If necessary create a diagram here to help make this clear.
- Then proceed to open the folder `01-IntroMERN > Unsolved`. Open the file `index.html` inside of the `public` folder. You should see the Click Counter application from before. Now, click the button a few times for students to see. Then close the browser. Ask students what the count will be when you re-open the file. (They should answer: 0). Re-open the browser, and show them that the value is in fact 0.
- Next, ask them what we could do to "save" the click count perpetually. (Possible Answers: LocalStorage or Database).
- Once you've heard the correct answers, open the folder `01-IntroMERN > Solved`. Then run `npm install` and finally run `node server.js`. Now open up a webrowser and visit `localhost:3000`. You should see our Click counter from before. Click the button a few times. Then close the browser and re-open it. This

time you should see the click count preserved.

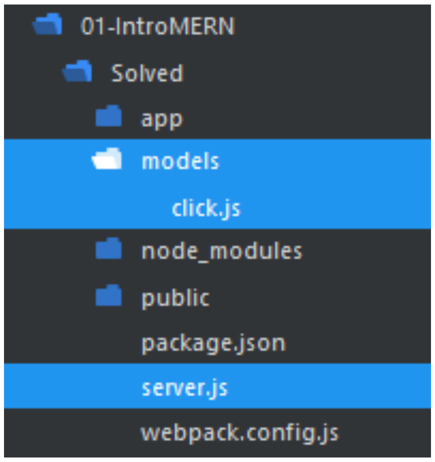
- Now, visit the page `localhost:3000/api` (while also on the main screen). Click the Click button a few times, then refresh the api page. Point out to students that as expected our application is updating the database and our api reflects that. Close the browser and re-open the page. Show that it preserves the number.
-


LearnReact!
localhost:3000
All my children: A Tale of Reactive Components
Now backed by the power of MongoDB! Just visit /api to check out the DB!
CLICK ME!!!!
Reset
Parent
21
Child
42
GrandChild
168
localhost:3000/api
Raw
Parsed
Matching value in DB

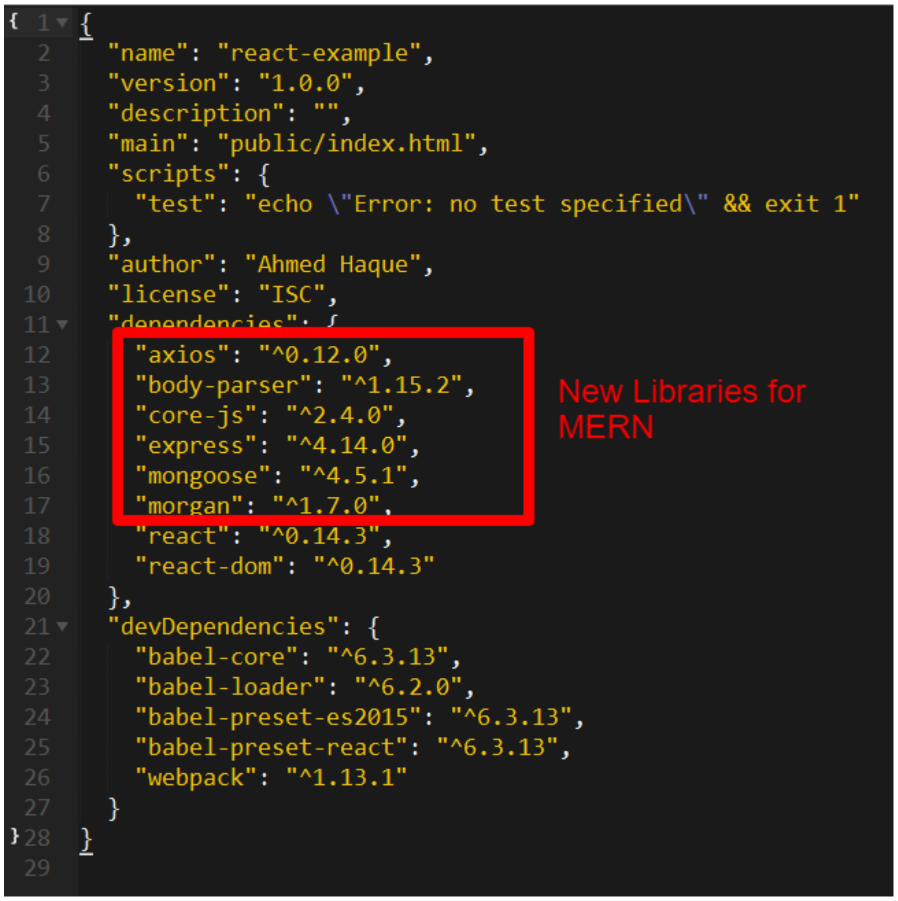
```
[
  {
    "_id": "576e5ec068c50cd3180dc994",
    "clickID": "Main",
    "clicks": 21,
    "__v": 0
  }
]
```

Images/1-MERN_1.png

- Answer any questions before proceeding to the code.
- **TAs:** Sack out the `SOLVED` and `UNSOLVED` code at this point, so students can follow-along with the differences.
- Open the `SOLVED` folder in Sublime. Then walk students through the key points of the MERN application:

- Begin by pointing out the folder structure, specifically the addition of the `server.js` file and the `click.js` model. Let students that these two files were the ONLY new files in the application.
- 

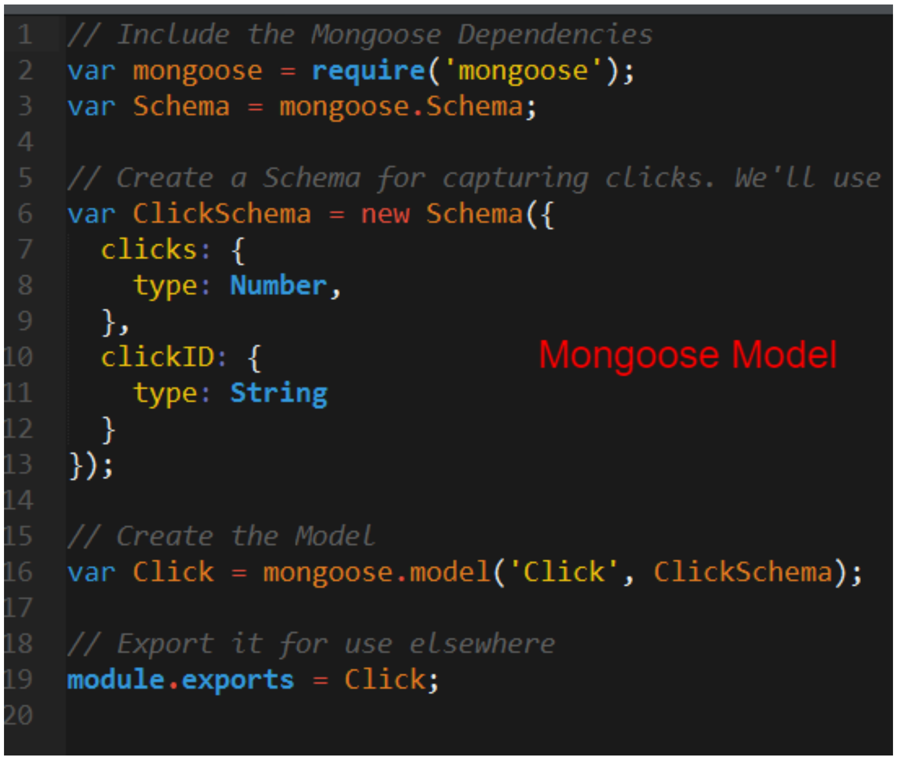
  Images/2-MERN_2.png
- Point out the new npm packages added. These are the same ones we'd expect in working with MongoDB and Express – packages like: `Express`, `Mongoose`, `Morgan`, and `body-parser`. We've also added in this package called `Axios` which we'll use to run HTTP requests.
-



```
{
  "name": "react-example",
  "version": "1.0.0",
  "description": "",
  "main": "public/index.html",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "Ahmed Haque",
  "license": "ISC",
  "dependencies": {
    "axios": "^0.12.0",
    "body-parser": "^1.15.2",
    "core-js": "^2.4.0",
    "express": "^4.14.0",
    "mongoose": "^4.5.1",
    "morgan": "^1.7.0",
    "react": "^0.14.3",
    "react-dom": "^0.14.3"
  },
  "devDependencies": {
    "babel-core": "^6.3.13",
    "babel-loader": "^6.2.0",
    "babel-preset-es2015": "^6.3.13",
    "babel-preset-react": "^6.3.13",
    "webpack": "^1.13.1"
  }
}
```

Images/2-MERN_3.png

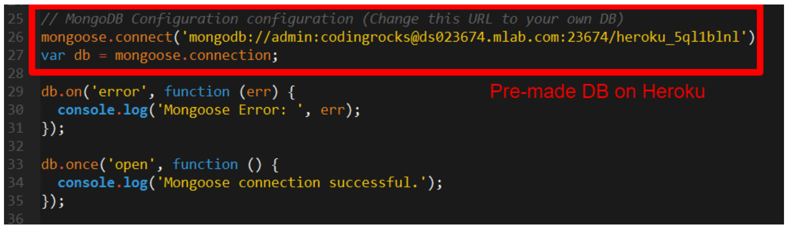
- Next, open the `click.js` model. Ask students to walk you through the code or explain it for them. The code is nothing more than a simple Mongoose schema and model. The one key point is that we've created a `clickID` property which we'll be using to update the same click counter every time.
-



```
// Include the Mongoose Dependencies
var mongoose = require('mongoose');
var Schema = mongoose.Schema;

// Create a Schema for capturing clicks. We'll use
var ClickSchema = new Schema({
  clicks: {
    type: Number,
  },
  clickID: {
    type: String
  }
});

// Create the Model
var Click = mongoose.model('Click', ClickSchema);

// Export it for use elsewhere
module.exports = Click;
```

Images/2-MERN_4.png

- Next, open the `server.js` file. Point out how it's setup like most of our other servers. Remind students that this means:
  - We've included our server dependencies at the top. We've included
  - Included the configurations for packages like `body-parser` and set up our `static` folder.
  - Included a link to our MongoDB database. (In this case, we pre-created this database using Heroku's mLab).
  -



```
// MongoDB Configuration configuration (Change this URL to your own DB)
mongoose.connect('mongodb://admin:codingrocks@ds023674.mlab.com:23674/heroku_5ql1blnl')
var db = mongoose.connection;

db.on('error', function (err) {
  console.log('Mongoose Error: ', err);
});

db.once('open', function () {
  console.log('Mongoose connection successful.');
});
```

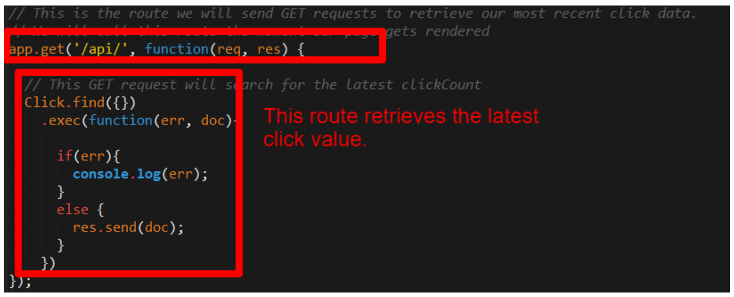
Images/2-MERN_5.png

- Then spend a significant amount of time walking students through the routing. Point out that we have three routes.

  1. A central GET '/' route. This is to point our server immediately to the React.js generated `index.html` file.

  2. An api based GET `/api` route. This route will be used to retrieve the click count. Here we use a simple mongoose `Click.find({})` to achieve this.

  3.



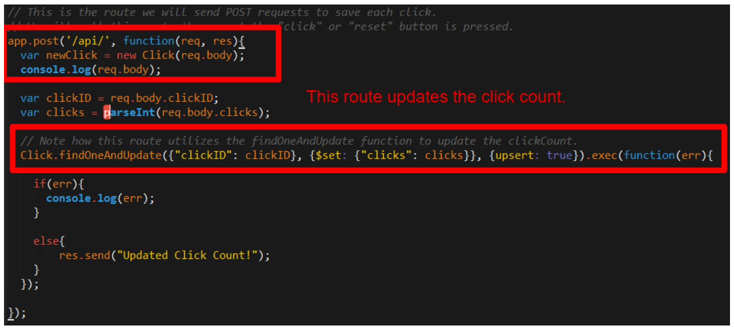
```
// This is the route we will send GET requests to retrieve our most recent click data.
// We will call this route the moment our page gets rendered
app.get('/api/', function(req, res) {

  // This GET request will search for the latest clickCount
  Click.find({})
    .exec(function(err, doc){

      if(err){
        console.log(err);
      }
      else {
        res.send(doc);
      }
    })
});
```

Images/2-MERN_6.png

  4. An api based POST `/api` route. This route will be used to UPDATE the click count each time. Here we

use a `findOneAndUpdate` function to find the click counter by its clickID and then change the click count.

5.

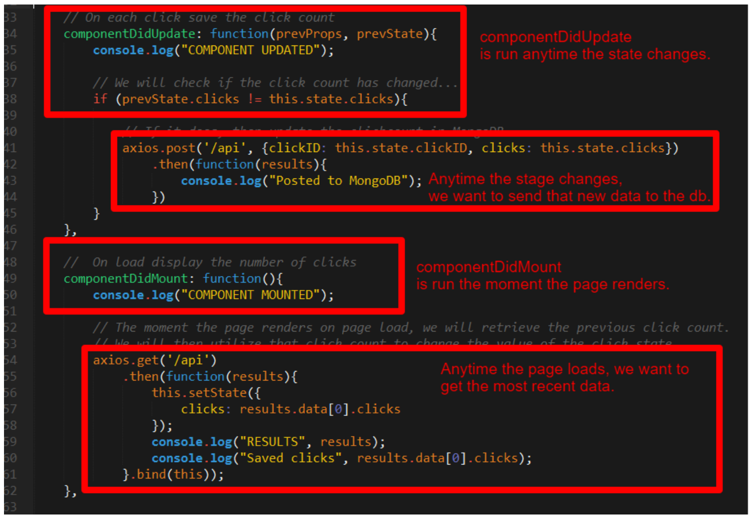


```
// This is the route we will send POST requests to save each click.
"click" or "reset" button is pressed.
app.post('/api/', function(req, res){
  var newClick = new Click(req.body);
  console.log(req.body);

  var clickID = req.body.clickID;
  var clicks = parseInt(req.body.clicks);

  // Note how this route utilizes the findOneAndUpdate function to update the clickCount.
  Click.findOneAndUpdate({"clickID": clickID}, {$set: {"clicks": clicks}}, {upsert: true}).exec(function(err){

    if(err){
      console.log(err);
    }

    else{
        res.send("Updated Click Count!");
    }
  });

});
```

Images/2-MERN_7.png

- Now open the file `Parent.js` inside of the `Components` folder. Point out how we added in a `ComponentDidUpdate` and `ComponentDidMount` lifecycle events. Effectively these are functions that will get run whenever the page loads or whenever a state updates. In discussing these functions, point out the use of the `axios` functions. These are for performing the HTTP requests to the Express API.

-



```
    // On each click save the click count
    componentDidUpdate: function(prevProps, prevState){
        console.log("COMPONENT UPDATED");

        // We will check if the click count has changed...
        if (prevState.clicks != this.state.clicks){

            // If it does, then update the clickcount in MongoDB
            axios.post('/api', {clickID: this.state.clickID, clicks: this.state.clicks})
                .then(function(results){
                    console.log("Posted to MongoDB");
                })
        }
    },

    //  On load display the number of clicks
    componentDidMount: function(){
        console.log("COMPONENT MOUNTED");

        // The moment the page renders on page load, we will retrieve the previous click count.
        // We will then utilize that click count to change the value of the click state
        axios.get('/api')
            .then(function(results){
                this.setState({
                    clicks: results.data[0].clicks
                });
                console.log("RESULTS", results);
                console.log("Saved clicks", results.data[0].clicks);
            }.bind(this));
    },
```

componentDidUpdate
is run anytime the state changes.
Anytime the stage changes,
we want to send that new data to the db.
componentDidMount
is run the moment the page renders.
Anytime the page loads, we want to
get the most recent data.

Images/2-MERN_8.png

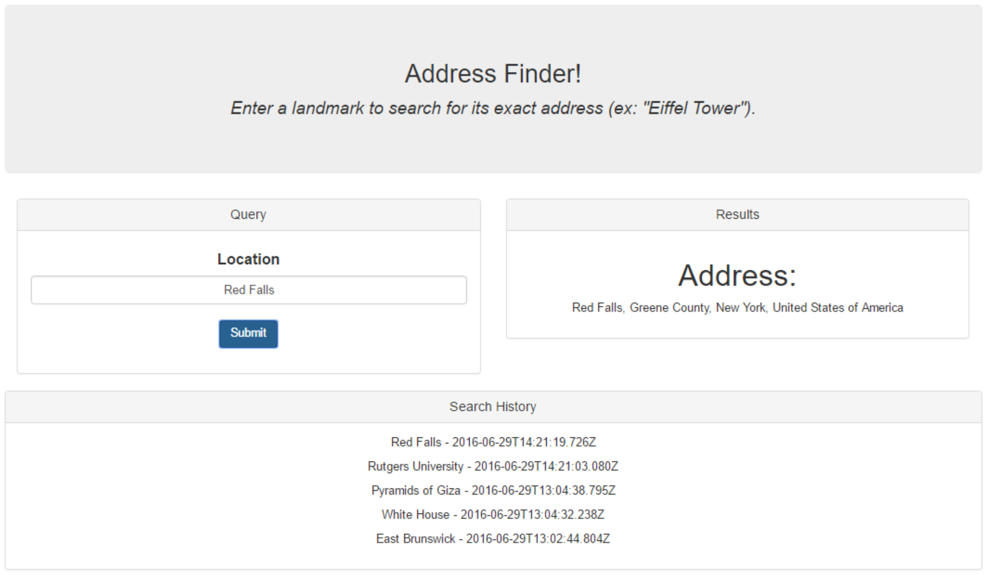
- Finally, re-run the application again for students so they can see how it works. As you navigate between the routes, point out to students how the results correlate with the code.

2. Partners Do: Discuss Intro to MERN (0:10)

- Have students form up into groups of four. Task them with listing out every one of the code changes required to enable Mongo, Express, and React integration.
- Encourage them to look at the code while completing this activity.
- When time is up have them share with the class.

3. Partners Do: Address MERN (1:15)

- Now, take a moment to warn them that this next activity will be challenging – but is very similar to the recommended homework assignment.

- Open the file `index.html` inside of `02-AddressMERN-Students` (`UNSOLVED`) in your browser. Show students that this application takes any city, location, zip code, or landmark and retrieves an understandable address.
- Now navigate to the folder `02-AddressMERN-Students` (`SOLVED`) and run `npm install` and `node server.js`. Open your browser and visit `localhost:3000`. Run the same search, but this time, point out how the application not only shows the search result – but also adds the search to the list of historic searches.
- 
Address Finder!
Enter a landmark to search for its exact address (ex: "Eiffel Tower").
Query
Location
Red Falls
Submit
Results
Address:

Red Falls, Greene County, New York, United States of America

Search History

Red Falls - 2016-06-29T14:21:19.726Z
Rutgers University - 2016-06-29T14:21:03.080Z
Pyramids of Giza - 2016-06-29T13:04:38.795Z
White House - 2016-06-29T13:04:32.238Z
East Brunswick - 2016-06-29T13:02:44.804Z

Images/3-Address_1.png

- Let them know that they will be taking the first app and incorporating the History component into their application.

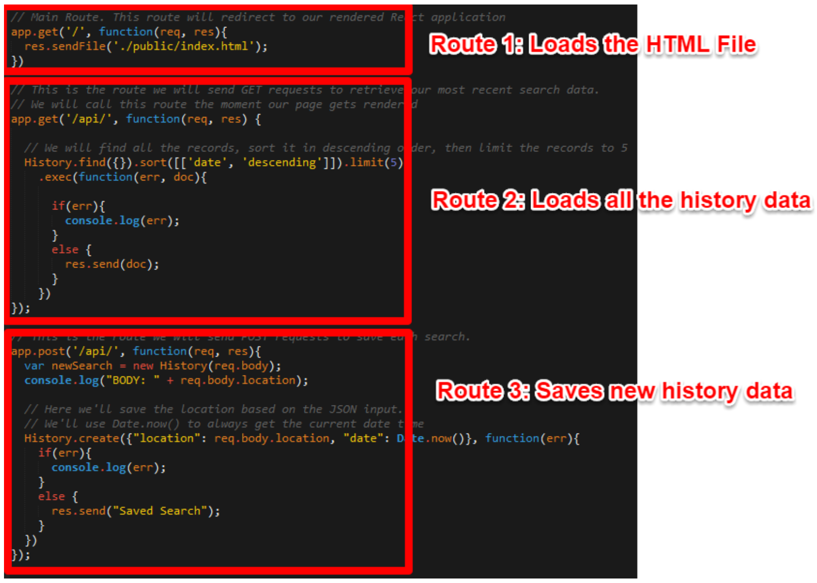
- Have groups form into 4 members then slack out the following files and instructions.
- **File:**
    - `UNSOLVED.zip` (`02-AddressMERN-Students`)
    - Screenshot of working app.
- **Instructions:**
    - Over the course of the next 1 hour and 15 minutes, you and your group will be “upgrading” the React application provided to incorporate MongoDB and Server integration. Your final application, should include a new React component for listing out the last five search terms and the datetime in which they were searched.
    - The activity is challenging, but it is very much a REAL-WORLD problem similar to ones you will have to solve in your career, so take your best shot!
    - As you proceed, consider the following as advice:
        - Divide your team into pairs. One pair will work on creating the front-end of the React application with the ability to capture data input. The second will create an initial server and API. Once you have both these pieces, work together to create the helper functions necessary to make HTTP requests to your API.

        - Remember the general steps involved with incorporating a database into a React application:
            - Add a server.js file.
            - Update your package list.
            - Add a database.
            - Add a model.
            - Add a set of API routes.
            - Create lifecycle events.
            - Use Axios to perform HTTP requests onto API.
            - Test, debug, test, debug, test, debug...
    - Reach out to the Instructors / TAs as you have questions.
    - Push yourselves! This will be a hard hour!

4. Instructor Do: Review Address MERN (0:15)

- Have TAs slack out the solution, prior to the discussion.
- Then spend the appropriate time necessary to discuss the following:
    - The general file structure of the final application – noting the inclusion of a `helper.js` file to more neatly organize all of the HTTP calls.
    - The routes found in the `server.js` file – specifically

mentioning that three routes were necessary: One to send the HTML file, one to load the history data, and one to post the history data. If you'd like you can also walk them through the Mongoose query for limiting the response to only the most recent five search terms.

- 

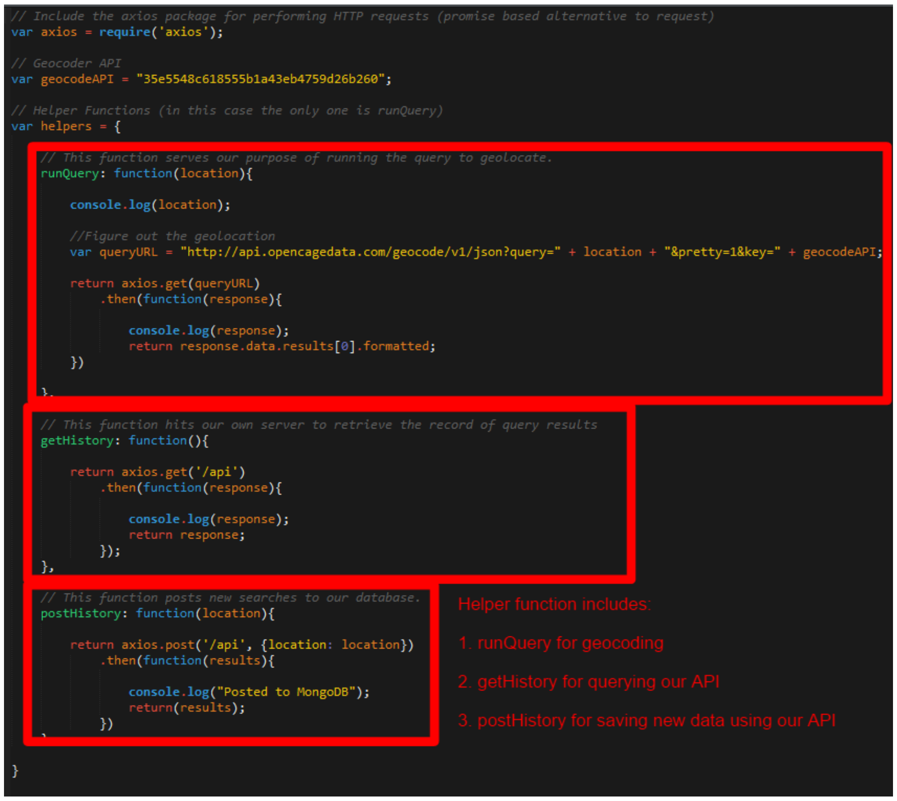
  ```
  // Main Route. This route will redirect to our rendered Re ct application
  app.get('/', function(req, res){
    res.sendFile('./public/index.html');
  })

  // This is the route we will send GET requests to retrieve our most recent search data.
  // We will call this route the moment our page gets render d
  app.get('/api/', function(req, res) {

    // We will find all the records, sort it in descending o der, then limit the records to 5
    History.find({}).sort([['date', 'descending']]).limit(5)
      .exec(function(err, doc){

        if(err){
          console.log(err);
        }
        else {
          res.send(doc);
        }
      })
  });

  // This is the route we will send POST requests to save ea h search.
  app.post('/api/', function(req, res){
    var newSearch = new History(req.body);
    console.log("BODY: " + req.body.location);

    // Here we'll save the location based on the JSON input.
    // We'll use Date.now() to always get the current date t me
    History.create({"location": req.body.location, "date": D te.now()}, function(err){
      if(err){
        console.log(err);
      }
      else {
        res.send("Saved Search");
      }
    })
  });
  ```

  Route 1: Loads the HTML File
  Route 2: Loads all the history data
  Route 3: Saves new history data

  Images/3-AddressSolved_1.png

- Point out the basic Mongoose model.

-



```javascript
// Include the axios package for performing HTTP requests (promise based alternative to request)
var axios = require('axios');

// Geocoder API
var geocodeAPI = "35e5548c618555b1a43eb4759d26b260";

// Helper Functions (in this case the only one is runQuery)
var helpers = {

    // This function serves our purpose of running the query to geolocate.
    runQuery: function(location){

        console.log(location);

        //Figure out the geolocation
        var queryURL = "http://api.opencagedata.com/geocode/v1/json?query=" + location + "&pretty=1&key=" + geocodeAPI;

        return axios.get(queryURL)
            .then(function(response){

                console.log(response);
                return response.data.results[0].formatted;
        })

    },
    // This function hits our own server to retrieve the record of query results
    getHistory: function(){

        return axios.get('/api')
            .then(function(response){

                console.log(response);
                return response;
            });
    },
    // This function posts new searches to our database.
    postHistory: function(location){

        return axios.post('/api', {location: location})
            .then(function(results){

                console.log("Posted to MongoDB");
                return(results);
            })
    }

}
```

Helper function includes:

1. runQuery for geocoding

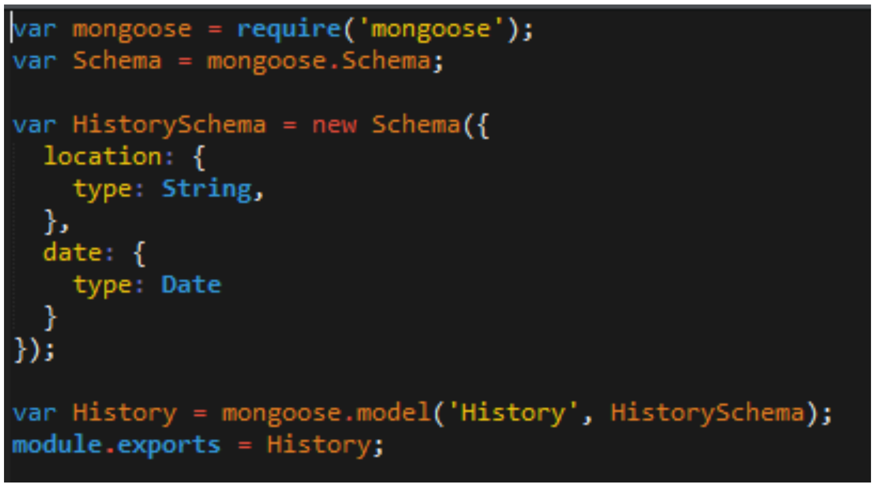
2. getHistory for querying our API

3. postHistory for saving new data using our API

Images/3-AddressSolved_2.png

- Open the `helper.js` file and walk students through the three functions we use to interact with the API shown previously. This code will require them to look at more closely, but the basic gist is that we use the axios library to perform API calls and return the data back to the React component.

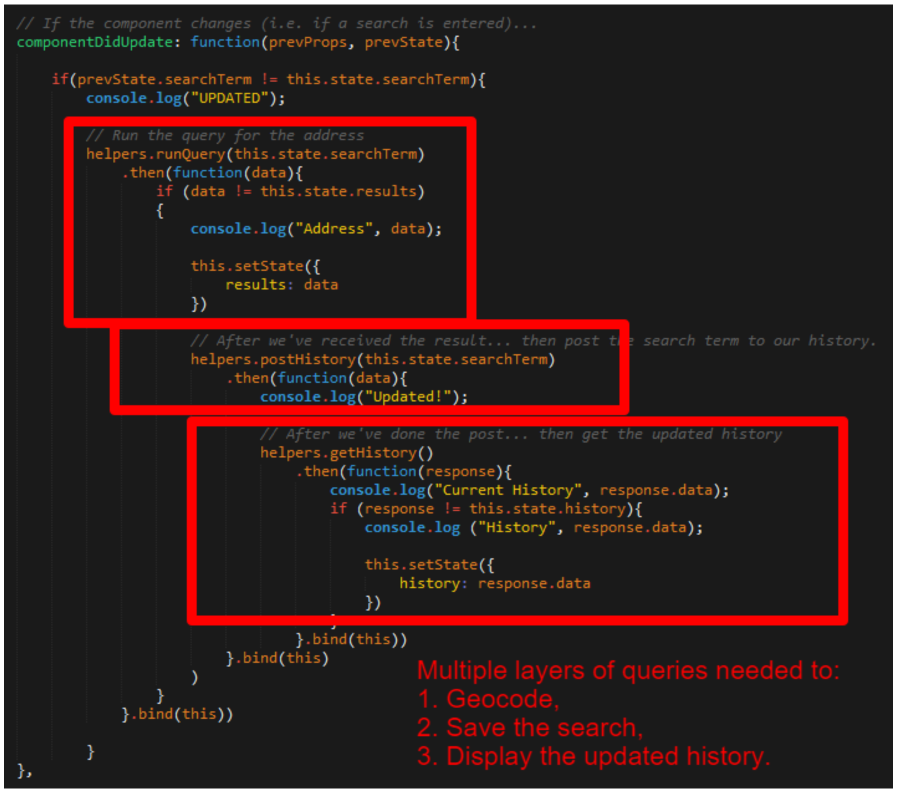
-



```
var mongoose = require('mongoose');
var Schema = mongoose.Schema;

var HistorySchema = new Schema({
  location: {
    type: String,
  },
  date: {
    type: Date
  }
});

var History = mongoose.model('History', HistorySchema);
module.exports = History;
```

Images/3-AddressSolved_3.png

- Open the `Main.js` file and point out how the helper functions are used anytime there is a component change (i.e. anytime a search is performed).
-

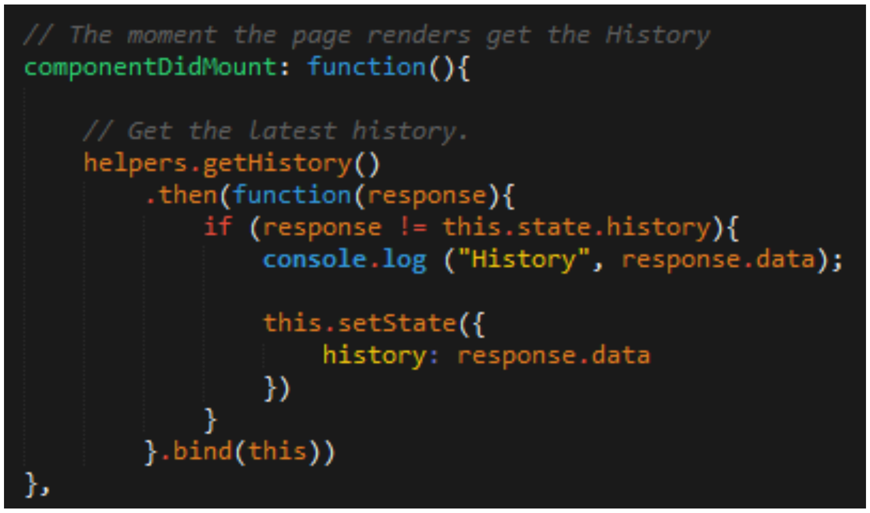


```
// If the component changes (i.e. if a search is entered)...
componentDidUpdate: function(prevProps, prevState){

    if(prevState.searchTerm != this.state.searchTerm){
        console.log("UPDATED");

        // Run the query for the address
        helpers.runQuery(this.state.searchTerm)
            .then(function(data){
                if (data != this.state.results)
                {
                    console.log("Address", data);

                    this.setState({
                        results: data
                    })

                    // After we've received the result... then post the search term to our history.
                    helpers.postHistory(this.state.searchTerm)
                        .then(function(data){
                            console.log("Updated!");

                            // After we've done the post... then get the updated history
                            helpers.getHistory()
                                .then(function(response){
                                    console.log("Current History", response.data);
                                    if (response != this.state.history){
                                        console.log ("History", response.data);

                                        this.setState({
                                            history: response.data
                                        })
                                    }
                                }.bind(this))
                        }.bind(this)
                    )
                }
            }.bind(this))

        }
},
```

Multiple layers of queries needed to:
1. Geocode,
2. Save the search,
3. Display the updated history.

Images/3-AddressSolved_4.png

- Point out that we also run the api call to get search results on the initial page load using `componentDidMount`
-



```
// The moment the page renders get the History
componentDidMount: function(){

    // Get the latest history.
    helpers.getHistory()
        .then(function(response){
            if (response != this.state.history){
                console.log ("History", response.data);

                this.setState({
                    history: response.data
                })
            }
        }.bind(this))
},
```

Images/3-AddressSolved_5.png

5. Partners Do: Review MERN Solution (0:10)

- Give students a few moments to review the solution before proceeding to break. Answer any questions that arise.
- Let them know it was a hard activity and that they should invest time outside to better understand this example.

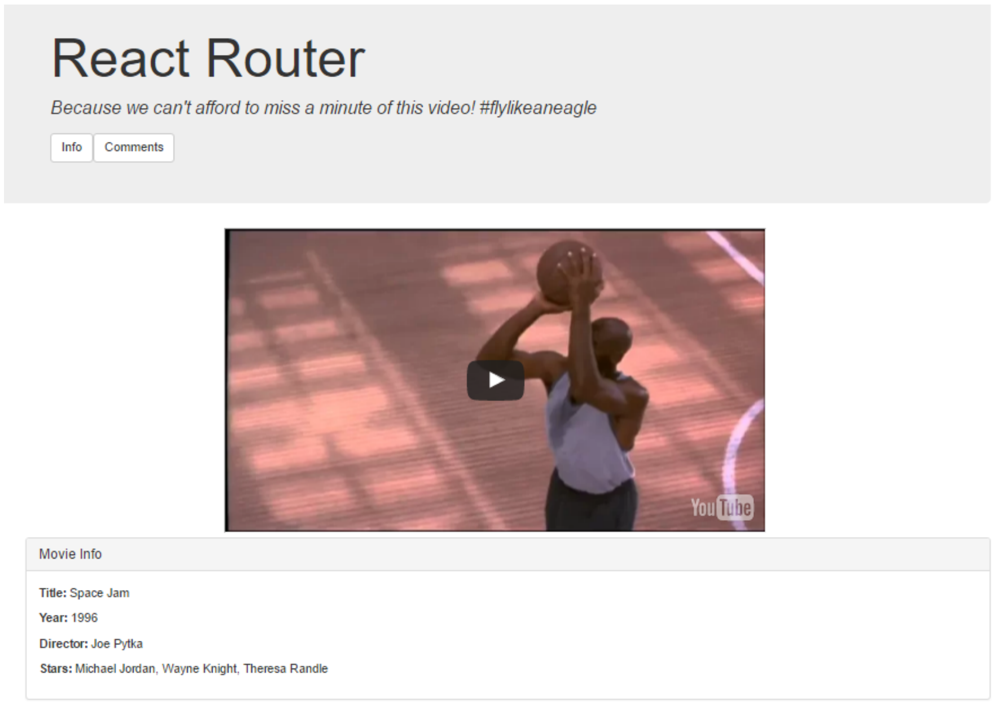
---

6. BREAK (0:40)

---

7. Instructor Do: Intro Routing and Single Page Applications (0:10)

- Begin the next example in class, by visiting YouTube and searching for a random video. Play the video, and as it’s playing click on various buttons on the Share, Embed, and Email buttons. Point out that the video isn’t stopped while we flip between sub-pages.
- Now open the file `info.html` inside of `03-IntroRouting` in your browser. Start the video, then click on the “Comments” button. Point out to students that the page refreshed – totally destroying the MJ Experience. Point out that this is the problem of using “hard links” to move between sub-pages in web development.
- Use this visual to introduce the concept of Single-Page Applications. Define an SPA as a web application that loads a single HTML page and dynamically updates the page with user interactivity.
-



React Router

Because we can't afford to miss a minute of this video! #flylikeaneagle

Movie Info

Title: Space Jam

Year: 1996

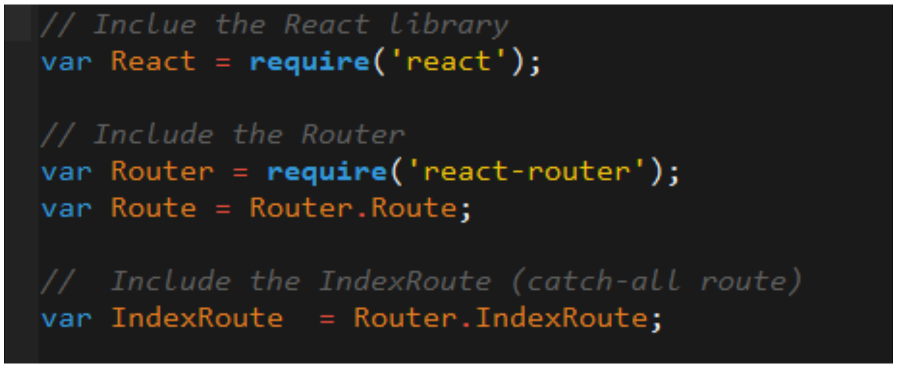
Director: Joe Pytka

Stars: Michael Jordan, Wayne Knight, Theresa Randle

Images/4-SPA_4.png

- Then open the file `index.html` inside of `2-SPA-React`. Play the video again and repeat the click experiment. Point out to students that in this example, the top part of the page is left unaffected by the user clicking on other parts. Let them know that this is a powerful concept that let's us build rich web applications. Also let them know that in order to achieve this we'll be using React-Router, a library which tracks the URL in React – and then uses React to interpret what displays (as opposed to the server sending a new HTML file).
- Open the folder `2-SPA-React` in Sublime. Then walk students through the code. Focus on the following as you do so:
  - The inclusion of the `Router`, `Route`, and `IndexRoute` dependencies inside of the `Route.js` file. (Note: React router

is also inside of the `package.json` file).

- 

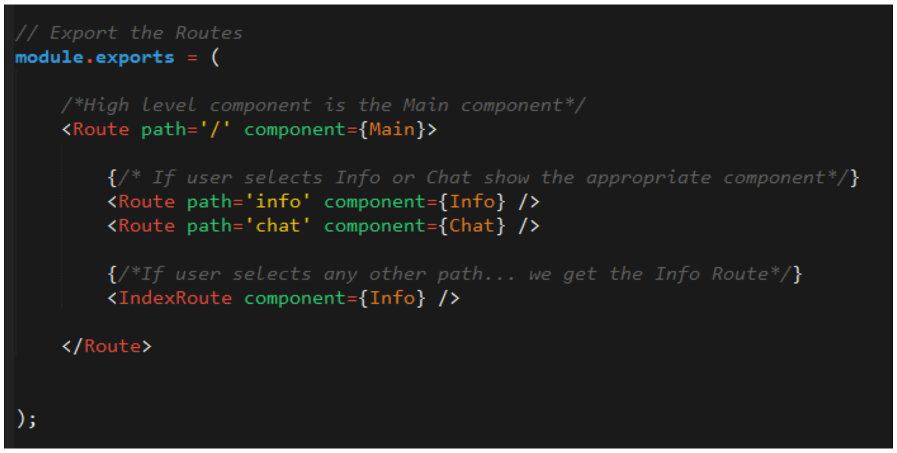
  ```
  // Inclue the React library
  var React = require('react');

  // Include the Router
  var Router = require('react-router');
  var Route = Router.Route;

  //  Include the IndexRoute (catch-all route)
  var IndexRoute  = Router.IndexRoute;
  ```

  Images/4-SPA_1.png

- Open the file `app.js` and point to students that instead of rendering the usual `Main` or `Parent` components, here we are rendering `routes`.

- Then focus your attention on the `routes.js` file. Show students that this file exports a set of routes with paths and components. Essentially these routes specify which component is rendered when. By nesting routes inside of others, we can further specify which children are rendered when.

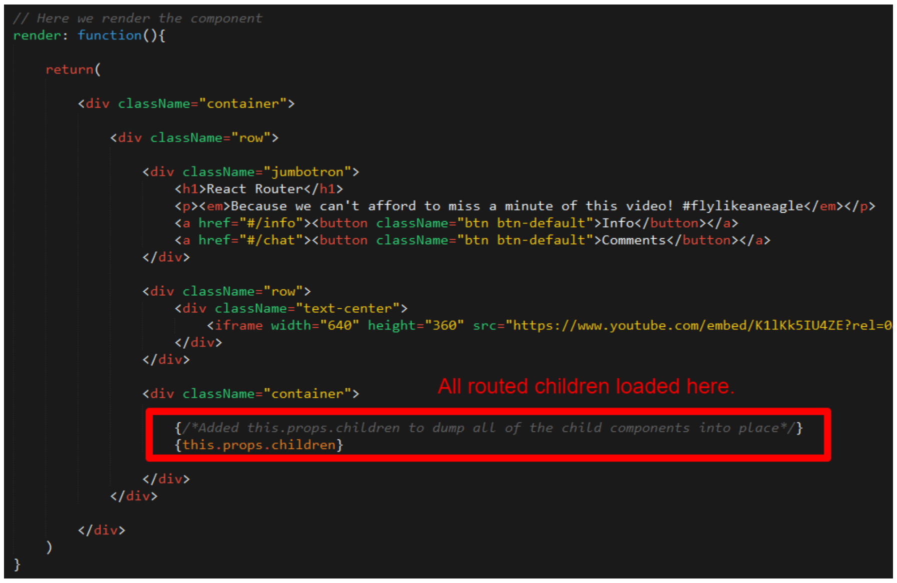
-



```
// Export the Routes
module.exports = (

    /*High level component is the Main component*/
    <Route path='/' component={Main}>

        {/* If user selects Info or Chat show the appropriate component*/}
        <Route path='info' component={Info} />
        <Route path='chat' component={Chat} />

        {/*If user selects any other path... we get the Info Route*/}
        <IndexRoute component={Info} />

    </Route>

);
```

Images/4-SPA_2.png

- Now open the file `Main.js`. Point out to students the line `this.props.children`. This line of code effectively serves as a placeholder, for which the "children" of this component are dumped. How do we know who the children of `Main.js` are? It's what we specified in `routes.js` via nesting.
-



```
// Here we render the component
render: function(){

    return(

        <div className="container">

            <div className="row">

                <div className="jumbotron">
                    <h1>React Router</h1>
                    <p><em>Because we can't afford to miss a minute of this video! #flylikeaneagle</em></p>
                    <a href="#/info"><button className="btn btn-default">Info</button></a>
                    <a href="#/chat"><button className="btn btn-default">Comments</button></a>
                </div>

                <div className="row">
                    <div className="text-center">
                        <iframe width="640" height="360" src="https://www.youtube.com/embed/K1lKk5IU4ZE?rel=0
                    </div>
                </div>

                <div className="container">

                    {/*Added this.props.children to dump all of the child components into place*/}
                    {this.props.children}

                </div>
            </div>

        </div>
    )
}
```

Images/4-SPA_3.png

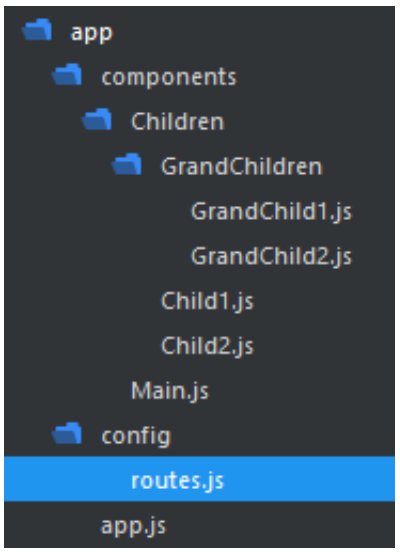
- Now if you re-run the code and flip between the links, you can show how the child rendered matches the one clicked on as specified by the routes.

8. Students Do: MyFirstRoute (0:35)

- Prepare students for a second challenge. Open the file `index.html` in `04-MyFirstRoute-Students` in your browser. Click on the buttons at the top and show how it changes the panel on the bottom. Also show that Child #1 has its own set of buttons that can cause a flip between two of its own children. Let students know that they will be building this for the next 35 minutes.
- Slack out the following file and instructions:
- **File:**
    - `index_design.html` (`04-MyFirstRoute-Students`)
    - Screenshot of working app
- **Instructions:**
    - Spend the next 35 minutes taking the design provided to you and breaking it into routed child and grandchild components.
    - As a starting point, create a basic route that flips between two panels on button click.
    - As a bonus, nest buttons and panels inside one of these child panels. The buttons should flip between these “grandchild” panels.

    - This may be tricky! Fight through it!

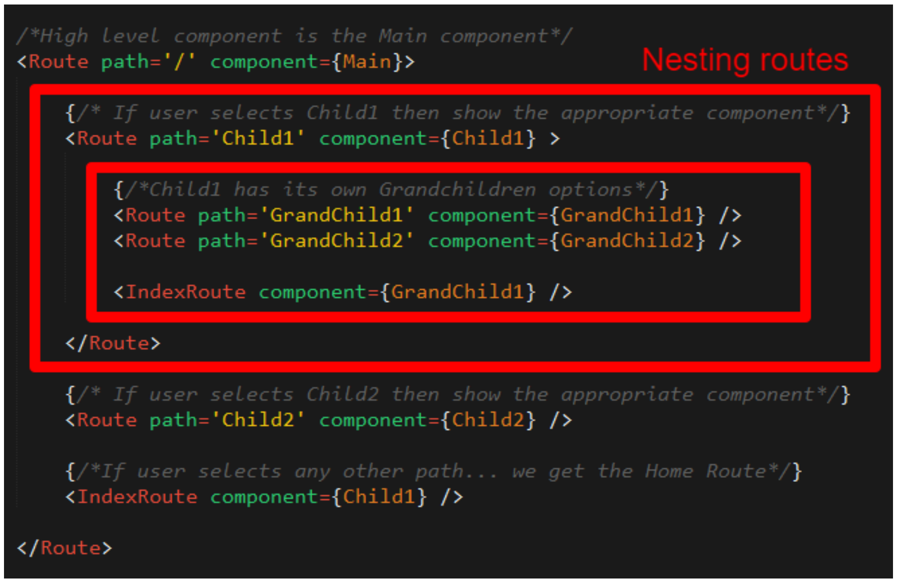
9. Instructor Do: Review MyFirstRoute (0:10)

- Once time is up, begin your discussion.
    - Start by pointing out the overall file structure. Pointing out all the files you needed.
    - 

```
app
  components
    Children
      GrandChildren
        GrandChild1.js
        GrandChild2.js
      Child1.js
      Child2.js
    Main.js
  config
    routes.js
  app.js
```

Images/5-StudentRoute_1.png

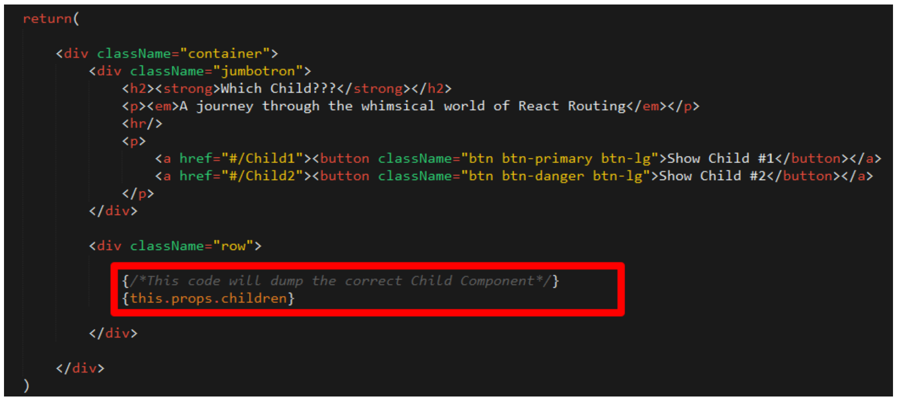
    - Then point out the code inside of `routes.js`. Point out how it has routes nested inside of routes.
    -



```
/*High level component is the Main component*/
<Route path='/' component={Main}>

    {/* If user selects Child1 then show the appropriate component*/}
    <Route path='Child1' component={Child1} >

        {/*Child1 has its own Grandchildren options*/}
        <Route path='GrandChild1' component={GrandChild1} />
        <Route path='GrandChild2' component={GrandChild2} />

        <IndexRoute component={GrandChild1} />

    </Route>

    {/* If user selects Child2 then show the appropriate component*/}
    <Route path='Child2' component={Child2} />

    {/*If user selects any other path... we get the Home Route*/}
    <IndexRoute component={Child1} />

</Route>
```

Images/5-StudentRoute_2.png

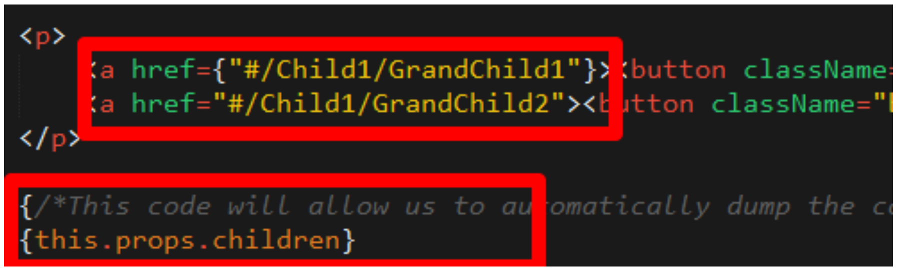
- Then open the file `Main.js`. Point out how `this.props.children` is used. Also point out the `href` and how it points to `#/Child1/GrandChild1` or `#/Child1/GrandChild2/`
-



```
return(

    <div className="container">
        <div className="jumbotron">
            <h2><strong>Which Child???</strong></h2>
            <p><em>A journey through the whimsical world of React Routing</em></p>
            <hr/>
            <p>
                <a href="#/Child1"><button className="btn btn-primary btn-lg">Show Child #1</button></a>
                <a href="#/Child2"><button className="btn btn-danger btn-lg">Show Child #2</button></a>
            </p>
        </div>

        <div className="row">

            {/*This code will dump the correct Child Component*/}
            {this.props.children}

        </div>

    </div>
)
```

Images/5-StudentRoute_3.png

- Then open the file `Child1.js`. Point out how `this.props.children` is used. Also point out the `href` and how it points to `#/Child1` or `#/Child2/`

- 

```
<p>
    <a href={"#/Child1/GrandChild1"}><button className=
    <a href="#/Child1/GrandChild2"><button className="
</p>

{/*This code will allow us to automatically dump the c
{this.props.children}
```

Images/5-StudentRoute_4.png

- Answer any questions that remain, before slacking out a zip of the solution.

10. Partners Do: Research ES6 (0:10)

- Take a moment to transition away from React Routing and MERN for a moment – to instead move to ES6. Let students know that ES6 is the latest standard of Javascript and that it opens up some new features. Let them know that learning ES6 is important for two reasons: 1) It’s a frequent desirable skill on job openings and 2) It’s very common on React Native tutorials.
- Slack out the following instructions:
    - With your partners: Find at least three new ES6 features and explain them to one another. Be prepared to share with the group.
- Once time is up, go over these features with students.

11. Instructor Do: ES6 Demos (0:10)

- Then proceed to open up the file `comparison.js` inside of `05-ES6Demos`. Walk students through the various new changes to ES6. The most useful ones to point out are likely “Template Literals”, “Fat Arrow”, and “Classes”. Don’t spend too much time here. It’s sufficient to just vaguely explain and slack out the code – letting students know that this is all extra information for them.

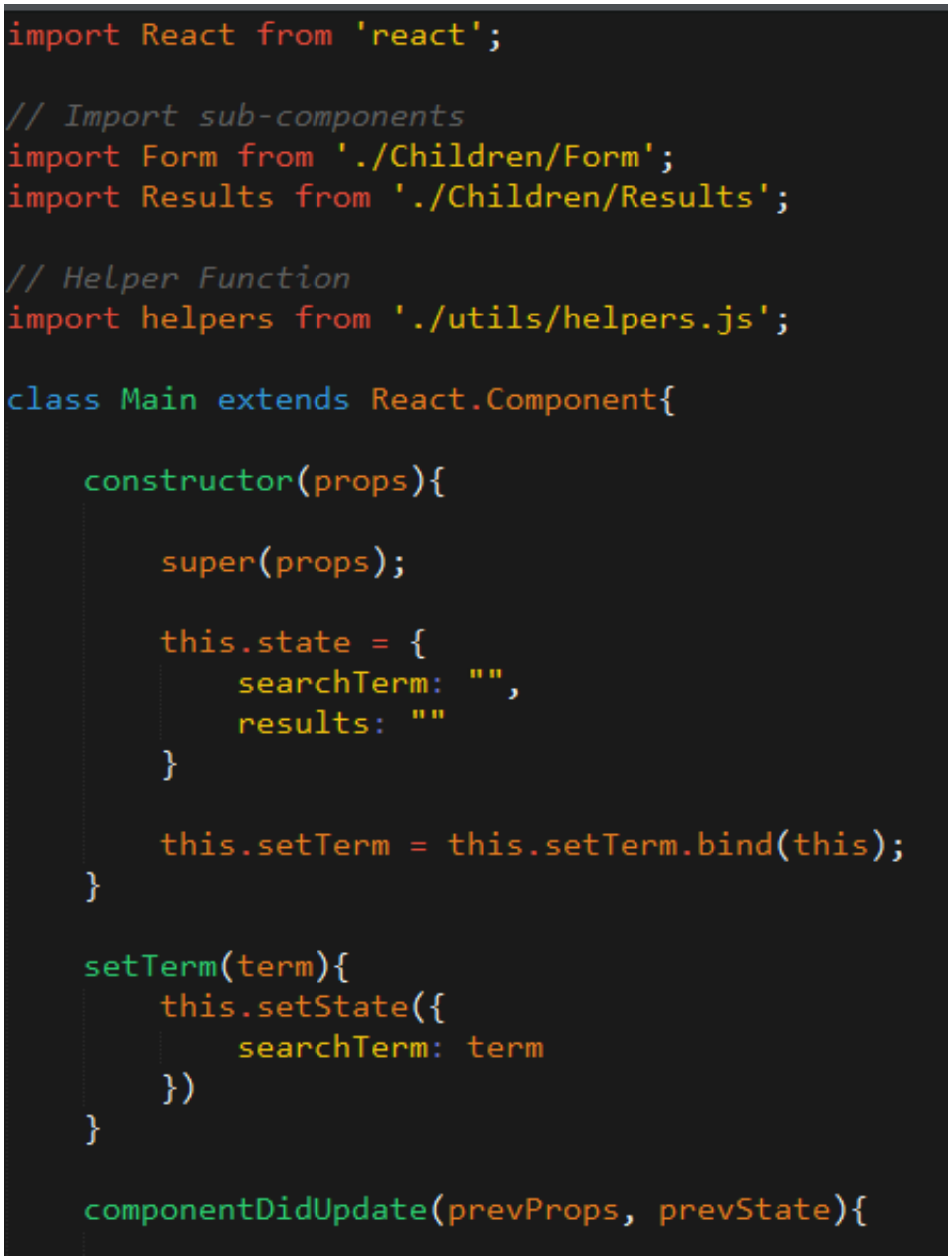
---

12. End

---

13. Bonus: Students Do: ES6 Address (0:40)

- If any time remains. Task students with converting the previous React Address application to ES6 standard. This primarily means changing the components to use classes. They can reference this guide to better understand the changes. https://facebook.github.io/react/docs/reusable-components.html#es6-classes
-



```
import React from 'react';

// Import sub-components
import Form from './Children/Form';
import Results from './Children/Results';

// Helper Function
import helpers from './utils/helpers.js';

class Main extends React.Component{

    constructor(props){

        super(props);

        this.state = {
            searchTerm: "",
            results: ""
        }

        this.setTerm = this.setTerm.bind(this);
    }

    setTerm(term){
        this.setState({
            searchTerm: term
        })
    }

    componentDidUpdate(prevProps, prevState){
```

Images/6-ES6React_1.png

Copyright