Hack DKU Course Syllabus:

Event Doc: Hackathon Proposal.pdf

Plans for expansion:

- Web Sockets / ws
- **Deep learning** / theory and pytorch

Hack DKU Course Notes:

1. Introduction to Web: A brief introduction to the design of the internet and basics on how to design a website. Content Covered: HTML, CSS, JS, git, GitHub.

2. Introduction to React: How to start a React project: organization, dependencies, basic components, and dynamic web design. Content Covered: npm, React.js.

3. Introduction to Node.JS: How to serve a website with Node.js and Express from basics to multiple page routing. Content Covered: React.js, Node.JS, Express.

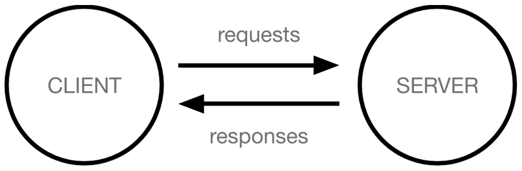
4. Introduction to Docker and AWS: Finishing the course by learning how to deploy your app to production on Amazon's Elastic Cloud Compute. Content Covered: Docker, AWS, EC2, EB

5. References.

1. Introduction to Web Development

A brief introduction to the design of the internet and basics on how to design a website. Content Covered: HTML, CSS, JS, git, GitHub.

A. How does the Web work?
   a. The internet exists as a **network** of data hubs that **centralize** the information displayed on websites/web apps. When you access a website, your request for the website's data is **routed** by your ISP to a **server** owned by the website or a hosting service.



   b. The diagram above does a good job explaining the basic structure of a web interaction. A **client**, or person using the internet, visits a website and sends a **request** to the **server**. The request may ask for images, html pages, or other data. These requests are processed and then responded/served.
   c. **Search engines** like Google or Baidu serve as **dictionaries** of websites. Searching in Google is like opening a phonebook and looking for phone numbers of places you're interested in. Except in this case, queries result in web addresses instead of a phone number.

B. What data do servers respond with?
   a. **HTML** pages: Code that describes the layout and content of a website. It holds information like.
      i. The website title
      ii. The organization of the page
      iii. Static text data
   b. **CSS** (Cascading Style Sheets): Information that describes the style of the web page (example: color, font, animations, margins).
   c. **JavaScript** files: Code that makes your site interactive.
   d. Other data: Images, audio, messaging data, dynamic text, game data.

C. How do we join the Web?
   a. Let's start coding. First we need a way to manage our code. **Git** is an open source project management tool that is very widely used. Git allows you to track the changes that you make to your files in an organized way and builds the framework for **sharing and developing** code.
   b. Using git:
      i. First we have to **install** git.

ii. Then we will make our first project. Start by going to the command line (cmd) and typing git --version to make sure your installation worked. Then start navigating to a new folder. Helpful commands:
    1. dir or ls: list all the folders and files in your current location
    2. cd <dir>: change to the given directory
    3. mkdir <name>: make a new directory with given name

iii. Once you're in a new directory use git init to create a new git repository. From here we need to add some files for git to **keep track of**. So, in this directory open your file editor and make a new file named "README.md". By convention this file should describe what your project is and how other people can use it. Right now let's just write "My first web app."

iv. Great! Now back in the command prompt type git add *. This command tells git to keep track of all files (* means all). Then type git commit -m "initial commit". A commit is a **checkpoint** and -m is a **message** that describes what you changed.

v. Now we have a git repository, how do we share the code with other people? There are a few options for this, the main one being GitHub.

c. Sharing your code with GitHub

i. First go to `github.com` and sign up for an account. Once you're finished visit your profile and click 'Create a repository.' This will create an online repository that will mirror the one you have on your computer. Then copy the link to the new repo, it should look something like this: `https://github.com/yourname/project.git`. Then go back to the terminal and type git remote add origin <copied web address>. In git, a remote is another name for a **mirror** of your repository. We want to add a new remote named 'origin' that links to our GitHub repo.

ii. Finally, type git push origin master. This will **upload** all of our changes on the 'master' (main) branch to our 'origin' remote repository. Go back to your GitHub repo and you should see your changes there.

iii. One last important git command is git pull. This is similar to git push but instead of uploading your changes, it **downloads** any new changes to the repository while you were gone.

d. Making an html page

i. HTML is a **markup language** used for describing web documents. HTML documents have the '.html' suffix. HTML uses **tags** to define different parts of the page.

ii. All HTML documents start with the tag <!DOCTYPE html> which specifies that the document is of type html. The other main parts of an html document are the <html></html>, <head></head>, and <body></body> tags. The html tag contains all other tags in the document.

iii. The <head> tag holds all of the meta data for the document. This includes some common tags like:

   1. <meta charset = "utf-8" />: defines the text encoding
   2. <title>**title**</title>: sets the title of the page
   3. <link href="**style**.css" rel="stylesheet" type="text/css"/>: points the html document to the style sheet you want to use.
   4. <script src="**main**.js" type="application/javascript"></script>: points the html document to the main JS file to use.

iv. The <body> tags contains all the rest of the html for your page. Some of example tags that you might use:

   1. <div id="main">**tags**</div>: creates a section of the document.
   2. <h1>**text**</h1>: creates a header.
   3. <p class="main-text">**text**</p>: creates a paragraph of text.
   4. <button onclick="handleClick();">**text**</button>: creates a button that when clicked starts the click() function in your JS doc.
   5. <a href="**site**.com">**text**</a>: a link to a website.
   6. <img src="site.com/image.jpeg"/>: embeds an image.

v. One last important part of html are attributes. Attributes are additional information about a tag passed in the opening tag. Good news, you have already used attributes. href, id, and class are all examples of attributes. So what do they do?

   1. href: a link to another page or website
   2. id: gives a name to the element for you to reference later
   3. class: the CSS class to use for this element
   4. src: the source of an image or other media
   5. style: allows you to set a style for this element
   6. And there are many more...

vi. Remember to add, commit, and push your changes.

e. Styling with CSS

i. Cascading Style Sheets is a **style sheet language** used for describing the presentation of a document written in a markup language like HTML. CSS files have the file extension ".css". CSS **defines styles for html tags** or specified classes

ii. The basic syntax of a CSS style is: `tag {style; style; ...}`. First specify the class or tag to apply the style to, then in brackets set the styles.

iii. Classes: a class in CSS is a way to define multiple styles for one HTML tag. The syntax for a class is:
   1. In the CSS file: `.classname {style; style; ...}`
   2. In the HTML doc: `<tag class= "classname">...</tag>`

iv. Common styles:
   1. `text-align: center;` centers text in a tag
   2. `font-size: 24px;` changes the font size
   3. `margin: 48px 24px;` sets the vertical and horizontal margin
   4. `width: 100px;` sets the width of the tag
   5. `height: 80px;` sets the height of the tag
   6. `color: black;` sets the color of the text in the tag
   7. `background-color: #2760b0;` the background color of the tag
   8. `border: 4px solid black;` creates a border around a tag with width of four, solid stroke, and black color
   9. `padding: 12px 12px;` sets the vertical and horizontal padding

v. There are **way** more styles that you can learn to make your designs cooler, cleaner, and easier. Here is a great resource.

f. Adding a little JS

i. I know this has been a lot but all we have left is to cover a few small things in JavaScript.

ii. JavaScript is a scripting language, obviously, but understanding what that means helps to understand how JS is used in web development. As a **scripting language**, JS doesn't have the tools to easily manage data or to create complicated projects with dozens of files (kinda). It does, however, have super simple ways to interact with **APIs and DOM**. It's simple syntax makes it perfect for code that relies heavily on external packages.

iii. JavaScript relies heavily on event driven programming, a style of programming where functions are called based on events (e.g. a mouse click or a sports score update). You already started the code for the first event we will talk about, a button press. In our html doc we used `<button onclick="handleClick();">`**text**`</button>`. So lets make a JavaScript file and write this `handleClick()` function.

        iv. All JavaScript files have the **‘.js’ file extension**. Like Python there is no real need for a main method; We can just start writing our function.

```
handleClick = () => {
   let main = document.getElementById(“main”);
   let newText = document.createElement("p");
   newText .appendChild(document.createTextNode(“New Text"));
   newText.classList.add(“new-text”);
   main.appendChild(newText); // Example comment
}
```

        v. Let's quickly break down the code above.
        vi. In the first line we define the function (I know it looks a little weird it's just one of the many ways to do it in JS). Then we declare main and set it equal to document.getElementById(“main”); ‘document’ is a **reserved** word that references your html doc. Then we use getElementByID to reference our main div we created earlier.
        vii. Then we declare and set newText equal to a new ‘p’. The next line is a bit confusing but don’t sweat it. All the 4th line does is change the text in newText to “New Text”.
        viii. In the 5th line we add the ‘new-text’ style to newText.
        ix. Finally we add newText to main and we are finished.
    g. Let’s check it out! Just open up your HTML document and you should see what we made.

D. What's next?
    a. This was a lot, but there is still a ton of a lot more to learn. As you can see it is really easy to make a simple webpage with these techniques but what if you want to do more? Next week we will teach you how to use React, a popular framework that will make designing websites super simple.
    b. Thanks for sticking with us.

2. Introduction to React.JS

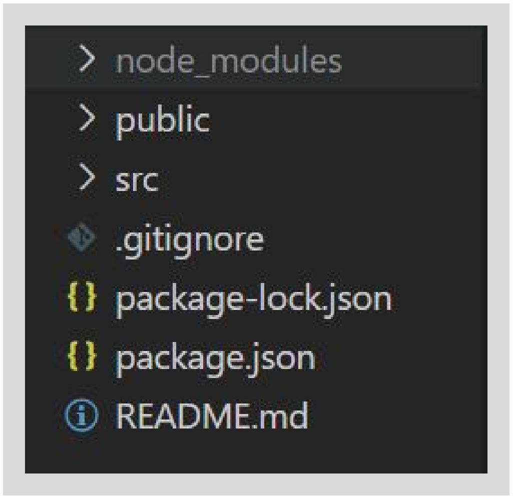
How to start a React project: organization, dependencies, basic components, and dynamic web design. Content Covered: npm, React.js.

A. Last week we learned about the **fundamentals** of web development, but how can we push that further to develop more sophisticated web apps?
   a. Today we will focus on creating a website's **front end**. The front end is what the user sees: the HTML, CSS, and JS scripts that run the page.
   b. As we saw last week it's pretty hard to quickly make **dynamic web pages**; Dealing with the JS document object can get difficult. So how can we get around it.
   c. There are many different UI (user interface) **frameworks** that are resources that make designing front end web pages easier. The most popular right now is **React.js**.

B. How do we start with React?
   a. First we have to start with **Node.js**, a Javascript **runtime environment** that allows you to run JS files **outside** of the web browser. Node is somewhat similar to Java's JVM. Let's go download Node: https://nodejs.org/en/download/.
   b. One of the most powerful aspects of Node is npm, the **Node Package Manager**. Npm automatically installs when you get Node so you already have it. If you have any experience with Python, I am sure you know pip, npm is very similar. In short, it allows you to easily access thousands of powerful web development packages.
   c. Ok so now we have what we need to get started. Let's open console and get to coding. First we just need to move around using dir/cd and create a new folder for your projects with mkdir.
   d. To start the project we will be using the commands:
      *You can change my-app to whatever you want to name your project

```
npx create-react-app my-app
cd my-app
npm start
```

   e. These three commands will initialize a basic React.js app (it might take a minute). Ok cool, now we have the basics for a great web app. Press ctlr + c in console to exit the demo. Now let's open up a text editor; if you use VS Code all you have to do is type code .. You should see something like this:



C. Understanding a React App.
   a. What do these things mean?
      i. node_modules: This folder contains all the **packages** that you need to run your app.
      ii. public: Contains all the non js files that you will **serve** to your clients
         1. E.g. HTML, Images, Icons…
      iii. src: Contains all the CSS and JS code for your project, all we really care about are App.js and App.css.
      iv. .gitignore: create-react-app initializes a **git repository** for your project, this file specifies what types of files your git repo shouldn't track. In this case it tells git to avoid all the files in node_modules.
      v. package.json: This file is a **summary** of your app for other people. It includes what packages you need, how to run and test your app, and other information about you.
      vi. README.md: An **introduction** to create-react-app but we can change it to describe our project.
      vii. **Please** take time to **explore** these files, you can really learn a lot just seeing what they have to offer.
   b. To keep things short we will start coding the app without getting too deep into the specifics just yet. If you have any questions feel free to reach out to me later.

D. Coding a React App:
   a. Navigate to src and open App.js.
   b. This is the home of your React code, let's start by making a quick list; this is the '**hello world**' of React apps. Start by deleting everything, we will start from scratch.

c. React is built using **components**, which are parts of your code that **render** as blocks in the browser. First we have to import React, {Component} from '*react*' and import our CSS file with import './App.css'.
d. There are two ways to create a component in React, as a **class** or as a **function**. For our list component we will declare it as a function to keep things simple. What React does is it allows you to **dynamically** render and generate HTML using a React/HTML format called **JSX** that is returned by each component. So, in our function we will return what looks a lot like HTML.

```
function List() {
    return (
      <div className="list">
        <h1>Homework List</h1>
        <ul>
          <li>Homework</li>
          <li>Volunteering</li>
          <li>Student Work</li>
        </ul>
      </div>
    );
}
```

e. Lastly, we have to **export** our List. Whatever component is exported from our App.js file will be the main component rendered in the browser. All we have to write is:

```
export default List;
```

f. Let's check out how this looks in the browser. First go back to our command prompt and type npm start. This is one of the helper scripts defined in our package.json file that will run our project on **port 3000**. If the page doesn't open automatically, visit localhost:3000 in your browser.
g. Well that seems like something we could do with basic HTML. How do we use React to make this more dynamic and interesting?
    i. First let's declare List as a **class** instead of a function. This will give us a lot more **power** when designing our components. All we have to do is change the way it's declared and instead of just returning our JSX we have to declare a function in our class called render that returns it there.
    ii. We can use the fact that our component is a class to give us the ability to store pass **information** between components. Let's try

this by changing the name in the list to whatever we want to pass to it.

iii. To add JS code inside JSX all we have to do is write it in **brackets**. This is what it looks like:

```
<h1>{this.props.name} List</h1>
```

h. Parent Child Relationships

i. In React, data flows **downwards**. This means that larger components, like the full webpage **control** the smaller components, like our list. This might be a little confusing so lets try implementing it on our code.

ii. First we need to make a **parent**. Let's name it ListHolder and declare it using the function method.

```
function ListHolder(){
  return (
    <div className="list-holder">
      <List name={"Homework"}/>
    </div>
  );
}
```

iii. Next we have to modify our old code in two ways. One lets add a constructor in our List class. All we need to write is:

```
  constructor(props){
    super();
  }
```

iv. Lastly we just have to change our export from List to ListHolder.

v. Let's take a **quick break** from the React to make everything look a little nicer using CSS. First open up our CSS file, it should be in src/App.css.

1. How you make it look is really up to you!
2. Here is a link to the styling that I used

i. Now back to business, it's cool that we can have a list that changes its name but how do we change the **actual values** of the list?

i. Another awesome feature of React is the ability to make **dynamically sized content** using build in JS lists.

ii. First we need to make another component called ListItem that represents each bullet point in our list.

```
class ListItem extends Component{
  constructor(props){
    super();
    this.state = { color: "#000000", selected: false };
    this.handleClick = this.handleClick.bind(this);
  }

  handleClick() {
    this.setState((state) => ({selected: !state.selected}));
    this.setState((state) => ({
      color: state.selected ? "#D6B85C" : "#000000",
    }));
  }

  render() {
    const style = {
      color: this.state.color,
    };

    return (
      <li className="no-select" onClick={this.handleClick}
.              style={style}>{this.props.value}</li>
    );
  }
}
```

iii. Code Breakdown:

1. First we call the **constructor** to create the component. For the first time we will be using React's **'states'** which serve as **instance variables** for React components. this.state is a JS Object (like a dictionary in Python).
2. We are storing the **color** that we want the bullet point to become as well as a **boolean** to hold a true or false value if the bullet point has been clicked.
3. Then we register our function to React.js with
   this.handleClick = this.handleClick.bind(this);
   I know this syntax is a little weird but remember you have to do it for all class functions.
4. Then we implement the handleClick() method that swaps our isSelected value and then decides what color the point should be given if it is selected.
5. Next, in render() we take advantage of React's ability to **dynamically style components** to set the color of our bullet point to whatever is in state.color.

6. Finally we return our **JSX** remembering to set the class, style, and onclick values. Also we will set the value of the bullet point to what is sent in this.props.value.

Remember, if at any point you feel confused feel free to reach out or read the source code: https://github.com/silva-nick/hackdku-course-1

---

iv. Next we have to move back to our List class to implement these new changes. There are only **3 differences** we have to deal with.

1. In our constructor we need to initialize our list values. We will use the state instance again in this case.

```
this.state = {listValue: ["Homework", "Volunteering",
                                   "Student Work"]};
```

2. Next we have to render a list of our ListItem components using the data in listValue. We do this by iterating through the **indices** of listValue and then pushing a new ListItem to our list of items (listItems). Note, each value we add has to have an associated and unique **key**. In our case, it is easiest to set this key value to the index.

```
In our render function:
const listItems = [];
    for(const item in this.state.listValue){
      listItems.push(<ListItem value={this.state.listValue[item]}
                                key={item} />);
    }
```

3. Finally instead of rendering three <li> structures we just pass our list into our JSX.

```
  <ul>
       {listItems}
  </ul>
```

v. And that's it, all we have to do to add, remove or change values in our list is to change the strings stored in listValue. In the real world we wouldn't use a hardcoded list; Instead we would make **API** calls or **database queries** to get our data.

E. The last thing we are going to talk about today is the child to parent **dataflow**. Before we discovered that by using props we can easily send data from a **parent** to a **child** component. However, what if we want to send information back from the child to the parent. The answer is called a **callback**. Which is a **function** in our parent that we will send to our child in props. Then all the child has to do is **call the function** and send its information to the parent. Let's try this out in our own code.

   a. First we have to change four things in our List component.

      i. In our constructor, we need to add a new state that keeps track of how many tasks we have **completed**. Also, we want to **bind** our new function. We only need to change two lines of code to do this.

```
this.state = {listValue: ["Homework", "Volunteering",
.                                "Student Work"], numCompleted: 0};
this.countCompleted = this.countCompleted.bind(this);
```

      ii. We need to then write the **callback** function that implements some **feature** that only the parent can handle. How about we display how many items on our list we have completed.

```
  countCompleted(change){
    this.setState((state) => ({numCompleted:
.                         state.numCompleted + change}));
  }
```

      iii. Then we need to pass this callback function to all of our ListItems when we construct them.

```
listItems.push(<ListItem value={this.state.listValue[item]}
.              callback={this.countCompleted} key={item} />);
```

      iv. Finally we have to add some more pieces to our JSX to display our new count.

```
<p>Completed
.  {this.state.numCompleted}/{this.state.listValue.length}</p>
```

   b. Good news! All we have to do is change one line of code in ListItem.

      i. In our handleClick() method we want to **trigger** the callback and send to the parent -1 if we are deselecting the item or +1 if we are selecting the item.

```
this.props.callback(this.state.selected ? -1 : 1);
```

   c. And wow we are done!

3. Finishing up React and Intro to Express.JS

How to serve a website with Node.js and Express.
Content Covered: Node.JS, Express.JS.

A. Finishing up from last week.
   a. Last week, we learned how to make a simple React app using the create-react-app framework. One of the core aspects of what we learned last week was how to **dynamically** render a website to different data. However, we never pushed this idea to its **limits**. So let's give it a try this week.
   b. One easy thing we can do to make our app more **scalable** is by **centralizing** the data we want to render in our ListHolder component. In this case, we need to move listValue from List to ListHolder.

```
0  <List name={"Homework"} listValue={["Homework",
   .                        "Volunteering", "Student Work"]}/>
```

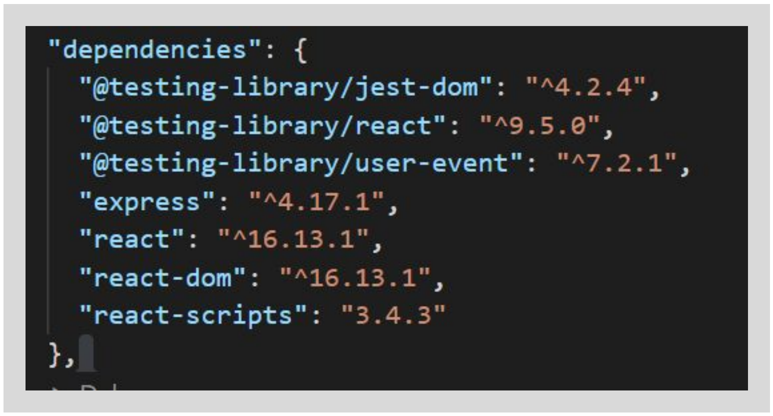
   c. Then all we have to do is change all the times this.state.listValue is referenced in List to this.props.listValue.
   d. Now, **experiment** by copying and pasting the List element in ListHolder. Our app now **supports** multiple lists of different sizes!
   e. But copying and pasting isn't exactly a great practice so instead we will put all of those in a list and let React do the rest. **Remember** to add the key property.

```
0  let listList = [<List key={0} name={"Homework"}
   .                listValue={["Homework", "Volunteering",
   .                "Student Work"]}/>]
```

B. Finalizing our code: Once you are finished with this section, we need to **build** our app. Because browsers can only render the type of html that we designed in our first lecture, before we can run our app **natively** on browsers we need to build it into a way that browsers can understand. Building web apps is analogous to the process of compiling Java or C.
   a. So how do we build our project. It's super simple, all we have to do is run npm run build in console. Let this script run until it's finished and you have finished building your app.

C. Now let's move on to designing a server for our app.
   a. To start we have to go back to the **console**. The framework we will be using is **Express.JS**. You can code nearly any server you want just using default Node.JS libraries, but frameworks like Express make it really easy to write simple servers like what we will be designing today.

b. Type npm add express to add express to our modules. Let’s check out our package.json file to see if anything has changed. In your dependencies entry you should see the latest version of Express, cool so now we can get started.



```
"dependencies": {
  "@testing-library/jest-dom": "^4.2.4",
  "@testing-library/react": "^9.5.0",
  "@testing-library/user-event": "^7.2.1",
  "express": "^4.17.1",
  "react": "^16.13.1",
  "react-dom": "^16.13.1",
  "react-scripts": "3.4.3"
},
```

c. To start we need to create a new folder in our **root** directory and name it server. Then in that folder let's make a new file named server.js. This is where we will put all of our code for the server.

d. The first thing we need to do is **import** the packages we will need. We have two imports, express and path.

```
const express = require("express")
const path = require("path")
```

e. Next we have to initialize both our port and our express app.

```
const PORT = process.env.PORT || 3001;
const app = express();
```

f. Once our express app is initialized we next have to tell express to statically serve our files. There are two main functions a server like this performs.

    i. Static serving: sending the main page documents to the client. When you visit a website you are always served with the same html, image, and css files.

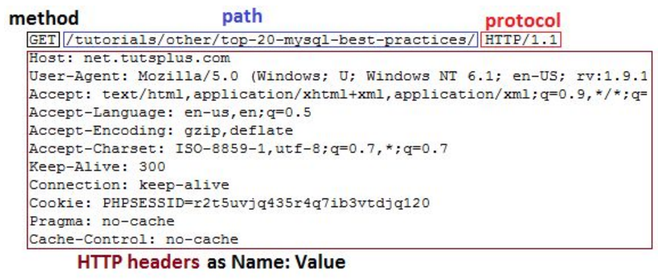
    ii. Dynamic serving: sending files or data only when the client requests it. For example, when your professor sends you an email the server dynamically sends you the file after noticing that your mailbox has new mail.

g. So right now all our application does is statically serve our built React.js files. We will set this up with express in just **one** line.

```
app.use(express.static(path.resolve(__dirname, "../build")));
```

h. In the first line we tell Express that we want to statically serve our files out of the build folder. Then we respond to our app’s **get** requests.

i. A brief overview of http:
   i. HTTP is a **protocol** which allows the fetching of **resources**, such as HTML documents, images, data, etc. What this means is that whenever your phone or computer visits a website it sends the website a message that follows a strict format. It then parses that input and responds with another HTTP request with the data you are interested in.
   ii. What does a HTTP request look like?
   Below is an example of a GET request



```
GET /tutorials/other/top-20-mysql-best-practices/ HTTP/1.1
Host: net.tutsplus.com
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 6.1; en-US; rv:1.9.1
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
Cookie: PHPSESSID=r2t5uvjq435r4q7ib3vtdjq120
Pragma: no-cache
Cache-Control: no-cache
```

   iii. What kinds of HTTP requests are there?
      1. GET: request information from a server
      2. POST: send information to the server
      3. There are a few more but these two are the main ones you will be using.
   iv. Want more information? Check out this.

j. Now all that's left is to tell our server to start.

```
app.listen(PORT, function () {
    console.log(`Listening on port ${PORT}`);
});
```

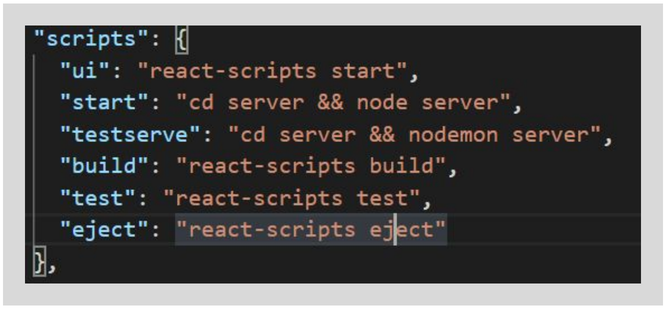
k. What this does is tells our app to start **listening** for HTTP requests on the port we defined above. The function is called when the server starts and just outputs the port our app is listening on.

D. One last thing.
   a. While we are developing our apps it becomes a lot of work to type npm start each time we want to test our changes. Luckily there is a great way around that. Type npm add -D nodemon in console. This is the command

to install **nodemon**, a command that lets us automatically update our page, as a development dependency.

   i. A **development dependency** is a package used while you are coding but won't change the final product.

b. Now that we have this cool tool lets go back to our package.json file to implement some React scripts to make it easier to run our project. First let's rename start to ui. Later, if we want to **deploy** our project, we are going to want our start script to run the server. Next, add the start and testserve scripts as below. These both use our new server to run our app.



```
"scripts": {
  "ui": "react-scripts start",
  "start": "cd server && node server",
  "testserve": "cd server && nodemon server",
  "build": "react-scripts build",
  "test": "react-scripts test",
  "eject": "react-scripts eject"
},
```

c. Let's test it out. Type npm run testserve in console to start the server. Then all we have to do is navigate to http://localhost:3001/ and we can check out our site.

d. Awesome, now we have our server running locally on our computer. Next we will learn how to take this code and host it online using Docker and Amazon Web Services.

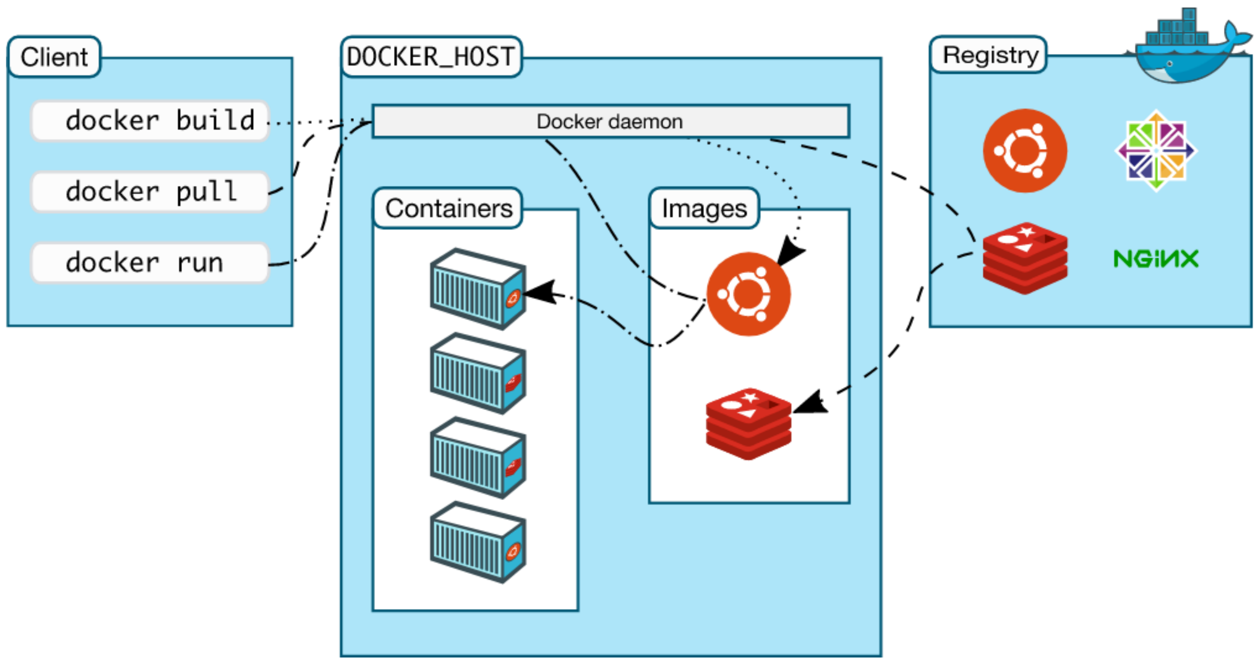
4. Deploying, Docker, and ECS

How to containerize your app and get it hosted with Amazon Web Services (AWS) Elastic Container Service (ECS)
Content Covered: Docker, Elastic Beanstalk, ECS

A. What is Docker?

a. Because **cloud hosting** has become so prevalent in the last few years, software container platforms like **Docker** have provided solutions to many **problems** developing for these new systems. The **root problem** stems from the necessity to **locally develop** software that will work on random **servers in the cloud**. Obviously this is a problem because to run your code on these servers you need to make sure they have the correct libraries, networks, and can be **configured for your application**. Traditionally, software devs would have to rigorously set up their servers or painstakingly configure virtual machines so that their code could be scaled into production.

b. The image below is a nice visual representation of how Docker is used in development. Feel free to refer back to it as we work through the next sections.



c. How does Docker solve this problem?

i. Docker allows you to define a lightweight **virtual machine**, called a **container**, that runs Docker **images**. Images are read-only versions of your code, libraries, and runtime environment.

ii. **For example**, if I want to run a python program that uses Numpy arrays and prints to the console I would create an image that has python, numpy, and my code. Then we would compile that into an image and anyone with Docker could run it on their computer in just a few commands.

iii. The key way Docker creates these images is through a configuration file called a **Dockerfile**. The Dockerfile defines the **base image** you want to build off of (python in the first example), and then has a series of **commands** that sets up your runtime environment.

B. Running a Dockerized version of our app

a. This is confusing so lets try it out ourselves to get a little more practice. First we want to install a useful utility called Docker Desktop. Once you have that installed open cmd and type docker -v to check if you installed everything correctly. Before we can define our Dockerfile we want to create a .dockerignore file that tells Docker what **not** to include in our image. React projects are really big so without this file our images will be really **oversized**. In our .dockerignore we want to add.

```
node_modules
npm-debug.log
```

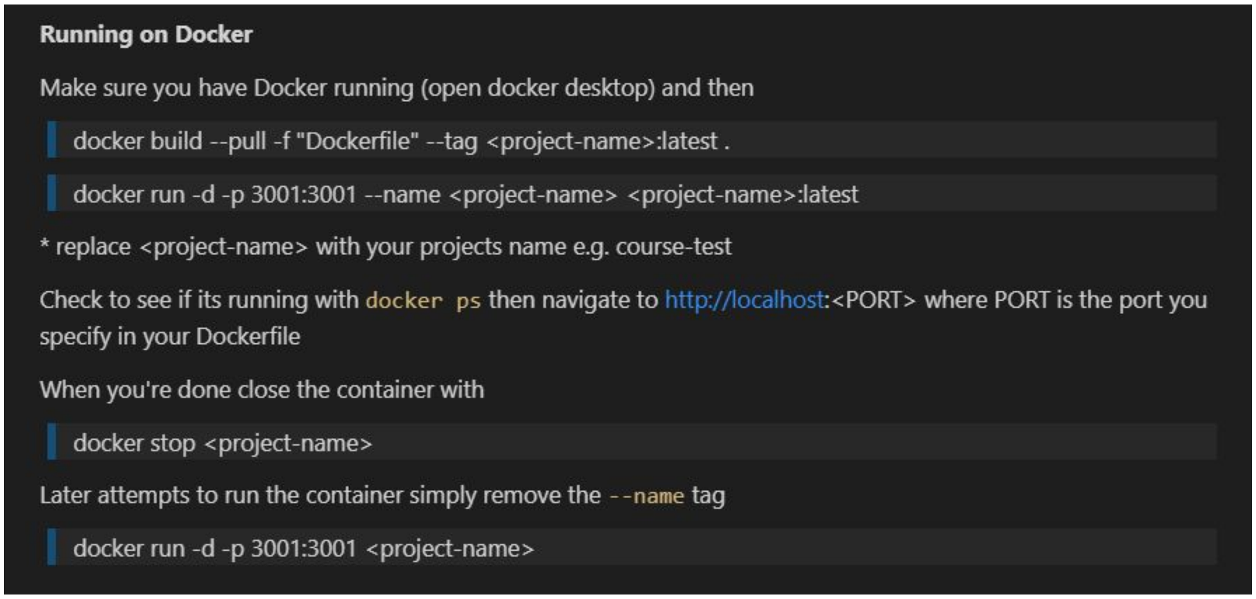
i. Next define a file named Dockerfile with no file extension, this is where we will start our configuration.

ii. A Dockerfile tells Docker to run a series of commands to set up your virtual environment. These are the commands we are using to set up our environment and an explanation.

FROM node:alpine	FROM defines a **starting image** to use. In our case we want to start from the Node.js image running on alpine Linux.
COPY . /app	**Moves** our files into the ./app directory of the VM.
WORKDIR /app	WORKDIR is the same as **cd** in console.
RUN npm install --production && npm run-script build	RUN **runs a command** in a new layer while building the image. It is Docker best practices to reduce the

	number of layers by chaining commands in RUN.
EXPOSE 3001	Tell Docker what **port** you want your app to run on.
CMD ["npm", "start"]	This is used as an **entrypoint** into your application, i.e. Docker runs this when it starts your container.
USER node	Setting the user to 'node' helps protect **privileges**.

b. Next, let's try running a **containerized** version of our app. For this section we'll follow along with the 'Running on Docker' section of the README.md.



Running on Docker

Make sure you have Docker running (open docker desktop) and then

```
docker build --pull -f "Dockerfile" --tag <project-name>:latest .
```

```
docker run -d -p 3001:3001 --name <project-name> <project-name>:latest
```

* replace <project-name> with your projects name e.g. course-test

Check to see if its running with `docker ps` then navigate to http://localhost:<PORT> where PORT is the port you specify in your Dockerfile

When you're done close the container with

```
docker stop <project-name>
```

Later attempts to run the container simply remove the `--name` tag

```
docker run -d -p 3001:3001 <project-name>
```

c. Dope, we have our program runner on our machine through Docker, so what's next?

C. Deploying our Docker image to AWS
   a. Everything we have done so far was pretty fun, but it was pointless if we can't share our app with the world. There are dozens of ways to get our application hosted but we chose Amazon Web Services (AWS) because it is [incredibly popular]() and is a popular skill for undergrad internships.
   b. To get started we first want to share our image on an image repository.
      i. Again there are a bunch of image registry options but DockerHub has easy integration with the Docker CLI. First, go to dockerhub.com and create an account; Remember your username

and password because now go to your terminal and type: docker login follow the prompts and login to Dockerhub.

    ii. Next, let's create an image that we will share on Dockerhub. All we have to do is rebuild, specifying your username this time: docker build --pull -f "Dockerfile" --tag <yourusername>/<project-name> .
    iii. Now we can push the image to our Dockerhub account with: docker push <yourusername>/<projectname>.
    iv. Navigate to your repository on Dockerhub and you should see your image hosted, dope. Uploading your image to a registry isn't necessary but it makes it easier to manage and connect your project with AWS.

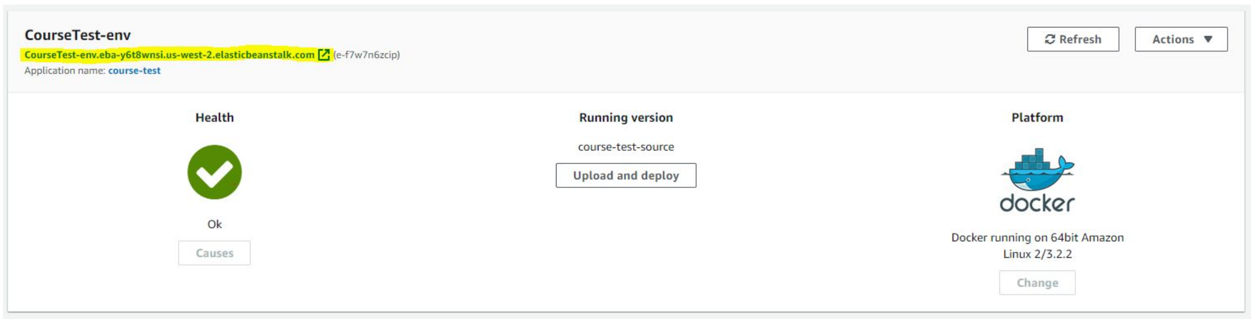
c. Now all we have to do is connect our repository to AWS. But first, what is AWS?
    i. AWS is Amazon's collection of **cloud-based web services**. AWS isn't just one thing, it's a **collection** of web hosting, database management, machine learning solutions, and more all under Amazon's management infrastructure. The service we will be using is called EC2 or **Elastic Compute Cloud**.
    ii. EC2 is a container based computation service that is **elastic**, meaning it allocates and deallocates computation power based on webflow to your service. This means we can set up our Docker container on AWS and it's able to **automatically scale** to our individual needs. Sadly scaling isn't that easy which is where **Elastic Beanstalk** (EB) comes in.
    iii. EB is a software layer to our application that deals with **interfacing** our source code with the details of the EC2 infrastructure. EB manages **load balancing**, **autoscaling**, and **health monitoring**. Anything that EB does we could do manually through EC2, but EB allows us to not worry about the details of AWS and instead focus on **deploying** our application.

d. So, sadly, this involves setting up another **account**. Signup/login through this or sign up through Amazon's education program to get some extra perks.
    i. Once you have an AWS account you should be able to access Elastic Beanstalk. If you can't find it try this link. Then we can start with **'Create Application'** and we can start our configuration.
    ii. Next walk through the basic configuration: Name your application, choose **'Docker'** as the platform, choose 'Upload your code', and double check to make sure the auto generated name is okay.

iii. Now we will go back to our source code to define a Dockerrun.aws.json file. This file **communicates** to where our image is from and how it should be run. Make sure to change the name to whatever you named your image on Dockerhub.

```
{
  "AWSEBDockerrunVersion": "1",
  "Image": {
    "Name": "silvanick/course-test",
    "Update": "true"
  },
  "Ports": [
    {
      "ContainerPort": 3001,
      "HostPort": 8000
    }
  ]
}
```

iv. Once you have finished this file, go back to the AWS console and select **'Upload File'** and upload this configuration file. The setup should take a few minutes so don't worry. When it looks like the command output is finished navigate to the environment and click the **link** to see **your code** running in the cloud.



CourseTest-env
CourseTest-env.eba-y6t8wnsi.us-west-2.elasticbeanstalk.com (e-f7w7n6zcip)
Application name: course-test

Health: Ok
Running version: course-test-source
Platform: Docker running on 64bit Amazon Linux 2/3.2.2

v. Finally, because we don't want to accidentally be **charged** for this practice example click the actions tab on the right side and **terminate** this environment. You can always start it up the same way we did before, this just saves us the extra cost.

D. Wow, we are finally done. From here the only thing stopping you from creating and deploying cool web apps is your creativity. Of course there are always more things you can learn: here's how you route traffic to your EC2 instance and here is how you use multiple EC2 instances to make your program more scalable.

5. References and Acknowledgements

1. Introduction to Web Development:
    a. https://developer.mozilla.org/en-US/docs/Learn
    b. https://gist.github.com/mindplace/b4b094157d7a3be6afd2c96370d39fad
    c. https://rogerdudler.github.io/git-guide/
    d. https://www.w3schools.com/css/
2. Introduction to React.js:
    a. https://nodesource.com/blog/an-absolute-beginners-guide-to-using-npm/
    b. https://reactjs.org/tutorial/tutorial.html
3. Introduction to Node.js:
    a. https://nodejs.dev/learn
    b. https://nodejs.org/en/docs/guides/anatomy-of-an-http-transaction/
4. Intermediate Topics in React and Node:
    a. https://docs.docker.com/get-started/overview/
    b. https://nodejs.org/en/docs/guides/nodejs-docker-webapp/
    c. https://docker-curriculum.com/#docker-on-aws