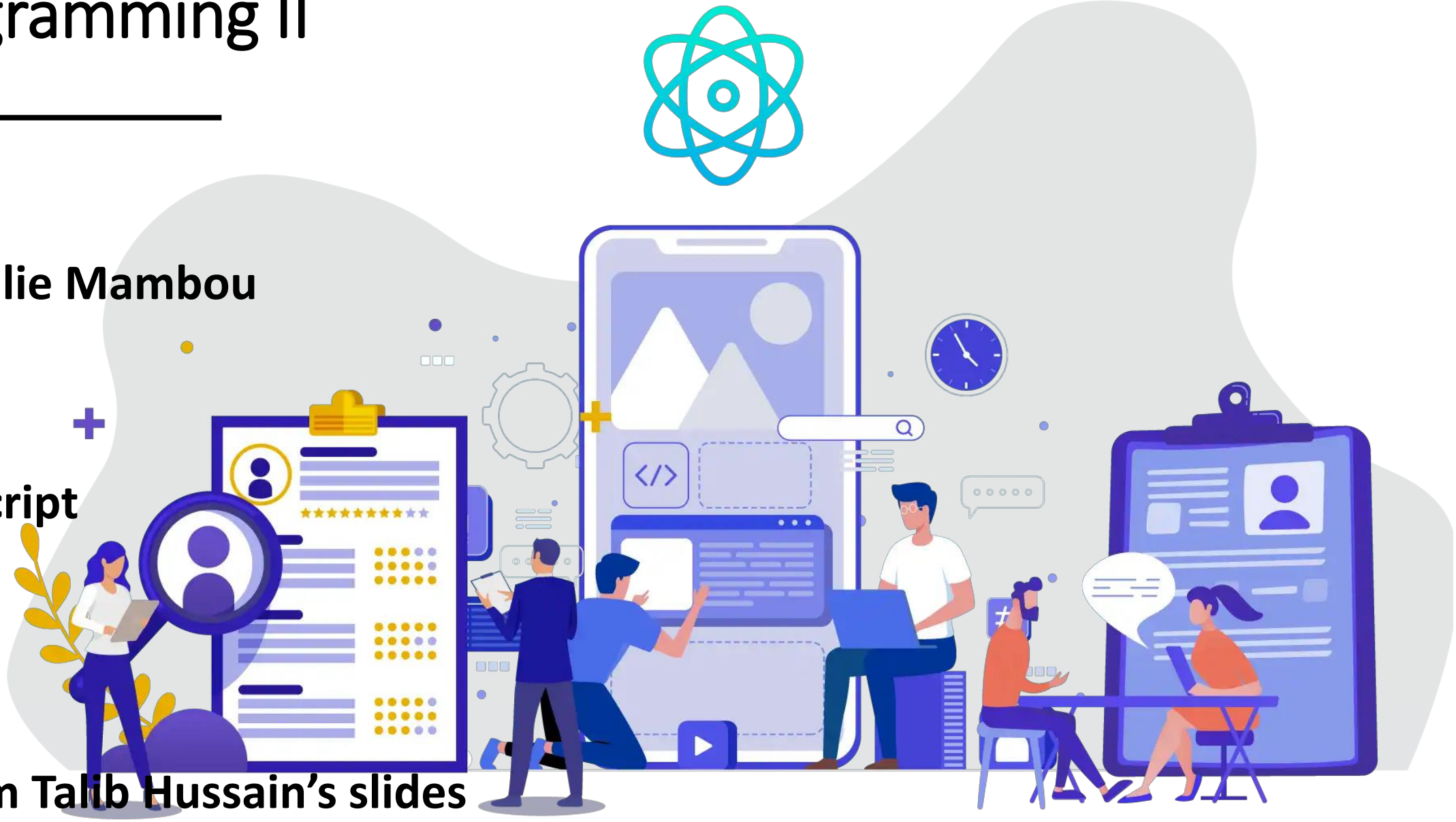
Web Programming II

420-JE5-AB

Instructor: Elie Mambou

Day 1:

Intro/JavaScript



Adapted from Talib Hussain's slides

Agenda

- Welcome
- Course Outline
- JavaScript Review
  - HTML+CSS
  - Manipulating Domain Object Model (DOM)
  - Events
  - Timing (setInterval, setTimeout)
  - User Interactions
  - State (Client-Side)
  - Asynchronous (Promises, Async/Await)

Competency: Develop Transactional Web applications

General Learning Objectives
Analyze the application development project
Prepare the computer development environment
Program the Web interface
Deploy the application
Produce the documentation
Develop a transactional web application

Evaluation Plan (Likely dates)	% of Grade
Exercises (~2)	10%
Test (~Class 8)	30%
Project Milestone 1 (~Class 10)	15%
Project Milestone 2 (~Class 13)	15%
Project (~Class 15)	30%

Late Penalty: Late submissions lose 10% per calendar day.

Transactional Web Applications: reservations, registrations, collaboration, inventory management, e-commerce, etc.

Course Schedule

- JavaScript Review (Day 1)
- Introduction to JavaScript Frameworks (Day 2)
- Exploring React Days 2 - 6
  - Installation
  - Components
  - Style
  - Events
  - Focus
- Project - Day 5, 6 for intro.
  - Plan/Design
  - Identify tasks
  - Prepare the development environment
  - Program the application
  - Deploy
  - Document
- Day 7 – Review
- Day 8 – Test
- Days 9 – 14 Project.
- Day 15 - Project Demo and Presentation

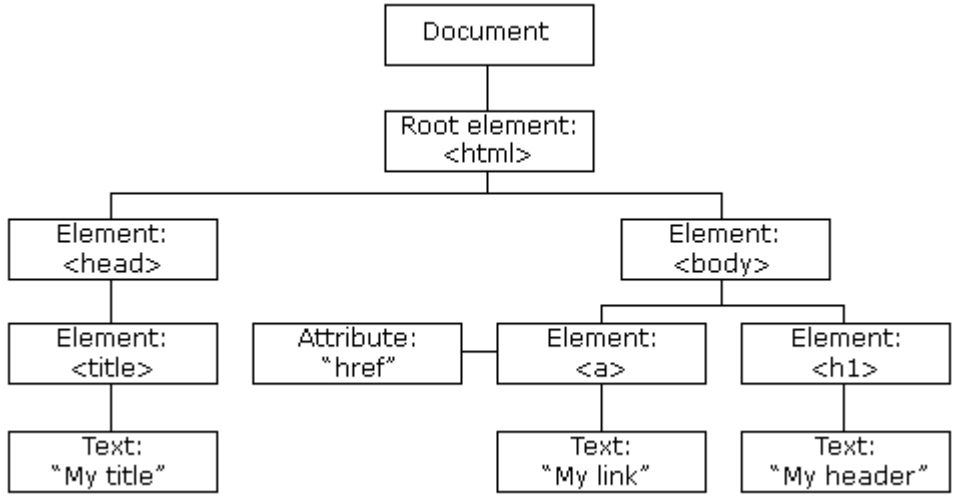
Quick refresh: Web languages

- HTML to define the content of web pages
  - Hypertext Markup Language
- CSS to specify the layout/formatting of web pages
  - Cascading Style Sheets
- JavaScript to program the behavior of web pages
  - Dynamically update content
  - Control multimedia
  - Animate images
  - And much more

DOM: Document Object Model

- A DOM is a tree of object representing all the HTML elements in a web page.
  - It is a standard for how to get, change, add or delete HTML elements
- JavaScript can:
  - Change/add/remove the HTML elements and their attributes in the page
  - Change the CSS styles in the page
  - React to HTML events in the page and create new events.
- Can use methods like `document.getElementById` to access a DOM object
- Can set a property on that object, such as `innerHTML`, to change the object's contents

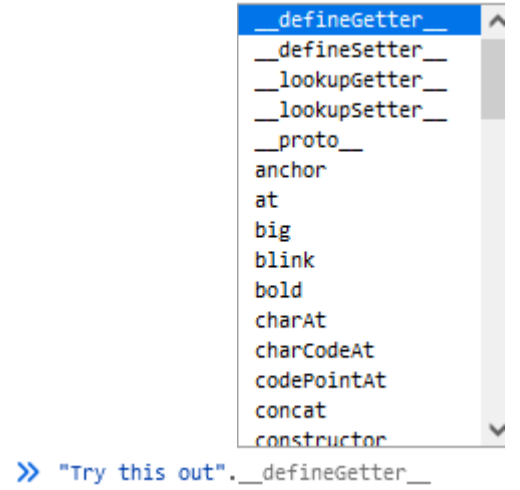
- https://www.w3schools.com/js/js_htmldom.asp
- https://www.w3schools.com/js/js_htmldom_document.asp



Client-Side Javascript

- Open your web browser of choice. Preferably Chrome or Firefox.
- Open the developer tools by hitting `CTRL + SHIFT + I` and make sure you're on the console tab.
- Type a mathematical expression like `1 + 1` into the console and hit ente

- Type any string and then . like you want to call a function. You should see a list appear of all the functions you can call on a string. There are lots!
  - var string="This is Elie";
  - Console.log(string.toUpperCase());
  - Choose the `.toUpperCase()` method and execute the statement. You should see the string in all caps as the output.



Try It! Manipulating Page Content

- Now let's try something a bit more fun!
- Go to Google's homepage, right-click the logo, and click on inspect element. In the DOM (Document Object Model) that appears in the dev tools, find the `id` attribute that the `<img>` tag is using to display the Google logo. Take note of that id value.
  - Make sure to go to google.com in the URL bar. Don't just use the default when you open a new tab (the below might not work in that case)
  - If there is no `id`, then take note of the `class` of the `<img>` tag

- In another tab, look up any (appropriate) image you think would be funny to display on the Google page and copy the URL to the image. Make sure the URL ends in a valid image file extension like `.jpg`, `.png`, etc.

- Back to the Google tab, in the dev tools console, type:

```
document.getElementById('id').srcset = '';
document.getElementById('id').src = 'image URL';
```

  - And you should see your image replace the Google logo!
  - Use `getElementByClassName('name')[0]` if you were not able to find an id.

Try It! Manipulating Page Content cont.

- Right-click on the text next to the Google search button that says "Google offered in: Français" and note the `id` of that element.

- In the dev tools console, change the text of the element to read `Made by <Your Name>` using JavaScript.
  - It will be similar to how you changed the `.src` attribute on the `<img>` element from step 8, but you'll have to use the `.innerHTML` property this time to change the text.

- Finished early?
  - Create a new function on the command line

```
function addDate(msg) { return msg + " on " +
new Date().toLocaleTimeString();}
```

  - Now, set the innerHTML to that function, passing in the earlier message you used as a parameter.

Example answer

- `document.getElementById("SIvCob").innerHTML = addDate("Made by Elie Mambou")`

Submit…

Save in Teams files: Documents/General/In-Class exercise/ folder

Submit screenshot of your revised google web page.

google.<*yourname_studentID*>

Shake some more Rust off…

→ Create a simple HTML page (index.html) on your file system that:

- Has a style for `<p>` that formats the text as blue
  - Note: Put this in the `<head>`
- Has a `<p>` element with a unique id
  - Note: Put this in the `<body>`
- Has a script tag that sets the `innerHTML` of that `<p>` element to “Welcome back to JavaScript”
  - Note: Put this in the `<body>`, but at the end
- Also, give the page a suitable title

- *Test your page by loading it in your browser*
  - If your page isn't loading properly, then you can "Inspect" the page to check for errors
    - Right-click on the web page, select Inspect.
    - Look at the Console tab for possible errors.

```
<!DOCTYPE html>
<html lang="en">
<title>Welcome Back</title>
<head>
   <style type="text/css">
      p {
        color:blue;
      }
     </style>
</head>

<body>
  <p id="main"></p>

  <script>
    document.getElementById("main").innerHTML = "Welcome back to JavaScript";
  </script>
</body>
</html>
```

.js file

- In addition to being able to put JavaScript directly in an HTML file via a <script> tag in the body, you can also load the script from a separate file.

1. Put the script in a separate `.js` file
2. Include the script file in the `<body>`
   - `<script src="filename.js"></script>`

- Similarly, you can load CSS from a separate file

1. Put the style(s) in a separate `.css` file
2. Include the style file in the `<head>`
   - `<link rel="stylesheet" href="filename.css">`

→ Change your previous HTML page to load the script and css from separate files.
   - For now, include the script file in the body.

- https://www.w3schools.com/js/js_whereto.asp

```
<!DOCTYPE html>
<html lang="en">

<title>Welcome Back External</title>
<head>
  <link rel="stylesheet" href="welcome.css">
</head>

<body>

  <p id="main"></p>
  <script src="welcome.js"></script>

</body>
</html>
```

welcome.js:

```
document.getElementById("main").innerHTML = "Welcome
back to JavaScript";
```

welcome.css:

```
p {
  color:blue;
}
```

Events

- It is a good idea to wait until the page is done loading before executing the script.
- To do this, we can create an **event listener** that will run the script at the appropriate time
  - `document.addEventListener("DOMContentLoaded", callback-function)`
- We can then include the script file in the `<head>`, and it will execute at the appropriate time.

→ Change your .js to use an event listener as above, and move the script reference to the `<head>`

- https://www.w3schools.com/jsref/met_document_addeventlistener.asp

```
<!DOCTYPE html>
<html lang="en">

<title>Welcome Back External</title>
<head>
  <link rel="stylesheet" href="welcome.css">
  <script src="welcome2.js"></script>
</head>

<body>

<p id="main"></p>

</body>
</html>
```

welcome2.js:

```
document.addEventListener("DOMContentLoaded", changeContent);

function changeContent() {
  document.getElementById("main").innerHTML = "Welcome back to JavaScript";
}
```

Or

```
document.addEventListener("DOMContentLoaded", function() {
  document.getElementById("main").innerHTML = "Welcome back to JavaScript";
});
```

Timing methods

- `setInterval` – A JavaScript function that will call another function at regular intervals
  - `setInterval(`*`otherFunction`*`, `*`intervalInMilliseconds`*`)`

- Note: This function executes something asynchronously on purpose.
  - i.e., After executing setInterval, the very next line(s) of code will execute right away. It is only after the specified interval that the "otherFunction" will execute. Thus, otherFunction executes asynchronously.
- Note: `setTimeout` is similar, but executes the otherFunction only once after a specified delay.

- https://www.w3schools.com/jsref/met_win_setinterval.asp
- https://www.w3schools.com/jsref/met_win_settimeout.asp

Exercise: Tick

- Update your HTML file to also show the time and update it every second.
  - Create a function `tick()` that will show the current time
    - `new Date().toLocaleTimeString()`
  - Use `setInterval` to call `tick()` every second.
  - Note: Keep the welcome message as before. Add a new HTML element to show the time.
  - You can embed the script or put it in an external file.

```
<!DOCTYPE html>
<html lang="en">
<title>Welcome Back With Clock</title>
<head>
  <link rel="stylesheet" href="welcome.css">
  <script src="welcome2.js"></script>
</head>
<body>
  <p id="main"></p>
  <p id="ticker"></p>
  <script>
    tick();
    setInterval(tick, 1000);

    function tick() {
      document.getElementById("ticker").innerHTML = new
Date().toLocaleTimeString();
    }
  </script>
</body>
</html>
```

Or

```
<!DOCTYPE html>
<html lang="en">
<title>Welcome Back With Clock</title>
<head>
  <link rel="stylesheet" href="welcome.css">
  <script src="welcome3.js"></script>
</head>
<body>
  <p id="main"></p>
  <p id="ticker"></p>
</body>
</html>
```

welcome3.js:

```
document.addEventListener("DOMContentLoaded", changeContent);
function changeContent() {
  document.getElementById("main").innerHTML = "Welcome back to
JavaScript";
}
document.addEventListener("DOMContentLoaded", function() {
    tick();
    setInterval(tick, 1000);
});
function tick() {
      document.getElementById("ticker").innerHTML = new
Date().toLocaleTimeString();
}
```

User Interaction

- Using events, JavaScript can also execute functions in response to user actions, such as clicking a button:
- An event can be defined/handled in several ways:
  - Inline within an HTML element
    - `<element onclick="myScript">`
  - By assignment
    - `object.onclick = function(){myScript};`
  - Using the `addEventListener()` method:
    - `object.addEventListener("click", myScript);`

→Update your HTML page to include a button that changes the colour of the Welcome Back message to alternate between red and blue every time the button is clicked.

- https://www.w3schools.com/js/js_htmldom_css.asp

```
 <button onclick="changeColour()">Change Colour</button>
```

```
function changeColour() {
  let currentStyle = getComputedStyle(document.getElementById("main"));
  console.log(currentStyle.color);
  if (currentStyle.color === "rgb(0, 0, 255)") {
    document.getElementById("main").style.color = "rgb(255, 0, 0)";
  } else {
    document.getElementById("main").style.color = "rgb(0, 0, 255)";
  }
}
```

```
function changeColour() {
  if (document.getElementById("main").style.color === "blue") {
    document.getElementById("main").style.color = "red";
  } else {
    document.getElementById("main").style.color = "blue";
  }
}
```

What's wrong with this approach?

State (Client-Side)

- Using the current value/style of an HTML element as a way to remember values is "ok", but a better approach is to explicitly store data as state.
- `localStorage` and `sessionStorage` can be used to save information on the browser.
  - The third way to store information client-side is using cookies.
- Data added to `sessionStorage` lasts only until the browser (or tab) is closed.
  - Survives over page reloads.
- Data added to `localStorage` lasts indefinitely
  - Survives over browser restarts
- The server does not have access to these storages (at least not directly).
  - The server does have access to cookies since their whole reason to exist is to allow some minor sharing of state with the server on top of the stateless HTTP protocol.
- https://www.loginradius.com/blog/engineering/guest-post/local-storage-vs-session-storage-vs-cookies/

Criteria	Local Storage	Session Storage	Cookies
Storage Capacity	5-10 mb	5-10 mb	4 kb
Auto Expiry	No	Yes	Yes
Server Side Accessibility	No	No	Yes
Data Transfer HTTP Request	No	No	Yes
Data Persistence	Till manually deleted	Till browser tab is closed	As per expiry TTL set

State (Client-Side)

- Both `localStorage` and `sessionStorage` use the same set of methods:
  - `setItem('key', 'value'),`
  - `getItem('key'),`
  - `removeItem('key')`
  - `clear()` – removes all saved data from storage
- For example,
  - `localStorage.setItem("clickCount", "0");`
- You can also directly set or get a value using the dot operator
  - `localStorage.clickCount = 0;`
- To check if the value has been set, do a truthy check
  - `if(localStorage.clickCount)`
- https://blog.bitsrc.io/how-to-store-data-on-the-browser-with-JavaScript-9c57fc0f91b0

Timing of Events

- The `DOMContentLoaded` event (on `document` object) is fired as soon as the page DOM has been loaded, without waiting for resources to finish loading.
  - i.e., stylesheets and images may not be loaded yet…
  - Stylesheets are loaded in parallel, so there is no guarantee that they will have loaded prior to the DOM.
- The `load` (lowercase) event (on `window` object) is fired when the whole page has been loaded, including all stylesheets and images.
- https://developer.mozilla.org/en-US/docs/Web/API/Document/DOMContentLoaded_event
- https://developer.mozilla.org/en-US/docs/Web/API/Window/load_event

Let's add some storage

- Let's change our HTML so that every time the button is clicked, two things happen
  1. The colour of the Welcome Message is toggled between red and blue
  2. The number of times the button is clicked is displayed in a new HTML element.
- Let's save the color in `sessionStorage`
  - This means that it will reset to default if the browser is closed
- Let's save the number of clicks in `localStorage`

- Add a listener that:
  - Initializes our `local/sessionStorage` appropriately after `window load`
    - `window.addEventListener("load", function() { … } )`
    - Set the initial style / content of our HTML elements appropriately
- Rename your `changeColor()` method to `changeColorAndCount()`
  - Update `localStorage/sessionStorage` appropriately
  - Set the updated style/values of our HTML elements appropriately.

```
// Initialize state variables and set element values/style appropriately

window.addEventListener("load", function() {

    if (!localStorage.clickCount) {

      localStorage.clickCount = 0;

    }

    document.getElementById("counter").innerHTML = localStorage.clickCount;

    if (!sessionStorage.welcomeColour) {

      sessionStorage.welcomeColour =
getComputedStyle(document.getElementById("main")).color;

    } else {

      document.getElementById("main").style.color = sessionStorage.welcomeColour;

    }

});
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>Welcome Back With Clock and Counter</title>

  <link rel="stylesheet" href="welcome.css">
  <script src="welcome6.js"></script>
</head>
<body>

  <p id="main"></p>

  <p id="ticker"></p>

  <p id="counter"></p>

  <button onclick="changeColourAndCount()">Click me</button>

</body>
</html>
```

```
function changeColourAndCount() {
  // ++localStorage.clickCount by itself works too, but including general approach
  //  in case need to increment by more than 1 in some other use case in the future.
  localStorage.clickCount = Number(localStorage.clickCount) + 1;
  document.getElementById("counter").innerHTML = localStorage.clickCount;


  if (sessionStorage.welcomeColour === "rgb(0, 0, 255)") {
    sessionStorage.welcomeColour = "rgb(255, 0, 0)";
  } else {
    sessionStorage.welcomeColour = "rgb(0, 0, 255)";
  }
  document.getElementById("main").style.color = sessionStorage.welcomeColour;
}
```

Submit…

Save in Teams files: Documents/General/In-Class exercise/Day1 folder

Submit two screenshots of your web page, one when it is red and one when it is blue

blue.<*yourname_studentID*>

red.<*yourname_studentID*>

Full solution for reference

All files in same folder

index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>Welcome Back With Clock and Counter</title>

  <link rel="stylesheet" href="welcome.css">
  <script src="welcome6.js"></script>
</head>
<body>

  <p id="main"></p>

  <p id="ticker"></p>

  <p id="counter"></p>

  <button onclick="changeColourAndCount()">Click me</button>

</body>
</html>
```

welcome.css

```
p {
  color:blue;
}
```

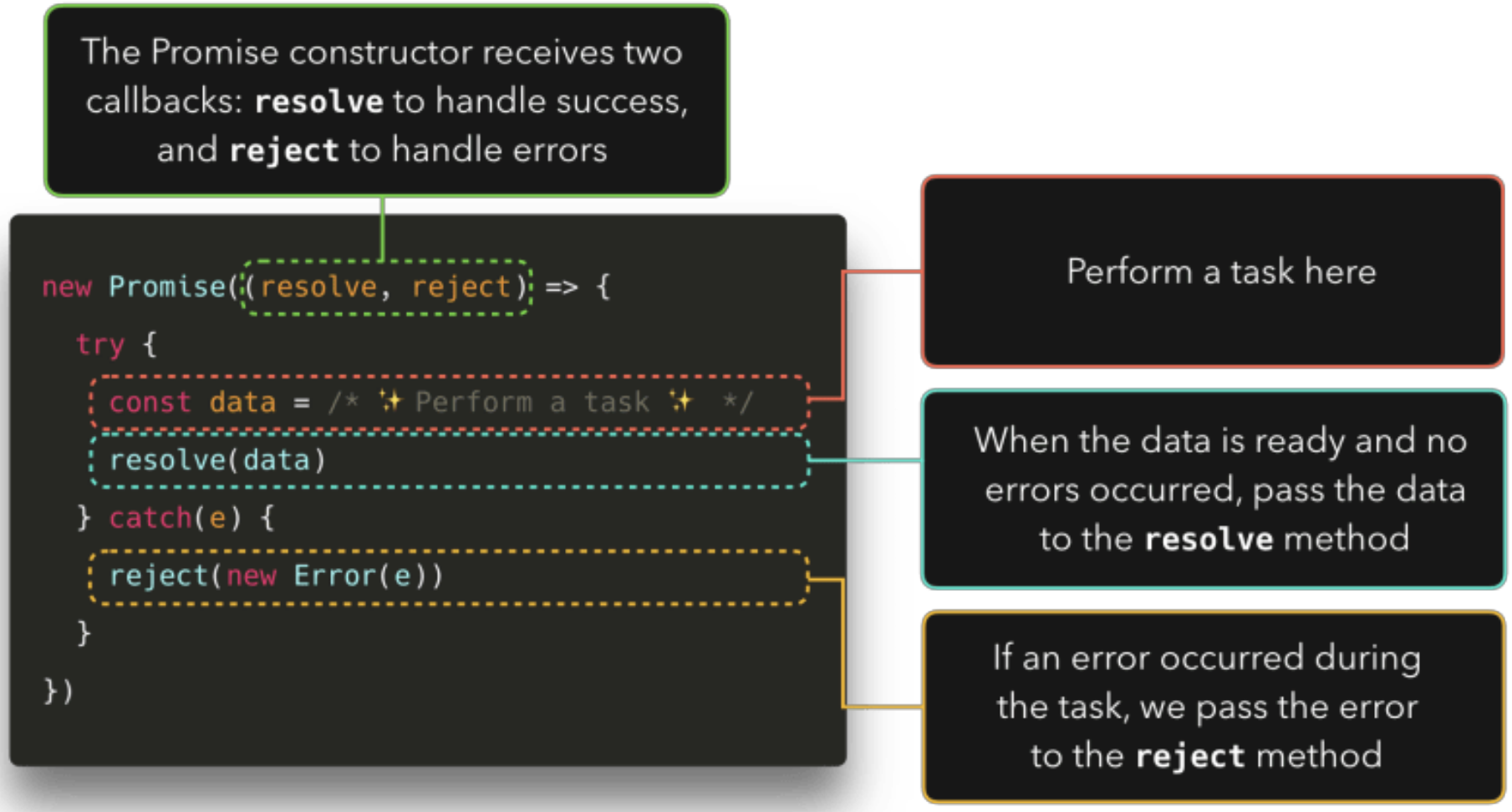
welcome6.js

```
document.addEventListener("DOMContentLoaded", function() {
    document.getElementById("main").innerHTML = "Welcome back to JavaScript";
});
document.addEventListener("DOMContentLoaded", function() {
    tick();
    setInterval(tick, 1000);
});
function tick() {
      document.getElementById("ticker").innerHTML = new Date().toLocaleTimeString();
}
// Initialize state variables and set element values/style appropriately
window.addEventListener("load", function() {
    if (!localStorage.clickCount) {
      localStorage.clickCount = 0;
    }
    document.getElementById("counter").innerHTML = localStorage.clickCount;
    if (!sessionStorage.welcomeColour) {
      sessionStorage.welcomeColour = getComputedStyle(document.getElementById("main")).color;
    } else {
      document.getElementById("main").style.color = sessionStorage.welcomeColour;
    }
});
function changeColourAndCount() {
  // ++localStorage.clickCount by itself works too, but including general approach
  //  in case need to increment by more than 1 in some other use case in the future.
  localStorage.clickCount = Number(localStorage.clickCount) + 1;
  document.getElementById("counter").innerHTML = localStorage.clickCount;
  if (sessionStorage.welcomeColour === "rgb(0, 0, 255)") {
    sessionStorage.welcomeColour = "rgb(255, 0, 0)";
  } else {
    sessionStorage.welcomeColour = "rgb(0, 0, 255)";
  }
  document.getElementById("main").style.color = sessionStorage.welcomeColour;
}
```

Promise

```
console.time("Callback Hell");
createPokemon({ name: "Eevee", type: "Normal" },
    () => {
        createPokemon({ name: "Cradily", type: "Grass" },
            () => {
                createPokemon({ name: "Ledian", type: "Grass" },
                    () => {
                        createPokemon({ name: "Gastly", type: "Psychic" },
                            () => {
                                createPokemon({ name: "Spearow", type: "Normal" },
                                    fetchPokemon)
                            })
                    })
            })
    });
```

- Promises were implemented to try and mitigate problems with the use of callback functions.
  - "Callback hell" can occur when callbacks contain callbacks in turn, and so on.
- The Promise object represents the eventual completion (or failure) of an asynchronous operation and its resulting value.
- A Promise is in one of these states:
  - **pending**: initial state, neither fulfilled nor rejected.
  - **fulfilled (or resolved)**: meaning that the operation was completed successfully.
  - **rejected**: meaning that the operation failed.
- See Mozilla developer docs: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise



- From https://dev.to/lydiahallie/javascript-visualized-promises-async-await-5gke

.then()

```
createPokemon({ name: "Eevee", type: "Normal" })
    .then(() => createPokemon({ name: "Cradily", type: "Grass" }))
    .then(() => createPokemon({ name: "Ledian", type: "Grass" }))
    .then(() => createPokemon({ name: "Gastly", type: "Psychic" }))
    .then(() => createPokemon({ name: "Spearow", type: "Normal" }))
    .then(() => fetchPokemon());
```

- Promise.then() provides a callback that will be executed once the Promise is resolved (*or rejected)

```
myPromise.then(
  function(value) { /* code if successful */ },
  function(error) { /* code if some error */ }
);
```

- In turn, it returns a promise.
- This allows promise chaining

```
Promise.then(f1)
       .then(f2)
       .then(f3)
```

- https://www.w3schools.com/js/js_promise.asp

.catch() and .finally()

- In addition to using .then() on a successful promise result, you can also use:
  - .catch() when there is an error
  - .finally() which will run whether successful or not.

```
promise
    .then(result => { ...})
    .catch(error => { ... })
    .finally(() => { ... })
```

- Note: .then() with a failure function vs .catch() are subtly different
  - https://javascript.info/task/then-vs-catch

Async/Await

```
/** Create all pokemon using async/await. */
async function createAllPokemon() {
    await createPokemon({ name: "Chadily", type: "Grass" });
    await createPokemon({ name: "Fletchling", type: "Normal" });
    await createPokemon({ name: "Dunsparce", type: "Normal" });
    await createPokemon({ name: "Gastly", type: "Psychic" });
    await createPokemon({ name: "Ledian", type: "Grass" });
    fetchPokemon();
}
```

- After a couple of years of using promises, JS developers found it cumbersome to have to chain the `.then()` calls.
- They're definitely better than using callbacks, but it would be nice if we could write asynchronous code in the same way we write synchronous code.
- In an attempt to do this, the keywords `async` and `await` were born.
  - These keywords are **syntactic sugar** for using promises.
  - From Wikipedia: Syntactic sugar is syntax within a programming language that is designed to make things easier to read or to express. It makes the language "sweeter" for human use: things can be expressed more clearly, more concisely, or in an alternative style that some may prefer.
- https://javascript.info/async-await

Web Programming II

420-JE5-AB

Instructor: Elie Mambou

Day 2:

Frameworks/React

反应

Adapted from Talib Hussain's slides

Agenda

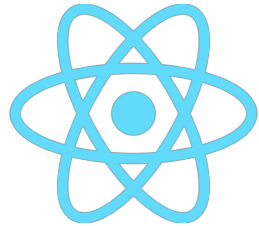
- Web Frameworks
- Introduction to React
  - Virtual DOM
  - React Element
  - JSX
  - Component
  - Props
  - Composability
  - State
  - Hooks

Web Frameworks

- Web frameworks provide a standard way to build and deploy web applications
- They often automate common activities to simplify coding
- They often promote code re-use
- They often provide many additional library function to allow developers to quickly incorporate a range of capabilities without coding unnecessarily from scratch.
- Often use Model-View-Controller approach or some variation
- Generally, there are 2 types of frameworks:
  - Server-side / back-end
    - Ruby on Rails, ASP.NET Core, Spring MVC, Django
    - Express (with Node.js)
  - Client-side / front-end
    - Angular, Vue
    - React

- https://en.wikipedia.org/wiki/Web_framework

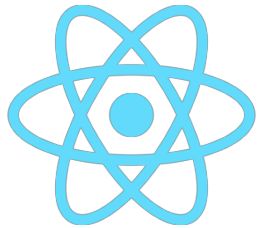
Client-Side Frameworks



- Angular and Vue are two JavaScript-based front-end web frameworks.
  - These frameworks come with many tools built-in that let you create a complete app using a particular approach.
- React is a JavaScript library that is sometimes referred to as a framework, though "technically" it is not a proper web framework.
  - React does provide a suite of capabilities that supports a particular component-based, declarative approach to web development so in that way it is does provide a certain standard way to approach building a web application.
    - It is concerned primarily with state management and rendering that state to the DOM
    - It is just the view layer
  - However, React is "unopinionated" beyond those key capabilities.
    - We might say it is an "un-opinionated framework"
  - You must use and augment those capabilities "from scratch" in your own particular way and/or use a variety of 3rd party tools to help you create a complete app
  - Becoming a skilled React developer involves having a good knowledge of 3rd-party React libraries.

- https://www.codeinwp.com/blog/angular-vs-vue-vs-react/
- https://www.freecodecamp.org/news/is-react-a-library-or-a-framework/
- https://www.htmlgoodies.com/javascript/angular-vs-react-a-tale-of-two-philosophies/
- https://rajeshnaroth.medium.com/react-convert-from-library-to-a-framework-1207786254bc

"React Frameworks"



- create-react-app
  - This is a tool that seeks to setup a React project with a set of additional tools to support a complete web application.
  - "React is a library for building user interfaces, which comprise only one part of an app. Deciding on all the other parts — styles, routers, npm modules, ES6 code, bundling and more — and then figuring out how to use them is a drain on developers. This has become known as javascript fatigue. Despite this complexity, usage of React continues to grow."
    - https://www.freecodecamp.org/news/the-cure-to-js-fatigue/
  - https://create-react-app.dev/
- next.js
  - This is a "React framework" that supports server-side rendering with React
  - https://nextjs.org/

- https://www.codeinwp.com/blog/angular-vs-vue-vs-react/
- https://www.freecodecamp.org/news/is-react-a-library-or-a-framework/
- https://www.htmlgoodies.com/javascript/angular-vs-react-a-tale-of-two-philosophies/
- https://rajeshnaroth.medium.com/react-convert-from-library-to-a-framework-1207786254bc

Let's Learn React

学习反应

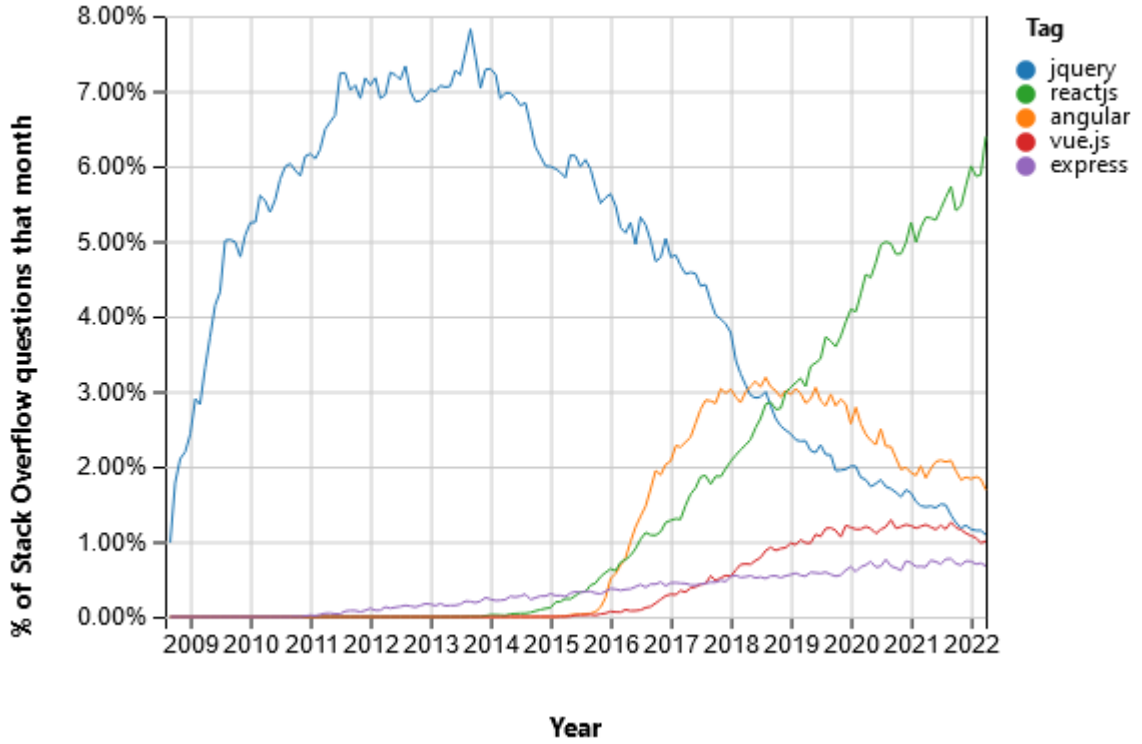
Xuéxí fǎnyìng

What is React?

- A JavaScript library
  - Created by Facebook and released in 2013
- It is a User Interface (UI) library
- It is a tool for building UI components.
- Useful when there is a lot of data changing on a page over time
- Helps build a "Single Page Application"
- React has been designed from the start for gradual adoption, and **you can use as little or as much React as you need**

- https://www.w3schools.com/whatis/whatis_react.asp
- https://reactjs.org/tutorial/tutorial.html
- https://reactjs.org/docs/hello-world.html
- https://reactjs.org/docs/add-react-to-a-website.html

- React is the most commonly used web framework as of 2021
- https://insights.stackoverflow.com/survey/2021#section-most-popular-technologies-web-frameworks
- https://insights.stackoverflow.com/trends?tags=jquery%2Cangular%2Creactjs%2Cvue.js%2Cexpress



What's the main idea?

- Essentially the core idea of React is to generate HTML dynamically.
- Using JavaScript and a special markup language called JSX (JavaScript XML), you can specify the HTML you want to generate.
  - This is called a React element. Think of it as a description of an HTML element, not the HTML itself.
  - The specifics can vary depending on data, conditional logic, loops, etc.
  - The element (typically) describes the entire UI and is made up of many other elements
- Then, that React element gets "rendered" in an HTML container.
  - The React element is rendered in the DOM element.
  - Essentially think of this as a "cut-and-paste" into the container.
- This rendered element is immutable – you can't change its children or elements.
  - An element is like a single frame in a movie – it represents the UI/web page at a certain point in time.
- To change the web page, you create a new React element and render that.
  - But, React only changes the parts of the UI that have actually changed.

→ Focus is on what the UI should look like at any given moment, not on how to change it over time.

- https://reactjs.org/docs/rendering-elements.html

Install Visual Studio Code (VSC)

- Let's get started today using an Integrated Development Environment (IDE) to help us with coding.
- For this course, we'll be using Visual Studio Code
- Download and install the correct version of Visual Studio Code for your compu
  - https://code.visualstudio.com/
- Once installed, click on the Extensions icon on the left.
- Search for and install the following extensions:
  - Node.js and Javascript Education Extension Pack
  - Prettier – Code formatter
    - Help-ShowAllCommands
    - Preferences: Open User Settings
    - Search for: "Format on save"
    - Check the box
  - Git extension pack
  - Highlight matching tag
  - Auto rename tag
  - VSCode React Refactor
  - React Extension Pack

Exercise 1: Let's jump right in…

- Let's add some React to our HTML page from our first JavaScript exercise
- Copy your `Exercise0` folder to a new folder `Exercise1`.
  - In the new folder, remove any "old" files and just keep the latest html, css and js versions.
- Open `Exercise1` in VSC

1. First, we must add 3 React script tags to `<head>`

```
<!-- Load React. -->
<!-- Note: when deploying, replace "development.js" with "production.min.js". -->
<script src="https://unpkg.com/react@18/umd/react.development.js" crossorigin></script>
<script src="https://unpkg.com/react-dom@18/umd/react-dom.development.js" crossorigin></script>

<!-- Load Babel - A Standalone build of Babel JavaScript compiler for use in non-Node.js environment -->
<script src="https://unpkg.com/babel-standalone@6.26.0/babel.js"></script>
```

   - Babel is a tool (a "transpiler") that helps you write code in the latest version of JavaScript. When your supported environments don't support certain features natively, Babel will help you compile those features down to a supported version.

2. Next, we must add a `<div>` element in `<body>` which will be used to render our React element. Usually, we use the id `"root"`.

```
<div id="root"></div>
```

   - Note: Put this after our button so that all the React content will be after our pure JavaScript content.

Exercise 1

3. In a `<script>` tag at the end of `<body>`, we will then create a React element. The script must be of type `"text/babel"` so it will be compiled as React code by Babel.

```
<script type="text/babel">
  const reactElement = <h1>Hello React World</h1>
</script>
```

   - This may look funny for now – it is JSX.

4. Finally, in the same script block (at the end), we'll get the root DOM element and render the react element into it.

```
const root = ReactDOM.createRoot(document.getElementById("root"));
root.render(reactElement);
```

   - Note: `createRoot + render` is the latest approach (React 18). Until two months ago, we would have used

     `ReactDOM.render(reactElement, document.getElementById('root'));`
   - That approach is now deprecated (will still work, but with a warning)
   - Most online examples you find will use the old approach.

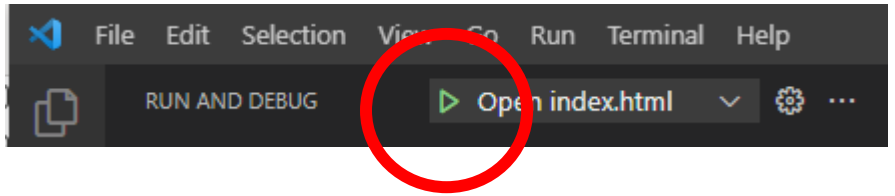
```
<!DOCTYPE html>
<html lang="en">
<title>First React Page</title>
<head>
  <link rel="stylesheet" href="welcome.css">
  <script src="welcome6.js"></script>
  <script src="https://unpkg.com/react@18/umd/react.development.js" crossorigin></script>
  <script src="https://unpkg.com/react-dom@18/umd/react-dom.development.js" crossorigin></script>
  <script src="https://unpkg.com/babel-standalone@6.26.0/babel.js"></script>
</head>
<body>
  <p id="main"></p>
  <p id="ticker"></p>
  <p id="counter"></p>
  <button onclick="changeColourAndCount()">Click Me</button>
  <div id="root"></div>

<script type="text/babel">
  const reactElement = <h1>Hello React World</h1>
  const root = ReactDOM.createRoot(document.getElementById("root"));
  root.render(reactElement);
</script>
</body>
</html>
```

Launch

- Let's get used to using some features of our IDE
- First, open index.html in the editor and/or make sure it is the active editor
- Click on the "Run and Debug" icon on the left
- Select "create a launch.json file"
- Choose "Chrome".
- This should automatically generate the following `launch.json` file

```
{
    // Use IntelliSense to learn about possible attributes.
    // Hover to view descriptions of existing attributes.
    // For more information, visit: https://go.microsoft.com/fwlink/?linkid=830387
    "version": "0.2.0",
    "configurations": [
        {
            "type": "pwa-chrome",
            "request": "launch",
            "name": "Open index.html",
            "file": "<full path to index.html>"
        }
    ]
}
```

Inspect



- Launch your HTML file by clicking the green play icon in the "Run and Debug" view.
  - Note: For now, you can also directly load this html file in your browser by double-clicking it in File Explorer / Finder. But, let's get used to using the IDE.
- Right-click on the web page and Inspect it.
- Expand the `<div>` element.
  - You'll see that our `<h1>` tag has been added as a child of `<div>`
  - Note that the div now contains actual HTML. This is what was generated by React.
- A browser can only display HTML, so React creates that HTML

- But… if you just "View Page Source", the element is still empty.
  - This is because the DOM was changed after the page was loaded.
  - Welcome to React…

Declarative vs Imperative

- Imperative programming is a programming paradigm that explicitly specifies each instruction/statement step-by-step process, often using statements that change a program's state.
  - Like describing HOW things work or HOW to do something
    - Go to the kitchen
    - Open fridge
    - Remove chicken from fridge
    - ...
    - Bring food to the table
- Declarative programming is a programming paradigm that describes what a program does without explicitly specifying its control flow
  - Like describing WHAT you want to achieve
    - I want dinner with chicken.

- https://medium.com/@myung.kim287/declarative-vs-imperative-251ce99c6c44

React is Declarative

- React's focus is on the result that will be displayed rather than on the steps for how to display it.

```
function addSingerNameToBody() {
  const bodyTag = document.querySelector('body')
  const divTag = document.createElement('div')
  let h1Tag = document.createElement('h1')
  h1Tag.innerText = "Adele"
  divTag.append(h1Tag)
  bodyTag.append(divTag)
}
```

vs.

```
function Singer(props) {
      return (<div>
                <h1>{props.name}</h1>
              </div>);
}
```

or

```
class Singer extends Component {
  render() {
    return(
      <div>
        <h1>{this.props.name}</h1>
      </div>);
  }
}
```

Multi-Page and Single Page applications

- Can use React to control parts of HTML pages or entire pages
  - Widget approach on a multi-page application
  - Some pages are still rendered and served by a backend server
- Can also use React to control the entire front-end as a "Single Page Application" (SPA)
  - Server only sends one page, then React takes over and controls the UI
  - Dynamically rewrites the web page based on user actions and changes in data
  - Makes a web site feel more like an application
  - Also, fast! Never need to go back to the server for more HTML, so eliminate most network delays

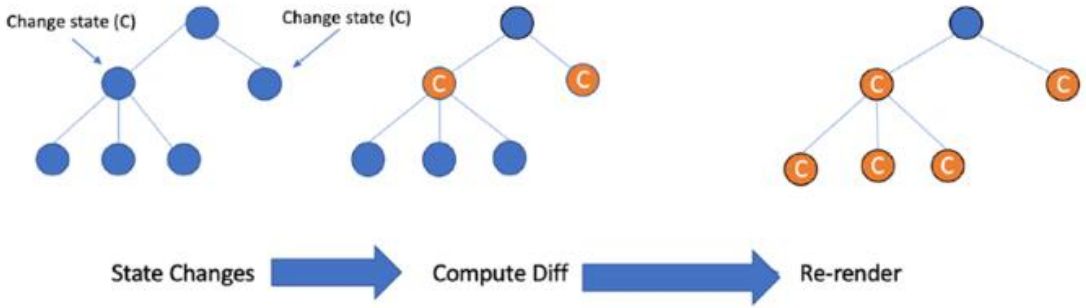
- https://en.wikipedia.org/wiki/Single-page_application

Virtual DOM

- In React, for every (real) DOM object, there is a corresponding "Virtual DOM object".
  - A virtual DOM object is like a lightweight copy of a DOM object
- A virtual DOM object has the same properties as a real DOM object, but it lacks the real thing's power to directly change what's on the screen.
- Manipulating the DOM is slow.
  - Re-rendering or re-painting of the UI is what makes it slow.
- Manipulating the virtual DOM is much faster, because nothing gets drawn onscreen.
  - Think of manipulating the virtual DOM as editing a blueprint, as opposed to moving rooms in an actual house.
- Here's what happens when you try to update the DOM in React (i.e., when you render a React element):
  1. The entire virtual DOM gets updated.
  2. The virtual DOM gets compared to what it looked like before you updated it. React figures out which objects have changed.
  3. The changed objects, and the changed objects only, get updated on the *real* DOM.
  4. Changes on the real DOM cause the screen to change.

- https://www.codecademy.com/article/react-virtual-dom
- https://dev.to/teo_garcia/understanding-rendering-in-react-i5i

Virtual DOM

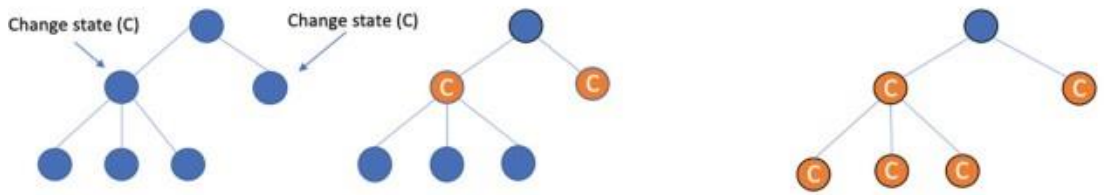


State Changes → Compute Diff → Re-render

Real DOM (Browser)

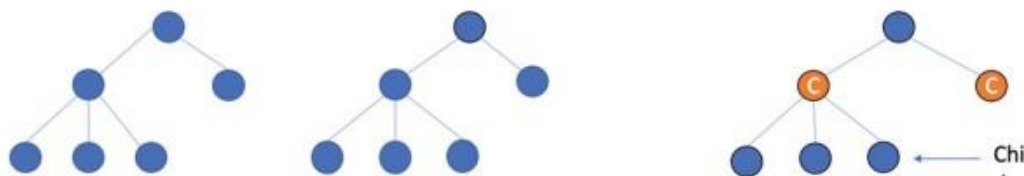
Child components re-render IF they depend on the updated state of the parent component

Virtual DOM



State Changes → Compute Diff → Re-render

Real DOM (Browser)



Child components do not re-render if they don't depend on the updated state of the parent component.


DOM
VIRTUAL DOM

Let's keep going

- Just as we did with plain old JavaScript, we can render React elements dynamically.
- So, let's duplicate our clock capability.
- As before, we must add a new div element that we will render into

```
<div id="reactTicker"></div>
```

- Next, in our script block, let's define a function that will render an appropriate React element

```
function tickReact() {
  const clockElement = <p>React Clock: {new Date().toLocaleTimeString()}</p>;
  const reactClock = ReactDOM.createRoot(document.getElementById("reactTicker"));
  reactClock.render(clockElement);
}
```

- Finally, let's setup a recurring render using `setInterval`

```
setInterval(tickReact, 1000);
```

```
<div id="root"></div>
<div id="reactTicker"></div>

<script type="text/babel">
  const reactElement = <h1>Hello React World</h1>

  const root = ReactDOM.createRoot(document.getElementById("root"));
  root.render(reactElement);

  function tickReact() {
    const clockElement = (<p>React Clock: {new Date().toLocaleTimeString()}</p>);
    const reactClock = ReactDOM.createRoot(document.getElementById("reactTicker"));
    reactClock.render(clockElement);
  }
  setInterval(tickReact, 1000);

</script>
```

JSX (JavaScript XML)

- React is designed to make it easy to create HTML elements. In particular, it does this through the use of JSX.
  - JSX makes it feel like you are "writing HTML" inside of JavaScript code.
  - JSX is a syntax extension to JavaScript
  - JSX produces React **elements**
- After compilation, JSX expressions become regular JavaScript function calls and evaluate to JavaScript objects.
  - You can use JSX inside of `if` statements and `for` loops, assign it to variables, accept it as arguments, and return it from functions:
- Instead of artificially separating technologies by putting markup and logic in separate files, React separates concerns with loosely coupled units called “components” that contain both.
  - i.e., Javascript and JSX together in the same function or class.

- https://reactjs.org/docs/introducing-jsx.html
- https://reactjs.org/docs/jsx-in-depth.html

JSX

- React doesn't require using JSX, but most people find it helpful as a visual aid when working with UI inside the JavaScript code.
  - Each JSX element is just syntactic sugar for calling React.createElement(component, props, ...children).
  - So, anything you can do with JSX can also be done with just plain JavaScript.

- However, JSX is closer to JavaScript than HTML and uses camelCase naming convention for HTML attribute names, plus a few other differences
  - `className` (instead of class)
  - The `style` attribute accepts a JavaScript object with camelCased properties rather than a CSS string.
  - `dangerouslySetInnerHTML` is React's replacement for using `innerHTML`

- https://reactjs.org/docs/dom-elements.html

React with vs. without JSX

```
class Hello extends React.Component {
  render() {
    return <div>Hello {this.props.toWhat}</div>;
  }
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Hello toWhat="World" />);
```

vs.

```
class Hello extends React.Component {
  render() {
    return React.createElement('div', null, `Hello ${this.props.toWhat}`);
  }
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(React.createElement(Hello, {toWhat: 'World'}, null));
```

Don't worry too much about the details here – but doesn't the bold code on the top look a lot simpler to write than the bold code the bottom?

- https://reactjs.org/docs/react-without-jsx.html

How do I write a comment in JSX?

```
<div>
  {/* Comment goes here */}
  Hello, {name}!
</div>

<div>
  {/* It also works
  for multi-line comments. */}
  Hello, {name}!
</div>
```

Research Activity: What React jobs are out there?

Search online job sites (Monster.ca, indeed.ca, Google jobs Canada, LinkedIn)

Find as many jobs as you can that you feel you would qualify for at the end of this program. Focus on junior programming / entry level positions

In a spreadsheet, write down what skills they require, how much they pay (if indicated) and any other key details.

When we're done, upload your spreadsheet to Teams folder.

We'll explore what you found a little then continue learning.

Component: Key Building Block of React

- Self-contained, reusable building blocks of a web application
- All about splitting up your application into small building blocks / "components"
  - Used kind of like a custom html element in JSX
  - Every component has a clear task
- The use of components makes it easy to reuse code. A component may be a cart on an e-commerce site or a login box on a social network.

- The purpose of a Component is to generate a particular React element or portion of an element
- A component generally gets an input, and changes behavior based on it.
- This behavior change generally manifests as a change in the UI of some part of the page.

- https://reactjs.org/docs/react-component.html

Component

- Components are data-dependent
  - It may vary what it generates based on input values (called **Props**) and/or stored values (called **State**)
- Components are composable
  - A component may create other child components
  - The parent "owns" the child and determines the data that flows down to the children.
    - Unidirectional Data Flow
- Components are idempotent
  - Same input produces same output

- https://reactjs.org/docs/react-component.html

Class Components vs Function Components

- Two key ways to define a Component
  - Note: A React component must always have a **capital first letter**.
- Function Components
  - Must return a React element
  - The preferred way now

```
function DisplayMessage() {
  return <h1>Hello React World</h1>;
}
```

- Class Components
  - Define a class that extends React.Component
  - Must have a render() function that returns a React element

```
class DisplayMessage extends React.Component {
  render() {
     return <h1>Hello React World</h1>;
  }
}
```

- https://www.w3schools.com/react/react_components.asp

Using a Component

- Once you've defined a component, you use it with similar syntax to normal HTML
  - The component name is essentially a capitalized HTML tag

```
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<DisplayMessage />);
```

→ Ok, let's update our HTML to use a component for our "Hello React World" message instead of a directly defined element

```
<script type="text/babel">
    function DisplayMessage() {
      return <h1>Hello React World</h1>;
    }

    const root = ReactDOM.createRoot(document.getElementById("root"));
    root.render(<DisplayMessage />);

    …
```

Props

- Props are like function arguments
- Props represent data for the component
  - Props are **read-only** (values cannot be changed within the component)
- Accessed by name of the specific prop
  - Here is the approach for a Function component. We'll discuss props for State components later.

```
function Clock(props) {
    return <p>React Clock: {props.time} </p>
}
```

- Props are sent into components like attributes of an HTML tag.
  - This is the same for both Function and Class components.

```
const reactClock = ReactDOM.createRoot(document.getElementById("reactTicker"));
reactClock.render(<Clock time={new Date().toLocaleTimeString()} />);
```

→Let's change our `tickReact()` function to use a `Clock` component

- https://reactjs.org/docs/components-and-props.html

```
function Clock(props) {
  return <p>React Clock: {props.time} </p>
}

function tickReact() {
  const reactClock = ReactDOM.createRoot(document.getElementById("reactTicker"));
  reactClock.render(<Clock time={new Date().toLocaleTimeString()} />);

}
setInterval(tickReact, 1000);
```

Composing Components

- So, now we have two components, but we are rendering each one separately.
- Generally, with React, we should be using a single root DOM element per page.
- So, let's create a new component that contains both of our existing components

```
function Main() {
       return (<div>
                  {/* This component has two children that display a message and a clock */}
                  <DisplayMessage />
                  <Clock time={new Date().toLocaleTimeString()}  />
                </div>
       );
   }
```

- → Add a Main component as above (render it in the div with id='root') and clean up your script block to simplify as appropriate.
  - Note: Inspect the page's console and you will notice we've been getting warnings for calling `createRoot` multiple times. We can move the `createRoot` up to the top of the script block to avoid this issue.
  - Note: Make sure we are showing our component right away, and not only after 1 second.

- https://zhenyong.github.io/react/docs/multiple-components.html

```
<script type="text/babel">
  const root = ReactDOM.createRoot(document.getElementById("root"));

  function Main() {
      return (<div>
                  {/* This component has two children that display a message and a clock */}
                  <DisplayMessage />
                  <Clock time={new Date().toLocaleTimeString()}  />
              </div>
      );
  }

  function DisplayMessage() {
    return <h1>Hello React World</h1>;
  }

  function Clock(props) {
    return <p>React Clock: {props.time} </p>
  }

  root.render( <Main />) // first load
  setInterval(() => root.render( <Main />), 1000);

</script>
```

Props for Class Component

- A Class component has an implicit constructor that accepts `props` as a parameter.
- You can also define a constructor explicitly (using the method name '`constructor`').
  - A call to `super(props)` as the first line in the constructor is needed to properly set things up.
- A given prop value can then be accessed anywhere within the Component using `this.props.`*`propname`*

```
class Clock2 extends React.Component {
  constructor(props) {
    super(props);
  }
  render() {
    return (<p>React Clock2: {this.props.time} </p>);
  }
}
```

→Add this class component to your script block and adjust the Main component to use Clock2 rather than Clock.

- Try it with and without the explicit constructor

State

- It is important for a React component to be able to store state
- When a state value changes, the component will automatically re-render.
- State is done differently in Function Components vs Class Components.
  - Used to only be possible to have state in Class components.
  - In 2019, React released **Hooks** capability for Function components
- When state is set, it is done asynchronously.
  - You can't set state on one line and assume that the state has been updated on the next line.
  - React will generally batch up multiple `setState()` calls
  - https://reactjs.org/docs/state-and-lifecycle.html

- We will learn the class approach as well as the functional approach, but will generally use functional components with Hooks in the course since it is now the preferred approach.
  - But, it is important to be able to use Class components since code you see and maintain in the real world will likely contain them.
  - A few (minor) capabilities are currently missing from Hooks, but are planned to be added soon.

Hooks

- React provide a state "hook" that allows function components to have access to state.
- There are 3 rules for Hooks:
  - Hooks can only be called inside React function components.
  - Hooks can only be called at the top level of a component.
  - Hooks cannot be conditional
- The state hook is called `useState`.
  - You can access it via `React.useState`
- For every variable you want to store in as state, you call `useState` with an initial value.
- `useState` returns an object with two values
  - A value for the state
  - A function for updating that state
- Using a destructuring assignment (https://www.w3schools.com/react/react_es6_destructuring.asp) , it looks like:

```
const [color, setColor] = React.useState("blue");
```

  - This gives a named value `color` that contains the state value, which is initially "blue". Think of this as a read-only "getter" for the state value.
  - This also gives us a function `setColor()` that we can use to change the state value. Think of this as a "setter" for the state value.
    - You **CANNOT** do `color = "red";` but instead have to do `setColor("red");`

- https://reactjs.org/docs/hooks-overview.html
- https://reactjs.org/docs/hooks-state.html

- Let's make a few changes to our HTML file so that we can the capability to change the colour of our message like we did with our JavaScript exercise.

1. Add a prop called `color` to our `DisplayMessage` component
   - We can use that prop to change the style used in our h1 tag in that component as follows:

     ```
     <h1 style = {{color: props.color}}>
     ```

   - Let's dissect why there are double-braces...
     - `{color: props.color}` is a JavaScript object.
     - And to embed this object in JSX you need curly braces
     - Hence, `{ {color: props.color} }`
2. For now, pass in a fixed `color` value into the `DisplayMessage` component that we are using in our `Main` component.
   1. As we did with our JavaScript exercise, let's set it to "blue" initially.

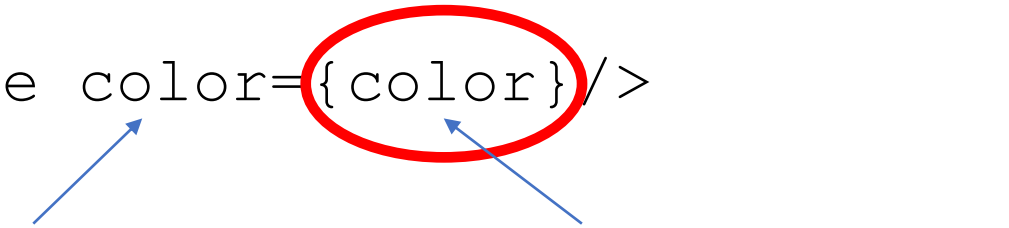
```
function Main() {
        return <div>
                  <DisplayMessage color="blue"/>
                  <Clock2 time={new Date().toLocaleTimeString()} />
                </div>
    }

    function DisplayMessage(props) {
       return <h1 style={ {color:props.color} }>Hello React World</h1>;
    }
```

3. Add a color state to our `Main` component:

```
const [color, setColor] = React.useState("blue");
```

4. Pass in the `color` state value as a prop when we render `DisplayMessage`
   - How do we use a state value? Just by its name.

```
<DisplayMessage color={color}/>
```



```
function Main() {
    const [color, setColor] = React.useState("blue");

    return (<div>
                <DisplayMessage color={color} />
                <Clock time={new Date().toLocaleTimeString()}  />
            </div>
    );
}

function DisplayMessage(props) {
  return <h1 style={{color: props.color}}>Hello React World</h1>;
}
```

- Now let's add some user interaction to our React component

1. Define a `toggle` function that accepts a color and returns `"red"` if it is `"blue"`, or `"blue"` if it is `"red"`.
   - Note: This is actually not the preferred "React" way, but we'll do it this way for now...
2. Add a button to our `Main` component
   - Remember that `onClick` is camel case.
   - `onClick` should set the color state value based on the result of the `toggle` function
     - Hint: An anonymous function is good here.

```
function Main() {
        const [color, setColor] = React.useState("blue");

        return (<div>
                    {/* This component has two children that display a message and a clock.
                        It also has a button that toggles style color. */}
                    <DisplayMessage color={color} />
                    <Clock time={new Date().toLocaleTimeString()}  />
                       <button onClick={() => setColor(toggle(color))}>
                           Click me React
                        </button>
                </div>
        );
    }

    function toggle(color) {
       if (color === "blue") {
          return "red";
       } else {
          return "blue";
       }
    }

    function DisplayMessage(props) {
      return <h1 style={{color: props.color}}>Hello React World</h1>;
    }
```

Web Programming II

420-JE5-AB

Instructor: Elie Mambou

Day 3:

React Intro cont.

反应

Adapted from Talib Hussain's slides

Agenda

- More Intro to React
  - State
  - Class Components
  - Component Lifecycle
- Getting setup
  - Node.js
  - React Developer Tools
- React from scratch
- React from create-react-app

State

- It is important for a React component to be able to store state
- When a state value changes, the component will automatically re-render.
- State is done differently in Function Components vs Class Components.
  - Used to only be possible to have state in Class components.
  - In 2019, React released **Hooks** capability for Function components
- When state is set, it is done asynchronously.
  - You can't set state on one line and assume that the state has been updated on the next line.
  - React will generally batch up multiple `setState()` calls
  - https://reactjs.org/docs/state-and-lifecycle.html

- We will learn the class approach as well as the functional approach, but will generally use functional components with Hooks in the course since it is now the preferred approach.
  - But, it is important to be able to use Class components since code you see and maintain in the real world will likely contain them.
  - A few (minor) capabilities are currently missing from Hooks, but are planned to be added soon.

State

- It is important for a React component to be able to store state
- When a state value changes, the component will automatically re-render.
- State is done differently in Function Components vs Class Components.
  - Used to only be possible to have state in Class components.
  - In 2019, React released **Hooks** capability for Function components
- When state is set, it is done asynchronously.
  - You can't set state on one line and assume that the state has been updated on the next line.
  - React will generally batch up multiple `setState()` calls
  - https://reactjs.org/docs/state-and-lifecycle.html

- We will learn the class approach as well as the functional approach, but will generally use functional components with Hooks in the course since it is now the preferred approach.
  - But, it is important to be able to use Class components since code you see and maintain in the real world will likely contain them.
  - A few (minor) capabilities are currently missing from Hooks, but are planned to be added soon.

Hooks

- React provide a state "hook" that allows function components to have access to state.
- There are 3 rules for Hooks:
  - Hooks can only be called inside React function components.
  - Hooks can only be called at the top level of a component.
  - Hooks cannot be conditional
- The state hook is called `useState`.
  - You can access it via `React.useState`
- For every variable you want to store in as state, you call `useState` with an initial value.
- `useState` returns an object with two values
  - A value for the state
  - A function for updating that state
- Using a destructuring assignment (https://www.w3schools.com/react/react_es6_destructuring.asp) , it looks like:

```
const [color, setColor] = React.useState("blue");
```

  - This gives a named value `color` that contains the state value, which is initially "blue". Think of this as a read-only "getter" for the state value.
  - This also gives us a function `setColor()` that we can use to change the state value. Think of this as a "setter" for the state value.
    - You **CANNOT** do `color = "red";` but instead have to do `setColor("red");`

- https://reactjs.org/docs/hooks-overview.html
- https://reactjs.org/docs/hooks-state.html

- Let's make a few changes to our HTML file so that we can the capability to change the colour of our message like we did with our JavaScript exercise.

1. Add a prop called `color` to our `DisplayMessage` component
   - We can use that prop to change the style used in our h1 tag in that component as follows:

     ```
     <h1 style = {{color: props.color}}>
     ```

   - Let's dissect why there are double-braces...
     - `{color: props.color}` is a JavaScript object.
     - And to embed this object in JSX you need curly braces
     - Hence, `{ {color: props.color} }`
2. For now, pass in a fixed `color` value into the `DisplayMessage` component that we are using in our `Main` component.
   1. As we did with our JavaScript exercise, let's set it to "blue" initially.

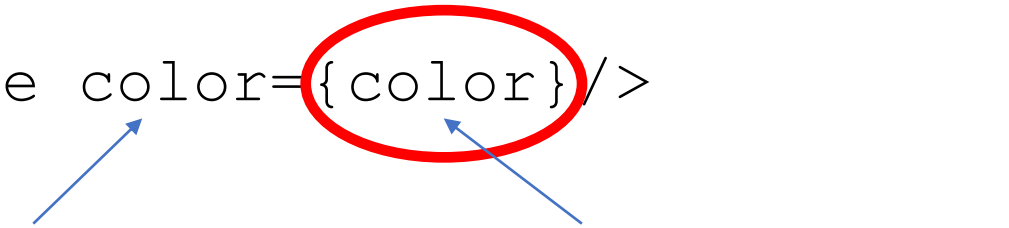
```
function Main() {
        return <div>
                  <DisplayMessage color="blue"/>
                  <Clock2 time={new Date().toLocaleTimeString()} />
                </div>
    }

    function DisplayMessage(props) {
       return <h1 style={ {color:props.color} }>Hello React World</h1>;
    }
```

3. Add a color state to our `Main` component:

```
const [color, setColor] = React.useState("blue");
```

4. Pass in the `color` state value as a prop when we render `DisplayMessage`
   - How do we use a state value? Just by its name.

```
<DisplayMessage color={color}/>
```



```
function Main() {
    const [color, setColor] = React.useState("blue");

    return (<div>
                <DisplayMessage color={color} />
                <Clock time={new Date().toLocaleTimeString()}  />
            </div>
    );
}

function DisplayMessage(props) {
  return <h1 style={{color: props.color}}>Hello React World</h1>;
}
```

- Now let's add some user interaction to our React component

1. Define a `toggle` function that accepts a color and returns `"red"` if it is `"blue"`, or `"blue"` if it is `"red"`.
   - Note: This is actually not the preferred "React" way, but we'll do it this way for now...
   - Better way: Use conditional (ternary) operator
2. Add a button to our `Main` component
   - Remember that `onClick` is camel case (JSX requires this).
   - `onClick` should set the color state value based on the result of the `toggle` function
     - Hint: An anonymous function is good here.

```
function Main() {
        const [color, setColor] = React.useState("blue");

        return (<div>
                    {/* This component has two children that display a message and a clock.
                        It also has a button that toggles style color. */}
                    <DisplayMessage color={color} />
                    <Clock time={new Date().toLocaleTimeString()}  />
                       <button onClick={() => setColor(toggle(color))}>
                           Click me React
                        </button>
                </div>
        );
    }

    function toggle(color) {
       if (color === "blue") {
          return "red";
       } else {
          return "blue";
       }
    }

    function DisplayMessage(props) {
      return <h1 style={{color: props.color}}>Hello React World</h1>;
    }
```

And a counter…

- To match the functionality of our earlier JavaScript example, let's add a counter too.

1. In `Main`, add a new `useState` for a counter variable, initialized to 0.
2. Above the button, add a `<p>` tag that displays the value of the counter.
3. In the `onClick` for the button, update the anonymous function to also increment the counter state variable.

- You should now have a web page that performs the same set of changes using JavaScript and using React.
  - The only difference now is that our React code's data doesn't persist if the tab is closed.

```
function Main() {
       const [color, setColor] = React.useState("blue");
       const [count, setCount] = React.useState(0);

       return (<div>
                  {/* This component has two children that display a message and a clock.
                        It also has a button that counts clicks and toggles style color. */}
                  <DisplayMessage color={color} />
                  <Clock time={new Date().toLocaleTimeString()}  />
                  <p>{count}</p>
                  <button onClick={() => {setColor(toggle(color)); setCount(count+1)}}>
                       Click me React
                  </button>
                </div>
       );
   }
```

Submit…

Save in Teams files: Documents/General/Class Exercise/Exercise1 folder

Submit two screenshots of your web page, one when it is red and one when it is blue

blue.<*yourname*>

red.<*yourname*>

State & Class Components

Before we proceed more with state, let's learn about Class components so we have a deeper understanding of how React components work.

State in Class Component

- A Class component has a built-in state object
- Initialize the state object in the constructor.
  - The state is a plain JavaScript object
  - `this.state = { key-value pairs }`
- Access state values using `this.state.variablename`
- Change the state with `this.setState()`
  - Calling the `setState` method results in a call to the `render()` method of the component
  - Do NOT use "this.state = … " in any method other than the constructor.

- https://www.w3schools.com/react/react_state.asp

- Let's create a `Main2` component that is a Class component but does all the same things as our `Main` component.

1. Create the class component
   - It needs to extend `React.Component`
   - Its `render()` function should contain essentially the same logic as the functional component.
2. Using class field syntax, create a function called `updateColorAndCount` that will:
   a) Determine new color and count values based on the existing state
      - Remember: Access a state variable using the format: `this.state.color`
   b) Call `this.setState` to update the state with those values.
3. Update the button to call `updateColorAndCount`
   - `<button onClick={this.updateColorAndCount}>`
   - ('this' works since we used class field syntax)
4. Instead of rendering Main, render Main2.

- For discussion on handling events using class field syntax, see https://reactjs.org/docs/handling-events.html

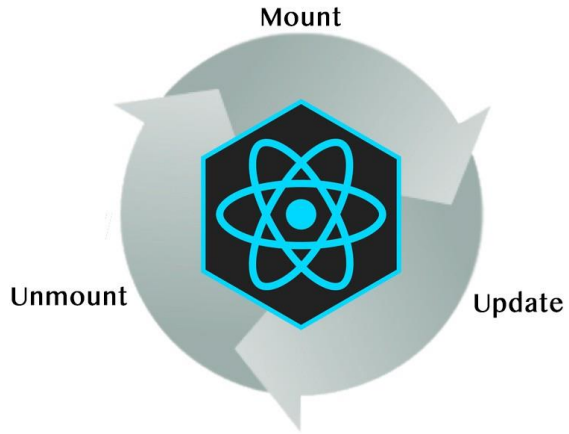
```
class Main2 extends React.Component {
    constructor(props) {
      super(props);
      this.state = {color: "blue", count: 0};
    }

  updateColorAndCount = () => {
      if (this.state.color == "blue") {
        this.setState({color: "red", count:this.state.count + 1});
      } else {
        this.setState({color: "blue", count:this.state.count + 1});
      }
    }

    render() {
      return (<div>
                <DisplayMessage color={this.state.color} />
                <Clock time={new Date().toLocaleTimeString()}  />
                <p>{this.state.count}</p>
                <button onClick={this.updateColorAndCount}>
                        Click me React
                </button>
              </div>
      );    }  }
```

Or…

```
    updateColorAndCount() {
      let newColor;
      if (this.state.color == "blue") {
        newColor = "red";
      } else {
        newColor = "blue";
      }
      let newCount = this.state.count + 1;
      this.setState({color: newColor, count: newCount});
    }
```

```
<button onClick={() => {this.updateColorAndCount();}}>
```

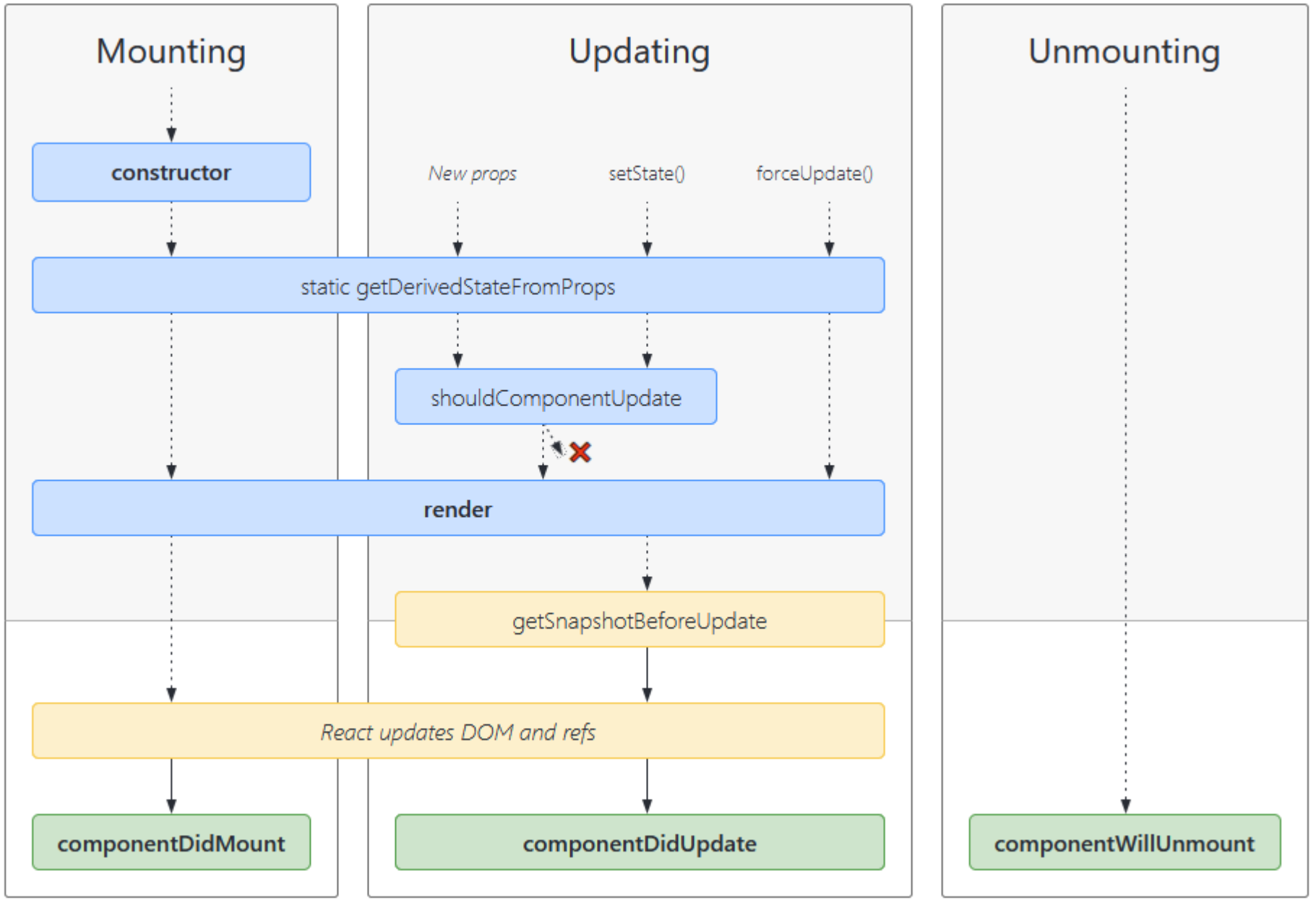
Component Lifecycle



- Mounting - Putting Elements into the DOM
- Updating – Happens whenever there is a change in a component's props or state
- Unmounting – Component is removed from the DOM

- https://reactjs.org/docs/react-component.html
- https://www.w3schools.com/react/react_lifecycle.asp
- https://payalpaul2436.medium.com/10-main-core-concept-you-need-to-know-about-react-303e986e1763

Component Lifecycle



Component Lifecycle: Mounting

- React has four built-in methods that gets called, in this order, when mounting a component:

- `constructor()`
  - Accepts props
  - Set up initial state (not based on props)
- `getDerivedStateFromProps()`
  - Set state based on props
  - Why? Since constructor is only called once and if different props are passed in by the parent during a re-render, state wouldn't change without this method
- `render()`
  - Outputs HTML to the DOM
- `componentDidMount()`
  - Run any statements that require the component to already be in the DOM.

- The `render()` method is required and will always be called, the others are optional and will be called if you define them.

- https://www.w3schools.com/react/react_lifecycle.asp

Component Lifecycle: Updating

- React has five built-in methods that gets called, in this order, when a component is updated:
- `getDerivedStateFromProps()`
  - Updates state based on any changes in props
- `shouldComponentUpdate()`
  - Return a Boolean value that specifies whether React should continue with the rendering or not.
  - This only exists as a performance optimization. Generally should not be used.
- `render()`
  - Re-render the HTML to the DOM, with appropriate changes based on new props and/or state
- `getSnapshotBeforeUpdate()`
  - Even though the component has been mounted, you still have access to the value of props and state from before the update as well as some information from the DOM before it is changed (e.g., scroll position).
  - Any value returned is passed as a parameter to `componentDidUpdate`
  - Not commonly used.

```
getSnapshotBeforeUpdate(prevProps, prevState) {
   document.getElementById("div1").innerHTML =
   "Before the update, the favorite was " + prevState.favoritecolor;
 }
```

- `componentDidUpdate()`
  - Executed last, with access to current props and state. Run any statements that require the component to already be in the DOM.

```
componentDidUpdate() {
   document.getElementById("div2").innerHTML =
   "The updated favorite is " + this.state.favoritecolor;
 }
```

- The render() method is required and will always be called, the others are optional and will be called if you define them
- In this course, we will mostly use `getDerivedStateFromProps()` and `render()`

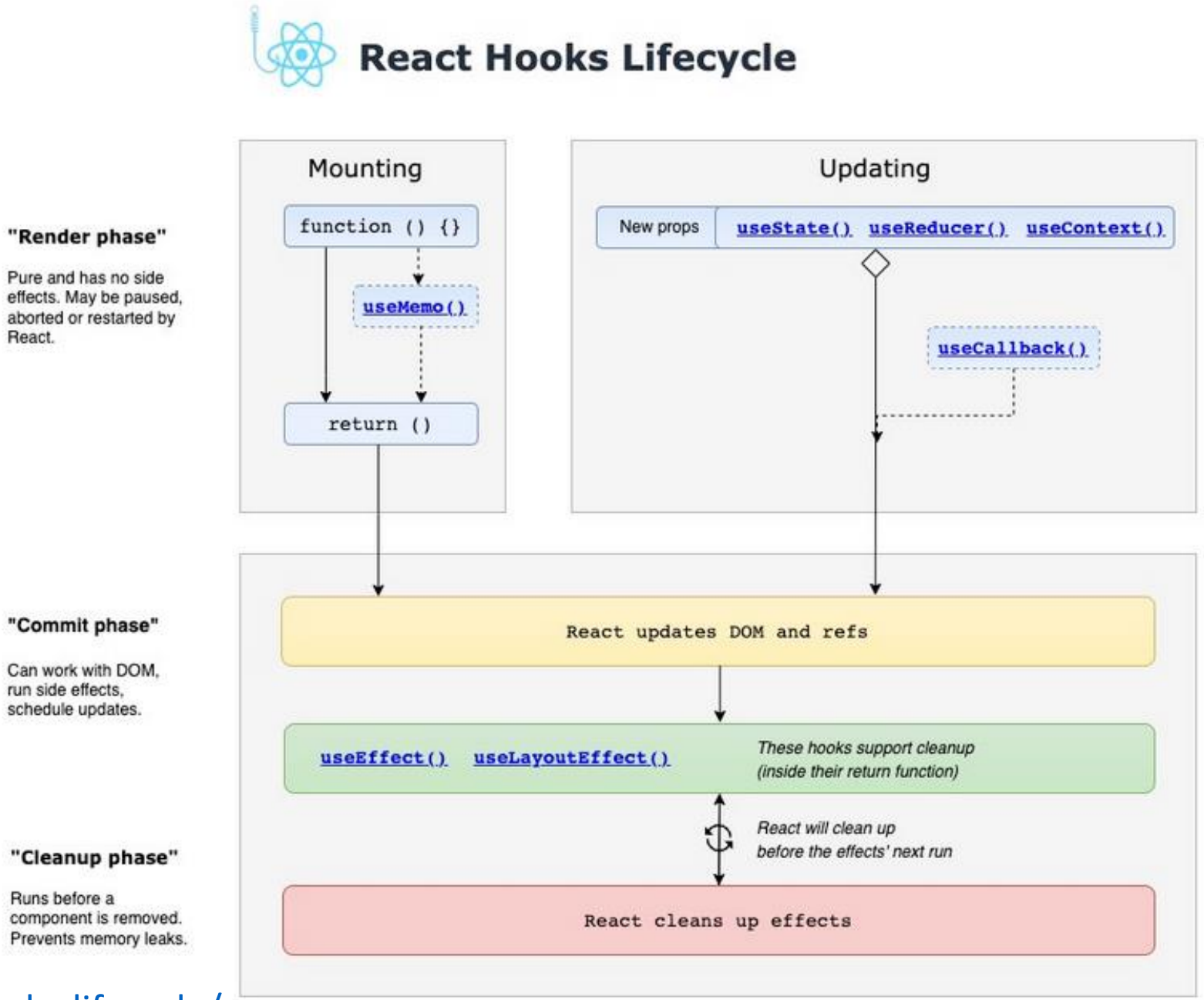
Component Lifecycle: Unmounting

- React has only one built-in method that gets called when a component is unmounted:
  - `componentWillUnmount()`
    - Called when component is about to be removed.
    - Should be used to perform any necessary cleanup.
    - Do not call `setState()` here since component will never be re-rendered
    - Once a component is unmounted, it will never be mounted again.

- https://reactjs.org/docs/react-component.html#componentwillunmount

Back to Function Components & Hooks

- These lifecycle methods for Class components can be somewhat complicated to learn and code.
- For function components, two hooks are used to provide most of the same capability, in a much easier-to-learn and use approach.
- `useState` we've learned about
  - State is actually created the first time our component renders, so this hook occurs during the first render phase, not the mount phase (unlike the constructor in a Class component).
  - During subsequent renders, `useState` gives the current state.
- Another hook is `useEffect`. This lets you perform side-effects in function components
  - `useEffect` is run after the first render and after every update
  - Does the same job as `componentDidMount` and `componentDidUpdate`.
  - React guarantees the DOM has been updated by the time it runs the effects.
- If `useEffect` returns a function, then that function is treated as a "cleanup" operation that React will call when the component unmounts.
  - Does the same job as `componentWillUnmount`.
  - But, since it is run after every render, it also cleans up effects from the previous render before running effects the next time.
- https://reactjs.org/docs/hooks-state.html
- https://reactjs.org/docs/hooks-effect.html



- https://wavez.github.io/react-hooks-lifecycle/
- https://reactjs.org/docs/hooks-reference.html

`useEffect` Syntax

- Run effect only on mount (empty array in 2nd parameter to useEffect)

```
React.useEffect(() => {
    // Perform appropriate update
}, []);
```

- Run effect only when state stateVar1 or stateVar2 have changed

```
React.useEffect(() => {
    // Perform appropriate update
}, [stateVar1, stateVar2]);
```

- Run effect on every re-render (no 2nd parameter)

```
React.useEffect(() => {
    // Perform appropriate update
});
```

- Run effect with cleanup (returns a function, 2nd parameter varies as above)

```
React.useEffect(() => {
    // Perform appropriate update
    return () => {
       // Perform cleanup operation
    }
});
```

- https://betterprogramming.pub/react-component-lifecycle-methods-with-react-hooks-efcd04987805
- https://javascript.plainenglish.io/lifecycle-methods-substitute-with-react-hooks-b173073052a

`localStorage` and `sessionStorage`

- So, let's add the missing piece to our code – using `localStorage` and `sessionStorage` to persist our data.
- When we initialize our state with `useState`, let's grab the value from storage if it exists, or set it to a default value if it does not.

```
const [color, setColor] = React.useState(
  JSON.parse(sessionStorage.getItem('color')) || "blue"
);
```

- After our component has rendered, let's update our `localStorage` with the latest values using `useEffect()`.

```
React.useEffect( () => {
  sessionStorage.setItem('color', JSON.stringify(color));
}, [color]);
```

- What does this syntax mean?
  - `useEffect( () => { actions to take } , [dependencies]);`
- The dependencies mean "if any of these state values change, then please run this effect"

→ Update your HTML file with the above, extended to include count. Put count into `localStorage` as we did in our JavaScript example

- https://www.robinwieruch.de/local-storage-react/

```
function Main() {
      const [color, setColor] = React.useState(JSON.parse(sessionStorage.getItem('color')) || "blue");
      const [count, setCount] = React.useState(JSON.parse(localStorage.getItem('count')) || 0);

      React.useEffect(() => {
          sessionStorage.setItem('color', JSON.stringify(color));
          localStorage.setItem('count', JSON.stringify(count));
      }, [color, count]);

      return (<div>
                {/* This component has two children that display a message and a clock.
                      It also has a button that counts clicks and toggles style color. */}
                <DisplayMessage color={color} />
                <Clock time={new Date().toLocaleTimeString()}  />
                <p>{count}</p>
                   <button onClick={() => {setColor(toggle(color)); setCount(count+1)}}>
                  Click me React
                   </button>
              </div>
      );
   }
```

Submit...

Save in Teams files: Documents/General/Class Exercise/Exercise1 folder

Submit two screenshots of your web page:

First, click a few times and capture a screenshot when the "Hello React World" message is red.

Then, open a new browser tab and visit localhost:3000. The "Hello React World" message should now be blue but the counter should be the same.

Without clicking anything, take a second screenshot.

session1.<*yourname*> [red, #]

session2.<*yourname*> [blue, same #]

Let's Start "Developing"

So far, we've been creating our simple React program "old-school" in an HTML file to create a simple page

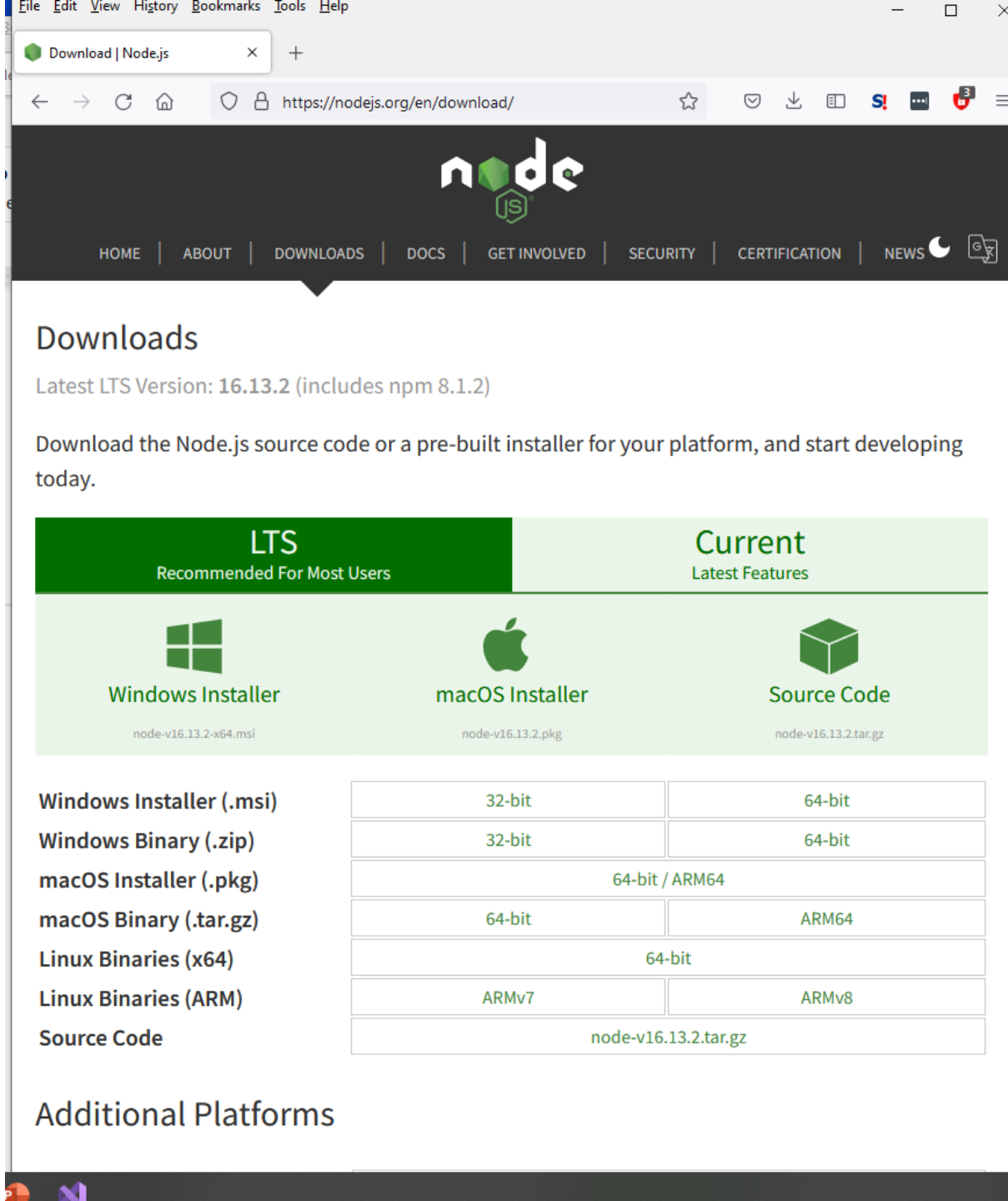
Now, let's start using appropriate development tools to start coding up a web application

Exercise 2

- Install Node.js
- Run a “Hello World” program.
- Debug a "Hello World" program

Install Node.js

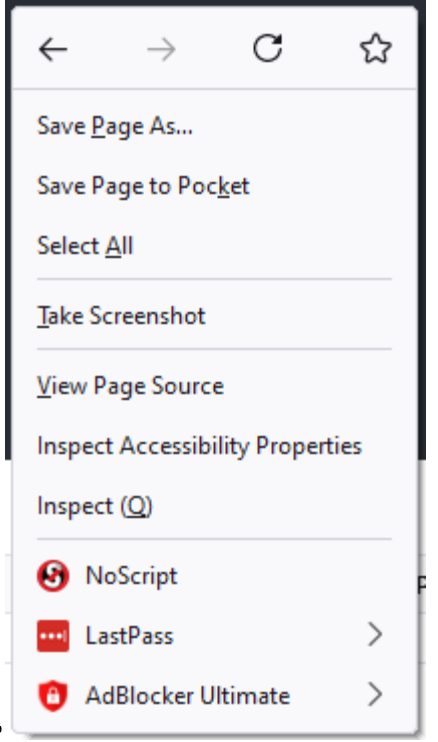
- https://nodejs.org/en/download/
- Choose LTS version (Long-Term-Support)
- Choose version appropriate for your operating system (usually 64 bit these days).
- During install: Don't need Chocolatey, so say no to that or else install will take a lot longer
- Go into terminal in VSC and type `node --version`



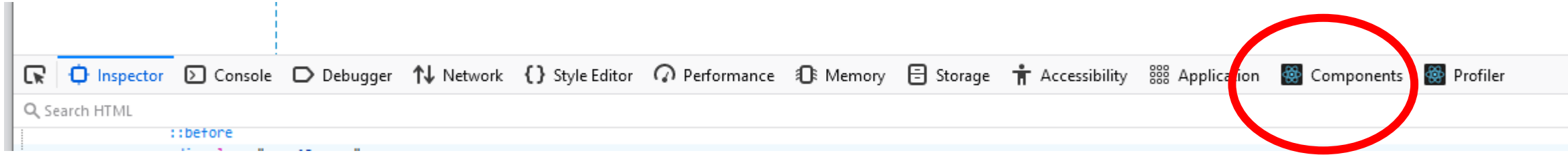
Node Package Manager (npm)

- One of the most popular package managers for JavaScript packages. A package manager is a tool to create project environments and easily import external dependencies.
  - There is also Yarn for Javascript
  - Other languages have their own popular package managers (e.g., pip in Python, RubyGems in Ruby on Rails, NuGet for .Net, Maven and Gradle for Java and other languages)
- You can use npm to install just about any library or framework available in the npm registry www.npmjs.com
  - On www.npmjs.com you can search for packages, view their readme, see stats on their popularity, access their github code, etc.
- From the command-line in a terminal, you can run a variety of npm commands to setup and configure your project and its packages, as well as run your program.
  - `npm init`
    - Sets up a new or existing npm package. This results in a file being created in your project called `Package.json`.
    - `Package.json` is a manifest file containing all key configuration info about your app and its dependencies
    - A dependency = a package that your project uses (and what version of that package)
  - `npm i <package-name>`
    - Installs the indicated package
  - `npm start`
    - Runs your program based on the "start" property of a package's "scripts" object. This can be found in `Package.json`.
- All packages are automatically installed in a project folder called `node_modules`.
  - This folder can be VERY large in size when you have many package (250MB – 500MB is common).
  - When submitting your code, this folder can be omitted to save space (Lea may not let you upload otherwise).
  - It is important to include a readme that contains the instructions listing all the packages that need to be installed.

React Developer Tools

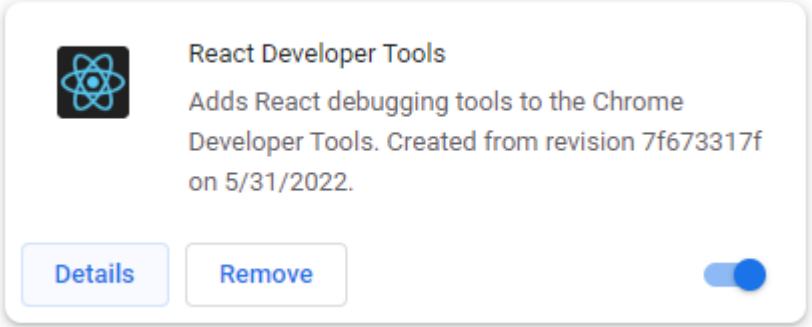


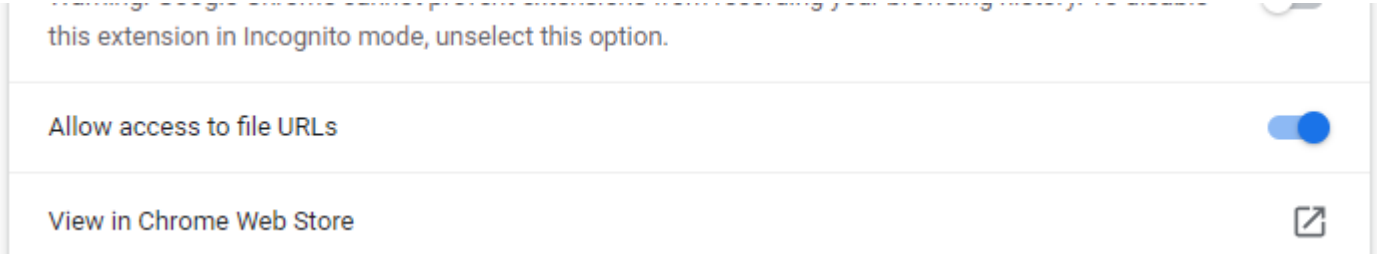
- In Chrome, search for "chrome web store react developer tools"
- Add the Extension
- Ctrl-Shift-J on PC Cmd-Option-J on Mac to open up the developer tools
- In Firefox, search for "firefox react developer tools", and add the add-on.
  - When you are on a webpage that uses React, the react icon will appear near the top right of the Firefox window
  - Right-click on a page, choose "Inspect".
  - You can see React details in the "Components" tab.



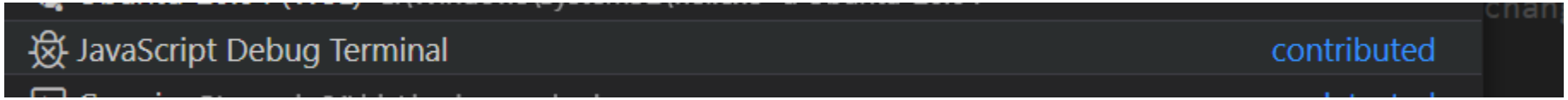
In Chrome

- Got to Setting (top right three dots)
- Select 'Extensions'
- Choose the React Developer Tools
- Enable "Allow Access to File URLs"

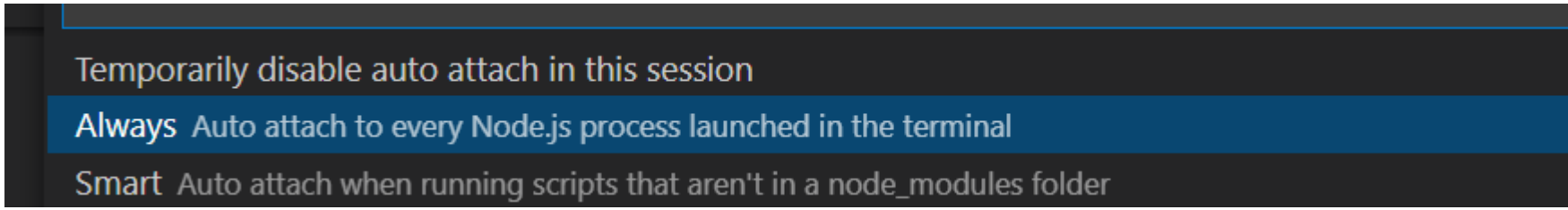




Configure VSC



- Help->Show All Commands (Ctrl-Shift-P or CMD-Shift-P)
  - Terminal: Select Default Profile
  - Choose JavaScript Debug Terminal
- Help->Show All Commands
  - Debug: Toggle Auto Attach
  - Choose Always



First React Project, from Scratch

- Create a repository folder where you will store projects for this course (e.g., `web2repos`)
- Create a subfolder `exercise2a`
  - Lower-case is a convention
- Open Visual Studio code
- Select File -> Open Folder and open the `exercise2a` folder

Exercise 2a

- Select Terminal -> New Terminal
- In terminal:
  - `npm init`
    - This creates a basic project setup
    - Press enter to accept all the default values and then click 'yes'
  - `npm i react react-dom react-scripts`
    - This installs the key packages needed for React.
- Create `src` folder
  - `App.js`

```
function App() {
    return <div>Hello World</div>
}
export default App;
```

    - This is defining a React component
    - We use export so that we access this component from another JavaScript file.
    - The component name must be capitalized. The JSX tag name convention is that lower-case names refer to built-in components and capitalized names refer to custom components.
      - https://reactjs.org/docs/jsx-in-depth.html#user-defined-components-must-be-capitalized
  - `index.js`

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import App from './App';

/* First code that is executed in the browser when loaded*/
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<App />);
```

Exercise 2a

- Create `public` folder. Create file `index.html`.

```
<!DOCTYPE html>
<html>
  <head>
    <title>My First React App</title>
</head>
<body>
    <noscript>You need to enable JavaScript to run this app.</noscript>
    <div id="root"></div>
  </body>
</html>
```

- Change the `scripts` section in `package.json`

```
"scripts": {
    "start": "react-scripts start",
    "build": "react-scripts build",
    "test": "react-scripts test",
    "eject": "react-scripts eject"
},
```

- We're ready to run. In terminal, run the app using:
  - `npm start`
- In browser, visit `localhost:3000`

Strict Mode

- Right-click to view page source or inspect.
  - Note there is a "Component" tab on the far right. Click on it to see our `<App />` component
  - You'll notice a warning that our component is not running in Strict Mode.
    - This is useful to enable so that we can catch more errors during development.
- Change our render call in `index.js` to:

```
root.render(<React.StrictMode>
              <App />
            </React.StrictMode>);
```

VS Code Debugger

- Let's configure the debugger:
  - Click on the Run and Debug Icon (Ctrl-Shift-D)
  - Click on the gear button or "Create a launch.json"
- Select "Chrome"
  - This will create a `launch.json` file
  - Change 8080 to 3000.
- Now, Click "Add Configuration"
  - Select "Run '`npm start`' in a debug terminal"
  - This will add a second configuration to the `launch.json`.
- Finally, we will manually add a "compound".
  - https://code.visualstudio.com/docs/editor/debugging#_compound-launch-configurations
  - After the closing square bracket for "configuration", add a comma and the following:

```
"compounds": [
        {
          "name": "React Server and Browser",
          "configurations": [
              "Run npm start",
              "Launch Chrome against localhost"
            ],
          "stopAll": true
        }
      ]
```

- Save the `launch.json` file.
- Click the Run and Debug icon, pull-down the launch menu and select the new "React Server and Browser" configuration.
  - You now have a 1-click method to launch in debugging mode.
- Set some breakpoints in index.js and/or App.js.
  - When you hit a breakpoint, note that there is a call stack on the left. You can click on different parts of the call stack to jump to that code.

VS Code misc.

- Shift-Alt-F (PC) or Shift-Option-F (Mac) to reformat code
- Ctrl-Click on a function name will pop-up a view. On the right of the pop-up, different code uses of the function appear. Double-click on one of them to go to that place in the code.

Submit…

Save in Teams files: Documents/General/Class Exercise/Exercise2 folder
Submit a screenshot of your web page

fromScratch.<yourname>

Exercise 2b – create-react-app

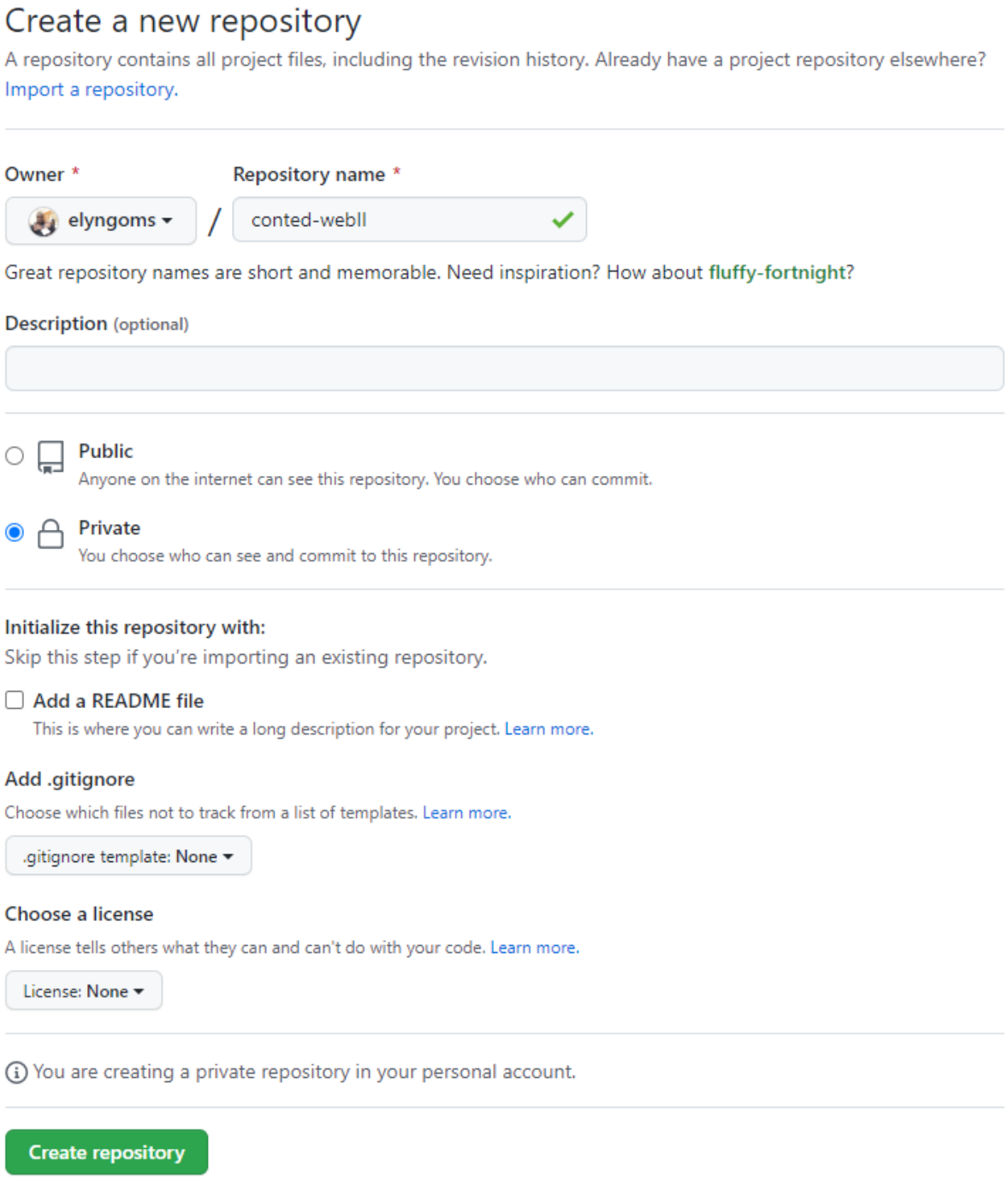
- In Visual Studio Code,
  - Open your repository folder
  - In terminal, type
    - `npx create-react-app myreactapp`
  - This will create and configure an initial React app in its own folder
- File -> Close Folder
- File -> Open Folder and open `myreactapp`
  - We want to do this to make sure the root of our workspace is right.
- Click around the files to get a sense of what is there. The main `src/` files we really need are `App.js` and `index.js`.
- In Terminal, run the program
  - `npm start`
- In a browser, visit `localhost:3000` to see the test app running.
- Also, if you click on the Source Control icon, you'll notice that we've already been setup with a git repository.
  - If you open the Commit pull-down, you'll see we have one commit already "Initialize Project using Create React App"

Git setup

- Before we start making changes to our code, let's get setup on GitHub.
- If you don't have a GitHub account, go to github.com and create one.
- Once signed in on GitHub, click the green "New" button on the left

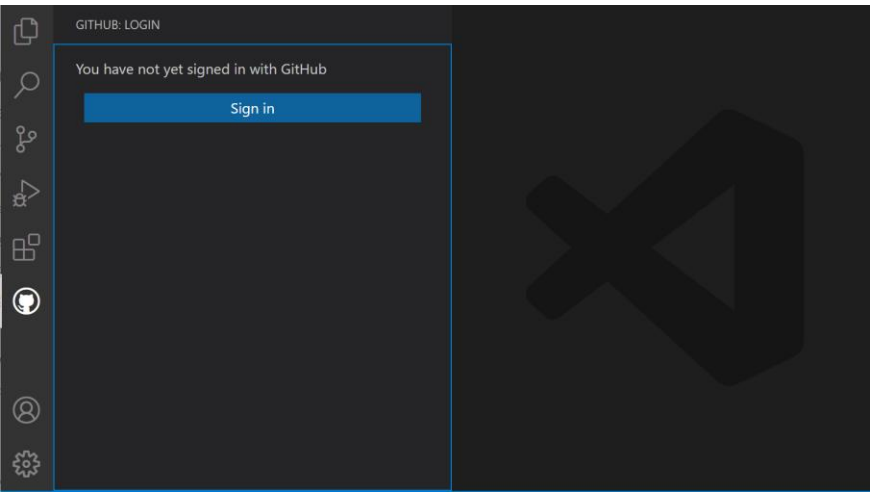
  - Choose "Private" repository
  - Click the green "Create Repository" button when done.

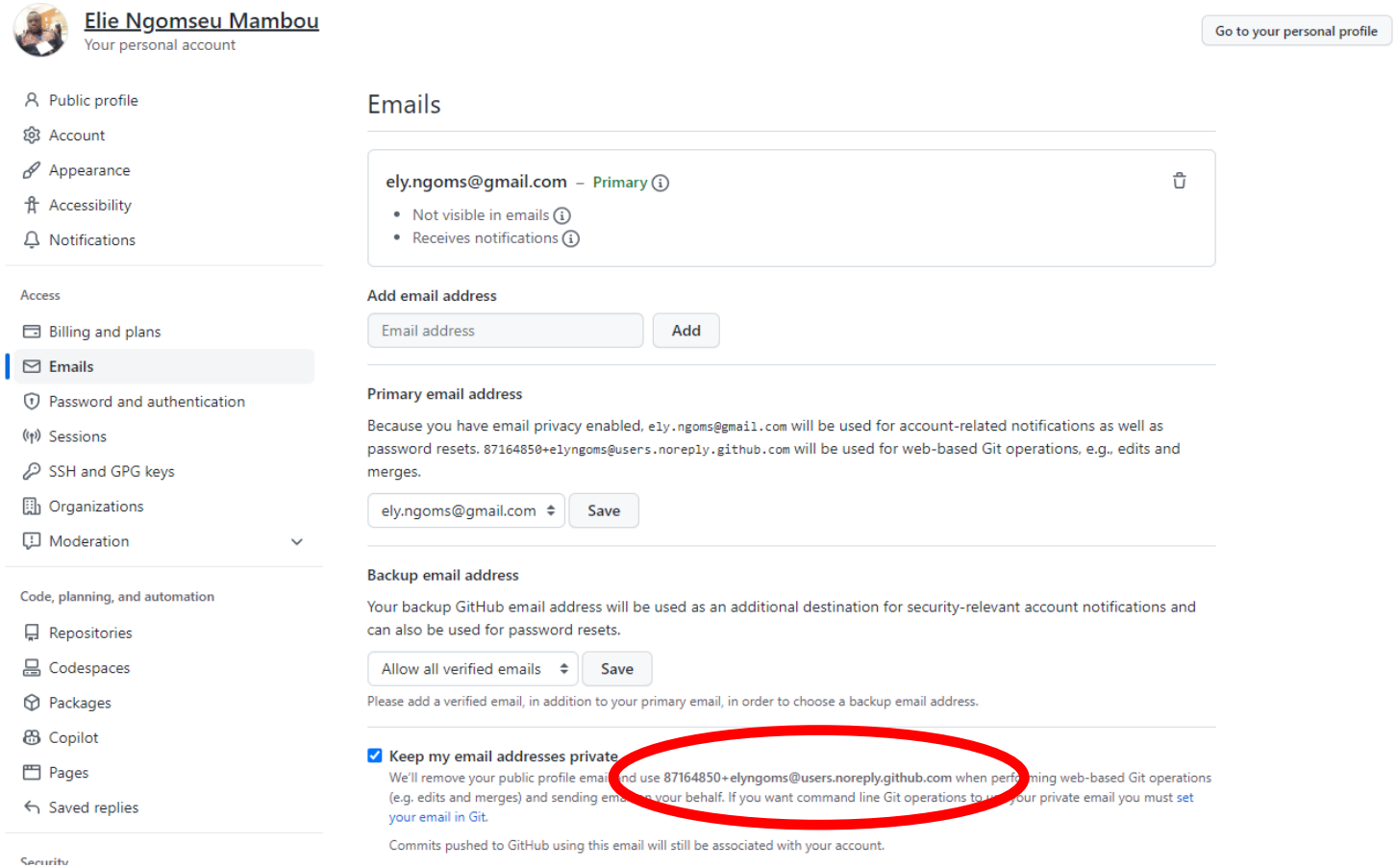
- https://docs.github.com/en/get-started/importing-your-projects-to-github/importing-source-code-to-github/adding-locally-hosted-code-to-github#adding-a-local-repository-to-github-using-git



Git setup

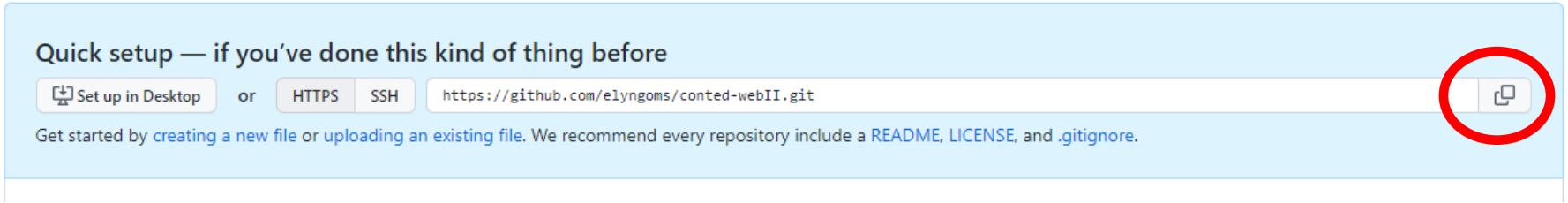
- Go to github profile > settings > email, Look under "Keep my email addresses private".
  - Recommend you check it if not checked.
- In VS Code,
  - Click on the Extensions icon. Search for and install the "Github Pull Request and Issues" extension.
  - Once installed, click on the new GitHub icon and sign in
    - Follow the prompts to authenticate with GitHub in the browser and return to VS code





Git setup

- Back in GitHub, go to your new project page.
  - At the top in the quick setup section, copy the link for your repository to the clipboard



- In VS Code Terminal,
  - https://www.jcchouinard.com/install-git-in-vscode/
- Go back to GitHub and you should now see your code in the project page.
- Now, you're good to go…

Web Programming II

420-JE5-AB

Instructor: Elie Mambou

Day 4:

React App

反应

Adapted from Talib Hussain's slides

Agenda

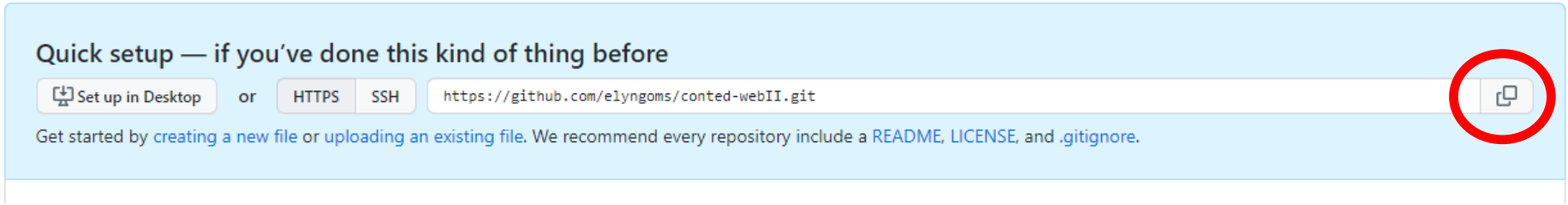
- React Coding
  - Component focus
  - Using useEffect()
  - Separation of concerns
- Test/Project dates
  - Note: Test is Wednesday, November 30.
- React conventions/organization
- Project Brainstorming
- Routing

Exercise 2b – create-react-app

- In Visual Studio Code,
  - Open your repository folder
  - In terminal, type
    - `npx create-react-app myreactapp`
  - This will create and configure an initial React app in its own folder
- File -> Close Folder
- File -> Open Folder and open `myreactapp`
  - We want to do this to make sure the root of our workspace is right.
- Click around the files to get a sense of what is there. The main `src/` files we really need are `App.js` and `index.js`.
- In Terminal, run the program
  - `npm start`
- In a browser, visit `localhost:3000` to see the test app running.
- Also, if you click on the Source Control icon, you'll notice that we've already been setup with a git repository.
  - If you open the Commit pull-down, you'll see we have one commit already "Initialize Project using Create React App"

Git setup

- Back in GitHub, go to your new project page.
  - At the top in the quick setup section, copy the link for your repository to the clipboard



- In VS Code Terminal,
  - `git config --global user.email xxxxxx@users.noreply.github.com`
    - Only if you checked the private email option. Replace xxxxxx with your username.
  - `git remote add origin  <remote url from clipboard>`
  - `git remote -v`
  - `git branch -M main`
  - `git push origin main`
- Go back to GitHub and you should now see your code in the project page.
- Now, you're good to go…

Alternative Git setup

- In VS Code terminal:
  - `git config --global user.email  xxxxx@users.noreply.github.com`
  - `git remote add origin https://github.com/xxxxx/deploy.git`
    - Where "deploy" would be the name of the Github project you created.
  - `git branch -M main`
  - `git pull https://github.com/xxxxx/deploy main`
    - May need to delete your local .gitignore and/or readme to avoid merge conflict
- Hit "refresh" in source control.
  - Don't want to accidentally commit `node_modules`.
- In terminal,
  - `git push -u origin main`
- Commit (stage all changes and commit)
  - Make sure `node_modules` does not show up as changes.
- Sync changes

- Now, you're good to go…

Let's bring over our earlier code

- In the `public` folder, create a subfolder called `css`
- In `App.js`,
  - Delete all existing content except the import and export
  - Copy all the scripts from our html file to `App.js`.
  - Delete the createRoot and render lines (they now will be in the index.js)
  - Rename `Main` to `App`.
  - Add `'import React from 'react';` to the top of the file.
- Copy our html file to `index.html` (overwriting the old one)
  - Delete all script blocks (including those in the head) and link
  - Delete the <p> blocks we used for our javascript exercise
- In `index.js`, wrap the render call with `setInterval` using the same syntax as before (including anonymous function).
- In the terminal, `npm start`.
  - The website should be working as in our earlier exercise.
- Create a `launch.json` using the same procedure from earlier.
  - Verify the new launch configuration works
- Commit your changes in Git.

Better timer…

- Now, using `setInterval` on the `render()` function is very brute-force.
- Let's use React's mechanisms to do it in the proper manner.
- Remove the `setInterval` from `index.js`
- Take a look at the `App` component and think about what we could do here.

- Thoughts?

→ Make changes as discussed.
  - Commit your changes once your code is working.

```
function App() {
    const [color, setColor] = React.useState(JSON.parse(sessionStorage.getItem('color')) || "blue");
    const [count, setCount] = React.useState(JSON.parse(localStorage.getItem('count')) || 0);
    const [time, setTime] = React.useState(new Date().toLocaleTimeString());

    React.useEffect(() => {
        sessionStorage.setItem('color', JSON.stringify(color));
        localStorage.setItem('count', JSON.stringify(count));
        const interval = setInterval(() => setTime(new Date().toLocaleTimeString(), 1000));
        return () => {clearInterval(interval);}
    }, [color, count, time]);

    return (<div>
              <DisplayMessage color={color} />
              <Clock time={time}  />
              <p>{count}</p>
       <button onClick={() => {setColor(toggle(color)); setCount(count+1)}}>
            Click me React
            </button>
            </div>
    );
}
```

- https://www.w3schools.com/jsref/met_win_clearinterval.asp

Separation of Concerns

- Unidirectional data flow: In React, data flows in a single direction from the top of the DOM tree to the bottom.
  - i.e., every component receives data from its parents
- In order to pass data from a child up to the parent, the parent must provide a callback function as one of the props.
  - This also allows siblings to communicate with each other via the parent.
- Looking at our application, we currently have all our state living in the root element.
  - Essentially all state has been "lifted up" to the parent and the child elements simply receive state as props.
  - Does it seem like certain state really belongs to a child?
- Separation of concerns: We want to create elements that have minimal dependencies on other elements. They own what they are responsible for, and they are not responsible for anything more than necessary.

- Let create a self-contained Clock element.

→Update the Clock element by moving the time state into it.
  - Commit your changes once your code is working

```
function App() {
    const [color, setColor] =
React.useState(JSON.parse(sessionStorage.getItem('color')) || "blue");
    const [count, setCount] =
React.useState(JSON.parse(localStorage.getItem('count')) || 0);

    React.useEffect(() => {
        sessionStorage.setItem('color', JSON.stringify(color));
        localStorage.setItem('count', JSON.stringify(count));
    }, [color, count]);

    return (<div>
              <DisplayMessage color={color} />
              <Clock />
              <p>{count}</p>
              <button onClick={() => {
                           setColor(toggle(color));
                           setCount(count+1)}}>
                Click me React
              </button>
            </div>
    );
}
```

```
function Clock() {
   const [time, setTime] = React.useState(
                        new Date().toLocaleTimeString());

   React.useEffect(() => {
      const interval = setInterval(() =>
         setTime(new Date().toLocaleTimeString(), 1000));
      return () => {clearInterval(interval);}
   }, [time]);

  return <p>React Clock: {time} </p>
}
```

Button component

- Right now, our button is in our `Main` component.
- Let's make a separate component `UserActions` that will contain that button
- But, now we have a problem… we want to capture the user's action in one component, but have the result of that action passed to the parent (`count`) and to a sibling component (`color`)
- To do this, we need to maintain state in the parent (`Main`).
- We give `UserActions` two props that are callback functions for changing the state, as well as two props for the values of the state

→Make the new `UserActions` component and adapt the `App` component as appropriate

  - Commit your changes once it is working

Change in function App():

```
return (<div>
                <DisplayMessage color={color} />
                <Clock />
                <p>{count}</p>
                <UserActions setColor={setColor} setCount={setCount} count={count} color={color}/>
             </div>
    );
}

function UserActions(props) {
   return <button onClick={() => {
      props.setColor(toggle(props.color));
      props.setCount(props.count+1)}}>
         Click me React
   </button>
}
```

Count display component

- Our `App.js` looks pretty component-based, except for that `<p>{count}</p>`
- One cool thing about components is that they introduce a separation of concerns – the specifics of how something is displayed/done can be left to another component.
  - This enables easier maintenance and easier changing of code.
- So, let's complete this process by creating one more component to show the count.

→Create a `CounterDisplay` component that accepts the appropriate props

  - Commit your changes once it is working

In function App():

```
 return (
    <div>
      <DisplayMessage color={color} />
      <Clock />
      <CounterDisplay count={count} />
      <UserActions
        setColor={setColor}
        setCount={setCount}
        count={count}
        color={color}
      />
    </div>
  );
}

function CounterDisplay(props) {
  return <p>{props.count}</p>;
}
```

Some Refactoring for Code Readability & Maintainability

- At the moment, our project and code is functional, but has some issues that make it less readable or maintainable.
  - We have a little extra syntax that we can avoid
  - We are doing a couple things in a "clunky" or not-preferred-in-React way.
  - We have defined all our components in a single file under a general `src` folder,
  - We have not included much code documentation

- Let's do some refactoring so that we are writing cleaner React code

- Note: There are a variety of conventions (and competing conventions) for how to do things in React.
  - https://www.loginradius.com/blog/engineering/guest-post/react-best-coding-practices/
  - https://www.jondjones.com/frontend/react/react-tutorials/react-coding-standards-and-practices-to-level-up-your-code/

Simplify some syntax

- So, far we have been using `React.useState` and `React.useEffect`
- We can avoid typing `React.` by directly importing those hooks

```
import React, {useState, useEffect} from "react";
```

- Now we can just use `useState()` instead of `React.useState()` and `useEffect()` instead of `React.useEffect()`

→ Clean up your code appropriately

Helper functions for handling events (and other logic in components)

- At the moment, we have a little bit of clunky code for our `onClick` event handler in `UserActions`
- A general convention of React components is that any logic should be broken out of the render function into a helper function.
  - This keeps the component clean, and separates the concerns inside of the component.
- So, ideally, we'd like our JSX to look like:

```
<button onClick={handleOnClick}>
```

  - Nice and simple element declaration, with the logic details separated out.
- But, we're using a functional component and our `handleOnClick` helper needs to access props and/or state.
  - If we define the function outside the component, then we need to pass it parameters.
  - There is a preferred way to define a helper function inside a functional component – assign an anonymous arrow function to a local variable

```
const handleOnClick = () => {
      props.setColor(toggle(props.color));
      props.setCount(props.count+1)}
```

- This is a preferred approach in React
  - But, if the helper function does not need access to function state, then it may be best to define the helper outside of the function. This way it won't be recreated upon re-render.

→ Clean up your code appropriately

Conditional operator

- Our toggle function is useful, but probably a lot of unneeded syntax in this case.
- Its logic can be handled simply in a single line using the JavaScript conditional (ternary) operator

```
props.color === "blue" ? "red" : "blue"
```

→ Clean up your code appropriately

- Conditional operators are also useful directly in JSX to perform conditional rendering that looks clean

```
return <div>
         loggedIn ? <LogoutButton /> : <LoginButton />
       </div>
```

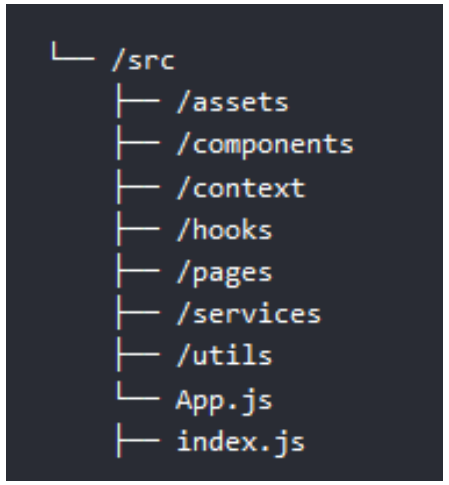
- They can also be assigned to local variables so that the JSX is even simpler.

```
const ButtonComponent = loggedIn ? <LogoutButton /> : <LoginButton />;

return <div>
         {ButtonComponent}
       </div>
```

- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Conditional_Operator
- https://reactjs.org/docs/conditional-rendering.html
- https://www.freecodecamp.org/news/learn-react-conditionals/

File-Folder Organization

- There are a variety of decisions we can make with regards to how we organize our components into distinct files and how we organize our files into folders in our project.
  - This will depend on conventions at a particular company, the size of the project and personal preferences.
  - Some sample conventions
    - https://www.robinwieruch.de/react-folder-structure/
    - https://blog.openreplay.com/react-architecture-patterns-for-your-projects/
- One common approach that we will adopt is the one in this image:



→For now, create `src/components`

- We'll add other folders as we need them.

Component File Structure

- A common React convention is to define each component in its own .js file.
  - The name of the file is generally the name of the component (and must start with a capital letter).
- The goal is to promote component re-use and reinforce separation of concerns
- When a component is in its own file, it needs to be exported at the end of the file, and then it can be imported into any other .js file and used.
- Car.js

```
function Car() {
  return <h2>Hi, I am a Car!</h2>;
}
export default Car;
```

- Another .js file:

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import Car from './Car.js';

const root = ReactDOM.createRoot(
             document.getElementById('root'));
root.render(<Car />);
```

Component File Structure

- Split up your `App.js` file into multiple files, with one component per file.
  - Each component file should be stored in `src/components`.

- Some components made need to start with an import line, such as

```
import React, { useState, useEffect } from "react";
```

- Each component file should end with an appropriate export line.

```
export default Clock;
```

- In `App.js`, import each module using syntax such as

```
import Clock from "./components/Clock";
```

- In `App.js`, move the contents of `App()` function into a separate component called `Main.js`. Then, have the `App` component use `<Main />`

- Hint: To save time, use the Visual Studio Code refactor capability
  - Highlight the component, choose refactor, then move to new file.
  - Then, move the file to the component folder (drag and drop) and say yes to updating the import statement

- https://www.w3schools.com/react/react_components.asp

Project Brainstorming

- Domain-level questions:
  - What is the problem we're solving?
  - What type of site to build?
  - What is the user's goal on the site?
  - What is our goal for the site?

- Design Questions
  - What is our UI style?
  - What User Interactions are needed?
  - What are the building blocks for my UI? (i.e., key components)
  - What are the common look-and-feel elements?

- Technical Questions
  - What information needs to be remembered? (i.e., state)
  - What 3rd party API to use?
  - (many more to answer later)

Project Groups

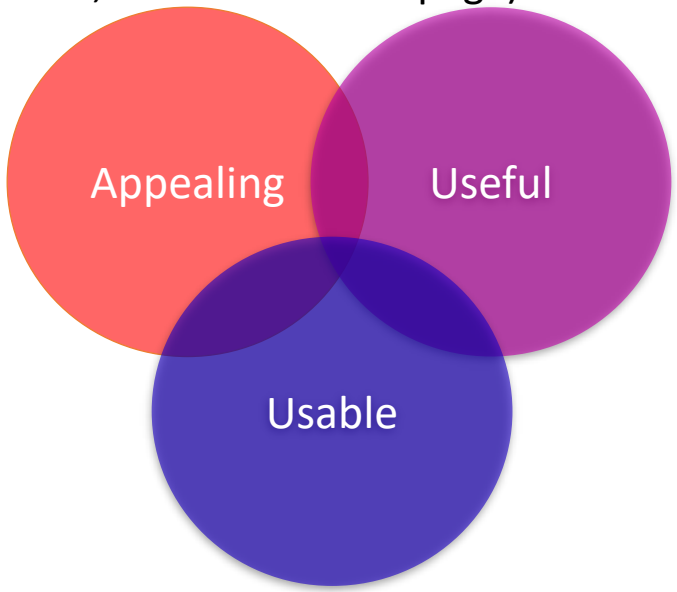
- Group composition
- Project Milestone 1 – Requirement and Design Document – Due Friday, December 2nd,
- Project Milestone 2 – Final project report (implementation, meetings, struggles, member contribution, code improvement, etc) – Due Wednesday, December 7th
- Project Demo and presentation – Friday, December 9th
- Send me the list of your group by Sunday, November 27th
  - Monday, November 28th, we finalize groups and random assign if no group
  - Must MIO me, team leader, team name and team members
- 27 students in the class
  - 6 groups of 4 students
  - 1 group of 3 students

Group

- Name your Group
- Name your Project
- Arrange roles
  - Primary coordinator
    - Monitor tasking and overall health of team
    - Make sure meetings happen and that people attend (google calendar)
    - Ensure development schedule is made and adhered to (google calendar)
  - Primary repository/task tracker maintainer
    - Ensure GIT and task tracking system (e.g., Azure DevOps, Google Sheet) is setup
    - Ensure that people are committing and pushing to branches,
    - Ensure merges go smoothly
    - → e.g., Google Sheet with columns like: Task Name, Task Description, Priority, Estimated Time Required (LOE Level-Of-Effort, 1-2-3), Assign To, Deadline, Misc Notes (depends on…)
  - Primary scribe/communication
    - Take notes on decisions
    - Enter tasks into tasking system
    - Make sure team is communicating
  - In team of 4, task maintainer and entering of tasks can be the 4th role
- All share on design, development, testing and documentation tasks.
- Arrange schedules / availability / contact-info
- What method(s) of remote communication will you use (Teams, Zoom, Discord, email, SMS, phone call)
  - Different people have different preferences and/or are more effective in different methods.

Key initial requirements for Project

- For the group project, your team must develop a transactional web application
  - Note: A transactional web application can take many forms, including applications for reservations, registrations, collaboration, inventory management, e-commerce, etc.
- Must have React front-end
- Must have a user authentication system
  - For teams of 4, must additionally have 2 tiers of users (e.g., regular vs admin, normal vs premium, etc.)
- Must have gated content
  - Website should still have some basic functionality if not logged in (e.g., informative, attractive home page)
  - For teams of 4, different types of users must have access to different content
- Must have be appealing (i.e., a good aesthetic)
  - Decent color scheme, intentional UI design philosophy, images
  - Dark mode/light mode
- Must be usable
  - Well laid out, obvious operations, user-friendly interface
  - Clear input directions, Understandable navigation
- Must be useful
  - Serve the functional goal of the users and site providers
  - No extraneous, distracting content



Technical requirements

- Must include Form text input
- Validation of input
- Handling of errors (fail gracefully)
- The application should not crash
- The user should be able to perform all interactions with the application by visiting a home page and then navigating appropriately within the website.
- The website must have an "About Us" that describes your team and the motivation for the website.
- Must get, display and manipulate data
  - 3rd party API or CRUD backend
- Must use props & state appropriately
- User must interact with data and site must respond to user actions appropriately
  - E.g., Add items to playlist or cart, sort items, filter items, tag items

- **What is the problem we're trying to solve**
- Find friends (10)
  - Make profile and put interests
  - Laugh
  - How to avoid swindlers
  - Questionnaire to determine matches
  - Speed dating for friends
  - Organize group events
- A place to help debug classmates (10)
  - Stackoverflow for students
- Create meal planner (9)
  - Recipes
  - An app that go straight to cooking recipe, skipping all the personal stories (when I was 3 years old cake was my whole life, blah blah...)
- How to learn to properly manage your finances and create a budget (8)
  - Provide lesson
  - Provide budgetting tool with tutorial
  - Help do your taxes
  - Help you get financing towards a goal
  - Common ways of wasting money to avoid
  - Monitor spending
  - Financial based game (buy sell real estate or monopoly based) (4)
- Workout planner/information (7)
- Teach kids coding (6)
- Learn how to do a crypto scam / NFT scam (people are stupid and free money) (6)
  - Joke site
  - Dangers of NFT
  - New JACoin
  - We would solve people falling for scam AND us being poor!
  - Cryptocurrency Trading Platform
- Memorizing terms or programming jargon/ relationships between terminology (5)
  - Memory aid tool
  - Programming Dictionary
  - Flash Cards
  - Memory exercises
  - An app that auto google my problems in codes
- Ukraine Support (5)
  - Info for refugees
  - Help hire Ukrainians (connect companies with people)
  - Help with visas/etc
  - Raise funds
  - Help dispersed people stay in touch / find relatives
- How to start my own business (4)
  - How to find resources to find a business
  - How to look for investors
  - How to build a business proposal
  - Provide templates, processes
  - How to register and laws related to it.
- Cheap Food (inflation!) / Eliminate food waste (4)
  - Connected restaurant with oversupply with hungry people
- Entertain people by augmenting reality (3)
  - Virtual touring using google maps and then connect it to pokemon go to catch more pokemon
- Keep class notes organized (2)
- How to get your life together (2)
  - Guide users on a milestone journey
  - Make a plan to improve skills
  - Choose AI that is nice to you or mean to improve and stop being lazy
  - Technique to inspire or cajol
  - An app that would make my parents not ashamed of me
  - Rent an actor who looks like you
- Quebec healthcare / Deal with modern age anxiety / How to stop feeling empty inside / Reduce stress (2)
  - Better logistics
  - Mental health
  - Get resources more quickly
  - Access to family doctors
  - Get people in touch with students in psychology (matchmaking)
- Bug Tracker (2)
  - Like Jira
- To Do List (2)

Routing

Client-Side Routing

- When creating a typical Single Page Application, the "single page" refers specifically to the one-and-only html page that the application will load from the server
- After that html page is loaded, all changes to the application interface are made on the client side using React.
- But, what if we want our application to have different "pages" in terms of the UI itself.
  - E.g., an About page, a Contact page, a Shopping page, etc.
- In other words, from a UI design perspective (rather than an `.html` perspective), we may have different parts of our web page that the user can navigate between
  - Think of navigating among pages in a navbar, or following links within a page to another page.
- Each of these different pages essentially will correspond to a different set of components to render.
- To support navigation between these pages, we need some way of specifying "links" to different "pages" and some way to enable React to "switch" among the pages based on the user's actions and render the appropriate components when a page is visited.
- We can use the **React Router** to provide this capability.

React Router

- React Router is a dynamic routing library for React
  - Composed of: `react-router-dom`
- Dynamic routing means that the routing takes place as your app is rendering.
- Dynamic client-side routing allows us to build a Single-Page Application with navigation without the page "refreshing" as the user navigates.
  - Fast and seamless
  - Allows use of browser "back button", "refresh page", etc.
  - Allows changing of browser url

- https://www.w3schools.com/react/react_router.asp
- https://medium.com/@marcellamaki/a-brief-overview-of-react-router-and-client-side-routing-70eb420e8cde

Basics of Routing

- To perform routing, we need to do several things

1. Define what router we are using
2. Define what routes are available in our app
   - What url paths will be used, and what component will be shown at each url.
3. Navigate to a route
   1. Define links within our UI that enable the user to navigate a route.
   2. Directly navigate in our code to a route

`BrowserRouter` component

- The React Router provides a `<BrowserRouter>` component that typically serves as a component that wraps our `<App>` component in `index.js`
  - React Router actually has several other types of routers, but `BrowserRouter` is the recommended one for web applications.
- This allows all the defined routes to be used throughout the tree of components.

```
<BrowserRouter>
  <App />
</BrowserRouter>
```

<Route> Component

- To define a route, we need to specify a url path and a component that will be rendered for that path
- The syntax to define a route is:

```
<Route path="/about" element={<About />} />
```

- Note: Online resources may show older syntax approaches. They don't work anymore…

<Routes> Component

- Our application will have multiple routes, and these are defined within a <Routes> component.
- The syntax for defining a set of routes is:

```
<Routes>
      <Route path="/" element={<Home />} />
      <Route path="/about" element={<About />} />
      <Route path="/contact" element={<Contact />} />
</Routes>
```

Exercise 3a

- Install React Router
  - `npm i react-router-dom`
- Create folder `src/pages`
  - In the folder, create the following files, each with an appropriate function component
    - `Home.js`
      - Should use our `Main` component
    - `Contact.js`
      - Just an `<h1>` with a contact us message
    - `About.js`
      - Just an `<h1>` with an about us message
- In `index.js`,

```
import { BrowserRouter } from  "react-router-dom";
```

  → Wrap the `<App>` component with a `<BrowserRouter>` component

    - Note: `<React.StrictMode>` can remain the highest-level component.

- In `App.js`,

```
import { Route, Routes} from  "react-router-dom";
```

  → Change the `App()` function to return a set of routes as per the previous slide. (Don't forget to import each of the components).

- Run your program. Manually visit the three different URLs and see what happens.
  - View page source on each one… notice that it doesn't change since we are always using the same html file.

Web Programming II

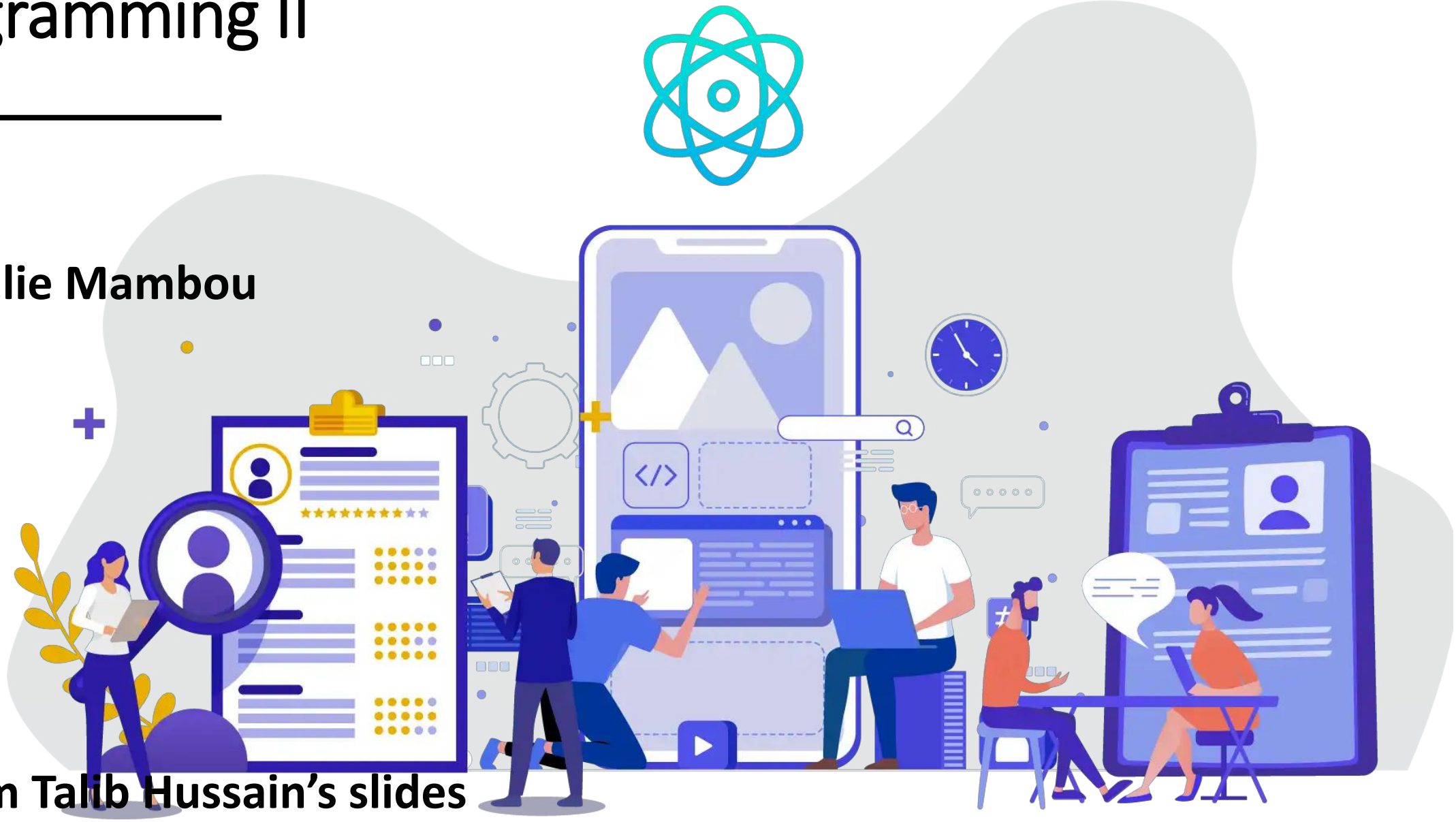
420-JE5-AB

Instructor: Elie Mambou

Day 5:

Routing/UI

反应



Adapted from Talib Hussain's slides

Routing

Client-Side Routing

- When creating a typical Single Page Application, the "single page" refers specifically to the one-and-only html page that the application will load from the server
- After that html page is loaded, all changes to the application interface are made on the client side using React.
- But, what if we want our application to have different "pages" in terms of the UI itself.
  - E.g., an About page, a Contact page, a Shopping page, etc.
- In other words, from a UI design perspective (rather than an `.html` perspective), we may have different parts of our web page that the user can navigate between
  - Think of navigating among pages in a navbar, or following links within a page to another page.
- Each of these different pages essentially will correspond to a different set of components to render.
- To support navigation between these pages, we need some way of specifying "links" to different "pages" and some way to enable React to "switch" among the pages based on the user's actions and render the appropriate components when a page is visited.
- We can use the **React Router** to provide this capability.

React Router

- React Router is a dynamic routing library for React
  - Composed of: `react-router` and `react-router-dom`
- Dynamic routing means that the routing takes place as your app is rendering.
- Dynamic client-side routing allows us to build a Single-Page Application with navigation without the page "refreshing" as the user navigates.
  - Fast and seamless
  - Allows use of browser "back button", "refresh page", etc.
  - Allows changing of browser url

- https://www.w3schools.com/react/react_router.asp
- https://medium.com/@marcellamaki/a-brief-overview-of-react-router-and-client-side-routing-70eb420e8cde

Basics of Routing

- To perform router, we need to do several things

1. Define what router we are using
2. Define what routes are available in our app
   - What url paths will be used, and what component will be shown at each url.
3. Navigate to a route
   1. Define links within our UI that enable the user to navigate a route.
   2. Directly navigate in our code to a route

`<BrowserRouter>` component

- The React Router provides a `<BrowserRouter>` component that typically serves as a component that wraps our `<App>` component in `index.js`
  - React Router actually has several other types of routers, but `BrowserRouter` is the recommended one for web applications.
- This allows all the defined routes to be used throughout the tree of components.

```
<BrowserRouter>
  <App />
</BrowserRouter>
```

<Route> Component

- To define a route, we need to specify a url path and a component that will be rendered for that path
- The syntax to define a route is:

```
<Route path="/about" element={<About />} />
```

- Note: Online resources may show older syntax approaches. They don't work anymore…

<Routes> Component

- Our application will have multiple routes, and these are defined within a <Routes> component.
- The syntax for defining a set of routes is:

```
<Routes>
      <Route path="/" element={<Home />} />
      <Route path="/about" element={<About />} />
      <Route path="/contact" element={<Contact />} />
</Routes>
```

Exercise 3a

- Install React Router
  - `npm i react-router-dom`
- Create folder `src/pages`
  - In the folder, create the following files, each with an appropriate function component
    - `Home.js`
      - Should use our `Main` component
    - `Contact.js`
      - Component that returns an `<h1>` with a contact us message
    - `About.js`
      - Component that returns an `<h1>` with an about us message
- In `index.js`,

```
import { BrowserRouter } from  "react-router-dom";
```

  → Wrap the `<App>` component with a `<BrowserRouter>` component
    - Note: `<React.StrictMode>` can remain the highest-level component.
- In `App.js`,

```
import { Route, Routes} from  "react-router-dom";
```

  → Change the `App()` function to return a set of routes as per the previous slide.
- Run your program. Manually visit the three different URLs and see what happens.
  - View page source on each one… notice that it doesn't change since we are always using the same html file.

<Link> Component

- A <Link> component functions similarly to an <a> href tag
- The syntax for a link is:

```
<Link to="/">Visit Home</Link>
```

- Just like an <a> tag, this will create a clickable link on the UI.
- When the user clicks the link, the application will render the appropriate component for the indicated path.

Navigation header

- Let's create a simple header that will let the user navigate the web application.

- Create a component `<Header>`
  - In the header, create a list of `<Link>`s for our 3 routes.

- Insert the `<Header>` component into our App right before the `<Routes>` component.

- Don't forget to import `Link` so you can use it.
  - `import { Link } from "react-router-dom";`

```
import React from "react";
import { Link } from "react-router-dom";

function Header() {
  return (
    <div>
      <Link to="/">Home</Link>
      <Link to="/about">About</Link>
      <Link to="/contact">Contact</Link>
    </div>
  );
}

export default Header;
```

In App.js

```
    <Header />
    <Routes>
          <Route path="/" element={<Home />} />
          <Route path="/about" element={<About />} />
          <Route path="/contact" element={<Contact />}
    />
    </Routes>
```

Nested Routes

- In our previous examples, we defined each route with a path that started with /
- Rather than this, we can nest the actual `<Route>` components and React will figure out the appropriate path.
- In addition, when using this approach, all pages in the nested routes will share a common parent component, which often provides a shared layout.

```
<Routes>
      <Route path="/" element={<MainLayout />} >
        <Route index element={<Home />} />
        <Route path="about" element={<About />} />
        <Route path="contact" element={<Contact />} />
      </Route>
</Routes>
```

- In the nested routes above, a common layout is defined by the element `<MainLayout />` in the parent Route.
  - This `<MainLayout />` component will contain a special `<Outlet>` tag that indicates where the child Route's element will be inserted.
- `<Home />` will be inserted into the layout for the path '/'. The route for this has a special attribute `index` instead of a `path`.
- `<About />` will be inserted for '`/about`' and `<Contact />` for '`/contact`'

<Outlet>

- The "parent" route element (i.e., <MainLayout>) that has nesting will contain any common components you want all "child" route elements in the nested set of Routes to share, like a header and/or footer.
- In addition, it should contain a special component named <Outlet />
- <Outlet> lets parents render their child route elements appropriately.
  - This allows nested UI to show up when child routes are rendered.
- Specifically, when the layout component is rendered, the <Outlet /> component will render the next <Route> match
  - If the parent route matched exactly, it will render a child index route or nothing if there is no index route
  - If the route matches a child route, then that child route will be rendered.
  - i.e., <About> if the endpoint is '/about', or <Home> is the endpoint is '/', etc.

Shared Layout for Routes

- Create a component in the `components` folder called `Footer.js`
  - This component should display a copyright message.
- Create a folder `src/layouts`
- Create a file `MainLayout.js`.

```
import React from "react";
import {Outlet} from "react-router-dom";
import Header from '../components/Header';
import Footer from '../components/Footer';

function MainLayout() {
   return <div>
            <Header />
            <Outlet />
            <Footer />

          </div>
}
```

- https://reactrouter.com/docs/en/v6/components/outlet
- After you get it working, commit your changes to git.

No Match Route

- If the user enters a URL that does not match a valid route, it is good practice to handle this case.
- We can use the special wildcard * to match any path that hasn't been matched yet.
  - This wildcard route should go at the end of our list of routes
- Put the following at the end of our list of Routes

```
<Route path="*" element={<p>Invalid URL</p>} />
```

- If you put it in the nested section, then the header and footer will still show up in addition to the message.
- If you put it after the nested section, then only the message will show up.

<Navigate>

- Any time that you want to force navigation, you can render a <Navigate> component.
- When a <Navigate> renders, it will navigate using its "to" prop.
  - i.e., This can be used to redirect the user to a different page.
- The route that is specified must have been declared before the setting up the redirect.
  - i.e., order of specifying <Route>s matters.

→ Change the No Match Route to redirect the user to '/'.

- Don't forget to import Navigate from react-router-dom
- https://stackabuse.com/redirects-in-react-router/

`useNavigate()`

- React Router also provides a hook that can be used in a functional component to redirect the user to a particular URL.
- This is useful, for example, for including a button that the user can click to cause a navigation to happen.
- Create a new file `components/HomeButton.js`
  - `<HomeButton>` will take the user to Home ('/') when clicked using `useNavigate()`.
- Update `Header.js` to include a `<HomeButton>` component.
  - Place the Home button in its own row (i.e., own `<div>`) at the top of the screen and aligned to the right side of the screen.
- Note: We can use flexbox styling to align to the right.
  - https://css-tricks.com/snippets/css/a-guide-to-flexbox/

```
import { useNavigate } from "react-router-dom";

function HomeButton() {
  const navigate = useNavigate();

  const handleSubmit = (event) => {
    try {
      navigate("/", { replace: true });
    } catch (error) {
      console.log(error);
    }
  };

  return (
    <div style={{ flexDirection: "row", direction: "rtl" }}>
      <button onClick={handleSubmit}>Home</button>
    </div>
  );
}

export default HomeButton;
```

```
import React from "react";
import { Link } from "react-router-dom";
import HomeButton from "./HomeButton";

function Header() {
  return (
    <div>
      <div>
        <HomeButton />
      </div>
      <div>
        <Link to="/">Home</Link>
        <Link to="/about">About</Link>
        <Link to="/contact">Contact</Link>
      </div>
    </div>
  );
}

export default Header;
```

<NavLink> : Active Link

- React Router comes with a special link component called <NavLink> that knows whether or not it is "active"
- This is useful when you want to have the navigation bar show which is currently selected.
- Essentially, NavLink provides a value isActive that can be used to determine what styling to apply
  - E.g., enables you to apply distinctive styling to the rendered link when it matches the current URL.
- In-line style approach:

```
<NavLink
        style={({ isActive }) =>
          isActive ? { color: "orange" } : { color: "blue" }
        }
        to="/"  >  Home
      </NavLink>
```

- Separate css approach:
  - We determine an appropriate className and then use a component-specific .css file to apply the styling

```
<NavLink
        className={({ isActive }) =>
          isActive ? "link-active" : "link-inactive"        }
        to="/about" > About
</NavLink>
```

- https://reactrouter.com/docs/en/v6/components/nav-link
- https://www.kindacode.com/article/react-router-how-to-highlight-active-link/

Component-specific CSS files

- One convention of React is that we can define a `.css` file for a particular component and have those styles applied to that component when it renders.
  - By convention, we name the css file based on the component name and save it in the same location as the component.
  - The css file must still be imported.
- In `components` folder, create a file called `Header.css`
- In the file, define class styles for active and inactive links

```
/* Specific styles for non-active links */
.link-inactive {
  color: red;
}

/* Specific styles for active links */
.link-active {
  color: black;
  background: orange;
}
```

- In `Header.js`, import the css file
  - `import "./Header.css";`
- Apply the className approach to NavLink styling from the previous slide
- Run your program

Approaches to CSS

- In React, there are a variety of approaches to adding css styling to components.
- As we've seen, we can use inline styles (using the `style` attribute) as well as component css files.
  - Inline styles are generally discouraged.
  - For inline styles, camel-case attributes must be used (e.g., backgroundColor, not background-color).
  - You can specify the style in an object variable to make it easier to use in the JSX.
- We can also create a typical external stylesheet that can be imported by any components.
  - E.g., `src/styles.css`
  - `import ('./styles.css')`
  - or `import('../styles.cs')` for a component in the `components` folder.
- If two css files are imported in a component and there is a conflicting style, then the last one loaded is used.
  - So, load the main stylesheet first and then the local one.

```
import "../styles.css";
import "./Header.css";
```

- There are also other ways to apply styling in React that are generally more complicated, but offer interesting capabilities.
- Recommend you keep it simple for this course… stick with a main stylesheet for general consistency, a component stylesheet for local variations, and inline styles for one-off or interesting cases.
- https://www.w3schools.com/react/react_css.asp
- https://reactjs.org/docs/faq-styling.html
- https://www.sitepoint.com/react-components-styling-options/
- https://css-tricks.com/a-thorough-analysis-of-css-in-js/
- https://programmingwithmosh.com/react/css-modules-react/

Link with a button

- You can wrap a button (or any other Component) with a `Link` or `NavLink`.
  - Clicking on the button will then cause the link to be triggered.
- Let's style a button and place it in our "home" `NavLink`

```
const buttonStyle = {
   backgroundColor: "green",
   border: "none",
   color: "white",
   padding: "15px 32px",
   textAlign: "center",
   textDecoration: "none",
   display: "inline-block",
   fontSize: "16px",
 };

...
     <NavLink>
        <button style={buttonStyle}>
          <p>Home</p>
        </button>
     </NavLink>
```

https://stackblitz.com/github/remix-run/react-router/tree/main/examples/custom-link?file=src%2FApp.tsx

https://reach.tech/router/api/useMatch

Active/Inactive Button

- But, making that button change based on being active is a bit challenging.
- Need to use a hook called `useMatch` which evaluates to true when the current location of the browser matches to "to" location of the `NavLink`.
- The following makes a value 'match' available that will be true when the button matches the current URL and false otherwise.
  - This match value can then be used in a conditional in JSX to determine styling.

```
function NavButton(props) {
  let resolved = useResolvedPath(props.to);
  let match = useMatch({ path: resolved.pathname, end: true });
```

  - The element returned by `NavButton` should be a button wrapped in a `NavLink`.
    - Use the value of match to determine the style of the button
  - React will update the value of `match` (via the hook) everytime the page is changed.
  - NavButton should accept a props value "to" representing the URL and a label representing the label for the button

→Create a new component `components/NavButton.js` that changes button style depending on whether it matches the current URL or not.

```
function NavButton(props) {
  let resolved = useResolvedPath(props.to);
  let match = useMatch({ path: resolved.pathname, end: true });

  const buttonStyle = {
    backgroundColor: "green",
    border: "none",
    color: "white",
    padding: "15px 32px",
    textAlign: "center",
    textDecoration: "none",
    display: "inline-block",
    fontSize: "16px",
  };
 const activeButtonStyle = {
    backgroundColor: "red",
    border: "none",
    color: "black",
    padding: "15px 32px",
    textAlign: "center",
    textDecoration: "none",
    display: "inline-block",
    fontSize: "16px",
  };

  return (
    <NavLink to={props.to}>
      <button style={match ?
          activeButtonStyle : buttonStyle}>
        <p>{props.label}</p>
      </button>
    </NavLink>
  );
}
```

In Header.js:

```
<NavButton to="/" label="Home" />
```

Make a Pretty Site with Navigation Bar

- Based on what you've learned so far and your knowledge of css, spend some time styling your site to make it look interesting.
- Make use of a general stylesheet as well as a couple component style sheets.
- In particular, style the `Header` navigation buttons and `DisplayMessage` locally.

→ 15 minutes

Submit…

Save in Teams files: Documents/General/Class Exercise/Exercise3 folder

Submit at least two screenshots of your styled web page showing different aspects of the styling you created.

styled1.<*yourname*>

styled2.<*yourname*>

etc.

Design

Design Patterns / Approaches

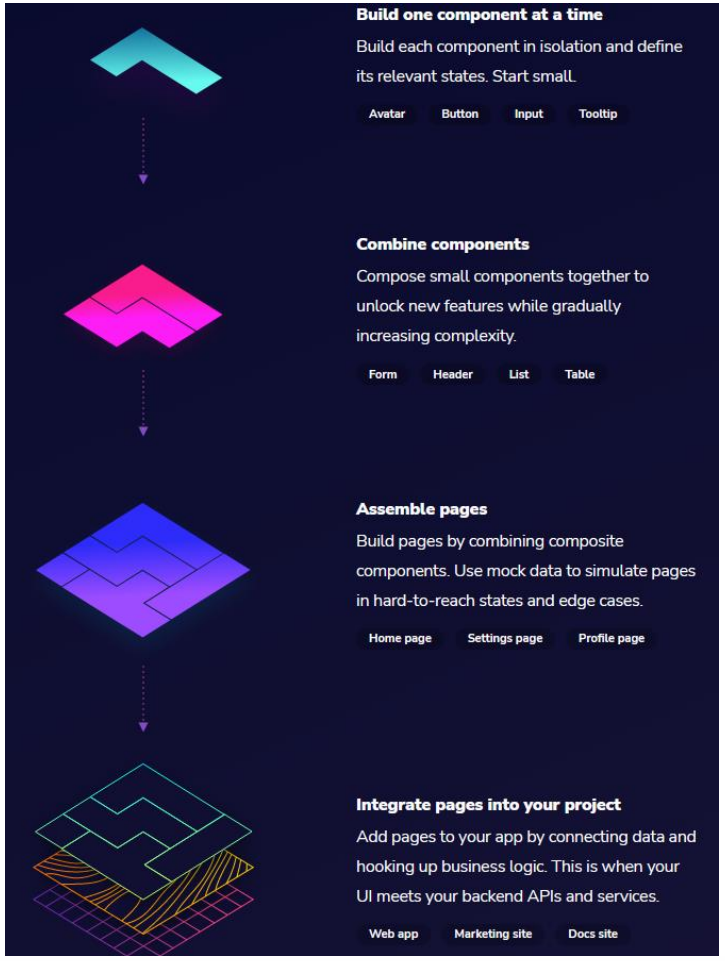
- When developing a React front-end, there are a variety of approaches you can take to the design of the User Interface (UI).
- In general, the component-based nature of React lends itself to a design that identifies the UI components that are needed and how they are arranged on the screen.
- Additionally, a good design provides good separation between logic focused on the view and logic focused on data and state.
- React design patterns and practices range from higher-level UI principles to lower-level component design
  - The former focuses on usability, flexibility and aesthetic
  - The latter focuses on cleanliness of code, maintainability, reusability and separation of concerns.

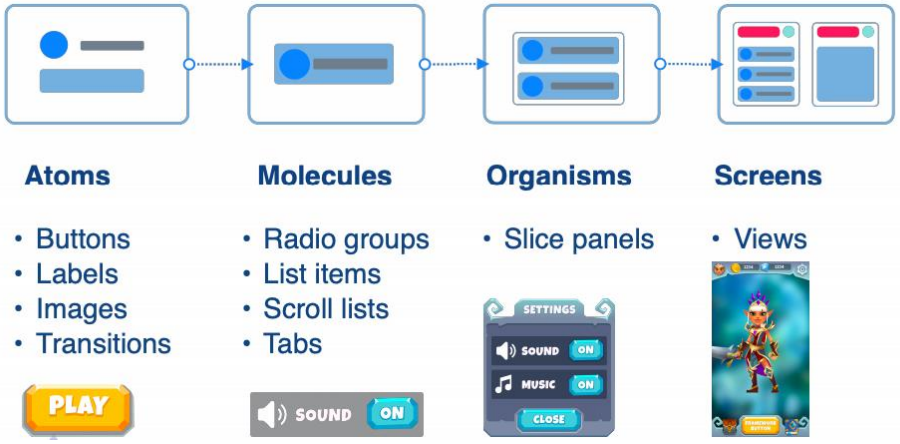
High-Level Design

- Component-based UI Design
- React-Bootstrap
- Material UI
- Scalable Vector Graphics (SVG)

UI Design

- Component-based approach to UI design
  - Build individual components for simple operations
  - Combine them to form components that provide functionality
  - Arrange components into UI elements that serve a key purpose for your page
    - Components get linked to data
  - Arrange components to form pages
    - Think "Navbar", "Hero Section", "Footer"
  - Ensure some consistency among pages
- Read about it:
  - https://www.componentdriven.org/
  - https://medium.com/swlh/the-importance-of-componen based-ui-design-666e5dfc7c1a
- Tools to support it
  - https://storybook.js.org/





Bootstrap

- Each Bootstrap component has been rebuilt as a React component
- https://react-bootstrap.github.io/
- Some stylesheet **is required** to use these compo`npm install react-bootstrap bootstrap`
- nents.
- In `index.js`, put the following line:

```
{/* The following line can be included in your src/index.js or App.js file*/}
import 'bootstrap/dist/css/bootstrap.min.css';
```

- Change `HomeButton` to use a Bootstrap button

```
import Button from 'react-bootstrap/Button';

<Button variant="primary" onClick={handleSubmit}>
```

- Once you've done this, explore some variations on the Button by visiting https://react-bootstrap.github.io/components/buttons/
  - Add more buttons to your page, using different variations.

Submit…

Save in Teams files: Documents/General/Class Exercise/Exercise3 folder

Submit a screenshot of your web page with multiple Bootstrap buttons.

bootstrap.<*yourname*>

Material

- https://material.io/design
- Material is a design system created by Google to help teams build high-quality digital experiences for Android, iOS, Flutter, and the web.
- Read about it here:
  - https://material.io/design/introduction

Material UI for React

- https://mui.com/
- https://github.com/mui/material-ui
- https://mui.com/material-ui/getting-started/usage/
- MUI components work in isolation.
  - They are self-supporting, and will only inject the styles they need to display.
  - They don't rely on any global style-sheets
- `npm install @mui/material @emotion/react @emotion/styled`

- Change our `HomeButton` to use the following:

```
import Button from "@mui/material/Button";

<Button variant="contained" onClick={handleSubmit}>
```

- Once you've done this, explore some variations on the Button by visiting https://mui.com/material-ui/react-button/

Other Material UI Components

- Box,
- Card,
- CardContent,
- Container,
- Grid,
- Typography,
- useTheme,

Submit…

Save in Teams files: Documents/General/Class Exercise/Exercise3 folder

Submit a screenshot of your web page with multiple Material buttons.

material.<*yourname*>

SVG

- Scalable Vector Graphics (SVG) are an XML-based markup language for describing two-dimensional based vector graphics.
  - https://developer.mozilla.org/en-US/docs/Web/SVG
- One of the most interesting techniques we can apply in the browser to draw icons and graphs is Scalable Vector Graphics (SVG).
- SVG is great, because it is a declarative way of describing vectors and it fits perfectly with the purposes of React.
- From a React point of view, it does not make any difference if we output a div or an SVG element from the render method, and this is what makes it so powerful.
- A common way to create SVGs in a web app with React is to wrap vectors into a React component and use the props to define their dynamic values.

```
const Circle = ({ x, y, radius, fill }) => (
  <svg>
    <circle cx={x} cy={y} r={radius} fill={fill} />
  </svg>
)
Circle.defaultProps = {
  fill: 'red',
}
<Circle x={20} y={20} radius={20} fill="blue" />

const RedCircle = ({ x, y, radius }) => (
  <Circle x={x} y={y} radius={radius} fill="red" />
)
```

- https://datalanguage.com/blog/graphical-uis-with-svg-and-react-part-1-declarative-graphics

SVG Links

- https://medium.com/tinyso/how-to-create-an-animated-svg-circular-progress-component-in-react-5123c7d24391
- https://blog.logrocket.com/how-to-build-an-svg-circular-progress-component-using-react-and-react-hooks/
- https://www.smashingmagazine.com/2015/12/generating-svg-with-react/

- Explore these and other links to see what's possible using SVG

Web Programming II

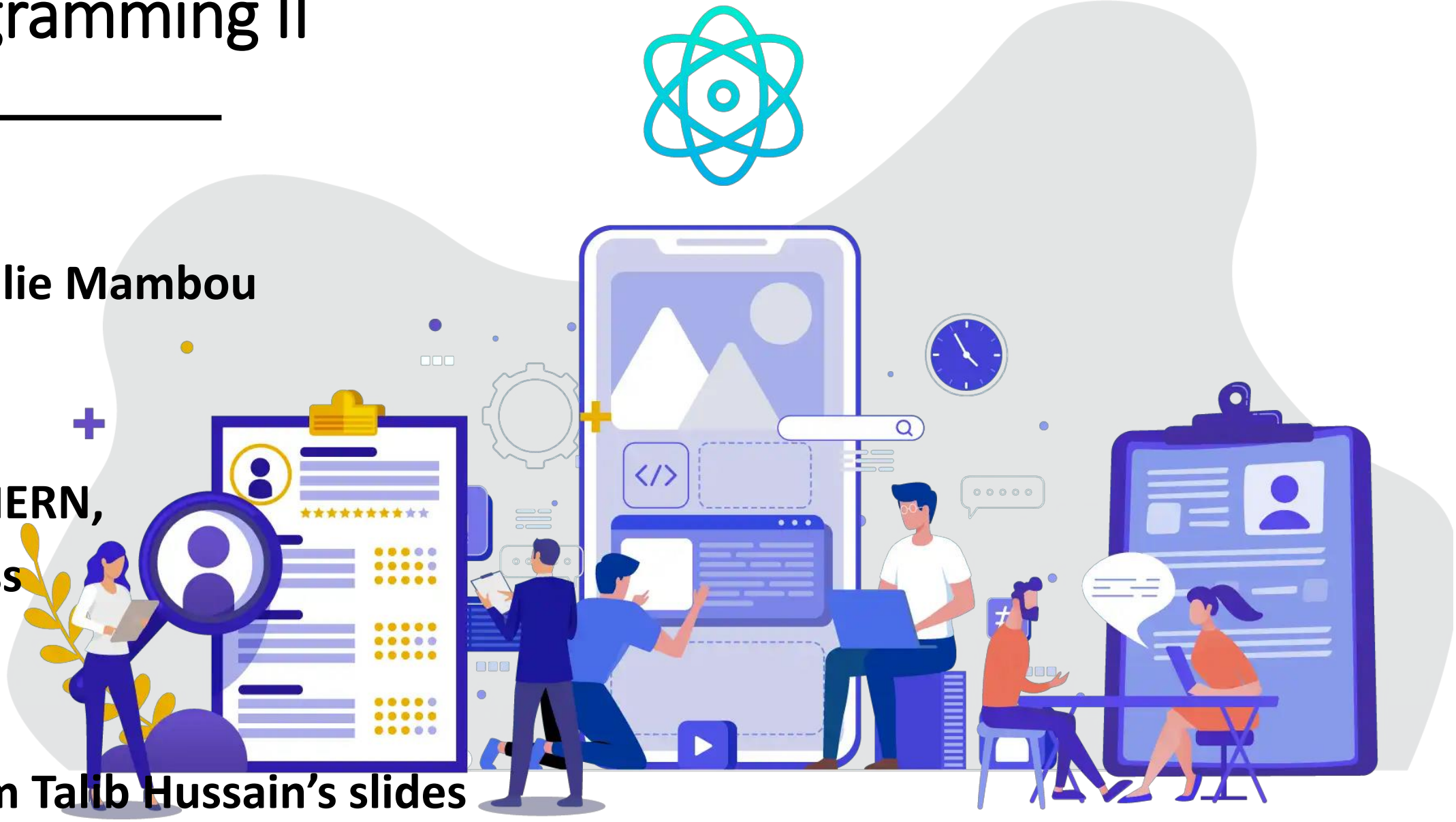
420-JE5-AB

Instructor: Elie Mambou

Day 6:

Full-Stack MERN,

Gated Access



Adapted from Talib Hussain's slides

Agenda

- Full Stack MERN!
  - NoSQL
  - MongoDb
  - Mongoose Schema
  - Storing Secrets
  - Node with Express
  - AJAX / Fetch
  - HTTP Methods (including PUT, PATCH, DELETE)
  - CRUD
- User Authentication
  - Hashing
- Group Project Planning
- In-Class exercise 2: Gated Access (Worth 5%)
  - Forms – Uncontrolled vs Controlled
  - Prop-drilling vs Context+Provider
  - Conditional Rendering

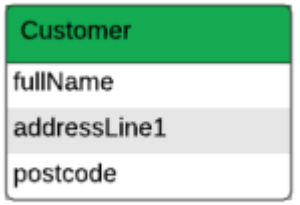
Full-Stack!

Let's build a simple back-end using Node, Express and MongoDB.

When connected to our React front-end, we'll have built a (simple) full-stack MERN application

NoSQL vs SQL

- A SQL database, like MySQL, is a **relational** database. This means that data is stored in in a structured format, usually called **tables**, containing specific pieces and types of data. Relationships are explicitly made between these tables. That is why you have foreign keys, joins, etc. along with the idea of one-to-one, one-to-many, many-to-one and many-to-many relationships.
  - A SQL database works best when the data they contain doesn't change very often, and when accuracy is crucial.
- By contrast, a NoSQL database is **non-relational**. Data is stored in "unstructured", non-tabular form. For instance, they might be stored in data structures called **documents**. A document can be highly detailed while containing a range of different types of information in different formats. This lets various types of information be organized side-by-side
  - A NoSQL database is often used when large quantities of complex and diverse data need to be organized, and/or when data changes frequently. They are also useful for developing applications where the data needs change quickly.
  - Unstructured data = not arranged according to a pre-set model or schema.
  - In many cases, a NoSQL document is specified in terms of name-value pairs and can be represented using JSON format.
- https://www.mongodb.com/databases/non-relational



NoSQL

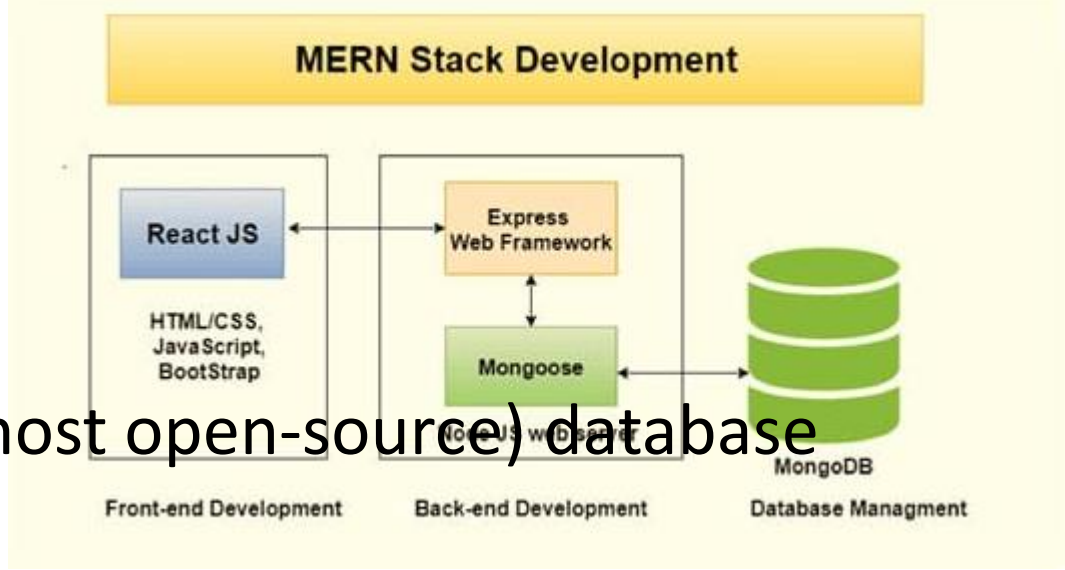
- In NoSQL, each "piece" of data is stored in a single **document**.
  - This may have multiple fields of information,
  - A document roughly corresponds to a row of information in a SQL database
  - The fields roughly correspond to the columns in a SQL database
- Multiple documents with the same type can be put together in a **collection**
  - This can roughly be thought of as corresponding to a table in MySQL.
- A non-relational database can be massive and can grow exponentially in principle. Since new data in different formats can be added at any time, it is highly flexible to use.
- 80 to 90% of data generated and collected by organizations is unstructured, and its volumes are growing rapidly.

- https://www.mongodb.com/unstructured-data
- https://rahmanfadhil.com/express-rest-api/

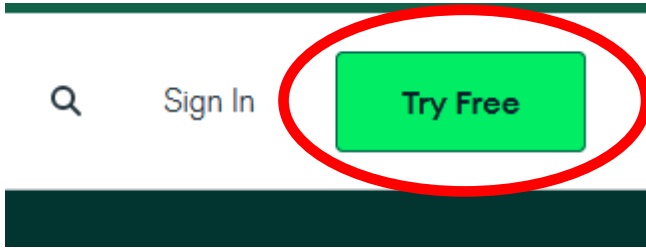
Mongoose

- Mongoose is a module that lets us use MongoDB with node.js.
- It provides a schema-based approach to modeling data and is called an ODM (Object Document Mapper)
  - Maps between an object model and a document database
  - Like an ORM (Object Relational Mapper) but for NoSQL databases
- Essentially, Mongoose enforces a particular document data structure via the application layer
  - The underlying database is still unstructured, but only structured data will be entered by the application.
- This adds some restrictions to using MongoDB in our application since we will only be able to store and retrieve objects that meet our schema
- But, it will help us avoid errors, such as invalid data, missing fields, etc.
- https://mongoosejs.com/
- https://www.freecodecamp.org/news/introduction-to-mongoose-for-mongodb-d2a7aa593c57/
- https://www.mongodb.com/developer/languages/javascript/mongoose-versus-nodejs-driver/

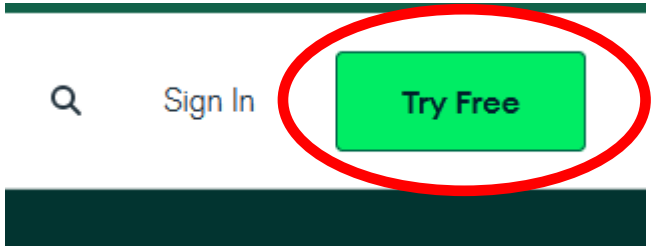
MongoDB & MERN Stack

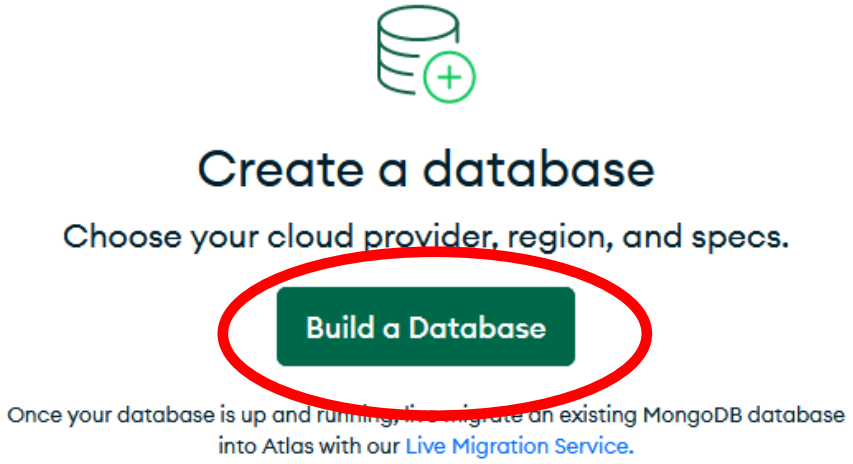


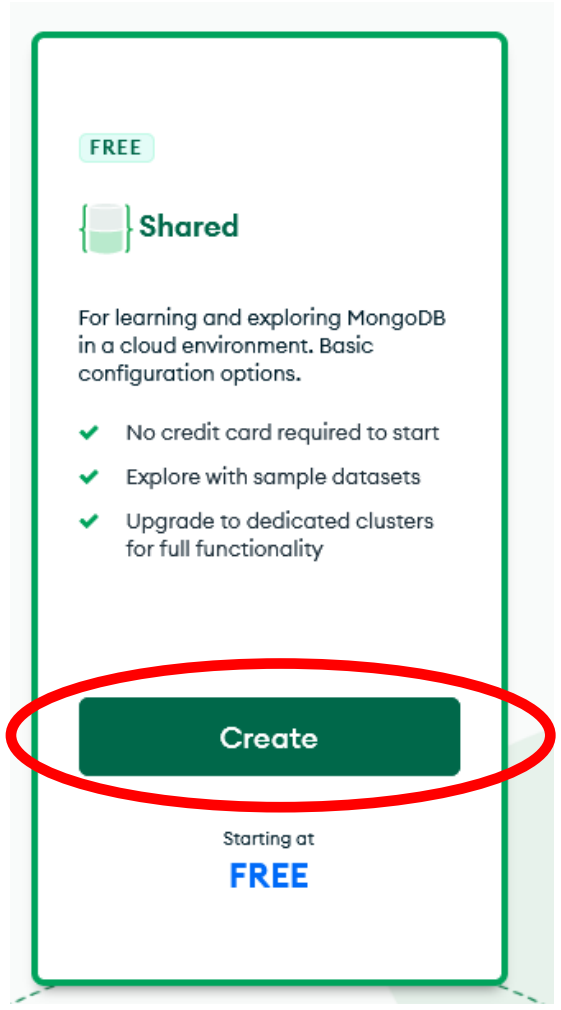
- MongoDB is a "source-available" (i.e., almost open-source) database program that works across platforms.
- MongoDB Atlas is an online Database-as-a-Service that is a key modern technology for enabling a company to deploy into the cloud.
- Together with an open-source back-end (Node.js and Express.js) and an open-source front-end (React), it offers an open-source full-stack capability that can help companies (and students) quickly develop new solutions with no costs.
- The MERN stack (MongoDb, Express, React, Node) is

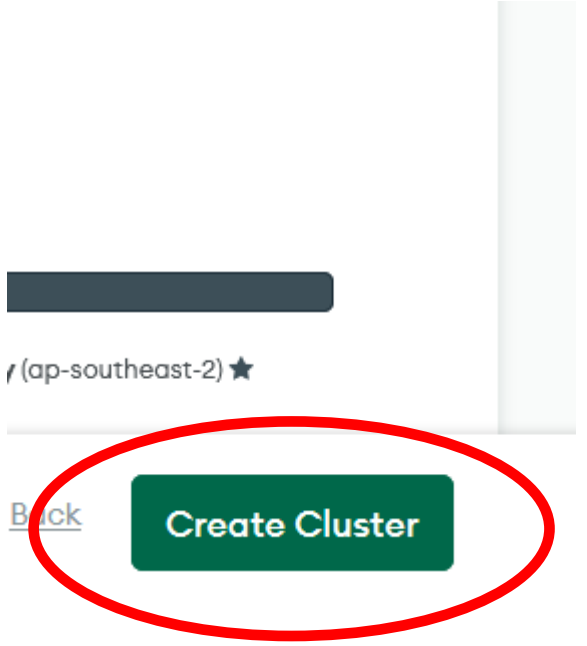


- So, let's get started with MongoDB
- Go to www.mongodb.com and sign up (click on "Try Free")
- Once signed up and signed in, create a free tier cluster in 3 steps
  - Click "Build a Database", then select the free tier and click "Create", and finally select "Create Cluster"

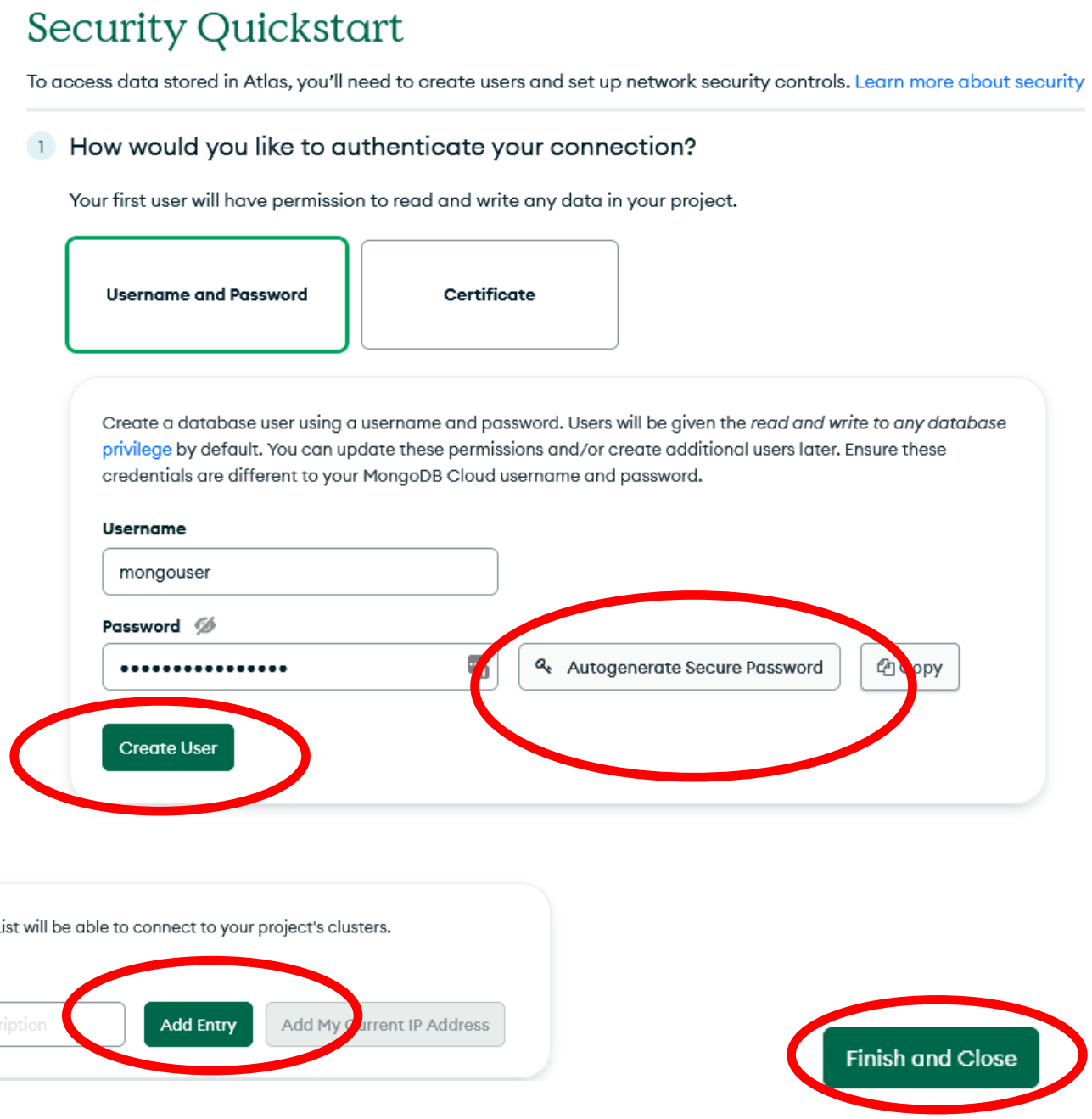


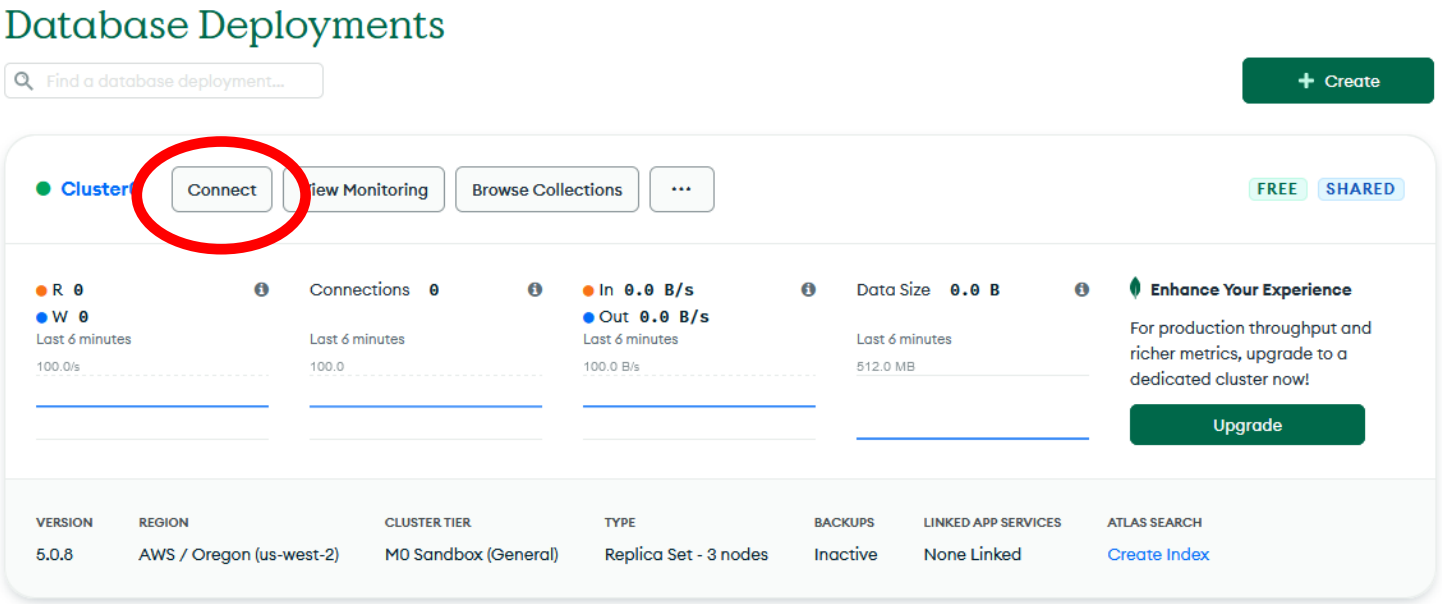


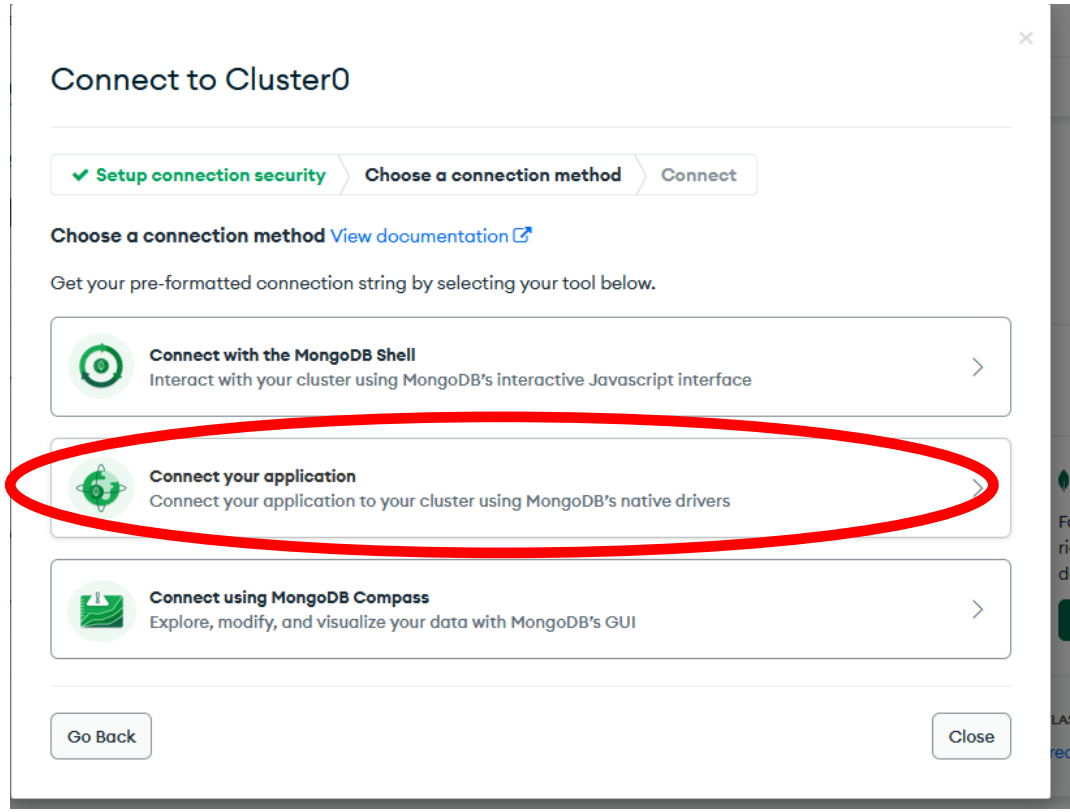




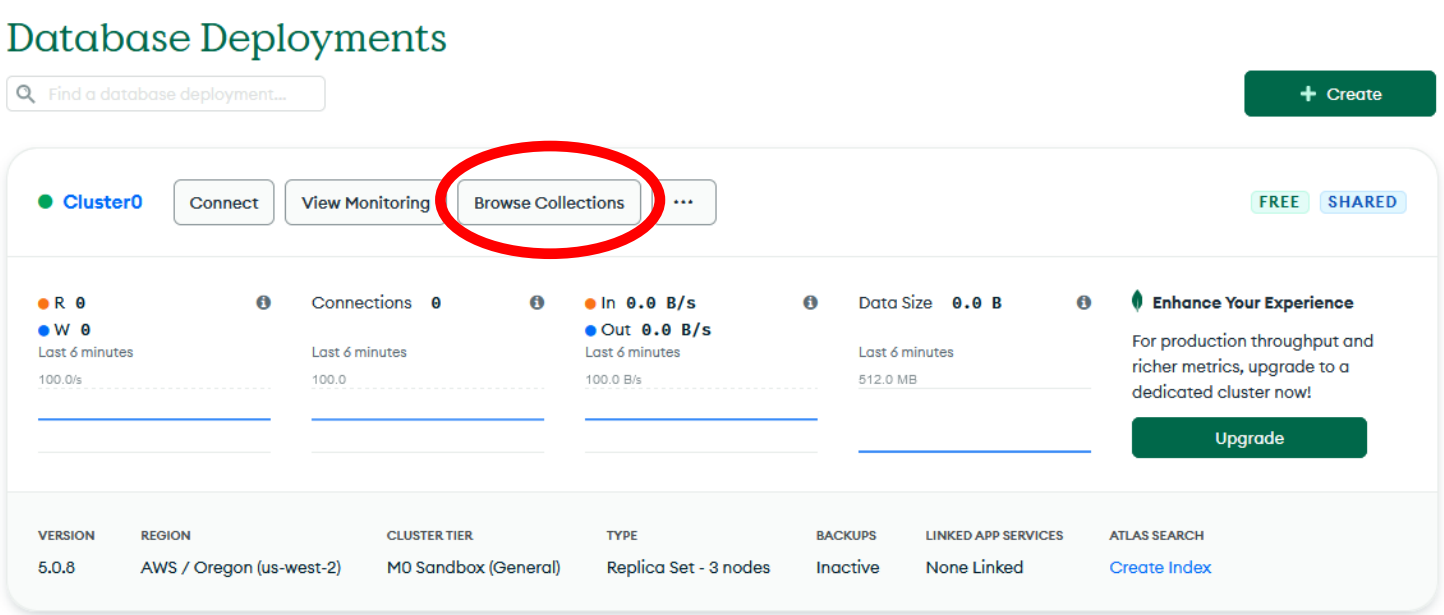
- Create a User
  - Enter username (e.g., mongouser)
    - This should not be the same as your MongoDb username. This is for accessing the new database remotely.
  - Enter or generate a secure password
    - Don't forget to copy it and save it (E.g., save it in a txt file or temporarily past it into an editor). We'll need it soon.
  - Click "Create User"
- Enter 0.0.0.0 in the IP address section and click "Add Entry"
- Click "Finish and Close"

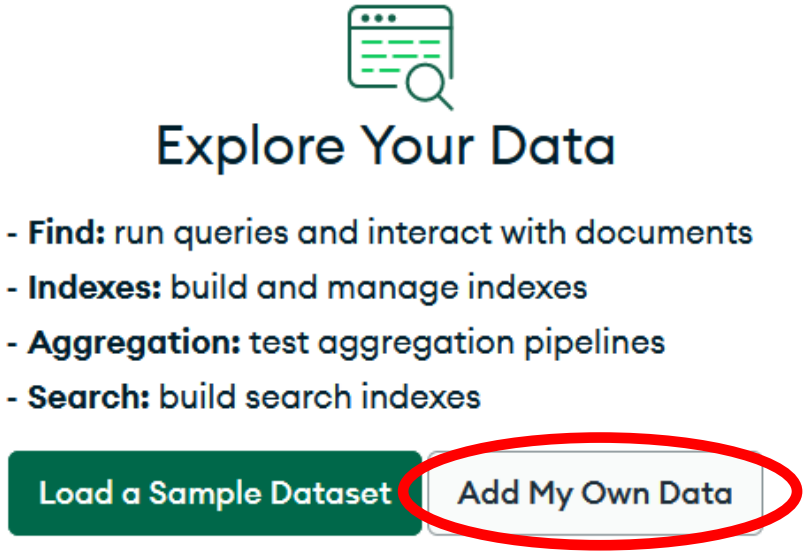




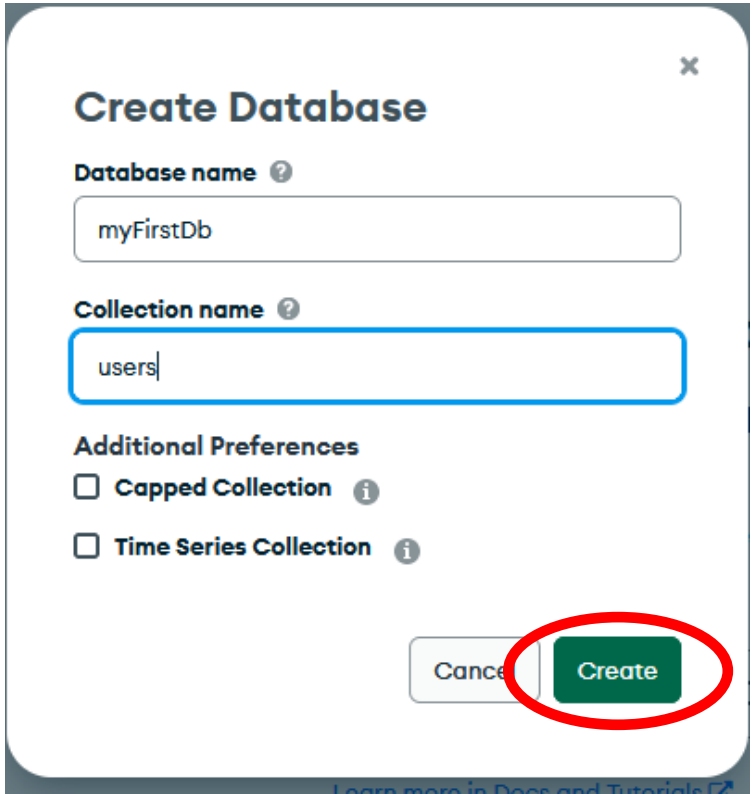


- Click "Connect" and then "Connect your application"
- You will see a connection URL like the following:
  - `mongodb+srv://mongouser:<password>@cluster0.omwq8.mongodb.net/?retryWrites=true&w=majority`
  - Copy it and store it in the same place you stored the password.
- Later, when trying to access the database, you will need to replace <password> with the password you just created. (Not now… later).

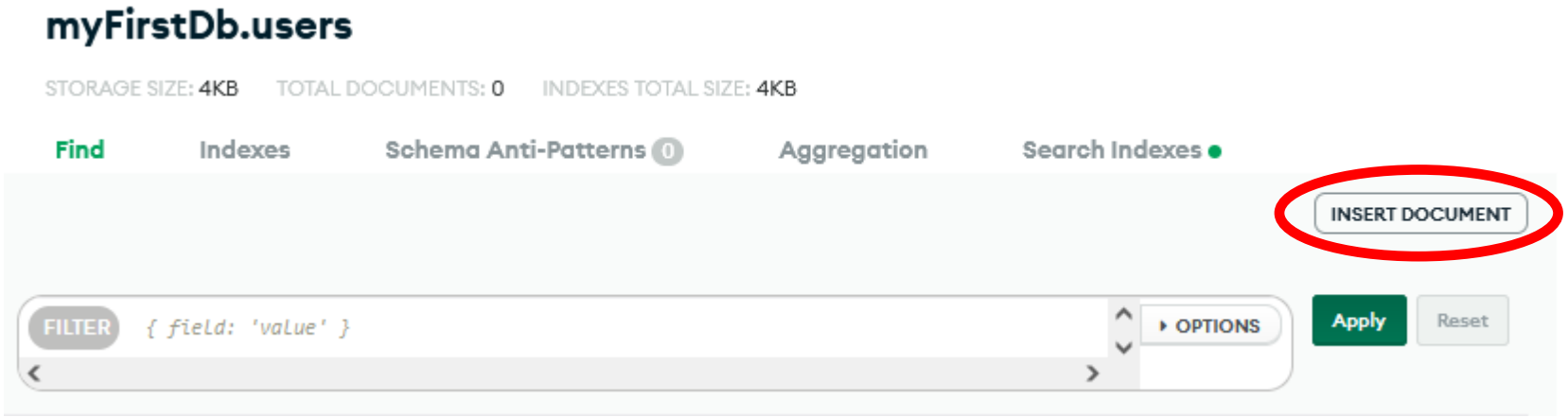




- Let's setup our database
- Click on "Browse Collections"
- Click on "Add my Own Data"
- Then, enter a name for your database
  - Remember this name (we'll need it for the connection URL)
- Enter a name for the collection
- Click "Create"



Add Data



- Click "Insert Document" to add data to the collection
  - This is like adding a row to a table in MySql
- Add a sample user with an 'username' field that is a string and a 'password' field that is a string
  - A NoSql database like MongoDB stores information simply as documents comprised of name-value pairs.
  - This is essentially JSON.
- For now, create a user with username of <yourname> and a password.

Simple Back-end

- Create a new folder in your repository folder called `simple-backend`
- Open a new Visual Studio Code window and open that folder
- In terminal,
  - `npm init`
  - `npm i express body-parser cors mongoose dotenv`
- In `package.json`, add the following to the `scripts` section:
  - `"start": "node index.js"`
- Create new file `index.js` at top-level of project

```
const express = require('express');
const bodyParser = require('body-parser');
const cors = require('cors'); //cross-orign resource sharing

const app = express();
const port = 3001; // Must be different than the port of the React app

app.use(cors());  // https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS
app.use(express.json()); // Allows express to read a request body
// Configuring body parser middleware
app.use(bodyParser.urlencoded({ extended: false }));
app.use(bodyParser.json());


app.listen(port, () => console.log(`Hello world app listening on port ${port}!`))
```

- https://niruhan.medium.com/creating-a-simple-mern-fullstack-application-2cbcfbdf3940

- To connect this project up to Github, follow the alternative set of directions given earlier (repeated here).
- In this setup approach, we'll start the repository on Git and then connect to that locally.

DB Schema

- Create a new file called `models.js` at the top-level of the project
  - This will store a database scheme for our User model. This specifies the data fields used in a given collection.
  - By default, mongoose assumes that there is a collection with the pluralized version of the name we give the model here.

```
const mongoose = require("mongoose");

const UserSchema = new mongoose.Schema({
    username: {
        type: String,
        required: true,
    },
    password: {
        type: String,
        required: true,
    },
});

// Mongoose will assume there is a collection called the plural of this name (i.e., 'users' in this case).
const User = mongoose.model("User", UserSchema);

module.exports = User;
```

Valid Schema Types

- The following Schema Types are permitted in Mongoose:
  - Array
  - Boolean
  - Buffer
  - Date
  - Mixed (A generic / flexible data type)
  - Number
  - ObjectId
  - String
- An internal validator will be triggered when the model is saved to the database. It will fail if the data type of a field value is not the type indicated in the schema for that field.

Secret Keys and Connection URL

- To create a complete connection URL, we need to change the one we got from Mongo Db in 2 ways: insert password and insert name of database.
  - `mongodb+srv://mongouser:<password>@cluster0.omwq8.mongodb.net/<nameOfDatabase>?retryWrites=true&w=majority`
  - For example,
  - `mongodb+srv://mongouser:2j3jjda982k@cluster0.omwq8.mongodb.net/myFirstDb?retryWrites=true&w=majority`
- But, using that password string directly in our code is a big security issue
  - Anyone with access to your github code can now access your application's data and steal or corrupt it.
- So, we want to store that password in a "Secrets" file in our Visual Studio Code. This file will not be committed to our git repository.

Storing secret key using `.env`

- Create a file at the root of your folder called `.env`
- Go into `.gitignore` and add `.env`
- Add the password in the `.env` file, giving it a name. Usually all capitals letters are used.
  - `MONGODB_PWD = xxxxxx`
- Then, in your server code in `index.js`, add this require line near the top:

```
require ('dotenv').config();
```

- Then, use `process.env.MONGODB_PWD` in your code to access the value
  - `"mongodb+srv://mongouser:" + process.env.MONGODB_PWD + "@cluster0.omwq8.mongodb.net/myFirstDb?retryWrites=true&w=majority"`

- Note: We can do this in our front-end too, but there are thing differences
  - The name must start with `REACT_APP` (e.g., `REACT_APP_MONGODB`)
  - And, when deployed, the product will actually still have the key in it, so this is still a security issue (since the client has access to the installed code).

- Note: This uses the `dotenv` package, which we installed earlier in our server, and which is installed for us in our front-end as part of create-react-app.
- https://www.pluralsight.com/guides/hiding-secret-keys-in-create-react-app

Connect to DB in back-end

- Back in our `simple-backend` project, let's modify `index.js` to connect to MongoDB
- First, require the packages we need. Enter the following near the top:

```
require ('dotenv').config();
const mongoose = require("mongoose");
const userModel = require("./models");
```

- Next, create a connection to the database. Enter the following after the `app.use(cors());` line
  - Make sure to replace the connection string details with YOUR connection string.
  - Make sure to include the name of the database after mogodb.net/
  - Make sure to user your username.
  - (items in bold might be different for you)

```
mongoose.connect("mongodb+srv://mongouser:" + process.env.MONGODB_PWD +
"@cluster0.omwq8.mongodb.net/myFirstDb?retryWrites=true&w=majority",
    {  useNewUrlParser: true,
        useUnifiedTopology: true    });

const db = mongoose.connection;
db.on("error", console.error.bind(console, "connection error: "));
db.once("open", function () {
    console.log("Connected successfully");
});
```

- In the terminal, type `npm start`
  - If all works properly, you should see `"Connected successfully"`
  - If you get an auth error, please verify that your username, password and cluster are correct in the connection string.

AJAX

- Short for: Asynchronous JavaScript and XML
- Set of web development techniques to enable client-side to send and receive data from a server asynchronously.
- Decouples data interchange layer from presentation layer.
- Allows web pages to change content dynamically without the need to reload the entire page.
- Modern approaches tend to use JSON (JavaScript Object Notation) rather than XML

Fetch

- A typical way to get data dynamically is using 'fetch'
- The fetch function returns a Promise and, when it gets resolved, we receive a response object with a JSON function that returns the JSON content of the response itself.

```
fetch(endpoint)
    .then(response => response.json())
    .then(data => this.setState({ data }))
```

Fetch in React

- Fetch is a tool for accessing API endpoints.
- There are two main ways you are likely to use those endpoints
  - To get data from a data source
  - To change data in a data source.
- There are three main times when you will perform those actions
  - As part of initial setup
  - In response to a user action
  - On re-render to ensure being in sync with latest data.
    - Especially important when data can change due to actions beyond the current user session.
- All these operations are asynchronous, so we want to handle them appropriately using promises and/or async/await.
  - We also don't want a render process to unnecessarily re-send a request.

- https://www.mariokandut.com/how-to-trigger-data-fetching-with-react-hooks/
- https://reactjs.org/docs/faq-ajax.html

Where to Make AJAX request in React

- In React, an AJAX request should be made after the component has been rendered to the browser.
  - Usually, the request only needs to be made once.
- The best place to make the request that happens automatically in a component is inside a `useEffect` hook.
  - E.g., Populate the state data for a list the component will display with the latest data values
- But, when the user initiates the action directly (e.g., via a button), we can make the AJAX request in a handler function.

React Front-End with Fetch

- Now that we have set up a simple backend, let's make our back-end and front-end communicate with each other via an endpoint.
- **IN OUR BACK_END**… in `index.js`, before the final `app.listen(port,…)` line, add the following:

```
app.get('/users', async (req, res) => {
    const users = await userModel.find();
    res.send(users);
});
```

  - This line setups up a server-side route for the `GET /users` endpoint that will call the callback function to handle the endpoint when it is visited.
- **IN OUR REACT APP**…. In `Home.js`, insert a `<DbData />` component on our page.
- Create `components/DbData.js` with the following initial content:

```
import React from "react";

function DbData() {
  return (
    <>
      <button onClick={callGetAll}>Get all users</button>
   </>
 )}

/* Function that will make an API call to get all users, and then pop-up an alert with the results.
*/
function callGetAll() {
  fetch("http://localhost:3001/users", { method: "GET" })
    .then((data) => data.json())
    .then((json) => alert(JSON.stringify(json)));
}
```

- After looking through the following slides, add additional buttons here to call each method.

Mongoose

- Mongoose queries provide several helper functions for CRUD operations.
- Each of these functions returns a mongoose Query object
  - A Query has a .then() function and you can use await, but it is NOT a promise.
  - https://mongoosejs.com/docs/queries.html#queries-are-not-promises

- https://mongoosejs.com/docs/queries.html
  - Model.deleteMany()
  - Model.deleteOne()
  - Model.find()
  - Model.findById()
  - Model.findByIdAndDelete()
  - Model.findByIdAndRemove()
  - Model.findByIdAndUpdate()
  - Model.findOne()
  - Model.findOneAndDelete()
  - Model.findOneAndRemove()
  - Model.findOneAndReplace()
  - Model.findOneAndUpdate()
  - Model.replaceOne()
  - Model.updateMany()
  - Model.updateOne()

HTTP Methods and Data/Parameters

- Several key HTTP methods to use to support CRUD operations
  - POST, GET, PUT, PATCH, DELETE
  - https://restful-api-design.readthedocs.io/en/latest/methods.html
- 3 ways to send data via an HTTP Request
  - Query parameters (used primarily with get)
    - `/users?username=elie`
  - Path parameters
    - `/users/:username` - generalized representation
    - `/users/elie` - actual call
    - Note: The API router cannot distinguish between `/users/:id` or `/users/:username`. They are essentially the same endpoint, so don't use this form multiple times (pick one use-case only).
  - Body (not with get)
    - Typically is JSON data provided via the request body.
    - Benefit to information sent via the body is that it is not "publicly" visible (i.e., doesn't show up in the URL).
    - This is very important for private information such as passwords and personally-identifiable information (PII)
    - https://en.wikipedia.org/wiki/Personal_data
    - https://danielmiessler.com/blog/sensitive-information-sent-in-the-url-over-https/
    - https://medium.com/@robert.broeckelmann/http-post-vs-get-is-one-more-secure-for-use-in-rest-apis-2469753121b0
- Main way to receive responses for most methods:
  - JSON in response body.

CRUD	HTTP Method	REST API Route (Express)	Mongoose Query
Create	POST Data in body	post("/users")	userModel.create(userObject) - Specify all user fields in an object as name value pairs - Can also pass in an array of new user objects
Read (All)	GET	get("/users")	userModel.find()
Read (One)	GET	get("/users/:username") - path parameter	userModel.findOne(match-condition) - match-condition: Look for documents whose fields match the given value(s) - e.g., userModel.findOne({username : 'thussain'})
Read (One)	GET	get("/user?username=xxx") - query parameter	
Read (One)	POST with username in body **(more secure)**	post("/users/get") - body param *(verb ok here)*	
Update (Full)	PUT with username & changed info in body	put("/users")	userModel.replaceOne(match-condition, userObject) - Specify all user fields, including updates in an object
Update (Partial)	PATCH with username as Path parameter	patch("/users/:username")	userModel.updateOne(match-condition, changedFields) - Specify just the changed fields in an object
Delete	DELETE with username in body	delete("/users")	userModel.deleteOne(match-condition)
Delete	DELETE with username as Path Parameter	delete("/users/:username")	userModel.deleteOne(match-condition)

Create (POST)

Handler in Front-end

```
function callPostBody() {
  fetch("http://localhost:3001/users", {
    method: "POST",
    body: JSON.stringify({
      username: “elie",
      password: “EliePassword",
    }),
    headers: {
      "Content-type": "application/json;
charset=UTF-8",
    },
  })
    .then((data) => data.json())
    .then((json) =>
alert(JSON.stringify(json)));
}
```

Handler in back-end

```
/* An API post request using body /users
*/
app.post("/users", async (req, res) => {
  const username = req.body.username;
  const password = req.body.password;
  const user = {
    username: username,
    password: password,
  };
  await userModel.create(user);
  res.send(user);
});
```

Read-All (GET)

Handler in Front-end

```
function callGetAll() {

  fetch("http://localhost:3001/users", {
method: "GET" })

    .then((data) => data.json())

    .then((json) =>
alert(JSON.stringify(json)));

}
```

Handler in back-end

```
/* An API get request to /users to get
all users */

app.get("/users", async (req, res) => {

  const users = await userModel.find();

  res.send(users);

});
```

Read-Single (GET) – Query params

Handler in Front-end

```
function callGetQuery() {

fetch("http://localhost:3001/user?usernam
e=elie", { method: "GET" })
    .then((data) => data.json())
    .then((json) =>
alert(JSON.stringify(json)));
}
```

Note: We do not user 'users' here since get /users would have already matched to read-all.

Handler in back-end

```
/* An API get request using query
parameters to /users?username=XXX */
app.get("/user", async (req, res) => {
  const username = req.query.username;
  const user = await userModel.findOne({
username: username });
  res.send(user);
});
```

Read-Single (GET) – Path params

Handler in Front-end

```
function callGetParams() {

fetch("http://localhost:3001/users/elie",
{ method: "GET" })
    .then((data) => data.json())
    .then((json) =>
alert(JSON.stringify(json)));
}
```

Handler in back-end

```
/* An API get request using URL path parameters
to /users/:username */
app.get("/users/:username", async (req, res) => {
  const username = req.params.username;
  const user = await userModel.findOne({
username: username });
  res.send(user);
});
```

Read-Single (POST) – Body params

Handler in Front-end

```
function callGetBody() {

fetch("http://localhost:3001/users/get",
{
    method: "POST",
    body: JSON.stringify({
      username: “elie",
    }),
    headers: {
      "Content-type": "application/json;
charset=UTF-8",
    },
  })
    .then((data) => data.json())
    .then((json) =>
alert(JSON.stringify(json)));
}
```

Handler in back-end

```
/* An API post request using body to get user
/users/get */
app.post("/users/get", async (req, res) => {
  const username = req.body.username;
  const user = await userModel.findOne({
username: username });
  res.send(user);
});
```

Note: Using POST for a read operation is more secure than GET when the information being transmitted as parameters includes potentially private information such as usernames.

Update – Full (PUT)

Handler in Front-end

```
function callPutBody() {
  fetch("http://localhost:3001/users", {
    method: "PUT",
    body: JSON.stringify({
      username: “elie",
      password: "125"
    }),
    headers: {
      "Content-type": "application/json;
charset=UTF-8",
    },
  })
    .then((data) => data.json())
    .then((json) =>
alert(JSON.stringify(json)));
}
```

Handler in back-end

```
/* An API post request using body /users.
Replaces the entire user. */
app.put("/users", async (req, res) => {
  const password = req.body.password;
  const username = req.body.username;
  const user = {
    username: username,
    password: password,  };
  const results = await userModel.replaceOne({
username: username }, user);
  console.log("matched: " + results.matchedCount);
  console.log("modified: " +
results.modifiedCount);
  res.send(results);
});
```

Update – Partial (PATCH)

Handler in Front-end

```
function callPatchBodyUsername() {

fetch("http://localhost:3001/users/elie/password", {
    method: "PATCH",
    body: JSON.stringify({
      password: "126"
    }),
    headers: {
      "Content-type": "application/json; charset=UTF-8",
    },
  })
    .then((data) => data.json())
    .then((json) => alert(JSON.stringify(json)));
}
```

Handler in back-end

```
/* An API post request using body /users/username that changes a single field */

app.patch("/users/:username/password", async (req, res) => {

  const username = req.params.username;

  const password = req.body.password;

  const results = await userModel.updateOne({ username: username }, { password: password });

  console.log("matched: " + results.matchedCount);

  console.log("modified: " + results.modifiedCount);

  res.send(results);

});
```

Delete (DELETE)

Handler in Front-end

```
function callDeleteParams() {

fetch("http://localhost:3001/users/elie",
{ method: "DELETE" })

    .then((data) => data.json())

    .then((json) =>
alert(JSON.stringify(json)));

}
```

Handler in back-end

```
/* An API delete request using URL path
parameters to /users/:username */

app.delete("/users/:username", async
(req, res) => {

  const username = req.params.username;

  const results = await
userModel.deleteOne({ username: username
});

  res.send(results);

});
```

Error Handling

- So far, our back-end code has no error-handling…
- We'd like to make sure our server doesn't crash every time the user enters invalid data, etc.
- Since we are using `await` for each call to the `userModel`, we can wrap that call with a `try…catch`

User Authentication

- Let's take what we know so far and create a user register / login capability
- But, we can't store the passwords directly ("in the clear")
  - This is a big security flaw.
- We'll store hashed passwords instead
  - Using a node module called bcrypt
- We'll also validate our usernames and passwords to ensure they are secure.
  - Using a node module called validator
- Delete the existing data in the database (either via the MongoDb web interface or using your delete button).

Hashing

- When we store a password, we want to ensure that it is stored in a form that cannot be read.
- **Encryption**, which we typically think of when trying to protect data, is technically an algorithm that is designed to be reversible.
  - i.e., if you encrypt it, then you can also decrypt it with a corresponding algorithm
  - Encryption is therefore "two-way"
- **Hashing**, by contrast, is "one-way". The original data is sufficiently changed to protect it, but you can't ever get back the original from just the hashed value.
- But, if you have the original value, you can hash it again and get back the same hash value.
  - This is useful for verifying that a file or piece of data hasn't been altered—that it is authentic. In other words, it serves as a check-sum, or as a way of comparing two pieces of information to see if they are the same.
- For passwords, this means:
  - Given an original password passwordOriginal, you store the value of hash(passwordOriginal).
  - Later on, if the user gives you a password (say, passwordToCheck). you can check to see if hash(passwordToCheck) equals the stored hash. If yes, then passwordToCheck is the same as passwordOriginal.
  - This let's you verify that the user has entered the correct password.
- One other term that is related to hashing is **salting**. Salting is the process of adding random information to the original password before hashing it. This makes the hash stronger (i.e., harder to break with brute force attacks).
  - A different random salt value should be used for every password. Then, even if someone cracks one password, they won't be able to crack a second one without starting all over again.
- https://www.thesslstore.com/blog/difference-encryption-hashing-salting/

Hashed, Strong Passwords

- To hash our passwords, we'll use the `bcrypt` module. To ensure strong passwords and good usernames, we'll use the `validator` module.
  - `npm i bcrypt validator`
  - https://www.npmjs.com/package/bcrypt
- In `index.js`, add the following near the top.

```
const validator = require('validator');
const bcrypt = require('bcrypt');
const saltRounds = 10;
```

  - `saltRounds` is a parameter that influences the randomization process used to make the salt value.
- In method for handling the `/register` endpoint,
  - Make the callback function `async`
  - Add a truthy check for password and username being non-empty
  - Add a check for password strength and username validity. This makes your application more secure.

```
validator.isStrongPassword(password) &&
validator.isAlphanumeric(username)
```

  - Make sure the user doesn't already exist
  - Store a hashed password:

```
const hashedPassword = await bcrypt.hash(password, saltRounds);
```

    - In other words, don't store the original password, but instead stored the hashed version.
- In method for `/users/login` endpoint:
  - Compare a password (unhashed) to the saved hashed version

```
const isSame = await bcrypt.compare(password, user.password);
```

```
app.post("/users/register", async (request, response) => {
  const id = request.body.id;
  const username = request.body.username;
  const password = request.body.password;
  try {
    if (
      username &&  validator.isAlphanumeric(username) &&
      password &&  validator.isStrongPassword(password)     ) {
      // Check to see if the user already exists. If not, then create it.
      const user = await userModel.findOne({ username: username });
      if (user) {
        console.log("Invalid registration - username " + username + " already exists." );
        response.send({ success: false });
        return;
      } else {
        hashedPassword = await bcrypt.hash(password, saltRounds);
        console.log("Registering username " + username);
        const userToSave = {  username: username,
                              password: hashedPassword };
        await userModel.create(userToSave);
        response.send({ success: true });
        return;
      } }
  } catch (error) {   console.log(error.message);  }
   response.send({ success: false });
});
```

```
app.post("/users/login", async (request, response) => {
  const username = request.body.username;
  const password = request.body.password;
  try {
    if (username && password) {
      // Check to see if the user already exists. If not, then create it.
      const user = await userModel.findOne({ username: username });

      if (!user) {
        console.log("Invalid login - username " + username + " doesn't
exist.");
        response.send({ success: false });
        return;
      } else {
        const isSame = await bcrypt.compare(password, user.password);
        if (isSame) {
          console.log("Successful login");
          response.send({ success: true });
          return;
        }
      }
    }
  } catch (error) {
    console.log(error.message);
  }
  response.send({ success: false });
});
```

Gated Access

- Now that we can register users and verify them for the purposes of logging in, we can now build our front-end to have content that is displayed depending on the logged-in status of the user.
  - We say that access to protected content is "gated".
- For example,
  - Admin dashboard
  - User dashboard
  - User profile
  - User shopping cart
  - Personalized UI (dark theme, etc.)

What do we need in our UI?

- Based on what we've learned so far and what we've just built in our back-end, what do we need in our front-end to implement a web site with login/register/gated-access capabilities.

→ Think and Discuss

Fill in username/password (Form)

Register form / button

Login form / button

→ Forget password

→ Choose what type of user you will login as

→ Dashboard to land on

→ Verification for user group

→ State to remember
  - → Boolean of logged in or logged out (IsVerified)
  - → Id (unique identifier)
  - → Custom profile

→ (Use sessions)

→ Logout
  - → Timeout

What do we need in our UI?

- We need to include UI elements that enable a user to login (or register)
  - Specifically, a form to get username and password
    - Right now our sample front-end code is using hard-coded values
- We need some way to handle the response of our back-end to detect a successful login (or register)
  - The back end is returning JSON with {success : true} or {success : false}
- We need some state to keep track of the fact that they are logged in
  - And some way to let multiple components know about logged-in state
- We need to make the rendering of some components conditional on whether they are logged in.
  - Gated content should never be visible when logged out
- We need to include UI element that enables a user to logout and updates logged-in state accordingly
  - E.g., Logout button when logged in (and vice versa)

- Advanced: We need to time-out a user if they are inactive for too long.

In-Class Exercise 2: Login/Register and Gated Access

- This exercise is worth marks (5%).
- It has 5 parts, each worth 1%.

Forms in React

- Just like HTML, React uses forms to allow a user to interact with a web page and enter data.
- React supports two types of forms: **uncontrolled** and **controlled**.
- In an uncontrolled form, the form behaves just like an HTML form and the value enter by the user is kept in the input's DOM node. The component accesses the value after it has been entered.
- In a controlled form, the value entered by the user is stored in the component's state, and every keypress changes the state of the component.
  - This allows us to insert logic after every keypress (e.g., to check for invalid inputs)
  - On the plus side, it is very much in line with how Reacts works – the UI reflects state (i.e., every key-press results in a re-render)
  - On the down side, this can result in a lot of re-renders making the app less performant in some cases.
- Additionally, HTML forms have a specific behavior – upon submission, the page refreshes.
  - Since we have a single page application, reloading the html page wouldn't be good…
  - We must always use a special function `event.preventDefault()` in the form's handler function to prevent this default behavior
  - https://reactjs.org/docs/handling-events.html
- https://reactjs.org/docs/forms.html
- https://reactjs.org/docs/uncontrolled-components.html
- https://www.w3schools.com/react/react_forms.asp
- https://daveceddia.com/react-forms/
- https://goshacmd.com/controlled-vs-uncontrolled-inputs-react/

Part 4-1: `LoginForm`

- We'll learn the uncontrolled form approach now, and the controlled approach later.
- In order to access the value of the input from the DOM, we must specify a reference (`ref=`) for each input.
  - The `useRef()` hook is provided for this.
- Create a new component called `LoginForm.js`:

```
import React, { useRef } from "react";

function LoginForm() {
   const usernameRef = useRef();
   const passwordRef = useRef();

   const handleSubmit = (event) => {
       event.preventDefault(); // prevent page reload

       // to fill in based on callPostBody
   }

   return (
    <form onSubmit={handleSubmit}>
      <label htmlFor="username">Username</label>
      <input id="username" type="text" ref={usernameRef} required />

      <label htmlFor="password">Password</label>
      <input id="password" type="password" ref={passwordRef} required />
      <button type="submit">Login</button>
    </form>
  );
```

- Copy the `callPostBody` method from earlier and use it as the basis for the logic for `handleSubmit`.
- Use `usernameRef.current.value` to access the value of the username submitted, and similarly for password.

Let's register a user (hard-coded) for testing purposes

- In DbData, let's create a function (and associated button) that will register a user for us to test with
  - We need to use our register endpoint so that the hashedPassword gets stored. Otherwise, there is no way to test our login form capability.
  - Just hard-code a test username & password. Make sure the username is unique (i.e., not in the database already)

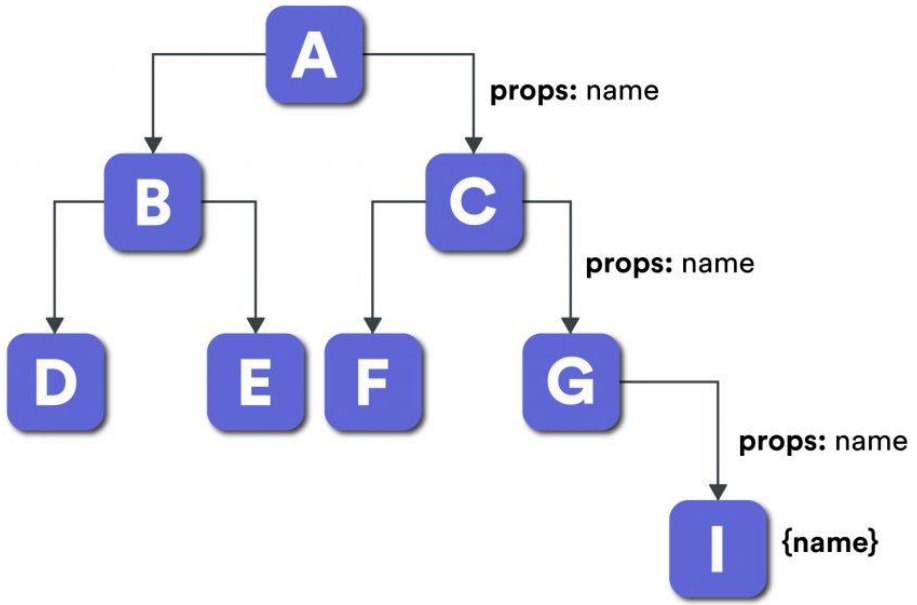
```
async function registerUserTemp() {
  fetch("http://localhost:3001/users/register", {
    method: "POST",
    body: JSON.stringify({
      username: "thussainz",
      password: "Abcd1234!",
    }),
    headers: {
      "Content-type": "application/json; charset=UTF-8",
    },
  })
    .then((data) => data.json())
    .then((json) => alert(JSON.stringify(json)));
}
```

State

- Let's say the user has logged in.
- We need to store that information (e.g., `isLoggedIn`) in the state of a particular component.
  - This component must be as high or higher in the DOM than any component that could show gated content.
- We need to make sure that state is available to all components that need it in the DOM below
- We need to include logic in any gated component so that it only shows gated content if the state allows it (i.e. when `isLoggedIn` is true)

The Problem of Prop Drilling

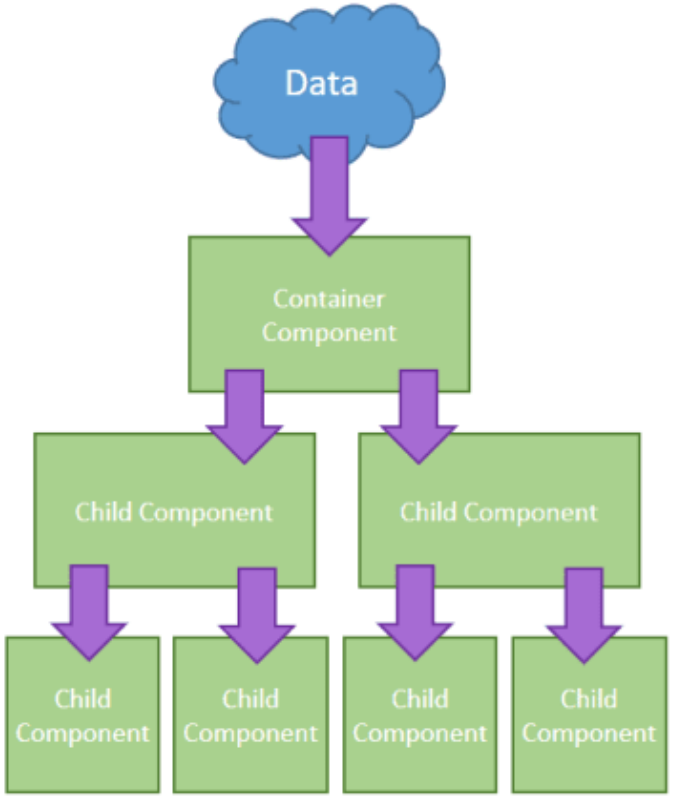
- A simple answer to this problem, based on what we know so far, is to pass the value of `isLoggedIn` down the React component tree to all components that need it using props.
- While this is fine in a small React component tree, it can become inelegant in larger trees since it means that all components along a path in the tree must have that prop – even if they don't need it – just so a component lower in the tree can use it.
- In a larger application, you may find yourself drilling through many layers of components – especially when many components require that prop value.
    - This can cause issues with maintaining the code, such as ensuring that changes to prop names or data format are properly handled everywhere and making sure intermediate components don't over-forward or under-forward props
- https://kentcdodds.com/blog/prop-drilling



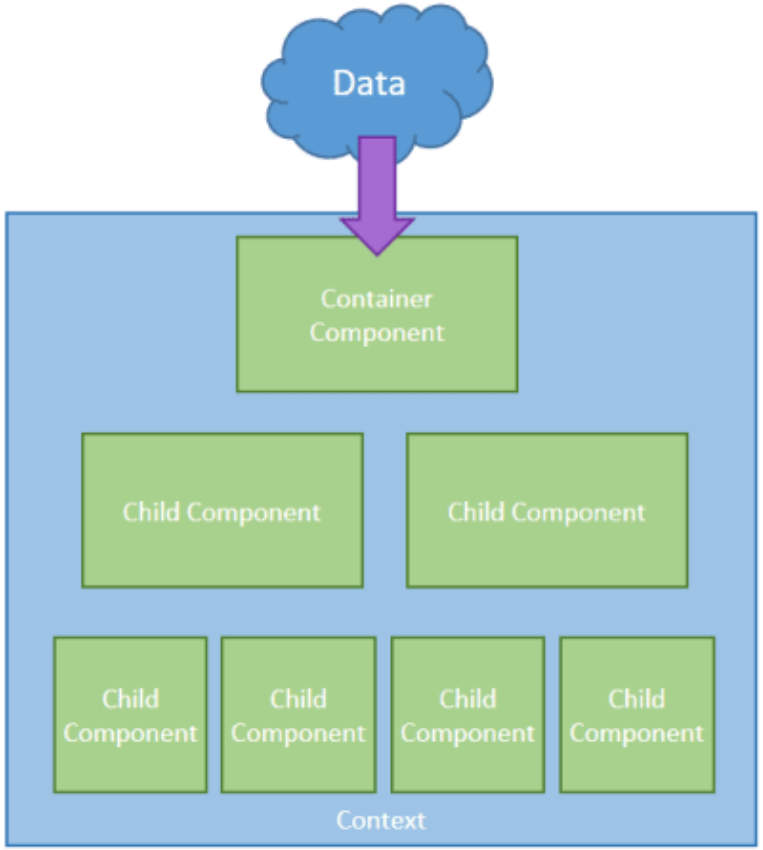
React Context & Provider

- React provides an elegant solution to this challenge.
- The Context API Context provides a way to pass data throughout (part of) the component tree without having to pass props down manually at every level.
- We setup a `Context` and `Provider` at the high level and then access it at a low level via a `useContext()` hook

- https://reactjs.org/docs/context.html#contextprovider
- https://flexiple.com/react/provider-pattern-with-react-context-api/
- https://stackoverflow.com/questions/41030361/how-to-update-react-context-from-inside-a-child-component
- https://www.freecodecamp.org/news/react-context-for-beginners/
- https://dev.to/vidhithakur/react-context-api-4fk4
- https://blog.openreplay.com/redux-who-handle-your-own-state-instead/



prop drilling



context API

Part 4-2: `LoggedInContext` & Provider

- First, we identify the highest component below which we likely need access to the logged-in state.
  - In our case, it is the `App.js` level.
- Next, we define a Context object using the `createContext()` function.
  - We can specify an initial value for it plus a setter. The setter is defined as empty and will be specified in the next step.
  - This will be defined in `App.js`, but outside of the `function App().`
  - The context must be exported so it can be used lower in the tree.

```
export const LoggedInContext = React.createContext({
  isLoggedIn: false,
  setIsLoggedIn: () => {},
});
```

- In that same component, we use `useState()` to create a value to store the logged-in state of the user.

```
const [isLoggedIn, setIsLoggedIn] = useState(false);
const loggedInValueToProvide = [isLoggedIn, setIsLoggedIn]; // So we can pass down both value and setter
```

- Then, we use a Provider component to provide access to that context throughout the tree. The Provider wraps the component at that same high level (e.g., `App.js`) and sets the value

```
<LoggedInContext.Provider value={loggedInValueToProvide}>

</LoggedInContext.Provider>
```

Part 4-3: Update Context Value

- Finally, in the low-level component (e.g., `LoginForm.js`) where we want to use or change the value in the context, we import the context and use `useContext()` hook.

```
import { LoggedInContext } from "App";

const [isLoggedIn, setIsLoggedIn] = React.useContext(LoggedInContext);
```

- Now that we have access to `isLoggedIn` and `setIsLoggedIn` in our `LoginForm` component, we just have to update the value accordingly in the form handler.

```
.then((json) => {
        alert(JSON.stringify(json));
        json.success ? setIsLoggedIn(true) :
                       setIsLoggedIn(false);
      });
```

Conditional Rendering

- So far, we've largely applied conditions within the attributes of a React element
- We can also apply a condition to determine the entire element. This is called conditional rendering.
  - https://reactjs.org/docs/conditional-rendering.html
- Generally, use the ternary operator to either render the gated element or render an alternative.
- Use the notation `condition && <element>` to either render the gated element or nothing
  - In JavaScript, `true && expression` always evaluates to expression, and `false && expression` always evaluates to false.

- https://www.freecodecamp.org/news/learn-react-conditionals/

Part 4-4: Gate Content

- To prove our `LoggedInContext` is working, let's update `Header.js` to indicate when we are logged in or not and hide our `Home` component.
- In `Header.js`:
  - First, import the context and use it.

```
import { LoggedInContext } from "App";
const [isLoggedIn, setIsLoggedIn] = useContext(LoggedInContext);
```

  - Then, add a message in our header that changes based on the whether the user is logged in or not

```
<div>
  {isLoggedIn ? "Welcome" : "Please login"}
  <HomeButton />
</div>
```

- In `App.js`, make the display of `Home` conditional. If the user is not logged in, then display the `LoginForm` instead.
  - Remember to use the ternary operator

Part 4-5: `LogoutButton`

- Create a `LogoutButton.js` that will `setIsLoggedIn()` to false when clicked.
  - You have enough knowledge now to build this

- Put the `LogoutButton` in the `Header`, but only render it if `isLoggedIn` is true.
  - Remember to use `condition && <element>`

Submit to Lea

Zip up the following and submit on Lea.

1. Screenshot of your home page when not logged in
2. Screenshot of your homepage when logged in.
3. Your src folder of your React app
   (exclude node_modules)

Upload the zip file to Lea

If Finished Early / To Try at Home

- Create a `RegisterForm.js`
  - Similar to `LoginForm`, but uses the `/register` endpoint.
- When logged out, show a `Header` with a `Register` and `Login` button.
  - Show a blank main section on the web page since you are logged out.
- Clicking a button takes you to `/registerform` or `/loginform` route, which displays the corresponding form in the main section of the web page.
- A successful register logs you in too.
- Explore adding styles to make your site look interesting.