|  |  |
| --- | --- |
| 0x00ES6 Basics: | |
| * Concepts:   Rest parameter, spread syntax, template literal, computed property name  **Spread syntax**:  Uses of Spread syntax:  Given : array1 = [1, 2, 3] array2 = [a, b, c]  Obj1 = { a: 1, b: 2} obj2 = {x: 3, y: 4}  String = “tom”   1. Array related:    1. shallow copy array   copiedArray1 = [. . . array1] >>> [1, 2, 3]   * 1. concatenate array   concatenated = [. . . array1, . . . array2]  >>> [1, 2, 3, a, b, c]   1. string related:    1. unpack: unpacked = [...string] >>> [‘t’, ‘o’ , ‘m’] 2. Object related:    1. Merger objects: Const newobj = {…obj1, …obj2} >>> { a: 1, b: 2, x: 3, y: 4}    2. Add new property to object   const addobj = {…obj1, c: 10}  >>> {a: 1, b: 2, c: 10}   1. As function parameter (as rest parameter)   FunctionName(… arguments)  Two types of copy shallow and deep   1. **Deep copy**: involves copying not only the top-level properties of an object but also ensuring that any nested objects within it are also copied and not just referenced. 2. **Shallow copy**: the outer object (or the first level of properties) is copied, the properties inside the object (nested objects or arrays) are still references to the original properties.   . const originalObject = {x: 1, y: {z: 2 } };  // Creating a deep copy  const deepCopyObject = JSON.parse(JSON.stringify(originalObject));  // creating shallow copy  Const shallowCopyObject = [... originalObject];  // if the properties are changed like this  originalObject.x = 10;  originalObject.y.z = 20;  console.log(originalObject); // {x: 10, y: { z: 20 }}  console.log(deepCopyObject); // {x: 1, y: { z: 2 }}  console.log(shallowCopyObject); // { x: 1, y: {z: 20}} | **Object Creation**  Syntax: **const myObject = { key: 'value'};**  when creating property for js object using object literal  the key can be created using three methods:  given : **const myKey = “key1”;**   1. **Const obj = {key1: value};//name variable** 2. **Const obj = {‘key1’: value};//single quote** 3. **Const obj = {“key1’: value};//double quote** 4. **Const obj = {[mykey]: value};// computed property (if it is calculated then it should be enclosed in [])**   **Object Methods**  **Object.entries()** method returns an array of a given object's own enumerable property [key, value]  **Object.keys()** method returns an array of a given object's own enumerable property names (keys).  **Use of [ ]**   1. Array creation: **let array = [ 1, 2, 3]** 2. Object property access **let valueName= obj[‘key name’]** 3. Destructuring : **let array = [ 1, 2, 3]**   **let [a, b, c] = array; >>> a = 1, b = 2, c = 3**   1. Regular expressions:   Let regex = /[abc]/ // matchs any a, b or c characters |
|  |  |

|  |  |
| --- | --- |
|  | |
|  |  |

|  |  |
| --- | --- |
| 0x02 ES6 Class | |
| **Type Checking with typeof:**  typeof(name) === "string";  // Validates if 'name' is a string  typeof(age) === "number"; // Validates if 'age' is a number   **Instance Checking:**  obj instanceof ClassName; // Checks if 'obj' is an instance of 'ClassName'   **Null or Undefined Checking:**  typeof variable !== 'undefined' && variable !== null; // Checks if variable is neither undefined nor null   **Checking Object Keys:**  obj.hasOwnProperty('keyName'); // Checks if 'obj' has a specific property  'keyName' in obj; // Another way to check if 'obj' has a specific property   **Array Checking**  Array.isArray(someArray); // Checks if 'someArray' is an array   **Checking for Functions:**  typeof someFunction === 'function'; // Checks if 'someFunction' is a function   **Truthy/Falsy Checking:**  if (variable) { /\* ... \*/ } // Checks if 'variable' is truthy (not null, undefined, 0, false, empty string, etc.)   **Regular Expression Validation:**  javascript   const regex = /^[a-zA-Z]+$/;  regex.test(someString); // Checks if 'someString' matches the regular expression pattern   **Type Checking using Object.prototype.toString:**  Object.prototype.toString.call(variable) === '[object Object]'; // Checks if 'variable' is an object   Custom **Validation Functions:** You can create custom functions to perform specific validation based on your requirements.   1. function isValidEmail(email) { 2. // Custom validation logic for email 3. } | **Static classes:**  Are functions that belong to the class itself rather than to instances created by the class. They are accessed using the class name rather than an instance of the class. These functions are declared with the static keyword within a class.  Symbol.toPrimitive method is a special symbol that allows objects to define their behavior when they are coerced to a primitive type (such as a number, string, or default). This method gets called by JavaScript internally when an object is converted to a primitive type explicitly or implicitly.  The Symbol.toPrimitive method takes a single argument hint, which represents the type of conversion being performed (e.g., 'number', 'string', or 'default').  There are 6 primitive types in js  Number, string, Boolean, null, undefined, symbol  Example: const class1 = class()  Class1.toString() ------- Symbol.toString() method is called  String(class1) ---when casted(coerced)—Symbol.toPremitive() method is called |

|  |  |
| --- | --- |
| 0x03. ES6 data manipulation | |
| **Map:** method creates a new array by calling a provided function on every element in the original array.  **Application:** Doubling each number in an array  const numbers = [1, 2, 3, 4];  const doubled = numbers.map(num => num \* 2);  **Filter:** method creates a new array with elements that pass a certain condition defined by a provided function.  **Application:**  Filtering event numbers in an array  const arr = [1, 2, 3, 4];  **const even = arr.filter(n => n % 2 === 0);**  **const e2 = arr.filter((n) => {return ( / 3 === 0)});**  **Reduce:** method applies  a function against an accumulator and each element in the array (from left to right) to reduce it to a single value.  **Application:** Summing all numbers in an array  const ar = [1, 2, 3, 4];  const sum = arr.reduce((acc, n) => acc + nu, 0);  **Every():**method checks if all elements in an array pass a certain condition provided by a function. It returns true if all elements satisfy the condition; otherwise, it returns false.  **Application:** Check if all are positive  const arr = [1, 2, 3, 4];  const rslt = arr.every((n) => n > 0);  **find():**method is used to get the first element in an array that satisfies a provided testing function. It returns the value of the first element in the array that satisfies the provided function.  **Application:** find the first even number  const arr = [1, 2, 3, 4];  const rslt = arr.find((n) => n % 1 === 0);  output>>> 2;  **slice ():**method in JavaScript is used to extract a section of an array or a string and create a new array or string from that section. It doesn't modify the original array or string; instead, it returns a shallow copy of the selected elements or characters.  **Application:** find the first even number  const arr = [1, 2, 3, 4];  const rslt = arr.find((n) => n % 1 === 0);  output>>> 2;  **startsWith()** method in JavaScript is used to determine whether a string starts with the characters of a specified string, returning true or false as the result.  const str = 'Hello, World!';  console.log(str.startsWith('Hello')); True  console.log(str.startsWith('World', 7)); True | ITERATE or LOOP  Iterate across a dictionary in javascript   1. for (const [key, value] of Object.entries(dictionary)) {   console.log(`${key}: ${value}`);  }   1. Object.entries(dictionary).forEach(([key, value]) => {   console.log(`${key}: ${value}`);  **Using Template Literals:**  javascript  console.log(`${key}: ${value}`);  This uses template literals, which is a modern JavaScript feature that allows you to embed expressions within strings using ${...}. It provides a cleaner and more readable way to interpolate variables into strings.  **check if key exists in a dictionary**  const myObject = { k1: 'v1', k2: 'v2'};  if (myObject.hasOwnProperty(keyToCheck))  console.log(‘key exists”);  the above can create error by ESlinter so use this instead  if (Object.prototype.hasOwnProperty.call(myObject, keyToCheck)) {  console.log (`${keyToCheck} exists in the object.`);  Sets:   * Store unique values * Are indexed by their insertion order   **Array Buffer:**   * It does not have methods to directly interact with the data it holds; instead, it serves as a container for raw binary data * It allocates a specific length of memory in bytes but does not provide methods to directly read or write data.   **DataView** is an interface that provides a low-level interface for reading and writing multiple number types in an ArrayBuffer. It allows you to interpret and manipulate the raw binary data stored in an ArrayBuffer.  ArrayBuffer and DataView are two concepts in JavaScript used to handle binary data.  **Bytes for datatypes**  null and undefined: typically take up 4 bytes of memory.  String: 2 bytes (16 bits) for individual characters. due to JavaScript's use of UTF-16 encoding.  Number: 64 bits (8 bytes) for both float and integer.  Boolean: 2byte  Object and array: depends on the property they contain. |

|  |  |
| --- | --- |
| 0x04 Typescript | |
|  |  |

|  |  |
| --- | --- |
| 0x00 Html Advanced | |
| 1. <!DOCTYPE html>   declaration represents the HTML5 standard.   1. <html lang="en" dir="ltr">   dir="ltr" specifies the text direction as left-to-right (ltr stands for left-to-right)   1. <meta name="description" content="Techium is a digital agency">   help search engines understand the content of your webpage, potentially influencing how it appears in search results. Additionally, it provides a concise summary for users browsing the web.   1. <link rel="icon" type="image/x-icon" href="./favicon.ico">       <link rel="icon" type="image/png" href="./favicon.png">  By providing both formats (favicon.ico and favicon.png) in the <link> tags, you cater to a wider range of browsers and devices, ensuring that your website's favicon displays correctly regardless of the browser's preference or compatibility with different file types. | * Definitions:   <dl> definition list  <dt> term  <dd> defnition   * <blockquote>   <p>This is an example of a blockquote. It represents a quotation taken from another source.</p>  <cite>Source: Author's Name</cite>   * </blockquote><q>: short quote on a single line, no citation * <hr> : horizontal rule * <address> : to write address * <small> : tag is commonly used to display side comments, copyright information, disclaimers, etc., in a smaller font size than the surrounding text. * <pre> tag in HTML stands for "preformatted text." It defines preformatted text where whitespace, line breaks, and spaces are displayed exactly as they appear in the HTML code. * **<details>** element in HTML is used to create a disclosure widget that can be toggled open or closed, revealing or hiding additional content within a document. It's typically used to create collapsible sections of content, where the additional details are initially hidden but can be expanded by the user. |
| For accessibility to a table scope attribute is added  <table>  **<caption>Star Wars Trilogy Data</caption>**  <thead>  <tr>  <th **scope="col">**Title</th>  <th scope="col">Director</th>  <th scope="col">Release Date</th>  </tr>  </thead>  <tbody>  <tr>  <th scope="row">Movie 1</th>  <td>Director 1</td>  <td>2022</td>  </tr>  <tr>  <th scope="row">Movie 2</th>  <td>Director 2</td>  <td>2023</td>  </tr>  <!-- Additional rows -->  </tbody>  </table>   * The scope="col" attribute in the <th> tags within the <thead> section specifies that these headers represent columns. * The scope="row" attribute in the <th> tags within the <tbody> section signifies that these headers represent rows and contain row-specific data (in this case, the names of movies).   Using the scope attribute helps screen readers interpret table headers correctly, allowing users with disabilities to understand the structure and data within the table more effectively. | * VIDEO and AUDIO    <video>: Embeds the video.  <video controls loop poster="thumbnail.jpg">  <source src="video.mp4" type="video/mp4">  **fallback message** Sorry, your browser doesn't support HTML5 video.  </video>   * controls: Displays the video controls (play, pause, etc.). * loop: Sets the video to loop playback. * poster: Displays an image (thumbnail.jpg) while the video is loading. * <source>: Specifies the video source (BigBuckBunny.mp4) and its MIME type (video/mp4). * The text "Sorry, your browser doesn't support HTML5 video." acts as alternative content displayed if the browser cannot render the video. This provides a **fallback message** for users whose browsers do not support HTML5 video or if the video fails to load.   .Iframe: embed websites within our page  <iframe src="https://www.example.com" width="600" height="400" frameborder="0" title="Embedded Example Website"> **fallback message** </iframe>   * title="Holberton School": Provides a title for the iframe. * width="350" and height="200": Sets the width and height of the iframe. * src="https://www.youtube.com/embed/41N6bKO-NVI": Specifies the source URL for the embedded content. * frameborder="0": Removes the iframe border. * allowfullscreen: Allows the embedded content to be displayed in full-screen mode if supported. |

|  |  |
| --- | --- |
| **ACCESIBILITY** | |
| * Inside <svg> tag: use <Title> <title> element provides a title or description for the SVG graphic. It's usually used for accessibility purposes, allowing screen readers to announce the title of the icon to users who may be visually impaired. |  |

|  |  |
| --- | --- |
| 0x01 Developer tools | |
| To find how many resources are loaded on a page Inspector🡪Network🡪filter by the resource  To mitigate this risk, it's recommended to include additional attributes with the target="\_blank" attribute:   * rel="noopener": This attribute tells the browser not to allow the newly opened tab to access the window.opener property in the parent window. This prevents the newly opened tab from having control over the original page, enhancing security. * rel="noreferrer": This attribute prevents the browser from sending the referrer information when the new tab is opened. Referrer information typically contains the URL of the page that linked to the new tab. By using noreferrer, you prevent the newly opened tab from sending this potentially sensitive information, adding an extra layer of privacy and security. * Extensions like **Axe, WAVE, or aXe** Coconut can help in accessibility audits. After installing these extensions, they usually provide an interface to highlight elements like <a> links with insufficient text. |  |

|  |  |  |
| --- | --- | --- |
| 0x02-CSS\_advanced | | |
| * **scroll-behavior: smooth;** instructs the browser to animate the scrolling behavior whenever the user triggers a scroll event within the HTML document. * **Line-height: 1.5 rem;** space between lines * **text-align: center;** property in CSS is used to control the horizontal alignment of text within an element. It works on block-level and inline-level elements and can be applied to align text within the specific element or its children. * **Clear: both:** if you have a container with floated elements inside and you want a subsequent element to start below these floated elements, you might apply clear: both to * **Fill: red;** attribute in SVG (Scalable Vector Graphics) is used to specify the color inside SVG shapes, such as paths, circles, rectangles, etc. It determines the color that will be applied to the interior of the SVG shapes. * **background-size: 90rem auto;** is equivalent to   **width: 90rem;**  **height: auto;**   * **Vertical-align:** bottom is a CSS property that controls the vertical alignment of   1. **inline elements**: like images   2. **table-cell** element within its parent container. * **Inset: 0;** property simplifies setting all four positions at once by accepting values for top, right, bottom, and left in a single declaration   **Inset: 0; is equivalent to**  **top: 0;**  **right: 0;**  **bottom: 0;**  **left: 0;**   * **transition-duration: 1s;** is a CSS property that determines the duration over which a CSS property transition occurs when its value changes.   <figure>  <img src="image.jpg" alt="Description of the image">  <figcaption>Caption or description of the image</figcaption>   * </figure> | Animations:  .but {**transition:color 1s, background-color 1s;**}  Or we can set properties separately  .but {  **transition-property: color, background-color;**  **transition-duration: 3s**} |

|  |  |
| --- | --- |
| 0x03 responsive design | |
| * For extra large devices (no media queries) * For desktop / large devices (max-width: 992px) * For tablet / medium styles (max-width: 767px) * For mobile styles (max-width: 480px)   <figure>  <img src="image.jpg" alt="Description of the image">  <figcaption>Caption or description of the image</figcaption>  </figure> | Responsive typography:  @media (max-width: 480px) {    html { font-size: 57%;   } }  @media (min-width: 481px) and (max-width: 767px) {    html { font-size: 60%; } } |

|  |  |
| --- | --- |
|  |  |
| **Node.js**  is similar in the world of computers. It's a runtime environment that allows developers to run JavaScript code on the server-side, outside the browser. It's known for its ability to handle many tasks concurrently, making it really good for building applications that need to handle lots of users or perform many operations simultaneously. It's like having a whole team of assistants working together to keep things running smoothly behind the scenes on a website or application.  **webpack**  Webpack is like the master builder who takes all these different pieces (like JavaScript, CSS, images, etc.) and organizes them efficiently. It puts similar pieces together, squeezes out any unnecessary bits, and creates a final, compact structure (a bundled file) that makes your website or app load faster and run smoother. Essentially, it's a tool that helps manage and optimize the different parts of your website so that it works really well when people use it.   * 1. **npm init -y**   is a command used to initialize a new package.json file for your project with default settings. The -y flag stands for "yes," meaning it automatically accepts all default values without prompting you to enter information interactively.  Node.js projects managed by npm, dependencies can be divided into two main categories: dependencies and devDependencies.   1. **Dependencies**: These are packages required for the application to run. For instance, libraries or frameworks used in the production code. 2. **DevDependencies**: These are packages needed during development but not in the final production version of the application. For example, testing frameworks, bundlers like webpack, linters, etc.   **webpack.config.js**  file is used to define the configuration for webpack. It provides webpack with instructions on how to process, transform, and bundle your project's assets, such as JavaScript files, stylesheets, images, and more. Here's a breakdown of the general structure of a webpack.config.js file:  **shallow(<Header />)**  shallow(<Header />) is a way to create a simple version of your Header component for testing. Without including other components within the Header component | NB:  To add dependencies manually to the package.json file To add devdependecies  Npm install –save-dev <package-name>  To add dependencies  Npm install –save <package-name>   * Difference between **package.json** and **package-lock.json**: files   **package.json**:-   * It's used to manage project configuration and dependency information * Contains metadata about the project and lists all the dependencies required by the project. It includes information about the project name, version, scripts, and dependencies (both direct and indirect).   **Package-locks.json:-**   * file is automatically generated by npm and is designed to lock down the specific versions of every package's dependencies that were installed. * This file maintains a detailed record of the exact versions of all packages and their sub-dependencies that were installed by npm. It prevents unwanted changes in dependency versions when working on the same project across different machines or when sharing the project with others. |

|  |  |
| --- | --- |
| module: {  rules: [  {  test: /\.css$/,  use: [  'style-loader', // Injects CSS into the DOM via <style> tags  'css-loader' // Interprets @import and url() like import/require() and resolves them  ],  },  // Other rules can be defined here for different file types  ],  },   * **module**: Represents an object in webpack configuration that deals with defining how different modules are treated during the bundling process. * **rules**: Refers to an array of rules used to match specific file types or modules and define how they should be processed. * **test: /\.css$/**: A regular expression that matches files ending with .css. This test property defines the condition for matching files against this rule. In this case, it matches any file with a .css extension. * **use**: Specifies an array of loaders or tools that should be applied to the matched files.   + **style-loader**: A webpack loader that injects CSS into the DOM by adding <style> tags to the HTML document.   + **css-loader**: Another webpack loader that interprets @import and url() within a CSS file and resolves them by treating them as import/require statements. | Regex in javascript:  In JavaScript, regular expressions are often written as literals and are enclosed within forward slashes (/). This notation is used to define the pattern of the regular expression.  For example:  javascript  let regex = /pattern/;  Note from project:  0x02 react props:   1. Write the tests for each component   To develop your tests faster, you can watch them. The test suite will run for every change you make:   * Add the following script to task\_1/package.json: "test-watch": "jest --watch" * Run your suite using npm run test-watch |
| <div dangerouslySetInnerHTML={{ \_\_html: getLatestNotification() }} />  Here's what each part means:   * dangerouslySetInnerHTML: This is a React property used to set HTML content in an element. * {{ \_\_html: getLatestNotification() }}: This is the object you pass to dangerouslySetInnerHTML. The \_\_html key is required by React for security reasons to make sure you intentionally use dangerouslySetInnerHTML. The value of \_\_html is the result of getLatestNotification(), which presumably returns HTML content.   It's important to exercise caution when using dangerouslySetInnerHTML because rendering untrusted content without proper sanitization can pose security risks. Always ensure that the content you're rendering via dangerouslySetInnerHTML is coming from a trusted source and has been sanitized to prevent any malicious scripts from being executed in your application.  **npm registry**  The npm registry is a repository of packages for Node.js, JavaScript, and frontend development. It's a central place where you can find and access various packages and libraries published by developers.  To search for a package like enzyme-adapter-react-18 in the npm registry, you can use the npm website or the command line interface.  **Using the npm Website:**   1. **Visit the npm website:** Go to [npmjs.com](https://www.npmjs.com/). 2. **Search for the package:** In the search bar at the top, type enzyme-adapter-react-18 and press Enter.  Using Command Line: If you want to search for the package via the command line interface, you can use the npm search command:  bash  **npm search enzyme-adapter-react-18** |  |

|  |  |
| --- | --- |
| **Test using Enzyme and Jest** | |
| Enzyme provides a variety of methods and properties that allow you to interact with and assert against rendered React components. Here are some widely used methods and properties when working with Enzyme wrapper instances:  General syntax for test:  // group tests using describe  Describe(“component name” , ()=> {  Test (“test name description”, ()=> {  Const wrapper = shallow(<Nav />);  Expect( wrapper.find(‘nav’ ).toHaveLength (1));  Expect(wrapper.find(‘nav’). **toBeGreaterThan(0));**  });  });  **Methods for querying and finding elements:**   1. **.find(selector)**: Searches for nodes that match the provided CSS selector within the current wrapper's nodes. 2. **.contains(selector)**: Checks if the current wrapper contains a node matching the given selector. 3. **.exists()**: Checks if any nodes are found in the current wrapper. 4. **.text()**: Retrieves the concatenated text of the current wrapper's nodes.   **Methods for simulating events and interactions:**   1. **.simulate(event[, ...args])**: Simulates events (e.g., click, change) on elements within the wrapper. 2. **.prop(propertyName)**: Retrieves the value of a prop of the first node in the current wrapper.   **Methods for component state and instance:**   1. **.state()**: Retrieves the state of the component in the current wrapper. 2. **.instance()**: Retrieves the instance of the component being rendered by the wrapper.   **Properties for traversal and manipulation:**   1. **.length**: Gets the number of nodes in the current wrapper. 2. **.first()**, **.last()**: Gets the first or last node in the current wrapper. 3. **.children()**, **.parent()**: Gets the children or parent of the nodes in the current wrapper.   **Debugging and information:**   1. **.debug()**: Logs the current wrapper's HTML structure for debugging purposes.   These methods and properties allow you to explore, query, interact with, and assert against the rendered components when writing tests using Enzyme. They're handy for simulating events, accessing component state, finding elements, and validating component behavior in your tests. | **Methods for querying and finding elements:**   1. **.find(selector)**:   javascript   const wrapper = shallow(<Header />);  const logo = wrapper.find('.logo'); // Assuming .logo is a class of the logo element   **.contains(selector)**:  javascript   const wrapper = shallow(<Header />);  const hasLogo = wrapper.contains('.logo'); // Checks if .logo element exists within the wrapper   **.exists()**:  javascript   const wrapper = shallow(<Header />);  const exists = wrapper.exists(); // Checks if any nodes exist within the wrapper   **.text()**:  javascript   1. const wrapper = shallow(<Header />); 2. const headerText = wrapper.text(); // Retrieves text content of the wrapper   **Methods for simulating events and interactions:**   1. **.simulate(event[, ...args])**:   javascript   const onClickMock = jest.fn();  const wrapper = shallow(<Button onClick={onClickMock} />);  wrapper.find('button').simulate('click');  expect(onClickMock).toHaveBeenCalled(); // Simulates a click event on the button and checks if the onClick function was called   **.prop(propertyName)**:  javascript   1. const wrapper = shallow(<Header title="Welcome" />); 2. const titleProp = wrapper.prop('title'); // Retrieves the value of the 'title' prop   **Methods for component state and instance:**   1. **.state()**:   javascript   const wrapper = shallow(<Counter />);  wrapper.setState({ count: 5 });  const state = wrapper.state(); // Retrieves the component's state   **.instance()**:  javascript   1. const wrapper = shallow(<Header />); 2. const instance = wrapper.instance(); // Retrieves the instance of the rendered component   **Properties for traversal and manipulation:**   1. **.length**:   javascript   1. const wrapper = shallow(<List items={['apple', 'banana', 'orange']} />); 2. const itemsCount = wrapper.find('li').length; // Gets the number of <li> elements 3. **.first()**, **.last()**:   javascript  const wrapper = shallow(<List items={['apple', 'banana', 'orange']} />);  const firstItem = wrapper.find('li').first();  const lastItem = wrapper.find('li').last();   1. **.children()**, **.parent()**:   javascript  const wrapper = shallow(<Header />);  const children = wrapper.children(); // Retrieves the children of the wrapper  const parent = wrapper.parent(); // Retrieves the parent of the wrapper  **Debugging and information:**   1. **.debug()**:   javascript  const wrapper = shallow(<Header />);  console.log(wrapper.debug()); // Logs the HTML structure of the wrapper for debugging |

|  |  |
| --- | --- |
| **Deploy React app to github pages**   1. Install git hub pages   **Npm install gh-pages –save-dev**   1. Add homepage: to package.json   **"homepage": "http://thomaskitaba.github.io/alx-practice/repository",**   1. Add this to script element of the package.json file.   **"deploy": "gh-pages -d build"**  **NB: build folder should be created using npm run build.** | When encountering a problem use this option to Reinstalling package.json  Step 1: **npm cache clean --force**  Step 2: **rm -rf node\_modules**  Step 3: **npm install**  **polyfill**  polyfill is a piece of code (usually a JavaScript library or script) that provides modern functionality on older browsers or environments that don't natively support it. It essentially "fills in" the gaps by implementing features or APIs that might not be available in older versions of browsers or in certain environments. |
|  |  |

|  |  |
| --- | --- |
| **React Classes** | |
| **Syntax for react classes**  import React, { Component } from 'react';  class MyComponent extends Component {  constructor(props) {  super(props);  // Initialize state or perform other setup tasks  this.state = {  // Initialize state properties  };  }  // Class methods  render() {  return (  // JSX representing the component's UI  <div>  {/\* JSX content \*/}  </div>  );  }  }  export default MyComponent;  **Definition and explanation:**  Constructors: are special methods in object-oriented programming languages, including JavaScript, **used for initializing objects when they are created from a class or a constructor function.**  **React lifecycle methods** class YourComponent extends React.Component {  // componentDidMount() is called after the component is mounted to the DOM  componentDidMount() {  // Perform operations such as data fetching, subscriptions, etc.  // This is a good place to interact with APIs or set up event listeners.  }  // componentWillUnmount() is called before the component is unmounted and destroyed  componentWillUnmount() {  // Clean up tasks like removing event listeners or subscriptions  // This is the ideal place to perform cleanup operations to avoid memory leaks.  }  render() {  // Render the component's UI  return (  // JSX content for your component  );  }  } | main categories of React lifecycle methods:  **(mounting, updating, unmounting)** Mounting Phase  1. **constructor()**: This method is called when an instance of the component is being created. It's used for initializing state and binding event handlers. 2. **render()**: This method is required and it returns the JSX or React elements to be rendered on the screen. 3. **componentDidMount()**: **Executed immediately after the first render.** It's used for operations that require the component to be rendered on the DOM, such as data fetching from APIs.  Updating Phase  1. **shouldComponentUpdate()**: Allows control over the component's re-rendering. It determines whether the component should update or not by returning true or false. 2. **render()**: **Re-renders** the component with updated state or props. 3. **componentDidUpdate()**: Executes after the component's updates are flushed to the DOM. It's used for post-update operations like interacting with the DOM or making further API calls based on props changes.  Unmounting Phase  1. **componentWillUnmount()**: Invoked just before the component is removed from the DOM. It's used for cleanup, like removing event listeners or cancelling API requests.  Deprecated Lifecycle Methods Previous versions of React included methods like componentWillReceiveProps(), componentWillUpdate(), and componentWillMount(). These methods have been marked as deprecated and replaced with safer alternatives.  **Remember, with the introduction of React hooks, functional components can also have lifecycle-like behavior using hooks like useEffect().** |

|  |  |
| --- | --- |
| **React Classes** | |
| To convert the functional component Notifications into a class-based component, follow these steps:   1. **Create a Class Component:**    * Create a class that extends React.Component.    * Move the logic from the functional component's body into the class component's render() method. 2. **Manage State:**    * If necessary, handle state by using constructor() to initialize state and setState() to update it. 3. **Convert Props to Class Properties:**    * Convert props to class properties by accessing them using this.props. |  |

|  |  |
| --- | --- |
| 0x03 React components | |
|  | **Semantic Versioning (SemVer)**  Semantic Versioning (SemVer) rules:  ^15:8:11 == **MAJOR.MINOR.PATC**  ”react”: “^18.2.0”   * **^ : caret** when used in versioning like this ”react”: “^18.2.0” it indicates that you are willing to accept any minor or patch updates that don't increment the left-most non-zero digit in the version number. * **MAJOR:** **significant changes or backwards-incompatible updates.** 14.x.x to 15.x.x signifies a major version change. * **MINOR:** **minor updates or new features added in a backwards-compatible manner**. Adds functionality that doesn't break existing functionality. 15.7.x to 15.8.x signifies a minor version update * **PATCH:** Indicates a patch or **bug fix release without adding new features**. These increments don't alter the existing functionality. * 15.8.10 to 15.8.11 signifies a patch-level update. |
| NB: ReactDOM.render is no longer the recommended way to render your React application in React version 18 and above. So use this instead  // import React from 'react';  // import ReactDOM from 'react-dom';  // import App from './App/App';  // ReactDOM.render(  //   <React.StrictMode>  //     <App />  //   </React.StrictMode>,  //   document.getElementById('root')  // );  import React from 'react';  import { createRoot } from 'react-dom';  import App from './App/App'; // Assuming App is your main component  const root = createRoot(document.getElementById('root'));  root.render(<App />); | In React 18, certain APIs, including createRoot, have been moved to specific locations within the package to better organize and structure the codebase.  Some of them are:  Before the introduction of the createRoot API and concurrent mode in React 18, React primarily used a synchronous rendering approach with its ReactDOM.render method. In earlier versions, when updates occurred in a React application, they were processed synchronously, and the entire rendering process was treated as a single unit of work.  createRoot is optimized for concurrent rendering, enabling React to manage updates more effectively by breaking them into smaller units and prioritizing them based on their importance and relevance to the user. |

|  |  |
| --- | --- |
| **Named export and default export**  Example:   1. Named export   File-name: test.js  **Export const add = (a, b) => a + b;**  **When imposted:** we must use curly braces  **Import {add} from ‘.**/test’   1. Default export   File-name: test.js  **Export default const add = (a, b) => a + b;**  **When imported:** you can leave the braces  **Import add from:** ‘./test’  When you have a default export, you can import it without using braces {}: | **Syntax to set Props**   1. **Syntax to define: proptype**   // Inside your component definition:  ComponentName.defaultProps = {  // Define default values for props  propName1: defaultValue1,  propName2: defaultValue2,  // ...  };   1. Syntax to create new porpType:   The types can be either of this  (string, number, bool, function, object, array)  comopnentName.propTypes = {  propName1: propType.string.isRequired,  propName2: propType.bool  }  Example:  import React from 'react';  import PropTypes from 'prop-types';  function Greeting(props) { return <h1>Hello, {props.name}!</h1>; }  Greeting.propTypes = { name: PropTypes.string.isRequired, };  Greeting.defaultProps = {  name: 'Guest', // Default value if 'name' prop is not provided  };  function App() {  return (  <div>  **<Greeting name="John"** /> {/\* Valid usage \*/}  <Greeting name={42} /> {/\* Invalid usage: PropTypes warning will be shown \*/}  </div>  );  }  export default Greeting; |
|  |  |

|  |  |
| --- | --- |
| React immutable | |
| Immutable.Map provides a safe, predictable, and performant way to manage data in JavaScript applications. Its immutability simplifies state management, optimizes operations, and enables advanced features, making it a valuable tool for various situations.  **Immutability:** Maps cannot be directly modified. Any attempt to change a value creates a new Map with the desired updates. This ensures da   * While Immutable.Map shares similarities with JavaScript's built-in Map, it offers the added benefits of immutability, persistent updates, and optimizations for efficient data management.   **getIn()**  **getIn() function is specifically provided by the Immutable.js library. It's not a built-in JavaScript function.**   * **Purpose: Designed to efficiently access nested values within Immutable.Maps.** * **Syntax: map.getIn(path, notSetValue)** * **Immutability: Ensures that data structures remain unchanged during access.** * **Performance: Optimized for navigating nested structures efficiently.** * **Default Value: Allows providing a default value if a path doesn't exist.** * **Alternatives for Mutable Objects:**   **Dot notation or bracket notation for direct property access.**  **\_.get from Lodash for nested path access in mutable objects.** | * Purpose: Designed to efficiently access nested values within Immutable.Maps. * Syntax: map.getIn(path, notSetValue) * Immutability: Ensures that data structures remain unchanged during access. * Performance: Optimized for navigating nested structures efficiently. * Default Value: Allows providing a default value if a path doesn't exist.   **const mappedObj = fromJS({**  **company: {**  **employees: {**  **"001": { name: "Bob", role: "Developer" } } } });**  **const result = mappedObj.getIn(['company', 'employees', '001', 'role'], undefined);**  **console.log(result); // Output: "Developer"**  list()  Here's a breakdown of the code:  1. Importing the List Function:   * import { List } from 'immutable';   + This line imports the List function from the Immutable.js library, which is used for creating and working with immutable lists.   2. Creating an Immutable List:   * const originalList = List([1, 2, 3]);   + This line creates a new immutable list named originalList containing the elements 1, 2, and 3.   + Key points:     - It uses the List function to construct the list.     - The list is immutable, meaning its elements cannot be directly changed after creation.     - Any operations that seemingly modify the list actually create new lists with the desired changes.   example  console.log(originalList); // Output: List [1, 2, 3]  const updatedList = originalList.push(4); // Creates a new list with 4 added  console.log(originalList); // Still List [1, 2, 3] (original remains unchanged)  console.log(updatedList); // Output: List [1, 2, 3, 4] (new list with added element) |