Week 17 REACT

Library/Framework
Functional/Class
Jordan walke-facebook and react
React advantages, different backends, json data, client side rendering-not server side rendering
pwa- progressive web app
npm i create-react-library
npx create-react-app my-new-app
Webpack
Diffing
Jsx
export default and export
Store todos in local storage using custom hooks
Unique TODO’s
Class component vs functional components
Pure function
Conditional rendering
Fragment
Strictmode
Why does react run console.log() in component twice
Difference between framework and library
Why use useState rather than a js variable
Can i give a function inside useEffect
State management
Controlled vs uncontrolled components
React.memo
Synthetic events
Native events
Event pooling
Self closing tag
useReducer
useLayoutEffect

React.js Topics

React:

- React is an open-source JavaScript library for building user interfaces. It was developed and is maintained by Facebook and a community of individual developers and companies. React is designed to create reusable UI components that efficiently render and update when the application's state changes. It follows a component-based architecture, allowing developers to compose complex user interfaces from smaller, self-contained building blocks.

Hooks

- Used to give functional components an access to use the states and are used to manage side-effects in React. They were introduced in React 16.8. They let

developers use state and other React features without writing a class. For example- State of a component. It is important to note that hooks are not used inside the classes.

- Why the need for Hooks?
    - Use of ‘this’ keyword: The first reason has to do more with javascript than with React itself. To work with classes one needs to understand how ‘this’ keyword works in javascript which is very different from how it works in other languages. It is easier to understand the concept of props, state, and unidirectional data flow but using ‘this’ keyword might lead to struggle while implementing class components. One also needs to bind event handlers to the class components. It is also observed by the React developers team that classes don’t concise efficiently which leads to hot reloading being unreliable which can be solved using Hooks.
    - Reusable stateful logics: This reason touches on advanced topics in React such as Higher-order components(HOC) and the render props pattern. There is no particular way to reuse stateful component logic to React. Though this problem can be solved by the use of HOC and render props patterns it results in making the code base inefficient which becomes hard to follow as one ends up wrapping components in several other components to share the functionality. Hooks let us share stateful logic in a much better and cleaner way without changing the component hierarchy.
    - Simplifying complex scenarios: While creating components for complex scenarios such as data fetching and subscribing to events it is likely that all related code is not organised in one place are scattered among different life cycle methods.Hooks solve this problem by rather than forcing a split based on life-cycle method Hooks to let you split one component into smaller functions based on what pieces are related.
- Rules for using hooks
    - Only functional components can use hooks
    - Calling of hooks should always be done at top level of components
    - Hooks should not be inside conditional statements
- **useState**: useState is a hook used to manage state in functional components. It returns an array with two elements: the current state value and a function to update it. You can have multiple useState calls to manage different pieces of state.
- **useEffect**:
    - **useEffect** is a hook used for handling side effects in functional components. It allows you to perform tasks like data fetching, DOM manipulation, and subscription management after rendering. You can think of it as a combination of componentDidMount, componentDidUpdate, and componentWillUnmount in class components.
    - **Dependency list in useEffect**: In the useEffect hook, the dependency list is an array of values. When one or more of these values change between renders, the effect function is re-executed. It's used to control when the effect should run.
    - The **cleanup function** is executed when the component unmounts or when any of the specified dependencies change and trigger a re-render. This is useful for cleaning up resources, such as removing event listeners, cancelling API requests, or unsubscribing from observables.

- **useRef**: useRef is a hook used to create mutable references to elements or values that persist across renders. It's often used for accessing and interacting with DOM elements directly or for storing mutable values without causing re-renders.
- **useHistory vs. useNavigate**:
    - **useHistory** is typically associated with React Router. It provides access to the navigation history of your application. You can use it to programmatically navigate to different routes and manage the browser's history stack.
    - **useNavigate** is associated with the newer react-router-dom versions (v6 and later). It provides a simpler way to navigate between routes, especially in a scenario where you don't need access to the full history object.
    - The choice between useHistory and useNavigate depends on the version of React Router you're using and your specific navigation needs. useNavigate is more straightforward and is recommended for new projects with React Router v6 or later. useHistory provides more fine-grained control and is used with older versions of React Router.
- **useContext**: useContext is a React hook used to access the context values provided by a Context.Provider component higher up in the component tree. It allows components to consume values from the context without prop drilling.
- **useMemo vs. useCallback**:
    - useMemo memoizes a computed value, while useCallback memoizes a function. They are used in different situations based on what you want to memoizes.
    - **useCallback**: useCallback is a hook used to memoize functions, preventing them from being recreated on each render. It's helpful for optimising performance, especially when passing functions as props to child components.
    - **useMemo**: useMemo is a hook used to memoize the result of an expensive computation. It re-computes the value only when the dependencies in the dependency array change.
- **Custom Hooks**: Custom hooks are reusable functions that can encapsulate and abstract complex logic or state management in a clean and composable way. They are created by prefixing a function name with "use" and can be shared across different components.

Lifecycle Methods

- Lifecycle methods were a fundamental part of class-based components in React before the introduction of hooks in React 16.8. These methods allowed developers to perform various actions at different stages in the lifecycle of a component, such as when it was created, updated, or destroyed.
    - **Constructor**: The constructor method is called when an instance of the component is created. It's used for setting up initial state and binding event handlers.
    - **componentDidMount**: componentDidMount is invoked immediately after the component is inserted into the DOM. It's often used for initiating data fetching, setting up timers, or performing other side effects.
    - **componentDidUpdate**: componentDidUpdate is called after a component's state or props change and the component re-renders. It's often used for responding to changes in state or props, such as updating the DOM in response to new data.

    - **componentWillUnmount**: componentWillUnmount is called just before a component is removed from the DOM. It's used for cleaning up resources, such as cancelling network requests, stopping timers, or removing event listeners.
    - **shouldComponentUpdate**: shouldComponentUpdate is used to determine if a component should re-render after receiving new props or state. By default, it returns true, but you can implement custom logic to optimise rendering.
    - **componentWillReceiveProps** (**Deprecated**): componentWillReceiveProps was used to react to changes in props before rendering. However, it has been deprecated in favour of componentDidUpdate and is no longer recommended for use.

DOM concepts in Javascript

- A programming interface for web documents. It represents the page so that programs can change the document structure, style, and content dynamically. The DOM is essentially a structured representation of the page's content, such as HTML elements, text, attributes, and more. In this representation, everything is considered a node.
    - **Document**: The document object represents the entire HTML document in the browser. It serves as the entry point for interacting with the DOM. You can access elements, modify content, and manipulate the structure of the document through this object.
    - **Elements**: HTML elements are represented as objects in the DOM. You can access and manipulate these elements using JavaScript. Common methods for accessing elements include getElementById, querySelector, and getElementsByClassName.
    - **Node**: All elements in the DOM are nodes. The Node interface is a base interface for different types of nodes, including elements, text, comments, etc. Nodes are organised in a tree-like structure, with the document node at the top.
    - **Parent and Child Nodes**: Elements can have parent and child nodes. You can navigate the DOM tree by accessing an element's parent, children, and siblings.
    - **Attributes**: Elements can have attributes, which can be accessed and modified using JavaScript. Common attributes include id, class, src, href, etc.
    - **Events**: You can attach event listeners to elements to respond to user interactions like clicks, mouse movements, and keyboard inputs.
    - **Manipulating Content**: You can change the content of elements, including text and HTML, using properties like innerHTML, textContent, and innerText.
    - **Creating and Modifying Elements**: You can create new elements and add them to the DOM or replace existing elements.
    - **Styles and CSS**: You can change an element's style by accessing its style property or by adding/removing CSS classes.

Actual DOM and Virtual DOM

- You can access and interact with the Actual DOM using the browser's built-in developer tools, but you cannot directly view the Virtual DOM in the same way.
- The Virtual DOM is an abstract concept used by certain JavaScript libraries and frameworks, like React. It is not directly accessible or viewable in browser developer tools because it exists within the memory of the JavaScript runtime and is used as an

optimization technique for updating the Actual DOM. You work with the Virtual DOM indirectly through the library or framework's API and tools provided for that purpose.

- **Actual DOM:**
  - **What it is:**
  - The Actual DOM represents the current state of the web page's structure and content as a tree of objects.
  - It is a direct representation of the HTML structure of a web page.
  - It is created by the browser when a web page is loaded, and it can be accessed and manipulated using JavaScript.
  - **Performance Implications:**
  - Directly manipulating the Actual DOM can be slow and inefficient, especially when dealing with large and complex web pages.
  - Any change to the Actual DOM can trigger reflows and repaints, which can be computationally expensive and lead to performance bottlenecks.
  - **Usage:**
  - Developers use JavaScript to interact with and modify the Actual DOM to update the content or structure of a web page in response to user interactions or other events.
  - Typical methods for manipulating the Actual DOM include getElementById, querySelector, and innerHTML, among others.
- **Virtual DOM:**
  - **What it is:**
  - The Virtual DOM is an abstraction of the Actual DOM.
  - It is a lightweight, in-memory representation of the web page's structure and content.
  - It is not rendered to the screen directly but is used as an intermediary step when making changes to the web page.
  - **Performance Implications:**
  - Manipulating the Virtual DOM is faster than manipulating the Actual DOM because it is an in-memory data structure and doesn't directly affect the rendering of the page.
  - The Virtual DOM allows for efficient batch updates and minimal reflows and repaints.
  - **Usage:**
  - The Virtual DOM is primarily used in the context of virtual DOM libraries or frameworks like React. In React, changes are made to the Virtual DOM rather than the Actual DOM.
  - When changes are made to the Virtual DOM, it calculates the minimal set of changes required to update the Actual DOM, reducing the performance overhead.
- **Key Benefits** of Using the **Virtual DOM**:
  - Improved performance: Virtual DOM updates are more efficient because they minimise direct manipulation of the Actual DOM.
  - Easier development: Virtual DOM libraries provide a declarative way to describe how the UI should look based on the application's state.
  - Automatic reconciliation: Virtual DOM libraries handle the diffing and updating of the Actual DOM, ensuring that only necessary changes are applied.

Difference between single Page and multi-page application

- SPAs offer a more dynamic and responsive user experience but can be more complex to develop and may require additional SEO considerations. MPAs are simpler to implement and often more SEO-friendly but may not provide the same level of interactivity and performance as SPA.

	single Page application	**multi-page application**
Navigation and Loading:	In SPAs, the initial HTML, CSS, and JavaScript are loaded when the user visits the application. Afterward, most content updates and interactions happen dynamically without requiring full page reloads. SPAs use AJAX requests to fetch data from the server and update the content on the same page.	In MPAs, each page of the application is a separate HTML file. When the user navigates to a different page, a full page reload occurs. MPAs rely on traditional server-driven navigation.
User Experience:	SPAs provide a smoother and more responsive user experience because they eliminate the delay associated with full page reloads. Transition animations and client-side routing can be used to enhance the user experience further.	MPAs may have slightly slower initial load times due to full page reloads, but subsequent navigation within the application is straightforward.
Performance	SPAs can be faster for users once the initial load is complete because they only need to fetch and update the necessary data, not the entire page.	MPAs can be less efficient in terms of performance because they involve reloading entire pages, which can be slower compared to updating specific parts of a page in a SPA.
Complexity:	SPAs are typically more complex to develop and maintain due to the need for client-side routing, handling application state, and managing asynchronous data loading.	MPAs are often simpler to develop and maintain because they follow a more traditional server-side architecture.
Search Engine Optimization (SEO):	By default, SPAs can have SEO challenges because search engine crawlers may have difficulty indexing dynamically loaded content. SEO in SPAs often requires additional techniques like server-side rendering or prerendering to make content more accessible to search engines.	MPAs are generally more SEO-friendly out of the box because each page has its own URL and content, making it easier for search engines to index.
Examples:	React applications, Angular applications, and Vue.js applications are often implemented as SPAs.	Traditional websites with multiple distinct pages and minimal client-side JavaScript are often implemented as MPAs.

CSR vs SSR

- **Client-Side Rendering (CSR):**
  - In CSR, the web browser is responsible for rendering the web page content. The server typically sends a minimal HTML document and a JavaScript bundle to the client.
  - The browser then executes the JavaScript code, which fetches data from APIs and generates the DOM (Document Object Model) on the client-side.
  - CSR is often used in single-page applications (SPAs) and is known for providing a smooth and responsive user experience once the initial load is complete.
  - However, the initial load can be slower because it requires downloading and executing JavaScript before rendering the page. This can negatively impact SEO and user experience for users with slower connections or less powerful devices.
- **Server-Side Rendering (SSR):**

    - SSR involves the server rendering the web page content on the server-side before sending it to the client's browser.
    - When a user requests a page, the server generates the HTML content, including dynamic data, and sends a fully rendered HTML page to the client.
    - SSR is beneficial for SEO because search engines can easily crawl and index the content since it's present in the initial HTML response.
    - It can also provide a faster initial page load because the client receives pre-rendered HTML, reducing the need for additional client-side processing.
    - SSR can be more complex to implement than CSR, as it often requires server-side frameworks and technologies.
- In some cases, developers use a hybrid approach known as "**Hybrid Rendering**" or "**Universal Rendering**" to combine the benefits of both CSR and SSR. This approach involves rendering the initial page on the server (SSR) to improve SEO and initial load times, and then transitioning to CSR for subsequent interactions, which can provide a smoother user experience.

Dynamic rendering

- It refers to a technique used in web development and content delivery where web pages are generated and rendered on the server side in real-time in response to a user's request. This is in contrast to traditional static rendering, where web pages are pre-rendered and served as-is to users.
- **Server-Side Rendering (SSR)**: Dynamic rendering often involves server-side rendering (SSR), where the web server processes a user's request by generating the HTML for the requested page on the server itself. This allows for the creation of web pages that can be highly customised and personalised for each user.
- **Data Integration**: Dynamic rendering is particularly useful when web pages need to integrate with databases, external APIs, or other data sources to provide real-time information to users. This can include e-commerce sites, social media platforms, news websites, and more.
- **User Interactivity**: Dynamic rendering enables web applications to be highly interactive, as it allows for user input and actions to trigger server-side processes, which can update the web page without requiring a full page reload. This is often associated with technologies like AJAX (Asynchronous JavaScript and XML) and modern JavaScript frameworks like React and Angular.
- **SEO Benefits**: Dynamic rendering can be used to improve search engine optimization (SEO) for web applications. By rendering dynamic content on the server and serving pre-rendered HTML to search engine crawlers, web developers can ensure that search engines can index the content effectively, even if the website relies heavily on client-side JavaScript for interactivity.
- **Performance Considerations**: While dynamic rendering offers many advantages, it also requires server resources to generate pages on-the-fly, which can impact server performance. Developers need to optimise their server-side code to handle requests efficiently, especially during traffic spikes.
- **Caching**: To mitigate performance issues, caching mechanisms are often used with dynamic rendering. Frequently accessed pages or parts of pages can be cached to reduce the load on the server and improve response times.
- **Security**: Dynamic rendering must also take into account security considerations. Data validation and protection against common web vulnerabilities like cross-site

scripting (XSS) and SQL injection are essential when processing user input and generating dynamic content on the server.
- In summary, dynamic rendering is a web development approach that involves generating and rendering web pages on the server side in real-time, allowing for interactivity, data integration, and SEO benefits. It's a key technique for building modern, data-driven web applications.

Concept of reusability in react.js

- In React.js, reusability is a fundamental concept that encourages developers to create components that can be used in various parts of an application, making code more maintainable, efficient, and easier to scale.
- **Component Composition**: React promotes the composition of components, where you can create complex UI elements by combining smaller, reusable components. This allows you to build user interfaces by assembling and nesting components, making it easier to manage and maintain your codebase.
- **Props**: React components can receive data through props (short for properties). Props allow you to pass information from parent components to child components. By making components reliant on props rather than hard coding data, you increase their reusability because you can use the same component with different data in various parts of your application.
- **State**: While props are used for passing data from parent to child components, state is used to manage the internal state of a component. By making components self-contained and managing their state internally, you create components that are more reusable because they are less dependent on their parent components.
- **Functional Components and Hooks**: React introduced functional components and hooks, which encourage a more modular and reusable code structure. Functional components are easier to read and write, and hooks like useState and useEffect enable the management of state and side effects within functional components.
- **Custom Hooks**: React allows you to create custom hooks, which are reusable functions that encapsulate component logic. Custom hooks can be shared across multiple components and even between projects, enhancing code reusability.
- **Component Libraries**: You can create your own component libraries or use existing libraries like Material-UI, Ant Design, or Bootstrap to encapsulate UI elements and behaviours. This approach not only promotes reusability but also helps maintain a consistent design language across your application.
- **Higher-Order Components (HOCs)**: While less common with the advent of hooks, HOCs are a technique for enhancing the functionality of components. They can be used to add common features, such as authentication or data fetching, to multiple components, increasing their reusability.
- **Context API**: React's Context API allows you to share state and functions across multiple components without having to pass props down through intermediate components. This can simplify the management of shared state and make components more reusable.
- **Component Libraries and Component Sharing**: You can package your reusable components into libraries or share them across projects by publishing them to npm or using tools like Bit, Storybook, or Lerna. This makes it easy to reuse components across different applications.

Arrow Functions:

- Arrow functions are a concise way to write anonymous functions in JavaScript. They are commonly used in React.js for defining functions, especially when working with functional components. Arrow functions automatically bind the ‘this’ keyword to the surrounding context, which can be very helpful in React.
- const sayHello = (name) => `Hello, ${name}!`;

Map:

- The map() function is used to iterate over an array and create a new array with modified or transformed values. In React, you can use map() to render a list of components based on an array of data.
- const doubledNumbers = numbers.map((number) => number * 2);

Destructuring:

- Destructuring allows you to extract values from objects or arrays and assign them to variables. This syntax is commonly used in React to extract props or state values.
- function MyComponent({ prop1, prop2 }) {
  // Use prop1 and prop2 here
  }

Filter:

- The filter() function is used to create a new array containing elements that meet a specified condition. It's often used to filter data in React based on certain criteria.
- const evenNumbers = numbers.filter((number) => number % 2 === 0);

Reduce:

- The reduce() function is used to accumulate values in an array and return a single result. It's commonly used for operations like summing numbers or aggregating data.
- const sum = numbers.reduce((accumulator, currentValue) => accumulator + currentValue, 0);
- const filteredAndMapped = numbers
  .filter((number) => number % 2 === 0) // Filter even numbers
  .map((evenNumber) => evenNumber * 2) // Double each even number
  .reduce((acc, current) => acc + current, 0); // Sum the doubled even numbers

Import/Export:

- ES6 introduced a more structured way to import and export modules in JavaScript. In React.js, you can use import and export to manage your components, utilities, and other modules.
- Use export when you want to export multiple named values from a module, and you must specify the name when importing them.
- Use export default when you want to export a single default value from a module, and you can choose any name when importing it. Default exports are often used for the primary export of a module.

Difference between JSX and HTML

- **Purpose:**
  - **JSX**: JSX is primarily used in React.js and other JavaScript frameworks to describe the structure of user interfaces in a more declarative and component-based way. It allows you to embed JavaScript expressions within the markup, making it dynamic.
  - **HTML**: HTML, on the other hand, is a markup language used to structure and describe the content of web pages. It is the standard for web documents and is interpreted by web browsers to render web pages.
- **Syntax:**

    - **JSX**: JSX resembles HTML but allows the use of JavaScript expressions within curly braces {}. This enables dynamic content and logic within the markup.
    - **HTML**: HTML uses tags and attributes to define elements and their attributes, but it doesn't allow embedded JavaScript code.
- **Rendering:**
    - **JSX**: JSX needs to be transpiled into JavaScript using a tool like **Babel** before it can be rendered in the browser. React components written in JSX are rendered into the DOM by JavaScript.
    - **HTML**: HTML is directly understood and rendered by web browsers without the need for transpilation.
- **Components:**
    - **JSX**: JSX is often used to define React components. You can create custom reusable UI components using JSX.
    - **HTML**: HTML defines the structure and content of a web page but doesn't inherently support custom components like JSX does.
- **Attributes:**
    - **JSX**: In JSX, attribute names are written in camelCase, and you can use JavaScript expressions for attribute values.
    - **HTML**: In HTML, attribute names are typically written in lowercase, and attribute values are enclosed in double or single quotes.
- **Event Handling:**
    - JSX: JSX allows you to attach event handlers using JavaScript expressions.
    - HTML: In HTML, you can add event handlers using attributes like onclick and onchange.

Class components and **Functional components**

- **Class components** are a traditional way of defining components in React. They are implemented as JavaScript classes that extend the React.Component class. Class components have lifecycle methods that allow you to hook into different stages of a component's life cycle, such as when it is mounted, updated, or unmounted.
- **Functional components** is a newer way to define components in React and are based on JavaScript functions. They are simpler and easier to read than class components and have gained popularity due to the introduction of hooks, which allow you to use state and other React features in functional components.
- Functional components with hooks have become the preferred way of writing React components for many developers due to their simplicity and the ability to handle complex state and side effects. However, class components are still widely used, especially in existing codebases or for specific use cases where lifecycle methods are necessary. React provides backward compatibility and support for both class and functional components.

Concept of React router

- React Router is a popular library in the React ecosystem that allows you to add routing and navigation to your single-page applications (SPAs). It enables you to build multi-page applications within a single HTML page, providing a seamless user experience.
- React Router simplifies client-side routing in React applications, making it easier to create single-page applications with multiple views and a smooth user experience. It is a fundamental tool for building modern web applications with React.

- **<Route path="/about" component={About} />**
- component refers to the React component that should be rendered when the URL matches the specified path ("/about" in this case). It tells React Router which component to display when the user navigates to the "/about" route.\
- **Multiple Pages**: React Router allows you to create a multi-page-like experience within a single-page application (SPA). You can define different routes, each associated with a specific component, and navigate between these routes using links or programmatically.

Concept of event handlers

- Event handlers are functions in JavaScript that are used to respond to specific events that occur within a web page, such as user interactions. In React, event handlers are commonly used to handle user input and interactions in components.
- **onClick** is an event handler that is used to respond to a click event, typically associated with user interactions like clicking a button, link, or any clickable element. It is used to specify what should happen when the element is clicked.
- **onChange** is an event handler used to respond to changes in the value of an input element, like text input fields, checkboxes, radio buttons, or select dropdowns. It is commonly used for forms and capturing user input.
- Event handlers are essential for creating interactive and user-friendly web applications in React. They allow you to capture user input and respond to user interactions effectively. Depending on the specific use case, you can use various other event handlers to handle different types of events, such as onSubmit, onMouseOver, onKeyDown, etc., to create rich user experiences.

Attributes vs properties

- **Attributes**:
    - Attributes are part of HTML: Attributes are defined in the HTML markup and provide initial values or settings for an HTML element. They are specified as name-value pairs within the element's opening tag.
    - Static and initial values: Attributes represent static and initial values that are set when the HTML is initially parsed by the browser. These values are typically predefined and do not change unless the HTML source code is modified.
    - Attributes are strings: In the HTML source code, attribute values are always represented as strings. For example, class="my-class" is an attribute with the value "my-class".
    - Accessing attributes: In JavaScript, you can access the attributes of an HTML element using the getAttribute() method, and you can set attributes using the setAttribute() method.
- **Properties**:
    - Properties are part of the DOM: Properties are part of the Document Object Model (DOM) and represent the current state or value of an HTML element as it exists in the browser. They can change dynamically based on user interactions or script modifications.
    - Dynamic and mutable: Properties are dynamic and mutable, meaning their values can change during the lifecycle of a web page, unlike attributes, which are typically static.

    - Types vary: The type of a property can vary depending on the element and its purpose. Some properties, like .value on an input element, represent the current user input, while others, like .style, control the element's styling.
    - Accessing properties: In JavaScript, you can access an element's properties directly using dot notation or bracket notation.
- In summary, attributes are static values specified in the HTML markup, while properties represent the dynamic state of elements in the DOM. When interacting with HTML elements in JavaScript, it's crucial to be aware of whether you are working with attributes or properties, as they serve different purposes and behave differently.

Dynamic Import

- Dynamic imports are a feature in modern JavaScript (ES6 and later) that allow you to load modules asynchronously at runtime.
- The import() function is used to perform dynamic imports. It returns a Promise that resolves to the module's namespace object once the module is loaded.
- Dynamic imports are typically used for code splitting, lazy loading, and on-demand loading of modules.
- **Lazy Loading** with Dynamic Imports:
    - Lazy loading is a specific use case of dynamic imports. It involves deferring the loading of certain parts of your application until they are actually needed.
    - This is commonly used in single-page applications (SPAs) and large web applications to reduce initial load times.
    - You can use dynamic imports to lazily load components, routes, or resources as the user interacts with the application.
- **Code Splitting**:
    - Code splitting is a broader strategy that involves breaking your application code into smaller, manageable chunks or bundles.
    - Dynamic imports are often used for code splitting because they allow you to split your code into separate bundles and load them on-demand.
    - Code splitting helps reduce the initial bundle size, which can lead to faster page loads and improved performance.
    - Build tools like Webpack and tools associated with modern JavaScript frameworks often have built-in support for code splitting and dynamic imports.
- In summary, dynamic imports are a JavaScript feature that enables lazy loading and code splitting in web applications. They allow you to load modules or resources asynchronously, which can lead to better performance and a smoother user experience by reducing initial load times and optimising resource delivery.

PropTypes

- It is a library used in React to specify the expected types of props that a component should receive. It helps catch potential runtime errors by validating the props passed to a component during development. PropTypes can be used to define the data types, required props, and even custom validation logic for your components.

Scaffolding

- Scaffolding refers to the process of generating a basic directory structure, boilerplate code, and project files to kickstart the development of a web application or a specific feature within it. Scaffolding can save developers time and effort by providing a starting point with the necessary files and configurations already in place.

- While scaffolding can be a valuable time-saving technique, it's essential to understand the generated code and configurations, as well as to customise them to suit your project's specific requirements. Over reliance on scaffolding without a good understanding of the underlying technologies can lead to maintenance challenges down the road.