**Lecture 1: Creating a React Project with Create React App or Vite**

**1. React Project Creation Options**

* **Create React App (CRA)**
  + Command: npx create-react-app my-app
  + **Explanation**: create-react-app is a popular tool used to create React applications quickly. However, it can be **bulky** and **slow** due to the many unnecessary files and dependencies included in the setup.
  + **Downsides**: Takes time to install, has many default configurations and files you may not need for simple projects.
* **Vite (Bundler)**
  + Vite is a fast, lightweight alternative to Create React App. It skips some of the heavy initial setups, allowing for faster app creation.
  + Command: npm create vite@latest my-app
  + After creating a Vite app, you install dependencies with:
  + **Benefits**: Faster startup, minimal boilerplate, modern build tool. No unnecessary files are included at the start, and it's optimized for performance.

**2. React to React-DOM**

* **In a Web App**: React works with **React-DOM**, a package that manages how React renders components on a web page.
* **In a Mobile App**: For mobile development, you use **React Native** instead, which is tailored for mobile platforms (iOS and Android).

**3. npm vs npx**

* **npm**: Stands for **Node Package Manager**. Used to manage packages (install, update, uninstall) in your project.
* **npx**: Stands for **Node Package Executor**. Used to execute packages or commands without globally installing them. Example: npx create-react-app allows you to create a new React project without having to install the create-react-app globally.

**4. Package Management**

* **npm install** or npm i: Used to install project dependencies listed in package.json.

**5. package.json**

* **Purpose**: This file acts as the **entry point** for your project. It contains:
  + Project **name**, **version**, and **dependencies** (all the libraries and frameworks your project relies on).
  + **Scripts**: Commands to run the project (e.g., npm run start to start the project).
  + **Browserlist**: Specifies which browsers the project supports.

**6. Starting the Project**

* Command: npm run start / npm start
  + This starts the development server.

**Lecture 2: Understanding Project Structure**

**1. How React Integrates into HTML**

* **index.html**: This is the main HTML file where the React app gets injected. React essentially **ejects** into this HTML file and uses it as the foundation of the app.
* <noscript>: This tag inside the HTML file displays a message if JavaScript is disabled, advising users to enable it for the React app to work.

**2. React Virtual DOM**

* **Virtual DOM**: React creates a virtual copy of the DOM (Document Object Model). When changes happen in the app, React updates the virtual DOM first, compares it with the real DOM (using a process called **reconciliation**), and efficiently updates only the necessary parts of the real DOM.
* **Why Important?**: This ensures fast and optimized updates, enhancing app performance.

**3. JSX (JavaScript XML)**

* **JSX** allows you to write HTML-like syntax directly inside JavaScript.
* Example:

const element = <h1>Hello, world!</h1>;

* **Custom Tags**: You can create your own custom components and use them like HTML tags. React components are written in JSX and must start with a **capital letter** (e.g., <MyComponent />).
  + Example:

**function MyComponent() {**

**return <div>This is a custom component!</div>;**

**}**

**4. Other Important Files**

* **package-lock.json**: Locks the specific versions of all installed packages, ensuring consistent behavior across different environments.
* **manifest.json**: Used in Progressive Web Apps (PWA) or mobile apps to manage how the app is installed and looks when saved on a home screen.
* **web-vitals.js**: Tracks the app's performance, helping developers optimize loading times and responsiveness.

**Lecture 3: Understanding React's UI Update Mechanism**

**1. UI Update Problem**

* **Issue**:
  + We created a variable counter and updated its value through a function.
  + While the value of counter was updated (verified via the console), the **UI did not reflect** this change.
* **Why?**:
  + React doesn't automatically update the UI when a variable is changed.
  + React controls how and when the UI is updated, and it "reacts" to changes in specific variables.

**2. React's Approach to State Management**

* **React’s Power**: React decides when and how the UI updates. It provides **hooks** to handle variable changes that should reflect in the UI.
* **useState Hook**:
  + The useState hook is used to manage the state in a React component.
  + It ensures the state (like counter) and UI are kept in sync.
  + When the state changes, React re-analyzes the virtual DOM and updates only the necessary parts of the UI.

**Syntax**:

let [counter, setCounter] = useState(15);

* + - counter: The current state value.
    - setCounter: Function to update the state value.

**Example**:

**function Counter() {**

**const [counter, setCounter] = useState(15);**

**const increment = () => {**

**setCounter(counter + 1); // Updates both the state and the UI**

**};**

**return (**

**<div>**

**<p>Counter Value: {counter}</p>**

**<button onClick={increment}>Increment</button>**

**</div>**

**);ss**

**}**

In this example, the counter value updates in the UI every time the button is clicked.

**Conclusion**

* **useState** helps manage state and ensures that changes are reflected in the UI.
* React's virtual DOM handles the process efficiently, only updating the parts of the UI that need to be changed.

**Lecture 4** **Virtual DOM in React**

**1. Virtual DOM Concept**

* **Example (Doctor analogy)**:
  + Similar to how a doctor predicts and treats symptoms in advance, React predicts what changes will happen in the UI.
  + When React runs createRoot, it creates a **virtual DOM**, which is a copy of the actual DOM (Browser DOM).
  + The **virtual DOM** is synced with the real DOM and only necessary updates are pushed to the real DOM.

**2. How React Handles Updates**

* **Efficient Updates**:
  + Instead of making changes immediately after each update, React uses an optimized approach.
  + React will **batch updates** and apply only the necessary changes, avoiding unnecessary DOM manipulation.
  + This prevents frequent UI updates, saving performance.
* **Scenario**:
  + If multiple changes happen one after the other (e.g., a value changes multiple times in a short period), React won’t update the UI with each change.
  + Instead, React waits, consolidates the changes, and applies them efficiently (only updating the final change).

**3. Fiber Algorithm**

* **Purpose**:
  + The **Fiber algorithm** is used by React to manage virtual DOM updates efficiently.
  + Not every UI needs to update instantly, and Fiber prioritizes tasks based on importance (like animation vs. data updates).
* **Fiber Features**:
  + **Pause work**: Fiber can pause updates and return to them later.
  + **Assign priorities**: React can prioritize critical updates (e.g., animations) over less urgent ones (e.g., background data fetching).
  + **Reuse work**: Fiber can reuse completed work, saving time.
  + **Abort updates**: If an update is no longer relevant, Fiber can cancel it.

**4. Reconciliation in React**

* **Definition**:
  + Reconciliation is the algorithm React uses to **diff** the old virtual DOM tree with the new virtual DOM tree to determine the minimal set of changes needed for the real DOM.
* **Process**:
  + React generates a tree of nodes that describes the app (the **virtual DOM**).
  + When data changes (like setState), React generates a new tree.
  + React **diffs** the new tree with the previous one, finding the minimal changes.
  + These changes are applied to the real DOM, making the update efficient.
* **Key Concepts**:
  + **Component Types**: React assumes different component types generate completely different trees, so it replaces the old tree with the new one.
  + **List Diffing**: When rendering lists, React uses **keys** to efficiently track changes. These keys should be **stable, predictable, and unique**.

**5. Push vs Pull Approach**

* **Push-based (Manual)**:
  + You, the developer, decide how and when to schedule updates.
* **Pull-based (Automatic)**:
  + React is smart enough to decide when to update based on priorities (like animations vs. data updates).

**6. Key Takeaways:**

* React’s **virtual DOM** and **Fiber** ensure that only necessary changes are applied to the UI, saving performance.
* **Reconciliation** efficiently updates the real DOM by comparing new and old virtual DOM trees.
* Fiber introduces the ability to pause, prioritize, reuse, and cancel updates, making React more flexible in managing UI rendering.

**Lecture 5 Tailwind CSS and Props**

**Tailwind CSS Setup in a Vite Project:** To set up Tailwind CSS in a Vite project, follow the official guidelines provided in the [Tailwind documentation for Vite](https://tailwindcss.com/docs/guides/vite). This will guide you through the installation process. After setting up, you can easily use Tailwind components to style your project.

**Important Note on className:** In React, the class keyword is reserved for JavaScript classes, so we use className to apply CSS classes to HTML elements. For example:

<div className="bg-blue-500 text-white p-4">This is a Tailwind-styled div.</div>

**Using Props to Make Components Reusable:** Props (short for "properties") in React allow you to pass data between components, making them reusable. It is important not to segregate your code based on technology (HTML, CSS, JavaScript). For example, when creating a card component, the structure (HTML) and style (CSS) should be written together in the component. This allows you to easily pass dynamic data (like book details) through props and have the card content update accordingly.

**Where Do We Need Props?** Let’s understand with an example.

Consider an online bookstore. On the "View All Books" page, there will be many cards, each displaying details of different books. While the card layout (structure and style) remains the same, the content (book title, author, etc.) needs to change dynamically for each book. Instead of creating a separate card for each book, we can create a single reusable card component and use props to pass different book details to each card.

// Card Component (BookCard.js)

function BookCard({ title, author }) {

return (

<div className="border p-4 rounded shadow">

<h3 className="text-xl font-bold">{title}</h3>

<p className="text-gray-600">by {author}</p>

</div>

);

}

// Parent Component (BooksList.js)

function BooksList() {

const books = [

{ title: "The Great Gatsby", author: "F. Scott Fitzgerald" },

{ title: "1984", author: "George Orwell" },

];

return (

<div>

{books.map((book) => (

<BookCard key={book.title} title={book.title} author={book.author} />

))}

</div>

);

}

In this example, the BookCard component receives title and author as props, allowing it to display different content for each book without changing the layout.

**Props as Objects:** Props in React are always passed as objects. For instance, when passing a value to a component like this:

<BookCard title="The Great Gatsby" author="F. Scott Fitzgerald" />

React internally treats this as:

props = { title: "The Great Gatsby", author: "F. Scott Fitzgerald" }

**Providing Default Values for Props:** You can also provide default values for props in case no value is passed:

function BookCard({ title = "Unknown Title", author = "Unknown Author" }) {

return (

<div className="border p-4 rounded shadow">

<h3 className="text-xl font-bold">{title}</h3>

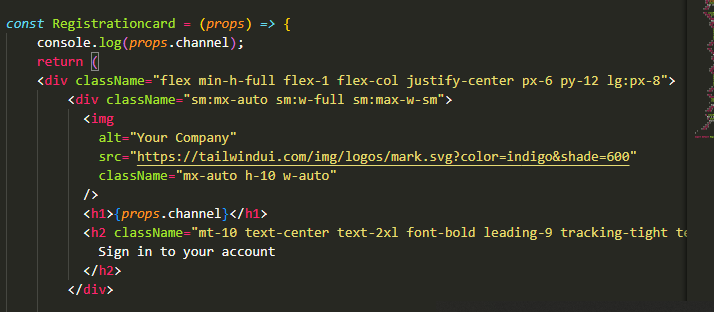
<p className="text-gray-600">by {author}</p>

</div>

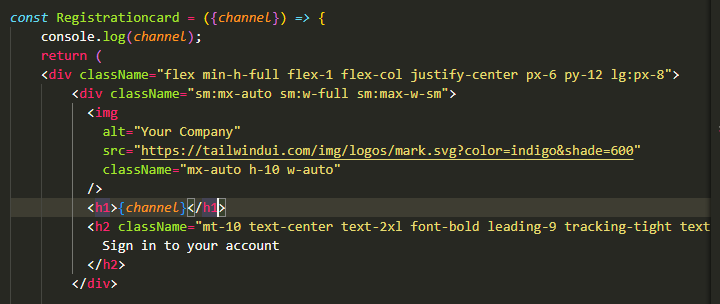
);

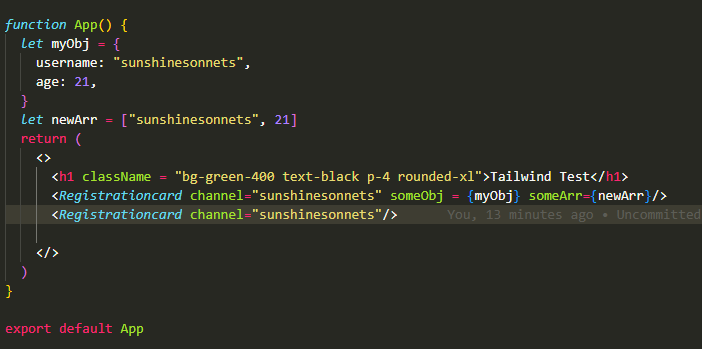
}

In this case, if no title or author is provided, the card will display "Unknown Title" and "Unknown Author" as default values.

Traditional syntax  


New method



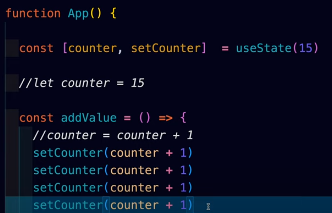
Value is coming from here  


**Lecture 6 interview question on counter**

Understanding Counter Updates in Concurrency

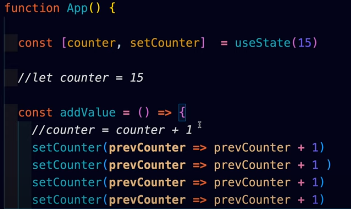
**Interview Question Breakdown**

1. **Initial Scenario**:
   * Counter starts at 15.
   * When addValue is called, the expectation might be that it directly updates to 19.
2. **Reality of Incrementing**:
   * Instead of jumping to 19, the counter will increment sequentially:
     + 16, then 17, then 18, and finally 19.
3. **Why This Happens**:
   * The addValue function is likely called in a concurrent environment.
   * Multiple threads may attempt to update the counter simultaneously.
   * Without proper handling, each thread may read the same initial value and attempt to increment it, leading to missed updates.
4. **Batch Processing**:
   * Algorithms that operate in batches can lead to all increments occurring at once, but still one at a time, resulting in intermediate values being skipped.



**Correct Update Approach**

* To ensure the counter can be set directly to a specific value:
  + Use a callback that takes the previous state into account. In this case, callback in setCounter is utilized.



prevCounter can be simply named counter in contexts where it's clear that it represents the previous state