React-2: Introduction to React Build Tools, lists , forms, props and state

title:Build Tools

1. build tools play a crucial role in transforming, bundling, and optimizing source code to make it ready for deployment
2. Some of the major use cases for build tools are
    a. Transpilation: React projects use modern JavaScript syntax (like JSX) and other language features (e.g., ES6/ESNext). Build tools like Babel transpile this code into a backward-compatible version that can run in older browsers. This integration is done in build tool
    b. Module Bundling: React applications are typically composed of multiple modules. Build tools like Webpack, Rollup, and Parcel bundle these modules into a single or multiple files, reducing the number of HTTP requests and improving loading times.
    c. Code Splitting: Large React applications can benefit from code splitting, where different parts of the application are loaded asynchronously. Build tools facilitate this by automatically analyzing dependencies and splitting the code accordingly.
    d. Optimization: Build tools optimize code for performance and efficiency. This includes minification (removing

    whitespace and comments), dead code elimination, and tree shaking (removing unused code).
    e. Development Server: Many build tools come with a built-in development server that provides features like hot module replacement (HMR), which allows developers to see changes in real-time without a full page reload.
    f.
3. Some of the majorly used Build tools are -
    a. Webpack
    b. VITE
    c. Parcel

Let's break down the differences between Webpack, Vite, and Parcel for building React applications in a way that's easy to understand We wont go into all the techical jargons as of now

Webpack:

1. Purpose: Webpack is a powerful module bundler that is widely used in React projects for bundling JavaScript files, CSS files, and other assets.
2. How it works: Webpack analyzes project dependencies and creates a dependency graph, then bundles them into optimized bundles.
3. Features: Supports code splitting, hot module replacement (HMR), and a wide range of loaders and plugins for handling various file types and optimizations.

CRA (Create React App):

1. Let us talk briefly about CRA which was a standard for setting up a react application
2. Purpose: CRA is a tool that helps you quickly create a new React project.
3. How it works: When you run a simple command (npx create-react-app my-app), it sets up a new project with all the necessary files and configurations. This includes setting up Webpack, Babel, and other tools behind the scenes.
4. Pros: It's beginner-friendly because everything is set up for you. You don't have to worry about configurations or installations.
5. Cons: Since it uses Webpack for bundling, it can be slower when dealing with large projects. Customizing it beyond the default settings can also be a bit challenging

Vite:

1. Purpose: Vite is a modern build tool specifically crafted for building React applications with speed and efficiency in mind.
2. How it works: Vite simplifies project setup by requiring minimal configuration. It utilizes ES modules (JavaScript's native module system) directly in the browser, speeding up development by serving files without the need for bundling during development.
3. Pros:

    a. Speed: Vite is exceptionally fast, even for large projects. It leverages native ES module support to deliver rapid development experiences.
    b. Hot Module Replacement (HMR): Vite supports HMR out of the box, allowing developers to see changes in real-time without reloading the entire page, enhancing productivity and workflow.
4. Cons:
    a. Newer Tool: Compared to Create React App (CRA), Vite is relatively newer in the ecosystem. While it's quickly gaining popularity, developers might encounter some limitations in terms of available resources or libraries, which are gradually catching up with its adoption.
5. Additional Clarifications:
    a. Beginner-Friendly: Vite's minimal configuration makes it beginner-friendly, enabling developers to get started quickly without being overwhelmed by complex setups.
    b. Developer Experience: Vite prioritizes developer experience by optimizing build times and providing features like HMR, making it an excellent choice for React projects where rapid iteration and feedback are crucial.

VITE EXTRA NOTES

What is ES Modules (ESM) and how it helps:

ES modules are the standardized module system for JavaScript, allowing developers to modularize their code and improve code organization.

Vite leverages ES modules directly in the browser during development. Instead of bundling all modules into a single file (as traditional build tools often do), Vite serves individual modules to the browser as separate files.

This approach eliminates the need for bundling during development, as the browser can handle loading and caching individual modules efficiently. As a result, developers can see their changes reflected in the browser almost instantly without waiting for a full bundle rebuild.

Speeding Up Development:
By serving files directly to the browser without bundling, Vite significantly speeds up the development process. Developers benefit from near-instantaneous feedback when making changes to their code.
This rapid feedback loop enhances developer productivity and workflow, allowing for faster iteration and experimentation during development

Parcel:

1. Purpose: Parcel is a bundler for JavaScript applications, including React.
2. How it works: Parcel automatically bundles your React app's assets (JavaScript, CSS, HTML) together into a smaller, optimized package. It also does this with minimal configuration.
3. Pros: It’s known for its simplicity and zero-config philosophy. It's also very fast, thanks to parallel processing and caching.
4. Cons: It may not be as feature-rich as Webpack, especially when handling more complex projects or customizations.

PARCEL EXTRA NOTES ON PARALLEL PROCESSING

Parcel utilizes parallel processing to handle multiple tasks simultaneously during the build process. This means that instead of executing tasks sequentially, Parcel distributes the workload across multiple CPU cores, allowing different parts of the build process to run concurrently.
By leveraging the computing power of multiple CPU cores, Parcel can perform tasks like dependency resolution, file parsing, and bundling in parallel, significantly reducing the overall build time.

This parallelization of tasks enables Parcel to scale effectively with the hardware it's running on, making it well-suited for projects of varying sizes, from small prototypes to large-scale applications with numerous dependencies.

Caching:
Parcel employs caching mechanisms to store intermediate build artifacts and dependencies between build runs. When a project is built, Parcel caches the results of various tasks, such as parsed files, compiled assets, and resolved dependencies.

On subsequent builds, Parcel checks the cache to determine if any files or dependencies have changed since the last build. If there are no changes, Parcel can reuse the cached results instead of reprocessing the same files, saving time and resources.

Caching is particularly beneficial during development iterations, where developers frequently make small changes to code. By only rebuilding what has changed and reusing cached results for unchanged files, Parcel can achieve faster incremental builds, providing quick feedback to developers.

Conclusion

In conclusion, Webpack offers powerful and highly customizable bundling capabilities, making it suitable for large-scale projects with complex requirements. However, it comes with a steeper learning curve and longer initial setup time.

Vite, on the other hand, prioritizes speed and developer experience with blazing-fast build times and minimal configuration. It's ideal for small to medium-sized projects where rapid iteration is crucial.

Parcel adopts a zero-config approach for easy project setup and fast build times, making it beginner-friendly and suitable for simplicity-focused projects. However, it may lack the customization options of Webpack.

Ultimately, the choice depends on project needs, with Webpack offering extensive customization, Vite emphasizing speed and developer experience, and Parcel prioritizing simplicity and ease of use.

We will be using Vite for our development purpose

React Development with Vite

In web development, we use HTML for content, CSS for styling, and JavaScript for interactivity. While these are great tools, as websites become more complex, managing all the moving parts can get tricky.

As web application gets more and more complex where lot of thing are happening on UI , doing all of this with only html, css and javascript can get daunting.

React, a library made by Facebook, makes things easier. It helps organize our code into reusable pieces called components, making it simpler to build and maintain websites. With React, we can create dynamic and interactive web pages more efficiently, making it a popular choice for modern web development

WHY VITE - https://vitejs.dev/guide/why

Setting Up Vite

Create a Vite Project:

Run the following commands in your terminal:

```
npm create vite@latest my-vite-app --template react
cd my-vite-app
npm install
```

Project Structure:

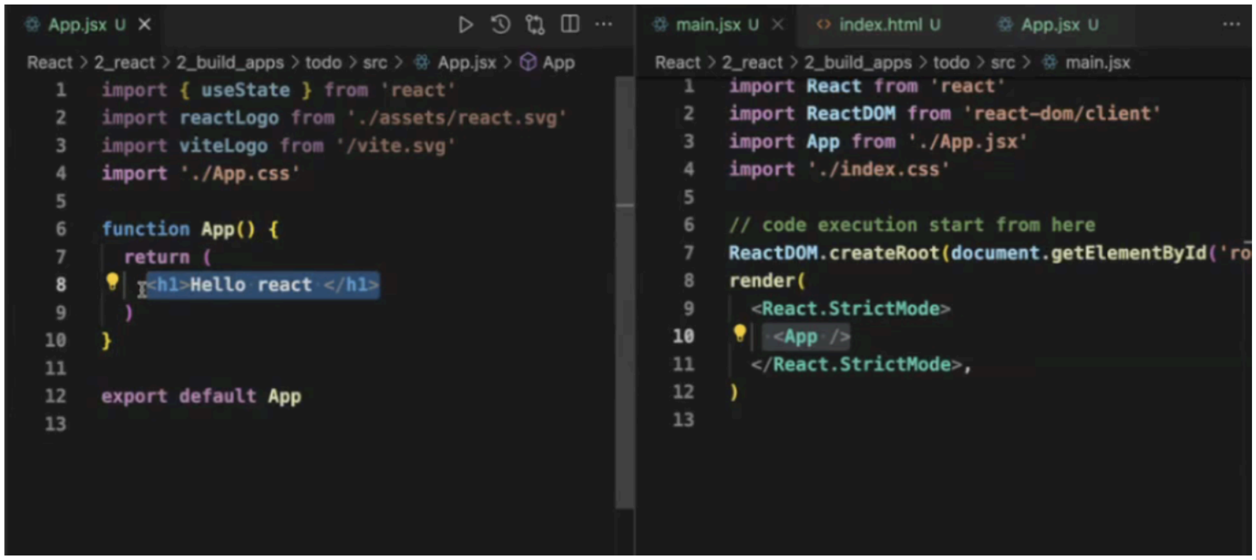
This will create a new React project using Vite, named "my-vite-app," and install the necessary dependencies.

Project Sctructure

The project structure will look like this:

1. package.json: Specifies project dependencies and metadata. It lists essential libraries like React, ReactDOM, Vite, and others, including various build and linting tools.
2. node_modules: A directory housing all installed packages and their dependencies, including development tools like Babel and ESLint etc.
3. index.html: The entry point for the web application. It contains a root <div> where ReactDOM will render the app's components. remember we created a root in our previous class as well
4. src: The source code directory containing the app's logic and components:
5. main.jsx: Initializes the ReactDOM rendering process by rendering the app component into the root <div> defined in index.html.
6. app.jsx: Houses the main application logic and structure, serving as the foundation for additional components and functionalities.
7. Installing Additional Tools:
    a. ES7 Extension: This extension, available for Visual Studio Code, provides quick access to React snippets,

streamlining component creation and other React-specific tasks.

8. Now Clean up the whole app.jsx file and write Hello React and see the output by running the app with the command npm run dev
9. 

App.jsx U
React > 2_react > 2_build_apps > todo > src > App.jsx > App
main.jsx U
index.html U
App.jsx U
React > 2_react > 2_build_apps > todo > src > main.jsx

```
import { useState } from 'react'
import reactLogo from './assets/react.svg'
import viteLogo from '/vite.svg'
import './App.css'

function App() {
  return (
    <h1>Hello react </h1>
  )
}

export default App
```

```
import React from 'react'
import ReactDOM from 'react-dom/client'
import App from './App.jsx'
import './index.css'

// code execution start from here
ReactDOM.createRoot(document.getElementById('ro
render(
  <React.StrictMode>
    <App />
  </React.StrictMode>,
)
```

title:Components and Reusablity of Components

1. Let's see How now to create a Component and use it
2. First you have to create a folder inside src directory let's name it components and create a file inside it by the name MyComponent.jsx , Now one thing to make sure of is always start a component's name with a Captial letter
3. Here's how you can create a React component named MyComponent and use it in your main application file app.jsx:
4. Create the MyComponent:

a. The component MyComponent is a simple functional component that returns an <h1> element with a greeting text.

```jsx
const MyComponent = () => {
   return <h1>Hello, I am a Component</h1>;
};

export default MyComponent;
```

5. use MyComponent in app.jsx:
    a. In the main application file, app.jsx, you can import MyComponent and render the output.

```jsx
import { useState } from "react";
import reactLogo from "./assets/react.svg";
import viteLogo from "/vite.svg";
import "./App.css";
import MyComponent from "./components/MyComponent"

function App() {
 const [count, setCount] = useState(0);

 return (
   <div>
     <MyComponent /> {/* First use */}
   </div>
 );
}

export default App;
```

6. Here, we import MyComponent into our main application file.
7. We create a root component App that returns a <div> containing one instance of MyComponent.

8. The application is rendered to the DOM using ReactDOM.render().
9. You can even Reuse this component again and again if you want
10. MOdify App.jsx

```
import { useState } from "react";
import reactLogo from "./assets/react.svg";
import viteLogo from "/vite.svg";
import "./App.css";
import MyComponent from "./components/MyComponent";

function App() {
 const [count, setCount] = useState(0);

 return (
   <div>
     <MyComponent /> {/* First use */}
     <MyComponent /> {/* Second use */}
     <MyComponent /> {/* Third use */}
   </div>
 );
}

export default App;
```

11. Reusability: By defining MyComponent separately and using it three times in the main app, you see the reusability and modularity of React components
12. But what use is Reusablity of if we cannot have a little dynamic nature in it. Extending the same example with props

title:Props in React

Props (short for properties) are a fundamental concept in React. They allow components to accept and use data that is passed to them from their parent component. This makes props a powerful tool for creating reusable, customizable components that can render different outputs depending on the data they receive.

Understanding Props:

1. Passing Data to Components:
    a. In functional components, props are accessed directly through the component's function signature.
2. Customizing Components:
    a. By accepting props, a component can behave differently depending on the data it receives. This allows you to create components that are highly reusable yet customizable.
    b. For example, a Button component could receive a label prop, which determines the text displayed on the button.
3. Reusability:
    a. Using props enables components to be reused with different configurations. This makes them a key part of React's component-based architecture, allowing for modular, maintainable code.

Modify MyComponent to Accept a Prop:

1. We'll introduce a message prop to MyComponent and use it to render different messages.

```
const MyComponent = ({ message }) => { // Accept the message prop
   return <h1>{message}</h1>; // Use the message prop in the JSX
};

export default MyComponent;
```

2. MyComponent is modified to accept a message prop. This prop is interpolated into an <h1> element to display different messages.
3. Reuse MyComponent with Different Prop Values in app.jsx:

```
import { useState } from "react";
import reactLogo from "./assets/react.svg";
import viteLogo from "/vite.svg";
import "./App.css";
import MyComponent from "./components/MyComponent";

function App() {
 const [count, setCount] = useState(0);

 return (
   <div>
     <MyComponent message="Hello, I am Component 1" />{" "}
     {/* First use with a different message */}
     <MyComponent message="Hi, I'm another Component" />{" "}
     {/* Second use with a different message */}
     <MyComponent message="Hey there, Component here!" />{" "}
     {/* Third use with a different message */}
   </div>
 );
}

export default App;
```

4. Benefits of Using Props:
    a. Customization: The message prop allows each instance of MyComponent to behave differently, making it customizable and flexible.
    b. Reusability: By utilizing props, the same component can be reused multiple times while allowing for different configurations.
    c. Separation of Concerns: MyComponent remains encapsulated and focused on its own functionality. This makes it easier to manage and extend both the component and the main application.
    d. Efficiency: This approach adheres to the DRY (Don't Repeat Yourself) principle, avoiding code duplication by using props to manage differences in behavior or content.

title:Pass Arrays and Objects as Prop and rendering a List

To create a React component that accepts an array and an object as props, and to use them effectively, let's consider a straightforward setup. We'll create a simple React application where the main component (App) renders a child component (DisplayData), passing an array and an object as props.

Step 1: Setting Up App.jsx

```
import "./App.css";
import DisplayData from "./components/Displaydata";

function App() {
 const fruits = ["Apple", "Banana", "Cherry"];
 const person = {
   name: "Alice",
   age: 25,
 };

 return (
   <div>
     <h1>React Props Example</h1>
     {/* Render the DisplayData component with props */}
     <DisplayData fruits={fruits} person={person} />
   </div>
 );
}

export default App;
```

DisplayData

```
const DisplayData = ({ fruits, person }) => {
   return (
     <div>
       <h2>Fruits List:</h2>
       <ul>
         {/* Render each fruit from the fruits array */}
         {fruits.map((fruit, index) => (
           <li key={index}>{fruit}</li>
         ))}
       </ul>
```

```
        <h2>Person Info:</h2>
       <p>Name: {person.name}</p>
       <p>Age: {person.age}</p>
     </div>
   );
 };
  export default DisplayData;
```

App.css

```
li{
 list-style-type: none;
}
```

How It Works:

App.jsx: It sets up an array (fruits) and an object (person), which are then passed as props to the DisplayData component. The rendering is initiated by calling ReactDOM.render.

DisplayData.jsx: This component takes fruits and person as props (using object destructuring in the function signature). Inside its render function:

It iterates over the fruits array using the map method, rendering each fruit as a list item.

It also displays the properties of the person object directly by accessing them as person.name and person.age.

The point to note is that when you need to write JS expressions inside JSX, we do it inside curly braces

title:Conditional Rendering in React

Conditional rendering in React is a way to render components or content conditionally based on a specific condition, such as a Boolean value or a function result. Let's walk through this process step by step, assuming you have a React setup already in place:

Step 1: Creating a Functional Component

1. Create a new file ConditionalRender.js

```
const ConditionalRendering = ({ isLoggedIn, username }) => {
   return (
     <div>
       {isLoggedIn ? (
         <h1>Welcome, {username}!</h1>
       ) : (
         <h1>Please log in to continue.</h1>
       )}
     </div>
   );
 }
  export default ConditionalRendering;
```

Explanation:
Props and Destructuring: Our component receives props: isLoggedIn and username. These props are then destructured directly in the function parameter list.

Conditional Logic: We use a ternary operator to decide what content to render:

If isLoggedIn is true, it renders a message welcoming the user.
If isLoggedIn is false, it renders a prompt asking the user to log in.

Step 2: Using the Component in the Main App

```
import "./App.css";
import DisplayData from "./components/Displaydata";
import ConditionalRender from "./components/ConditionalRender"

function App() {
 const fruits = ["Apple", "Banana", "Cherry"];
 const person = {
   name: "Alice",
   age: 25,
 };
 const isLoggedIn = true; // Change this to false to see the
conditional behavior

 return (
   <div>
     <h1>React Props Example</h1>
     {/* Render the DisplayData component with props */}
     {/* <DisplayData fruits={fruits} person={person} /> */}
      <ConditionalRender isLoggedIn={isLoggedIn}
username={person.name} />
   </div>
 );
}

export default App;
```

title:Simple Event handling in React

In React, event handling involves creating functions that respond to user actions, such as clicking a button. We'll create a simple React component that renders a button, and when that button is clicked, it will print "Button clicked" to the console.

Now, let's create our button component. Inside the src folder, create a new file called Button.jsx:

```
const Button = () => {
 const handleClick = () => {
   console.log("Button clicked");
 };

 return (
   <button onClick={handleClick}>Click me</button>
 );
};

export default Button;
```

In this code:

We define a functional component Button.

Inside the component, we define a function handleClick that will be called when the button is clicked. This function simply logs "Button clicked" to the console.

We render a <button> element and attach an onClick event handler to it. This event handler is set to call the handleClick function when the button is clicked.

Now, let's use this Button component in our main App component. Open App.js file in the src folder and replace its content with the following code:

```
import Button from "./components/Button"
```

```
<div>
     <h1>React Props Example</h1>
     {/* Render the DisplayData component with props */}
     {/* <DisplayData fruits={fruits} person={person} /> */}
     {/* <ConditionalRener isLoggedIn={isLoggedIn} username={person.name} /> */}
     <Button />
   </div>
```