Class 1 ReactJS

What is React?

It is an open source javascript library for building user interfaces.
Key Points from definition:

1. It is a Javascript library not Framework.
   Why?
   It does not include routing, state management, or build tooling by default.

   It gives you flexibility to pick your own tools (like Redux, React Router, etc.)

   Compare this to Angular (which is a full framework with batteries included).

2. It is used for building user interfaces.

Why learn React?

Popularity

- It is created and maintained by Facebook
- It has more than 200k stars on Github.
- It has Huge Community
- Based on the stackoverflow developer survey 2024, React is a popular choice of 39.5% developers.

Technically

- Component Based Architecture
  - UI as a Collection of Components
  - Reusability reduces Repetition
  - Easier debugging and Testing
- React is Declarative
  - Tell React what you want and React will build the actual UI

- React will handle efficiently updating and rendering if the components
- DOM updates are handled gracefully in React.

Prerequisites

1. HTML, CSS and Javascript fundamentals

Virtual DOM?

1. Why Virtual DOM?
   As web applications become more complex, managing updates to the user interface becomes a challenging task. This is where the Virtual DOM (Document Object Model) comes into play – particularly in React, the leading JavaScript library for building user interfaces.

2. What is Virtual DOM?
   The virtual DOM is a lightweight copy of the real DOM that allows React to manage changes more efficiently by minimizing the direct manipulation required on the real DOM.

3. How does Virtual DOM work?
   The virtual DOM is an in-memory representation of the real DOM elements. Instead of interacting directly with the real DOM, which can be slow and costly in terms of performance, React creates a virtual representation of the UI components. This virtual representation is a lightweight JavaScript object that mirrors the structure of the real DOM.

   Step by Step process of how the virtual DOM works:

   Step 1 – Initial Rendering: when the app starts, the entire UI is represented as a Virtual DOM. React elements are created and rendered into the virtual structure.

Step 2 – State and Props Changes: as the states and props change in the app, React re-renders the affected components in the virtual DOM. These changes do not immediately impact the real DOM.

Step 3 – Comparison Using Diff Algorithm: React then uses a diffing algorithm to compare the current version of the Virtual DOM with the previous version. This process identifies the differences (or "diffs") between the two versions.

Step 4 – Reconciliation Process: based on the differences identified, React determines the most efficient way to update the real DOM. Only the parts of the real DOM that need to be updated are changed, rather than re-rendering the entire UI. This selective updating is quick and performant.

Step 5 – Update to the Real DOM: finally, React applies the necessary changes to the real DOM. This might involve adding, removing, or updating elements based on the differences detected in step 3.

NPM Packages

npm (Node Package Manager) is like an app store for JavaScript packages — it provides reusable code written by other developers that you can use in your project instead of building everything from scratch.
In React, npm packages help you:

- Add extra features (like routing, animations, charts)
- Speed up development
- Use well-tested solutions

Create First React App

We use a tool called Create React App (CRA) to quickly set up a React project with all the necessary files, configuration, and dependencies - without manually doing it all.

Prerequisites:

1. Node.js installed (check node -v)
2. Npm installed (check npm -v)

Create React App Commands:

1. With npx:
   npx create-react-app *APP_NAME*

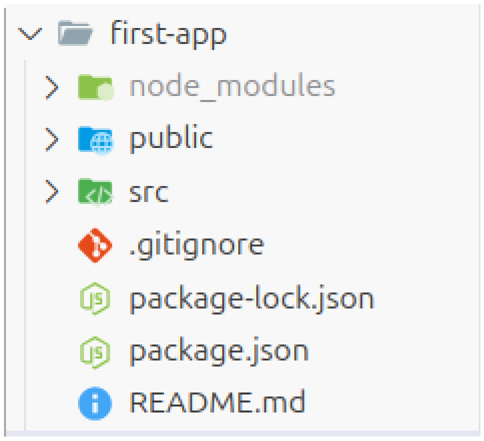
   npx: It is a npm package runner
2. With npm:
   npm i create-react-app -g //install the package globally

   create react-app *APP_NAME* // Creates a react application

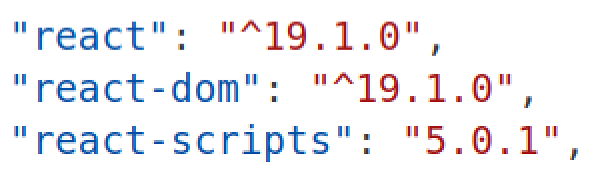
Run React App Command:

1. Go to your *react-app-folder*
2. Run *npm start*.
   It will start the app on http://localhost:3000/

React App Folder Structure

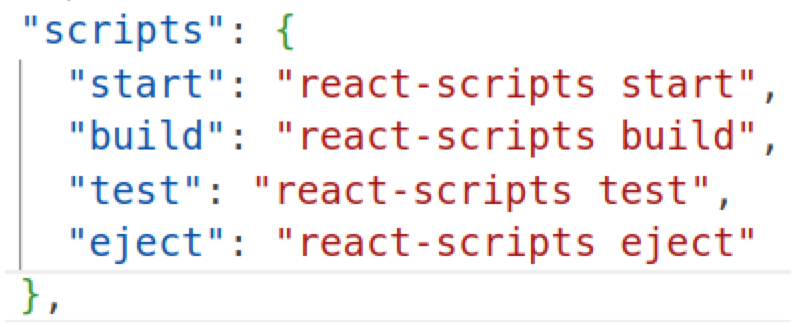


1. package.json: This file contains the dependencies and the scripts required to run the project.
   - Dependencies:



```
"react": "^19.1.0",
"react-dom": "^19.1.0",
"react-scripts": "5.0.1",
```

- Scripts:



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

2. package-lock.json: This file simply ensures the consistent installation of your app dependencies and its sub-dependencies.
   It contains:
   - Exact version of each dependency and sub-dependency
   - Resolved URLs where the packages were downloaded from
   - Integrity checks (to ensure security)

   Example:

   ```
   // package.json
   "dependencies": {
     "axios": "^1.5.0"
   }

   // package-lock.json
   "axios": {
           "version": "1.5.0",
           "resolved":"https://registry.npmjs.org/axios/-/axios-1.5.0.tgz",
            ...
    }
   ```

3. node_modules: This is the folder in which all the dependencies are installed required for the application to run. It is generated when you run the create-react-app command or npm install.
4. public folder:

This directory houses static assets that are served directly by the web server without being processed by tools like Webpack. Key files here include:

   a. index.html: The *index.html* is the only html file in your application. We are building single page applications and it is said that the view might dynamically change in the browser but it is this HTML file that gets served up.
   b. manifest.json: This file is for the progressive web app introduction in react-app.
   c. robots.txt: This file is for search engine optimization improvement.

5. src folder:
   The src folder (short for source) is the heart of your React app.
   It contains all the code you write — components, styles, logic, and more.
   a. index.js :
      i. The entry point of your React app
      ii. React uses this file to render the App component inside the root div of public/index.html.

      //ReactDOM.createRoot(document.getElementById('root')).render(<App />);
   b. App.js
      i. The main root component of your app.
      ii. Other components are usually imported and used here.
      iii. You can think of it as your app's "layout manager".
   c. App.css
      i. It contains classes which are applied in the app component
   d. App.test.js
      i. It contains a simple test case which is also called unit testing.
   e. Index.css
      i. It contains the styling of body tags
   f. serviceWorker.js

        i. It is concerned with progressive web apps and can be ignored for now.

Class 2 ReactJS

Components

ReactJS applications are built around a component-based architecture. A component in React "represents a part of the user interface." This modular approach allows for the breakdown of complex UIs into smaller, manageable, and self-contained pieces.

Key properties of React Components:

1. **Represent a Part of the UI:** Each component describes a specific section or element of the user interface. For example, an application might have components for a "header," "side nav," "main content," and "footer."
2. **Hierarchical Structure (Nesting):** Components can contain other components. A "root component" (often named App component) typically contains all other components, forming a tree-like structure. The source notes, "app component contains the other components."
3. **Reusability:** Components are designed to be reusable. The same component can be employed multiple times, potentially with "different properties to display different information." An example given is a "side nav component" being used for both a left and right side navigation.
4. **Building Blocks:** Components are fundamental to any React application.

Component Code Structure and File Naming

1. File Extension: .js and .jsx
2. Code Location: The "component code is usually placed in a JavaScript file." For instance, the App component is found in App.js.

Types of React Components

1. **Stateless Functional Components:**

Description: These are "literally JavaScript functions."
Functionality: Their main purpose is to "return HTML which describes the UI."
Example:

```
// HelloFunctional.js
import React from 'react';

function HelloFunctional() {
  return <h1>Hello, World!</h1>;
}

export default HelloFunctional;
```

Key Feature: They do not manage their own internal state (hence "stateless").

2. **Stateful Class Components:**
Description: These are "regular es6 classes that extend the Component class from the react library."
Requirement: They "must contain a render method."
Functionality: The render method "in turn returns HTML."
Example:

```
// HelloClass.js
import React, { Component } from 'react';

class HelloClass extends Component {
  render() {
    return <h1>Hello, World!</h1>;
  }
}

export default HelloClass;
```

Key Feature: They can manage their own internal state (hence "stateful"). The source indicates that the App component in the simple "hello world" example is a class component.

Functional Components

Functional components are a fundamental building block in ReactJS. They are defined as simple JavaScript functions that are responsible for describing a part of the user interface (UI).
Definition: "Functional components are just JavaScript functions."
Inputs: They "can optionally receive an object of properties which is referred to as props."
Outputs: They "return HTML which describes the UI." The tutorial clarifies that this "HTML is actually something known as JSX, but for the sake of understanding from a beginner's point of view let's just call it HTML."

Creating a Functional Component: Step-by-Step

1. *File Structure:* Create a new folder (e.g., components) and a new JavaScript file for the component (e.g., Greet.js).
2. *Naming Convention:* For component files, use "Pascal case" (e.g., Greet.js).
3. *Import React:* Every component file must import React.
   ```
   import React from 'react';
   ```
4. *Define the Function:* Create a JavaScript function that returns the UI.
5. *Initial Approach:*
   ```
   function Greet() {
           return <h1>Hello LPU</h1>;
   }
   ```
6. *Export the Component:* The component must be exported so it can be used in other parts of the application.

Exporting and Importing Components

There are two primary ways to export and import components: Default Exports and Named Exports.

1. Default Exports (Most Common)
   Definition: "This is what allows us to import the component with any name."
   Export Syntax: Prepend export default to the function or variable declaration.

   ```
   // In Greet.js
   const Greet = () ⇒ {
      return <h1>Hello LPU</h1>;
   };
   export default Greet;
   ```

   Import Syntax: Import the component without curly braces. The imported name can be different from the exported name.

   ```
   // In App.js
   import MyComponent from './components/Greet'; // 'MyComponent'
   can be any name
   // Usage: <MyComponent />
   ```

2. Named Exports
   Definition: "In this situation you have to import the component with the exact same name."
   Export Syntax: Prepend export to the function or variable declaration.

   ```
   // In Greet.js
   export const Greet = () ⇒ { // Note 'export' keyword
      return <h1>Hello LPU</h1>;
   };
   ```

   Import Syntax: Import the component using curly braces and the exact same name as it was exported.

   ```
   // In App.js
   import { Greet } from './components/Greet'; // Must be 'Greet'
   ```

// Usage: <Greet />

Attempting to import a named export as a default export will result in an "Attempted import error."

Including Components in the Application

Once a component is created and exported, it needs to be imported into another component (e.g., App.js) to be rendered.

1. Import:// In App.js
   import Greet from './components/Greet'; // Assuming default export
   // The '.js' extension can be omitted: import Greet from './components/Greet';
2. Usage (as a custom HTML tag):// In App.js's return statement
```
<div>
   <Greet></Greet>
   {/* Or, if no content between tags, use a self-closing tag: */}
   <Greet />
</div>
```

This process "allows us to include it in the app component." After saving and viewing in the browser, "your first functional component is up and running."

JSX

JSX, or JavaScript XML, is described as "*an extension to the JavaScript language syntax with the React library.*" It allows developers to "write XML-like code for elements and components" directly within their JavaScript files.

How Does JSX Work (Behind the Scenes)?

The JSX element is just syntactic sugar for calling React.createElement."

What is React.createElement?

The React.createElement method is the underlying JavaScript function that JSX sugarcoats. It accepts a minimum of three parameters:

1. *HTML Tag (as a string):* The type of HTML element to be rendered (e.g., 'div', 'h1').
2. *Optional Properties (as an object):* A JavaScript object containing key-value pairs representing attributes for the element (e.g., { id: 'hello', className: 'dummy-class' }). If no properties are needed, null is passed.
3. *Children (variable arguments):* Subsequent parameters represent the children of the element. These can be:
    a. *Plain text strings*.

*The React.createElement method can accept "*any number of elements as children.*"

Example Comparison (JSX vs. React.createElement):

This demonstrates the complexity of writing a simple "Hello LPU" component without JSX:

JSX Version:

```
<div>
  <h1>Hello LPU</h1>
</div>
```

React.createElement Version:

```
React.createElement(
  'div',
  null,
  React.createElement(
    'h1',
    null,
    'Hello LPU
  )
```

);
This comparison highlights how React.createElement can become "really clumsy" for components with many elements, underscoring the value of JSX.

Key Differences: JSX vs. Regular HTML Attributes

While JSX resembles HTML, there are crucial differences, primarily due to JavaScript reserved words and camelCasing conventions:

class vs. className:

1. In HTML, you use class for CSS classes.
2. In JSX, class is a JavaScript reserved word (used for defining classes/components). Therefore, you must use className for applying CSS classes.

for vs. htmlFor:

1. Similar to class, for is a JavaScript keyword.
2. In JSX, for form element labels (<label>), the for attribute is replaced by htmlFor.

CamelCasing for HTML Attributes:

1. HTML attributes are typically kebab-cased (e.g., onclick, tabindex).
2. In JSX, many attributes follow camelCasing convention (e.g., onClick, tabIndex). "We will see these differences as we progress through the series so don't worry about having to memorize them."

Introduction to Props

Props is a fundamental mechanism for passing data between components and making them dynamic and reusable.

Key Concepts:

1. *Optional Input:* Props are optional inputs that a React component can accept.

2. *Dynamic Components:* They allow components to be dynamic, meaning they can display different information based on the data passed to them.
3. *Reusability:* Props enhance component reusability.
4. *Immutable:* Props value cannot be changed within the component that receives them.
5. *Any DataTypes:* Props has the ability to pass different kinds of data, including primitive values (strings) and complex JSX/HTML structures

How Props Work

Props are specified as attributes when invoking a component, similar to HTML attributes.

Syntax: <ComponentName attributeName="value" />
Example: To pass a name property to a Greet component, you would write <Greet name="Munit" />.

Retrieving Props in Functional Components

1. *Add a Parameter*: The functional component's definition should include a parameter, conventionally named props. "You can actually name this anything you want to but the convention is to name it drops and I highly recommend you don't deviate from this."
2. *Use the Parameter:* Access individual prop values using dot notation (e.g., props.name, props.college, props.branch).

Example:
```
function Greet(props) {
   console.log(props); // Logs the props object to the console
   return <h1>Hello {props.name} from {props.college}</h1>;
}
```
JSX Evaluation: To display prop values within JSX, they must be wrapped in curly braces {}. This tells React to evaluate the expression rather than treating it as plain text.

Destructure Props:

Destructuring is a JavaScript ES6 feature that allows you to unpack properties from objects into distinct variables.

Instead of doing:
```
const studentName = props.studentName;
const college = props.college;
```

You can do:
```
const { studentName, college } = props;
```

Example:
```
function Greet(props) {
  const { studentName, college } = props;
  return (
    <h1>
      Hello {studentName}, from {college}
    </h1>
  );
}
```

Spread/Rest Operator (...):

The ... (spread/rest operator) is used to collect the remaining properties that were not destructured.

In our code:
```
const { studentName, college, ...prop } = props;
```

```
function Greet(props) {
  const { studentName, college, ...prop } = props; //destructure props
  return (
    <h1>
```

```
      Hello {studentName}, from {prop.branch} Branch, {college}
    </h1>
  );
}
```

Props Drilling:

Props drilling refers to the process of passing data from a top-level component to deeply nested child components via props - even if intermediate components don’t need that data.

Example:

Component Structure:
App → Parent → Child → Grandchild

Code:

```
// App.js
function App() {
  const studentName = "Nitesh";
  return <Parent studentName={studentName} />;
}

// Parent.js
function Parent({ studentName }) {
  return <Child studentName={studentName} />;
}

// Child.js
function Child({ studentName }) {
  return <Grandchild studentName={studentName} />;
}

// Grandchild.js
function Grandchild({ studentName }) {
  return <h2>Hello, {studentName}</h2>;
}
```