Any web application can be built using HTML, CSS & JavaScript.

HTML – Skeleton.

CSS – Beautifying or styling the skeleton but static.

JavaScript – Making it interactive.

Why React Js?

In order scale, maintain and easily build applications we need Library or framework like react to build cool things.

React was originally developed by Facebook (now Meta). It was created by Jordan Walke, a software engineer at Facebook, and was first deployed on Facebook's news feed in 2011 and later on Instagram in 2012. React was then released as an open-source project in May 2013.

React Js is a Javascript Library created for building fast and interactive user interfaces for web and mobile applications. It is opensource, Component based and front-end library responsible only for the application’s view layer.

React JS is a JavaScript library nor framework.

**Difference between Library and framework:**

**Library:**

1. **Definition**: A library is a collection of pre-written code that developers can use to optimize tasks.
2. **Usage**: Developers call library functions whenever they need to perform a specific task. The control remains with the application code.
3. **Flexibility**: Libraries are typically more flexible as developers have the freedom to choose which parts to use and how to use them.
4. **Examples**: jQuery (JavaScript library for DOM manipulation), momentJS. Loadash etc.

**Framework:**

1. **Definition**: A framework is a comprehensive set of guidelines and tools designed to facilitate the development of software applications.
2. **Usage**: Frameworks dictate the architecture of the application. They provide predefined structures and the developer writes code that fits into this framework.
3. **Inversion of Control**: Frameworks often implement the "Inversion of Control" (IoC) principle, where the framework calls the developer's code rather than the other way around. The control flow is managed by the framework.
4. **Examples**: Angular (JavaScript framework for building web applications), Django (Python framework for web development).

**Key Differences:**

* **Control**: In a library, the developer is in control and decides when and how to use the library. In a framework, the control is inverted, and the framework dictates the flow and structure of the application.
* **Structure**: Libraries provide specific functionality and are generally less opinionated about how they should be used. Frameworks are opinionated and provide a structured way to build applications.
* **Scope**: Libraries are typically focused on a narrow set of functionalities, whereas frameworks provide a broad set of functionalities and tools to build and manage an entire application.

**Open source:** An open-source library is a collection of code that is made freely available for use, modification, and distribution by anyone. Here are the key characteristics of an open-source library:

**Benefits:**

1. **Cost Efficiency**: Free to use, reducing development costs.
2. **Flexibility**: Can be modified to fit specific needs.
3. **Security**: Transparent code allows for public scrutiny and prompt identification of vulnerabilities.
4. **Innovation**: Encourages innovation through community contributions and collaborative development.
5. **Support**: Community-driven support through forums, mailing lists, and repositories.

Note: we need to be careful about the MIT licence because, there might be libraries not given licence for corporate use but only for personal use.

Model – View – Controllers

View is which we see or displayed to the user.

Model is the data which powers the view.

Controller is like bridge between Model and view.

Model-View-Controller (MVC) is a design pattern commonly used in web applications to separate concerns and organize code in a more manageable way. Here’s an overview of how MVC works in the context of web applications:

**Components of MVC:**

1. **Model**:
   * **Definition**: Represents the data and the business logic of the application. It directly manages the data, logic, and rules of the application.
   * **Responsibilities**:
     + Retrieve data from the database.
     + Perform data validation.
     + Apply business rules and logic.
     + Update the database with changes.
   * **Example**: In a blogging application, a Post model might handle retrieving, validating, and saving blog post data.
2. **View**:
   * **Definition**: Represents the UI (User Interface) of the application. It displays the data to the user and sends user commands to the controller.
   * **Responsibilities**:
     + Render data from the model into a user-friendly format (HTML, JSON, etc.).
     + Present forms and other interactive elements to the user.
3. **Controller**:
   * **Definition**: Acts as an intermediary between the Model and the View. It listens to the user input from the View, processes it (possibly updating the Model), and returns the output display to the View.
   * **Responsibilities**:
     + Handle user input.
     + Interact with the model to process data.
     + Pass the processed data to the view.
   * **Example**: In the blogging application, a Controller might handle requests to create, read, update, or delete blog posts.



Unidirectional data flow: We can send data form parent to child not from child to parent.

Angular is bidirectional data flow.

Basic React App:

<https://codepen.io/engineerchirag/pen/oNzJRVV>

JSX:

JSX (JavaScript XML) is a syntax extension for JavaScript that looks similar to XML or HTML. It is used with React to describe what the UI should look like.

**Key Features of JSX:**

1. **HTML-like Syntax**: JSX allows you to write HTML tags within JavaScript, making it intuitive to build UI components.
2. **Embedded Expressions**: You can embed JavaScript expressions within JSX using curly braces {}.
3. **Component Integration**: JSX enables you to integrate React components seamlessly.
4. **Transforms to JavaScript**: JSX is not valid JavaScript but is transformed into JavaScript code by tools like Babel.

**Example of JSX:**

Here's a simple example of how JSX is used in a React component:

import React from 'react';

function Greeting() {

const name = "John";

return (

<div>

<h1>Hello, {name}!</h1>

</div>

);

}

export default Greeting;

**Advantages of JSX:**

1. **Readability**: Combines HTML and JavaScript in a way that is easy to read and understand.
2. **Enhanced Developer Experience**: Familiar HTML-like syntax within JavaScript improves productivity.
3. **Integration with Tools**: Works seamlessly with development tools and frameworks, such as Babel and Webpack, to transform JSX into browser-compatible JavaScript.

**Transforming JSX to JavaScript:**

Under the hood, JSX is transformed into React.createElement() calls. For example, the above Greeting component is transformed into:

javascript

Copy code

import React from 'react';

function Greeting() {

const name = "John";

return React.createElement(

'div',

null,

React.createElement(

'h1',

null,

`Hello, ${name}!`

)

);

}

export default Greeting;

**Using JSX with React:**

JSX is a core part of the React ecosystem, enabling developers to define components more concisely and readably.

It promotes the separation of concerns by allowing the definition of UI elements alongside their logic and behaviour.

**Key Points to Remember:**

* **JSX Must Be Wrapped**: Adjacent JSX elements must be wrapped in an enclosing tag or React fragment (<></>).
* **JavaScript Expressions**: Use curly braces {} to embed JavaScript expressions within JSX.
* **Class vs. className**: Use className instead of class to define CSS classes in JSX, as class is a reserved keyword in JavaScript.

Overall, JSX enhances the React development experience by providing a clear and expressive way to define UI components.

NPX:

npx create-react-app is a command used to create a new React application with the help of the create-react-app tool. create-react-app is an officially supported way to create single-page React applications, and it offers a modern build setup with no configuration required. Using npx allows you to run the tool without having to install it globally on your system.

**What Happens When You Run npx create-react-app my-app:**

1. **Fetch and Execute**:
   * npx fetches the latest version of create-react-app from the npm registry and runs it.
   * This ensures you are using the latest version without needing to install it globally.
2. **Project Setup**:
   * A new directory named my-app is created.
   * Inside this directory, a new React application is set up with a standard structure.
3. **Install Dependencies**:
   * The necessary dependencies for a React project (like React, ReactDOM, Webpack, Babel, etc.) are installed automatically.
4. **Create Default Files and Structure**:
   * A standard project structure is created, including essential files like package.json, public/index.html, and src/index.js.

### react-scripts:

react-scripts is a package that is part of the Create React App (CRA) toolchain, which is used to set up a modern React application with zero configuration. It includes scripts and configuration used by Create React App to start, build, test, and eject your application.+

### Conclusion:

npx create-react-app simplifies the process of setting up a new React application by handling the configuration and dependency management for you. It is an excellent tool for both beginners and experienced developers to quickly bootstrap a new React project and start building applications.

**-> package.json :** file is a crucial component of any Node.js and JavaScript project. It serves as the manifest file for the project, containing metadata, dependencies, scripts, and configuration needed to manage the project. Here are the primary reasons why package.json is required:

**Key Functions of package.json:**

1. **Project Metadata**:
   * Contains essential information about the project such as its name, version, description, author, and license.
2. **Dependencies Management**:
   * Lists the packages (dependencies) required by the project to run, as well as development dependencies needed for building and testing the project.
   * Allows for easy installation and management of these dependencies using npm or yarn.

}

1. **Scripts**:
   * Defines custom scripts that can be run using npm or yarn. These scripts automate common tasks like starting the development server, building the project, running tests, and more.
2. **Version Control**:
   * Facilitates versioning and sharing of the project. By specifying dependencies with exact versions or version ranges, it ensures consistent behavior across different environments.
3. **Configuration**:
   * Holds configuration settings for various tools and modules. For instance, ESLint configuration, Babel presets, and more can be specified directly in the package.json.
   * Example:

json

Copy code

{

"eslintConfig": {

"extends": "react-app"

},

"browserslist": [

"defaults"

]

}

1. **Project Scripts and Lifecycle**:
   * npm scripts can hook into various lifecycle events of the package, such as preinstall, postinstall, prepublish, and postpublish.
   * Example:

json

Copy code

{

"scripts": {

"prestart": "echo 'Running before start'",

"start": "react-scripts start"

}

}

1. **Distribution**:
   * When publishing a package to the npm registry, package.json is used to provide all necessary details about the package, ensuring it is correctly listed and described on the registry.

**Benefits of package.json:**

1. **Consistency**:
   * Ensures that everyone working on the project uses the same versions of dependencies, avoiding the "it works on my machine" problem.
2. **Automation**:
   * Automates common tasks through npm scripts, reducing manual steps and potential errors.
3. **Easy Setup**:
   * Simplifies project setup for new developers. Running npm install or yarn install sets up the entire project environment as defined in package.json.
4. **Integration**:
   * Integrates with various tools and services (like CI/CD pipelines, deployment tools, etc.) by providing standardized scripts and configurations.

**Conclusion:**

The package.json file is an indispensable part of Node.js and JavaScript projects, acting as the central hub for project configuration, dependency management, and automation. It streamlines development workflows, ensures consistency across different environments, and facilitates easy setup and maintenance of projects.

The package-lock.json file plays a crucial role in managing dependencies for any Node.js project, including React applications. It complements the package.json file by providing a detailed snapshot of the entire dependency tree, ensuring consistent installations across different environments.

**Key Functions and Significance of package-lock.json:**

1. **Exact Dependency Versions**:
   * While package.json specifies version ranges for dependencies, package-lock.json locks the versions of every installed package, including nested dependencies.
2. **Consistency**:
   * Ensures that everyone on a team gets the exact same dependency versions when they run npm install, reducing the "works on my machine" issue.
   * Critical for continuous integration (CI) environments where consistent builds are necessary.
3. **Performance**:
   * Speeds up the installation process. Since package-lock.json includes the complete dependency tree, npm can quickly resolve and fetch the dependencies without needing to look up versions and resolve ranges.
4. **Security**:
   * Helps in auditing and managing security vulnerabilities. Tools like npm audit can use the lock file to identify and fix known vulnerabilities in the exact versions of dependencies you are using.
5. **Dependency Resolution**:
   * Provides a deterministic dependency resolution, meaning that the dependency tree will not change unless you explicitly update it. This predictability is vital for debugging and stability.
6. **Version Control**:
   * Should be committed to version control (e.g., Git) alongside package.json. This ensures that the entire team, as well as automated build systems, uses the exact same versions of dependencies.

**Interaction Between package.json and package-lock.json:**

* **package.json**:
  + Specifies the basic metadata and desired version ranges for dependencies.
  + Example:

json

Copy code

{

"dependencies": {

"react": "^18.0.0"

}

}

* **package-lock.json**:
  + Records the exact versions of dependencies and nested dependencies that were installed.
  + Ensures that subsequent installations reproduce the same dependency tree.

**Common Scenarios and Commands:**

1. **Initial Installation**:
   * When you run npm install for the first time, npm creates a package-lock.json file based on package.json.
2. **Adding a Dependency**:
   * When you add a new dependency using npm install <package>, npm updates both package.json and package-lock.json to reflect the new dependency.
3. **Updating Dependencies**:
   * When you update a dependency using npm update <package> or manually modify package.json and then run npm install, npm updates the package-lock.json to reflect any changes in the dependency tree.
4. **Removing a Dependency**:
   * When you remove a dependency using npm uninstall <package>, npm updates both package.json and package-lock.json to remove the dependency and update the tree accordingly.

**Conclusion:**

The package-lock.json file is vital for ensuring consistency, security, and performance in dependency management for React applications and other Node.js projects. It provides a detailed snapshot of the entire dependency tree, ensuring that installations are reproducible and predictable across different environments and setups.

A dependency tree in the package-lock.json file represents the structure and relationships between the packages (dependencies) that are installed in your project. It shows not only the direct dependencies of your project but also the dependencies of those dependencies (nested dependencies). This hierarchical structure ensures that the exact versions of all packages are recorded and can be consistently replicated.

**Structure of Dependency Tree in package-lock.json:**

1. **Top-Level Dependencies**:
   * These are the packages that are directly listed in your package.json file under dependencies or devDependencies.
2. **Nested Dependencies**:
   * These are the dependencies of the top-level dependencies. They can have their own dependencies, creating a nested structure.

**Example package-lock.json File:**

Let's consider a simple package.json file:

json

Copy code

{

"name": "my-app",

"version": "1.0.0",

"dependencies": {

"react": "^18.0.0",

"lodash": "^4.17.21"

}

}

After running npm install, a package-lock.json file is generated. Below is a simplified example showing a portion of the dependency tree:

{

"name": "my-app",

"version": "1.0.0",

"lockfileVersion": 2,

"requires": true,

"packages": {

"": {

"name": "my-app",

"version": "1.0.0",

"dependencies": {

"react": "^18.0.0",

"lodash": "^4.17.21"

}

},

"node\_modules/react": {

"version": "18.0.0",

"resolved": "https://registry.npmjs.org/react/-/react-18.0.0.tgz",

"integrity": "sha512-...",

"dependencies": {

"loose-envify": "^1.4.0",

"object-assign": "^4.1.1"

}

},

"node\_modules/lodash": {

"version": "4.17.21",

"resolved": "https://registry.npmjs.org/lodash/-/lodash-4.17.21.tgz",

"integrity": "sha512-..."

},

"node\_modules/loose-envify": {

"version": "1.4.0",

"resolved": "https://registry.npmjs.org/loose-envify/-/loose-envify-1.4.0.tgz",

"integrity": "sha512-...",

"dependencies": {

"js-tokens": "^4.0.0"

}

},

"node\_modules/object-assign": {

"version": "4.1.1",

"resolved": "https://registry.npmjs.org/object-assign/-/object-assign-4.1.1.tgz",

"integrity": "sha512-..."

},

"node\_modules/js-tokens": {

"version": "4.0.0",

"resolved": "https://registry.npmjs.org/js-tokens/-/js-tokens-4.0.0.tgz",

"integrity": "sha512-..."

}

},

**Breakdown of the Example:**

1. **Top-Level Packages**:
   * react and lodash are the top-level dependencies specified in package.json.
2. **Nested Dependencies**:
   * react depends on loose-envify and object-assign.
   * loose-envify in turn depends on js-tokens.
3. **Fields in package-lock.json**:
   * **version**: The exact version of the package installed.
   * **resolved**: The URL from where the package was fetched.
   * **integrity**: A hash ensuring the package's integrity.
   * **requires**: Lists the dependencies required by this package.

**Significance of Dependency Tree:**

1. **Ensures Consistency**:
   * By recording the exact versions and the entire dependency tree, package-lock.json ensures that every time npm install is run, the same versions of dependencies are installed. This prevents issues caused by updates to packages that might introduce breaking changes.
2. **Facilitates Debugging**:
   * Having a detailed dependency tree helps in debugging issues, as you can trace exactly which versions of packages are being used and where they are sourced from.
3. **Security and Integrity**:
   * The integrity field ensures that the package has not been tampered with and is exactly what was originally fetched. This enhances security by preventing compromised packages from being installed.
4. **Performance**:
   * Speeds up installation by avoiding version resolution on every install. Since package-lock.json provides the exact tree, npm can quickly fetch and install the packages.

**Conclusion:**

The dependency tree in package-lock.json is crucial for maintaining a consistent and reliable set of dependencies for your React application. It captures the entire structure of direct and indirect dependencies, ensuring that the same versions are installed across different environments, which helps in achieving consistency, security, and performance in your development workflow.

React-sripts: bundles all the src files in to single bundle.js file

<script defer src="[/static/js/bundle.js](http://localhost:3000/static/js/bundle.js)"></script></head>

Start Recat App:

const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(

  <h1>Hello World!</h1>

  // <React.StrictMode>

  //   <App />

  // </React.StrictMode>

);

In React, components can be exported and imported in different ways, depending on whether you want to export a single component as the default export or multiple components from a single file. Understanding export default and named exports allows for flexible and organized code management.

### export default:

When you use export default, you can export a single entity (usually a component) from a module. This entity can be imported without curly braces.

#### Example:

**Single Component File:**

Button.js:

jsx

Copy code

import React from 'react';

function Button(props) {

return <button>{props.label}</button>;

}

export default Button;

**Importing the Default Export:**

App.js:

jsx

Copy code

import React from 'react';

import Button from './Button';

function App() {

return (

<div>

<Button label="Click Me!" />

</div>

);

}

export default App;

### Named Exports:

Named exports allow you to export multiple entities from a single module. Each export must be imported using its exact name and with curly braces.

#### Example:

**Multiple Components File:**

Components.js:

jsx

Copy code

import React from 'react';

export function Button(props) {

return <button>{props.label}</button>;

}

export function Header(props) {

return <h1>{props.title}</h1>;

}

**Importing Named Exports:**

App.js:

jsx

Copy code

import React from 'react';

import { Button, Header } from './Components';

function App() {

return (

<div>

<Header title="Welcome!" />

<Button label="Click Me!" />

</div>

);

}

export default App;

### Combining Default and Named Exports:

You can combine default and named exports in a single file. This approach is useful when you want to export a primary component as the default export and additional components or utilities as named exports.

#### Example:

**Combined Exports File:**

MainComponent.js:

jsx

Copy code

import React from 'react';

export function SubComponent(props) {

return <div>{props.content}</div>;

}

function MainComponent(props) {

return (

<div>

<h2>Main Component</h2>

{props.children}

</div>

);

}

export default MainComponent;

**Importing Combined Exports:**

App.js:

jsx

Copy code

import React from 'react';

import MainComponent, { SubComponent } from './MainComponent';

function App() {

return (

<MainComponent>

<SubComponent content="This is a subcomponent" />

</MainComponent>

);

}

export default App;

### Summary:

* **export default**: Used for a single primary export from a module. Import without curly braces.
* **Named exports**: Used for multiple exports from a module. Import with curly braces.
* **Combining exports**: You can mix default and named exports in a single file for flexibility.

className="App-header"

Instead of class, className should be used in React JSX for css class names, since class is reserved word.

* CSS files and images can directly be imported.

**Expressions:**

**Key Differences:**

1. **Syntax**:
   * Angular: {{ expression }}
   * React: { expression }
2. **Language**:
   * Angular expressions are a subset of JavaScript, restricted to simple operations.
   * React expressions are full JavaScript, allowing complex logic and operations.
3. **Binding Context**:
   * Angular expressions are bound to the scope (AngularJS) or component context (Angular 2+).
   * React expressions are bound to the component's state and props.
4. **Evaluation and Updates**:
   * Angular handles expression evaluation and DOM updates via its digest cycle (AngularJS) or change detection (Angular 2+).
   * React handles expression evaluation and DOM updates using its virtual DOM and reconciliation process.

**Conclusion:**

Both Angular and React use expressions to bind data to the UI, but they differ in syntax, scope, and complexity. Angular expressions are more template-centric, while React expressions leverage the full power of JavaScript within JSX. Understanding these differences is key to mastering each framework and effectively building dynamic web applications

**Summary of Differences:**

1. **Syntax**:
   * **JavaScript**: Uses addEventListener or onclick attribute.

<button onclick="myFunction()">Click me</button>

* + **Angular**: Uses parentheses () for event binding in the template.

<button (click)="handleClick()">Click Me</button>

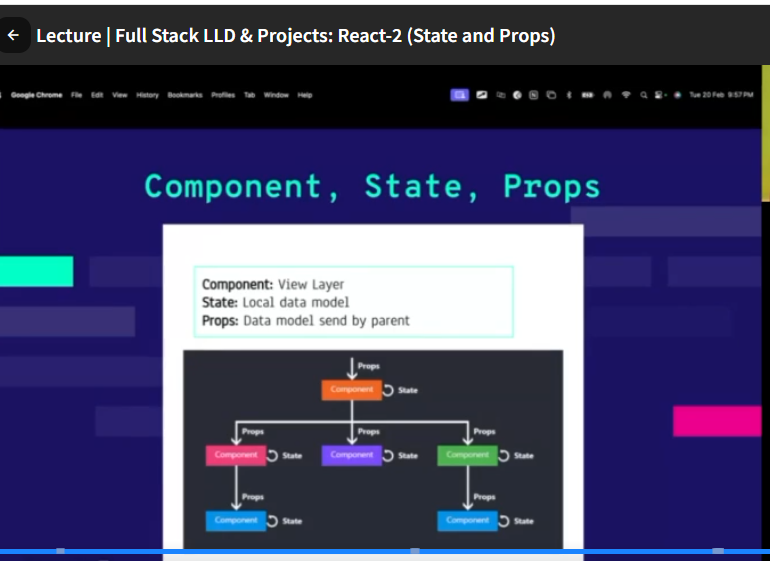
* + **React**: Uses camelCase onClick and passes event handlers as props.

<button onClick={handleClick}>Click Me</button>

1. **Event Handling Location**:
   * **JavaScript**: Event handlers are directly attached to DOM elements.
   * **Angular**: Event handlers are defined in the component class and referenced in the template.
   * **React**: Event handlers are defined within the component (as methods or functions) and passed as props.
2. **Scoping**:
   * **JavaScript**: Directly tied to the DOM element.
   * **Angular**: Tied to the Angular component’s context.
   * **React**: Tied to the React component’s context, with this in class components or functions in functional components.

**Conclusion:**

Event handling in JavaScript, Angular, and React involves different paradigms and syntax, reflecting the underlying design philosophies of each framework/library. JavaScript uses a direct approach with addEventListener, Angular uses template-based event binding, and React leverages JSX and props for handling events. Understanding these differences is crucial for working effectively within each environment.



**About useState()**

* 1. we have to use useState in react, without useState we can update variable.
  2. Dom rerenders only possible through props or state.
  3. Dont update state or prop directly.