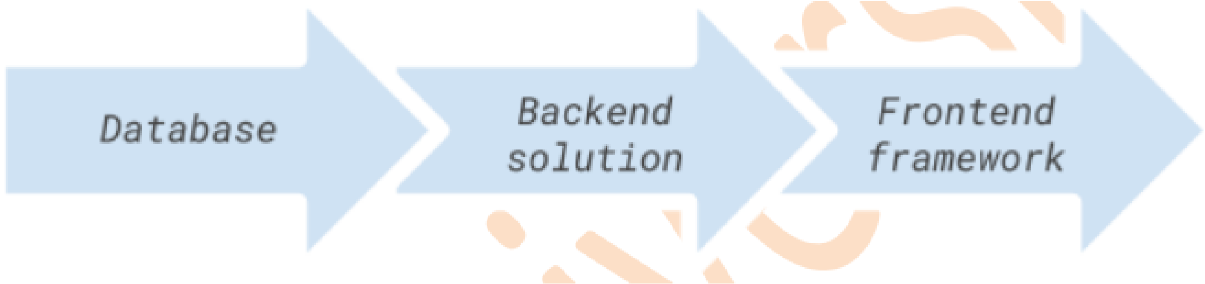
React

Introduction to Web Development

Web development stack is nothing but a set of tools typically used in tandem to develop web apps. It refers to the technologies that individual developer specializes in and use together to develop new pieces of software.
Web technology sets that include all the essential parts of a modern app are as follows: the **frontend framework**, the **backend solution** and the **database** (relational or document-oriented)



Introduction to stack:

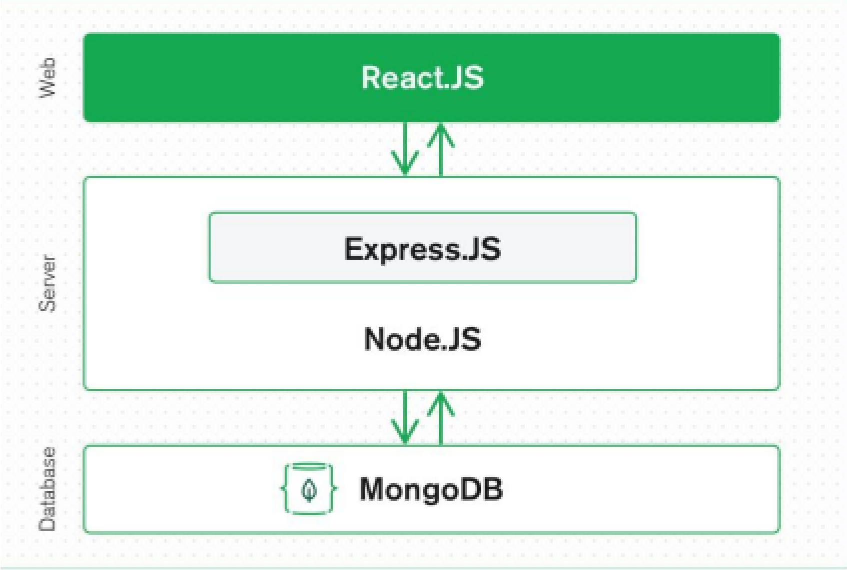
Any web application made by using multiple technologies. The combination of these technologies is called a “stack,” popularized by the LAMP stack, which is an acronym for Linux, Apache, MySQL, and PHP, which are all open-source components. As the web development world is continually changing, its technology stacks too changing. Top web development stacks are

- MEAN

- MERN
- Meteor.js
- Flutter
- The serverless Technology stack
- The LAMP technology stack
- Ruby on Rails Tech Stack

MERN stands for MongoDB, Express, React, Node

- MongoDB - Document database
- Express(.js) - Node.js web framework
- React(.js) - A client-side JavaScript library
- Node(.js) - The premier JavaScript web server Allows you to easily construct a 3-tier architecture (frontend, backend, database) entirely using JavaScript and JSON.



Why MERN?

- Ideally suited for web applications that have a large amount of interactivity built into the front-end.
- JavaScript Everywhere
- JSON Everywhere
- Isomorphic

REACT.JS

- Free and open source front end JS Library for building UI and UI Components.
- Maintained by Facebook, a community of individual developers and companies.
- This can be used as base in the development of Single page applications and mobile applications and is concerned with state management and rendering the state to DOM.
- It is the declarative JavaScript Library for creating dynamic client-side applications.
- Builds up complex interfaces through simple Components, connect them to data on your backend server, and render them as HTML.
- Provides support for forms, error handling, events and render them as HTML.

Key points about React.js

- Properties of React: Declarative, Simple, Component based, Supports server side, Mobile support, Extensive, Fast , Easy to learn

- Single way data flow
    - A set of immutable values are passed to the components renderer as properties in its HTML tags. The component cannot directly modify any properties but can pass a call back function with the help of which we can do modifications.
    - This complete process is known as “properties flow down; actions flow up”.
- Virtual DOM
    - Creates an in-memory data structure cache which computes the changes made and then updates the browser.
    - Allows a special feature that enables the programmer to code as if the whole page is rendered on each change whereas react library only renders components that actually change

Creation of React App

npm create vite@latest

- React is used for handling the view layer for web and mobile apps.
- Allows developers to create large web applications that can change data, without reloading the page.
- Also allows the creation of reusable UI components.
- The main purpose of React is to be fast, scalable, and simple. Works only on user interfaces in the application.

React Elements:

- The browser DOM is made up of DOM elements. Similarly, the React DOM is made up of React elements.
- DOM elements and React elements may look the same, but they are actually quite different.
- A React element is a description of what the actual DOM element should look like. In other words, React elements are the instructions for how the browser DOM should be created

Syntax: React.createElement(type,{props},children);

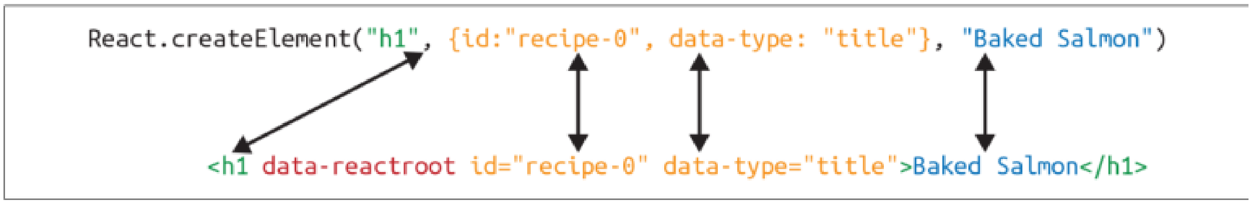
The first one is the type of element we're creating, in this case an <h1> tag. This could also be another React component. Second is the properties list in the form of objects. The third argument is the content of the element used in the first argument. We can create a React element to represent an h1 using React.createElement

```
React.createElement(“h1”,null,”Baked Salmon”)
```

When an element has attributes, they can be described with properties. Here is a sample of an HTML h1 tag that has id and data-type attributes.

```
React.createElement("h1", {id: "recipe-0", 'data-type': "title"}, "Baked Salmon")
```

Relationship between createElement and the DOM element



```
React.createElement("h1", {id:"recipe-0", data-type: "title"}, "Baked Salmon")

<h1 data-reactroot id="recipe-0" data-type="title">Baked Salmon</h1>
```

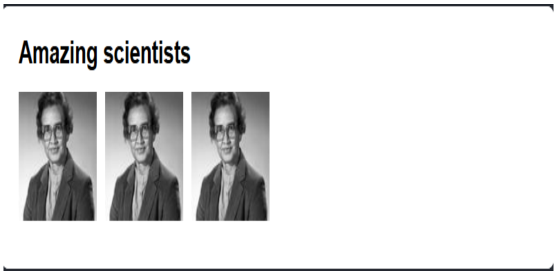
Note: data-reactroot will always appear as an attribute of the root ele- ment of your React component

React Components

React applications are built from isolated pieces of UI called components. A React component is a JavaScript function that you can sprinkle with markup. Example:

```
function Profile() {
  return (
    <img
      src="https://i.imgur.com/MK3eW3As.jpg"
      alt="Katherine Johnson"
    />
  );
}

export default function Gallery() {
  return (
    <section>
```

```
      <h1>Amazing scientists</h1>
      <Profile />
      <Profile />
      <Profile />
    </section>
  );
}
```



Amazing scientists

Importing and exporting components

You can declare many components in one file, but large files can get difficult to navigate. To solve this, you can export a component into its own file, and then import that component from another file:

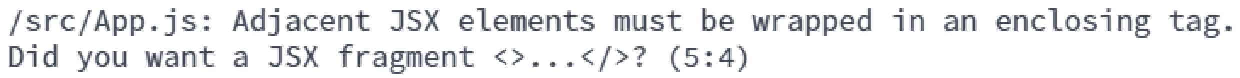
```
// Gallery.js

import Profile from './Profile.js';

export default function Gallery() {
  return (
    <section>
      <h1>Amazing scientists</h1>
      <Profile />
      <Profile />
      <Profile />
    </section>
  );
}

//Profile.js

export default function Profile() {
  return (
    <img
      src="https://i.imgur.com/QIrZWGIs.jpg"
      alt="Alan L. Hart"
    />
  );
}
```

Introduction to JSX markup

Each React component is a JavaScript function that may contain some markup that React renders into the browser. React components use a syntax extension called JSX to represent that markup. JSX looks a lot like HTML, but it is a bit stricter and can display dynamic information.

If we paste existing HTML markup into a React component, it won’t always work:

```
export default function TodoList() {
  return (
    // This doesn't quite work!
    <h1>Hedy Lamarr's Todos</h1>
    <img
      src="https://i.imgur.com/yXOvdOSs.jpg"
      alt="Hedy Lamarr"
      class="photo"
    >
    <ul>
      <li>Invent new traffic lights
      <li>Rehearse a movie scene
      <li>Improve spectrum technology
    </ul>
  );
}
```



```
/src/App.js: Adjacent JSX elements must be wrapped in an enclosing tag.
Did you want a JSX fragment <>...</>? (5:4)
```

This can be resolved by wrapping the code in the return with <></> or <div></div>

Javascript in JSX elements

JSX lets you write HTML-like markup inside a JavaScript file, keeping rendering logic and content in the same place. Sometimes you will want to add a little JavaScript logic or reference a dynamic property inside that markup. In this situation, you can use curly braces in your JSX to “open a window” to JavaScript:

```
const person = {
  name: 'Gregorio Y. Zara',
  theme: {
    backgroundColor: 'black',
    color: 'pink'
  }
};

export default function TodoList() {
  return (
    <div style={person.theme}>
      <h1>{person.name}'s Todos</h1>
      <img
        className="avatar"
        src="https://i.imgur.com/7vQD0fPs.jpg"
        alt="Gregorio Y. Zara"
      />
      <ul>
        <li>Improve the videophone</li>
        <li>Prepare aeronautics lectures</li>
        <li>Work on the alcohol-fuelled engine</li>
      </ul>
    </div>
  );
}
```

Passing props to a component

React components use props to communicate with each other. Every parent component can pass some information to its child components by giving them props. Props might remind you of HTML attributes, but you can pass any JavaScript value through them, including objects, arrays, functions, and even JSX!

```
// ChildComponent.jsx

import React from 'react';

function ChildComponent(props) {
  // Access props using dot notation on the 'props' object
  return (
    <div>
      <h2>Child Component</h2>
      <p>{props.greeting}</p>
      <p>Welcome, {props.user}!</p>
    </div>
  );
}

export default ChildComponent;



// ParentComponent.jsx

import React from 'react';
import ChildComponent from './ChildComponent';
```

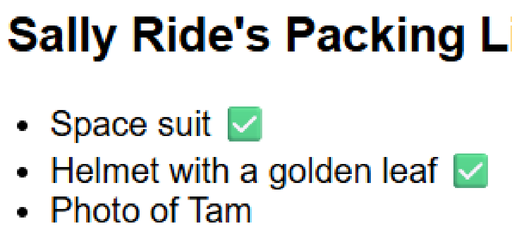
```
function ParentComponent() {
  const message = "Hello from Parent!";
  const userName = "Alice";

  return (
    <div>
      <h1>Parent Component</h1>
      <ChildComponent greeting={message} user={userName} />
    </div>
  );
}

export default ParentComponent;
```

Conditional Rendering

Your components will often need to display different things depending on different conditions. In React, you can conditionally render JSX using JavaScript syntax like if statements, && and ? : operators.

In this example, the JavaScript && operator is used to conditionally render a checkmark:

```
function Item({ name, isPacked }) {
  return (
    <li className="item">
      {name} {isPacked && '✅'}
    </li>
  );
}

export default function PackingList() {
  return (
    <section>
      <h1>Sally Ride's Packing List</h1>
      <ul>
        <Item
          isPacked={true}
          name="Space suit"
        />
        <Item
          isPacked={true}
          name="Helmet with a golden leaf"
        />
        <Item
          isPacked={false}
          name="Photo of Tam"
        />
      </ul>
    </section>
  );
}
```



Sally Ride's Packing List

- Space suit ✅
- Helmet with a golden leaf ✅
- Photo of Tam

Rendering Lists

You will often want to display multiple similar components from a collection of data. You can use JavaScript’s filter() and map() with React to filter and transform your array of data into an array of components.

For each array item, you will need to specify a key. Usually, you will want to use an ID from the database as a key. Keys let React keep track of each item’s place in the list even if the list changes.

```
const people = [{
  id: 0,
  name: 'Creola Katherine Johnson',
  profession: 'mathematician',
  accomplishment: 'spaceflight calculations',
  imageId: 'MK3eW3A'
}, {
  id: 1,
  name: 'Mario José Molina-Pasquel Henríquez',
  profession: 'chemist',
  accomplishment: 'discovery of Arctic ozone hole',
  imageId: 'mynHUSa'
}, {
  id: 2,
  name: 'Mohammad Abdus Salam',
  profession: 'physicist',
  accomplishment: 'electromagnetism theory',
  imageId: 'bE7W1ji'
}
}];
```

```
function getImageUrl(person) {
  return (
    'https://i.imgur.com/' +
    person.imageId +
    's.jpg'
  );
}

export default function List() {
  const listItems = people.map(person =>
    <li key={person.id}>
      <img
        src={getImageUrl(person)}
        alt={person.name}
      />
      <p>
        <b>{person.name}:</b>
        {' ' + person.profession + ' '}
        known for {person.accomplishment}
      </p>
    </li>
  );
  return (
    <article>
      <h1>Scientists</h1>
      <ul>{listItems}</ul>
    </article>
  );
}
```

Responding to events

React lets you add event handlers to your JSX. Event handlers are your own functions that will be triggered in response to interactions like clicking, hovering, focusing form inputs, and so on.

You can make it show a message when a user clicks a button by following these three steps:

1. Declare a function called handleClick inside your Button component.
2. Implement the logic inside that function (use alert to show the message).
3. Add onClick={handleClick} to the <button> JSX.

```
export default function Button() {
  function handleClick() {
    alert('You clicked me!');
  }

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

You defined the handleClick function and then passed it as a prop to <button>. handleClick is an event handler.

Event handler functions:

Are usually defined inside your components.
Have names that start with handle, followed by the name of the event.
By convention, it is common to name event handlers as handle followed by the event name. You’ll often see onClick={handleClick}, onMouseEnter={handleMouseEnter}, and so on.

Alternatively, you can define an event handler inline in the JSX:

```
<button onClick={function handleClick() {
  alert('You clicked me!');
}}>
```

Or, more concisely, using an arrow function:

```
<button onClick={() => {
  alert('You clicked me!');
}}>
```

Important:

Functions passed to event handlers must be passed, not called.

passing a function (correct)

```
<button onClick={handleClick}>
```

calling a function (incorrect)

```
<button onClick={handleClick()}>
```

The difference is subtle. In the first example, the handleClick function is passed as an onClick event handler. This tells React to remember it and only call your function when the user clicks the button.

In the second example, the () at the end of handleClick() fires the function immediately during rendering, without any clicks. This is because JavaScript inside the JSX { and } executes right away.

When you write code inline, the same pitfall presents itself in a different way:

passing a function (correct)

```
<button onClick={() => alert('...')}>
```

calling a function (incorrect)

```
<button onClick={alert('...')}>
```

Passing inline code like this won’t fire on click, rather, it fires every time the component renders:

```
// This alert fires when the component renders, not when clicked!
<button onClick={alert('You clicked me!')}>
```

If you want to define your event handler inline, wrap it in an anonymous function like so:

```
<button onClick={() => alert('You clicked me!')}>
```

Rather than executing the code inside with every render, this creates a function to be called later.

In both cases, what you want to pass is a function:

<button onClick={handleClick}> passes the handleClick function.
<button onClick={() => alert('...')}> passes the () => alert('...') function.

Reading props in event handlers

Because event handlers are declared inside of a component, they have access to the component’s props. Here is a button that, when clicked, shows an alert with its message prop:

```
function AlertButton({ message, children }) {
  return (
    <button onClick={() => alert(message)}>
      {children}
    </button>
  );
}

export default function Toolbar() {
  return (
    <div>
      <AlertButton message="Playing!">
        Play Movie
      </AlertButton>
      <AlertButton message="Uploading!">
```

```
        Upload Image
      </AlertButton>
    </div>
  );
}
```

Event Propagation

Event handlers will also catch events from any children your component might have. We say that an event “bubbles” or “propagates” up the tree: it starts with where the event happened, and then goes up the tree.

This <div> contains two buttons. Both the <div> and each button have their own onClick handlers. Which handlers do you think will fire when you click a button?

```
export default function Toolbar() {
  return (
    <div className="Toolbar" onClick={() => {
      alert('You clicked on the toolbar!');
    }}>
      <button onClick={() => alert('Playing!')}>
        Play Movie
      </button>
      <button onClick={() => alert('Uploading!')}>
        Upload Image
      </button>
    </div>
  );
}
```

If you click on either button, its onClick will run first, followed by the parent <div>'s onClick. So two messages will appear. If you click the toolbar itself, only the parent <div>'s onClick will run.

Stopping propagation

Event handlers receive an event object as their only argument. By convention, it's usually called e, which stands for "event". You can use this object to read information about the event.

That event object also lets you stop the propagation. If you want to prevent an event from reaching parent components, you need to call e.stopPropagation() like this Button component does:

```
function Button({ onClick, children }) {
  return (
    <button onClick={e => {
      e.stopPropagation();
      onClick();
    }}>
```

```
   {children}
  </button>
 );
}

export default function Toolbar() {
 return (
  <div className="Toolbar" onClick={() => {
   alert('You clicked on the toolbar!');
  }}>
   <Button onClick={() => alert('Playing!')}>
    Play Movie
   </Button>
   <Button onClick={() => alert('Uploading!')}>
    Upload Image
   </Button>
  </div>
 );
}
```

How this works:

When you click on a button:

1. React calls the onClick handler passed to <button>.
2. That handler, defined in Button, does the following:
   - Calls e.stopPropagation(), preventing the event from bubbling further.

    - Calls the onClick function, which is a prop passed from the Toolbar component.

3. That function, defined in the Toolbar component, displays the button’s own alert.
4. Since the propagation was stopped, the parent <div>’s onClick handler does not run.

   As a result of e.stopPropagation(), clicking on the buttons now only shows a single alert (from the <button>) rather than the two of them (from the <button> and the parent toolbar <div>). Clicking a button is not the same thing as clicking the surrounding toolbar, so stopping the propagation makes sense for this UI.

React Hooks

1. useState

useState is a React Hook that lets you add a state variable to your component.

Syntax:

```
const [state, setState] = useState(initialState)
```

Example:

```
import { useState } from 'react';
```

```
function MyComponent() {
  const [age, setAge] = useState(28);
  const [name, setName] = useState('Taylor');
```

Parameters

initialState: The value you want the state to be initially. It can be a value of any type, but there is a special behavior for functions. This argument is ignored after the initial render.
If you pass a function as initialState, it will be treated as an initializer function. It should be pure, should take no arguments, and should return a value of any type. React will call your initializer function when initializing the component, and store its return value as the initial state.

useState returns an array with exactly two values:

The current state. During the first render, it will match the initialState you have passed.
The set function that lets you update the state to a different value and trigger a re-render.

The set function

The set function returned by useState lets you update the state to a different value and trigger a re-render. You can pass the next state directly, or a function that calculates it from the previous state:

```
const [name, setName] = useState('Edward');

function handleClick() {
  setName('Taylor');
  setAge(a => a + 1);
```

Parameters:

nextState: The value that you want the state to be. It can be a value of any type, but there is a special behavior for functions.
If you pass a function as nextState, it will be treated as an updater function. It must be pure, should take the pending state as its only argument, and should return the next state. React will put your updater function in a queue and re-render your component. During the next render, React will calculate the next state by applying all of the queued updaters to the previous state.

Basic useState examples:

1. Counter

```
import { useState } from 'react';

export default function Counter() {
  const [count, setCount] = useState(0);
  function handleClick() {
    setCount(count + 1);
  }

  return (
    <button onClick={handleClick}>
      You pressed me {count} times
    </button>
  );
}
```

2. Text field

```
import { useState } from 'react';

export default function MyInput() {
  const [text, setText] = useState('hello');

  function handleChange(e) {
    setText(e.target.value);
  }

  return (
    <>
      <input value={text} onChange={handleChange} />
      <p>You typed: {text}</p>
      <button onClick={() => setText('hello')}>
        Reset
      </button>
    </>
  );
}
```

In this example, the text state variable holds a string. When you type, handleChange reads the latest input value from the browser input DOM element, and calls setText to update the state. This allows you to display the current text below.

Updating state based on the previous state

Suppose the age is 42. This handler calls setAge(age + 1) three times:

```
function handleClick() {
  setAge(age + 1); // setAge(42 + 1)
  setAge(age + 1); // setAge(42 + 1)
  setAge(age + 1); // setAge(42 + 1)
}
```

However, after one click, age will only be 43 rather than 45! This is because calling the set function does not update the age state variable in the already running code. So each setAge(age + 1) call becomes setAge(43).

To solve this problem, you may pass an updater function to setAge instead of the next state:

```
function handleClick() {
  setAge(a => a + 1); // setAge(42 => 43)
  setAge(a => a + 1); // setAge(43 => 44)
  setAge(a => a + 1); // setAge(44 => 45)
}
```

Here, a => a + 1 is your updater function. It takes the pending state and calculates the next state from it.

React puts your updater functions in a queue. Then, during the next render, it will call them in the same order:

a => a + 1 will receive 42 as the pending state and return 43 as the next state.
a => a + 1 will receive 43 as the pending state and return 44 as the next state.
a => a + 1 will receive 44 as the pending state and return 45 as the next state.
There are no other queued updates, so React will store 45 as the current state in the end.

By convention, it’s common to name the pending state argument for the first letter of the state variable name, like a for age. However, you may also call it like prevAge or something else that you find clearer.

Updating objects and arrays in state

You can put objects and arrays into state. In React, state is considered read-only, so you should replace it rather than mutate your existing objects. For example, if you have a form object in state, don’t mutate it:

```
// Don't mutate an object in state like this:
form.firstName = 'Taylor';
```

Instead, replace the whole object by creating a new one:

```
//  Replace state with a new object
setForm({
  ...form,
  firstName: 'Taylor'
});
```

How does this work:

This uses the spread operator(...)
The spread operator expands the object’s properties into a new object literal

```
const user = { name: 'Alice', age: 30 };
    const updatedUser = { ...user, age: 31, city: 'New York' };
    console.log(updatedUser); // Output: { name: 'Alice', age: 31, city: 'New York' }
```

2. useEffect

useEffect is a React Hook that lets you synchronize a component with an external system.

useEffect(setup, dependencies?)

Parameters

setup: The function with your Effect's logic. Your setup function may also optionally return a cleanup function. When your component is added to the DOM, React will run your setup function. After every re-render with changed dependencies, React will first run the cleanup function (if you provided it) with the old values, and then run your setup function with the new values. After your component is removed from the DOM, React will run your cleanup function.

optional dependencies: The list of all reactive values referenced inside of the setup code. Reactive values include props, state, and all the variables and functions declared directly inside your component body. If your linter is configured for React, it will verify that every reactive value is correctly specified as a dependency. The list of dependencies must have a constant number of items and be written inline like [dep1, dep2, dep3]. React will compare each dependency with its previous value using the Object.is comparison. If you omit this argument, your Effect will re-run after every re-render of the component

Usage

Eg. Connecting to an external system

Some components need to stay connected to the network, some browser API, or a third-party library, while they are displayed on the page. These systems aren’t controlled by React, so they are called external.

```
import { useState, useEffect } from 'react';
import { createConnection } from './chat.js';

function ChatRoom({ roomId }) {
  const [serverUrl, setServerUrl] = useState('https://localhost:1234');

  useEffect(() => {
          const connection = createConnection(serverUrl, roomId);
    connection.connect();
          return () => {
      connection.disconnect();
          };
  }, [serverUrl, roomId]);
  // ...
}
```

Working:

You need to pass two arguments to useEffect:

A setup function with setup code that connects to that system.
It should return a cleanup function with cleanup code that disconnects from that system.

A list of dependencies including every value from your component used inside of those functions.
React calls your setup and cleanup functions whenever it’s necessary, which may happen multiple times:

Your setup code runs when your component is added to the page (mounts).
After every re-render of your component where the dependencies have changed:
First, your cleanup code runs with the old props and state.
Then, your setup code runs with the new props and state.
Your cleanup code runs one final time after your component is removed from the page (unmounts).
Let’s illustrate this sequence for the example above.

When the ChatRoom component above gets added to the page, it will connect to the chat room with the initial serverUrl and roomId. If either serverUrl or roomId change as a result of a re-render (say, if the user picks a different chat room in a dropdown), your Effect will disconnect from the previous room, and connect to the next one. When the ChatRoom component is removed from the page, your Effect will disconnect one last time.

3. useContext

It is a React Hook that allows components to access data from a React Context without having to pass props down through multiple levels of the component tree
Syntax:
const value = useContext(SomeContext)

Parameters:

SomeContext: The context that you’ve previously created with createContext. The context itself does not hold the information, it only represents the kind of information you can provide or read from components.

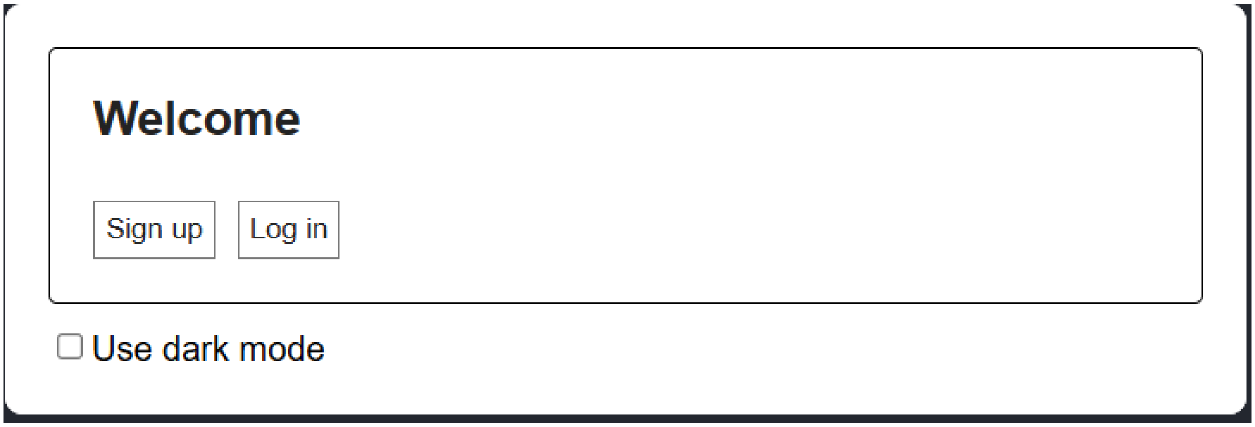
useContext returns the context value for the calling component. It is determined as the value passed to the closest SomeContext above the calling component in the tree. If there is no such provider, then the returned value will be the defaultValue you have passed to createContext for that context. The returned value is always up-to-date. React automatically re-renders components that read some context if it changes.

Popular use case: Dark mode toggle

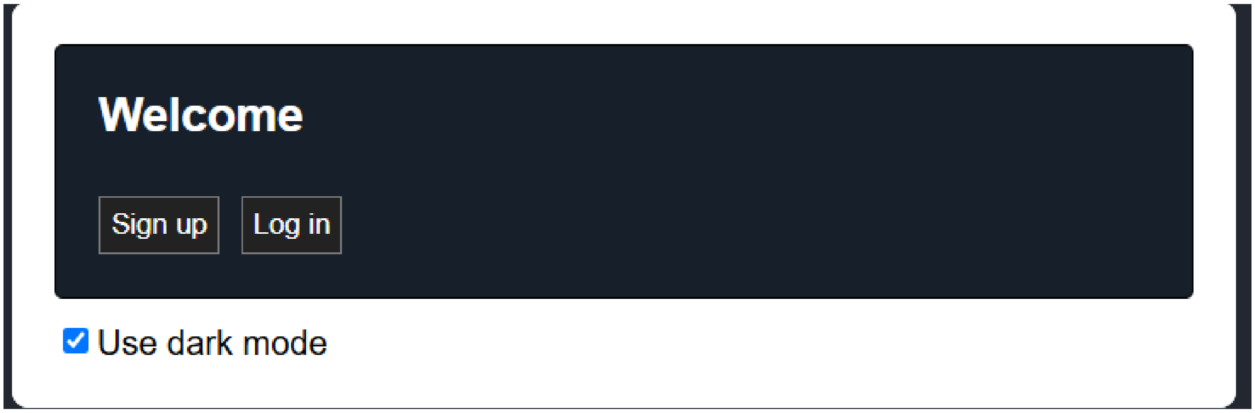
```
import { createContext, useContext, useState } from 'react';

const ThemeContext = createContext(null);

export default function MyApp() {
  const [theme, setTheme] = useState('light');
  return (
    <ThemeContext value={theme}>
      <Form />
      <label>
        <input
          type="checkbox"
          checked={theme === 'dark'}
```

```
     onChange={(e) => {
      setTheme(e.target.checked ? 'dark' : 'light')
     }}
    />
    Use dark mode
   </label>
  </ThemeContext>
 )
}

function Form() {
 return (
  <Panel title="Welcome">
   <Button>Sign up</Button>
   <Button>Log in</Button>
  </Panel>
 );
}

function Panel({ title, children }) {
 const theme = useContext(ThemeContext);
 const className = 'panel-' + theme;
 return (
  <section className={className}>
   <h1>{title}</h1>
   {children}
  </section>
 )
}
```

```
function Button({ children }) {
  const theme = useContext(ThemeContext);
  const className = 'button-' + theme;
  return (
    <button className={className}>
      {children}
    </button>
  );
}
```



□ Use dark mode



☑ Use dark mode

4. useRef

useRef is a React Hook that lets you reference a value that’s not needed for rendering.

const ref = useRef(initialValue)
It is the “getElementByid” of React

Parameters:

initialValue: The value you want the ref object’s current property to be initially. It can be a value of any type. This argument is ignored after the initial render.

useRef returns an object with a single property:

current: Initially, it’s set to the initialValue you have passed. You can later set it to something else. If you pass the ref object to React as a ref attribute to a JSX node, React will set its current property.
On the next renders, useRef will return the same object.

Difference between useState and useRef

Changing a ref does not trigger a re-render. This means refs are perfect for storing information that doesn’t affect the visual output of your component. For example, if you need to store an interval ID and retrieve it later, you can put it in a ref.

Example:

This component uses a ref to keep track of how many times the button was clicked.

```
import { useRef } from 'react';

export default function Counter() {
  let ref = useRef(0);

  function handleClick() {
    ref.current = ref.current + 1;
    alert('You clicked ' + ref.current + ' times!');
  }

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

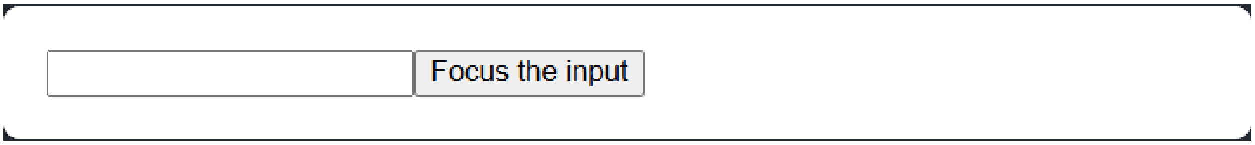
Manipulating DOM with useRef

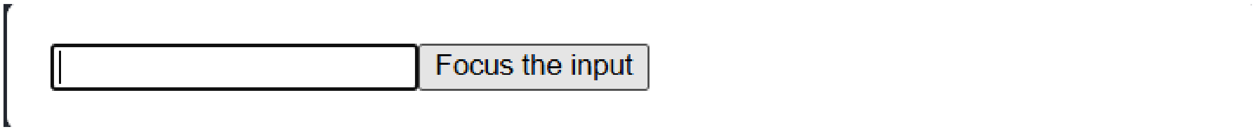
It’s particularly common to use a ref to manipulate the DOM. React has built-in support for this.

```
import { useRef } from 'react';

function MyComponent() {
  const inputRef = useRef(null);
  // …

 return <input ref={inputRef} />;
```

Example:

```
import { useRef } from 'react';

export default function Form() {
  const inputRef = useRef(null);

  function handleClick() {
    inputRef.current.focus();
  }

  return (
    <>
      <input ref={inputRef} />
      <button onClick={handleClick}>
        Focus the input
      </button>
    </>
  );
}
```





5. useReducer

useReducer is a React Hook that lets you add a reducer to your component.

const [state, dispatch] = useReducer(reducer, initialArg, init?)

Parameters:

reducer: The reducer function that specifies how the state gets updated. It must be pure, should take the state and action as arguments, and should return the next state. State and action can be of any types.

initialArg: The value from which the initial state is calculated. It can be a value of any type. How the initial state is calculated from it depends on the next init argument.
optional init: The initializer function that should return the initial state. If it’s not specified, the initial state is set to initialArg. Otherwise, the initial state is set to the result of calling init(initialArg).

useReducer returns an array with exactly two values:

The current state. During the first render, it’s set to init(initialArg) or initialArg (if there’s no init).
The dispatch function that lets you update the state to a different value and trigger a re-render.

The dispatch function

The dispatch function returned by useReducer lets you update the state to a different value and trigger a re-render. You need to pass the action as the only argument to the dispatch function:

```
const [state, dispatch] = useReducer(reducer, { age: 42 });

function handleClick() {
  dispatch({ type: 'incremented_age' });
  // ...
```

React will set the next state to the result of calling the reducer function you’ve provided with the current state and the action you’ve passed to dispatch.

Parameters

action: The action performed by the user. It can be a value of any type. By convention, an action is usually an object with a type property identifying it and, optionally, other properties with additional information.

```
import { useReducer } from 'react';

function reducer(state, action) {
  if (action.type === 'incremented_age') {
    return {
      age: state.age + 1
    };
  }
  throw Error('Unknown action.');
}

export default function Counter() {
  const [state, dispatch] = useReducer(reducer, { age: 42 });

  return (
    <>
      <button onClick={() => {
        dispatch({ type: 'incremented_age' })
      }}>
        Increment age
      </button>
      <p>Hello! You are {state.age}.</p>
    </>
  );
}
```

useReducer is very similar to useState, but it lets you move the state update logic from event handlers into a single function outside of your component.

React forms

React forms allow user interaction and data collection within web applications. While built using standard HTML <form> elements, React manages form data through component state, offering two primary approaches:

1. Controlled Components:

Mechanism: The value of form elements (like <input>, <textarea>, <select>) is controlled by React state. The component's state is the single source of truth for the input's value.

Updates: When the user types or interacts with the input, an onChange event handler updates the component's state, which in turn re-renders the component and updates the input's displayed value.

Benefits: Provides fine-grained control over input values, enables real-time validation, and allows for programmatic manipulation of input values.
Example:

```
import React, { useState } from 'react';

function MyForm() {
  const [name, setName] = useState('');

  const handleSubmit = (e) => {
    e.preventDefault();
    alert(`Hello, ${name}`);
  };

  return (
    <form onSubmit={handleSubmit}>
      <label>
        Name:
        <input type="text" value={name} onChange={(e) => setName(e.target.value)} />
      </label>
      <button type="submit">Submit</button>
    </form>
  );
}
```

2. Uncontrolled Components:

Mechanism: Form data is handled by the DOM itself, similar to traditional HTML forms. React doesn't directly manage the input's value in its state.

Updates: You access the input's value directly from the DOM, typically using refs or FormData API on form submission.

Benefits: Can be simpler for very basic forms where real-time validation or complex state management isn't required.

Example (using useRef):

```
import React, { useRef } from 'react';

function MyUncontrolledForm() {
  const nameInputRef = useRef(null);

  const handleSubmit = (e) => {
    e.preventDefault();
    alert(`Hello, ${nameInputRef.current.value}`);
  };

  return (
    <form onSubmit={handleSubmit}>
      <label>
        Name:
        <input type="text" ref={nameInputRef} />
      </label>
      <button type="submit">Submit</button>
    </form>
  );
}
```

React Router

Used to navigate between pages in large scale react applications

Installation:
npm i react-router-dom

BrowserRouter

Wraps the App component.
This allows us to use all features of react-router in App component.

Two components: Routes and Route

Route component wraps the Routes Component.
Route component is used to define individual paths.
It has 2 features, path and element to be rendered

Link tag

Component of react-router-dom which helps to navigate between the routes. It is like an anchor tag, but helpful in terms of simplicity
It is essentially an anchor tag with the path as href

Example:

```
import { BrowserRouter, Routes, Route, Link } from "react-router-dom";

function App() {
  return (
    <BrowserRouter>
      <nav>
        <Link to="/">Home</Link> | <Link to="/about">About</Link>
      </nav>
      <Routes>
        <Route path="/" element={<Home />} />
        <Route path="/about" element={<About />} />
      </Routes>
    </BrowserRouter>
  );
```

Node.js

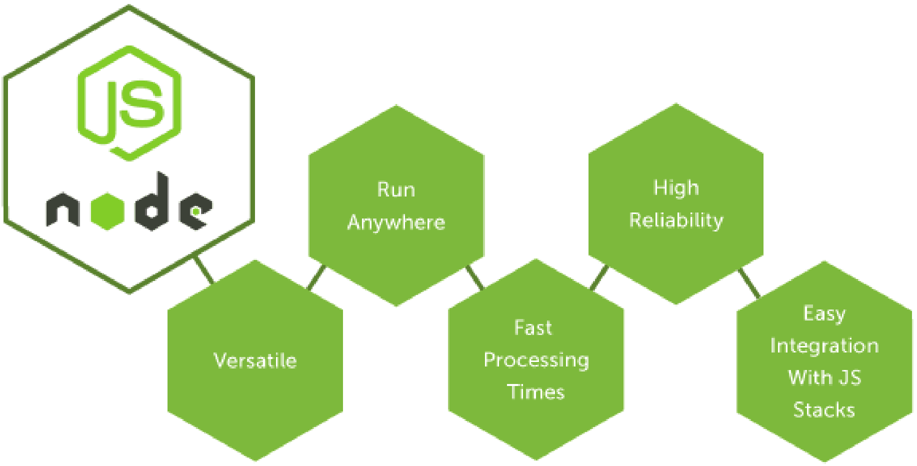
Node.js is an open-source, cross-platform runtime built on Google's V8 JavaScript engine. It allows developers to run JavaScript code outside the browser to build fast and scalable network applications.

Node.js Architecture

Node.js follows a Single-Threaded Event Loop Model.

1. Single-Threaded: Node.js runs on a single thread to handle requests, unlike traditional multi-threaded servers. This makes it lightweight and efficient.

2. Event Loop: The event loop allows Node.js to handle multiple requests without waiting for one to finish before moving to the next. This non-blocking behavior provides high performance.



Components of Node.js Architecture

V8 Engine:

- Developed by Google and used in Chrome.
- Converts JavaScript code into machine code for faster execution.

Libuv Library:

- Handles asynchronous I/O operations such as file system tasks and networking.
- Provides the event loop and thread pool for non-blocking operations.

Event Queue:

- Holds incoming client requests waiting to be processed by the event loop.

Event Loop:

- Continuously checks for tasks, executes them, and waits for new events.

Callback Functions:

- Functions passed as arguments that run after certain tasks complete (for example, reading a file).

Node.js Server Architecture

Node.js manages multiple concurrent clients using a Single Threaded Event Loop.
It uses two main concepts:

1. Asynchronous Model
2. Non-blocking I/O Operations

These features improve scalability, performance, and throughput of web applications.

Components of the Node.js Architecture

1. Requests: Client requests can be blocking (complex) or non-blocking (simple).

2. Node.js Server:Accepts, processes, and returns responses for user requests.

3. Event Queue: Stores incoming requests and passes them sequentially to the event loop.

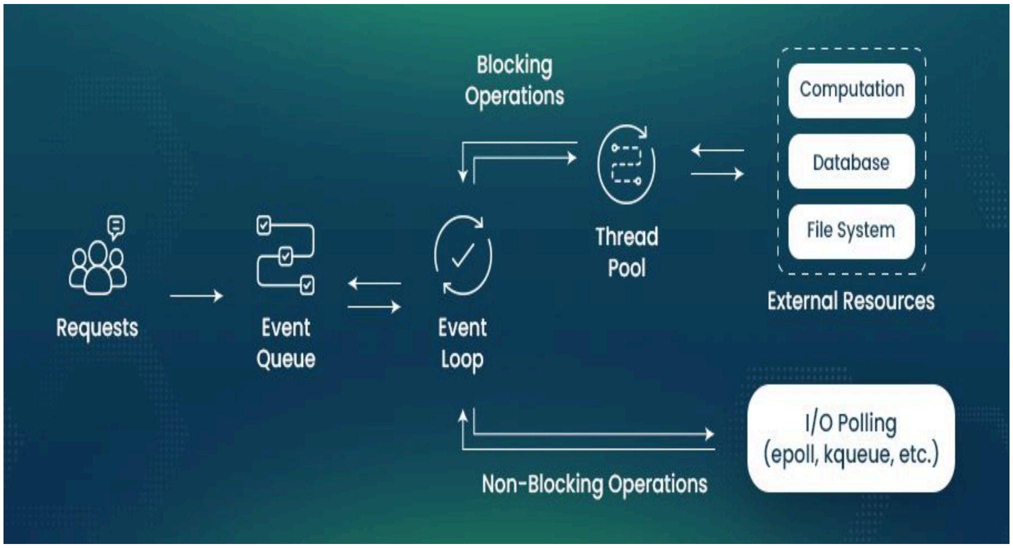
4. Thread Pool:Contains threads available for performing operations required to process requests.

5. Event Loop: Receives requests from the event queue and returns responses to clients.

6. External Resources:Used to handle blocking client requests such as computations or database operations.

Node.js server Workflow

1. Users send blocking or non-blocking requests to the server.
2. Requests enter the Event Queue.
3. The Event Loop checks each request type.
4. Non-blocking requests are processed directly and responses are sent back.
5. Blocking requests are assigned a thread for execution using external resources.
6. Once completed, the response is returned through the Event Loop to the client.



Benefits of Node.js Architecture

- Scalable: Handles a large number of simultaneous connections efficiently.
- Non-Blocking I/O: Reduces waiting time and improves efficiency.
- Fast Execution: The V8 engine ensures quick JavaScript execution.

For Node.js installation refer to the slides provided on PESU Academy

Node.js Basics

Node.js is a cross-platform JavaScript runtime environment.

It allows creation of scalable servers using JavaScript without threads.

It supports both console-based and web-based applications.

Datatypes in Node.js

Node.js supports the following data types (similar to JavaScript):

1. Boolean
2. Undefined
3. Null
4. String
5. Number

Loose Typing in Node.js

Node.js supports loose typing, meaning variable types need not be declared in advance.

The same variable can hold different data types at different times.

Variables are declared using the “var” keyword.

Examples:

```
// Variable storing a Number data type
var a = 665;
console.log(a);         // Output: 665
console.log(typeof a);   // Output: number

// Variable storing a String data type
```

```
a = "PES";
console.log(a);          // Output: PES
console.log(typeof a);   // Output: string



// Variable storing a Boolean data type
a = true;
console.log(a);          // Output: true
console.log(typeof a);   // Output: boolean

// Variable storing an Undefined data type
a = undefined;
console.log(a);          // Output: undefined
console.log(typeof a);   // Output: undefined

// Variable storing a Null data type
a = null;
console.log(a);          // Output: null
console.log(typeof a);   // Output: object
```

Objects in Node.js

Node.js objects are the same as JavaScript objects.
Objects store data as name-value pairs and can hold multiple values.

Example:

```
var company = {
    Name: "Lamar",
    Address: "LA",
    Contact: "+919876543210",
    Email: "www.gta.com"
};

// Display the object information
console.log("Information of variable company:", company);

// Display the type of variable
console.log("Type of variable company:", typeof company);
```

Output:

```
Information of variable company: {
  Name: 'Lamar',
  Address: 'LA',
  Contact: '+919876543210',
  Email: 'www.gta.com'
}
Type of variable company: object
```

Strings in Node.js

Strings represent textual data. They can be defined using single or double quotes and manipulated using built-in functions.

Example:

```
var x = "Welcome ";
var y = 'To the';
var z = ['Classroom', 'Of', 'The', 'Elite'];
console.log(x); //Output: Welcome
console.log(y); //Output: To the
```

String Functions Example:

```
// Defining string variables
var x = "Welcome ";
var y = 'To the';
var z = ['Classroom', 'Of', 'The', 'Elite'];
console.log("String 1:", x);
console.log("String 2:", y);
console.log("Array:", z);

// Concatenation using (+)
console.log(" Concat Using (+):", x + y);

//  Concatenation using concat() function
console.log("Concat Using Function:", x.concat(y));
```

```
//  Splitting a string into an array using split()

console.log("Split String:", x.split(' '));


// Joining array elements into a string using join()

console.log("Join String:", z.join(', '));


// Getting a character at a specific index using charAt()

console.log("Character at Index 5:", x.charAt(5));
```

Output:

```
String 1: Welcome

String 2: To the

Array: [ 'Classroom', 'Of', 'The', 'Elite' ]

 Concat Using (+): Welcome To the

Concat Using Function: Welcome To the

Split String: [ 'Welcome', '' ]

Join String: Classroom, Of, The, Elite

Character at Index 5: m
```

Node Packages and Modules

What is NPM?

NPM (Node Package Manager) is the official package manager for Node.js. It helps developers install, share, and manage open-source Node.js packages easily.

- Official website: https://www.npmjs.com
- Installed automatically with Node.js
- Stores downloaded packages in a folder named node_modules/

Each package contains all necessary files for a module that you can use in your code.

Example of installing and using a package:

Step 1: Initialize NPM

Open the terminal in your project folder and run:

```
npm init -y
```

This creates a package.json file.

Step 2: Install the Package

```
npm install validator
```

After installation:

- A folder named node_modules is created.

- The validator package is added inside it.

Step 3: Use the Installed Package

//This step is essential to follow the examples given later on

Make sure your package.json includes "type": "module"

```
{
  "type": "module"
}
```

Then create a file named app.js in that:

```
// Import validator package (ESM syntax)
import validator from "validator";

const email = "student@pes.edu";
if (validator.isEmail(email)) {
  console.log(`${email} is a valid email address!`);
} else {
  console.log(`${email} is NOT a valid email address!`);
}
```

Output:

```
student@pes.edu is a valid email address!
```

What is a Module in Node.js?

A module is a reusable piece of code (single file or a collection) that can be imported into other files.
Modules improve reusability, readability, and code organization.

Types of Modules

1. Built-in Modules – Provided by Node.js itself
   Examples: fs, http, os, path, events
2. Local Modules – Created by the user
3. Third-party Modules – Installed using NPM (like validator, express, etc.)

Viewing All Built-in Modules

You can see all available built-in modules here: https://nodejs.org/docs/v22.20.0/api/modules.html#built-in-modules

Importing Built-in Modules

```
import fs from "fs";
fs.writeFileSync("example.txt", "Hello from Node.js v20+!");
const data = fs.readFileSync("example.txt", "utf-8");
console.log("File Content:", data);
```

Detailed examples will be given later on.

Global Modules (No Import Required)

Global modules like console, setTimeout, and setInterval are always available.

```
console.log("This message is from the console module");
setTimeout(() => {
console.log("This appears after 2 seconds (using Timer Module)");
        }, 2000);
```

Timer module

Method	Description
clearImmediate()	Cancels an Immediate object
clearInterval()	Cancels an Interval object
clearTimeout()	Cancels a Timeout object
ref()	Makes the Timeout object active. Will only have an effect if the Timeout.unref() method has been called to make the Timeout object inactive.
setImmediate()	Executes a given function immediately.
setInterval()	Executes a given function at every given milliseconds
setTimeout()	Executes a given function after a given time (in milliseconds)
unref()	Stops the Timeout object from remaining active

Example 1:

```
const timeoutId = setTimeout(() => {

  console.log("This message appears after 3 seconds");

}, 3000);
```

```
console.log("Timer started...");


setTimeout(() => {
  clearTimeout(timeoutId);
  console.log("Timeout cleared!");
}, 4000);
```

Output:

```
Timer started...
This message appears after 3 seconds
Timeout cleared!
```

Example 2:

```
let count = 0;


const intervalId = setInterval(() => {
  count++;
  console.log(`Running... ${count}`);

  if (count === 5) {

    clearInterval(intervalId);

    console.log("Interval cleared after 5 runs");

  }

}, 1000);
```

Output:

```
Running... 1

Running... 2

Running... 3

Running... 4

Running... 5

Interval cleared after 5 runs
```

Example 3:

```
const immediateId = setImmediate(() => {

  console.log("This runs immediately after current I/O events");

});
```

```
console.log("This message logs first.");

clearImmediate(immediateId); // Cancels the immediate
console.log("Immediate cleared before it ran!");
```

Output:

```
This message logs first.
Immediate cleared before it ran!
```

Creating and Importing Your Own Modules

1. Exporting Functions

Create a file named mathUtils.js in that:

```
export function multiply(a, b) {
  return a * b;
}
```

```
export function add(a, b) {
  return a + b;
}
```

Create a file named main.js and in that:

```
import { multiply, add } from "./mathUtils.js";
console.log("Addition:", add(5, 3));
console.log("Multiplication:", multiply(5, 3));
```

Make sure both the files are in the same directory level. If not, change the path in the import accordingly.

Run main.js and Output:

```
Addition: 8
Multiplication: 15
```

1. Exporting Objects

Make a file named intro.js in that:

```
export const intro = {
  name: "Alan Wake",
  occupation: "Writer",
  city: "Bengaluru",

  display() {

    console.log(`My name is ${this.name} and I am a ${this.occupation} based
in ${this.city}.`);

  }

};
```

Create another file named app.js in that:

```
import { intro } from './intro.js';

intro.display();
```

Output:

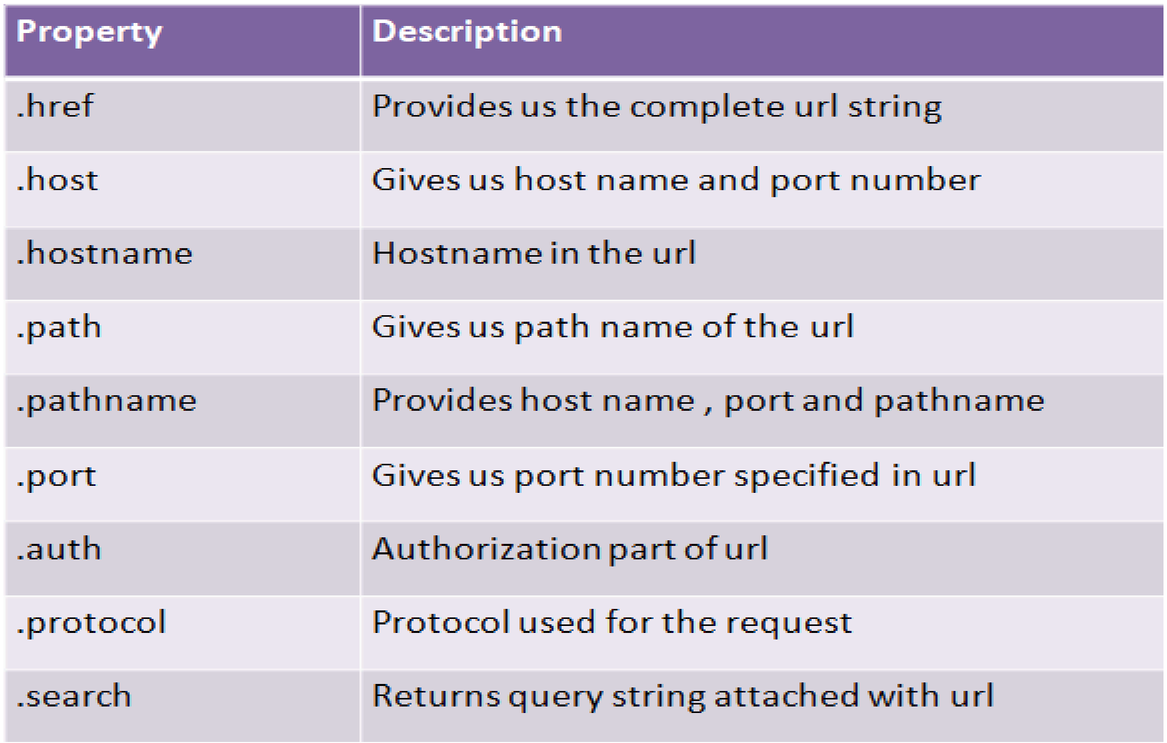
```
My name is Alan Wake and I am a Writer based in Bengaluru.
```

URL Module

The URL module in Node.js is used to parse, construct, normalize, and encode URLs.

It provides utilities for working with web addresses in a structured way instead of using plain strings.

This module helps extract parts of a URL such as protocol, hostname, path, query string, and hash.



Property	Description
.href	Provides us the complete url string
.host	Gives us host name and port number
.hostname	Hostname in the url
.path	Gives us path name of the url
.pathname	Provides host name , port and pathname
.port	Gives us port number specified in url
.auth	Authorization part of url
.protocol	Protocol used for the request
.search	Returns query string attached with url

Example:

```
import url from 'url';

const myUrl = new URL("https://user:pass@www.example.com:8080/pathname/page?name=Diddee&age=55");

console.log("href      :", myUrl.href);

console.log("host      :", myUrl.host);
console.log("hostname  :", myUrl.hostname);

console.log("path      :", myUrl.pathname + myUrl.search);

console.log("pathname  :", myUrl.pathname);

console.log("port      :", myUrl.port);

console.log("username  :", myUrl.username);

console.log("password  :", myUrl.password);

console.log("protocol  :", myUrl.protocol);

console.log("search    :", myUrl.search);
```

Output:

```
href      : https://user:pass@www.example.com:8080/pathname/page?name=Diddee&age=55

host      : www.example.com:8080

hostname  : www.example.com

path      : /pathname/page?name=Diddee&age=55

pathname  : /pathname/page

port      : 8080

username  : user
```

password : pass

protocol : https:

search   : ?name=Diddee&age=55

HTTP Module

The HTTP module in Node.js allows creating web servers and handling HTTP requests and responses.
It is one of the most commonly used built-in modules for backend development.

To use it, import it using:

import http from "http";

Method / Property	Description
http.STATUS_CODES	A collection of all the standard HTTP response status codes, and the short description of each.
http.request(options, [callback])	Allows one to transparently issue HTTP requests.

http.get(options, [callback])	Sets the method to GET and calls req.end() automatically.
http.createServer([requestListener])	Returns a new web server object. The requestListener is a function automatically added to the 'request' event.
server.listen(port, [hostname], [backlog], [callback])	Begins accepting connections on the specified port and hostname.
server.close([callback])	Stops the server from accepting new connections.

Method / Property	Description
request.write(chunk, [encoding])	Sends a chunk of the request body.
request.end([data], [encoding])	Finishes sending the request. Flushes any unsent body parts.
request.abort()	Aborts a request.
response.write(chunk, [encoding])	Sends a chunk of the response body. If response.writeHead() hasn’t been called, it switches to implicit header mode and flushes headers.
response.writeHead(statusCode, [reasonPhrase], [headers])	Send a response header to the request.

response.getHeader(name)	Reads out a header that’s already been queued but not yet sent (case-insensitive).
response.removeHeader(name)	Removes a header that’s queued for implicit sending.
response.end([data], [encoding])	Signals that all response headers and body have been sent; marks the response as complete. Must be called for each response.

Example 1: Basic HTTP Server

```
import http from "http";

const server = http.createServer((req, res) => {
 res.writeHead(200, { "Content-Type": "text/plain" });
 res.write("Welcome to the Node.js HTTP Server!");
 res.end();
 });

server.listen(3000, () => {
 console.log("Server is running at http://localhost:3000");
 });
```

What is happening:

1. The createServer() method creates a new web server.
2. res.writeHead() sets the response status and content type.

3. res.write() sends the response body.
4. res.end() marks the end of the response.
5. The server listens on port 3000.

Output:

When you open your browser and go to http://localhost:3000
The page displays:
Welcome to the Node.js HTTP Server!

The terminal displays:
Server is running at http://localhost:3000

Example 2: Handling Different Routes

```
import http from "http";

const server = http.createServer((req, res) => {
 if (req.url === "/") {
 res.writeHead(200, { "Content-Type": "text/plain" });
 res.end("Home Page");
 } else if (req.url === "/about") {
 res.writeHead(200, { "Content-Type": "text/plain" });
 res.end("About Page");
 } else if (req.url === "/contact") {
 res.writeHead(200, { "Content-Type": "text/plain" });
 res.end("Contact Page");
 } else {
 res.writeHead(404, { "Content-Type": "text/plain" });
 res.end("404 Not Found");
 }
 });
```

```
server.listen(3000, () => {
 console.log("Server running at http://localhost:3000");
 });
```

Output:

If you visit:
 http://localhost:3000 → Home Page
 http://localhost:3000/about → About Page
 http://localhost:3000/contact → Contact Page
 http://localhost:3000/xyz → 404 Not Found

Example 3: Sending HTML Response

```
import http from "http";const server = http.createServer((req, res) => {
 res.writeHead(200, { "Content-Type": "text/html" });
 res.write("<h1>Welcome to Node.js HTTP Server</h1>");
 res.write("<p>This is an HTML response.</p>");
 res.end();
 });
```

```
server.listen(4000, () => {
 console.log("Server running at http://localhost:4000");
 });
```

Output:

Open http://localhost:4000
It shows a web page with:

```
Welcome to Node.js HTTP Server
This is an HTML response.
```

CHALK Module

The chalk module is a third-party Node.js package used to style and colorize terminal output.
It makes console logs more readable and visually appealing with colored text, background colors, and text styles.

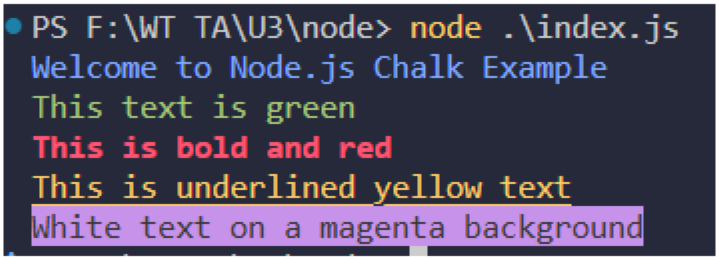
To use chalk, first install it using NPM.

```
npm i chalk
```

Example:

```
import chalk from "chalk";

console.log(chalk.blue("Welcome to Node.js Chalk Example"));
console.log(chalk.green("This text is green"));
console.log(chalk.red.bold("This is bold and red"));
console.log(chalk.yellow.underline("This is underlined yellow text"));
console.log(chalk.bgMagenta.white("White text on a magenta background"));
```

Output in terminal:



```
PS F:\WT TA\U3\node> node .\index.js
Welcome to Node.js Chalk Example
This text is green
This is bold and red
This is underlined yellow text
White text on a magenta background
```

File System Module

The File System (fs) module in Node.js allows interacting with the file system on your computer.

It is used for performing operations such as reading, writing, updating, deleting, and renaming files and directories.

It supports both Synchronous (Blocking) and Asynchronous (Non-Blocking) methods.

The fs module is a built-in module, so no installation is required.

Import it using:

import fs from "fs";

Synchronous vs Asynchronous

Synchronous methods block the execution of further code until the current operation completes.

Asynchronous methods execute operations in the background and allow the program to continue running without waiting.

Common fs Module Functions

1. fs.writeFile() and fs.writeFileSync() → Write to a file
2. fs.readFile() and fs.readFileSync() → Read from a file
3. fs.appendFile() and fs.appendFileSync() → Add data to an existing file
4. fs.rename() and fs.renameSync() → Rename a file
5. fs.unlink() and fs.unlinkSync() → Delete a file
6. fs.mkdir() and fs.mkdirSync() → Create a directory
7. fs.rmdir() and fs.rmdirSync() → Remove a directory
8. fs.existsSync() → Check if a file or directory exists

Synchronous Example (Blocking)

```
import fs from "fs";

try {

  // Write to a file

  fs.writeFileSync("example.txt", "This is a synchronous write example.");

  // Read from the file
  const data = fs.readFileSync("example.txt", "utf8");
  console.log("File content:", data);
  // Append new content
  fs.appendFileSync("example.txt", "\nAppending some more text.");

  // Rename the file
  fs.renameSync("example.txt", "example_renamed.txt");
  // Delete the file
  fs.unlinkSync("example_renamed.txt");
  console.log("All synchronous operations completed successfully!");
} catch (err) {
  console.error("Error during synchronous operation:", err);
}
```

Output:

```
File content: This is a synchronous write example.
All synchronous operations completed successfully!
```

Explanation:

Initially example.txt is empty, then the content - ‘This is a synchronous write example.’ is written to it. Then it is read and in the next line - ‘Appending some more text.’ is added. Then it is renamed to example_renamed.txt and finally it is deleted.

writeFileSync() – Creates or overwrites a file.

readFileSync() – Reads file content (blocking).

appendFileSync() – Adds new data to an existing file.

renameSync() – Renames a file.

unlinkSync() – Deletes a file.

Asynchronous Example (Non-blocking):

```
import fs from "fs";

// Write to a file

fs.writeFile("example.txt", "This is an asynchronous write example.", (err) => {

  if (err) {

    console.error(err);

    return;

  }

  console.log("File written successfully.");

  // Read from the file

  fs.readFile("example.txt", "utf8", (err, data) => {
```

```
if (err) {
  console.error(err);
  return;
}
console.log("File content:", data);
// Append new content
fs.appendFile("example.txt", "\nAppending async text.", (err) => {
  if (err) {
    console.error(err);
    return;
  }
  console.log("File appended successfully.");
  // Rename the file
  fs.rename("example.txt", "example_renamed.txt", (err) => {
    if (err) {
      console.error(err);
      return;
    }
    console.log("File renamed successfully.");
    // Delete the file
```

```
        fs.unlink("example_renamed.txt", (err) => {
          if (err) {
            console.error(err);
            return;
          }
          console.log("All asynchronous operations completed successfully!");
        });
      });
    });
  });
});
```

Output:

```
File written successfully.
File content: This is an asynchronous write example.
File appended successfully.
File renamed successfully.
All asynchronous operations completed successfully!
```

Explanation:

Initially example.txt is empty, then the content - ‘This is an asynchronous write example.’ is written to it. Then it is read and in the next line - ‘Appending async text.’ is added. Then it is renamed to example_renamed.txt and finally it is deleted.

writeFile() – Writes data asynchronously.

readFile() – Reads file without blocking the main thread.

appendFile() – Adds data asynchronously.

rename() – Renames file asynchronously.

unlink() – Deletes file asynchronously.

Opening modes:

Flag	Mode	If File Exists	If File Doesn’t Exist	Use Case
r	Read	OK	Error	Reading only
r+	Read + Write	OK	Error	Read & write without creating
rs	Read (sync)	OK	Error	Direct disk read
rs+	Read + Write (sync)	OK	Error	Direct disk read/write
w	Write	Truncate	Create new	Overwrite/create file

		(empty)		
wx	Write (exclusive)	Error	Create new	Safe create (no overwrite)
w+	Read + Write	Truncate (empty)	Create new	Read & overwrite/create
wx+	Read + Write (ex)	Error	Create new	Safe read/write create
a	Append	Append	Create new	Add data at end
ax	Append (exclusive)	Error	Create new	Safe append create
a+	Read + Append	Append	Create new	Read & add data
ax+	Read + Append (ex)	Error	Create new	Safe read+append create

Buffers and Streams

Buffers in Node.js

Buffers are temporary memory areas that hold binary data.

They are used mainly when dealing with file system operations, network streams, or binary data (like images or videos).

Unlike strings, Buffers handle raw binary data directly.

Example:

```
// Creating a buffer from a string
const buf1 = Buffer.from("Beach Lasagna");
console.log("Buffer 1:", buf1);
console.log("String from buffer:", buf1.toString());


// Allocating a buffer of 20 bytes
const buf2 = Buffer.alloc(20);
console.log("Empty buffer:", buf2);

// Writing data into buffer
buf2.write("Hey! Listen!");
console.log("After writing:", buf2.toString());

// Concatenating buffers
const buf3 = Buffer.concat([buf1, buf2]);
console.log("Concatenated Buffer:", buf3.toString());
```

```
// Comparing buffers

const isEqual = Buffer.compare(buf1, buf2);

console.log("Buffer comparison result:", isEqual);
```

Output:

```
Buffer 1: <Buffer 42 65 61 63 68 20 4c 61 73 61 67 6e 61>

String from buffer: Beach Lasagna

Empty buffer: <Buffer 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00>

After writing: Hey!Listen!

Concatenated Buffer: Beach LasagnaHey!Listen!

Buffer comparison result: -1
```

Explanation:

Buffer.from() – Creates a buffer from a string or array.

Buffer.alloc() – Allocates memory for the buffer.

write() – Writes data to a buffer.

toString() – Converts buffer data back to a readable string.

Buffer.concat() – Combines multiple buffers into one.

Buffer.compare() – Compares two buffers lexicographically.

Streams in Node.js

Streams are used to handle continuous data flow (reading or writing) efficiently.

They don’t load the whole data into memory at once — they process it in chunks.

There are four main types of streams:

1. Readable – For reading data (e.g., from files)
2. Writable – For writing data (e.g., to files)
3. Duplex – For both reading and writing (e.g., TCP sockets)
4. Transform – For modifying data while reading/writing (e.g., compression)

Example 1:

//Make sure to create an input.txt file beforehand the new file will be created if not present else overwritten

```
import fs from "fs";


// Create a readable stream

const readStream = fs.createReadStream("input.txt", "utf8");
```

```
// Create a writable stream
const writeStream = fs.createWriteStream("output.txt");

// Pipe the read stream into the write stream
readStream.pipe(writeStream);

console.log("File is being copied using streams...");
```

Output:

Input.txt:

```
Never gonna give you up, never gonna let you down
Never gonna run around and desert you
Never gonna make you cry, never gonna say goodbye
Never gonna tell a lie and hurt you
```

Output.txt: (after running code)

```
Never gonna give you up, never gonna let you down
Never gonna run around and desert you
```

Never gonna make you cry, never gonna say goodbye

Never gonna tell a lie and hurt you

Explanation:

fs.createReadStream() – Reads file data in chunks.

fs.createWriteStream() – Writes data in chunks.

pipe() – Connects readable and writable streams directly.

This method is memory-efficient because it doesn't load the entire file into RAM — perfect for large files.

Example 2:

```
import fs from "fs";

const stream = fs.createReadStream("input.txt", { encoding: "utf8" });

stream.on("data", (chunk) => {
  console.log("Received chunk:", chunk);
});
```

```
stream.on("end", () => {
  console.log("No more data to read.");
});

stream.on("error", (err) => {
  console.error("Error:", err);
});
```

Output:

```
Received chunk: Never gonna give you up, never gonna let you down
Never gonna run around and desert you
Never gonna make you cry, never gonna say goodbye
Never gonna tell a lie and hurt you
No more data to read.
```

Explanation:

The data event is emitted whenever a chunk is available.

The end event indicates the end of the file.

The error event handles any read errors.