React

HTML, CSS & JS.

What is NPM. (Node Package Manager)

- npm is a package manager for the JavaScript programming language. It is used to manage and install packages and dependencies that are required in a JavaScript project.
- npm is included with Node.js, which is a JavaScript runtime environment that allows developers to run JavaScript code outside of a web browser.
- npx is a tool that is included with npm and is used to execute packages without having to install them globally on your machine. (react-scripts)
- Overall, npx is a convenient way to run scripts from packages without having to install them globally, and is particularly useful when working with create-react-app.
- Commands - npm init, npm install uuid, npm update, npx create-react-app

What is React ?

React is a JavaScript library used for building user interfaces (UIs) on the web.

It allows developers to create reusable components that can be easily composed together to create complex UIs.

Function Button(){

return <button>Heelo</button>

}

<Button />

<Button />

Difference between Library (react) and Framework. (nextjs, angular, vue)

- A library is a collection of pre-written code that provides specific functionality for use in a program.
- A framework is a more comprehensive software architecture that provides a set of rules and guidelines for building applications from start to finish.
- The main difference between the two is that a library offers specific functions that a developer can pick and choose to use, while a framework provides a complete structure for building an application.
- Library - react
- Framework - nextjs, angular

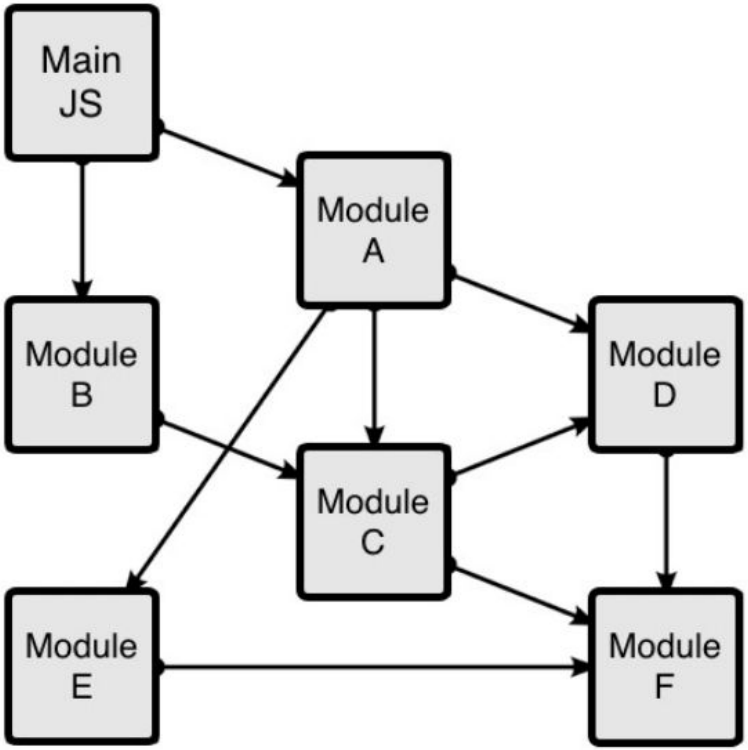
- Getting started with React
- Package.json
- Node Modules - Store all codes related to packages
- Public Folder
- Src Folder
- Components
- JSX - babel
- Git Ignore

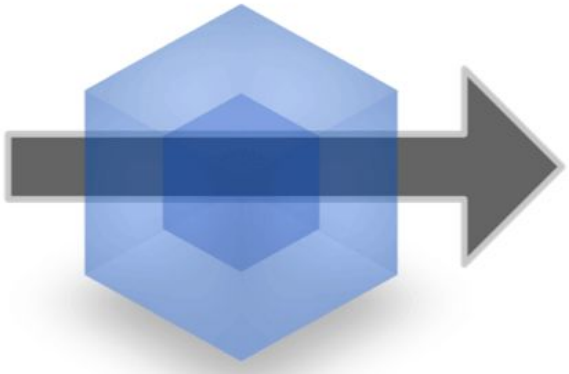
DID YOU KNOW?

Browser not understand jsx, scss, typescript


BABEL
JS

Webpack : Webpack helps developers to bundle and optimize their web assets (JavaScript, CSS, images, and fonts) for use in a web application. Overall, the concept of bundling in Webpack is about taking a collection of smaller code files, optimizing and transforming them as needed, and producing a single bundle file that can be used to run a web application.

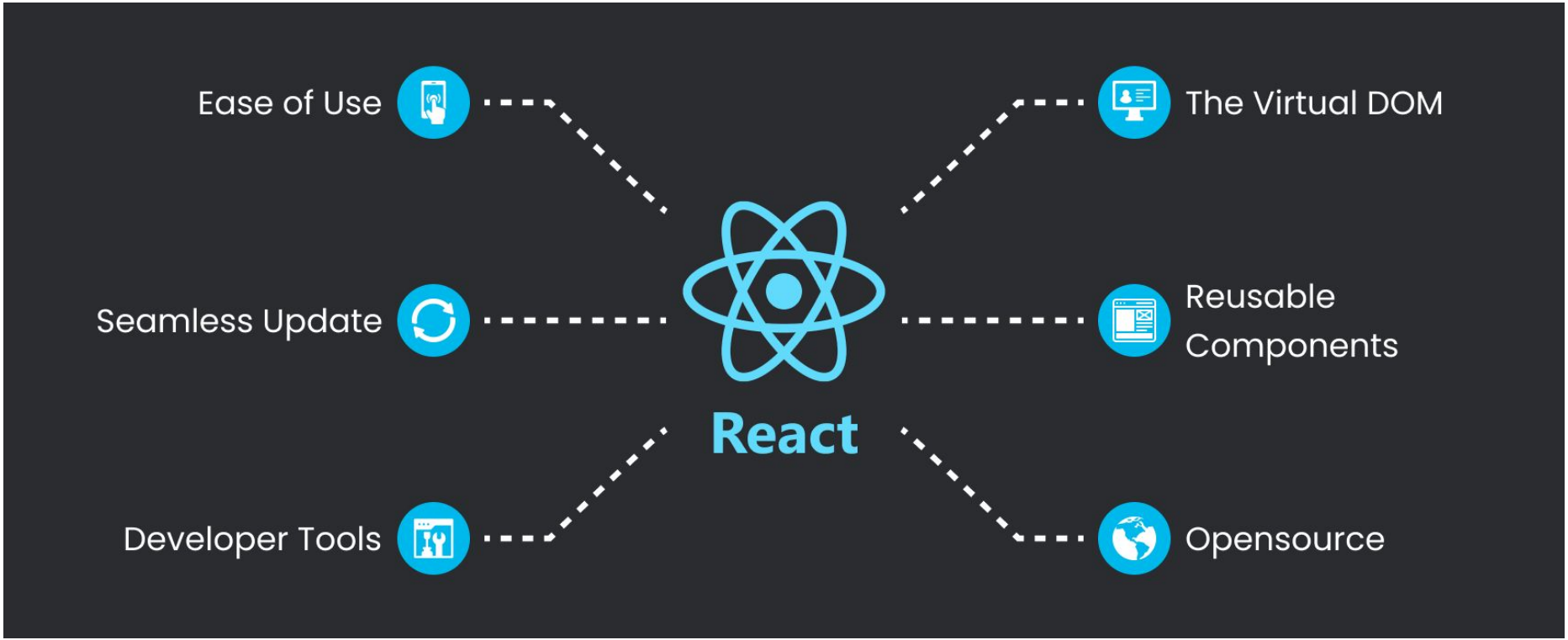




webpack
MODULE BUNDLER

Bundled JavaScript

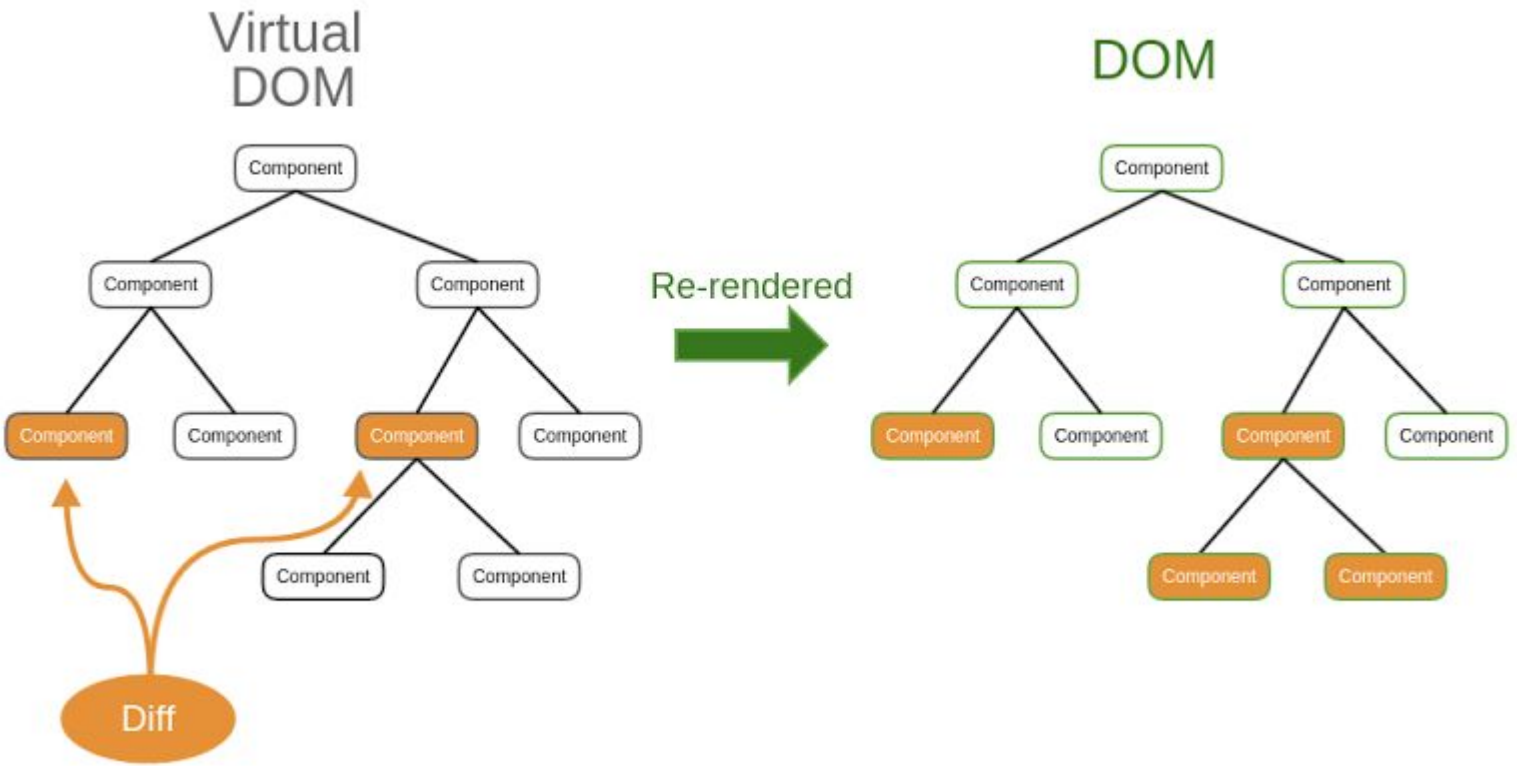
Why React ?

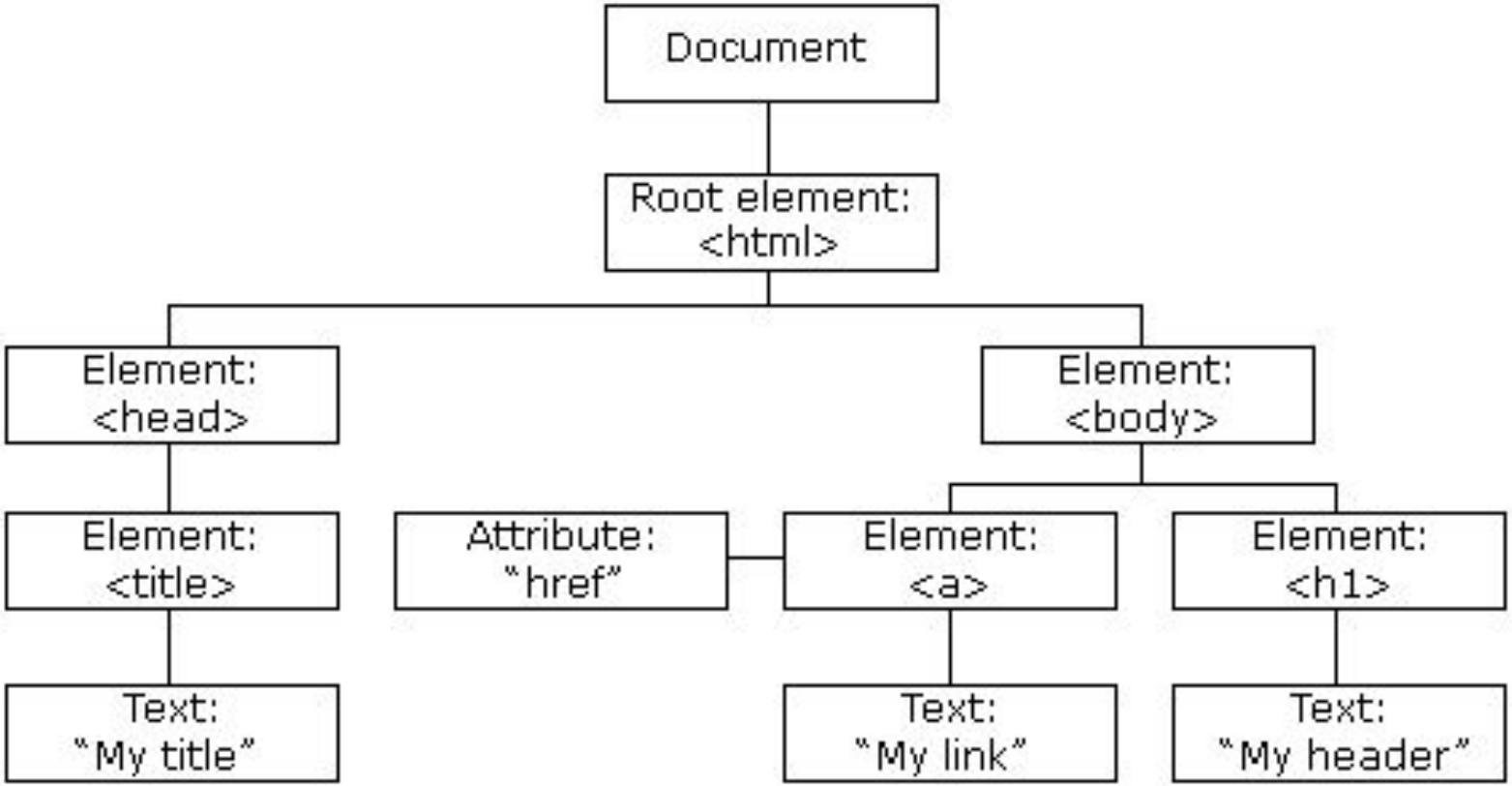

Ease of Use
The Virtual DOM
Seamless Update
Reusable Components
React
Developer Tools
Opensource

Virtual DOM

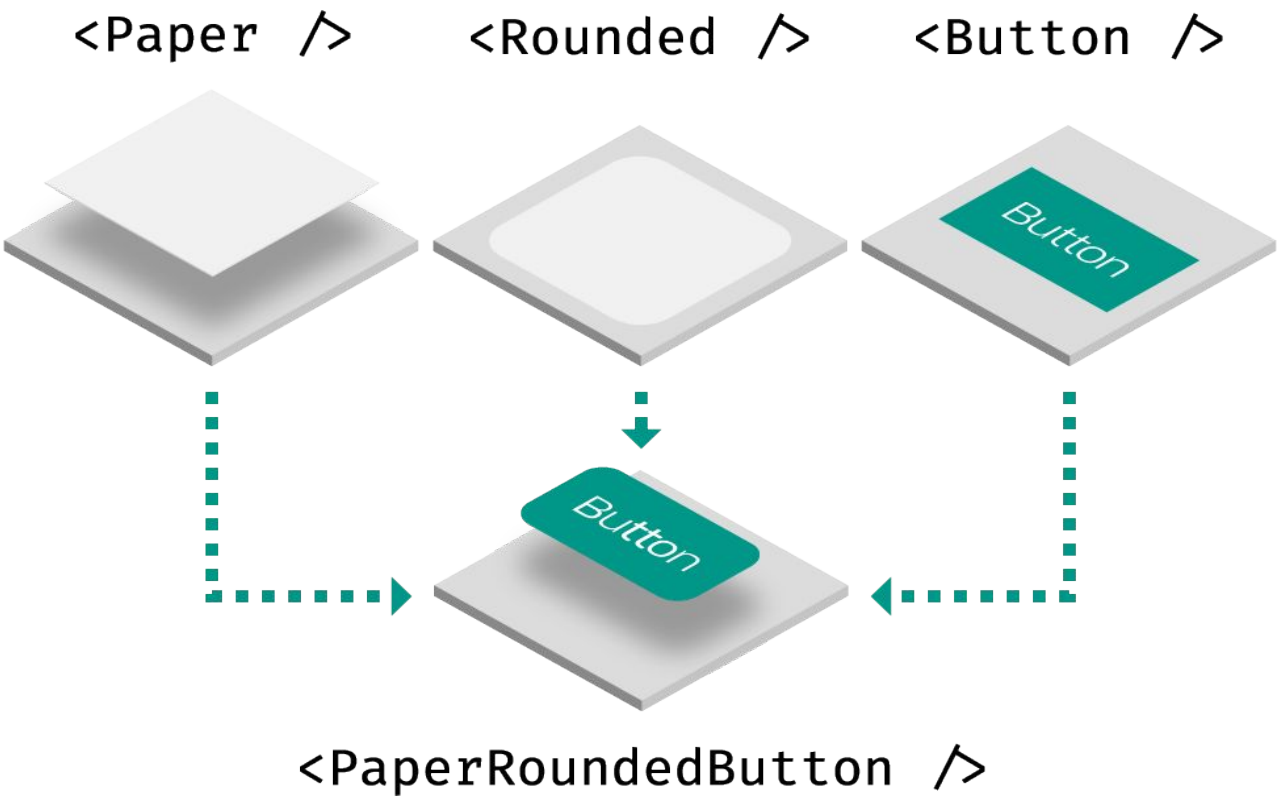
- In React, the Virtual DOM (Document Object Model) is a lightweight copy of the actual DOM tree, which is the hierarchical structure that represents the HTML content of a web page.
- The Virtual DOM is an abstraction that allows React to efficiently update the actual DOM by comparing the current Virtual DOM with the previous one, and applying only the necessary changes to the real DOM.
- Using the Virtual DOM helps to reduce the number of costly DOM operations that are required to update the UI, which can lead to significant performance improvements in complex applications.

Virtual DOM vs Real DOM




Document
Root element:
<html>
Element:
<head>
Element:
<body>
Element:
<title>
Attribute:
"href"
Element:
<a>
Element:
<h1>
Text:
"My title"
Text:
"My link"
Text:
"My header"

Reusable Component



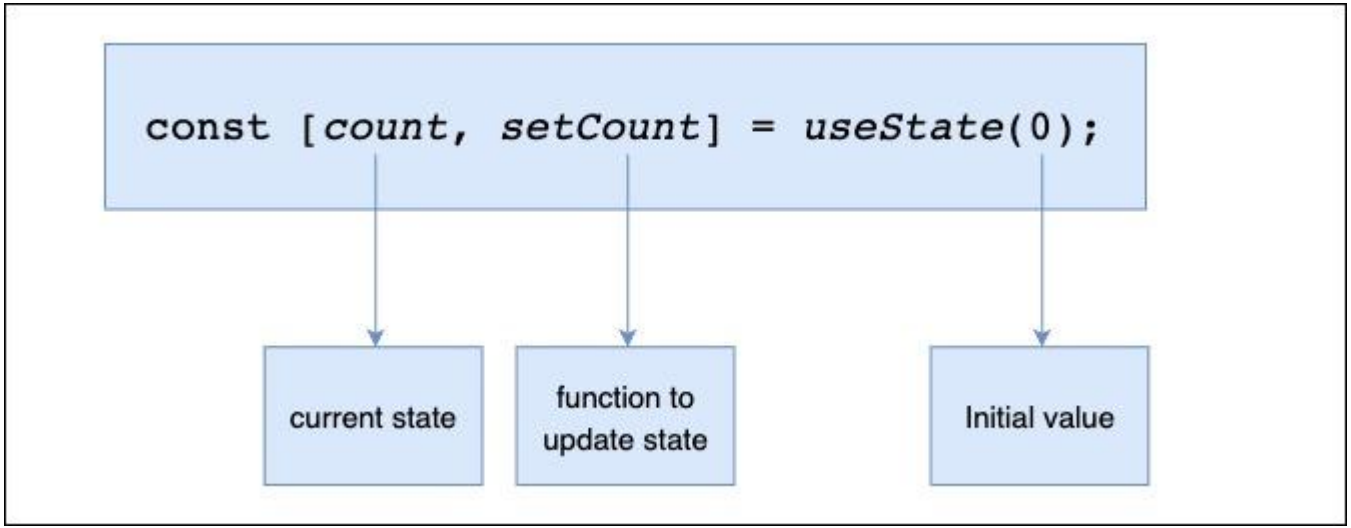
Routing in React

- React Router DOM is a library that provides a set of higher-level routing components that are specifically designed for web applications using React.
- Routing in React is the process of determining which components should be rendered based on the current URL.
- React Router is a popular library used to handle routing in React applications. It allows users to navigate between pages without requiring a full page reload, and is essential for building complex React applications with multiple pages.
- BrowserRouter, Routes, Route, path, element.
- 1. Add BrowserRouter - index.js
- 2. Add routes and route into app.jsx file

Hooks - useState()

- useState is a built-in hook in React that allows you to add state to function components.
- By using useState, you can make your function components stateful and update the UI in response to changes in the state.

const [state, function] = useSate(“initial Value”)


const [count, setCount] = useState(0);
current state
function to update state
Initial value

Hook - useEffect

- useEffect is a hook in React that allows you to run side effects in function components.
- Side effects refer to any changes made outside the scope of the current function component. This can include manipulating the DOM, fetching data from an API.
- It takes two arguments: a function that contains the side effect code, and an array of dependencies that the effect depends on.
- The effect function is called after the component has rendered and whenever any of the dependencies have changed.
- useEffect(() => { console.log(“Hello world”) } , [user]);

Type 1 :- No dependency

- useEffect(( ) => { console.log(“called”) })
- It’ll Execute the function on every render.
- Render - on initial page load on browser, whenever any state changes.
- Ex. Timer, Session period.

Type 2 :- Empty dependencies

- useEffect(( ) => { console.log(“called”) } , [ ])
- It’ll Execute the function only on very first render.
- Ex. Welcome message, Auth check.

Type 3 :- Single dependency

- useEffect(( ) => { console.log(“called”) } , [counter])
- It’ll Execute the function when counter updates and very first render.
- Ex. To save user typed data.

Type 4 :- Multiple dependencies

- useEffect(( ) => { console.log(“called”) } , [counter1, counter2])
- It’ll Execute the function whenever any dependency will updates and at very first render.

Why useEffect executing function twice ?

- StrictMode renders components twice (on dev but not production) in order to detect any problems with your code and warn you about them (which can be quite useful).

React Fragment

- A React Fragment is a component in React that allows you to group a list of children without creating an extra DOM element.
- It lets you return multiple elements from a component's render method without having to wrap them in a parent element.
- <>

  </>

useNavigate

- This hook help to navigate from one component to another component,

Dynamic routing and useParams();

Declarative Way

- Instead of manually manipulating the DOM to make changes to the user interface, React provides a declarative syntax to describe the desired state of the UI. The developer simply defines the components and their properties, and React updates the UI accordingly.
- This declarative approach makes it easier to write and maintain complex applications. It also helps to improve performance, as React uses a virtual DOM to efficiently update only the necessary parts of the UI when changes are made.
- Overall, the declarative way of programming in React allows developers to focus on what the UI should look like, rather than how to update it.

```
const list = document.createElement('ul');
items.forEach((item) => {
  const li = document.createElement('li');
  li.textContent = item;
  list.appendChild(li);
});
document.body.appendChild(list);
```

```
function List({ items }) {
  return (
    <ul>
      {items.map((item) => (
        <li >{item}</li>
      ))}
    </ul>
  );
}
ReactDOM.render(<List items={['Item 1', 'Item 2', 'Item 3']} />, document.getElementById('root'));
```

Wrapper Component

In React, a wrapper component is a component that wraps another component and provides additional functionality or props. This is often used to share common functionality or styling between multiple components.

Using wrapper components allows you to encapsulate common styles or behavior and reuse them across multiple components, providing a convenient way to apply consistent styling or add shared functionality in your React application.

```
import React from 'react';

function withWrapper(WrappedComponent) {
  return function WrapperComponent(props) {
    return (
      <div style={{ backgroundColor: 'lightgray', padding: '20px' }}>
        <WrappedComponent {...props} />
      </div>
    );
  };
}

function MyComponent(props) {
  return <div>Hello, {props.name}!</div>;
}

const MyComponentWithWrapper = withWrapper(MyComponent);

function App() {
  return <MyComponentWithWrapper name="John" />;
}

export default App;
```

Writing More Complex JSX Code

Type 1 : Use conditional rendering:

You can conditionally render elements or components based on some condition using the ternary operator or the logical AND operator. For example:7

```
10 == 10  ? “yes”  : “no”
```

```
const MyComponent = ({ isLoggedIn }) => {
  return (
    <div>
      {isLoggedIn ? <p>Welcome, user!</p> : <p>Please log in.</p>}
    </div>
  );
};
```

Type 2 : Use array.map() for dynamic rendering:

```
const MyListComponent = ({ items }) => {
  return (
    <ul>
      {items.map((item) => (
        <li key={item.id}>{item.name}</li>
      ))}
    </ul>
  );
};
```

Type 3 : Use props to pass data and callbacks:

```
const MyParentComponent = () => {
  const handleClick = () => {
    console.log('Button clicked!');
  };
  return <MyChildComponent text="Click me" onClick={handleClick} />;
};
const MyChildComponent = ({ text, onClick }) => {
  return <button onClick={onClick}>{text}</button>;
};
```

Type 4 : Use CSS-in-JS libraries for styling:

```
import styled from 'styled-components';
const MyStyledComponent = styled.div`
  color: red;
  font-size: 16px;
`;
const MyComponent = () => {
  return <MyStyledComponent>Hello, world!</MyStyledComponent>;
};
```

Working with State & Events in React

- In functional components, you can use the useState hook to manage state and the useEffect hook to handle side effects and events. Here's an example of using state and events in a functional component.

```
import React, { useState, useEffect } from 'react';
const Counter = () => {
  const [count, setCount] = useState(0);
  const handleIncrement = () => {
    setCount(count + 1);
  };
  const handleDecrement = () => {
    setCount(count - 1);
  };
  useEffect(() => {
    console.log('Count updated:', count);
  }, [count]);
  return (
```

- In this example, we define a Counter component that uses the useState hook to maintain its own state. We define two state variables, count and setCount, that represent the current count value and a function to update it, respectively.
- We also define two methods, handleIncrement and handleDecrement, that update the count value using the setCount function when called.
- In the useEffect hook, we log a message to the console whenever the count value changes. The useEffect hook takes a function and an array of dependencies as arguments. The function is called after each render, and the array of dependencies specifies which variables to watch for changes.
- In the return statement, we display the current count value using the {count} syntax. We also render two buttons, each with an onClick event handler that calls the handleIncrement or handleDecrement method when clicked.
- When the button is clicked, the corresponding event handler method is called, which updates the state using the setCount function. The setCount function takes a new count value as an argument and updates the state, triggering a re-render of the component.
- By using hooks in functional components, you can achieve the same functionality as class components with less boilerplate code and a more concise syntax.

React Basic CSS styling

Type 1 : Inline styling

You can apply inline styles to a React component using the style attribute. The value of this attribute should be an object with the CSS properties as keys and their corresponding values as values.

```
import React from 'react';

const MyComponent = () => {
  const style = {
    backgroundColor: 'blue',
    color: 'white',
    padding: '10px',
    borderRadius: '5px',
  };
  return (
    <div style={style}>
      <h1>Hello, World!</h1>
      <p>This is a paragraph.</p>
    </div>
  );
};

export default MyComponent;
```

Type 2 : CSS classes

- You can define CSS classes in a separate CSS file and then apply them to React components using the className attribute.

Jsx file

```
import React from 'react';
import './MyComponent.css';
const MyComponent = () => {
  return (
    <div className="my-component">
      <h1>Hello, World!</h1>
      <p>This is a paragraph.</p>
    </div>
  );
};
export default MyComponent;
```

Css file

```
.my-component {
  background-color: blue;
  color: white;
  padding: 10px;
  border-radius: 5px;
}
```

Render

- It is a method provided by the React DOM library that renders a React element into the DOM.
- The method takes two arguments:
- The first argument is the React element to be rendered, typically created using JSX or React.createElement().
- The second argument is the container where the element should be rendered, typically a DOM element such as a div.

```
import React from 'react';
import ReactDOM from 'react-dom';
const App = () => {
  return <h1>Hello, World!</h1>;
};
ReactDOM.render(<App />, document.getElementById('root'));
```

- In this example, we define a functional component App that returns a simple h1 element with the text "Hello, World!".
- We then call ReactDOM.render with the App component as the first argument and a DOM element with an id of root as the second argument. This will render the component inside the specified container element.
- Note that ReactDOM.render only needs to be called once to render the entire React application. Subsequent changes to the application's state or props will trigger a re-render of the affected components, but the initial render is done using this method.

Rendering Lists

To render a list of items in React, you can use the map() method to create an array of React elements based on an array of data.

```
import React from 'react';
const App = () => {
  const items = ['Apple', 'Banana', 'Orange'];
  return (
    <ul>
      {items.map((item) => (
        <li key={item}>{item}</li>
      ))}
    </ul>
  );
};
export default App;
```

Conditional Content:

To conditionally render content in React, you can use the ternary operator or the logical && operator.

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

const App = () => {
  const [isLoggedIn, setIsLoggedIn] = useState(false);
  return (
    <div>
      {isLoggedIn ? (
        <p>Welcome, user!</p>
      ) : (
        <button onClick={() => setIsLoggedIn(true)}>Log in</button>
      )}
    </div>
  );
};

export default App;
```

Using Stateful Lists

Using stateful lists in React involves managing the state of a list of items and updating the UI based on changes to that state.

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

const List = () => {
  const [items, setItems] = useState(['Apple', 'Banana', 'Orange']);
  const [newItem, setNewItem] = useState('');
  const handleInputChange = (event) => {
    setNewItem(event.target.value);
  };
  const handleAddItem = () => {
    setItems([...items, newItem]);
    setNewItem('');
  };
  return (
    <div>
      <ul>
        {items.map((item, index) => (
          <li key={index}>{item}</li>
        ))}
      </ul>
      <input type="text" value={newItem} onChange={handleInputChange} />
      <button onClick={handleAddItem}>Add Item</button>
    </div>
  );
};
```

Outputting Conditional Content in react

To output conditional content in React, you can use conditional rendering techniques like the ternary operator or the logical && operator.

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

const App = () => {
  const [isLoggedIn, setIsLoggedIn] = useState(false);
  return (
    <div>
      {isLoggedIn ? (
        <p>Welcome, user!</p>
      ) : (
        <button onClick={() => setIsLoggedIn(true)}>Log in</button>
      )}
    </div>
  );
};
export default App;
```

Adding Conditional Return statement

In React, you can add conditional return statements to conditionally render different components or content based on some conditions.

```
import React from 'react';

const Greeting = ({ name, isLoggedIn }) => {
  if (isLoggedIn) {
    return <h1>Welcome back, {name}!</h1>;
  } else {
    return <h1>Please log in.</h1>;
  }
};

export default Greeting;
```

Adding Dynamic Styles

- To add dynamic styles to a React component, you can make use of the style prop.
- The style prop accepts an object of CSS properties and values, and applies those styles to the component.
- You can also generate the styles dynamically based on props or state using JavaScript expressions.

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

const MyComponent = () => {
  const [backgroundColor, setBackgroundColor] = useState('red');
  const handleButtonClick = () => {
    setBackgroundColor('blue');
  };
  const styles = {
    backgroundColor,
    color: 'white',
    padding: '10px',
    borderRadius: '5px',
    cursor: 'pointer'
  };
  return (
    <div style={styles} onClick={handleButtonClick}>
      Click me to change background color!
    </div>
  );
};

export default MyComponent;
```

Setting CSS Classes Dynamically

- To set CSS classes dynamically in React, you can use the className prop.
- The className prop accepts a string of space-separated CSS class names, and applies those classes to the component.
- You can also generate the class names dynamically based on props or state using JavaScript expressions.

```
import React, { useState } from 'react';
const MyComponent = () => {
  const [isButtonActive, setIsButtonActive] = useState(false);
  const handleButtonClick = () => {
    setIsButtonActive(!isButtonActive);
  };
  const buttonClassName = isButtonActive ? 'active-button' : 'inactive-button';
  return (
    <button className={buttonClassName} onClick={handleButtonClick}>
      {isButtonActive ? 'Active' : 'Inactive'}
    </button>
  );
};
export default MyComponent;
```

```
.active-button {
  background-color: green;
  color: white;
  border: none;
  padding: 10px;
  border-radius: 5px;
  cursor: pointer;
}
.inactive-button {
  background-color: gray;
  color: white;
  border: none;
  padding: 10px;
  border-radius: 5px;
  cursor: not-allowed;
}
```

Props in react

- In React, props (short for properties) are used to pass data from a parent component to its child components.

```
import React from 'react';

import Greeting from './Greeting';

const App = () => {

  return <Greeting name="Alice" />;

};

export default App;
```

```
import React from 'react';

const Greeting = (props) => {
  return <h1>Hello, {props.name}!</h1>;
};

export default Greeting;
```

Adding normal JS logic to components

- You can add normal JavaScript logic to your React components by writing JavaScript code inside the component's body or in separate functions that are called from within the component.

```
import React, { useState } from 'react';
const MyComponent = () => {
  const [showText, setShowText] = useState(false);
  const handleClick = () => {
    setShowText(!showText);
  };
  return (
    <div>
      <button onClick={handleClick}>Toggle Text</button>
      {showText && <p>Some text to show</p>}
    </div>
  );
};
export default MyComponent;
```

Splitting a component into multiple components

- Splitting a component into multiple smaller components is a common practice in React to improve reusability and maintainability of the code.

```
import React from 'react';

const Header = () => {
  return <h1>My App</h1>;
};

const Navbar = () => {
  return (
    <nav>
      <ul>
        <li>Home</li>
        <li>About</li>
        <li>Contact</li>
      </ul>
    </nav>
  );
};

const Content = () => {
  return (
    <div>
      <h2>Content Title</h2>
      <p>Some content goes here...</p>
    </div>
  );
};

const Footer = () => {
```

Children prop (Concept of Composition) in react

- In React, the children prop is a special prop that allows you to pass child components to a parent component.
- This is known as the concept of composition, which is a key principle in React for building reusable and modular components.

```
import React from 'react';
const Button = ({ children }) => {
  return (
    <button>
      {children}
    </button>
  );
};
const App = () => {
  return (
    <div>
      <Button>Click Me</Button>
    </div>
  );
};
export default App;
```

Organizing Component Files

- In this example, we have grouped the components by functionality. The "Header" folder contains all the components related to the header of the website, such as the logo, menu, and search bar.
- The "Product" folder contains all the components related to displaying and interacting with product information, such as a list of products, product details, and reviews.
- The "Cart" folder contains all the components related to the shopping cart functionality, such as the cart itself, cart items, and a summary of the cart. Finally, the "Footer" folder contains all the components related to the footer of the website, such as contact information and useful links.
- The top-level "App.js" component is the main entry point for the application.

```
src/
├── components/
│   ├── Header/
│   │   ├── Header.js
│   │   ├── HeaderLogo.js
│   │   ├── HeaderMenu.js
│   │   └── HeaderSearch.js
│   ├── Product/
│   │   ├── Product.js
│   │   ├── ProductList.js
│   │   ├── ProductDetails.js
│   │   └── ProductReviews.js
│   ├── Cart/
│   │   ├── Cart.js
│   │   ├── CartItem.js
│   │   └── CartSummary.js
│   └── Footer/
│       ├── Footer.js
│       ├── FooterContact.js
│       └── FooterLinks.js
└── App.js
```

- Header
- Footer
- Login
- Register

Working with Forms & User inputs in react

```
import React, { useState } from 'react';
function ContactForm() {
  const [firstName, setFirstName] = useState('');
  const [lastName, setLastName] = useState('');
  const [email, setEmail] = useState('');
  const [message, setMessage] = useState('');
  const handleSubmit = (event) => {
    event.preventDefault();
    alert(“Data Submitted”)
  };
  return (
```

Dealing With Form Submission & Getting User Input Values

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


function ContactForm() {
  const [formData, setFormData] = useState({
    firstName: '',
    lastName: '',
    email: '',
    message: '',
  });
  const handleSubmit = (event) => {
    event.preventDefault();
    console.log(formData);
  };
  const handleInputChange = (event) => {
    setFormData({
      ...formData,
      [event.target.name]: event.target.value,
    });
  };
  return (
```

Adding Basic Validation

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

function ContactForm() {
  const [formData, setFormData] = useState({
    firstName: '',
    lastName: '',
    email: '',
    message: '',
  });
  const [formErrors, setFormErrors] = useState({});
  const handleSubmit = (event) => {
    event.preventDefault();
    // Check for form errors
    const errors = validateForm();
    if (Object.keys(errors).length === 0) {
      // No errors, submit the form
      console.log(formData);
    } else {
      // Set the form errors
      setFormErrors(errors);
    }
```

Class

In JavaScript, a class is a template for creating objects that share the same properties and methods. It is a way to define a new type of object, based on a set of properties and behaviors that are common to all instances of that type.

```
class Person {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }
  sayHello() {
    console.log(`Hello, my name is ${this.name} and I am ${this.age} years old.`);
  }
}
const john = new Person('John', 30);
john.sayHello(); // logs "Hello, my name is John and I am 30 years old."
```

Class based components

- Class components are defined as JavaScript classes that extend the React.Component class.
- They are created using the class keyword and have a render method that returns a React element.
- React.Component is a base class that provides the functionality necessary for creating class components.
- When you write class Counter extends React.Component, you are creating a new class called Counter that extends React.Component. This means that the Counter class will inherit all of the methods and properties of React.Component, including the render() method, which is used to render the component to the screen.

- In this example, MyComponent is a class component that extends the React.Component class. It has a render method that returns a React element, which in this case is a heading that says "Hello, world!"

```
import React from 'react';

class MyComponent extends React.Component {

  render() {

    return <h1>Hello, world!</h1>;

  }

}
```

Pass Props to Class Component

```
import React from 'react';
class Greeting extends React.Component {
  render() {
    return (
      <div>
        <h1>Hello, {this.props.name}!</h1>
        <p>{this.props.message}</p>
      </div>
    );
  }
}
export default Greeting;
```

- <Greeting name="John" message="Welcome to my website" />,

Props - <Counter initialCount={5} />

```
import React from 'react';

class Counter extends React.Component {

  constructor(props) {

    super(props);

    this.state = { count: props.initialCount };

  }

  handleClick() {

    this.setState({ count: this.state.count + 1 });

  }

  render() {

    return (

      <div>

        <p>Count: {this.state.count}</p>

        <button onClick={() => this.handleClick()}>Click me!</button>

      </div>

    );

  }

}
```

Counter Application by using Class component

```
import React from 'react';

class Counter extends React.Component {

  constructor(props) {

    super(props);

    this.state = { count: 0 };

  }

  incrementCount = () => {

    this.setState({ count: this.state.count + 1 });

  }

  decrementCount = () => {

    this.setState({ count: this.state.count - 1 });

  }

  render() {

    return (

      <div>

        <h1>Count: {this.state.count}</h1>

        <button onClick={this.incrementCount}>Increment</button>

        <button onClick={this.decrementCount}>Decrement</button>

      </div>

    );

  }
```

Class based Component Lifecycle

- In React, class components have several lifecycle methods that allow developers to control the sequence of events during a component's lifecycle. These methods can be divided into three phases: mounting, updating, and unmounting.

1. Mounting Phase.
2. Updating Phase.
3. Unmounting Phase.

Mounting Phase.

- constructor(): It is called when a component is initialized and is used for setting the initial state and binding methods to the component.
- static getDerivedStateFromProps(): It is called before the initial render method and also before the new props are received. It is used to update the state of the component based on the props.
- render(): It is called after the getDerivedStateFromProps() method and is used to create the component's structure.
- componentDidMount(): It is called after the component is rendered to the DOM and is used to perform any side effects, such as making API requests, updating the document title, etc.

Updating Phase.

- static getDerivedStateFromProps(): It is called when the component is updated and receives new props.
- shouldComponentUpdate(): It is called when the component is about to update and is used to determine whether the component should update or not. It returns a boolean value.
- render(): It is called after the shouldComponentUpdate() method and is used to create the component's structure.
- componentDidUpdate(): It is called after the component is updated and is used to perform any side effects, such as making API requests, updating the document title, etc.

Unmounting Phase

- componentWillUnmount(): It is called just before the component is unmounted from the DOM and is used to clean up any resources used by the component, such as removing event listeners, cancelling API requests, etc.

1. constructor(props):

The constructor() method is used to initialize the component's state. This method is called when the component is first created. It is used to initialize the component's state.

```
import React from 'react';

class Counter extends React.Component {

  constructor(props) {

    super(props);

    this.state = { count: 0 };

  }

  incrementCount = () => {

    this.setState({ count: this.state.count + 1 });

  }

  decrementCount = () => {

    this.setState({ count: this.state.count - 1 });

  }

  render() {

    return (

      <div>

        <h1>Count: {this.state.count}</h1>

        <button onClick={this.incrementCount}>Increment</button>

        <button onClick={this.decrementCount}>Decrement</button>
```

2. componentDidMount():

This method is called after the component has been added to the DOM (Mounting Phase). It is used to perform any necessary setup that requires access to the DOM, such as fetching data from an API or setting up event listeners.

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      data: null
    };
  }
  componentDidMount() {
    fetch('/api/data')
      .then(response => response.json())
      .then(data => this.setState({ data }));
  }
  render() {
    return (
      <div>
        <p>{this.state.data}</p>
```

3. shouldComponentUpdate(nextProps, nextState):

This method is called before the component is updated. It is used to determine whether the component should be re-rendered. By default, this method returns true, but it can be overridden to optimize performance.

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      count: 0
    };
    this.handleClick = this.handleClick.bind(this);
  }

  handleClick() {
    this.setState(prevState => ({
      count: prevState.count + 1
    }));
  }

  shouldComponentUpdate(nextProps, nextState) {
```

4. componentDidUpdate(prevProps, prevState):

This method is called after the component has been updated. It is used to perform any necessary cleanup or additional updates.

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      count: 0
    };
    this.handleClick = this.handleClick.bind(this);
  }

  handleClick() {
    this.setState(prevState => ({
      count: prevState.count + 1
    }));
  }

  componentDidUpdate(prevProps, prevState) {
    if (this.state.count !== prevState.count) {
      console.log(`Count changed from ${prevState.count} to ${this.state.count}`);
```

5. componentWillUnmount():

This method is called when the component is about to be removed from the DOM. It is used to perform any necessary cleanup, such as removing event listeners or cancelling API requests.

```
import React from 'react';

class MyComponent extends React.Component {

  constructor(props) {

    super(props);

    this.state = { count: 0 };

    this.intervalId = null;

    this.handleClick = this.handleClick.bind(this);

  }

  handleClick() {

    this.intervalId = setInterval(() => {

      console.log('Interval tick');

    }, 1000);

  }

  componentWillUnmount() {

    clearInterval(this.intervalId);

    console.log('Component unmounted');

  }
```

Registeration by Class Component

```
import React from 'react';


class RegistrationForm extends React.Component {
  constructor(props) {
    super(props);

    this.state = {
      username: '',
      password: '',
      confirmPassword: ''
    };
  }

  handleInputChange = (event) => {
    const { name, value } = event.target;

    this.setState({
      [name]: value
```

Super()

- In JavaScript, the super keyword is used to call the constructor of a parent class from a subclass.
- In React, when you create a class component, you typically extend the React.Component class to inherit its functionality.
- When you define a constructor in a subclass, you need to call the constructor of the parent class using super before you can use this.
- This is because the super keyword is used to initialize the parent class and its properties before you can use them in the subclass.

Building a First Custom Functional Component

```
import React from 'react';
function MyComponent(props) {
  return (
    <div>
      <h1>{props.title}</h1>
      <p>{props.description}</p>
    </div>
  );
}
export default MyComponent;
```

React State

In React, state and events are closely related. State refers to the current data of a component, while events are actions that occur in response to user interactions or other triggers.

Events can be used to modify the state of a component, which in turn causes the component to re-render.

To add state to a component in React, you can use the useState() hook. The useState() hook takes an initial value for the state and returns an array containing the current value of the state and a function that can be used to update the state.

React Event

React events are similar to native DOM events, but they are slightly different in syntax. Instead of using onclick, for example, you would use onClick.

Working with Event Handlers

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

function MyComponent() {
  const [count, setCount] = useState(0);
  function handleClick() {
    setCount(count + 1);
  }
  return (
    <div>
      <p>You clicked the button {count} times.</p>
      <button onClick={handleClick}>Click me!</button>
    </div>
  );
}

export default MyComponent;
```

Working with "State"

- In React, state is an object that stores data that can change over time and affect the way a component is rendered.
- It's a way for components to manage their own data and update their rendering when the data changes.
- To use state in a functional component, you can use the useState hook, which is a function provided by React.

A Closer eye at "useState" Hook

- The useState hook in React is used to manage state in functional components. It allows you to define a state variable and a function to update that state variable. Here's the syntax for using the useState hook:
- const [stateVariable, setStateFunction] = useState(initialState);
- The useState hook returns an array with two elements: the first element is the current value of the state variable (which we've named stateVariable above), and the second element is a function (which we've named setStateFunction) that can be used to update the state variable.
- When you call setStateFunction, React will update the value of the state variable, and then trigger a re-render of the component so that the new state value is reflected in the UI.

Child-to-Parent Component Communication

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

function ChildComponent(props) {

  const [inputValue, setInputValue] = useState('');

  function handleChange(event) {

    setInputValue(event.target.value);

    props.onChange(event.target.value);

  }

  return (

    <div>

      <label>

        Input value:

        <input type="text" value={inputValue} onChange={handleChange} />

      </label>

    </div>

  );

}

function ParentComponent() {

  const [parentValue, setParentValue] = useState('');

  function handleChildChange(childValue) {

    setParentValue(childValue);
```

Controlled vs Uncontrolled Component

Controlled Component

Controlled functional components are those where the value of the input field is controlled by the component state. The component state is managed using React hooks, such as the useState hook.

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

function ControlledInput() {
  const [value, setValue] = useState('');
  const handleChange = (event) => {
    setValue(event.target.value);
  };
  const handleSubmit = (event) => {
    alert('A name was submitted: ' + value);
    event.preventDefault();
  };
  return (
    <form onSubmit={handleSubmit}>
      <label>
        Name:
        <input type="text" value={value} onChange={handleChange} />
      </label>
```

Uncontrolled Component

Uncontrolled functional components are those where the value of the input field is managed by the DOM. In other words, the input field is not tied to the component state, and any changes made to the input field are directly reflected in the DOM.

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

function UncontrolledInput() {
  const inputRef = useRef(null);
  const handleSubmit = (event) => {
    alert('A name was submitted: ' + inputRef.current.value);
    event.preventDefault();
  };
  return (
    <form onSubmit={handleSubmit}>
      <label>
```

Adding a "Not Found" Page

- Adding a "Not Found" page in a React application is important for providing a user-friendly experience when a user navigates to a route that doesn't exist.

1. Create a new component for the "Not Found" page:

```
import React from 'react';

function NotFound() {
  return (
    <div>
      <h2>Not Found</h2>
      <p>The page you're looking for doesn't exist.</p>
    </div>
  );
}

export default NotFound;
```

2. Import the "Not Found" component into your App.js file:

```
import React from 'react';
import { BrowserRouter as Router, Switch, Route } from 'react-router-dom';
import NotFound from './NotFound';
import Home from './Home';
import About from './About';
function App() {
  return (
    <Router>
      <Switch>
        <Route exact path="/" component={Home} />
        <Route path="/about" component={About} />
        <Route component={NotFound} />
      </Switch>
    </Router>
  );
}
export default App;
```

Working with APIs(React)

- Working with APIs in a React application involves making HTTP requests to retrieve or send data to a server.
- There are several ways to make HTTP requests in React, but the most common approach is to use the fetch() function, which is built into modern web browsers.

```
import React, { useState, useEffect } from 'react';
function MyComponent() {
  const [data, setData] = useState([]);
  const [error, setError] = useState(null);
  useEffect(() => {
    const fetchData = async () => {
      try {
        const response = await fetch('https://api.example.com/my-data');
        if (!response.ok) {
          throw new Error('Network response was not ok');
        }
        const data = await response.json();
        setData(data);
```

try/catch

- The try/catch statement is a way to handle errors in JavaScript. It allows you to wrap a block of code in a try block, and if any errors are thrown during the execution of that block, the code in the corresponding catch block will be executed.
- In the context of the code I provided, the try/catch block is being used to catch any errors that occur during the fetch() request and set the error state accordingly.

```
try{


 }catch(error){

console.log()

}
```

new

- The new keyword is used in JavaScript to create an instance of an object from a constructor function.
- In the context of the code I provided, the new Error() statement is creating a new instance of the Error object with a custom error message.
- This is then thrown in the catch block to create a new error object and pass it to the setError() function.

Sending a GET Request using fetch (class based component)

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

class MyComponent extends Component {
  componentDidMount() {
    fetch('https://example.com/api/data')
      .then(response => response.json())
      .then(data => console.log(data))
      .catch(error => console.error(error));
  }
  render() {
    return <div>Fetching data...</div>;
  }
}
```

Handling loading of data building a loader

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

function MyComponent() {

  const [data, setData] = useState(null);

  const [loading, setLoading] = useState(true);

  useEffect(() => {

    async function fetchData() {

      try {

        const response = await fetch('https://example.com/api/data');

        const json = await response.json();

        setData(json);

        setLoading(false);

      } catch (error) {

        console.error(error);

      }

    }

    fetchData();

  }, []);

  if (loading) {

    return <div>Loading data...</div>;

  }

  return (
```

Using useEffect() For handling Requests

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

function MyComponent() {
  const [data, setData] = useState(null);
  const [error, setError] = useState(null);
  useEffect(() => {
    async function fetchData() {
      try {
        const response = await fetch('https://example.com/api/data');
        const json = await response.json();
        setData(json);
      } catch (error) {
        setError(error);
      }
    }
    fetchData();
  }, []);
  if (error) {
    return <div>Error: {error.message}</div>;
  }
  if (!data) {
    return <div>Loading data...</div>;
```

Step to Deploy React Project on Github

1. Run command called, npm run build.
2. Login into netlify website, Then go on sites section, scrool to bottom, Drap and drop that build which is created by Step 1 (or alos you can brows).
3. Click on submit…
4. Wait for a min until you can see Published and click on the link….
5. Go on to Domain setting,
6. Inside Production domains, click on option and edit the site name…

Sending a POST Request

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

function MyComponent() {
  const [formData, setFormData] = useState({ name: '', email: '' });

  async function handleSubmit(event) {
    event.preventDefault();
    try {
      const response = await fetch('https://example.com/api/user', {
        method: 'POST',
        headers: {
          'Content-Type': 'application/json'
        },
        body: JSON.stringify(formData)
      });
      const json = await response.json();
      console.log(json);
    } catch (error) {
      console.error(error);
    }
  }

  function handleChange(event) {
    setFormData({
```

Intro React Context API

- The React Context API is a feature in React that allows you to manage and share state data across components without the need for prop drilling (passing props through multiple levels of components).
- It provides a way to create global or local data stores that can be accessed by any component within a React application.
- With the Context API, you can define a context object that holds the shared data and functions.
- This context object is then used to provide and consume the data within components.

1. Create a Context:

Start by creating a new context using the createContext function from the react package. This function returns a context object that has a Provider and a Consumer.

```
import React from 'react';


const MyContext = React.createContext();
```

2. Provide the Context:

Wrap the parent component or a higher-level component with the Provider component from the context object. The Provider component accepts a value prop that holds the data you want to share.

```
<MyContext.Provider value={/* Your shared data */}>
  {/* Child components */}
</MyContext.Provider>
```

3. Consume the Context:

In any child component that needs access to the shared data, use the Consumer component from the context object. The Consumer component uses a render prop pattern, where you provide a function as the child that receives the context value as an argument.

```
<MyContext.Consumer>
  {value => (
    /* Use the context value in your component */
  )}
</MyContext.Consumer>
```

4. Access the Context using useContext (optional):

If you are using React version 16.8 or above, you can leverage the useContext hook to access the context value directly within a functional component.

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


const value = useContext(MyContext);

/* Use the context value in your component */
```

React Context Limitations

- Performance: Using React Context can slow down your app if you have many components relying on it. It can cause unnecessary re-renders, even if the data they need hasn't changed. This can impact the app's speed, especially in larger projects.
- Updates: When the context value changes, all components using that context will re-render by default. This can be inefficient if only a few components actually need the updated data. You can optimize this manually or use techniques like memoization to improve performance.
- Complexity: As your app grows, managing multiple contexts and their interactions can become complex. It can be challenging to debug issues or understand how data flows through the context. This complexity increases as your project becomes more extensive.
- Global state management: While React Context can be used for managing global app state, it may not be the best choice for complex state management. Dedicated state management libraries like Redux or MobX offer additional features and a more structured approach to handle global state.

Refs:

- Refs provide a way to access and interact with DOM elements or class components directly.
- They allow you to reference a specific element or component instance and access its properties or methods.
- Refs are commonly used to focus input fields, play/pause media, measure DOM elements, or interact with third-party libraries that require direct access to the DOM.
- Refs can be created using the createRef method or the useRef hook.
- In the below example, the useRef hook is used to create a ref that references the input element. When the button is clicked, the handleClick function is called, which focuses the input field using the inputRef.current.focus() statement.

```
import React, { useRef } from 'react';
function MyComponent() {
  const inputRef = useRef(null);
  const handleClick = () => {
    inputRef.current.focus();
  };
  return (
    <div>
      <input ref={inputRef} type="text" />
      <button onClick={handleClick}>Focus Input</button>
    </div>
  );
}
```

Diving into "Forward Refs"

- "Forwarding refs" in React is a technique that allows you to pass a ref from a parent component to one of its child components.
- It enables you to access and interact with the child component's underlying DOM element or class component instance directly.
- Forwarding refs allows you to maintain a clean and encapsulated component structure while still enabling direct interaction between parent and child components.
- It's particularly useful when you need to access specific functionality or data of a child component without exposing it to other parts of the application.

To implement forward refs, you need to follow these steps:

- Create a ref in the parent component using either createRef() or useRef().
- Pass the ref as a prop to the child component.
- In the child component, use the React.forwardRef() function to define the component. This function takes a render function as its parameter, along with the ref as the second parameter.
- In the child component's render function, attach the ref to the desired DOM element or class component using the ref attribute.

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

// Child component that forwards the ref
const ChildComponent = forwardRef((props, ref) => {
  return <input ref={ref} />;
});

// Parent component
const ParentComponent = () => {
  const inputRef = useRef(null);
  const focusInput = () => {
    inputRef.current.focus();
  };
  return (
    <div>
      <ChildComponent ref={inputRef} />
      <button onClick={focusInput}>Focus Input</button>
    </div>
  );
};
```

Component Updates In Action

- Component updates in React refer to the process of re-rendering components when changes occur in their props or state.
- When a component's props or state changes, React automatically triggers a re-rendering of that component and its child components to reflect the updated data.
- When a parent component re-renders due to changes in its props or state, React also re-evaluates the child components.
- This process ensures that the child components receive the updated data and reflect any changes accordingly.

Here's how component updates work in action:

1. Initial Render:

   When a component is first rendered, React calls the component's render method to create the initial UI representation.

   React creates a virtual DOM representation of the component's UI, including its child components.

2. Prop or State Change:

   If the component's props or state changes, either from internal updates or from parent component updates, React identifies that the component needs to be re-rendered.

   React compares the new props or state with the previous values to determine if there are any differences.

3. Reconciliation:

   React performs a process called reconciliation to update the component's virtual DOM representation efficiently.

   During reconciliation, React compares the previous virtual DOM with the new virtual DOM to identify the minimal set of changes needed to update the actual DOM.

   React updates only the parts of the DOM that have changed, minimizing the impact on performance.

4. Re-rendering:

   After the reconciliation process, React re-renders the component by calling its render method again.

   The component's render method returns a new virtual DOM representation based on the updated props or state.

   React updates the actual DOM with the changes identified during reconciliation.

5. Child Component Updates:

   If a component has child components, React repeats the same process of reconciliation and re-rendering for each child component.

   React ensures that the child components are updated in the correct order based on their dependencies and the changes in their props or state.

6. Lifecycle Methods:

   During the update process, React invokes certain lifecycle methods of the component, such as componentDidUpdate, which you can use to perform additional actions or side effects after the update.

   Lifecycle methods provide hooks into the component's update process, allowing you to manage any necessary cleanup, perform additional computations, or interact with external APIs.

Preventing Unnecessary Re-Evaluations with React.memo()

- In React, the React.memo() function is used to optimize functional components by preventing unnecessary re-evaluations.
- It's a higher-order component (HOC) that memoizes the result of a component's rendering, caching it and skipping re-renders if the component's props remain the same.

```
// Memoized version of ListItem

const MemoizedListItem = React.memo(ListItem);
```

memo() - function

useMemo() - hook

```
import { useMemo, useState } from "react";


const Memo = () => {
    const [count, setCount] = useState(0);
    const [todos, setTodos] = useState([]);
    // const calculation = expensiveCalculation(count);
    const calculation = useMemo(() => expensiveCalculation(count), [count]);


    const increment = () => {
        setCount((c) => c + 1);
    };
    const addTodo = () => {
        setTodos((t) => [...t, "New Todo"]);
    };


    return (
        <div>
            <div>
```

memo()

- memo() is a higher-order component provided by React. It is used to wrap the Todos component in order to memoize its rendering.
- When a component is wrapped with memo, React will optimize its rendering by performing a shallow comparison of the component's props. If the props have not changed, React will reuse the previously rendered result and skip the re-rendering process. This can help improve the performance of the component by preventing unnecessary re-renders when its props remain the same.
- export default memo(Todos);
- By applying memo to the Todos component, React will only re-render it when there are changes to its props (todos and addTodo). If there are no changes in the props, the previously rendered result will be reused.
- The memo function takes the component as an argument and returns a memoized version of the component. It's an optimization technique that can be used for functional components to prevent unnecessary re-renders when the props haven't changed.

UseCallback() and its Dependencies

- useCallback() is a React hook used for memoizing functions. It returns a memoized version of the callback function that only changes if one of the dependencies has changed.
- This can be useful for optimizing performance in scenarios where the callback function is passed down to child components, as it prevents unnecessary re-creations of the function.
- The useCallback() hook takes two arguments: the callback function and an array of dependencies.
- The callback function is the function that you want to memoize, and the dependencies array specifies the values that the callback function depends on.
- If any of the dependencies in the array change, the memoized callback function will be re-created; otherwise, it will be reused from the previous render.

```
import { useState, useCallback } from "react";
import ReactDOM from "react-dom/client";
import Todos from "./Todos";
const App = () => {
  const [count, setCount] = useState(0);
  const [todos, setTodos] = useState([]);
  const increment = () => {
    setCount((c) => c + 1);
  };
  const addTodo = useCallback(() => {
    setTodos((t) => [...t, "New Todo"]);
  }, [todos]);
  return (
    <>
```

```
import { memo } from "react";
const Todos = ({ todos, addTodo }) => {
  console.log("child render");
  return (
    <>
      <h2>My Todos</h2>
      {todos.map((todo, index) => {
        return <p key={index}>{todo}</p>;
      })}
      <button onClick={addTodo}>Add Todo</button>
    </>
  );
};
export default memo(Todos);
```

State Scheduling & Batching

- State Scheduling & Batching in React refers to the process of batching multiple state updates into a single update to optimize rendering performance.
- By default, React applies state updates immediately and triggers re-rendering. However, to improve performance, React utilizes a concept called "state batching" or "deferred updates."
- Instead of applying each state update immediately, React batches multiple state updates together and performs a single re-rendering process.

State batching provides several benefits:

- Performance optimization: Batching multiple state updates reduces the number of re-renders, improving overall performance by avoiding unnecessary render cycles.
- Prevents redundant rendering: When multiple state updates occur within a single event handler or lifecycle method, React batches them together and performs a single re-render. This helps avoid intermediate renders and updates to the DOM.
- Improved efficiency: Batching state updates reduces the workload on the JavaScript engine and can lead to better overall application performance.

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

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

  const handleClick = () => {
    // Multiple state updates within a single event handler
    setCount(count + 1);
    setCount(count + 1);
    setCount(count + 1);
  };

  console.log('Render');

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={handleClick}>Increment</button>
    </div>
  );
}

export default Counter;
```

- In this example, we have a Counter component that maintains a count state using the useState hook. When the button is clicked, the handleClick event handler is triggered, which updates the count state three times in a row.
- If you run this code and click the "Increment" button, you'll notice that the console output displays "Render" only once, even though we called setCount three times. This is because React batches the state updates together and performs a single re-render.
- React's state batching mechanism ensures that the latest value of count is used for all state updates, even though the updates are called in quick succession. This behavior helps prevent redundant rendering and improves performance.

UseReducer Hook

- The useReducer Hook is similar to the useState Hook.
- It allows for custom state logic.
- If you find yourself keeping track of multiple pieces of state that rely on complex logic, useReducer may be useful.
- The useReducer Hook accepts two arguments.
- useReducer(<reducer>, <initialState>)
- The reducer function contains your custom state logic and the initialState can be a simple value but generally will contain an object.
- The useReducer Hook returns the current state and a dispatch method.

```
import React, { useReducer } from "react";

// Reducer function
const reducer = (state, action) => {
  switch (action.type) {
    case "INCREMENT":
      return { count: state.count + 1 };
    case "DECREMENT":
      return { count: state.count - 1 };
    case "RESET":
      return { count: 0 };
    default:
      return state;
  }
};
const Counter = () => {
  // Initial state
  const initialState = { count: 0 };
```

UseReducer vs useState for State Management

useState:

- useState is a simple and straightforward hook that allows you to add state to your functional components.
- It takes an initial value and returns an array with two elements: the current state value and a function to update the state.
- The state is managed independently for each individual state variable created using useState.
- Updating the state using useState replaces the previous state with the new value.

useReducer:

- useReducer is a more advanced hook that follows the concept of the reducer pattern from Redux.
- It is useful when the state logic becomes more complex and involves multiple sub-values or actions.
- It takes a reducer function and an initial state and returns the current state and a dispatch function to update the state.
- The reducer function receives the current state and an action and returns the new state based on the action.
- The state is updated based on the dispatched actions, and the reducer determines how the state changes.
- useReducer is especially useful when the state transitions depend on the previous state or when the state updates are more intricate.

In summary, here are some factors to consider when choosing between useState and useReducer:

- Use useState when the state management is simple and doesn't involve complex state transitions or actions.
- Use useReducer when the state logic becomes more complex, involves multiple sub-values, or requires intricate state updates based on actions.
- useReducer is typically more suitable for managing larger and more complex state in your application.
- useState is generally easier to use and has a simpler syntax compared to useReducer.
- If your component's state management is simple and doesn't require complex state updates or actions, useState is likely sufficient and more straightforward.
- However, if your component's state logic becomes more complex or involves multiple state variables and actions, useReducer can provide a more organized and scalable approach.

It's important to choose the appropriate state management approach based on the specific needs and complexity of your application's state.

Introducing React Portals:

React Portals provide a way to render components outside of the normal DOM hierarchy of your React application. It allows you to render components into a different part of the DOM tree, such as a container outside of the current component's parent or in the root of the document.

Here are some key points to understand about React Portals:

- Usage: To create a portal, you use the ReactDOM.createPortal(child, container) method. The child is the component you want to render, and the container is a DOM element where you want to render the component.
- Rendering Position: The component rendered through a portal can be positioned anywhere in the DOM hierarchy, even outside of the parent component's DOM structure. This allows you to render components in a different part of the document, such as modals, popovers, or tooltips.

- Event Bubbling: Events from the portal component will bubble up through the normal React event system. This means that event handlers on parent components can still capture and handle events from the portal component.

- Context: The portal component will still have access to the context of the parent component, allowing you to pass down context values and access them in the portal component.

- Limitations: React Portals cannot break out of the parent's DOM tree completely due to security restrictions. They are still subject to the same origin policy, which means you can't use portals to render content into a different domain.

React Portals provide a powerful mechanism for rendering components in different parts of the DOM while still maintaining the benefits of React's component-based architecture. They are particularly useful for creating overlays, modals, or other UI components that need to be rendered outside of the normal component hierarchy.

Here are a few use cases for ReactDOM.createPortal:

- Modal Dialogs: Modals are UI components that typically overlay the main content and require rendering outside of the main component hierarchy. By using ReactDOM.createPortal, you can render the modal content into a separate container, such as the modal-root element, which is typically placed at the end of the document.
- Tooltips: Tooltips are small informational or contextual overlays that appear when the user hovers over an element. To render a tooltip outside of the hovered element's hierarchy, you can use ReactDOM.createPortal to render the tooltip content in a separate container, such as the tooltip-root element.
- Portals for Accessibility: Portals can also be used for improving the accessibility of certain UI components. For example, you can render a screen reader-only message outside of the component hierarchy using ReactDOM.createPortal, ensuring that it's read by screen readers without affecting the visual rendering of the component.

Building custom React Hook

React hooks are a way to reuse stateful logic between functional components in React. They allow you to extract and share stateful logic without the need for class components.

To create a custom React hook, you'll need to follow a few conventions:

1. The hook name should start with "use" to indicate that it's a hook.
2. The hook should be a function that can take in parameters.
3. The hook should return values or functions that can be used within a React component.

Let's create an example custom React hook called useCounter, which will provide a counter value and functions to increment and decrement the counter.

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

function useCounter(initialValue = 0) {
  const [count, setCount] = useState(initialValue);
  const increment = () => {
    setCount(count + 1);
  };
  const decrement = () => {
    setCount(count - 1);
  };
  return [count, increment, decrement];
}
```

```
import React from 'react';
import useCounter from './useCounter';
function Counter() {
  const [count, increment, decrement] = useCounter();
  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={increment}>Increment</button>
      <button onClick={decrement}>Decrement</button>
    </div>
  );
}
```

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

function useLocalStorage(key, initialValue) {
  const [value, setValue] = useState(() => {
    const storedValue = localStorage.getItem(key);
    return storedValue ? JSON.parse(storedValue) : initialValue;
  });
  useEffect(() => {
    localStorage.setItem(key, JSON.stringify(value));
  }, [key, value]);
  return [value, setValue];
}
```

```
import React from 'react';
import useLocalStorage from './useLocalStorage';
function MyComponent() {
  const [name, setName] = useLocalStorage('name', '');
  const handleChange = (event) => {
    setName(event.target.value);
  };
  return (
    <div>
      <input type="text" value={name} onChange={handleChange} />
      <p>Hello, {name}!</p>
    </div>
  );
}
```

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

function useOnlineStatus() {
  const [isOnline, setIsOnline] = useState(navigator.onLine);
  useEffect(() => {
    const handleOnline = () => {
      setIsOnline(true);
    };
    const handleOffline = () => {
      setIsOnline(false);
    };
    window.addEventListener('online', handleOnline);
    window.addEventListener('offline', handleOffline);
    return () => {
      window.removeEventListener('online', handleOnline);
      window.removeEventListener('offline', handleOffline);
    };
  }, []);
  return isOnline;
}
```

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

function MyComponent() {
  const isOnline = useOnlineStatus();
  return (
    <div>
      {isOnline ? (
        <p>Online</p>
      ) : (
        <p>Offline</p>
      )}
    </div>
  );
}
```

Redux

Redux is a predictable state container for JavaScript applications. It's commonly used with libraries like React, but it can be used with any JavaScript framework or library. Redux helps manage the state of your application in a centralized and predictable way.

Create-react-app

npm install redux react-redux

Here are the key concepts you need to understand when working with Redux:

1. Store: The store holds the global state of your application. It's a JavaScript object that contains all the data that your application needs. You can think of it as a single source of truth.
2. Actions: Actions are plain JavaScript objects that represent an event or intention in your application. They are dispatched to the Redux store, triggering changes to the state. Actions have a type property that describes the type of action being performed.
3. Reducers: Reducers specify how the application's state changes in response to actions. They are pure functions that take the current state and an action as parameters and return a new state. Reducers should not modify the state directly but create a new state object.
4. Dispatch: Dispatching an action means sending it to the Redux store. This is typically done through a dispatch function provided by Redux. The store then calls the reducers, which update the state based on the action.
5. Selectors: Selectors are functions that retrieve specific pieces of state from the Redux store. They provide an abstraction layer to access the state in a more convenient and efficient way.
6. Middleware: Middleware sits between the dispatching of an action and the moment it reaches the reducer. It can intercept actions and perform additional logic, such as logging, async operations, or modifying the action itself before it reaches the reducers.

To get started with Redux, you'll need to install it as a dependency in your project. You can do this by running npm install redux or yarn add redux in your project directory.

Once you have Redux installed, you can create a store, define your actions and reducers, and start dispatching actions to update the state of your application.

Here's a basic example of how you would use Redux with React:

- Create your actions: Define action types and action creators, which are functions that return actions.
- Create your reducers: Write pure functions that handle the state changes based on the actions dispatched.
- Create your store: Use the createStore function from Redux, passing in your reducers. This will create the Redux store.
- Connect your components: Use the connect function from the react-redux library to connect your components to the Redux store. This allows your components to access the state and dispatch actions.
- Dispatch actions: Inside your components, you can dispatch actions using the dispatch function provided by Redux. This triggers the reducers, which update the state accordingly.

Steps for Redux

1. Install dependencies - npm install redux react-redux
2. Create Action -
3. Create Reducer -

How Redux Works + State Management

What is state?

In programming, "state" refers to the data that represents the current condition or snapshot of your application. It can include things like user inputs, UI states, or any other data that changes over time.

Why do we need state management?

As your application grows, managing and updating state becomes more challenging. State management libraries like Redux provide a structured way to handle and update state, making it easier to track changes, share data between components, and maintain consistency throughout your app.

What is Redux?

Redux is a popular state management library for JavaScript applications, commonly used with frameworks like React. It provides a predictable and centralized way to manage state, making it easier to understand how data changes occur and enabling better debugging and testing.

Key concepts in Redux:

- **Store**: The store is a centralized container that holds the entire state of your application. It's created using the createStore function from Redux.
- **Actions**: Actions are plain JavaScript objects that describe an intention to change the state. They have a type property that defines the type of action being performed and can include additional data called the payload.
- **Reducers**: Reducers are pure functions responsible for handling actions and updating the state accordingly. They take in the current state and an action, and return a new state based on the action's type.

- **Dispatch**: Dispatching an action is the process of sending the action to the store. It triggers the state update flow, where the reducers analyze the action and update the state accordingly.
- **Subscribe**: Subscribing allows components to be notified whenever the state changes. Components can subscribe to the store and receive updates whenever the state is updated.

The Redux flow in simple terms:

Your application's components dispatch actions, which describe what change they want to make.

The actions flow through the reducers, which update the state based on the actions' type.

The updated state is then available to all subscribed components, triggering re-renders and ensuring data consistency.

Remember, Redux is most beneficial for larger applications with complex state requirements. For smaller projects or learning purposes, you might start with simpler state management approaches provided by your chosen framework, such as React's useState hook.

To start using Redux, you'll need to install the necessary dependencies (redux and react-redux), set up your store with reducers, create actions, dispatch those actions from your components, and connect your components to the store using the connect function or hooks provided by react-redux.

Don't worry if it feels overwhelming at first. With practice and hands-on coding experience, you'll gradually become more comfortable with Redux and state management concepts.

Redux vs React Context

1. **Complexity:**

- **Redux**: Redux is a more comprehensive and powerful state management library. It has a more complex setup compared to React Context. Redux introduces concepts like actions, reducers, and the store, which provide a structured way to manage state. It's well-suited for large-scale applications with complex state requirements.
- **React Context**: React Context is built into React and provides a simpler and more lightweight approach to managing state. It allows you to create a centralized state container and share state across components without the need for additional libraries or setup.

2. **Centralized vs. Local State:**

- **Redux**: Redux promotes a centralized state management approach, where the entire application state is stored in a single global store. Components can access and update the state through actions and reducers.

- **React Context**: React Context allows you to create local state containers specific to certain components or component hierarchies. Each context instance manages its own state, which is accessible to the components within its scope. This makes it more suitable for managing smaller-scale or localized state.

3. **Scalability and Flexibility:**

- **Redux**: Redux is highly scalable and provides excellent support for handling complex state interactions, asynchronous actions, and middleware integration. It has a large ecosystem of middleware, dev tools, and extensions, making it flexible and extensible.

- **React Context**: React Context is simpler and more straightforward, but it may become less optimal when dealing with deeply nested components or complex state interactions. It lacks some advanced features and middleware options available in Redux.

4. **Learning Curve:**

- **Redux**: Redux has a steeper learning curve due to its additional concepts and boilerplate code. Understanding the Redux flow, actions, reducers, and connecting components to the store can take some time for beginners.

- **React Context**: React Context is relatively easier to grasp and has a shallower learning curve. It leverages familiar React concepts like context providers and consumers, making it more accessible for beginners.

5. **Use Cases:**

- **Redux**: Redux is well-suited for large applications with complex state management needs, such as applications with extensive data flow, multiple interconnected components, or a need for time-travel debugging. It shines in scenarios where state changes are frequent and need to be tracked more closely.

- **React Context**: React Context is great for simpler state management needs, such as sharing data across a few related components or implementing theme switching, user authentication, or language selection. It's useful when you want to avoid prop drilling and keep the state localized to specific components.

Redux Actions

In Redux, actions are like messengers that carry information to tell the application how the state should change.

Actions are simple JavaScript objects with a type field that describes the type of action being performed.

You can also include additional data or payload in the action to provide more information.

Actions are typically created using action creator functions, which are functions that return action objects.

```
const incrementCounter = () => {
  return {
    type: 'INCREMENT_COUNTER'
  };
};
```

Reducer

Reducers are responsible for handling actions and updating the state of the application.

Reducers are pure functions, which means they don't have side effects and always produce the same output for the same input.

Reducers take in the current state and an action as parameters, and return a new state based on the action.

It's important to remember that reducers should not mutate the existing state, but rather create a new state object with the desired changes.

```
const initialState = {
  counter: 0
};
const counterReducer = (state = initialState, action) => {
  switch (action.type) {
    case 'INCREMENT_COUNTER':
      return {
        ...state,
        counter: state.counter + 1
      };
    default:
      return state;
  }
};
```

Step 1 : Define the initial state:

```
const initialState = {
  count: 0
};
```

Step 2 : Create action types:

```
const INCREMENT = 'INCREMENT';
const DECREMENT = 'DECREMENT';
```

Step 3 : Create action creators:

```
const increment = () => {
  return {
    type: INCREMENT
  };
};
const decrement = () => {
  return {
    type: DECREMENT
  };
};
```

Step 4 : Create the reducer:

```
const counterReducer = (state = initialState, action) => {
  switch (action.type) {
    case INCREMENT:
      return {
        ...state,
        count: state.count + 1
      };
    case DECREMENT:
      return {
        ...state,
        count: state.count - 1
      };
    default:
      return state;
  }
};
```

Step 5 : Set up the Redux store:

```
import { createStore } from 'redux';

const store = createStore(counterReducer);
```

Step 6 : Dispatch actions to update the state:

```
store.dispatch(increment()); // Dispatch an increment action

console.log(store.getState()); // Output: { count: 1 }

store.dispatch(decrement()); // Dispatch a decrement action

console.log(store.getState()); // Output: { count: 0 }
```

Redux Store

Redux is a state management library commonly used with JavaScript applications, particularly those built with frameworks like React. At the core of Redux is the concept of a "store," which is an object that holds the application state.
To create a Redux store, you need to follow a few steps. First, you need to install the Redux library using a package manager like npm or Yarn. Then, you can import the necessary functions and set up the store.

```
// Import the required functions from Redux
import { createStore } from 'redux';
// Define a reducer function
const reducer = (state = 0, action) => {
  // Handle different types of actions
  switch (action.type) {
    case 'INCREMENT':
      return state + 1;
    case 'DECREMENT':
      return state - 1;
    default:
      return state;
  }
};
```

```
// Create the Redux store
const store = createStore(reducer);

// Subscribe to changes in the store
store.subscribe(() => {
  console.log('Current state:', store.getState());
});
// Dispatch actions to update the state
store.dispatch({ type: 'INCREMENT' });
store.dispatch({ type: 'INCREMENT' });
store.dispatch({ type: 'DECREMENT' });
```

In the code above, we first import the createStore function from the Redux library. Then, we define a reducer function, which takes the current state and an action as parameters and returns the updated state based on the action type. In this example, the initial state is set to 0, and the reducer handles two types of actions: 'INCREMENT' and 'DECREMENT'.

Next, we create the Redux store by calling createStore and passing in the reducer function. The store holds the application state and provides methods to interact with it.

To get notified of changes in the store, we subscribe to it by using the store.subscribe() method and providing a callback function that will be called whenever the state changes. In this example, we simply log the current state to the console.

Finally, we dispatch actions to update the state using the store.dispatch() method. In this case, we dispatch three actions: two 'INCREMENT' actions and one 'DECREMENT' action. Each time an action is dispatched, the reducer is called, and the state is updated accordingly.

Store Provider

In Redux, the Provider component is a higher-order component (HOC) provided by the react-redux library. It is used to wrap your application's root component and provide the Redux store to all the components in your application.

```
import React from 'react';
import ReactDOM from 'react-dom';
import { Provider } from 'react-redux';
import { createStore } from 'redux';
import rootReducer from './reducers';
import App from './App';
// Create the Redux store
const store = createStore(rootReducer);
ReactDOM.render(
  <Provider store={store}>
    <App />
  </Provider>,
  document.getElementById('root')
);
```

In the code above, we first import the necessary dependencies: react, react-dom, Provider from react-redux, createStore from Redux, your root reducer (rootReducer), and your main application component (App).

Next, we create the Redux store using the createStore function and passing in your root reducer.

Then, we wrap the App component with the Provider component, passing the store as a prop to the Provider. This makes the Redux store available to all the components in the App component hierarchy.
Finally, we render the wrapped App component using ReactDOM.render() and specify the target DOM element where the application should be mounted (in this case, an element with the id of 'root').

By wrapping your application with the Provider component and providing the Redux store, you enable all the components within your application to access the store and interact with the state using Redux's connect function or hooks like useSelector and useDispatch.

Redux connect()

In Redux, the connect() function is a higher-order function provided by the react-redux library. It allows components to connect to the Redux store and access the state and dispatch actions.

The connect() function takes two parameters: mapStateToProps and mapDispatchToProps. These parameters define how the component will interact with the Redux store.

```
import React from 'react';
import { connect } from 'react-redux';
// Define your component
const MyComponent = ({ count, increment }) => {
  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={increment}>Increment</button>
    </div>
  );
};
```

```
// Define mapStateToProps function
const mapStateToProps = (state) => {
  return {
    count: state.count,
  };
};
// Define mapDispatchToProps function
const mapDispatchToProps = (dispatch) => {
  return {
    increment: () => dispatch({ type: 'INCREMENT' }),
  };
};
// Connect the component to the Redux store
export default connect(mapStateToProps, mapDispatchToProps)(MyComponent);
```

In the code above, we define a simple functional component called MyComponent. This component displays a count value from the Redux store and has a button to increment the count.

To connect the component to the Redux store, we use the connect() function. First, we define a mapStateToProps function, which takes the Redux store's state as a parameter and returns an object with the props that we want to map from the state. In this example, we map the count property from the Redux store's state to the count prop of the component.

Next, we define a mapDispatchToProps function, which takes the dispatch function as a parameter and returns an object with the action creators that we want to map to props. In this example, we define an increment action creator that dispatches an 'INCREMENT' action when called.

Finally, we export the component by wrapping it with the connect() function, passing in the mapStateToProps and mapDispatchToProps functions as arguments. This connects the component to the Redux store, allowing it to access the mapped state and action creators as props.

Now, the MyComponent component can access the count prop from the Redux store's state and call the increment action creator to dispatch an action and update the state.

Redux Middleware

- Redux middleware is a powerful feature that allows you to add additional functionality to the Redux dispatch process.
- It sits between the dispatch of an action and the point at which the action reaches the reducer. Middleware can intercept and modify actions, dispatch new actions, or perform asynchronous operations.

```
import { createStore, applyMiddleware } from 'redux';
import thunk from 'redux-thunk';

// Define a reducer function
const reducer = (state = 0, action) => {
  switch (action.type) {
    case 'INCREMENT':
      return state + 1;
    case 'DECREMENT':
      return state - 1;
    default:
      return state;
  }
};
```

```
// Create the Redux store with middleware
const store = createStore(reducer, applyMiddleware(thunk));
// Define an async action creator using thunk middleware
const incrementAsync = () => {
  return (dispatch) => {
    setTimeout(() => {
      dispatch({ type: 'INCREMENT' });
    }, 1000);
  };
};
// Dispatch an async action
store.dispatch(incrementAsync());

// Subscribe to changes in the store
store.subscribe(() => {
  console.log('Current count:', store.getState());
});
```

- In this example, we start by importing the necessary functions from Redux: createStore and applyMiddleware. We also import the redux-thunk middleware.
- We define a simple reducer function that increments or decrements the counter state based on the action type.
- To create the Redux store with middleware, we use the applyMiddleware function and pass in redux-thunk as the middleware argument.

- Next, we define an async action creator function called incrementAsync. It uses the thunk middleware to allow us to dispatch a function instead of a plain action object. Inside the dispatched function, we use setTimeout to simulate an asynchronous operation, and after one second, we dispatch an action of type 'INCREMENT'.
- We dispatch the incrementAsync action, which triggers the asynchronous operation.
- We also subscribe to changes in the store and log the current count to the console whenever the state updates.
- When you run this code, you will see that the counter is incremented after one second, showcasing the asynchronous behavior handled by the redux-thunk middleware.

Steps to use Middleware in redux

1. npm i react-redux redux redux-thunk
2. Inside the index.js - Import Provider, createStore, applyMiddleware, thunk
3. Create store by using reducer and thunk , use createStore and applyMiddleware methods
4. Add Provider (HOF) to add store to the all component to the App..
5. Create Reducer : use initial state as state and action, and define switch case
6. Inside App :
    - Import connect from react-redux
    - Create async action and import it..
    - For Connect() create two function to access state and increment
    - Create function to access state and another function to access dispatch
    - Add component to connect() method.
    - Now Component can access state and dispatch from redux.

Redux async actions

- Redux is a popular state management library used in JavaScript applications, often in conjunction with React.
- It provides a predictable state container that helps manage the state of an application in a consistent manner.
- Redux async actions refer to actions that involve asynchronous operations, such as making API requests or performing asynchronous computations.
- These actions typically have multiple stages, including initiating the asynchronous operation, handling loading or error states, and updating the state with the retrieved data.
- To handle async actions in Redux, you typically use a middleware like Redux Thunk or Redux Saga. These middleware allow you to write action creators that return functions instead of plain action objects.
- This enables you to perform asynchronous operations inside the action creators before dispatching the actual actions.

Steps :

1. Index.js

   Provider - HOF,

   Pass store to provider

2. store.js

Redux Hooks

- Redux Hooks are a set of hooks provided by the react-redux library that allow you to interact with Redux state and dispatch actions in functional components.
- They provide a more intuitive and streamlined way to work with Redux in functional components without the need for higher-order components (HOCs) or the connect function. The main Redux Hooks are useSelector, useDispatch, and useStore.
- useSelector: Selects and accesses data from the Redux store in a component.
- useDispatch: Provides access to the dispatch function of the Redux store to dispatch actions.
- useStore: Provides direct access to the Redux store instance.
- By using Redux Hooks, you can simplify and improve the readability of your code when working with Redux in functional components.

Introduction to NodeJS & NPM

- Node.js is a server-side JavaScript runtime environment that allows you to run JavaScript code on the server.
- It is event-driven, scalable, and efficient, making it suitable for building various types of server-side applications.
- With a vibrant ecosystem and extensive package manager (NPM), Node.js enables developers to write server-side code in JavaScript and leverage existing libraries and modules.
- It is cross-platform and widely adopted in the web development community.

Why to use it?

1. JavaScript Everywhere: Node.js allows developers to use JavaScript on both the front-end and back-end, which promotes code reusability, reduces the learning curve, and simplifies the development process.
2. Fast and Scalable: Node.js is built on the V8 JavaScript engine, which is known for its high-performance execution. It uses an event-driven, non-blocking I/O model, making it highly scalable and able to handle a large number of concurrent requests.
3. Efficient Development: Node.js has a vast ecosystem of open-source libraries and modules available through NPM. These modules provide ready-to-use functionalities, saving development time and effort. Additionally, Node.js has a rich set of tools and frameworks that enhance productivity and code maintainability.

4. Real-time Applications: Node.js is well-suited for building real-time applications such as chat applications, collaborative tools, gaming servers, and streaming platforms. Its event-driven architecture allows for efficient handling of real-time communication and data synchronization.

5. Microservices and APIs: Node.js provides a lightweight and modular approach to building microservices and APIs. Its small footprint and low resource consumption make it ideal for developing scalable and distributed systems.

6. High Community Support: Node.js has a large and active community of developers, which means you can find plenty of resources, tutorials, and support when working with Node.js. This vibrant community ensures continuous development and improvement of the platform.

Using Core Modules

In Node.js, core modules are pre-installed modules that provide essential functionalities for various tasks. These modules are part of the Node.js runtime environment and can be used without the need for additional installation. Here are some commonly used core modules in Node.js:

1. fs (File System): This module provides methods for working with the file system, allowing you to read, write, and manipulate files and directories.
2. http and https: These modules enable you to create HTTP and HTTPS servers or make HTTP/HTTPS requests to interact with web servers.
3. path: The path module provides utilities for working with file paths, including resolving, joining, and parsing file paths.

- os (Operating System): This module provides information and methods related to the operating system, allowing you to access details about the system's CPU, memory, network interfaces, and more.

- events: The events module provides an event-driven architecture and allows you to create custom events and event listeners.

- util: The util module contains various utility functions that are commonly used in Node.js applications, such as formatting strings, handling errors, and working with objects.

To use a core module, you need to import it using the require function. For example:

```
const fs = require('fs'); // es 5

Import fs from ‘fs’; // es 6

// Example usage of the fs module

fs.readFile('file.txt', 'utf8', (err, data) => {

  if (err) throw err;

  console.log(data);

});
```

OS Module

The os module in Node.js provides a set of methods and properties for interacting with the operating system. It allows you to retrieve information about the underlying operating system, such as the system's architecture, network interfaces, CPU usage, and more. Here are some commonly used features of the os module:

Retrieving System Information:

1. os.platform(): Returns the platform of the operating system (e.g., "darwin" for macOS, "win32" for Windows).
2. os.release(): Returns the release version of the operating system.
3. os.hostname(): Returns the hostname of the operating system.

Working with CPU Information:

1. os.cpus(): Returns an array of objects containing information about each logical CPU core, such as the model, speed, and times (CPU usage).

Memory Information:

- os.totalmem(): Returns the total amount of system memory in bytes.
- os.freemem(): Returns the amount of free system memory in bytes.

Working with Network Interfaces:

- os.networkInterfaces(): Returns an object containing information about the network interfaces available on the system, including IP addresses and MAC addresses.

Operating System Constants:

- os.constants: Provides an object with various operating system-specific constants, such as error codes and signal names.

```
const os = require('os');

console.log('Platform:', os.platform());

console.log('Release:', os.release());

console.log('Hostname:', os.hostname());

console.log('CPUs:', os.cpus());

console.log('Total Memory:', os.totalmem() / 1024 / 1024, 'MB');

console.log('Free Memory:', os.freemem() / 1024 / 1024, 'MB');

console.log('Network Interfaces:', os.networkInterfaces());

console.log('Constants:', os.constants);
```

File module : Creating & Writing files

- The fs module in Node.js provides a set of methods for interacting with the file system.
- It allows you to create, read, write, and modify files in a Node.js application. One of the common use cases is creating and writing files.
- Here's an example of how to create and write content to a file using the fs module:

```
const fs = require('fs');


const content = 'This is the content to be written to the file.';
// Create a new file and write content to it
fs.writeFile('example.txt', content, (err) => {
  if (err) {
    console.error('Error writing file:', err);
  } else {
    console.log('File created and content written successfully.');
  }
});
```

- In this example, we first require the fs module using require('fs'). Then, we define the content that we want to write to the file.
- The writeFile method is used to create a new file and write content to it. It takes three arguments: the file path (in this case, 'example.txt'), the content to be written, and a callback function to handle the result.
- If an error occurs during the file write operation, the error will be passed as the first argument to the callback function. If the operation is successful, the callback will be called without an error.
- After running this code, a new file named 'example.txt' will be created in the current directory, and the specified content will be written to it. If the file already exists, the existing content will be overwritten.
- Remember to handle errors appropriately and provide proper error handling logic based on your specific requirements.

Events module : Emitting events

- The events module in Node.js provides an event-driven architecture that allows objects to emit and listen to events.
- It is a key component in building event-driven applications in Node.js.
- Here's an example of how to emit events using the events module:

```
const EventEmitter = require('events');
// Create a new instance of EventEmitter
const myEmitter = new EventEmitter();
// Define an event handler function
const eventHandler = () => {
  console.log('Event occurred!');
};
// Attach the event handler to the 'myEvent' event
myEmitter.on('myEvent', eventHandler);
// Emit the 'myEvent' event
myEmitter.emit('myEvent');
```

- In this example, we first require the events module using require('events'). Then, we create a new instance of EventEmitter using new EventEmitter().
- Next, we define an event handler function named eventHandler that will be called when the event is emitted. In this case, it simply logs a message to the console.
- We attach the event handler to the 'myEvent' event using the on method of the EventEmitter instance. The on method takes two arguments: the event name and the event handler function.
- Finally, we emit the 'myEvent' event using the emit method of the EventEmitter instance. This triggers the execution of the attached event handler function.
- When you run this code, you will see the message "Event occurred!" printed to the console.
- This is a basic example of emitting events using the events module. You can create custom events, attach multiple event handlers, and pass data along with events for more complex event-driven programming scenarios.

Event propagation

Event propagation, also known as event bubbling and event capturing, refers to the order in which events are handled when multiple elements are nested within each other and an event is triggered on one of those elements.

There are two phases of event propagation:

- Event Capturing: In this phase, the event is captured and handled by the outermost element first, then propagated inward to the inner elements. This phase traverses down the DOM hierarchy from the root element to the target element.
- Event Bubbling: In this phase, after the event is handled by the target element, it continues to propagate upward to the parent elements and further up the DOM hierarchy. This phase traverses up the DOM hierarchy from the target element to the root element.

  The default behavior in most browsers is event bubbling, where the event first triggers on the target element and then propagates up the DOM tree. However, you can also choose to handle events during the capturing phase by setting the capture option to true when registering event listeners.

```
<script>
const outer = document.getElementById('outer');
const inner = document.getElementById('inner');
const btn = document.getElementById('btn');
outer.addEventListener('click', () => {
  console.log('Outer clicked');
}, false); // Event bubbling
inner.addEventListener('click', () => {
  console.log('Inner clicked');
}, false); // Event bubbling
btn.addEventListener('click', () => {
  console.log('Button clicked');
}, false); // Event bubbling
</script>
```

```
<div id="outer">
  <div id="inner">
    <button id="btn">Click me!</button>
  </div>
</div>
```

In this example, we have an outer , an inner , and a button. We attach event listeners to each element for the click event. When you click the button, you will see the following output in the console:

Button clicked

Inner clicked

Outer clicked

As you can see, the event propagates from the button to the inner and then to the outer , following the event bubbling phase.

Event propagation allows you to handle events at different levels of the DOM hierarchy, providing flexibility in managing event behavior and implementing event-driven functionality in your applications.

Overview of how client-server works

- In client-server architecture, you can think of the server as a big "hub" that stores information or performs tasks, and the client as a user or device that wants to access that information or use those services.
- When a client wants something from the server, it sends a request. This request tells the server what the client wants, like asking for a web page, sending data to be stored, or requesting a specific action to be performed.
- The server receives the request and processes it. It does whatever is needed to fulfill the client's request, which may involve retrieving data, performing calculations, or executing specific functions.

- Once the server has completed its task, it sends a response back to the client. This response contains the information or result that the client requested. It could be a web page, updated data, or a message confirming that the requested action was performed successfully.
- The client then receives the response and uses the information or result as needed. For example, a web browser would display the web page received, or an application would use the data in its operations.
- The communication between the client and server relies on protocols, which are like languages that both the client and server understand. These protocols ensure that the client and server can exchange information effectively and accurately.
- Overall, the client-server architecture allows clients to access the resources and services provided by the server. It's like a "request and response" system, where the client asks for something, the server processes it, and then sends back the desired result.

Introduction to Backend-Development

- Backend development involves creating the hidden, behind-the-scenes components of a software application.
- Backend developers write code that handles data processing, storage, and interaction with users or other systems. They work with databases, build APIs for communication, ensure security, optimize performance, and integrate with external services.
- In summary, backend development focuses on making the application work smoothly and securely, without users directly seeing or interacting with the backend code.

Using third-party modules for backend development involves the following steps:

1. Identify the functionality you need for your backend application.
2. Search for relevant modules in package registries like npm.
3. Install the desired module using a package manager (e.g., npm or Yarn).
4. Import or require the module in your backend code.
5. Utilize the module's functions, classes, or APIs to implement the desired functionality.
6. Keep the module and its dependencies up to date by regularly checking for updates.
7. Manage the module's dependencies using the package manager's capabilities.
8. Consult the module's documentation for detailed instructions on installation, usage, and configuration.

By leveraging third-party modules, you can save development time, enhance your application's functionality, and benefit from the expertise of the module's developers.

What is NPM. (Node Package Manager)

- npm is a package manager for the JavaScript programming language. It is used to manage and install packages and dependencies that are required in a JavaScript project.
- npm is included with Node.js, which is a JavaScript runtime environment that allows developers to run JavaScript code outside of a web browser.
- npx is a tool that is included with npm and is used to execute packages without having to install them globally on your machine. (create-react-app)
- Overall, npx is a convenient way to run scripts from packages without having to install them globally, and is particularly useful when working with create-react-app.
- Commands - npm init, npm install uuid, npm update, npx create-react-app

Creating & Using our own modules

To create and use your own modules for backend development in Node.js, you can follow these steps:

Creating a Module:

1. Choose a meaningful name for your module. It should reflect the functionality it provides.
2. Create a new file with the same name as your module, using the .js file extension.
3. Define the functions, classes, or variables that you want to include in your module within the file.
4. Export the functions, classes, or variables you want to make accessible from the module using the module.exports object.

```
// module_name.js
function toLowerCase(str) {
  return str.toLowerCase();
}
function toUpperCase(str) {
  return str.toUpperCase();
}
module.exports = {
  toLowerCase,
toUpperCase
};
```

Using a Module:

Access the functions, classes, or variables defined in the module using the module name followed by the function or variable name.

```
// app.js

const module_name = require('./module_name');

module_name.myFunction(); // calling a function from the module
```

HTTP Module : Creating a simple web server, Building GET & POST API

Here's another simple example that demonstrates creating a web server using the HTTP module in Node.js. In this example, we'll create a basic API that returns a list of books.

```
const http = require('http');

// Sample book data
const books = [
  { id: 1, title: 'Book 1', author: 'Author 1' },
  { id: 2, title: 'Book 2', author: 'Author 2' },
  { id: 3, title: 'Book 3', author: 'Author 3' }
];

const server = http.createServer((req, res) => {
  if (req.method === 'GET' && req.url === '/api/books') {
    // Handle GET request for books API
    res.writeHead(200, { 'Content-Type': 'application/json' });
    res.end(JSON.stringify(books));
  } else {
    // Handle other requests
    res.writeHead(404, { 'Content-Type': 'text/plain' });
    res.end('Endpoint not found');
  }
});

const port = 3000;
server.listen(port, () => {
  console.log(`Server running on port ${port}`);
});
```

- In this example, the server listens for GET requests on the /api/books endpoint. When a GET request is received for this endpoint, it responds with a JSON representation of the books array.
- To test this API, you can run the script and access http://localhost:3000/api/books in a web browser or use tools like cURL or Postman. It will return the list of books in JSON format.
- Feel free to modify this example to suit your specific requirements. You can add more endpoints, implement additional HTTP methods like POST, DELETE, etc., and incorporate any desired logic to handle the requests accordingly.

Frontend Versus Backend

Frontend and backend are two important parts of web development that work together to create websites and applications.

Frontend:

- Frontend is the part of web development that users interact with directly.
- It involves using languages like HTML, CSS, and JavaScript to build the user interface (UI) of a website or application.
- Frontend developers focus on designing and creating the visual elements, layout, and navigation that users see and interact with.
- They also make sure the website or application works well on different devices and screen sizes.
- Frontend developers use frameworks and libraries like React, Angular, or Vue.js to make the development process easier and enhance UI interactions.

Backend :

- Backend is the part of web development that handles the behind-the-scenes operations.
- It involves using programming languages like Python, Java, or PHP to write the server-side code.
- Backend developers work on tasks like managing data, performing operations on databases, and implementing the business logic of the application.
- They handle things like user authentication, data storage, and integration with external services.
- Backend developers also focus on security measures to protect the application and optimize performance to handle high traffic loads.

Frontend and backend are interconnected. The frontend communicates with the backend through APIs, which allow them to exchange data and information. The backend provides the necessary data and functionality to the frontend, and the frontend presents it to the users in an interactive and user-friendly way.

Remember, both frontend and backend are essential for creating functional and engaging websites and applications. They require different skills and technologies, but together they make a complete and successful web development project.

How requiring module works?

Let's take an example of a JavaScript module that provides some utility functions for working with strings. We'll create a module called stringUtils.js with two functions: capitalize and reverse.

stringUtils.js:

```
// Function to capitalize the first letter of a string
function capitalize(str) {
  return str.charAt(0).toUpperCase() + str.slice(1);
}
// Function to reverse a string
function reverse(str) {
  return str.split('').reverse().join('');
}
// Export the functions to make them accessible to other modules
module.exports = {
  capitalize: capitalize,
  reverse: reverse
};
```

In the above code, we define two functions within the stringUtils.js module: capitalize and reverse. The capitalize function takes a string as input and returns the same string with the first letter capitalized. The reverse function takes a string and returns the reversed version of it.

To make these functions available in another JavaScript file, we need to require the stringUtils module using the require function.

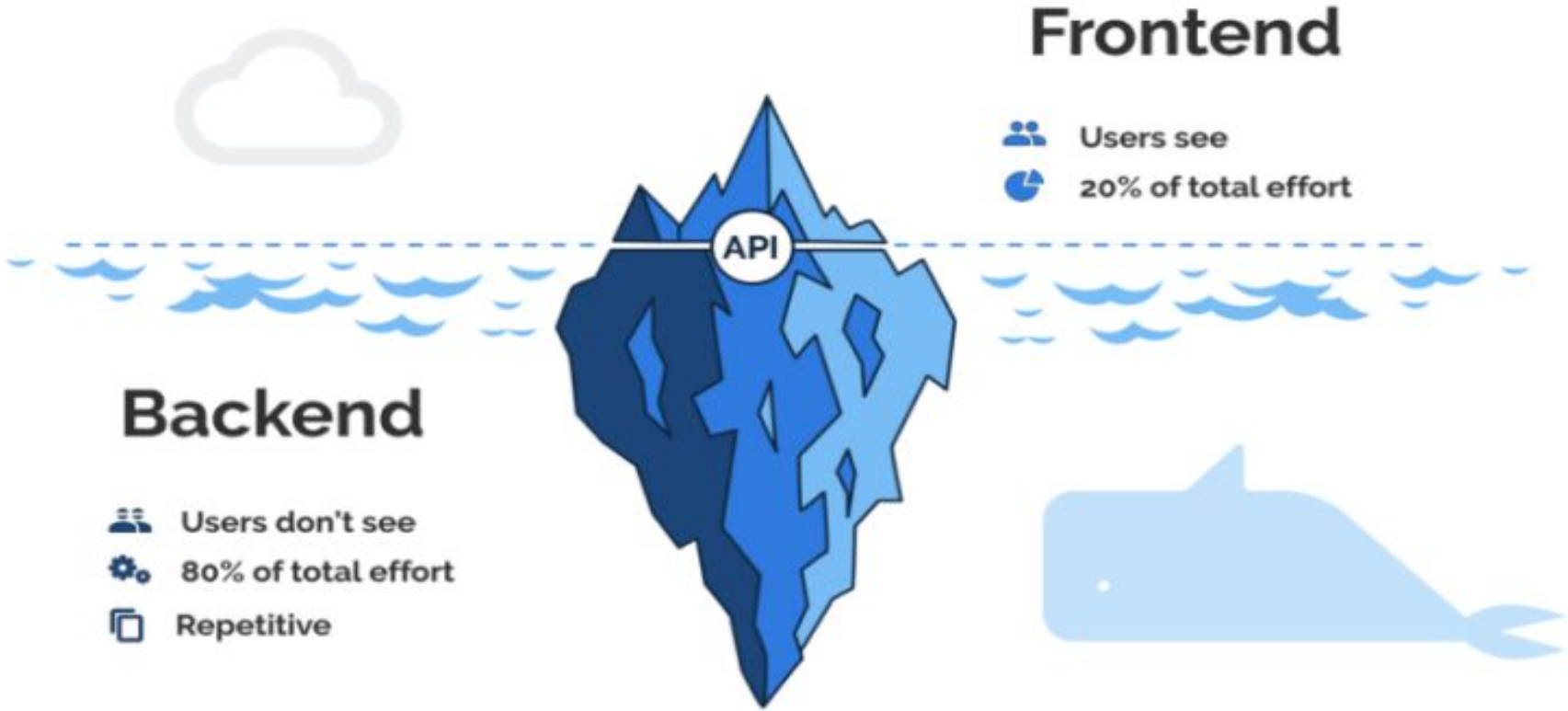
App.js

```
// Require the stringUtils module

const stringUtils = require('./stringUtils');

// Use the functions from the stringUtils module

console.log(stringUtils.capitalize('hello'));  // Output: Hello

console.log(stringUtils.reverse('world'));     // Output: dlrow
```

In the app.js file, we require the stringUtils module by specifying its path using the require function. We assign the imported module to the stringUtils variable. Now, we can access the capitalize and reverse functions from the stringUtils module using the stringUtils variable.

When we run the app.js file, it will output "Hello" and "dlrow" to the console, which are the results of calling the capitalize and reverse functions respectively.

By separating functionality into modules, we can keep our code organized and maintainable. Other parts of the application can easily require and use the specific functions they need, promoting code reuse and modularity.


Frontend
Users see
20% of total effort
API
Backend
Users don't see
80% of total effort
Repetitive

NodeJS Event Loop

Let's illustrate the Node.js Event Loop with a simple example:

```
// Step 1: Import required modules
const fs = require('fs');


// Step 2: Initiate an asynchronous I/O operation
fs.readFile('file.txt', 'utf8', (err, data) => {
  if (err) {
    console.error('Error reading file:', err);
  } else {
    console.log('File contents:', data);
  }
});
```

Event Loop

console.log(“Hi 1”);

setTimeout(()=>{console.log(“Hi 2”)},3000)

console.log(“Hi 3”);

setTimeout(()=>{console.log(“Hi 4”)},1000)

End

Call stack , 1 executed, 2 , 3 executed , 4 , 4 executed, 2 executed

Async stack - 2 wait until 3 sec , 4 wait until 1 sec

Callback queue - 4 ,2 after completion ,

Event loop - continuously checking if call stack is empty, if found empty it’ll take First completed task from callback queue and it’ll add it in call stack..

In this example, we perform various operations to demonstrate the behavior of the Node.js Event Loop:

1. We import the fs module, which provides file system-related functionality in Node.js.
2. We initiate an asynchronous I/O operation using the fs.readFile method. This operation reads the contents of a file named file.txt. When the operation completes, it triggers the provided callback function.
3. We execute a synchronous operation by logging "Synchronous operation: Hello, World!" to the console. Unlike the asynchronous I/O operation, this operation is executed immediately.
4. We set a timer using setTimeout. After one second (1000 milliseconds), the timer expires and the associated callback function is executed, logging "Timer expired: 1 second" to the console.
5. We execute a callback function called callbackFunction directly, which logs "Callback function executed" to the console.

Events and Event-Driven Architecture

Events and Event-Driven Architecture are concepts that revolve around the idea of communication and interaction between components or modules in a software system.

In an event-driven architecture, components or modules communicate by emitting and responding to events. An event represents a specific action or occurrence that has happened within the system. It can be triggered by user actions, system events, or changes in the application's state.

Here are the key elements of an event-driven architecture:

1. Event: An event represents a significant action or occurrence within the system. It encapsulates relevant information about the event and may include additional data or metadata. Events can be predefined or custom-defined for specific purposes.
2. Event Emitter: An event emitter is responsible for emitting or publishing events when certain conditions are met. It acts as a source or origin of events and notifies other components or modules that something has occurred.
3. Event Listener: An event listener, also known as a subscriber or observer, is responsible for listening to specific events emitted by event emitters. It registers itself with the emitter to receive notifications when the associated events occur. Once an event is emitted, the listener can respond or react accordingly.
4. Callback Function: A callback function is a piece of code that is executed when a particular event occurs. It is attached or associated with an event listener and defines the behavior or actions to be performed in response to the event.

Benefits of Event-Driven Architecture:

1. Loose Coupling: Components in an event-driven architecture are loosely coupled, meaning they are independent and can operate without knowledge of each other. They communicate through events, allowing for flexibility and modularity.
2. Scalability: Event-driven architectures are well-suited for handling scalability and high concurrency. They can handle a large number of events and distribute processing across multiple components.
3. Extensibility: By adding or modifying event emitters and event listeners, new functionalities can be easily introduced without impacting the existing components. This promotes extensibility and allows for easy integration of new features.
4. Asynchronous Nature: Events and event-driven architectures inherently support asynchronous operations. Components can respond to events as they occur, allowing the system to be non-blocking and responsive.

Event-driven architecture is commonly used in various software systems, including graphical user interfaces, web applications, real-time systems, and distributed systems. It helps in building modular, scalable, and flexible applications by decoupling components and promoting efficient communication through events.

Event Emitter → Event Listener → Attached callback function

In an event-driven architecture, the flow of communication typically follows a pattern of Event Emitter, Event Listener, and attached callback function. Let's break down how these components interact:

Event Emitter:

An Event Emitter is an entity or object that emits or triggers events. It serves as the source or origin of events within the system. When a specific action or condition occurs, the Event Emitter emits an event, notifying interested parties (Event Listeners) that the event has occurred.

Event Listener:

An Event Listener is an entity or object that listens for specific events emitted by the Event Emitter. It registers itself with the Event Emitter to receive notifications when the desired events occur. The Event Listener specifies the events it wants to listen to and defines the actions it should take when those events are emitted.

Attached Callback Function:

The attached callback function is a function defined by the Event Listener. When an event is emitted by the Event Emitter and received by the Event Listener, the attached callback function associated with that event is executed. The callback function determines the behavior or actions to be performed in response to the event.

```
// Step 1: Create an Event Emitter
const EventEmitter = require('events');
const myEmitter = new EventEmitter();

// Step 2: Register an Event Listener with an Attached Callback Function
myEmitter.on('myEvent', (data) => {
  console.log('Event occurred with data:', data);
});

// Step 3: Emit an Event
myEmitter.emit('myEvent', 'Hello, World!');
```

In this example:

1. We create an instance of the EventEmitter class called myEmitter, which serves as the Event Emitter.
2. We register an Event Listener on myEmitter using the on method. It listens for an event called 'myEvent'. The second argument to on is the callback function that will be executed when 'myEvent' is emitted.
3. We emit an event called 'myEvent' using the emit method on myEmitter. This triggers the event and sends the provided data ('Hello, World!') along with the event.
4. When the 'myEvent' event is emitted, the Event Listener associated with it is triggered. The callback function defined in the Event Listener is executed, and it logs the event data ('Hello, World!') to the console.

The output of running this code will be: "Event occurred with data: Hello, World!"

This example demonstrates how the Event Emitter emits an event, the Event Listener listens for that event, and the attached callback function is executed in response to the event. This pattern allows for decoupling and flexible communication between components in an event-driven architecture.

How NodeJS Works?

- Node.js is a runtime environment that executes JavaScript code outside of a web browser. It uses an event-driven, non-blocking I/O model and the V8 JavaScript engine. It operates on a single thread with an event loop, allowing for concurrent and asynchronous operations.
- Node.js provides a modular approach with its core libraries and npm package manager. It is known for its scalability, performance, and ability to handle numerous concurrent connections.

Here are some additional details on how Node.js works, along with examples:

Event-Driven, Non-Blocking I/O:

Node.js follows an event-driven, non-blocking I/O model, allowing for efficient handling of concurrent operations. For example, when making an HTTP request, Node.js initiates the request and continues executing other tasks without waiting for the response. Once the response is received, a callback function is executed to handle the response. This non-blocking behavior enables Node.js to handle multiple concurrent connections without blocking the execution flow.

```
const http = require('http');

// Initiating an HTTP request

const request = http.get('http://example.com', (response) => {

  // Callback function executed when response is received

  response.on('data', (data) => {

    console.log(data.toString());

  });

});
```

Single-Threaded Event Loop:

Node.js operates on a single thread but utilizes an event loop to handle concurrent tasks efficiently. The event loop continuously iterates over a queue of events, executing their associated callback functions.

```
setTimeout(() => {
  console.log('Timer expired');
}, 1000);
console.log('Before timer');
// Output:
// Before timer
// Timer expired
```

In the above example, the setTimeout function initiates a timer, and the callback function is executed after one second. However, the program doesn't block and continues to execute the next line of code, resulting in the "Before timer" message being logged before the timer expires.

Modules and npm:

Node.js provides a module system that allows developers to organize code into reusable modules. Modules encapsulate specific functionalities and can be imported and used in other modules.

```
// In a module named "math.js"

exports.add = (a, b) => {

  return a + b;

};

// In another module

const math = require('./math.js');

console.log(math.add(2, 3)); // Output: 5
```

In this example, the math.js module exports the add function, which can be imported and used in another module using the require function. This modular approach promotes code organization and reusability.

Asynchronous APIs:

Node.js provides asynchronous APIs for performing I/O operations, such as reading and writing files or making network requests. These APIs allow for concurrent execution without blocking the event loop.

```
const fs = require('fs');

fs.readFile('file.txt', 'utf8', (err, data) => {
  if (err) {
    console.error('Error reading file:', err);
  } else {
    console.log('File contents:', data);
  }
});
```

In this example, the readFile function asynchronously reads the contents of a file. When the operation is complete, the provided callback function is executed with an error (if any) and the data read from the file.

These examples illustrate how Node.js leverages its event-driven, non-blocking I/O model, modules, and asynchronous APIs to provide a scalable and efficient runtime environment for building server-side applications.

Introduction to API & methods

API stands for Application Programming Interface. It is a set of rules and protocols that allows different software applications to communicate with each other. APIs define how different components of software systems should interact, enabling developers to access and utilize the functionalities and data of other applications or services.

APIs provide a standardized way for applications to request specific actions or data from another application or service, hiding the complexities of the underlying implementation. They abstract away the internal workings and provide a simplified interface for developers to interact with.

APIs can expose various methods or operations that define the actions that can be performed. Here are some commonly used methods in APIs:

GET:

The GET method is used to retrieve or fetch data from a specified resource. It is an idempotent and safe operation, meaning it should not have any side effects or modify the data on the server.

POST:

The POST method is used to submit or send data to be processed by a specified resource. It is often used for creating new resources or submitting data to be stored or processed on the server.

PUT:

The PUT method is used to update or replace an existing resource with new data. It replaces the entire resource with the new representation provided in the request.

PATCH:

The PATCH method is used to partially update an existing resource. It allows for modifying specific fields or properties of a resource without replacing the entire representation.

DELETE:

The DELETE method is used to delete or remove a specified resource.

- These methods, along with others like HEAD, OPTIONS, and more, form the foundation of HTTP (Hypertext Transfer Protocol) and RESTful APIs. REST (Representational State Transfer) is an architectural style commonly used in web services, where resources are identified by URLs (Uniform Resource Locators) and can be manipulated using these HTTP methods.
- APIs can also use other protocols and methods specific to their implementation, such as SOAP (Simple Object Access Protocol), GraphQL, or WebSocket.
- In summary, APIs provide a way for applications to interact with each other by defining a set of methods or operations. These methods, such as GET, POST, PUT, PATCH, and DELETE, enable developers to retrieve, create, update, and delete data or perform specific actions on remote services or resources. APIs play a crucial role in integrating different systems and enabling the development of complex and interconnected software applications.

Static vs Dynamic API

Static API:

A Static API, also known as a Fixed API, is an API that has a fixed structure and functionality. It follows a predefined set of endpoints, request formats, and response formats that do not change unless the API provider explicitly updates them. The behavior and available data of a static API remain constant over time.

Static APIs are commonly used when the data or functionality being exposed is relatively stable and doesn't require frequent updates. They are suitable for scenarios where the API consumers do not need real-time or dynamic data.

Example:

Consider an API that provides information about a country, such as its name, capital, population, and area. If this data rarely changes, a static API can be created with fixed endpoints like /country or /country/{countryCode} to fetch specific country details.

Dynamic API:

A Dynamic API, also known as a Live API, is an API that dynamically generates responses based on the current state or conditions at the time of the request. The structure and content of the API responses can change dynamically based on various factors such as user input, real-time data updates, or contextual information.

Dynamic APIs are commonly used when the data or functionality being exposed requires real-time updates or customization based on user preferences or contextual information.

Example:

Consider an API that provides weather information. A dynamic API can retrieve real-time weather data for a specific location, taking into account factors such as time of the day, current weather conditions, and user preferences. The API response may vary based on these dynamic factors.

Key Differences:

Static APIs have a fixed structure and functionality, while Dynamic APIs generate responses based on real-time or contextual information.

Static APIs are suitable for stable and non-changing data or functionality, while Dynamic APIs are used for real-time updates and customization.

Static APIs provide a consistent response structure, whereas Dynamic APIs can have varying response structures based on the request and conditions.

In some cases, APIs can have elements of both static and dynamic behavior, where certain parts of the API remain static while others dynamically generate responses.

Choosing between a Static API and a Dynamic API depends on the specific requirements of the application and the nature of the data or functionality being exposed.

Dependencies of Node runtime - Node, V8, Libuv & C++

To understand the dependencies of the Node.js runtime, let's break down the key components involved: Node.js, V8, Libuv, and C++.

- Node.js: Node.js is a JavaScript runtime built on the V8 JavaScript engine. It provides an environment that allows you to execute JavaScript code outside of a web browser. Node.js provides various features and APIs, such as file system operations, networking capabilities, and HTTP servers, making it suitable for building server-side applications.
- V8: V8 is an open-source JavaScript engine developed by Google. It executes JavaScript code and provides high-performance execution by compiling JavaScript into machine code. V8 is responsible for interpreting and executing JavaScript code in the Node.js runtime. It includes features like just-in-time (JIT) compilation, garbage collection, and optimizations to improve the performance of JavaScript execution.

- Libuv: Libuv is a multi-platform library that provides asynchronous I/O operations and event-driven programming. It abstracts the differences in the underlying operating systems and provides a unified API for handling events, timers, file system operations, networking, and more. Libuv is utilized by Node.js to enable non-blocking I/O operations and support the event-driven architecture of Node.js, which allows for efficient handling of concurrent requests and scalability.
- C++: While Node.js and its core functionality are primarily implemented in JavaScript, certain parts of Node.js and its dependencies are implemented in C++. C++ is a programming language used to write low-level and performance-critical code. In the context of Node.js, C++ is utilized for implementing core modules, bindings to system libraries, and integrating with the underlying operating system and hardware. These C++ components provide the necessary interfaces and functionalities to enable JavaScript execution, I/O operations, networking, and more.

In summary, Node.js relies on the V8 JavaScript engine for executing JavaScript code, Libuv for providing event-driven and asynchronous I/O operations, and C++ for implementing core modules and integrating with the underlying system. These components work together to deliver the functionality and performance that Node.js provides as a runtime environment for server-side JavaScript applications.

Processes, threads, and thread pools

Processes, threads, and thread pools are concepts related to concurrent programming and multitasking. Let's understand each of these concepts:

1. Processes: A process is an instance of a program that is being executed by the operating system. It has its own memory space, resources, and execution context. Each process runs independently of other processes and is isolated from them. Processes can communicate with each other through inter-process communication mechanisms provided by the operating system.
2. Threads: A thread is a unit of execution within a process. It represents a single sequence of instructions that can be scheduled and executed by the operating system. Threads within the same process share the same memory space and resources. Multiple threads can exist within a single process and run concurrently, allowing for parallel execution and improved performance. Threads are lighter-weight than processes and have less overhead in terms of memory and resource consumption.

3. Thread Pool: A thread pool is a managed group of pre-initialized threads that are available for executing tasks. Instead of creating and destroying threads for each individual task, a thread pool maintains a pool of reusable threads. When a task needs to be executed, it is assigned to an available thread from the pool. Once the task is completed, the thread is returned to the pool for future use. Thread pools provide better performance and resource management by avoiding the overhead of thread creation and destruction for each task.

In summary, processes represent independent instances of programs, threads are units of execution within processes that enable concurrent execution, and thread pools provide a managed group of reusable threads for efficient task execution. These concepts are fundamental in concurrent programming and multitasking, allowing for efficient utilization of system resources and improved application performance.

Environment variables

Environment variables are variables that are part of the environment in which a process runs. They are key-value pairs that can be accessed by applications or scripts to configure and customize behavior based on the specific environment.

Here are some key points about environment variables:

1. Definition and Usage: Environment variables are typically set outside of the application code and are used to provide configuration parameters or system-specific information to the application. They are accessed by the application during runtime to adjust its behavior or access external resources.
2. Operating System Support: Environment variables are supported by various operating systems, including Windows, macOS, Linux, and others. Each operating system has its own way of setting and accessing environment variables.

3. Key-Value Pair: Environment variables consist of a name (key) and a corresponding value. The key is usually uppercase and can contain letters, numbers, and underscores. The value can be a string representing any data, such as paths, URLs, database connection strings, or other configuration values.

4. Setting Environment Variables: Environment variables can be set in different ways, depending on the operating system and the context. Some common methods include configuring them in the shell or command line, defining them in a startup script or configuration file, or using a tool or IDE-specific settings.

5. Accessing Environment Variables: Applications can access environment variables through language-specific APIs or libraries. For example, in Node.js, you can use process.env to access environment variables, while other programming languages provide similar mechanisms.

6. Common Use Cases: Environment variables are commonly used for sensitive information like API keys and database credentials, configuring application behavior based on the deployment environment (e.g., development, staging, production), or specifying paths to external dependencies.

Introducing express

Express is a popular web application framework for Node.js that simplifies the process of building robust and scalable web applications and APIs. It provides a minimalistic and flexible approach to web development, allowing developers to create server-side applications with ease.

Here are some key features and concepts associated with Express:

1. Routing: Express provides a simple and intuitive routing mechanism that allows developers to define routes for handling HTTP requests. With Express, you can define routes for different HTTP methods (GET, POST, PUT, DELETE, etc.) and specify the corresponding callback functions to handle the requests.
2. Middleware: Middleware functions in Express are functions that have access to the request and response objects in the application's request-response cycle. Middleware functions can perform tasks such as request parsing, authentication, logging, error handling, and more. They can be chained together to create a pipeline of processing logic for incoming requests.

3. Templating: Express supports various templating engines such as EJS, Pug (formerly Jade), and Handlebars, allowing developers to render dynamic HTML pages easily. Templating engines help generate HTML content by merging data with pre-defined templates, enabling the creation of dynamic and interactive web pages.

4. Error Handling: Express provides mechanisms for handling errors that occur during the request-response cycle. Developers can define error-handling middleware functions that receive any errors thrown during request processing. These functions can then handle the errors, log them, and send appropriate error responses to clients.

5. Middleware and Route Modules: Express allows you to modularize your application by creating separate modules for middleware and routes. This helps in organizing and reusing code, making your application more maintainable and scalable.

```
const express = require('express');
const app = express();
// Define a route
app.get('/', (req, res) => {
  res.send('Hello, Express!');
});
// Start the server
app.listen(3000, () => {
  console.log('Server is running on port 3000');
});
```

Introducing Nodemon

Nodemon is a developer tool that enhances the development workflow when working with Node.js applications. It automatically monitors your Node.js application for changes and restarts it whenever a file is modified, saving you the hassle of manually stopping and restarting the server during development.

Here are some key features and benefits of using Nodemon:

1. Automatic Restart: Nodemon monitors the files in your Node.js project for any changes. When it detects a file modification, it automatically restarts the Node.js application. This feature speeds up the development process, as you don't have to manually stop and restart the server each time you make a code change.
2. Enhanced Development Experience: By automatically restarting the application on file changes, Nodemon provides a seamless development experience. You can see the changes you make in real-time without needing to manually trigger the application restart.
3. Support for Different File Types: Nodemon can monitor and restart your Node.js application based on changes to JavaScript files (.js), configuration files, JSON files, and other file types that are relevant to your project.
4. Customization and Configuration: Nodemon allows you to customize its behavior through configuration options. You can specify which files to monitor, ignore specific directories or files, delay the restart after file changes, and more. This flexibility makes it adaptable to different project setups and requirements.
5. Integration with Node.js Ecosystem: Nodemon works seamlessly with popular frameworks and tools in the Node.js ecosystem, such as Express, Koa, Hapi, and others. It complements these frameworks by providing an automatic restart mechanism during development.

Restful services

RESTful services, also known as RESTful APIs or REST APIs, are a style of web services that follow the principles of Representational State Transfer (REST). REST is an architectural style for designing networked applications that leverage the existing technologies of the web.

Here are some key concepts and characteristics of RESTful services:

1. Resources: In REST, a resource is the fundamental concept and the key building block of an API. A resource represents an object or entity that can be accessed, manipulated, or interacted with. Examples of resources could be users, articles, products, or any other data entity within an application.

1. Uniform Interface: RESTful services adhere to a uniform interface, which consists of standard HTTP methods (GET, POST, PUT, DELETE) for performing operations on resources. Each HTTP method has a specific purpose: GET for retrieving resource data, POST for creating new resources, PUT for updating existing resources, and DELETE for deleting resources.
2. Stateless Communication: REST is stateless, meaning that each request from a client to a server must contain all the necessary information to process that request. The server does not store any client state between requests. This simplifies scalability and allows requests to be processed independently.
3. Resource Identifiers: Resources in RESTful services are identified by unique URIs (Uniform Resource Identifiers). Each resource has a unique URI that clients can use to access and interact with it. For example, /users could represent the collection of user resources, and /users/123 could represent a specific user with the ID 123.
4. Representation of Resources: Resources in RESTful services are represented in a specific format, such as JSON (JavaScript Object Notation) or XML (eXtensible Markup Language). The client and server communicate by exchanging representations of resources, allowing for flexible data formats and decoupling of the client and server.
5. Hypermedia as the Engine of Application State (HATEOAS): HATEOAS is a principle of REST that encourages including links or hypermedia in the API responses. These links provide navigation and discoverability within the API, allowing clients to dynamically explore and interact with available resources.

By following the principles of REST, RESTful services provide a scalable, interoperable, and stateless approach to building web APIs. They enable client-server communication over standard HTTP methods, allowing clients to perform CRUD (Create, Read, Update, Delete) operations on resources.

Here's an example of a RESTful API endpoint for managing users:

```
GET /users        // Retrieve a list of users

GET /users/{id}   // Retrieve a specific user

POST /users       // Create a new user

PUT /users/{id}   // Update an existing user

DELETE /users/{id}  // Delete a user
```

In this example, the API provides endpoints for retrieving a list of users, retrieving a specific user by ID, creating a new user, updating an existing user, and deleting a user. These endpoints follow the RESTful principles and use the appropriate HTTP methods to perform the corresponding operations on the user resources.

RESTful services have become a widely adopted approach for building APIs due to their simplicity, scalability, and compatibility with the web's existing infrastructure.

Building First Express Server

- Start by creating a new directory for your project and navigate into it using the terminal:

  mkdir my-express-server

  cd my-express-server

- Initialize a new npm package in the project directory:

  npm init -y

- Install Express as a dependency:

  npm install express

- Create a new file called server.js and open it in a code editor. Add the following code:

```
const express = require('express');
// Create an instance of Express
const app = express();
// Define a route
app.get('/', (req, res) => {
  res.send('Hello, Express!');
});
// Start the server
app.listen(3000, () => {
  console.log('Server is running on port 3000');
});
```

- Save the file and go back to the terminal. Run the server using the following command:

  node server.js, npm start

- You should see the message "Server is running on port 3000" in the console, indicating that the server is up and running.
- Open your web browser and visit http://localhost:3000. You should see the message "Hello, Express!" displayed in the browser.
- Congratulations! You have successfully built your first Express server. It listens for incoming HTTP GET requests on the root URL ("/") and sends the response "Hello, Express!" back to the client.
- You can create additional routes by using different HTTP methods (e.g., app.post(), app.put(), app.delete()) and specifying the route path and corresponding callback function.

HTTP GET REQUEST

1. HTTP GET Method:
   - The GET method is one of the HTTP methods used for retrieving data from a server.
   - When a client sends a GET request, it requests a specific resource or information from the server.
   - The GET request is considered safe and idempotent, meaning it should not have any side effects on the server and can be repeated without changing the server state.
2. Retrieving Data:
   - GET requests are commonly used to retrieve data from a server by specifying a URL or endpoint.
   - The requested data can be in various formats such as HTML, JSON, XML, or plain text, depending on the server's response and the client's specified content type.

3. Express.js and GET Requests:

- In Express.js, you can handle GET requests by defining routes using the app.get() method.
- The app.get() method takes the URL pattern and a callback function that executes when a GET request is made to that URL.
- The callback function takes two parameters: req (request) and res (response).
- Inside the callback function, you can access the request data, query parameters, and any other necessary information.

4. Route Parameters:

- GET requests can include route parameters to provide additional information or identify a specific resource within the URL.
- Route parameters are defined as placeholders in the URL pattern and can be accessed in the callback function using req.params.

5. Sending the Response:

- To send a response back to the client for a GET request, you can use the res.send(), res.json(), or other response methods provided by Express.js.
- The response can include data, HTML, or any desired content to be displayed by the client.

6. Error Handling:

- It's essential to handle scenarios where a requested URL doesn't match any defined routes.
- Express.js provides a catch-all route using app.get('*', ...), where you can handle such cases and send an appropriate response, such as returning a "Page not found" message.

```
const express = require('express');
// Create an instance of Express
const app = express();
// Define a GET route
app.get('/hello', (req, res) => {
  res.send('Hello, world!');
});
// Start the server
app.listen(8000, () => {
  console.log('Server is running on port 8000');
});
```

HTTP POST Request

```
const express = require('express');

const app = express();

app.use(express.json()); // Parse JSON request bodies

// Define a POST route

app.post('/api/users', (req, res) => {

  const { name, email } = req.body;  // Access the data from the request body

  // Perform validation or any other necessary logic

  if (!name || !email) {

    return res.status(400).json({ message: 'Name and email are required' });

  }  // Create a new user or save the data to a database

  res.status(201).json({ message: 'User created successfully' });   // Send a response indicating success
```

HTTP PUT Request

```
const express = require('express');
const app = express();
// Parse JSON request bodies
app.use(express.json());
// Simulated data - to be replaced with your actual data storage
let users = [
  { id: 1, name: 'John Doe', email: 'john@example.com' },
  { id: 2, name: 'Jane Smith', email: 'jane@example.com' }
];
// Define a PUT route
app.put('/api/users/:id', (req, res) => {
  // Extract the user ID from the request parameters
  const userId = parseInt(req.params.id);
  // Find the user by ID
  const user = users.find((user) => user.id === userId);
  // If the user doesn't exist, return a 404 Not Found response
  if (!user) {
    return res.status(404).json({ message: 'User not found' });
  }
  // Update the user's name and email based on the request body
  user.name = req.body.name;
  user.email = req.body.email;
  // Send a response indicating success
  res.json({ message: 'User updated successfully', user });
});
// Start the server
app.listen(8000, () => {
  console.log('Server is running on port 8000');
});
```

HTTP DELETE Request

```
const express = require('express');
const app = express();
// Simulated data - to be replaced with your actual data storage
let users = [
  { id: 1, name: 'John Doe', email: 'john@example.com' },
  { id: 2, name: 'Jane Smith', email: 'jane@example.com' }
];
// Define a DELETE route
app.delete('/api/users/:id', (req, res) => {
  // Extract the user ID from the request parameters
  const userId = parseInt(req.params.id);
  // Find the index of the user by ID
  const userIndex = users.findIndex((user) => user.id === userId);
  // If the user doesn't exist, return a 404 Not Found response
  if (userIndex === -1) {
    return res.status(404).json({ message: 'User not found' });
  }
  // Remove the user from the array
  users.splice(userIndex, 1);
  // Send a response indicating success
  res.json({ message: 'User deleted successfully' });
});
// Start the server
app.listen(8000, () => {
  console.log('Server is running on port 8000');
});
```

Route Parameters

Route parameters are placeholders in the URL of a web application or API that allow dynamic values to be passed and processed by the server. They are often used in routing frameworks to define specific endpoints or routes.

When defining a route, you can specify a parameter by including a placeholder in the URL pattern. Commonly, these placeholders are indicated by using curly braces {} or colon : before the parameter name. For example, consider the following route pattern for a blog application:

/posts/{postId}

In this example, {postId} is a route parameter that can hold a specific value. When a client makes a request to /posts/123, the server knows that 123 corresponds to the postId parameter. The server can then use this value to fetch the blog post with ID 123 from a database or perform any other necessary operations.

Route parameters provide flexibility by allowing URLs to be more descriptive and expressive. They enable the creation of dynamic routes that can handle varying input values. In addition, route parameters are often used for RESTful APIs to specify resource identifiers.

Some routing frameworks provide additional features for route parameters, such as specifying data types, constraints, or default values. These features help validate and control the values passed through the parameters.

```
const express = require('express');

const app = express();

app.get('/posts/:postId', (req, res) => {
  const postId = req.params.postId;
  // Process the postId value
  res.send(`Fetching post with ID ${postId}`);
});

app.listen(3000, () => {
  console.log('Server is running on port 3000');
});
```

In this example, when a GET request is made to /posts/123, the server extracts the value 123 from the postId route parameter using req.params.postId. You can then use this value to perform the required logic or retrieve the corresponding blog post.

Advanced Express & MVC Architecture

Advanced Express.js with MVC (Model-View-Controller) architecture allows for a more organized and scalable web application. Here's an overview of how Express.js can be used with MVC architecture:

Model-View-Controller (MVC) Architecture:

- MVC is a software design pattern that separates an application into three components: Model, View, and Controller.
- The Model represents the data and business logic of the application.
- The View handles the presentation layer and displays the data to the user.
- The Controller manages the flow of data between the Model and the View, handling user input and triggering appropriate actions.

Setting Up the Project Structure:

- Create separate directories for each MVC component: models, views, and controllers.
- Each directory will contain files specific to their respective responsibilities.
- Additionally, create a directory for routes to define the application's endpoints.

Model:

- Models represent the data structure and business logic of the application.
- They interact with the database or any external data source.
- Models can be defined as separate JavaScript files within the models directory, representing entities or business logic functions.

View:

- Views handle the presentation and rendering of data to the user.
- They can be HTML templates or dynamic templates using template engines like EJS, Pug, or Handlebars.
- Views are stored in the views directory and are rendered by the controller.

Controller:

- Controllers handle the logic and flow of the application.
- They receive user input, interact with models, and render appropriate views.
- Controllers define the route handlers and connect the models and views.
- Controllers are defined as separate JavaScript files within the controllers directory.

Routing:

- Routes map incoming requests to the appropriate controller and its associated action.
- Routes can be defined in a separate file or directly in the main application file.
- Express.js provides a routing mechanism using the express.Router() object to define routes and their corresponding controller actions.

Integrating Express.js with MVC Architecture:

- In the main application file, set up Express.js and configure middleware.
- Configure the application to use the routes defined by the router object.
- Each route defined in the router object should specify the corresponding controller action.
- The controller action interacts with the model, fetches data, performs business logic, and renders the appropriate view.

1. Model (models/blog.js):

```
const mongoose = require('mongoose');


const blogSchema = new mongoose.Schema({
  title: { type: String, required: true },
  content: { type: String, required: true },
  author: { type: String, required: true },
  createdAt: { type: Date, default: Date.now },
});
module.exports = mongoose.model('Blog', blogSchema);
```

2. Controller (controllers/blogController.js):

```
const Blog = require('../models/blog');


// Display all blogs
exports.blogList = function (req, res) {
  Blog.find({}, (err, blogs) => {
    if (err) {
      console.error(err);
      res.status(500).send('Internal Server Error');
    } else {
      res.json(blogs);
    }
  });
};
```

```
// Create a new blog
exports.createBlog = function (req, res) {
  const { title, content, author } = req.body;
  const newBlog = new Blog({ title, content, author });
  newBlog.save((err) => {
    if (err) {
      console.error(err);
      res.status(500).send('Internal Server Error');
    } else {
      res.json({ message: 'Blog created successfully' });
    }
  });
};
```

Express Routes (routes/index.js):

```
const express = require('express');

const router = express.Router();

const blogController = require('../controllers/blogController');


// GET request to display all blogs

router.get('/', blogController.blogList);


// POST request to create a new blog

router.post('/create', blogController.createBlog);


module.exports = router;
```

Express App (app.js):

```
const express = require('express');

const mongoose = require('mongoose');

const indexRouter = require('./routes/index');

const app = express();

// Connect to MongoDB

mongoose.connect('mongodb://localhost/blog', {

  useNewUrlParser: true,

  useUnifiedTopology: true,

});

// Middleware

app.use(express.urlencoded({ extended: true }));

// Routes

app.use('/', indexRouter);

// Start the server

app.listen(3000, () => {

  console.log('Server started on port 3000');

});
```

Middlewares

- In simple words, middleware is a piece of code or a function that sits between the incoming request and the outgoing response in an application. It acts as a bridge or a layer that adds extra functionality to the request-response cycle.
- Imagine you are in a restaurant, and you place an order with the waiter. The waiter takes your order to the kitchen, where the chef prepares the food. However, before the food is served to you, the waiter may add some additional elements to the plate, such as garnishing or seasoning, to enhance the presentation or taste. In this scenario, the waiter acts as middleware, adding value to the food before it reaches you.
- Similarly, in web development, middleware functions intercept the incoming requests to the server before they reach the final route handler. They can perform tasks like logging, data parsing, authentication, error handling, and more. After executing their specific functionality, middleware can either pass the request to the next middleware in line or send a response directly to the client.

- Middleware plays a crucial role in extending the capabilities of an application, enhancing security, and maintaining code organization by separating concerns. It allows developers to modularize and reuse code, making it easier to add, modify, or remove functionalities as needed.
- Overall, middleware acts as an intermediary layer in an application, providing additional processing or functionality to requests and responses, much like a waiter adding value to your meal before it reaches your table.

Different types of middleware in Express.js

1. Application-Level Middleware:

Application-level middleware is registered using app.use() and is executed for every incoming request to the application.

Example:

```
app.use((req, res, next) => {
  console.log('This is an application-level middleware');
  next();
});
```

2. Router-Level Middleware:

Router-level middleware is applied to specific routes or groups of routes using router.use() within an Express router.

Example:

```
const router = express.Router();


router.use((req, res, next) => {
  console.log('This is a router-level middleware');
  next();
});
```

3. Error Handling Middleware:

Error handling middleware is used to handle errors that occur during the request-response cycle. It takes four parameters: (err, req, res, next).

Example:

```
app.use((err, req, res, next) => {
  console.error(err);
  res.status(500).send('Internal Server Error');
});
```

4. Built-in Middleware:

Express provides built-in middleware that can be used without installing any additional packages. Examples include express.json() for parsing JSON in the request body and express.static() for serving static files.

Example:

```
app.use(express.json());

app.use(express.static('public'));
```

5. Third-Party Middleware:

Third-party middleware is developed by the Express community and can be installed using npm or yarn. They provide additional functionality for tasks like authentication, logging, compression, etc.

Example (using the morgan logging middleware):

```
const morgan = require('morgan');

app.use(morgan('combined'));
```

6. Custom Middleware:

Custom middleware functions are created by the developer to add specific functionality or perform custom tasks in the request-response cycle.

Example:

javascript

Copy code

```
const myMiddleware = (req, res, next) => {
  console.log('This is a custom middleware');
  next();
};
app.use(myMiddleware);
```

Building RESTful API's Using Express - CRUD for Ecommerce API

express.static

- express.static is a middleware function provided by the Express.js framework.
- It is used to serve static files (e.g., HTML, CSS, JavaScript, images) in an Express application.
- Static files are files that don't change dynamically based on user requests but remain the same for all users.
- The middleware function takes a directory path as an argument, which represents the root directory for serving static files.
- When a request is made for a static file, Express looks for the file in the specified directory and serves it to the client.

- It automatically sets the appropriate headers and caching mechanisms for efficient file delivery.
- By using express.static, you can easily serve an entire directory of static files without writing separate routes for each file.
- The middleware should be added to the Express application using app.use() before defining other routes, to ensure it is executed for every incoming request.
- The middleware works by matching the requested URL path with the files in the specified directory, and if a matching file is found, it is sent as a response to the client.
- The client can access the static files by using the relative URL path corresponding to the file's location within the static directory.
- express.static provides a convenient way to organize and serve client-side assets, making it ideal for building web applications, websites, and APIs with Express.js.

express.urlencoded

The express.urlencoded middleware is specifically designed to handle URL-encoded form data in an Express.js application. It parses the form data sent in the request body and makes it accessible through the req.body object. Here are some common use cases where express.urlencoded is beneficial:

- Handling HTML form submissions: When you have an HTML form in your application and want to extract the submitted data, express.urlencoded is useful. It automatically parses the form data and provides easy access to the values submitted by the user.

- Processing form data for authentication: User login and registration forms often require handling sensitive data such as usernames and passwords. express.urlencoded simplifies the process of extracting and validating these form inputs, allowing you to securely process and authenticate user data.
- Accepting input from API clients: If you have an API built with Express.js that accepts data from API clients, express.urlencoded helps parse and handle the data sent as URL-encoded form parameters. It allows you to access and process the incoming data easily.
- Working with third-party APIs: Many third-party APIs expect data to be sent in URL-encoded form. By using express.urlencoded, you can format the data according to the API's requirements and send requests with the necessary form data.
- Handling webhook data: Webhooks often send data in URL-encoded format. By utilizing express.urlencoded, you can easily extract and process the data received through webhooks in your Express application.

Same Origin Policy

The Same Origin Policy (SOP) is a security mechanism enforced by web browsers to restrict web page scripts from making requests to a different origin (domain, protocol, and port) than the one from which they originated. The SOP is an important security measure to prevent cross-site scripting (XSS) attacks and unauthorized data access.

Under the Same Origin Policy:

- Scripts from one origin cannot access the content of web pages from a different origin.
- XMLHttpRequest and Fetch API requests are restricted to the same origin. That means, by default, a web page can only make requests to its own origin.
- Cookies and other credentials are not sent in cross-origin requests unless explicitly allowed through mechanisms like CORS (Cross-Origin Resource Sharing).
- DOM access restrictions prevent scripts from accessing or modifying elements of a different origin's web page.

The Same Origin Policy is crucial for maintaining the security and integrity of web applications. Without it, malicious scripts from one site could access sensitive data or perform actions on behalf of another site, leading to potential security vulnerabilities.

To enable controlled access to resources across different origins, web standards like CORS have been introduced. CORS allows servers to specify which origins are allowed to access their resources and under what conditions, thereby relaxing the strict Same Origin Policy for those specified origins.

It's important to note that the Same Origin Policy applies to scripts running in the context of a web browser and does not impose restrictions on server-to-server communication or communication between different web technologies, such as APIs or backend systems.

Cross Origin Resource Sharing (CORS)

Cross-Origin Resource Sharing (CORS) is a mechanism that relaxes the Same Origin Policy (SOP) in web browsers. It allows servers to specify which origins (domains, protocols, and ports) are allowed to access their resources and under what conditions.

By default, web browsers enforce the Same Origin Policy, which restricts cross-origin requests, such as AJAX requests or Fetch API requests, to ensure security. However, there are legitimate cases where a web application hosted on one origin needs to make requests to resources hosted on a different origin. CORS provides a standardized way for servers to specify which cross-origin requests are allowed and which are not.

Here's how CORS works:

- Origin Header: When a browser makes a cross-origin request, it includes an Origin header that indicates the origin from which the request originated.
- Preflight Request: For certain types of cross-origin requests, such as those with non-simple HTTP methods (e.g., PUT, DELETE) or custom headers, the browser sends a preflight request (HTTP OPTIONS) to the server. The preflight request includes an additional set of headers, including the requested method, headers, and any custom headers.
- Access-Control-Allow-Origin: The server receiving the request can include the Access-Control-Allow-Origin response header to indicate which origins are allowed to access its resources. It can specify a single origin (e.g., "https://example.com") or use the wildcard "*" to allow access from any origin. The server can also include additional headers like Access-Control-Allow-Methods, Access-Control-Allow-Headers, and Access-Control-Allow-Credentials to further control the behavior of cross-origin requests.

- Simple Requests: For simple cross-origin requests (e.g., GET, POST with no custom headers), the browser directly sends the request to the server without a preflight request. The server can respond with the Access-Control-Allow-Origin header to allow or deny access.
- Error Handling: If a cross-origin request is made without proper CORS headers, or the server rejects the request, the browser blocks the response from being accessed by the requesting script due to the Same Origin Policy.

Introduction of MongoDB

- MongoDB is an open-source document-oriented NoSQL database.
- It stores data in flexible JSON-like documents called BSON.
- MongoDB is designed for scalability and high performance.
- It supports horizontal scaling through sharding and replication.
- The database offers flexible querying using a JSON-based query language.
- MongoDB automatically indexes data for faster query performance.
- It has a dynamic schema, allowing for easy data modification without predefined structures.

- Replication ensures data durability and high availability.
- MongoDB has official drivers for popular programming languages.
- It has a rich ecosystem of tools, libraries, and cloud-hosted solutions (MongoDB Atlas).
- MongoDB is used in various applications, including web, mobile, and IoT platforms.

In summary, MongoDB is a flexible, scalable, and high-performance NoSQL database that allows for easy data storage, querying, and scalability in modern applications.

Comparison query operators

MongoDB provides a wide range of comparison query operators that can be used to compare values in queries. Here are some commonly used comparison query operators in MongoDB:

$eq: Matches values that are equal to a specified value.

Example: { field: { $eq: value } }

$ne: Matches values that are not equal to a specified value.

Example: { field: { $ne: value } }

$gt: Matches values that are greater than a specified value.

Example: { field: { $gt: value } }

$gte: Matches values that are greater than or equal to a specified value.

Example: { field: { $gte: value } }

$lt: Matches values that are less than a specified value.

Example: { field: { $lt: value } }

$lte: Matches values that are less than or equal to a specified value.

Example: { field: { $lte: value } }

$in: Matches any of the values specified in an array.

Example: { field: { $in: [value1, value2, ...] } }

$nin: Matches none of the values specified in an array.

Example: { field: { $nin: [value1, value2, ...] } }

$regex: Matches values based on a regular expression pattern.

Example: { field: { $regex: /pattern/ } }

$exists: Matches documents that have the specified field, regardless of its value.

Example: { field: { $exists: true } }

These operators can be used in conjunction with the field name to create powerful and flexible queries in MongoDB, allowing you to filter and retrieve documents based on specific comparison criteria.

Logical Query Operators of mongodb

MongoDB provides several logical query operators that can be used to combine multiple conditions in a query. Here are some commonly used logical query operators in MongoDB:

$and: Performs a logical AND operation on an array of two or more query conditions. It selects documents that satisfy all the conditions.

Example: { $and: [{ condition1 }, { condition2 }] }

$or: Performs a logical OR operation on an array of two or more query conditions. It selects documents that satisfy at least one of the conditions.

Example: { $or: [{ condition1 }, { condition2 }] }

$not: Performs a logical NOT operation on a query condition. It selects documents that do not match the specified condition.

Example: { field: { $not: { condition } } }

$nor: Performs a logical NOR operation on an array of two or more query conditions. It selects documents that do not satisfy any of the conditions.

Example: { $nor: [{ condition1 }, { condition2 }] }

$exists: Matches documents that have the specified field, regardless of its value.

Example: { field: { $exists: true } }

$type: Matches documents based on the BSON type of a field.

Example: { field: { $type: "string" } }

These logical query operators can be combined with other comparison operators to construct complex queries in MongoDB. They provide flexibility in expressing various conditions and filtering criteria when retrieving documents from a collection.

Aggregation Pipeline : Matching & Grouping

Aggregation Pipeline:

The Aggregation Pipeline is a powerful feature in MongoDB that allows you to process and transform data in a collection using a sequence of stages. Each stage takes the input documents, performs some operation, and passes the output to the next stage. The pipeline stages can perform various operations like filtering, transforming, grouping, sorting, and more.

Matching Stage:

The matching stage, also known as the $match stage, is used to filter documents in the aggregation pipeline. It allows you to select only the documents that match certain conditions. The $match stage uses MongoDB's query language to specify the filtering criteria. You can use various query operators such as $eq, $ne, $gt, $lt, $in, $and, $or, etc., to build complex conditions. The matching stage is usually used as the first stage in the pipeline to reduce the data set before further processing.

Grouping Stage:

The grouping stage, represented by the $group stage, is used to group documents based on specified criteria. It allows you to perform various operations like counting, summing, averaging, finding maximum or minimum values, and more on grouped data. The $group stage requires an _id field to specify the grouping key, which can be a field from the documents or a computed value using expressions. You can use various aggregation operators like $sum, $avg, $max, $min, $push, $addToSet, etc., to perform operations on grouped data.

Here's an example that demonstrates the usage of the matching and grouping stages in the Aggregation Pipeline:

Consider a collection named sales with documents representing sales transactions, each containing fields like product, category, quantity, and price.

```
db.sales.aggregate([
  {
    $match: { category: "Electronics", price: { $gt: 1000 } }
  },
  {
    $group: {
      _id: "$product",
      totalQuantity: { $sum: "$quantity" },
      totalPrice: { $sum: { $multiply: ["$quantity", "$price"] } }
    }
  }
])
```

In this example, the $match stage filters the documents to select only those in the "Electronics" category with a price greater than 1000. Then, the $group stage groups the filtered documents by the product field and calculates the total quantity and total price for each product.

This pipeline would output the total quantity and total price for each product in the "Electronics" category with a price greater than 1000.

Please note that this is just a basic example, and the Aggregation Pipeline is capable of handling much more complex operations and stages.

Feel free to explore more about the Aggregation Pipeline and its stages to deepen your understanding and experiment with different use cases.

pipeline: This is an array that represents the stages of the aggregation pipeline. Each stage in the array defines an operation to be performed on the data.

$match: This is the first stage in the pipeline. It filters the documents based on a specified condition. In this case, it matches documents where the category field equals 'phone'.

$group: This is the second stage in the pipeline. It groups the documents based on a specified field ($category in this case) and performs a calculation on the grouped data. In this case, it calculates the count of documents in each group using the $sum operator.

result: This variable holds the result of the aggregation operation. The aggregate method is called on the Products model, passing the pipeline array as the argument. The await keyword is used to wait for the aggregation to complete and assign the result to the result variable.

So, in summary, the code matches documents with the category 'phone' and then groups the matched documents by the category field. The result is the count of documents in each group.

Unwinding:

The $unwind stage in the aggregation pipeline is used to deconstruct an array field into multiple documents, each containing one element of the array. This stage is particularly useful when you want to perform operations on individual array elements.

Syntax:

{ $unwind: <arrayField> }

Example:

Consider a collection of "books" where each document has an array field called "authors" containing multiple authors. If you want to perform operations on each author individually, you can use the $unwind stage as follows:

```
db.books.aggregate([
  { $unwind: "$authors" }
])
```

This will produce a separate document for each author in the "authors" array.

Projecting:

The $project stage in the aggregation pipeline is used to reshape documents and include or exclude specific fields. It allows you to create new fields, modify existing fields, and remove unwanted fields.

{ $project: { <field1>: <expression1>, <field2>: <expression2>, ... } }

Example:

Suppose you have a collection of "employees" where each document contains fields like "name," "age," and "salary." If you want to project only the "name" and "salary" fields in the output, you can use the $project stage as follows:

```
db.employees.aggregate([

  { $project: { name: 1, salary: 1 } }

])
```

This will exclude all other fields except "name" and "salary" in the output documents.

Combining Unwinding and Projecting:

You can use both the $unwind and $project stages together in an aggregation pipeline to perform complex transformations on your data.

Example:

Let's say you have a collection of "orders" where each document has an array field called "products" containing details of products ordered. If you want to unwind the "products" array and then project only the "productName" and "quantity" fields in the output, you can use the following pipeline:

```
db.orders.aggregate([

  { $unwind: "$products" },

  { $project: { productName: "$products.name", quantity: "$products.quantity" } }

])
```

This will create separate documents for each product in the "products" array and only include the "productName" and "quantity" fields in the output.

The aggregation pipeline, with stages like $unwind and $project, provides a flexible and powerful way to transform and aggregate data in databases. It allows you to perform complex operations and extract meaningful information from your collections.

Topic: Validation and its Need

Content:

Validation is the process of ensuring that data or information meets certain predefined criteria or rules. It is a crucial step in data processing to ensure accuracy, reliability, and consistency of the data. The primary purpose of validation is to prevent erroneous or invalid data from being accepted or processed, which can lead to errors, incorrect results, or system failures.

The need for validation arises from several factors:

Data Integrity: Validation helps maintain data integrity by ensuring that the data is complete, accurate, and consistent. It helps identify and eliminate errors or discrepancies in the data.

Error Prevention: Validation acts as a preventive measure by catching errors or inconsistencies in the data before they propagate throughout the system. It helps avoid potential issues or problems that may arise due to invalid data.

User Experience: Validating user input is crucial for providing a good user experience. By validating data entered by users, you can provide real-time feedback and guidance, preventing users from submitting incorrect or incomplete information.

Security: Validation plays a significant role in ensuring the security of a system. By validating input data, you can protect against various security vulnerabilities such as SQL injection, cross-site scripting (XSS), or other types of attacks that exploit invalid or maliciously crafted data.

Example:

Let's consider a simple example of a registration form on a website. The form requires users to enter their email address, password, and age. Here's how validation can be applied to each field:

Email Address: The validation process checks if the entered value follows the correct email format (e.g., containing "@" and "."). It can also ensure that the email address is unique and not already registered in the system.

Password: The validation process can enforce certain criteria for the password, such as a minimum length, the presence of both uppercase and lowercase letters, and special characters. This ensures that users create strong and secure passwords.

Age: The validation process can check if the entered age is within a valid range (e.g., 18-99). It prevents users from entering invalid or nonsensical values.

By implementing validation in this example, the system can prevent users from submitting incorrect or incomplete data, ensuring that only valid and reliable information is processed.

Built-in Validators

Content:

Built-in validators are pre-defined validation functions or methods provided by programming frameworks or libraries. These validators offer a range of commonly used validation rules that can be easily applied to validate data. They save development time and effort by providing ready-to-use validation functions for various types of data.

Built-in validators often cover standard validation requirements such as checking for required fields, data types, string lengths, numeric ranges, and email formats. They are typically part of a larger validation framework or module.

Using built-in validators has several advantages:

Convenience: Built-in validators simplify the validation process by providing pre-configured validation rules that can be easily applied to data fields. Developers don't need to write custom validation logic for common scenarios.

Consistency: By using built-in validators, you can ensure consistent validation across different parts of the system. The same validation rules can be applied to multiple data fields or objects, maintaining uniformity in data validation.

Extensibility: Built-in validators often provide options for customization and extension. Developers can configure the validators to fit specific validation requirements or even create custom validators by extending the built-in functionality.

Example:

Let's consider a web application that allows users to submit a contact form. The form has fields for name, email address, and message. Here's how built-in validators can be used to validate the form data:

Name: A built-in validator can check if the name field is not empty or contains only valid characters (e.g., letters and spaces).

Email Address: A built-in validator can verify that the email address is in the correct format (e.g., "example@example.com") using regular expressions or dedicated email validation functions.

Message: A built-in validator can ensure that the message field is not empty and does not exceed a certain character limit.

By utilizing built-in validators, developers can easily implement these validation rules without having to write custom validation logic for each field, making the validation process more efficient and reliable.

Custom Validators

Custom validators are validation functions or methods created by developers to implement specific validation rules that are not covered by built-in validators. These validators are tailored to meet unique or complex validation requirements specific to a project or domain.

Custom validators allow developers to define their own validation logic and rules, making it possible to handle complex data validation scenarios that may not be covered by generic validators. They provide flexibility and control over the validation process, enabling fine-grained validation tailored to the specific needs of an application.

Creating custom validators typically involves defining a validation function or method and integrating it into the validation pipeline or framework used by the application.

Advantages of custom validators:

Tailored Validation Rules: Custom validators enable developers to define validation rules that align with the specific requirements of the application or domain. This allows for more accurate and precise validation, ensuring that only valid data is accepted.

Complex Validation Scenarios: Custom validators are useful when dealing with complex validation scenarios that cannot be easily handled by built-in validators. They allow developers to implement intricate business rules or validations involving multiple fields or dependencies.

Code Reusability: Once created, custom validators can be reused across multiple parts of the application. They provide a centralized and consistent approach to validation, reducing code duplication and promoting maintainability.

Example:

Let's consider a scenario where a web application requires a custom validation rule for a unique username during user registration. The rule states that the username should be unique across the system and not already taken by another user. Here's how a custom validator can be implemented:

Custom Validator: The custom validator can include logic to query the database and check if the entered username already exists. If a matching username is found, the validation fails, and an error message is returned.

By implementing this custom validator, the application can ensure that each user registers with a unique username, preventing duplicate usernames in the system.

Custom validators provide developers with the flexibility to handle specific validation requirements that are unique to their applications or domains, enhancing the accuracy and reliability of the validation process.

Topic: Data Validation

Data validation is the process of ensuring that data is accurate, consistent, and meets certain predefined criteria or rules. It involves checking data for errors, completeness, integrity, and conformity to specified formats or patterns. Data validation is crucial in various domains, including software development, databases, data analysis, and data entry.

The main objectives of data validation are as follows:

Accuracy: Data validation ensures that the data entered or received is accurate and free from errors or inconsistencies. It helps identify and correct mistakes, reducing the likelihood of incorrect conclusions or decisions based on faulty data.

Completeness: Validation verifies that all required data is present and accounted for. It checks if any mandatory fields are missing or if data is incomplete, preventing incomplete or partial data from being processed or analyzed.

Consistency: Data validation ensures that data adheres to predefined rules, formats, or patterns. It verifies relationships between different data elements, such as ensuring that data values fall within specified ranges or that dependencies between fields are maintained.

Integrity: Validation checks the integrity of data by verifying its reliability and trustworthiness. It identifies anomalies, discrepancies, or outliers that may indicate data corruption or manipulation.

Data validation can be performed through various techniques, such as rule-based validation, format checks, range checks, referential integrity checks, and cross-field validations. These techniques help maintain data quality, reliability, and usability across different applications and systems.

Example:

Consider a scenario where an online shopping application requires data validation during the checkout process. Here are a few examples of data validation checks:

Quantity: The validation ensures that the quantity entered for each item is a positive integer and does not exceed the available stock.

Price: The validation ensures that the price of each item is a positive number and falls within a valid price range.

Address: The validation verifies that the entered shipping address is complete, including street, city, state, and postal code.

By implementing data validation in this scenario, the application can prevent users from submitting incorrect quantities, invalid prices, or incomplete addresses, ensuring a smooth and accurate checkout process.

Topic: Aggregation Middleware

Aggregation middleware is a concept commonly used in web development or API frameworks to combine or aggregate multiple requests or responses into a single consolidated result. It acts as a middleware layer between clients and the server, intercepting and processing incoming requests or outgoing responses.

The main purpose of aggregation middleware is to improve performance, reduce network latency, and optimize data transfer. It eliminates the need for multiple round trips between clients and servers by combining related requests or responses into a single call.

Aggregation middleware typically follows these steps:

Request Interception: The middleware intercepts incoming requests from clients and analyzes them to determine if they can be aggregated. It identifies requests that share common characteristics or parameters.

Aggregation Logic: Once the middleware identifies aggregatable requests, it combines or merges them into a single aggregated request. The aggregation logic can involve grouping requests based on specific criteria, merging payloads, or applying certain transformations.

Server Communication: The middleware sends the aggregated request to the server or backend services. It waits for the responses from the server.

Response Processing: After receiving the responses, the middleware processes them to extract the relevant data. It applies any necessary transformations or formatting before sending the final aggregated response back to the client.

Aggregation middleware can significantly improve performance and reduce network overhead in scenarios where multiple similar requests or responses are involved, such as fetching related data from a database or retrieving data from multiple microservices.

Query Middleware

Query middleware refers to a middleware layer or component in a software system that intercepts and processes database queries or commands before they are executed. It sits between the application code and the database engine, providing additional functionality or logic to enhance the query execution process.

The key features and benefits of query middleware include:

Query Optimization: Query middleware can optimize database queries by analyzing the query structure, indexes, and statistics. It can restructure or rewrite queries to improve performance, reduce execution time, and utilize database resources efficiently.

Caching: Query middleware can implement query result caching, storing the results of frequently executed queries. This helps reduce the load on the database and improves response times for subsequent identical queries.

Security and Access Control: Query middleware can enforce security measures and access controls on queries. It can authenticate and authorize users, validate query parameters, and ensure that only authorized users can execute specific queries.

Logging and Monitoring: Query middleware can log and monitor query execution, providing insights into query performance, bottlenecks, and potential issues. It helps in identifying slow queries, optimizing them, and troubleshooting database-related problems.

Query middleware is commonly used in frameworks, ORM (Object-Relational Mapping) libraries, or database proxy layers to add an extra layer of functionality and control over database interactions.

Document Middleware

Document middleware, also known as document-level middleware, is a concept often associated with document-oriented databases, such as MongoDB. It allows developers to add custom logic, processing, or transformations to documents before they are saved to or retrieved from the database.

Document middleware operates on individual documents within a collection and provides hooks or interceptors at various stages of the document lifecycle, such as before saving, after saving, before updating, or before removing a document. These hooks allow developers to modify or augment the document data or perform additional operations.

Common use cases and benefits of document middleware include:

Data Validation: Document middleware can perform data validation before saving documents to the database. It ensures that the data conforms to predefined rules or requirements, preventing invalid or inconsistent data from being stored.

Data Transformation: Document middleware can transform or modify document data before it is saved or retrieved. It allows developers to manipulate document fields, add calculated properties, format data, or apply specific business rules.

Auditing and Logging: Document middleware can be used to log or record changes to documents. It captures information such as who made the changes, when they occurred, and the nature of the modifications, providing an audit trail for accountability and traceability.

Enforcing Business Rules: Document middleware enables developers to enforce business rules or apply additional logic specific to the application domain. It allows for custom validations, cross-field calculations, or complex workflows related to document processing.

Document middleware provides flexibility and extensibility to document-oriented databases, allowing developers to incorporate custom logic and processing within the database layer itself. It simplifies the implementation of document-centric features and workflows, enhancing the overall functionality and data integrity.

Virtual Properties

Virtual properties, in the context of software development, refer to dynamically generated properties or attributes that are not directly stored in the underlying data model but are computed or derived based on other properties or external factors. These properties are typically defined as methods or functions rather than actual data fields.

The key characteristics and benefits of virtual properties include:

Computed Values: Virtual properties allow developers to define properties that are computed or derived from other data fields or external sources. They can perform calculations, transformations, or aggregations on the fly, providing dynamic and up-to-date values.

Abstraction and Encapsulation: Virtual properties help abstract the underlying data model and provide a higher level of abstraction. They encapsulate complex or derived information, making it easily accessible and reusable without exposing the underlying implementation details.

Data Integrity and Consistency: Virtual properties ensure data integrity and consistency by automatically recalculating or updating derived values whenever the underlying data changes. This helps maintain accurate and synchronized data across related properties or entities.

Performance Optimization: Virtual properties allow developers to defer expensive computations or data retrieval until they are actually needed. By lazily computing values, unnecessary overhead can be avoided, improving overall performance and efficiency.

Virtual properties are commonly used in object-oriented programming, ORM frameworks, and data modeling to represent derived or computed attributes. They provide a flexible and convenient mechanism for working with derived data without the need to store redundant or duplicated information.

Async Validators

Async Validators are used in backend development with Node.js to validate user input or data asynchronously. They are particularly useful when the validation process involves time-consuming operations like making API calls or querying a database.

Async validators involve the use of promises or async/await syntax to handle the asynchronous nature of the validation process. Here's a simplified example in JavaScript:

```
async function validateUsername(username) {
  // Simulating an asynchronous API call
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      if (username === 'johnDoe') {
        reject('Username already taken');
      } else {
        resolve();
      }
    }, 2000); // Simulating a delay of 2 seconds
  });
}
```

```
// Usage
async function submitForm() {
  const username = 'johnDoe';
  try {
    await validateUsername(username);
    console.log('Username is available');
    // Proceed with form submission
  } catch (error) {
    console.error('Username validation failed:', error);
    // Display error message to the user
  }
}
```

Validation Errors

Validation errors are crucial in backend development with Node.js as they provide feedback to users when their input or data fails to meet the specified validation criteria.

Here's a simplified example of handling validation errors in a Node.js application:

```
app.post('/user', (req, res) => {
  const { username, email } = req.body;
  if (!username) {
    return res.status(400).json({ error: 'Username is required' });
  }
  if (!email) {
    return res.status(400).json({ error: 'Email is required' });
  }
  // Proceed with user creation
});
```

SchemaType Options

SchemaType options are essential when defining database schemas in backend development with Node.js. They allow you to specify additional properties or behaviors for individual fields in the schema.

```
const userSchema = new mongoose.Schema({
  name: {
    type: String,
    required: true
  },
  email: {
    type: String,
    required: true,
    unique: true
  },
  age: Number
});
const User = mongoose.model('User', userSchema);
```

Modelling Relationships

Modelling relationships between entities is important in backend development with Node.js. Relationships define how different entities or objects are associated with each other.

When creating relationships, consider the following steps:

Identify the entities: Determine the entities or objects you want to represent in your system, such as users, products, or orders.

Determine the types of relationships: Identify the types of relationships that exist between the entities, such as one-to-one, one-to-many, or many-to-many. For example, a user can have multiple orders, but an order belongs to only one user.

Design the schema: Use the appropriate techniques provided by your chosen database (e.g., MongoDB or relational databases) to establish relationships between entities. This typically involves referencing other entities or using embedded documents.

```
const userSchema = new mongoose.Schema({
  name: String,
  // ...
});
const orderSchema = new mongoose.Schema({
  user: { type: mongoose.Schema.Types.ObjectId, ref: 'User' },
  // ...
});
const User = mongoose.model('User', userSchema);
const Order = mongoose.model('Order', orderSchema);
```

In this example, the user field in the orderSchema references the User model. This establishes a one-to-many relationship where each order belongs to a user.

By properly modelling relationships, you can efficiently query and manipulate data, ensuring data consistency and enabling powerful data retrieval operations.

200 OK: This status code indicates that the request was successful. It is commonly used for successful GET, POST, PUT, or DELETE operations.

201 Created: This status code is typically returned when a new resource has been successfully created. It is commonly used in response to a POST request.

204 No Content: This status code indicates that the request was successful, but there is no content to return in the response. It is commonly used for successful DELETE or PUT operations.

400 Bad Request: This status code indicates that the server cannot process the request due to a client error, such as malformed syntax or invalid parameters. It is commonly used when the request cannot be fulfilled due to user error.

401 Unauthorized: This status code indicates that the request requires authentication, and the client must provide valid credentials to access the requested resource.

403 Forbidden: This status code indicates that the server understood the request, but the client does not have permission to access the requested resource. It differs from 401 in that authentication is not possible or has been refused.

404 Not Found: This status code indicates that the requested resource could not be found on the server. It is commonly used when the URL or endpoint does not exist.

500 Internal Server Error: This status code indicates that an unexpected error occurred on the server, preventing it from fulfilling the request. It is a generic error response when the server encounters an error that does not fall into any specific category.

503 Service Unavailable: This status code indicates that the server is temporarily unable to handle the request due to overloading or maintenance. It is commonly used when the server is undergoing maintenance or experiencing high traffic.

These are just a few examples of commonly used status codes in backend development. There are many more status codes available depending on specific scenarios and requirements.

Referencing Documents and Subdocuments in a Database

In modern database management systems, referencing documents and subdocuments play a crucial role in organizing and linking data efficiently.

This approach allows for the creation of complex data structures, such as parent-child relationships and population tracking. Let's explore some key concepts related to referencing documents and subdocuments.

Child Referencing:

Child referencing involves establishing a relationship between two documents, where one document is considered the child of another. This relationship can be represented using a unique identifier or a reference to the parent document within the child document. Child referencing is commonly used to represent one-to-many or many-to-many relationships. For example, in a student management system, a "Course" document can reference multiple "Student" documents, where each student can be associated with multiple courses.

Example:

{

"_id": "course123",

"name": "Physics 101",

"students": ["student1", "student2", "student3"]

}

Parent Referencing:

Parent referencing is the opposite of child referencing. It involves storing references to the parent document within the child document. This approach enables easy navigation from child to parent documents, enabling queries and data retrieval in hierarchical structures. For instance, in an e-commerce system, a "Product" document can reference its "Category" document as its parent.

Example:

{

"_id": "product123",

"name": "Laptop",

"category": "category456"

}

Population:

Population refers to the act of replacing references with the actual referenced documents during a database query.

It allows you to retrieve a complete document with all its referenced subdocuments in a single query, simplifying data retrieval and reducing the number of queries needed.

This technique is often used when dealing with large and complex data models to enhance performance and simplify programming logic.

Example:


```
{
"_id": "course123",
"name": "Physics 101",
"students": [
{
"_id": "student1",
"name": "John Doe"
},
{
"_id": "student2",
"name": "Jane Smith"
},
{
"_id": "student3",
"name": "Alex Johnson"
}
]
}
```

Embedding Documents:

Embedding documents involves nesting subdocuments within a parent document. Instead of referencing separate documents, the subdocuments are stored directly within the parent document. This approach can be useful when the subdocuments are tightly related to the parent document and don't need to be accessed or modified independently.

Example:


```
{
"_id": "category456",
"name": "Electronics",
"products": [
{
"_id": "product123",
"name": "Laptop"
},
{
"_id": "product789",
"name": "Smartphone"
}
]
}
```

Using an Array of Subdocuments:

Arrays of subdocuments provide a flexible way to store multiple subdocuments within a parent document. This approach allows for easy expansion and modification of the subdocuments without the need for predefined schema structures.

Example:


```
{
"_id": "course123",
"name": "Physics 101",
"lectures": [
{
"title": "Introduction to Physics",
"duration": 60
},
{
"title": "Newton's Laws of Motion",
"duration": 90
}
]
}
```

Transactions

In software development, a transaction is a logical unit of work that consists of multiple database operations. Transactions ensure data integrity by guaranteeing that either all the operations within the transaction are completed successfully, or none of them are applied to the database.

Transactions typically follow the ACID properties:

- Atomicity: A transaction is atomic, meaning it is treated as a single, indivisible operation. If any part of the transaction fails, the entire transaction is rolled back, and the database returns to its previous state.
- Consistency: Transactions maintain the consistency of the database by enforcing integrity constraints. Any changes made by a transaction must adhere to predefined rules and constraints.
- Isolation: Transactions are isolated from each other, meaning that the changes made by one transaction are not visible to other transactions until the first transaction is committed. This ensures that concurrent transactions do not interfere with each other.
- Durability: Once a transaction is committed, its changes are permanent and survive any subsequent failures, such as power outages or system crashes. The changes are stored securely in the database.

Example:

Let's consider a banking application where users can transfer money between accounts. To ensure data integrity and prevent inconsistencies, the money transfer operation should be performed within a transaction.

Begin Transaction: The transaction begins when the user initiates a money transfer request.

Deduct Amount from Sender's Account: The transaction deducts the transfer amount from the sender's account balance.

Add Amount to Receiver's Account: The transaction adds the transfer amount to the receiver's account balance.

Commit Transaction: If both the deduction from the sender's account and addition to the receiver's account are successful, the transaction is committed. The changes become permanent and are reflected in the database.

Rollback Transaction: If any step within the transaction fails, such as insufficient funds in the sender's account, the transaction is rolled back. The sender's account balance remains unchanged, ensuring data consistency.

Authentication & Authorization

Authentication and authorization are two essential concepts in application security that work together to control access to resources and ensure that only authorized users can perform specific actions.

Authentication:

Authentication is the process of verifying the identity of a user or system. It ensures that the user is who they claim to be before granting access to protected resources. Common authentication methods include:

Username and Password: Users provide a unique username and password combination to verify their identity.

Two-Factor Authentication (2FA): In addition to a username and password, users provide a second form of verification, such as a temporary code sent to their mobile device.

Biometric Authentication: Users authenticate using their unique physical or behavioral characteristics, such as fingerprints, facial recognition, or voice recognition.

Authorization:

Authorization determines the actions and resources that an authenticated user can access. It verifies whether a user has the necessary permissions to perform specific operations or access certain data. Authorization is typically based on user roles and privileges. Common authorization mechanisms include:

Role-Based Access Control (RBAC): Users are assigned roles (e.g., admin, manager, employee), and each role has a set of permissions associated with it. Users can only access resources and perform actions allowed by their assigned role.

Access Control Lists (ACL): Access rights are defined for individual users or groups, specifying what they can or cannot do within the application.

Attribute-Based Access Control (ABAC): Access decisions are based on various attributes, such as user attributes (e.g., department, location) and resource attributes (e.g., sensitivity level, ownership).

Example:

Consider a web application with different user roles: "Admin," "Manager," and "Employee." The application allows admins to create, update, and delete user accounts, managers to view employee details, and employees to update their own profiles.

Authentication: Users provide their username and password to log in to the application.

Authorization:

Admin Role: Admins have full access to all application features, including user management.

Manager Role: Managers can access employee details, but they cannot modify user accounts.

Employee Role: Employees can update their own profiles but have no access to other user accounts.

Admin Access: After successful authentication, the system verifies the user's role. If the user is an admin, they gain access to all administrative functionalities.

Manager Access: If the user is a manager, the system grants access to employee details but restricts them from modifying user accounts.

Employee Access: For employees, the system allows them to update their own profiles but denies access to other user accounts or administrative features.

Authentication vs Authorization

Authentication and authorization are often used together, but they serve different purposes in application security:

Authentication:

Authentication verifies the identity of a user or system and ensures that the claimed identity is valid. It answers the question, "Who are you?" Authentication mechanisms determine whether a user is allowed to access the system. Common authentication methods include username and password, biometric authentication, and two-factor authentication.

Authorization:

Authorization determines what an authenticated user can do within the application and what resources they can access. It answers the question, "What are you allowed to do?" Authorization mechanisms control user permissions based on user roles, access control lists, or attribute-based access control. They specify which actions and resources are accessible to a particular user or group.

In summary, authentication is about verifying identity, while authorization is about granting access based on that identity.

Example:

Consider an online document management system. Users must authenticate themselves before accessing the system and then be authorized to perform certain actions based on their roles and permissions.

Authentication: Users provide their username and password to authenticate themselves.

Authorization:

Role-Based Access Control (RBAC): The system assigns users to different roles, such as "Admin," "Manager," and "Employee."

Access Control Lists (ACL): The system defines access rights for individual users or groups, specifying what actions they can perform on specific documents.

Authentication Process: After successful authentication, the system verifies the user's identity and confirms that they are who they claim to be.

Authorization Process: Based on the user's role and permissions, the system determines the actions and resources the user is allowed to access.

Admin Role: Admins have full access to all documents, including creating, reading, updating, and deleting them.

Manager Role: Managers can view and update documents within their department but cannot delete them.

Employee Role: Employees can only read and update their own documents.

Creating User Model

In many applications, a user model is a fundamental component that represents a user's account and associated data. Creating a user model involves defining the attributes and behaviors of a user object, which can vary depending on the specific requirements of the application. Some common attributes of a user model include:

Username: A unique identifier for the user's account.

Password: A secure, hashed representation of the user's password.

Email: The user's email address for communication and account verification.

Name: The user's full name or display name.

Role: The user's role or permissions within the application (e.g., admin, manager, user).

Created At: The timestamp when the user account was created.

Updated At: The timestamp when the user account was last updated.

Example:

Let's consider a social media application. The user model for this application might have the following attributes and behaviors:

Username: A unique identifier for each user's account, such as "@johnsmith."

Password: A secure, hashed representation of the user's password to protect account security.

Email: The user's email address, used for account verification and password reset.

Name: The user's full name, which can be displayed on their profile and in interactions with other users.

Role: The user's role within the application, such as "standard user" or "admin." This determines the level of access and permissions granted to the user.

Created At: A timestamp indicating when the user account was created, useful for auditing and tracking user registration.

Updated At: A timestamp indicating when the user account was last updated, which can be used for various purposes, such as displaying the most recent activity on the user's profile.

By creating a user model with the appropriate attributes and behaviors, you can manage user data, authentication, authorization, and other user-related functionalities in your application.

Lodash

Lodash is a popular JavaScript utility library that provides helpful functions for manipulating and working with arrays, objects, strings, and more. It simplifies common programming tasks and enhances code readability. Here's an example of how you can use Lodash:

Lodash provides numerous functions to handle various scenarios.

```
// Import the Lodash library

const _ = require('lodash');

// Example usage of Lodash functions

const numbers = [1, 2, 3, 4, 5];

// Find the sum of all numbers

const sum = _.sum(numbers);

console.log(sum); // Output: 15

// Remove duplicate values from an array

const uniqueNumbers = _.uniq(numbers);

console.log(uniqueNumbers); // Output: [1, 2, 3, 4, 5]
```

JWT

JSON Web Tokens (JWT):

JSON Web Tokens (JWT) is a compact, URL-safe means of representing claims between two parties. It is commonly used for authentication and authorization purposes in web applications. Here's an example of how you can work with JWTs in JavaScript using the jsonwebtoken library:

```
const jwt = require('jsonwebtoken');

const payload = { // Example of generating a JWT
  userId: '123456789',
  username: 'example_user'
};

const secretKey = 'your-secret-key';

const token = jwt.sign(payload, secretKey);

console.log(token); // Output: <generated JWT>

try { // Example of verifying and decoding a JWT
  const decoded = jwt.verify(token, secretKey);
  console.log(decoded); // Output: { userId: '123456789', username: 'example_user', iat: 1626373820 }
} catch (error) {
  console.log('Invalid token');
}
```

In this example, the jsonwebtoken library is used to generate a JWT with a payload containing user information. Later, the same library is used to verify and decode the JWT using the secret key.

Setting Response Headers:

Setting response headers is essential for controlling the behavior of HTTP responses in web applications. In JavaScript, when building web servers using frameworks like Express.js, you can set response headers using the set method of the response object. Here's an example:

```
const express = require('express');

const app = express();

app.get('/api/data', (req, res) => {

  // Set response headers

  res.set('Content-Type', 'application/json');

  res.set('Cache-Control', 'public, max-age=3600');

  // Send JSON response

  res.json({ message: 'Data response' });

});

app.listen(3000, () => {

  console.log('Server is running on port 3000');

});
```

In this example, when the client makes a GET request to /api/data, the server sets the response headers using the res.set method. The Content-Type header is set to indicate that the response is JSON, and the Cache-Control header specifies caching instructions.

Catching Errors in Async Functions & Removing Try-Catch

- In traditional JavaScript, handling errors in asynchronous functions often involves using try-catch blocks.
- While try-catch is a powerful mechanism to catch synchronous errors, it becomes less convenient and less efficient when dealing with asynchronous code.
- Thankfully, with modern JavaScript and the introduction of async/await, we have a cleaner way to handle errors in async functions.

1. Using Try-Catch (Traditional Approach)

In the traditional approach, asynchronous code is wrapped inside a try block, and errors are caught in the catch block. Here's an example:

```
async function fetchUserData(userId) {
  try {
    const user = await getUserFromDatabase(userId);
    const posts = await getPostsFromDatabase(user.id);
    return { user, posts };
  } catch (error) {
    console.error('Error fetching user data:', error);
    throw new Error('Failed to fetch user data');
  }
}
```

2. Catching Errors in Async Functions (Modern Approach)

With async/await, you can handle errors more elegantly using the .catch() method on the promise returned by the async function. Here's how the above example looks with the modern approach:

```
async function fetchUserData(userId) {
  const user = await getUserFromDatabase(userId).catch((error) => {
    console.error('Error fetching user:', error);
    throw new Error('Failed to fetch user');
  });
  const posts = await getPostsFromDatabase(user.id).catch((error) => {
    console.error('Error fetching posts:', error);
    throw new Error('Failed to fetch posts');
  });
  return { user, posts };
}
```

3. Removing Try-Catch with Promise Rejection

In the modern approach, you can avoid using try-catch altogether by leveraging Promise rejection. If an async function throws an error, it will be automatically caught by the nearest .catch() method up the call stack. This approach makes the code more concise and readable:

```
async function fetchUserData(userId) {

  const user = await getUserFromDatabase(userId);

  const posts = await getPostsFromDatabase(user.id);

  return { user, posts };

}

fetchUserData('user123')

  .then((data) => {

    console.log('User data:', data);

  })

  .catch((error) => {

    console.error('Error fetching user data:', error);

  });
```

Conclusion:

Catching errors in async functions using the modern approach with async/await and .catch() is the preferred method as it simplifies code, improves readability, and leads to better error handling. Removing try-catch blocks in favor of .catch() or Promise rejection can significantly improve the maintainability and clarity of your backend code.

By understanding and implementing these error-handling techniques, developers can create more robust and efficient backend applications, providing a better experience for users and making it easier to manage complex asynchronous operations.

Logging errors using winston package

- Logging errors using Winston is an essential practice in web development.
- It enables developers to gain insights into the application's behavior, diagnose problems, and maintain the application's overall health.
- By utilizing the flexibility and features of the Winston package, developers can build robust error logging mechanisms to improve the reliability and maintainability of their Node.js applications.

Creating the Logger:

Here, the code creates a new logger instance using winston.createLogger(). The logger is configured with several options:

level: 'info': This sets the logging level to "info," which means that any logs with a severity level equal to or higher than "info" will be logged. You can adjust the level to control which logs get recorded.

format: This defines the log format using winston.format.combine(). In this case, it combines two log formats:

- winston.format.timestamp(): This adds a timestamp to each log entry, indicating when the log entry was created.
- winston.format.json(): This formats log entries as JSON objects, making it easier to parse and analyze logs programmatically.

transports: This is an array of transports that dictate where the logs will be stored. In this code, there are two transports specified:

- winston.transports.File({ filename: 'logs/error.log', level: 'error' }): This transport will log messages with severity level "error" and above to a file named "error.log" in the "logs" directory.
- winston.transports.File({ filename: 'logs/combined.log' }): This transport will log all messages to a file named "combined.log" in the "logs" directory.

Adding Console Transport (Optional for Non-Production Environments):

```
if (process.env.NODE_ENV !== 'production') {
  logger.add(new winston.transports.Console({
    format: winston.format.simple(),
  }));
}
```

This part checks if the Node environment is not set to "production" (which usually means it's a development or testing environment). If that condition is true, it adds an additional transport to the logger:

winston.transports.Console: This transport logs messages to the console, so log entries will be visible in the terminal during development and testing.

The format: winston.format.simple() option sets a simpler log format for console logging, making it easier to read in the terminal.

NextJS

Server-Side Rendering (SSR):

Next.js offers built-in server-side rendering capabilities, allowing your React components to be rendered on the server before sending HTML to the client. This improves performance, SEO, and initial load times, especially for content-rich websites.

Static Site Generation (SSG):

Next.js supports static site generation, where pages are pre-rendered at build time, enabling fast and efficient delivery of content. This approach is ideal for websites with mostly static content, such as blogs, marketing sites, or documentation.

Automatic Code Splitting:

Next.js automatically splits your JavaScript code into smaller chunks, ensuring that only the necessary code is loaded when navigating between pages. This results in faster page loads and improved performance.

File-Based Routing:

Next.js uses a simple and intuitive file-based routing system, where each file in the pages directory corresponds to a route in your application. This makes it easy to organize and maintain your project structure.

API Routes:

Next.js allows you to create API routes using the pages/api directory, enabling you to build serverless API endpoints directly within your Next.js application. This simplifies backend development and integration with frontend components.

Built-in CSS and Sass Support:

Next.js provides built-in support for CSS and Sass, allowing you to import stylesheets directly into your components. It also supports CSS Modules for scoped styling and better encapsulation.

Home.module.css

Import ‘./../Home.css ;

Import style from ‘./../home.module.css

Image Optimization:

Next.js offers built-in image optimization features, including automatic image resizing, lazy loading, and support for modern image formats like WebP. This helps improve performance and reduce bandwidth usage on your website.

TypeScript Support:

Next.js has excellent support for TypeScript out of the box, allowing you to write type-safe code and catch errors during development. This leads to better code quality and developer productivity.

Incremental Static Regeneration (ISR):

Next.js introduced Incremental Static Regeneration, a feature that allows you to update static content without rebuilding the entire site. This enables real-time updates for dynamic data while still benefiting from the performance advantages of static site generation.

Community and Ecosystem:

Next.js has a vibrant community and a rich ecosystem of plugins, libraries, and tools that make it easy to extend and customize your application. It's widely used by companies of all sizes and has strong community support.