

1. Write a React component that features a controlled input element for capturing a user's name. Implement the required state and event handlers to make sure the input value is managed by the component's state. Additionally, include a submit button that logs the entered name to the console when clicked.

jsx

Copy code

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

const NameForm = () => {
  const [name, setName] = useState('');

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

  const handleSubmit = (event) => {
    event.preventDefault();
    console.log('Entered name:', name);
  };

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

export default NameForm;
```

jsx

Copy code

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

const App = () => {
  return (
    <div>
      <h1>Enter Your Name</h1>
      <NameForm />
    </div>
  );
};

export default App;
```

2. Create a React application that features a parent component (ListManager) and a stateless functional child component (ListItem). The ListManager component should maintain a list of items with each item having a name and a delete button. The ListItem component should be responsible for displaying an individual item and should communicate with the ListManager to remove itself from the list when its delete button is clicked.
  - i. Implement the ListItem component as a stateless functional component.
  - ii. Use the ListManager component to manage an array of items, each represented by the ListItem component.
  - iii. Allow each ListItem component to trigger the removal of its corresponding item from the list when its delete button is clicked.

## ListManager Component

The `ListManager` component manages an array of items and renders a `ListItem` component for each item.

```
jsx Copy code

import React, { useState } from 'react';
import ListItem from './ListItem';

const ListManager = () => {
  const [items, setItems] = useState([
    { id: 1, name: 'Item 1' },
    { id: 2, name: 'Item 2' },
    { id: 3, name: 'Item 3' },
  ]);

  const handleDelete = (id) => {
    setItems(items.filter(item => item.id !== id));
  };


  return (
    <div>
      {items.map(item => (
        <ListItem key={item.id} item={item} onDelete={handleDelete} />
      ))}
    </div>
  );
};

export default ListManager;
```

## ListItem Component

The `ListItem` component is a stateless functional component responsible for displaying an individual item and triggering its deletion.

jsx

 Copy code

```
import React from 'react';


const ListItem = ({ item, onDelete }) => (
  <div>
    {item.name}
    <button onClick={() => onDelete(item.id)}>Delete</button>
  </div>
);

export default ListItem;
```

## App Component

The `App` component imports and uses the `ListManager` component.

jsx

 Copy code

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

const App = () => (
  <div>
    <h1>Item List</h1>
    <ListManager />
  </div>
);

export default App;
```

Create a React project with the following file structure: In the Navbar.js file, create a functional component called Navbar that renders a navigation bar with links to Home, About, and Contact. In the MainContent.js file, create a functional component called MainContent that renders the main content of the website. Include a section with a heading and some dummy text. In the Sidebar.js file, create a functional component called Sidebar that renders a sidebar with links to different sections of the website. In the Footer.js file, create a functional component called Footer that renders a footer element with the text "© 2024 My Website". In the App.js file, import the Navbar, MainContent, Sidebar, and Footer components from their respective files. Render these components within a parent div element in the following order: Navbar, Sidebar, MainContent, and Footer.

### `Navbar.js`

Create a functional component called `Navbar` that renders a navigation bar with links to Home, About, and Contact.

jsx

Copy code

```
import React from 'react';


const Navbar = () => (
  <nav>
    <ul>
      <li><a href="#home">Home</a></li>
      <li><a href="#about">About</a></li>
      <li><a href="#contact">Contact</a></li>
    </ul>
  </nav>
);

export default Navbar;
```

## `MainContent.js`

Create a functional component called `MainContent` that renders the main content of the website.

jsx

 Copy code

```
import React from 'react';


const MainContent = () => (
  <main>
    <section>
      <h1>Welcome to My Website</h1>
      <p>This is some dummy text for the main content section of the website.</p>
    </section>
  </main>
);

export default MainContent;
```

## `Sidebar.js`

Create a functional component called `Sidebar` that renders a sidebar with links to different sections of the website.

jsx

 Copy code

```
import React from 'react';


const Sidebar = () => (
  <aside>
    <ul>
      <li><a href="#section1">Section 1</a></li>
      <li><a href="#section2">Section 2</a></li>
      <li><a href="#section3">Section 3</a></li>
    </ul>
  </aside>
);

export default Sidebar;
```

### `Footer.js`

Create a functional component called **`Footer`** that renders a footer element with the text "© 2024 My Website".

jsx

 Copy code

```
import React from 'react';


const Footer = () => (
  <footer>
    © 2024 My Website
  </footer>
);

export default Footer;
```

### `App.js`

Import and render the **`Navbar`**, **`MainContent`**, **`Sidebar`**, and **`Footer`** components within a parent **`div`** element.

jsx

 Copy code

```
import React from 'react';
import Navbar from './components/Navbar';
import MainContent from './components/MainContent';
import Sidebar from './components/Sidebar';
import Footer from './components/Footer';


const App = () => (
  <div>
    <Navbar />
    <Sidebar />
    <MainContent />
    <Footer />
  </div>
);

export default App;
```

### `index.js`

Ensure your project has an entry point for rendering the **`App`** component.

jsx

 Copy code


```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App';

ReactDOM.render(<App />, document.getElementById('root'));
```

1. Create a React form that captures a user's email and password. Implement form validation to ensure the email follows a valid format (e.g., contains an '@' symbol) and the password meets specific criteria (e.g., minimum length). Display appropriate error messages when validation fails, and enable form submission only when the input meets the validation criteria. Provide a detailed explanation of your validation approach and any libraries used.

## App.js

jsx

 Copy code

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

const App = () => {
  const [email, setEmail] = useState('');
  const [password, setPassword] = useState('');
  const [errors, setErrors] = useState({});

  const validateEmail = (email) => {
    // Basic email validation: checks for "@" symbol
    return email.includes('@');
  };

  const validatePassword = (password) => {
    // Password validation: minimum length of 6 characters
    return password.length >= 6;
  };

  const handleSubmit = (e) => {
    e.preventDefault();
    const newErrors = {};

    if (!validateEmail(email)) {
      newErrors.email = 'Invalid email format.';
    }

    if (!validatePassword(password)) {
      newErrors.password = 'Password must be at least 6 characters long.';
    }
  };
}
```



```

if (Object.keys(newErrors).length === 0) {
  // Form is valid, proceed with form submission
  console.log('Form submitted:', { email, password });
} else {
  // Set errors to display them
  setErrors(newErrors);
}
};

return (
  <div>
    <h1>Sign In</h1>
    <form onSubmit={handleSubmit}>
      <div>
        <label>Email:</label>
        <input
          type="email"
          value={email}
          onChange={(e) => setEmail(e.target.value)}
        />
        {errors.email && <p style={{ color: 'red' }}>{errors.email}</p>}
      </div>
      <div>
        <label>Password:</label>
        <input
          type="password"
          value={password}
          onChange={(e) => setPassword(e.target.value)}
        />
        {errors.password && <p style={{ color: 'red' }}>{errors.password}</p>}
      </div>
      <button type="submit" disabled={!email || !password}>
        Submit
      </button>
    </form>
  </div>
);


```

2. Build a React application with three functional components: Title, Description, and Page. Start by defining the Title component in the Title.js file to render a static title, such as "Welcome to My React App." Next, create the Description component in the Description.js file to display a static description, such as "Explore the world of React components." Finally, compose both the Title and Description components within the Page component, implemented in the Page.js file. The Page component should render a cohesive webpage with a title and description without explicitly passing any data as props between the components. Ensure your code adheres to React best practices, maintains an organized project structure, and consider incorporating additional styling or content for an enhanced user experience.

### `Title.js`

Create a functional component called **`Title`** that renders a static title.

jsx

 Copy code

```
import React from 'react';


const Title = () => (
  <h1>Welcome to My React App</h1>
);

export default Title;
```

### `Description.js`

Create a functional component called **`Description`** that displays a static description.

jsx

 Copy code

```
import React from 'react';


const Description = () => (
  <p>Explore the world of React components.</p>
);

export default Description;
```

### `Page.js`

Compose both the ``Title`` and ``Description`` components within the ``Page`` component.

jsx

 Copy code

```
import React from 'react';
import Title from './Title';
import Description from './Description';


const Page = () => (
  <div>
    <Title />
    <Description />
  </div>
);

export default Page;
```

### `App.js`

Import and render the ``Page`` component.

jsx

 Copy code

```
import React from 'react';
import Page from './components/Page';


const App = () => (
  <div>
    <Page />
  </div>
);

export default App;
```

### `index.js`

Ensure your project has an entry point for rendering the ``App`` component.

jsx

 Copy code

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App';


ReactDOM.render(<App />, document.getElementById('root'));
```

3. Create a React functional component named `UserInfo` that accepts two properties, `username` and `userRole`, from its parent component. Inside the `UserInfo` component, display the provided `username` and `userRole` within a paragraph element. Ensure the component structure adheres to React best practices. This component is designed to be reusable and can dynamically showcase user information based on the data passed to it from the parent component.

### `'UserInfo.js'`

Create the `'UserInfo'` component that accepts `'username'` and `'userRole'` as props.

jsx

 Copy code

```
import React from 'react';
import PropTypes from 'prop-types';

const UserInfo = ({ username, userRole }) => (
  <div>
    <p>
      Username: {username}<br />
      Role: {userRole}
    </p>
  </div>
);


UserInfo.propTypes = {
  username: PropTypes.string.isRequired,
  userRole: PropTypes.string.isRequired,
};

export default UserInfo;
```

### `'App.js'`

Create a parent component that uses the `'UserInfo'` component and passes `'username'` and `'userRole'` as props.

jsx

 Copy code

```
import React from 'react';
import UserInfo from '../components/UserInfo';


const App = () => (
  <div>
    <h1>User Information</h1>
    <UserInfo username="JohnDoe" userRole="Admin" />
    <UserInfo username="JaneSmith" userRole="User" />
  </div>
);

export default App;
```

4. Create a React component for a simple login form that includes fields for username and password. Implement client-side form validation to ensure both fields are non-empty before allowing form submission. Display appropriate error messages if the fields are empty and prevent form submission until they are filled.

### ``LoginForm.js``

jsx

 Copy code

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

const LoginForm = () => {
  const [username, setUsername] = useState('');
  const [password, setPassword] = useState('');
  const [error, setError] = useState('');

  const handleSubmit = (e) => {
    e.preventDefault();
    if (!username || !password) {
      setError('Both fields are required.');
```

```
    } else {
      setError('');
      console.log('Form submitted:', { username, password });
    }
  };

  return (
    <div>
      <h1>Login</h1>
      <form onSubmit={handleSubmit}>
        <div>
          <label>Username:</label>
          <input
            type="text"
            value={username}
            onChange={(e) => setUsername(e.target.value)}
          />
        </div>
        <div>
          <label>Password:</label>
          <input
            type="password"
            value={password}
            onChange={(e) => setPassword(e.target.value)}
          />
        </div>
        {error && <p style={{ color: 'red' }}>{error}</p>}
        <button type="submit">Submit</button>
      </form>
    </div>
  );
};

export default LoginForm;
```

5. Develop a React form component (StyledForm) that includes input fields for username, email, and password. Implement styling using both native CSS and a SaaS (e.g., SCSS) preprocessor. The form should include client-side validation for the email field, checking for a valid email format. The form should display appropriate visual feedback for validation errors, such as changing the border color of the input field.
- Create the StyledForm component using both native CSS and a SaaS preprocessor for styling.
  - Implement input fields for username, email, and password within the form.
  - Add client-side validation specifically for the email field to ensure it follows a valid email format.
  - Apply styling changes to visually indicate validation errors, focusing on the email input field.

### Solution

Certainly! Here's a simplified version of a React form component (`StyledForm`) that includes input fields for username, email, and password with basic styling and client-side validation for the email field.

```
### `StyledForm.js`
```

```
``jsx
```

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

```
import './StyledForm.css';
```

```
const StyledForm = () => {
```

```
  const [username, setUsername] = useState("");
```

```
  const [email, setEmail] = useState("");
```

```
  const [password, setPassword] = useState("");
```

```
  const [error, setError] = useState("");
```

```
  const validateEmail = (email) => {
```

```
    const emailRegex = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;
```

```
    return emailRegex.test(email);
```

```
  };
```

```
  const handleSubmit = (e) => {
```

```

e.preventDefault();
if (!validateEmail(email)) {
  setError('Invalid email format. ');
} else {
  setError('');
  console.log('Form submitted:', { username, email, password });
}
};

```

```

return (
  <div className="styled-form">
    <h1>Sign Up</h1>
    <form onSubmit={handleSubmit}>
      <div className="form-group">
        <label>Username:</label>
        <input
          type="text"
          value={username}
          onChange={(e) => setUsername(e.target.value)}
        />
      </div>
      <div className="form-group">
        <label>Email:</label>
        <input
          type="email"
          value={email}
          onChange={(e) => setEmail(e.target.value)}
          className={error ? 'error' : ''}
        />
        {error && <p className="error-message">{error}</p>}
      </div>
    </form>
  </div>
)

```



```

    </div>

    <div className="form-group">
      <label>Password:</label>

      <input
        type="password"
        value={password}
        onChange={(e) => setPassword(e.target.value)}
      />
    </div>

    <button type="submit">Submit</button>
  </form>
</div>

);
};

export default StyledForm;
'''

### `StyledForm.css`

```css
.styled-form {
  max-width: 400px;
  margin: 0 auto;
  padding: 20px;
  border: 1px solid #ccc;
  border-radius: 8px;
  background-color: #f9f9f9;
}

```

```
h1 {  
  text-align: center;  
  margin-bottom: 20px;  
}
```

```
.form-group {  
  margin-bottom: 15px;  
}
```

```
label {  
  display: block;  
  margin-bottom: 5px;  
}
```

```
input {  
  width: 100%;  
  padding: 8px;  
  border: 1px solid #ccc;  
  border-radius: 4px;  
  transition: border-color 0.3s;  
}
```

```
input.error {  
  border-color: red;  
}
```

```
.error-message {  
  color: red;  
  margin-top: 5px;  
}
```

```
button {  
  width: 100%;  
  padding: 10px;  
  background-color: #007bff;  
  border: none;  
  border-radius: 4px;  
  color: #fff;  
  font-size: 16px;  
  cursor: pointer;  
}
```

```
button:hover {  
  background-color: #0056b3;  
}  
...
```

```
### `App.js`
```

```
``jsx  
import React from 'react';  
import StyledForm from './StyledForm';  
  
const App = () => (  
  <div>  
    <StyledForm />  
  </div>  
)  
  
export default App;
```

'''

### ### Explanation

#### 1. **\*\*Component (`StyledForm.js`)\*\*:**

- The component uses `useState` to manage state for `username`, `email`, `password`, and `error`.
- The `validateEmail` function checks if the email format is valid using a regular expression.
- The `handleSubmit` function handles form submission, validates the email, and sets an error message if validation fails.
- The `className` for the email input field is conditionally applied based on validation results.

#### 2. **\*\*Styling (`StyledForm.css`)\*\*:**

- Basic CSS is used to style the form.
- The `error` class changes the border color of the input field when validation fails.
- Additional styling ensures a user-friendly and visually appealing form layout.


This simplified version focuses on essential functionality and readability, ensuring both ease of use and clarity.

6. Create three React functional components: Title, Description, and Page. In Title.js, develop a component that renders a static title, like "Welcome to My React App." In Description.js, create a component that displays a static description, such as "Explore the world of React components." Then, in Page.js, compose both the Title and Description components. The Page component should render a page with a title and description without explicitly passing any data as props. Ensure the project structure is organized, adheres to React best practices, and displays the composed components on a webpage. Additional styling or content can be applied to enhance the page's appearance and user experience.

### **`Title.js`**

Create a component that renders a static title.

jsx

 Copy code

```
// src/Title.js
import React from 'react';


const Title = () => {
  return <h1>Welcome to My React App</h1>;
};

export default Title;
```

### **`Description.js`**

Create a component that renders a static description.

jsx

 Copy code


```
// src/Description.js
import React from 'react';

const Description = () => {
  return <p>Explore the world of React components.</p>;
};

export default Description;
```

7.

jsx

 Copy code

```
// src/Page.js
import React from 'react';
import Title from './Title';
import Description from './Description';


const Page = () => {
  return (
    <div>
      <Title />
      <Description />
    </div>
  );
};

export default Page;
```

### ``App.js``

Update ``App.js`` to use the ``Page`` component.

jsx

 Copy code

```
// src/App.js
import React from 'react';
import Page from './Page';

const App = () => (
  <div>
    <Page />
  </div>
);

export default App;
```



8. Explain the concept of indexing in MongoDB and how it can be used for query optimization. Provide an example scenario where indexing would significantly improve the performance of a MongoDB query.

## Simple Explanation of Indexing in MongoDB

**Indexing** in MongoDB is like an index in a book. It helps you find information quickly without having to read every page. In a MongoDB database, an index is a special data structure that improves the speed of data retrieval operations on a collection.

### How Indexing Works

- **Without an Index:** MongoDB must scan every document in a collection to find those that match a query. This is slow, especially for large collections.
- **With an Index:** MongoDB can quickly locate the documents that match the query criteria without scanning the entire collection.

### Types of Indexes

1. **Single Field Index:** Indexes on one field.
2. **Compound Index:** Indexes on multiple fields.
3. **Multikey Index:** Indexes for fields that contain arrays.
4. **Text Index:** Indexes for text search.
5. **Geospatial Index:** Indexes for location-based queries.
6. **Hashed Index:** Indexes based on a hashed value of a field.

### Example Scenario

Imagine you have a collection called `users` with many documents. Each document has fields like `name`, `email`, and `age`.

#### Query Without Index


If you want to find a user by their email:



## Query Without Index

If you want to find a user by their email:

javascript

 Copy code


```
db.users.find({ email: "alice@example.com" })
```

Without an index on the `email` field, MongoDB has to look at every document in the `users` collection to find a match. This is slow for large collections.

## Adding an Index

You can create an index on the `email` field to make this query faster:

javascript

 Copy code

```
db.users.createIndex({ email: 1 })
```

Here, `1` means the index is in ascending order.

## Query With Index

Now, when you run the same query, MongoDB can quickly use the index to find the matching document. It doesn't need to scan the entire collection, which makes the query much faster.

## Benefits

- **Faster Queries:** Using indexes makes finding data much quicker.
- **Efficient Searches:** Indexes help MongoDB locate data without scanning all documents.

## Conclusion

Indexes in MongoDB are essential for optimizing query performance. They act like a book's index, allowing the database to find information quickly without checking every single document. By creating indexes on fields you frequently search, you ensure your database runs efficiently.


9. Explain in simple terms what a MongoDB document is and how it differs from a traditional row in a relational database. Additionally, describe how you would query a MongoDB collection to find all documents where the "status" field equals "active."

## Simple Explanation of a MongoDB Document

### MongoDB Document:

- A MongoDB document is a single record in a MongoDB collection.
- It is a JSON-like format, meaning it is made up of key-value pairs.
- Example of a document:

json

 Copy code

```
{
  "_id": "12345",
  "name": "Alice",
  "age": 30,
  "status": "active"
}
```

## Difference from a Traditional Row in a Relational Database

### Traditional Row in a Relational Database:

- A row in a relational database is a single record in a table.
- It consists of values organized into columns.
- Example of a row in a `users` table:

id	name	age	status
12345	Alice	30	active

## Key Differences:

### 1. Schema Flexibility:

- **MongoDB:** Each document in a collection can have a different structure (schema).
- **Relational Database:** Each row in a table must adhere to a fixed schema.

### 2. Embedded Data:

- **MongoDB:** Documents can contain nested structures (documents within documents).
- **Relational Database:** Rows cannot directly contain other rows.

### 3. Storage:

- **MongoDB:** Stores data in collections.
- **Relational Database:** Stores data in tables.

## Querying a MongoDB Collection

To find all documents where the "status" field equals "active" in a MongoDB collection, you use the `find` method.


### Example Scenario

Suppose you have a collection named `users` and you want to find all documents with the "status" field set to "active".

## MongoDB Query

### 1. Using the Mongo Shell:


javascript

 Copy code

```
db.users.find({ status: "active" })
```

### 2. Using a MongoDB Driver in JavaScript (Node.js):

javascript

 Copy code

```
const { MongoClient } = require('mongodb');

async function findActiveUsers() {
  const uri = "your_mongodb_connection_string";
  const client = new MongoClient(uri, { useNewUrlParser: true, useUnifiedTopology: true });

  try {
    await client.connect();
    const database = client.db('your_database_name');
    const collection = database.collection('users');


    const activeUsers = await collection.find({ status: "active" }).toArray();
    console.log(activeUsers);
  } finally {
    await client.close();
  }
}

findActiveUsers().catch(console.error);
```

10. Implement a Node.js script that utilizes the built-in events module. Create an EventEmitter object that emits a custom event, for example, "dataReceived," with some payload. Develop a listener function that logs the received payload when the custom event is emitted. Demonstrate the functionality by emitting the custom event and observing the log output.

### Simple Code Example

javascript

 Copy code

```
// Import the events module
const EventEmitter = require('events');

// Create an EventEmitter object
const eventEmitter = new EventEmitter();

// Define a listener function for the 'dataReceived' event
eventEmitter.on('dataReceived', (payload) => {
  console.log('Data received:', payload);
});

// Emit the 'dataReceived' event with a payload
const dataPayload = { id: 1, message: 'Hello, world!' };
eventEmitter.emit('dataReceived', dataPayload);
```

11. Define the concept of middleware in the context of Express.js. Discuss the role of middleware functions and how they are used in an Express.js application. Provide examples of common use cases for middleware, such as logging, authentication, and error handling

## Middleware in Express.js

**Middleware** in Express.js refers to functions that execute during the lifecycle of an HTTP request to the Express server. Each middleware function can:

1. Execute any code.
2. Modify the request and response objects.
3. End the request-response cycle.
4. Call the next middleware function in the stack.

If a middleware function does not end the request-response cycle, it must call `next()` to pass control to the next middleware function, otherwise, the request will be left hanging.

## Role of Middleware Functions

Middleware functions are essential in Express.js applications for:

1. **Processing Requests:** They can parse incoming request bodies, cookies, and headers.
2. **Enhancing Security:** They can handle authentication and authorization.
3. **Logging:** They can log request details for monitoring and debugging.
4. **Error Handling:** They can catch and handle errors uniformly.

## How Middleware Functions are Used

Middleware functions can be applied at:


1. **Application Level:** Applied to every request.
2. **Router Level:** Applied to specific routers.
3. **Route Level:** Applied to specific routes.

## Common Use Cases for Middleware

### 1. Logging

Logging middleware captures details about each incoming request and logs them for monitoring and debugging.

javascript


 Copy code

```
const morgan = require('morgan');
app.use(morgan('combined'));
```

### 2. Authentication

Authentication middleware verifies the identity of a user before allowing access to certain routes.

javascript

 Copy code


```
const authenticate = (req, res, next) => {
  if (req.isAuthenticated()) {
    return next();
  }
  res.redirect('/login');
};

app.get('/profile', authenticate, (req, res) => {
  res.send('User Profile');
});
```

### 3. Error Handling

Error handling middleware catches errors and sends a standardized response to the client.

javascript


 Copy code

```
app.use((err, req, res, next) => {
  console.error(err.stack);
  res.status(500).send('Something broke!');
});
```



12. Develop an Express.js application that exposes RESTful endpoints for a simple resource, such as "books." Create routes to handle GET, POST, PUT, and DELETE requests for the "books" resource. Use an in-memory array as a data store for simplicity, and implement appropriate responses for each HTTP method.

javascript

 Copy code

```
const express = require('express');
const bodyParser = require('body-parser');

const app = express();
const PORT = 3000;

// Middleware to parse JSON bodies
app.use(bodyParser.json());

// In-memory array to store books
let books = [];

// Utility function to generate a new book ID
const generateId = () => books.length ? Math.max(...books.map(book => book.id)) + 1 : 1;

// GET /books - Retrieve all books
app.get('/books', (req, res) => {
  res.json(books);
});

// GET /books/:id - Retrieve a book by ID
app.get('/books/:id', (req, res) => {
  const book = books.find(b => b.id === parseInt(req.params.id));
  if (!book) {
    return res.status(404).json({ message: 'Book not found' });
  }
  res.json(book);
});
```

```

// POST /books - Create a new book
app.post('/books', (req, res) => {
  const { title, author } = req.body;
  if (!title || !author) {
    return res.status(400).json({ message: 'Title and Author are required' });
  }
  const newBook = { id: generateId(), title, author };
  books.push(newBook);
  res.status(201).json(newBook);
});

// PUT /books/:id - Update a book by ID
app.put('/books/:id', (req, res) => {
  const { title, author } = req.body;
  const bookIndex = books.findIndex(b => b.id === parseInt(req.params.id));
  if (bookIndex === -1) {
    return res.status(404).json({ message: 'Book not found' });
  }
  if (!title || !author) {
    return res.status(400).json({ message: 'Title and Author are required' });
  }
  books[bookIndex] = { id: parseInt(req.params.id), title, author };
  res.json(books[bookIndex]);
});

```

```

// DELETE /books/:id - Delete a book by ID
app.delete('/books/:id', (req, res) => {
  const bookIndex = books.findIndex(b => b.id === parseInt(req.params.id));
  if (bookIndex === -1) {
    return res.status(404).json({ message: 'Book not found' });
  }
  const deletedBook = books.splice(bookIndex, 1);
  res.json(deletedBook);
});

// Start the server
app.listen(PORT, () => {
  console.log(`Server is running on port ${PORT}`);
});

```

15. Explain the concept of the Node.js event loop and how it enables asynchronous operations in Node.js. Discuss the importance of non-blocking I/O operations in the event loop. Provide an example scenario where the event loop enhances the performance of a Node.js application.

## Concept of the Node.js Event Loop

The **Node.js event loop** is a core part of the Node.js runtime, responsible for handling asynchronous operations. It allows Node.js to perform non-blocking I/O operations, even though JavaScript is single-threaded. The event loop enables Node.js to execute callbacks and handle multiple operations concurrently, making it efficient for I/O-heavy tasks.

## How the Event Loop Works

1. **Phases:** The event loop operates in multiple phases. Each phase has a specific purpose and handles a specific type of callback. The primary phases include:
  - **Timers:** Executes callbacks scheduled by `setTimeout` and `setInterval`.
  - **Pending Callbacks:** Executes I/O callbacks deferred to the next loop iteration.
  - **Idle, Prepare:** Internal use.
  - **Poll:** Retrieves new I/O events; executes I/O-related callbacks.
  - **Check:** Executes callbacks from `setImmediate`.
  - **Close Callbacks:** Executes close event callbacks, e.g., `socket.on('close', ...)`.
2. **Callback Queue:** When an asynchronous operation completes, its callback is pushed to the callback queue. The event loop picks up these callbacks and executes them in their respective phases.
3. **Non-Blocking I/O:** Node.js uses non-blocking I/O operations to perform multiple tasks simultaneously. This approach ensures that the event loop can continue processing other operations without waiting for I/O operations to complete.

## Importance of Non-Blocking I/O

Non-blocking I/O operations are crucial for the event loop because they prevent the entire application from being blocked by a single I/O operation. Instead of waiting for a file to be read or a database query to complete, Node.js can continue executing other tasks. This behavior leads to more efficient resource utilization and better performance, especially in I/O-bound applications.

## Example Scenario


Consider a web server that handles multiple requests from clients. In a traditional blocking I/O model, the server would handle one request at a time, waiting for I/O operations (like reading a file or querying a database) to complete before moving on to the next request. This approach can lead to significant delays and inefficient resource use.

In contrast, Node.js, with its event loop and non-blocking I/O, can handle multiple requests simultaneously. When a request involves an I/O operation, Node.js initiates the operation and continues processing other requests. When the I/O operation completes, the event loop picks up the callback and processes it.

## Example Code

Here's an example of how the event loop and non-blocking I/O enhance performance in a Node.js application:

javascript

 Copy code

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

const server = http.createServer((req, res) => {
  if (req.url === '/file') {
    // Non-blocking I/O operation
    fs.readFile('largefile.txt', 'utf8', (err, data) => {
      if (err) {
        res.writeHead(500);
        res.end('Error reading file');
        return;
      }
      res.writeHead(200, { 'Content-Type': 'text/plain' });
      res.end(data);
    });
  } else {
    res.writeHead(200, { 'Content-Type': 'text/plain' });
    res.end('Hello, World!');
  }
});


server.listen(3000, () => {
  console.log('Server listening on port 3000');
});
```

13. Create a Node.js script that reads the content of a text file named "example.txt" using the File System module. Log the file's content to the console. Implement error handling to log an error message if the file reading operation fails.

## Simple Code

1. **Create the `example.txt` file:** Make sure you have a file named `example.txt` in the same directory as your script.
2. **Create `readFile.js`:** Create a file named `readFile.js` and add the following code:

javascript

 Copy code

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

// Read the file content
fs.readFile('example.txt', 'utf8', (err, data) => {
  if (err) {
    console.error('Error reading the file:', err.message);
    return;
  }
  console.log('File content:', data);
});
```

14. Describe the purpose of `module.exports` in Node.js and how it aids in creating modular code. Explain the difference between exporting a single function or object versus exporting multiple functions or objects. Discuss how using modules enhances code maintainability and reusability in Node.js applications.

### Purpose of `module.exports` in Node.js

In Node.js, `module.exports` is used to export functions, objects, or values from a module so that they can be imported and used in other modules. This feature enables the creation of modular code, allowing developers to break down their application into smaller, reusable components.

### How `module.exports` Aids in Creating Modular Code

1. **Encapsulation:** Each module can encapsulate its functionality, exposing only what is necessary and hiding the rest.
2. **Reusability:** Modules can be reused across different parts of an application or even in different applications.
3. **Maintainability:** By breaking down the code into smaller, manageable pieces, it becomes easier to maintain and update.
4. **Separation of Concerns:** Each module can focus on a specific aspect of the application, promoting a clean and organized codebase.


### Exporting a Single Function or Object vs. Multiple Functions or Objects

#### Exporting a Single Function or Object

When you export a single function or object, `module.exports` is assigned directly to that function or object. This approach is straightforward and useful when a module has one primary responsibility or functionality.

### Example: Exporting a Single Function


javascript

 Copy code

```
// greet.js
module.exports = function(name) {
  return `Hello, ${name}!`;
};
```

### Importing and Using the Single Function

javascript

 Copy code


```
// app.js
const greet = require('./greet');
console.log(greet('Alice')); // Output: Hello, Alice!
```

### Exporting Multiple Functions or Objects

When exporting multiple functions or objects, you typically assign them as properties of the `module.exports` object. This method is useful when a module needs to expose multiple functionalities.

### Example: Exporting a Single Function


javascript

 Copy code

```
// greet.js
module.exports = function(name) {
  return `Hello, ${name}!`;
};
```

### Importing and Using the Single Function

javascript

 Copy code


```
// app.js
const greet = require('./greet');
console.log(greet('Alice')); // Output: Hello, Alice!
```

### Exporting Multiple Functions or Objects

When exporting multiple functions or objects, you typically assign them as properties of the `module.exports` object. This method is useful when a module needs to expose multiple functionalities.



javascript


 Copy code

```
// math.js
module.exports.add = function(a, b) {
  return a + b;
};

module.exports.subtract = function(a, b) {
  return a - b;
};
```

### Importing and Using the Multiple Functions

javascript

 Copy code

```
// app.js
const math = require('./math');
console.log(math.add(5, 3)); // Output: 8
console.log(math.subtract(5, 3)); // Output: 2
```

### Enhancing Code Maintainability and Reusability with Modules

1. **Maintainability:** By dividing code into separate modules, it becomes easier to isolate and fix bugs, update functionality, and manage dependencies. Each module can be tested independently, leading to more reliable and robust code.
2. **Reusability:** Modules can be reused across different parts of an application or in different projects. For example, a utility module for date formatting can be reused wherever date formatting is needed.
3. **Collaboration:** Modular code allows multiple developers to work on different parts of an application simultaneously without interfering with each other's work. Each developer can focus on specific modules, leading to more efficient development.
4. **Organization:** Using modules helps keep the codebase organized. Related functionalities are grouped together, making it easier to navigate and understand the code structure.


## Shopping Cart Application using React

### 2. Create Components

`ProductList.js`

Create a `ProductList` component that displays a list of products with an "Add to Cart" button.

jsx

 Copy code

```
// src/ProductList.js
import React from 'react';


const ProductList = ({ products, addToCart }) => {
  return (
    <div>
      <h2>Product List</h2>
      <ul>
        {products.map((product) => (
          <li key={product.id}>
            {product.name} - ${product.price}
            <button onClick={() => addToCart(product)}>Add to Cart</button>
          </li>
        ))}
      </ul>
    </div>
  );
};

export default ProductList;
```

### ``ShoppingCart.js``

Create a ``ShoppingCart`` component that displays the items added to the cart and calculates the total price.

jsx

 Copy code

```
// src/ShoppingCart.js
import React from 'react';

const ShoppingCart = ({ cartItems }) => {
  const totalPrice = cartItems.reduce((total, item) => total + item.price, 0);


  return (
    <div>
      <h2>Shopping Cart</h2>
      <ul>
        {cartItems.map((item, index) => (
          <li key={index}>
            {item.name} - ${item.price}
          </li>
        ))}
      </ul>
      <h3>Total: ${totalPrice.toFixed(2)}</h3>
    </div>
  );
};

export default ShoppingCart;
```

### `App.js`

Create the main **`App`** component to manage state and handle the addition of products to the shopping cart.

jsx

 Copy code

```
// src/App.js
import React, { useState } from 'react';
import ProductList from './ProductList';
import ShoppingCart from './ShoppingCart';

const App = () => {
  const [cartItems, setCartItems] = useState([]);

  const products = [
    { id: 1, name: 'Product 1', price: 10.0 },
    { id: 2, name: 'Product 2', price: 20.0 },
    { id: 3, name: 'Product 3', price: 30.0 },
  ];

  const addToCart = (product) => {
    setCartItems([...cartItems, product]);
  };

  return (
    <div>
      <h1>Shopping Cart Application</h1>
      <ProductList products={products} addToCart={addToCart} />
      <ShoppingCart cartItems={cartItems} />
    </div>
  );
};

export default App;
```




## React Application with MongoDB Integration

### Backend (server.js)

Set up a basic Express server to handle MongoDB operations:

javascript

 Copy code

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

const app = express();
app.use(cors());
app.use(express.json());

mongoose.connect('mongodb://localhost:27017/mydatabase', { useNewUrlParser: true, u

const DataSchema = new mongoose.Schema({
  name: String,
  value: Number
});
const DataModel = mongoose.model('Data', DataSchema);

app.get('/api/data', async (req, res) => {
  const data = await DataModel.find();
  res.json(data);
});


app.post('/api/data', async (req, res) => {
  const newData = new DataModel(req.body);
  await newData.save();
  res.json(newData);
});

app.listen(5000, () => console.log('Server running on http://localhost:5000'));
```



Frontend (React)

javascript

 Copy code

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

const MongoDBIntegration = () => {
  const [data, setData] = useState([]);
  const [name, setName] = useState('');
  const [value, setValue] = useState('');

  useEffect(() => {
    axios.get('http://localhost:5000/api/data').then(res => setData(res.data));
  }, []);

  const addData = () => {
    axios.post('http://localhost:5000/api/data', { name, value: Number(value) })
      .then(res => setData([...data, res.data]));
  };

  return (
    <div>
      <h2>Data</h2>
      <ul>
        {data.map(item => (
          <li key={item._id}>{item.name}: {item.value}</li>
        ))}
      </ul>
      <h2>Add Data</h2>
      <input placeholder="Name" value={name} onChange={e => setName(e.target.value)} />
      <input placeholder="Value" value={value} onChange={e => setValue(e.target.value)} />
      <button onClick={addData}>Add</button>
    </div>
  );
};

export default MongoDBIntegration;
```




## Frontend (React)

Create the React components and integrate them.

### App.js

javascript

 Copy code

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

const App = () => (
  <div className="App">
    <h1>MongoDB Integration Example</h1>
    <MongoDBIntegration />
  </div>
);

export default App;
```