Answer Key

Question 1: Write a parent component `Parent` and a child component `Child`. The parent should pass a function to the child that updates the parent's state. Demonstrate this with a message being sent from the child to the parent.

Answer:

- Parent:

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

function Parent() {
 const [message, setMessage] = useState('');

 const handleMessage = (newMessage) => {
  setMessage(newMessage);
 };

 return (
  <div>
   <h1>Parent Component</h1>
   <p>Message from Child: {message}</p>
   <Child onMessageSend={handleMessage} />
  </div>
 );
}

export default Parent;
```

- Child:

```
import React from 'react';

function Child({ onMessageSend }) {
 return (
  <div>
   <h2>Child Component</h2>
   <button onClick={() => onMessageSend('Hello from Child!')}>
    Send Message to Parent
   </button>
  </div>
 );
}

export default Child;
```

Question 3: Create a component `ItemList` that takes an array of items (strings) as props and renders each item in an unordered list. Include PropTypes validation for the array.

Answer:

- Index.js:

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

function ItemList({ items }) {
  return (
    <ul>
      {items.map((item, index) => (
        <li key={index}>{item}</li>
      ))}
    </ul>
  );
}

ItemList.propTypes = {
  items: PropTypes.arrayOf(PropTypes.string).isRequired
};

export default ItemList;
```

- App.js:

```
import React from 'react';
import ItemList from './ItemList'; // Adjust the import path based on your file structure

function App() {
  const myItems = ['Item 1', 'Item 2', 'Item 3'];

  return (
    <div>
      <h1>My Items</h1>
      <ItemList items={myItems} />
    </div>
  );
}

export default App;
```

Question 4: Build a React application that incorporates Material UI components for creating a contact list. Use Material UI's AppBar, Grid, Card, and Button components to display a list of contacts. Style these components using your custom CSS.

Answer:

- **ContactList.js:**

```
import React from 'react';
import { AppBar, Toolbar, Typography, Grid, Card, CardContent, Button } from
'@mui/material';

function ContactList() {
 const contacts = [
  { name: 'John Doe', email: 'johndoe@example.com' },
  { name: 'Jane Doe', email: 'janedoe@example.com' },
  // Add more contacts here
 ];

 return (
  <div>
   <AppBar position="static">
    <Toolbar>
     <Typography variant="h6">
      Contact List
     </Typography>
    </Toolbar>
   </AppBar>
   <Grid container spacing={2} style={{ padding: 20 }}>
    {contacts.map((contact, index) => (
     <Grid item xs={12} sm={6} md={4} key={index}>
      <Card style={{ height: '100%' }}>
       <CardContent>
        <Typography variant="h5">{contact.name}</Typography>
        <Typography variant="body1">{contact.email}</Typography>
        <Button variant="contained" color="primary" style={{ marginTop: 10 }}>
         Contact
        </Button>
       </CardContent>
      </Card>
     </Grid>
    ))}
   </Grid>
  </div>
 );
}

export default ContactList;
```

Question 5: Write a `SignupForm` component with two input fields for username and password and a submit button. On submission, log the username and password to the console without reloading the page.

Answer:

- SignUpForm.js:

```
import React, { Component } from 'react'
import { Button, Card, CardActions, CardContent, Grid, TextField } from '@mui/material';

export default class SimpleForm extends Component {
  constructor(props) {
    super(props);
    this.state = {
      fullName: '',
      fullNameError: false,
      fullNameHelperText: '',
      email: '',
      emailError: false,
      emailHelperText: '',
      password: '',
      passwordError: false,
      passwordHelperText: '',
      confirmPassword: '',
      confirmError: false,
      confirmHelperText: '',
    };
  }

  handleFullname = (e) => {
    this.setState({
      fullName: e.target.value,
      fullNameError: false,
      fullNameHelperText: ''
    });
  }

  handleEmail = (e) => {
    this.setState({
      email: e.target.value,
      emailError: false,
      emailHelperText: ''
    });
  }
```

```
handlePassword = (e) => {
  this.setState({
    password: e.target.value,
    passwordError: false,
    passwordHelperText: ''
  });
}

handleConfirm = (e) => {
  this.setState({
    confirmPassword: e.target.value,
    confirmError: false,
    confirmHelperText: ''
  });
}

handleSubmit = (e) => {
  e.preventDefault();
  if (this.state.fullName === '') {
    this.setState({
      fullNameError: true,
      fullNameHelperText: 'You must fill in your full name'
    });
  }

  if(this.state.email === ''){
    this.setState({
      emailError: true,
      emailHelperText: 'You must fill in your email address'
    });
  }
  if(this.state.password === ''){
    this.setState({
      passwordError: true,
      passwordHelperText: 'You must fill in your password'
    });
  }
  if(this.state.confirmPassword === ''){
    this.setState({
      confirmError: true,
      confirmHelperText: 'You must fill in your confirm password'
    });
  }
  if(this.state.password !== this.state.confirmPassword){
    this.setState({
      confirmError: true,
      confirmHelperText: 'Incorrect confirm password'
    });
```

```
    }
    if (this.state.fullName !== '' && this.state.emai !== '' && this.state.password !== '' &&
this.state.confirmPassword !== '' && this.state.password === this.state.confirmPassword) {
      alert('Form Submitted Successfully');
    }
  }

  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <Card variant="outlined">
          <CardContent>
            <Grid container spacing={1}>
              <Grid item md={6} xs={12}>
                <TextField
                  name="fullName"
                  value={this.state.fullName}
                  onChange={this.handleFullname}
                  variant="outlined"
                  label="Full Name"
                  type='text'
                  error={this.state.fullNameError}
                  helperText={this.state.fullNameHelperText}
                />
              </Grid>
              <Grid item md={6} xs={12}>
                <TextField
                  name="email"
                  value={this.state.email}
                  onChange={this.handleEmail}
                  variant="outlined"
                  label="Email"
                  type='email'
                  error={this.state.emailError}
                  helperText={this.state.emailHelperText}/>
              </Grid>
            </Grid>
            <Grid container spacing={1} sx={{ marginTop: '10px' }}>
              <Grid item md={6} xs={12}>
                <TextField
                  name="password"
                  value={this.state.password}
                  onChange={this.handlePassword}
                  variant="outlined"
                  label="Password"
                  type='password'
                  error={this.state.passwordError}
                  helperText={this.state.passwordHelperText}/>
```

```
                    </Grid>
                    <Grid item md={6} xs={12}>
                      <TextField
                        name="password"
                        value={this.state.confirmPassword}
                        onChange={this.handleConfirm}
                        variant="outlined"
                        label="Confirm Password"
                        type='password'
                        error={this.state.confirmError}
                        helperText={this.state.confirmHelperText}/>
                    </Grid>
                  </Grid>
                </CardContent>
                <CardActions>
                  <Button type="submit" size="small">Submit</Button>
                </CardActions>
              </Card>
            </form>
        )
    }
}
```

Question 6: Build a parent component that contains a list of items todo list items. Each item should be a child component. Pass the list of items as props to the child components and allow the child components to have a state to track whether the item is completed or not. Implement functionality to mark items as completed when clicked.

Answer:

- TodoItem.js:

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

class TodoItem extends Component {
  constructor(props) {
    super(props);
    this.state = {
      completed: false
    };
  }

  toggleCompleted = () => {
    this.setState(prevState => ({
      completed: !prevState.completed
```

```
    }));
  }

  render() {
    const { content } = this.props;
    const { completed } = this.state;

    return (
      <li
        style={{
          textDecoration: completed ? 'line-through' : 'none',
          cursor: 'pointer'
        }}
        onClick={this.toggleCompleted}
      >
        {content}
      </li>
    );
  }
}

export default TodoItem;
```

- TodoList.js:

```
import React from 'react';
import TodoItem from './TodoItem'; // Adjust the import path as necessary

function TodoList() {
  const items = ["Item 1", "Item 2", "Item 3"]; // Example to-do items

  return (
    <ul>
      {items.map((item, index) => (
        <TodoItem key={index} content={item} />
      ))}
    </ul>
  );
}

export default TodoList;
```

Note: Render Only the TodoList.js in index.js

Question 7: Implement a class component `Timer` that starts a timer using `setInterval` when the component mounts and clears the timer when the component unmounts. Display the elapsed time in seconds on the screen.

Answer:

- Timer.js:

```
import React from 'react';

class Timer extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      elapsedSeconds: 0
    };
  }

  componentDidMount() {
    this.timerID = setInterval(
      () => this.tick(),
      1000 // Update every second
    );
  }

  componentWillUnmount() {
    clearInterval(this.timerID);
  }

  tick() {
    this.setState(prevState => ({
      elapsedSeconds: prevState.elapsedSeconds + 1
    }));
  }

  render() {
    return (
      <div>
        <h2>Timer: {this.state.elapsedSeconds} seconds</h2>
      </div>
    );
  }
}

export default Timer;
```

Question 10: Write a React component that renders a list of at least 5 user names. Use `.map()` and ensure each list item has a unique `key` prop.

Answer:

- UserList.js

```
import React from 'react';

function UserList() {
  const users = ['Alice', 'Bob', 'Charlie', 'Diana', 'Edward']; // Array of user names

  return (
    <ul>
      {users.map((user, index) => (
        <li key={index}>{user}</li>
      ))}
    </ul>
  );
}

export default UserList;
```

Question 11: Demonstrate how to style a `UserProfile` component using an external CSS file. Include styles for at least 3 different CSS properties.

Answer:

- UserProfile.js:

```
import React from 'react';
import './UserProfile.css'; // Importing the CSS file

function UserProfile() {
  return (
    <div className="user-profile">
      <h2 className="user-name">John Doe</h2>
      <p className="user-description">Web Developer and Designer</p>
    </div>
  );
}

export default UserProfile;
```

- UserProfile.css:

```
.user-profile {
    border: 1px solid #ddd;
    padding: 20px;
    border-radius: 8px;
    margin: 50px;
```

```
}

.user-name {
  color: #333;
  font-size: 24px;
}

.user-description {
  color: #666;
  font-style: italic;
}
```

Question 12: Explain the concept of scoping styles in React and create a component with inline styles that only affect that particular component without affecting child components.

Answer:

- StyledComponent.js:

```
import React from 'react';

function StyledComponent() {
  // Define the inline styles
  const componentStyle = {
    backgroundColor: 'lightblue',
    padding: '20px',
    borderRadius: '8px'
  };

  const headingStyle = {
    color: 'navy',
    fontSize: '24px'
  };

  return (
    <div style={componentStyle}>
      <h2 style={headingStyle}>This is a Styled Component</h2>
      <p>This component's styles are scoped to it and do not affect child components.</p>
    </div>
  );
}
```

```
export default StyledComponent;
```

Question 14: Describe two limitations of using inline styles in React components and write a code snippet to illustrate one of the limitations.

Answer:

- ButtonComponent.js:

```
import React from 'react';

function ButtonComponent() {
  const buttonStyle = {
    backgroundColor: 'blue',
    color: 'white',
    padding: '10px 20px',
    border: 'none',
    borderRadius: '5px',
    cursor: 'pointer'
    // Unable to define :hover styles here
  };

  return (
    <button style={buttonStyle}>
      Hover Over Me
    </button>
  );
}

export default ButtonComponent;
```

Question 15: Develop a component that uses Material UI's Grid to layout a form with several checkboxes, each representing a day of the week.

Answer:

- DaysForm.js:

```
import React from 'react';
import { Grid, Checkbox, FormControlLabel } from '@mui/material';

function DaysForm() {
  const days = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday', 'Sunday'];

  return (
    <Grid container spacing={2}>
      {days.map((day, index) => (
        <Grid item xs={12} sm={6} md={4} key={index}>
```

```
            <FormControlLabel
              control={<Checkbox />}
              label={day}
            />
          </Grid>
        ))}
      </Grid>
    );
  }

  export default DaysForm;
```

Question 19: Write a React component that uses Material-UI to create an AppBar with a Toolbar. The AppBar should contain a logo and a navigation menu with at least three links.

Answer:

- Navbar.js:

```
import React from 'react';
import { AppBar, Toolbar, Typography, Button } from '@mui/material';

function Navbar() {
  return (
    <AppBar position="static">
      <Toolbar>
        <Typography variant="h6" style={{ flexGrow: 1 }}>
          Logo
        </Typography>
        <Button color="inherit">Link 1</Button>
        <Button color="inherit">Link 2</Button>
        <Button color="inherit">Link 3</Button>
      </Toolbar>
    </AppBar>
  );
}

export default Navbar;
```

- App.js:

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

```
import Navbar from './NavBar';

const App = () => {
 return (
  <div>
  <Navbar/> // rending navbar.js above button component
  <div style={{ textAlign: 'center', marginTop: '20px' }}>
   < Button />
  </div>
  </div>
 );
};

export default App;
```

Question 20: Create a `UserProfileCard` component using Material UI that includes a `Card` with user information and a `Button` to send a friend request.

Answer:

- UserProfileCard.js:

```
import React from 'react';
import { Card, CardContent, Typography, Button, Avatar } from '@mui/material';

function UserProfileCard() {
 // Dummy user data - replace with actual data as needed
 const user = {
  name: 'John Doe',
  bio: 'Software Developer',
  avatarUrl: '/path/to/avatar.jpg' // Replace with actual image path
 };

 return (
  <Card style={{ maxWidth: 345, margin: 'auto' }}>
   <CardContent>
    <Avatar
     src={user.avatarUrl}
     alt={user.name}
     style={{ width: 60, height: 60, margin: '10px auto' }}
    />
    <Typography gutterBottom variant="h5" component="div">
     {user.name}
    </Typography>
    <Typography variant="body2" color="text.secondary">
```

```
        {user.bio}
       </Typography>
       <Button variant="contained" color="primary" style={{ marginTop: 20 }}>
        Send Friend Request
       </Button>
      </CardContent>
    </Card>
  );
}

export default UserProfileCard;
```

Question 23: Write a React functional component named `Greeting` that accepts a `name` prop and displays a greeting message. Ensure it checks for the prop's type and requirement using PropTypes.

Answer:

- Greeting.js:

```
import React from 'react'

export default function Greeting(props) {
  return (
    <div><h1>Hello, {props.name}!</h1></div>
  )
}
```

- Index.js:

```
<Greeting name="John" />
```

Question 27: Create a class component `Counter` that has a state property `count` initialized to 0. Include a button in the render method that, when clicked, increments the count by 1.

Answer:

- Counter.js:

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

export default class Counter extends Component {
  constructor(props) {
    super(props);
    this.state = { count: 0 };
```

```
  }

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

  render() {
    return (
      <div style={{ textAlign: "center", margin: "20px" }}>
        <p style={{ fontSize: "20px" }}>Count: {this.state.count}</p>
        <button
          style={{
            backgroundColor: "#4CAF50",
            color: "white",
            padding: "10px 20px",
            border: "none",
            borderRadius: "4px",
            cursor: "pointer",
          }}
          onClick={this.increment}
        >
          Increment
        </button>
      </div>
    );
  }
}
```

Question 28: Explain the role of handler functions in React. Provide an example of a handler function that formats user input and checks validation in a form component.

Answer:

- EmailForm.js:

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

class EmailForm extends Component {
  constructor(props) {
    super(props);
    this.state = {
      email: '',
      error: ''
    };
  }

  handleEmailChange = (event) => {
    this.setState({ email: event.target.value });
```

```
  };

  validateEmail = (inputEmail) => {
    const emailRegex = /\S+@\S+\.\S+/;
    return emailRegex.test(inputEmail);
  };

  handleSubmit = (event) => {
    event.preventDefault();
    const { email } = this.state;
    if (this.validateEmail(email)) {
      console.log('Submitted Email:', email);
      this.setState({ error: '' });
      // Additional submit logic here
    } else {
      this.setState({ error: 'Please enter a valid email address.' });
    }
  };

  render() {
    const { email, error } = this.state;

    return (
      <form onSubmit={this.handleSubmit}>
        <label>
          Email:
          <input
            type="email"
            value={email}
            onChange={this.handleEmailChange}
          />
        </label>
        {error && <div style={{ color: 'red' }}>{error}</div>}
        <button type="submit">Submit</button>
      </form>
    );
  }
}

export default EmailForm;
```

Question 29: Explain the concepts of component composition and reusability in React. Provide a detailed example of how components can be designed for reuse, including props drilling and the use of higher order components or render props.

Answer:

- UserCard.js:

```
import React from 'react';

const UserCard = ({ user }) => {
  return (
    <div style={cardStyle}>
      <img src={user.image} alt={user.name} style={imageStyle} />
      <h2>{user.name}</h2>
      <p>{user.bio}</p>
    </div>
  );
};

const cardStyle = {
  textAlign: 'center',
  margin: '20px',
  padding: '20px',
  border: '1px solid #ddd',
  borderRadius: '8px',
  boxShadow: '0 4px 8px rgba(0, 0, 0, 0.1)'
};

const imageStyle = {
  width: '100px',
  height: '100px',
  borderRadius: '50%'
};

export default UserCard;
```

- Index.js:

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import './index.css';
import UserCard from './UserCard';

// Define the user data
const user = {
  name: 'John Doe',
  bio: 'Software Developer at XYZ Corp',
  image: 'https://via.placeholder.com/150' // Replace with actual image URL
};

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
  <React.StrictMode>
```

```
    <UserCard user={user} />
    <UserCard user={user} />
  </React.StrictMode>
);
```

Question 32: Provide an example of a React component handling a user event, such as a button click, using a method within the component. Include the event handler and explain the use of 'this' in the context of the handler.

Answer:

- ClickCounter.js

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

class ClickCounter extends Component {
  constructor(props) {
    super(props);
    this.state = {
      count: 0
    };
  }

  // Arrow function automatically binds 'this' to the component instance
  handleClick = () => {
    this.setState(prevState => ({
      count: prevState.count + 1
    }));
  }

  render() {
    return (
      <div>
        <button onClick={this.handleClick}>
          Click me
        </button>
        <p>Button Clicks: {this.state.count}</p>
      </div>
    );
  }
}

export default ClickCounter;
```

Question 33: Illustrate two-way data binding in React by providing a code snippet that shows how a form input's value can be controlled by the state, and how changes to the input can update the state.

Answer:

- TwoWayBindingComponent.js:

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

class TwoWayBindingComponent extends Component {
 constructor(props) {
  super(props);
  this.state = { inputValue: "" }; // Initialize the state
 }

 handleInputChange = (event) => {
  this.setState({ inputValue: event.target.value }); // Update state on change
 };

 render() {
  return (
   <div style={{ textAlign: "center", margin: "20px" }}>
    <textarea
     style={{
      width: "300px",
      height: "100px",
      padding: "10px",
      fontSize: "16px",
      border: "1px solid #ccc",
      borderRadius: "4px",
      marginBottom: "10px",
     }}
     value={this.state.inputValue}
     onChange={this.handleInputChange}
    />
    <p style={{ fontSize: "18px", color: "#333" }}>
     Current Value: {this.state.inputValue}
    </p>
   </div>
  );
 }
}

export default TwoWayBindingComponent;
```

Question 37: Create a React functional component named UserProfile that meets the following requirements:

- Use Material UI's Card component to display user information.
- The component should accept a user object as a prop, which contains firstName, lastName, email, and profilePicture properties.
- Use Material UI's Avatar to display the profilePicture and Typography to show the user's name and email.
- Include a Material UI Button that, when clicked, alerts the user's full name using a handler function.
- The component should be styled with Material UI's styling solution to space the elements within the card neatly.

Provide the code for this component, ensuring you include PropTypes to validate the user prop's shape and draw the output page. Demonstrate how you would import and use the necessary Material UI components and icons.

Answer:

- UserProfileTwo.js:

```
import React from 'react';
import PropTypes from 'prop-types';
import { Card, CardContent, Avatar, Typography, Button } from '@mui/material';

const UserProfile = ({ user }) => {
  const { firstName, lastName, email, profilePicture } = user;

  const handleButtonClick = () => {
    alert(`${firstName} ${lastName}`);
  };

  return (
    <Card sx={{ maxWidth: 345, m: 2 }}>
      <CardContent style={{ textAlign: 'center' }}>
        <Avatar
          src={profilePicture}
          alt={`${firstName} ${lastName}`}
          sx={{ width: 80, height: 80, mb: 2 }}
        />
        <Typography variant="h5">
          {firstName} {lastName}
        </Typography>
        <Typography color="text.secondary">
          {email}
        </Typography>
        <Button variant="contained" color="primary" onClick={handleButtonClick} sx={{ mt: 2 }}>
          Show Name
        </Button>
```

```
      </CardContent>
    </Card>
  );
};

UserProfile.propTypes = {
  user: PropTypes.shape({
    firstName: PropTypes.string.isRequired,
    lastName: PropTypes.string.isRequired,
    email: PropTypes.string.isRequired,
    profilePicture: PropTypes.string.isRequired
  }).isRequired
};

export default UserProfile;
```

- App.js:

```
import React from 'react';
import UserProfile from './UserProfileTwo';

function App() {
  const user = {
    firstName: 'John',
    lastName: 'Doe',
    email: 'johndoe@example.com',
    profilePicture: 'https://via.placeholder.com/150' // Replace with actual URL
  };

  return (
    <div>
      <UserProfile user={user} />
    </div>
  );
}

export default App;
```

Question 2: Explain the key requirements for setting up a React application. What tools and technologies are essential?

Answer:

Setting up a React application involves several key requirements and tools. The process can be broadly categorized into the following steps:

1. Installing Node.js and npm (Node Package Manager)

Node.js: A JavaScript runtime environment that lets you run JavaScript on the server-side.

npm: A package manager for JavaScript, comes bundled with Node.js. It manages dependencies for an application.

Command: You can download Node.js from nodejs.org. npm is included in the Node.js installation.

2. Creating a React Application using Create React App (CRA)
Create React App (CRA): A boilerplate or starter kit provided by the React team to set up a new React application with sensible defaults.

Command: Once Node.js and npm are installed, you can create a new React application by running:

npx create-react-app my-app

Here, npx is a package runner tool that comes with npm 5.2+.
my-app is the name of your new application. You can choose any name for your project.
This command sets up the project with all the necessary configurations and dependencies.

3. Project Structure
Folders and Files: CRA creates a project with a specific structure. Key elements include:
src directory: Where you write your application's JavaScript and CSS.
public directory: Contains assets such as index.html, images, etc.
package.json: Lists dependencies and available scripts.
node_modules: Directory where npm modules are installed.

4. Running the Development Server
Development Server: CRA includes a built-in development server that you can use to run your application.

Command: Inside your project directory, run:

npm start

This command starts the development server. Your application will be available at http://localhost:3000 in the browser.
The development server provides features like live-reloading, which automatically refreshes your app as you make changes to the code.

5. Essential Tools and Technologies
React and ReactDOM: Core libraries for building the application.
Babel: A JavaScript compiler that converts modern JavaScript code into a format that can be run in older browsers.
Webpack: A module bundler that bundles JavaScript and other assets (like CSS and images) for the browser.
ESLint: A linter tool for identifying and reporting on patterns in JavaScript, helping you write cleaner code.

6. Editor/IDE
Code Editor: A text editor or an Integrated Development Environment (IDE) for writing code. Popular choices include Visual Studio Code, Atom, and Sublime Text.

Question 8: What is the event loop in JavaScript, and how does it relate to React performance and responsiveness?

Answer:
The event loop in JavaScript is a fundamental concept that plays a crucial role in the execution of JavaScript code, particularly in environments like web browsers. Understanding the event loop is essential for grasping how asynchronous operations are handled in JavaScript, which directly impacts the performance and responsiveness of applications, including those built with React.

Basics of the Event Loop:

Single-Threaded Nature: JavaScript is single-threaded, meaning it can execute only one command at a time. This characteristic could lead to performance issues if not for the event loop.

Call Stack: When a JavaScript function is executed, it's placed on the call stack. Functions are executed from the stack in a last-in, first-out order.

Event Loop Mechanism: The event loop continually checks the call stack to see if there's any function that needs to be run. If the stack is empty, it looks to the queue.

Callback Queue: Asynchronous callbacks (like events, HTTP requests) are queued up in the callback queue waiting to be executed.

Execution Order: Once the call stack is empty, the event loop transfers the first callback in the queue to the call stack to be executed.

Non-Blocking: This mechanism ensures that JavaScript can perform long tasks without blocking the main thread, as the event loop handles the execution of callbacks asynchronously.

Relationship with React Performance and Responsiveness:

Asynchronous Operations: React often deals with asynchronous operations like API calls or data fetching. The event loop enables these operations to be handled efficiently without blocking UI updates.

State Updates and Rendering: React batches state updates and re-renders components asynchronously. The event loop plays a role here, as it schedules these updates, enhancing performance and ensuring a smooth user experience.

Handling User Interactions: React’s event system is designed to be non-blocking and uses asynchronous callbacks. The event loop helps in managing these callbacks, ensuring that user interactions are handled promptly.

Optimization: Knowing how the event loop works, developers can optimize their React applications by avoiding long-running operations on the main thread, which can make the app unresponsive.

Concurrent Mode (Advanced): React's Concurrent Mode takes advantage of the event loop to interrupt and prioritize updates, allowing React to work on multiple tasks in a non-blocking way. This results in better handling of user inputs, animations, and other interactions, making applications more responsive.

Question 9: Discuss the fundamentals of Node.js and its significance in building React applications.

Answer:
Node.js is a significant technology in the development of modern web applications, particularly in the ecosystem of React. Understanding its fundamentals and role in building React applications is crucial for developers.

Fundamentals of Node.js:

JavaScript Runtime: Node.js is a runtime environment that allows you to execute JavaScript code on the server side, outside a web browser. It's built on the V8 JavaScript engine used in Google Chrome.

Non-Blocking I/O Model: Node.js operates on a non-blocking, event-driven architecture, making it efficient for building scalable network applications. It can handle many concurrent connections without incurring the cost of thread context switching.

Single-Threaded: Despite being single-threaded, Node.js can handle high concurrency through its event-driven architecture and asynchronous APIs.

npm (Node Package Manager): Node.js comes with npm, a package manager that gives access to a vast repository of libraries and tools. npm enhances Node.js's capabilities, allowing for easy integration of third-party solutions.

Modules and Packages: Node.js uses a module system that allows you to organize your code into reusable components. npm packages can be easily integrated into Node.js applications.

Significance in Building React Applications:

Environment Consistency: Node.js enables JavaScript to be used on both the client and server sides. This consistency is beneficial for React developers, as they can work with a single programming language across the entire stack.

Development Tools: Many tools essential for React development, such as Create React App, Webpack, and Babel, run on Node.js. These tools help in transpiling JSX and modern JavaScript, bundling assets, and managing development servers.

Build Process: Node.js is integral to the build process of React applications. It runs the build scripts, compiles the JSX into JavaScript, and optimizes the application for production deployment.

Server-Side Rendering (SSR): For React applications, Node.js can be used to render components on the server side. This SSR improves performance, especially for content-heavy applications, and aids in SEO.

API Development: Node.js can be used to build backend APIs for React applications. This is particularly useful in full-stack development, where the same language (JavaScript) is used across both frontend and backend.

Package Management: npm (or Yarn, an alternative to npm) is used to manage libraries and dependencies in React projects. This simplifies the process of adding, updating, and removing packages.

Community and Ecosystem: Node.js has a large and active community. This community support translates to a wide array of libraries and tools, making the development process more efficient and less challenging.

Question 13: Explain the concept of React DOM. How does it enable interaction between React components and the web browser?

Answer:
The concept of React DOM is central to how React applications interact with web browsers. It acts as the bridge between React components and the browser's Document Object Model (DOM), enabling dynamic and efficient updates to the user interface.

Basics of React DOM:

What is React DOM?: React DOM is a library in the React ecosystem specifically designed for web browsers. It provides methods to render components into the DOM and to interact with it.

Virtual DOM: React DOM utilizes a concept known as the Virtual DOM. The Virtual DOM is a lightweight copy of the actual DOM. It is a JavaScript representation of the DOM tree.

Efficient Updates: When a React component's state changes, React first updates the Virtual DOM. Then, React DOM compares the updated Virtual DOM with a snapshot of the Virtual DOM taken before the update. This process is known as "diffing."

Reconciliation: Through the diffing algorithm, React DOM identifies the exact changes made to the Virtual DOM. It then updates only those parts of the actual DOM that have changed. This selective rendering makes the process highly efficient.

Batching: React DOM batches multiple updates into a single update cycle. This reduces the number of re-rendering cycles and improves performance.

Event Handling: React DOM also handles events in a cross-browser compatible way. It provides a synthetic event system that wraps the browser's native event system, ensuring consistency across different browsers.

Interaction Between React Components and Web Browser:

Rendering Components: React DOM provides the ReactDOM.render() method, which is used to render a React component into a specified DOM node. This is typically used in the root component of an application.

Manipulating the DOM: React components define their UI declaratively. When their state changes, React automatically updates the DOM to match the specified UI. Developers do not directly manipulate the DOM; instead, they work with component states and props.

Event Handling: React components can define event handlers, such as onClick or onChange. React DOM ensures these handlers work consistently across all browsers.

Server-Side Rendering (SSR): React DOM can be used in server-side environments to pre-render initial HTML. This improves the performance of React applications and helps with SEO.

Accessibility: React DOM manages focus and offers various features to build accessible web applications, following WAI-ARIA guidelines.

Question 16: Explain the essential NPM commands used in managing React projects. Provide examples of when and how these commands are useful.

Answer:
NPM (Node Package Manager) is a fundamental tool in managing React projects. It streamlines the process of working with JavaScript packages, allowing developers to easily manage dependencies, scripts, and more. Below are the essential NPM commands commonly used in React projects:

1. Initializing a New Project
- **Command**: **npm init** or **npm init -y**
- **Usage**: Initializes a new Node.js project. It creates a **package.json** file in your project directory. The **-y** flag generates a default **package.json** without asking questions.

2. Installing Packages
- **Command**: **npm install <package-name>**
- **Usage**: Installs a package and adds it to the **dependencies** in **package.json**. For example, **npm install react** installs React in your project.

3. Installing Development Dependencies
- **Command**: **npm install <package-name> --save-dev**
- **Usage**: Installs a package as a development dependency. Development dependencies are used in the development process but not in the production build. For instance, **npm install @babel/core --save-dev** for Babel, a JavaScript compiler.

4. Installing Packages Globally
- **Command**: **npm install -g <package-name>**
- **Usage**: Installs a package globally on your machine. Useful for packages that provide command-line utilities (e.g., **create-react-app**).

5. Creating a React Application
- **Command**: **npx create-react-app my-app**

- **Usage**: Uses **npx** to run **create-react-app** without installing it. It creates a new React application with the name **my-app**.

6. Running Scripts

- **Command**: **npm run <script>**
- **Usage**: Executes a script defined in the **package.json** file. Common scripts include **start** for launching the development server, **build** for creating a production build, and **test** for running tests.

7. Updating Packages

- **Command**: **npm update <package-name>**
- **Usage**: Updates a package to its latest version based on the specified version range in **package.json**.

8. Uninstalling Packages

- **Command**: **npm uninstall <package-name>**
- **Usage**: Removes a package from the **node_modules** directory and your project's **package.json**.

9. Listing Installed Packages

- **Command**: **npm list**
- **Usage**: Lists all installed packages and their dependencies for the current project.

10. Checking for Vulnerable Dependencies

- **Command**: **npm audit**
- **Usage**: Scans your project for vulnerable dependencies and suggests or automatically applies fixes.

Question 18: Compare and contrast functional (stateless) components with class (stateful) components in React. When should you use one over the other?

Answer:

In React, components can be created using either functional or class syntax. Each approach has its own characteristics and use cases, and understanding the differences is key for effective React development.

Functional (Stateless) Components

Characteristics:

1. **Simplicity**: Functional components are plain JavaScript functions that take props as an argument and return React elements.
2. **Stateless**: Traditionally, they were stateless, meaning they didn't manage state or lifecycle methods. However, with the introduction of hooks in React 16.8, functional components can now use state and other features like lifecycle methods.
3. **Hooks**: Functional components use hooks (like **useState**, **useEffect**) for state management and side effects, making them more powerful.
4. **Reusability and Composition**: They are typically easier to read and test due to their straightforward nature and are great for simple UI components.

Usage:

- Use functional components for most components, especially when you need simpler or stateless components.

- After React 16.8, with hooks, you can use functional components even for stateful logic and side effects.

Class (Stateful) Components
Characteristics:

1. **State Management**: Class components can hold and manage local state and handle lifecycle events using methods like **componentDidMount** and **componentDidUpdate**.
2. **Lifecycle Methods**: They provide more explicit control over the component's lifecycle.
3. **Verbose**: They tend to be more verbose than functional components.
4. **this Keyword**: The use of **this** can make them slightly complex, especially for beginners.

Usage:

- Use class components if you prefer the class-style syntax or are working on a project that was started before hooks were introduced.
- They are also useful when you need to make use of certain lifecycle methods that aren't directly replicable with hooks (like **shouldComponentUpdate**).

When to Use One Over the Other

1. **Before React 16.8**: Functional components were mostly used for presentational purposes, while class components were used for stateful logic and lifecycle events.
2. **After React 16.8**: With the introduction of hooks, functional components have become capable of handling state and lifecycle features. The choice often comes down to personal or team preference, and the specific needs of the project.
3. **Legacy Codebases**: In older projects, you might see more class components, and it might make sense to continue using them for consistency.
4. **Readability and Simplicity**: For simple components, functional components are often more readable and concise. For complex logic, some developers prefer class syntax.

Question 21: Explain the core requirements of a React application and detail the setup process, including the role of Node.js and NPM in steps.

Answer:

Setting up a React application involves several core steps and requirements, with Node.js and npm (Node Package Manager) playing pivotal roles in the process. Here's a basic overview of the setup process and the functions of Node.js and npm in each step:

1. Installation of Node.js and npm

- **Node.js**: A JavaScript runtime that allows you to run JavaScript on the server side.
- **npm**: A package manager for JavaScript that comes with Node.js.
- **Purpose**: Node.js provides the runtime environment for running JavaScript outside the browser, which is essential for various development tools used in React applications. npm is used to manage the packages and dependencies of the project.
- **Installation**: Download and install Node.js from the official website. npm is included in the installation.
-

2. Setting Up a New React Project

- **Create React App (CRA)**: A popular tool to set up a new React application. It sets up the build system (Webpack, Babel) and a development server without requiring manual configuration.

- **Command**: To create a new React project, use the command:

npx create-react-app my-app

npx comes with npm 5.2+ and is used to execute packages. **my-app** is the name of your new application.

3. Understanding the Project Structure

- **Key Directories and Files**:
  - **src**: Contains the source code (JavaScript, CSS, etc.).
  - **public**: Contains static assets like **index.html**.
  - **package.json**: Lists project dependencies and available scripts.
  - **node_modules**: Contains all installed npm packages.

4. Running the Development Server

- **Development Server**: A built-in server provided by CRA for testing the application during development.
- **Command**: Inside the project directory, run:

npm start

This starts the development server, typically available at **http://localhost:3000**.

5. Role of Node.js and npm

- **Node.js**: Provides the environment to run React and its build tools. Tools like Babel (for JSX and ES6+ support) and Webpack (for bundling) run on Node.js.
- **npm**: Manages the dependencies of the project. It allows you to install, update, and remove packages.
-

6. Additional Tools and Dependencies

- **React and ReactDOM**: The core libraries for building the application.
- **Babel**: Transpiles JSX and ES6+ JavaScript into a format compatible with most browsers.
- **Webpack**: Bundles JavaScript, CSS, and images into a few files optimized for browsers.

7. Adding Additional Packages

- **Using npm**: You can add additional packages (like React Router, Redux, Axios, etc.) using npm commands, for example:

npm install react-router-dom

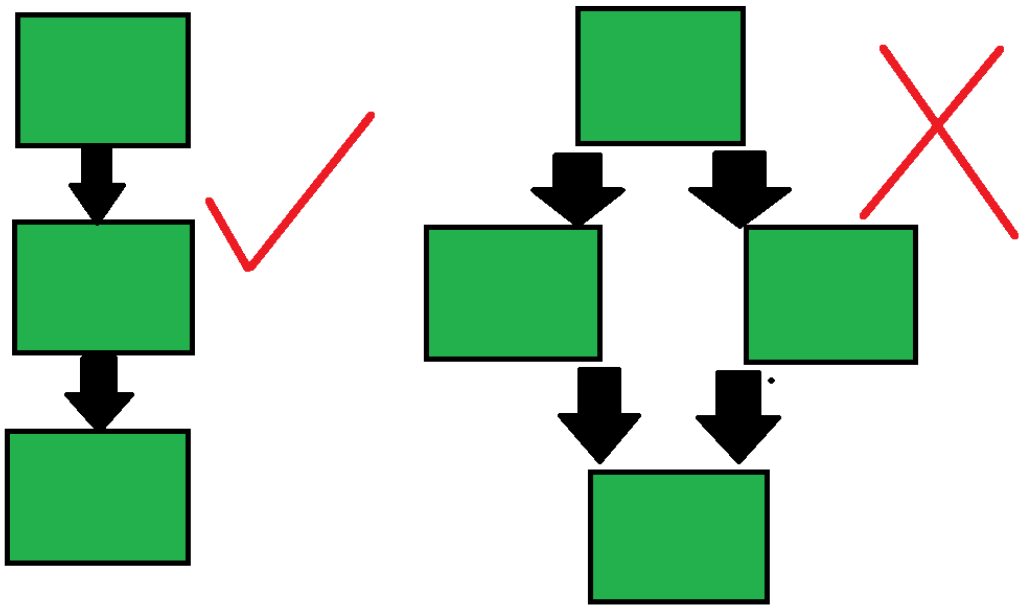
Question 22: Describe the event loop in JavaScript. Illustrate with a diagram and explain how it impacts the performance of a React application.

Answer:

JavaScript is a single-threaded asynchronous programming language. But what does it mean? What is this event loop in JavaScript that we all keep talking about?

What does it mean when we say JavaScript is single-threaded?

It means that the main thread where JavaScript code is run, runs in one line at a time manner and there is no possibility of running code in parallel.



Example:

- Javascript

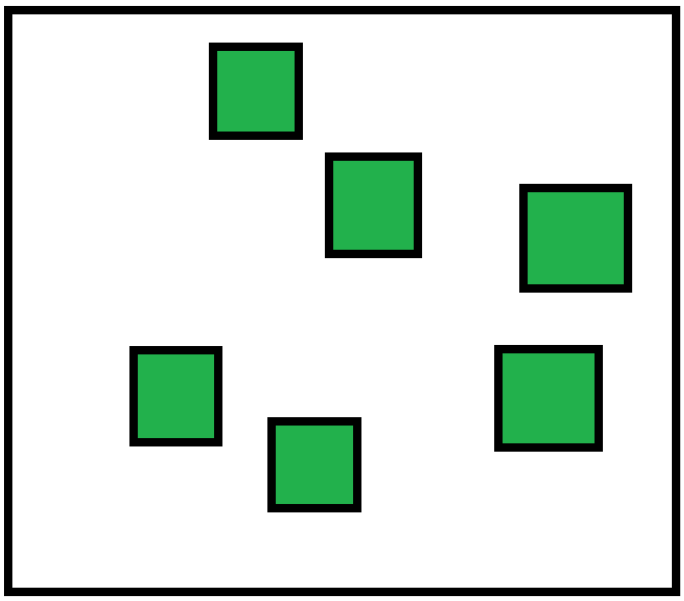
```
console.log("Before delay");

function delayBySeconds(sec) {
  let start = now = Date.now()
  while(now-start < (sec*1000)) {
    now = Date.now();
  }
}

delayBySeconds(5);

// Executes after delay of 5 seconds
console.log("After delay");
```

Output:

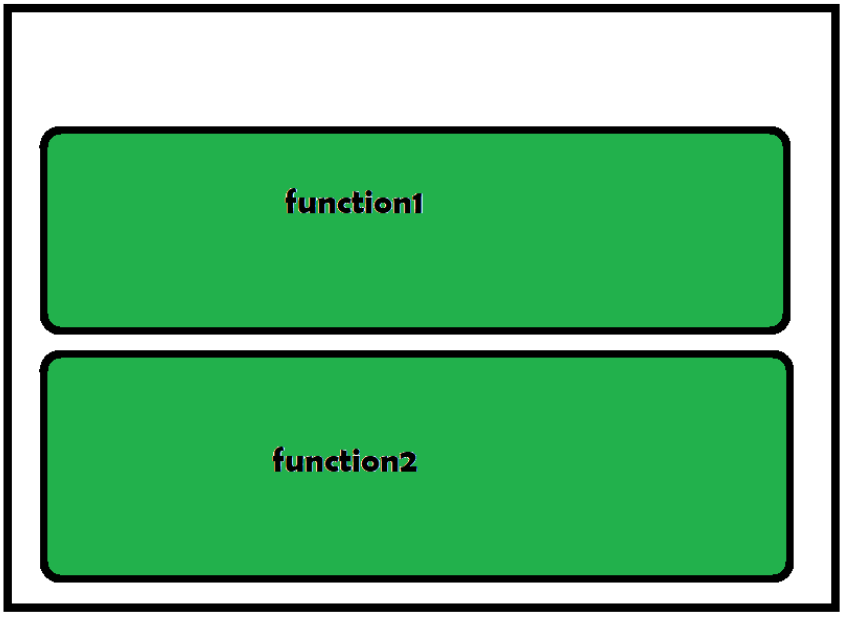
Before delay
(... waits for 5 seconds)
After delay

Memory allocation in JavaScript:

1) Heap memory: Data stored randomly and memory allocated.



2) Stack memory: Memory allocated in the form of stacks. Mainly used for functions.

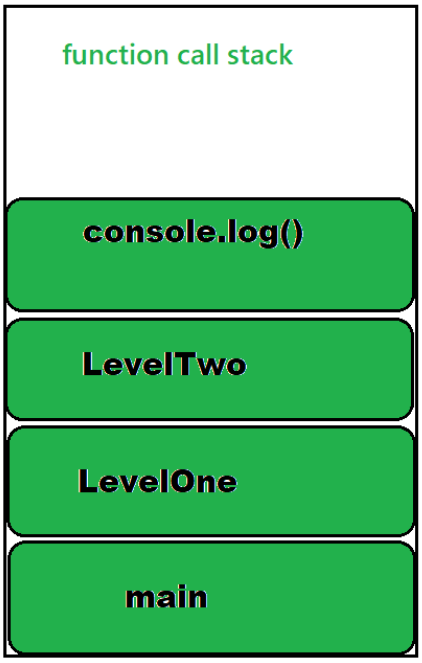


Function call stack: The function stack is a function that keeps track of all other functions executed in run time. Ever seen a stack trace being printed when you ran into an error in JavaScript? That is nothing but a snapshot of the function stack at that point when the error occurred.
Example:

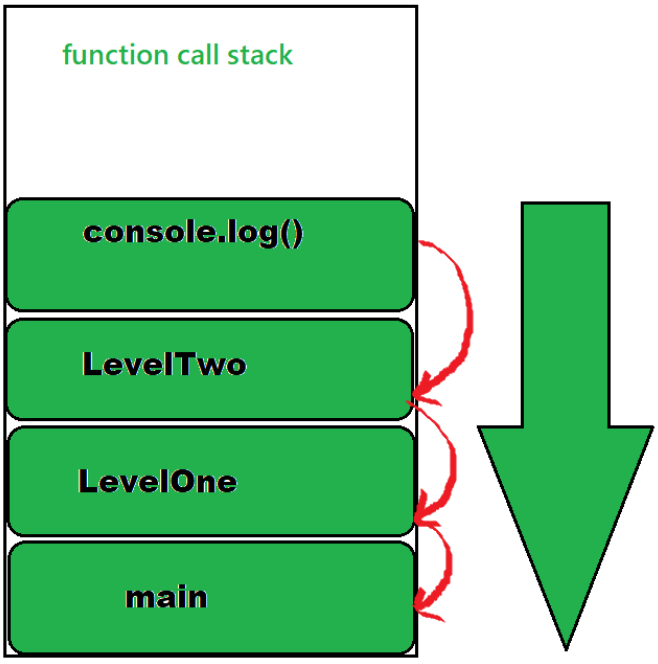
- Javascript

```
function LevelTwo() {
  console.log("Inside Level Two!")
}

function LevelOne() {
```

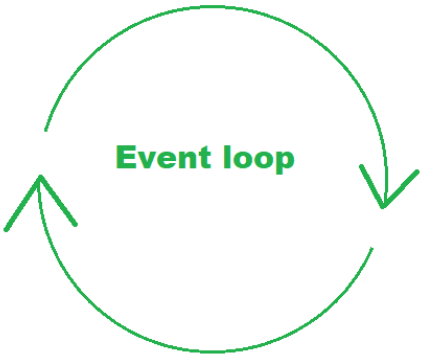
```
  LevelTwo()
}

function main() {
  LevelOne()
}

main()
```



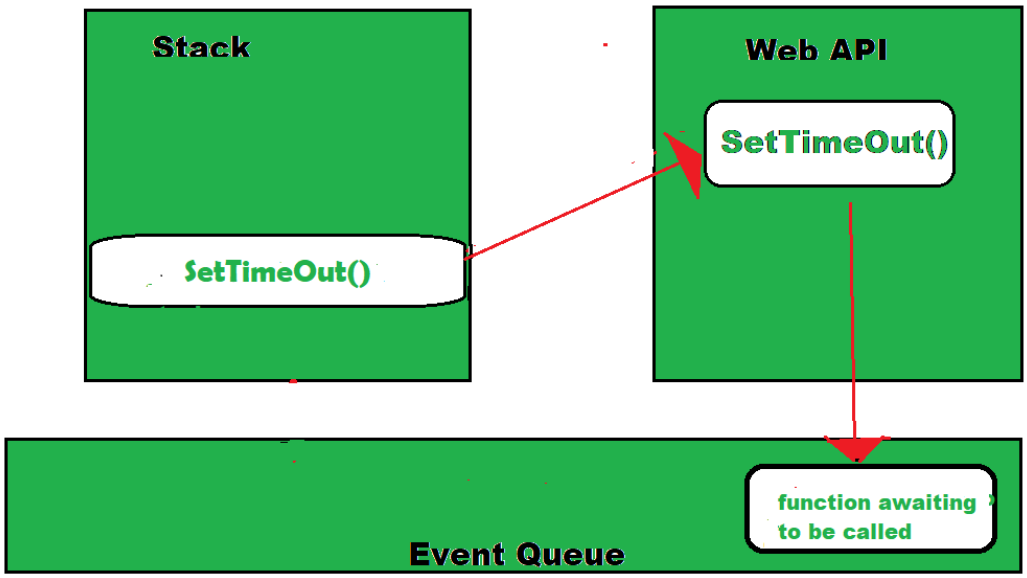
Order at which functions get executed i.e get popped out of the stack after a function's purpose gets over as shown below:



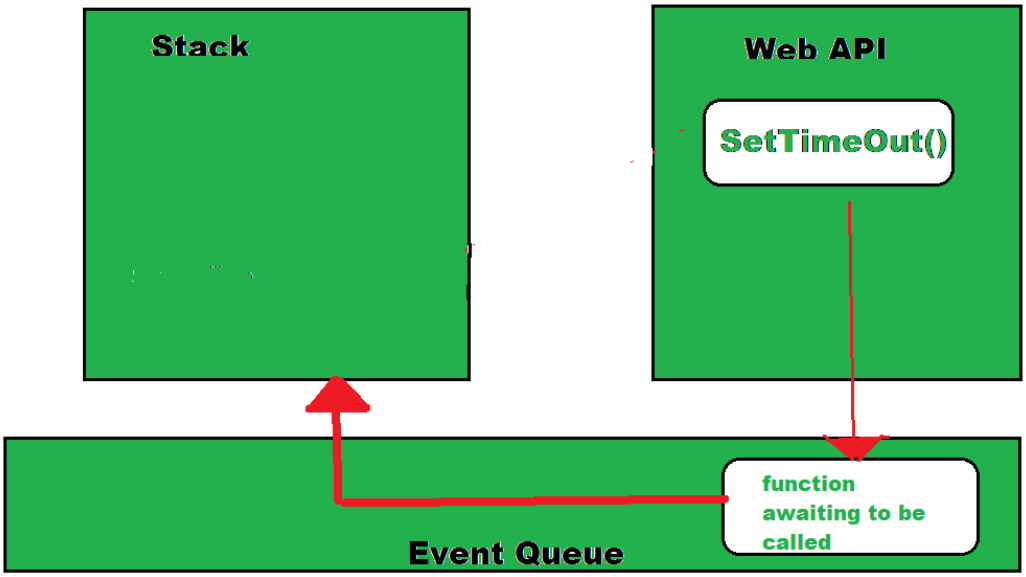
Event loop: An event loop is something that pulls stuff out of the queue and places it onto the function execution stack whenever the function stack becomes empty.



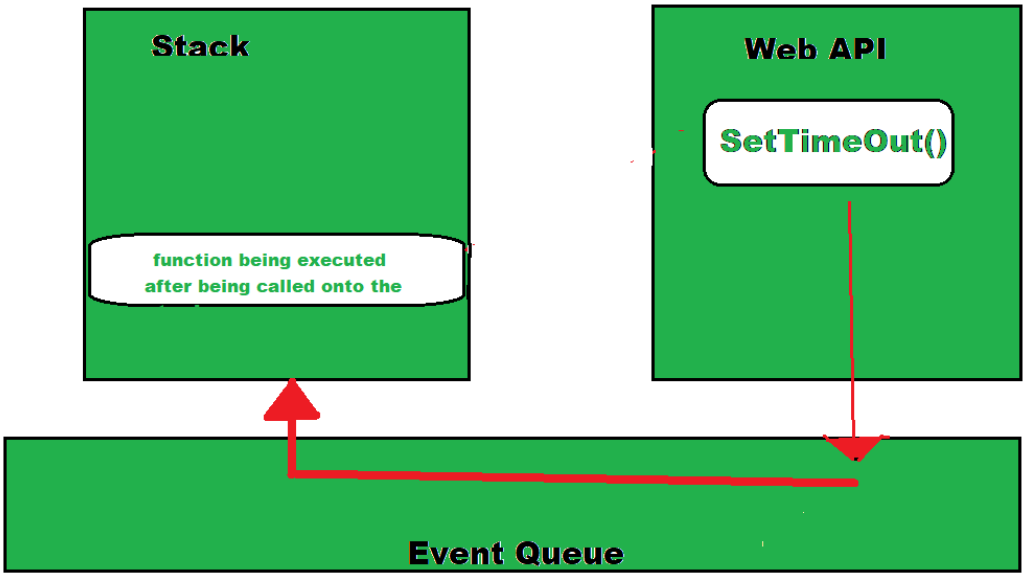
The event loop is the secret by which JavaScript gives us an illusion of being multithreaded even though it is single-threaded. The below illusion demonstrates the functioning of the event loop well:



Here the callback function in the event queue has not yet run and is waiting for its time into the stack when the SetTimeOut() is being executed and the Web API is making the mentioned wait. When the function stack becomes empty, the function gets loaded onto the stack as shown below:



That is where the event loop comes into the picture, it takes the first event from the Event Queue and places it onto the stack i.e. in this case the callback function. From here, this function executes calling other functions inside it, if any.



This cycle is called the **event loop** and this is how JavaScript manages its events.

Question 24: Provide a detailed explanation of JSX, its purpose in React, and how it differs from regular HTML. Include a code snippet to demonstrate its syntax.

Answer:

JSX (JavaScript XML) is a syntax extension for JavaScript widely used with React to describe the UI of a web application. It resembles HTML in its appearance and usage but comes with its own distinct features and benefits.

Purpose of JSX in React

1. **Descriptive UI Structure**: JSX provides a concise and readable way to describe the layout of the UI in a way that resembles HTML. It enhances the development experience by making the structure of the UI clear and visually intuitive.
2. **Enhanced JavaScript Power**: Unlike regular HTML, JSX is integrated with JavaScript, which means you can use JavaScript expressions within JSX. This enables dynamic content rendering and powerful UI manipulation.
3. **Component Structure**: JSX is used in React to define the structure of components. It represents the component's output and how the UI should appear.

Differences from Regular HTML

1. **JavaScript Integration**: JSX allows JavaScript expressions to be written inside curly braces **{}**. This integration lets developers dynamically render content and attributes.
2. **Syntax Variations**: Some HTML attributes have different names in JSX due to JavaScript naming restrictions. For example, **class** becomes **className** in JSX, and **tabindex** becomes **tabIndex**.
3. **Single Root Node**: Every JSX element must be wrapped in a single parent element or a fragment. This is because JSX ultimately compiles to JavaScript functions that return one element.
4. **Self-Closing Tags**: Any JSX element can be self-closed, similar to XML, which is not always the case in HTML.

Code Snippet:

```
import React from 'react';

function WelcomeMessage() {
  const name = 'Alice';
  return (
    <div className="welcome-message">
      <h1>Hello, {name}!</h1>
      <p>Welcome to our website.</p>
    </div>
  );
}

export default WelcomeMessage;
```

In this example:

- The JSX in the **WelcomeMessage** component looks similar to HTML but includes curly braces **{}** containing a JavaScript expression (**{name}**).
- **className** is used instead of **class**.
- The content is wrapped in a single **div** to maintain a single root node.

Question 25: Create a guide on using NPM commands to manage a React project, including initializing a new project, installing packages, and managing package versions.

Answer:

Managing a React project efficiently involves understanding various NPM (Node Package Manager) commands. NPM is an integral tool in handling dependencies, scripts, and more in a React project. Here's a guide on using NPM commands for various tasks in a React project:

1. Initializing a New Project

- **Creating a React App**:
    - If you're starting from scratch, the easiest way to create a new React project is by using **Create React App** (CRA).
    - **Command**:

npx create-react-app my-app

    - This command sets up a new React project named **my-app** with a sensible default configuration.

2. Installing Packages

- **Adding Dependencies**:
    - You can add new packages (dependencies) to your project using NPM.
    - **Command**:

npm install <package-name>

    - For example, **npm install axios** would add the **axios** HTTP client to your project.
- **Development Dependencies**:
    - Some packages are only needed during development (e.g., testing frameworks).
    - **Command**:

npm install <package-name> --save-dev

    - This installs the package as a development dependency.

3. Updating Packages

- **Updating a Single Package**:
    - To update a specific package to its latest version, use:

npm update <package-name>

    - This command will update the package to the latest version allowed by the version range specified in **package.json**.
- **Updating All Packages**:
    - To update all packages to their latest permissible versions, run:

npm update

4. Removing Packages

- **Uninstalling a Package**:
    - If a package is no longer needed, you can remove it with:

npm uninstall <package-name>

5. Managing Package Versions

- **Checking Installed Versions**:
    - To see the installed version of a package, use:

npm list <package-name>

    - For a list of all installed packages and their versions, just run **npm list**.
- **Specifying Versions**:
    - When installing a package, you can specify a version:

npm install <package-name>@<version>

    - For example, **npm install react@16.13.1** will install that specific version of React.

6. Using package.json

- **Dependency Management**:
    - **package.json** is a key file in your project that keeps track of dependencies, scripts, and project metadata.
    - When you install packages, they are automatically added to this file under **dependencies** or **devDependencies**.
- **Version Ranges**:
    - In **package.json**, you can specify version ranges for your dependencies. This allows for flexibility while ensuring compatibility.
    - Symbols like **^** and **~** are used to accept minor and patch updates, respectively.

Question 26: Create a step-by-step guide on how to build a "Hello World" application in React, including the essential files and their contents in brief.

Answer:

Creating a "Hello World" application in React is a great way to get started with React development. Here's a step-by-step guide to building this basic application:

Step 1: Setting Up the Environment

1. **Install Node.js and npm**:
    - Ensure Node.js and npm (Node Package Manager) are installed on your system. They are essential for React development.
    - You can download Node.js from nodejs.org, which includes npm.

Step 2: Creating a New React Application

2. **Create a New React App**:
    - Use the Create React App (CRA) tool to set up a new React project. CRA provides a ready-to-use setup.
    - Run the following command in your terminal:

npx create-react-app hello-world

    - This command creates a new directory named **hello-world** with all the necessary files and configurations.

Step 3: Understanding the Project Structure

3. **Project Structure**:

- **src** folder: Contains your JavaScript and CSS files.
- **public** folder: Contains the **index.html** file and other public assets.
- **package.json**: Lists dependencies and scripts for your project.
- **node_modules**: Contains all the npm packages.

Step 4: Writing the "Hello World" Code

4. **Modify src/App.js**:
    - Open the **hello-world** project in your code editor.
    - Navigate to **src/App.js**.
    - Replace the contents of **App.js** with the following code:

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

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

    - This code creates a simple functional component that returns a "Hello World" heading.

5. **Modify src/index.js** (if needed):
    - Ensure **index.js** is rendering the **App** component. This file mounts your React application to the DOM.
    - It should look something like this:

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

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

5: Running the Application

6. **Start the Development Server**:
    - In your terminal, navigate to the **hello-world** directory.
    - Run the command:

npm start

- This starts the development server and opens your new React application in the default web browser.
- The application will be available at **http://localhost:3000**.

Step 6: Viewing the Output

7. **View Your Application**:
    - The browser should display the "Hello World" message on the screen.
    - Any changes you make to **App.js** will be reflected in the browser almost instantly, thanks to the hot reload feature.

Question 30: Illustrate the difference between functional and class components with examples.

Answer:

In React, components can be defined either as class components or functional components. Each type has its unique syntax and features, and understanding these differences is crucial in React development.

Functional Components

Functional components are JavaScript functions that return React elements. They are simpler and more concise compared to class components.

Characteristics:

1. **Stateless**: Initially, they were stateless, meaning they did not manage state or lifecycle methods. However, with the introduction of hooks in React 16.8, this has changed.
2. **Hooks**: Hooks allow functional components to use state and other React features like lifecycle methods.
3. **Simplicity**: They are generally easier to write and understand, especially for smaller components or components without complex logic.

Example:

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

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

  return (
    <div>
      <h1>Hello, {name}!</h1>
      <button onClick={() => setName('Bob')}>Change Name</button>
    </div>
  );
}
```

In this example, **GreetingFunctional** is a functional component that uses the **useState** hook to manage state.

Class Components

Class components are more traditional in React. They are ES6 classes that extend **React.Component** and typically manage state and lifecycle methods.

Characteristics:

1. **Stateful**: They can hold and manage internal state.
2. **Lifecycle Methods**: They provide lifecycle methods like **componentDidMount**, **componentDidUpdate**, etc.
3. **Verbose**: They are more verbose than functional components but can be more readable for complex logic.

Example:

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

class GreetingClass extends Component {
  constructor(props) {
    super(props);
    this.state = { name: 'Alice' };
  }

  changeName = () => {
    this.setState({ name: 'Bob' });
  }

  render() {
    return (
      <div>
        <h1>Hello, {this.state.name}!</h1>
        <button onClick={this.changeName}>Change Name</button>
      </div>
    );
  }
}
```

In this example, **GreetingClass** is a class component. It uses the **constructor** to set the initial state and defines a method **changeName** to update the state.

Question 31: Discuss the difference between props and state in React components, including how they are used within a component's lifecycle. Provide code examples for each.

Answer:

In React, understanding the distinction between props and state is fundamental for creating interactive UIs. Both props and state are plain JavaScript objects, but they serve different purposes and adhere to different rules within a component's lifecycle.

Props
Definition: Props (short for properties) are read-only and immutable data passed from a parent component to a child component. They are used to pass data down the component tree.
Characteristics:

1. **Read-Only**: Props are immutable, meaning they cannot be modified by the component that receives them.
2. **Data Flow**: They enable unidirectional data flow, making the data structure easier to understand and debug.
3. **Use Case**: Typically used for rendering dynamic data and for configuring child components.

Example:

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

// Usage
<Greeting name="Alice" />
```

In this example, **Greeting** is a functional component that receives **name** as a prop and uses it to display a greeting message.

State
Definition: State is a mutable data structure that is local to the component and cannot be accessed or modified outside of it. It's used to store data that changes over time.
Characteristics:

1. **Mutable**: State can be changed using the **setState** method in class components or the state setter function in functional components.
2. **Local**: State is local to the component where it's declared.
3. **Lifecycle**: State changes can trigger re-renders, updating the UI with new data.

Example:

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

  incrementCount = () => {
```

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

  render() {
    return (
      <div>
        <p>Count: {this.state.count}</p>
        <button onClick={this.incrementCount}>Increment</button>
      </div>
    );
  }
}
```

In this class component example, **Counter** manages its own state, **count**, which is incremented when the button is clicked.

Difference in Lifecycle

- **Props**: Passed from a parent component, props are set before a component mounts and can change when the parent component re-renders. Components receive new props and can re-render accordingly.
- **State**: Initialized when a component mounts. State changes typically happen in response to event handlers or lifecycle methods. A state change causes the component to re-render.

Question 38: Discuss the different methods of styling React components, including CSS Styling and inline styles. Provide examples and best practices for each.

Answer:

Styling in React is an essential aspect of building visually appealing applications. There are several methods to style React components, each with its own advantages and use cases. Here, we'll discuss two popular methods: CSS Styling and Inline Styles.

CSS Styling

CSS Styling involves using external **.css** files or CSS modules to style React components.

External CSS:

- **Usage**: Create a **.css** file and import it into your React component file.
- **Best Practices**:
    - Keep styles organized by component.
    - Use naming conventions like BEM (Block Element Modifier) to avoid naming conflicts.
- **Example**:

```
/* App.css */
.app {
```

```
  text-align: center;
}
// App.js
import React from 'react';
import './App.css';

function App() {
  return <div className="app">Hello World</div>;
}
```

CSS Modules:

- **Usage**: Similar to external CSS, but with CSS Modules, class names are locally scoped.
- **Best Practices**:
  - Use camelCase for class names as they are imported as JavaScript objects.
- **Example**:

```
/* App.module.css */
.container {
  text-align: center;
}
// App.js
import React from 'react';
import styles from './App.module.css';

function App() {
  return <div className={styles.container}>Hello World</div>;
}
```

Inline Styles

Inline styles involve defining styles directly within the component as an object.

Usage:

- Styles are defined as objects with camelCased properties.
- **Best Practices**:
  - Suitable for dynamic styles that depend on the component's state.
  - Use sparingly, as they can make the component less readable.
- **Example**:

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

function App() {
  const style = {
    textAlign: 'center',
    color: 'blue'
  };

  return <div style={style}>Hello World</div>;
}
```

Question 39: Create a `FormInput` component with a text input and a state to store its value. Ensure that the input's value is both controlled by the state and can update the state when it changes.

Answer:

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

class FormInput extends Component {
  constructor(props) {
    super(props);
    this.state = {
      inputValue: '' // Initial state for the input value
    };
  }

  handleChange = (event) => {
    this.setState({ inputValue: event.target.value });
  }

  render() {
    return (
      <div>
        <input
          type="text"
          value={this.state.inputValue}
          onChange={this.handleChange}
        />
      </div>
    );
  }
}

export default FormInput;
```

Question 17: Explain the usage of Radium in applying hover effects to a `Button` component. Provide the setup for Radium and the necessary styles within the component.

Answer:

- Button.js:

```
import React from 'react';
import Radium from 'radium';

// Button component with Radium
const Button = (props) => {
```

```
  // Styles for the button
  const styles = {
    button: {
      backgroundColor: '#007bff',
      border: 'none',
      color: 'white',
      padding: '10px 20px',
      textAlign: 'center',
      textDecoration: 'none',
      display: 'inline-block',
      fontSize: '16px',
      margin: '4px 2px',
      cursor: 'pointer',

      // Radium hover styles
      ':hover': {
        backgroundColor: '#0056b3'
      }
    }
  };

  return (
    <button style={styles.button}>
      {props.children}
    </button>
  );
}

// Wrap your component with Radium
export default Radium(Button);
```

- App.js:

```
import React from 'react';
import Button from './Button'; // Import the Button component

const App = () => {
  return (
    <div>
      <Button>Click Me</Button>
    </div>
  );
}

export default App;
```

Note: Render App.js in Index.js.