Node.js

1. **What is Node.js, and how does it differ from traditional server-side technologies?**
    - **Node.js** is a JavaScript runtime built on Chrome's V8 JavaScript engine. It differs from traditional server-side technologies in that it is event-driven and non-blocking, using a single thread to handle multiple requests concurrently, rather than spawning a new thread for each request.
2. **Explain the event-driven, non-blocking I/O model in Node.js and why it's beneficial.**
    - **Event-driven, non-blocking I/O** allows Node.js to handle multiple operations like reading files, making HTTP requests, or querying databases without blocking the main execution thread, which results in better scalability and faster performance, especially for I/O-heavy operations.
3. **How does the Node.js runtime handle multiple requests simultaneously with a single-threaded architecture?**
    - Node.js uses the **Event Loop** and **Callback Queue** to handle multiple requests asynchronously. While the event loop processes a single thread, it delegates blocking operations to the system's kernel, which can perform multiple tasks in parallel.
4. **What are some core modules in Node.js, and what are they used for? (e.g., fs, http, path)**
    - **fs**: File system operations (read/write files).
    - **http**: Creating HTTP servers.
    - **path**: Utilities for working with file and directory paths.
5. **What is npm, and how does it help in managing dependencies in Node.js projects?**
    - **npm** is the Node Package Manager, a tool for managing and installing third-party packages in Node.js. It helps with dependency management, ensuring that the right versions of libraries are used in your projects.
6. **Explain how you would create a basic HTTP server in Node.js. Provide example code.**

```
const http = require('http');
const server = http.createServer((req, res) => {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Hello, Node.js!');
});
server.listen(3000, () => {
  console.log('Server running on port 3000');
});
```

7. **What is the purpose of the package.json file, and what key information does it contain?**
    - **package.json** contains metadata about the project, including the project name, version, dependencies, scripts, and other configurations. It's essential for managing dependencies and project settings.
8. **How do you handle asynchronous operations in Node.js? Explain the role of callbacks, promises, and async/await.**
    - **Callbacks**: Functions passed as arguments to other functions, executed after a task completes.

- **Promises**: An object representing the eventual completion (or failure) of an asynchronous operation.
- **Async/await**: Syntactic sugar over promises, allowing asynchronous code to be written in a synchronous style.

9. **What are streams in Node.js, and how are they used? Provide an example.**
   - **Streams** are objects that allow reading or writing data in a continuous flow. Example:

     ```
     const fs = require('fs');
     const readStream = fs.createReadStream('input.txt');
     readStream.pipe(process.stdout);
     ```

10. **Explain error handling in Node.js. How would you handle errors globally across an application?**
    - Use **try-catch** for synchronous errors and **.catch()** for promise rejections. For global error handling, use an **uncaughtException** listener for unhandled errors and an **unhandledRejection** listener for unhandled promise rejections.

ExpressJS

11. **What is ExpressJS, and why is it commonly used in web development?**
    - **ExpressJS** is a web application framework for Node.js that simplifies building robust APIs and web applications by providing features like routing, middleware, and template rendering.
12. **How do you set up a basic ExpressJS server? Provide example code.**

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

13. **What is middleware in ExpressJS? How does it work?**
    - **Middleware** is a function that processes requests before they reach the route handler, used for tasks like logging, authentication, and parsing request bodies.
14. **How do you define routes in ExpressJS?**
    - Use methods like `app.get()`, `app.post()`, etc., to define routes. Example:

      ```
      app.get('/home', (req, res) => res.send('Home Page'));
      ```

15. **What is the difference between app.get(), app.post(), app.put(), and app.delete() in Express?**
    - These methods define different HTTP verbs:

- **app.get()**: Handles GET requests.
- **app.post()**: Handles POST requests.
- **app.put()**: Handles PUT requests.
- **app.delete()**: Handles DELETE requests.

16. **How can you handle errors in ExpressJS applications?**
    - Define a custom error-handling middleware at the end of the middleware stack using `app.use()`. Example:

```
app.use((err, req, res, next) => {
  res.status(500).send('Something went wrong');
});
```

17. **Explain how to set up a static file server in ExpressJS.**
    - Use `express.static()` to serve static files. Example:

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

18. **What is req.params vs. req.query in ExpressJS, and how are they used?**
    - **req.params**: Used for route parameters (e.g., `/users/:id`).
    - **req.query**: Used for query string parameters (e.g., `/search?term=node`).
19. **How can you use environment variables in Express applications?**
    - Use `process.env.VARIABLE_NAME` to access environment variables. It's common to use the `dotenv` package for loading environment variables from a `.env` file.
20. **How do you secure an ExpressJS application? Give examples of best practices.**
    - Use **HTTPS**, sanitize user inputs, enable **CORS** properly, use **helmet** for security headers, and manage **session/cookies** securely.

AJAX

21. **What is AJAX, and how does it differ from a traditional HTTP request?**
    - **AJAX (Asynchronous JavaScript and XML)** allows for sending HTTP requests without reloading the page. Unlike traditional requests, AJAX sends data in the background, improving user experience.
22. **Explain the core steps in making an AJAX request using JavaScript.**
    - Create an `XMLHttpRequest` object, open the request, send the request, and define a callback function to handle the response.
23. **What is the role of XMLHttpRequest in AJAX?**
    - **XMLHttpRequest** is an API used to send HTTP requests to a server and handle the response asynchronously.
24. **How do you perform an AJAX request with jQuery? Provide example code.**

```
$.ajax({
  url: 'https://api.example.com',
  type: 'GET',
  success: (data) => { console.log(data); },
  error: (err) => { console.error(err); }
```

```
});
```

25. **What are the different states of an XMLHttpRequest, and what do they mean?**
    - **0 (UNSENT)**: Request not initialized.
    - **1 (OPENED)**: Request has been set up.
    - **2 (HEADERS_RECEIVED)**: Response headers received.
    - **3 (LOADING)**: Response body is being received.
    - **4 (DONE)**: Request completed.

ReactJS

31. **What is ReactJS, and what problem does it solve in web development?**
    - **ReactJS** is a JavaScript library for building user interfaces. It solves the problem of efficiently updating the UI by using a virtual DOM, ensuring fast rendering.
32. **What is JSX, and how does it differ from HTML?**
    - **JSX** is a syntax extension for JavaScript that allows writing HTML-like code within JavaScript. It's transformed into regular JavaScript by tools like Babel.
33. **Explain the React lifecycle methods in detail.**
    - Lifecycle methods like **componentDidMount**, **shouldComponentUpdate**, **componentWillUnmount**, etc., are used in class components to manage side effects, re-rendering, and cleanup.
34. **Explain the role of props in React components.**
    - **Props** (short for properties) are used to pass data from a parent component to a child component.
35. **What is the purpose of state in a React component?**
    - **State** holds data that changes over time and triggers re-rendering of components when updated.
36. **Differentiate between functional and class-based components in React.**
    - **Class components** have a constructor and lifecycle methods, while **functional components** are simpler and can use hooks like `useState` and `useEffect`.
37. **How does React use the virtual DOM to improve performance?**
    - React uses the **virtual DOM** to create a lightweight copy of the actual DOM. When the state changes, React compares the virtual DOM to the real DOM and updates only the parts that have changed, improving performance.
38. **What are React Hooks, and why were they introduced?**
    - **Hooks** like `useState`, `useEffect`, and `useContext` were introduced to allow functional components to manage state and side effects without needing class components.
39. **What is the role of useState in React? How is it used?**
    - `useState` is a hook used to declare and update state in functional components. Example:

      ```
      const [count, setCount] = useState(0);
      ```

40. **Explain useEffect and provide an example of its usage.**

- `useEffect` runs side effects like data fetching, subscriptions, etc. It runs after every render unless dependencies are specified.

```
useEffect(() => {
  console.log('Component mounted');
}, []); // Empty array means it runs once after the initial
render.
```

41. What are controlled components in React, and why are they important for form handling?

- **Controlled components** are components where form inputs (such as `<input>`, `<textarea>`, etc.) are bound to the component's state. The value of the input is controlled by React, and any changes to the input's value are stored in the component's state.

  Importance for form handling:

  - They make it easier to handle form validation, state synchronization, and dynamic updates because the form data is managed within the React component itself.
  - Example:

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

  const handleChange = (e) => {
    setName(e.target.value);
  };

  return <input type="text" value={name} onChange={handleChange}
/>;
}
```

42. What are the new features introduced in React 18?

- **Concurrent Rendering**: React 18 introduces **Concurrent Mode**, which allows React to work on multiple tasks at once, improving the app's responsiveness by rendering UI updates in the background.
- **Automatic Batching**: React 18 automatically batches state updates for better performance.
- **`useId` hook**: Helps generate unique IDs for accessibility and SSR (Server-Side Rendering).
- **Suspense for Data Fetching**: Suspense can now be used for data fetching, making asynchronous data loading more seamless.
- **`startTransition`**: Lets you mark certain updates as non-urgent, helping React prioritize high-priority updates over low-priority ones.

43. How does one pass data between components in React?

- **Props**: Data is passed from a parent component to a child component using **props**. Example:

```
function Parent() {
  const name = 'John';
  return <Child name={name} />;
}

function Child({ name }) {
  return <p>{name}</p>;
}
```

- **State lifting**: If a child component needs to pass data to a parent, you can "lift the state" up to the parent component.

44. How does React handle forms and controlled components?

- React handles forms by using **controlled components**, where form data is stored in the component's state and updated through event handlers. It allows the app to have full control over form behavior and validation. Example:

```
function Form() {
  const [value, setValue] = useState('');

  const handleChange = (e) => {
    setValue(e.target.value);
  };

  return <input type="text" value={value} onChange={handleChange} />;
}
```

45. What is useContext, and how does it simplify state management?

- **useContext** is a hook used to access values from a React Context, which allows you to pass data through the component tree without having to pass props down manually at every level.
    - Example:

```
const ThemeContext = createContext('light');

function Child() {
  const theme = useContext(ThemeContext);
  return <p>Current theme: {theme}</p>;
}

function Parent() {
  return (
    <ThemeContext.Provider value="dark">
      <Child />
    </ThemeContext.Provider>
  );
}
```

46. Explain the purpose of React's useReducer Hook and provide an example.

- **useReducer** is a hook used for managing more complex state logic in React. It's similar to `useState`, but allows you to update state based on actions in a more structured way, similar to Redux.

  Example:

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

function reducer(state, action) {
  switch (action.type) {
    case 'increment':
      return { count: state.count + 1 };
    case 'decrement':
      return { count: state.count - 1 };
    default:
      return state;
  }
}

function Counter() {
  const [state, dispatch] = useReducer(reducer, initialState);

  return (
    <div>
      <p>Count: {state.count}</p>
      <button onClick={() => dispatch({ type: 'increment'
})}>Increment</button>
      <button onClick={() => dispatch({ type: 'decrement'
})}>Decrement</button>
    </div>
  );
}
```

47. What is React Router, and how does it manage navigation in a React app?

- **React Router** is a library used for managing navigation in single-page React applications. It uses **URL-based routing** to render different components based on the URL path.

  Example:

```
import { BrowserRouter as Router, Route, Switch } from 'react-router-
dom';

function App() {
  return (
    <Router>
      <Switch>
        <Route path="/home" component={Home} />
        <Route path="/about" component={About} />
      </Switch>
```

```
    </Router>
  );
}
```

48. How do you implement conditional rendering in React?

- Conditional rendering in React can be done using JavaScript operators like **if**, **ternary operators**, or **short-circuiting**.

  Example:

```
function Greeting({ isLoggedIn }) {
  return (
    <div>
      {isLoggedIn ? <p>Welcome back!</p> : <p>Please log in</p>}
    </div>
  );
}
```

49. What is prop drilling, and how can it be minimized?

- **Prop drilling** occurs when you pass data through many layers of components, which makes the code harder to maintain. To minimize it, you can use:
  - **React Context API**: To share data across components without explicitly passing props.
  - **State management libraries** like Redux or Zustand.

50. What is the purpose of React.memo()? How does it work?

- **React.memo()** is a higher-order component (HOC) that memoizes a component, preventing unnecessary re-renders if the props have not changed. It is useful for functional components that render the same output given the same props.

  Example:

```
const MyComponent = React.memo(function MyComponent(props) {
  return <div>{props.name}</div>;
});
```

51. Explain the role of React.forwardRef and when it should be used.

- **React.forwardRef** is used to forward a ref to a child component. It allows a parent component to directly interact with a child component's DOM node.

  Example:

```
const Input = React.forwardRef((props, ref) => {
  return <input ref={ref} {...props} />;
});
```

```
const Parent = () => {
  const inputRef = useRef();
  return <Input ref={inputRef} />;
};
```

52. What is React.lazy() and Suspense? How do they support code-splitting?

- **React.lazy()** allows you to dynamically import components only when they are needed (code-splitting).
- **Suspense** is used to show a fallback (like a loading spinner) while the lazy-loaded component is being fetched.

  Example:

```
const LazyComponent = React.lazy(() => import('./LazyComponent'));

function App() {
  return (
    <Suspense fallback={<div>Loading...</div>}>
      <LazyComponent />
    </Suspense>
  );
}
```

53. How can you optimize performance in a React app?

- Use **React.memo()**, **useMemo()**, and **useCallback()** to prevent unnecessary re-renders.
- Code splitting with **React.lazy()** and **Suspense**.
- Optimize images and other assets.
- Implement virtual lists for large data sets (e.g., **react-window**).
- Avoid using inline functions in JSX.

54. What is higher-order component (HOC) in React? Provide an example.

- **HOC** is a function that takes a component and returns a new component with additional props or functionality.

  Example:

```
function withLoading(Component) {
  return function WithLoading(props) {
    if (props.isLoading) {
      return <p>Loading...</p>;
    }
    return <Component {...props} />;
  };
}
```

55. How do you handle errors in React using error boundaries?

- **Error boundaries** are components that catch JavaScript errors anywhere in their child component tree and display a fallback UI instead of crashing the app.

  Example:

```
class ErrorBoundary extends React.Component {
  state = { hasError: false };

  static getDerivedStateFromError() {
    return { hasError: true };
  }

  componentDidCatch(error, info) {
    console.log(error, info);
  }

  render() {
    if (this.state.hasError) {
      return <h1>Something went wrong.</h1>;
    }
    return this.props.children;
  }
}
```

56. How does one implement lazy loading in React?

- **Lazy loading** in React can be implemented using `React.lazy()` in combination with `Suspense` to load components only when needed (e.g., when they are rendered). This improves performance by splitting the code into smaller chunks.

  Example:

```
const LazyComponent = React.lazy(() => import('./LazyComponent'));

function App() {
  return (
    <Suspense fallback={<div>Loading...</div>}>
      <LazyComponent />
    </Suspense>
  );
}
```

  - `React.lazy()` dynamically imports the `LazyComponent`, and `Suspense` provides a fallback UI (like a loading spinner) until the component is loaded.

57. What is React Context API, and when would you use it?

- **React Context API** is a way to share data (like state, theme, language settings) across components without passing props down manually at every level.
  - It allows you to avoid **prop drilling**, which occurs when you pass data through many layers of components.

- You would use it when you need global state across multiple components but don't want to use a full-fledged state management solution like Redux.

Example:

```
const ThemeContext = React.createContext('light'); // default value

function App() {
  return (
    <ThemeContext.Provider value="dark">
      <Child />
    </ThemeContext.Provider>
  );
}

function Child() {
  const theme = useContext(ThemeContext);
  return <div>Current Theme: {theme}</div>;
}
```

58. What is Redux, and why would you use it in a React application?

- **Redux** is a predictable state container for JavaScript applications. It helps manage the application's state in a centralized store, enabling components to access and modify state through actions and reducers.
  - Use Redux in React when the application has complex state logic or requires state to be shared across many components, especially in large applications with multiple levels of nesting.

59. Explain the concept of actions, reducers, and store in Redux.

- **Actions**: Plain JavaScript objects that describe *what happened*. Each action has a `type` property and optionally a `payload` (data related to the action). Example:

```
const action = { type: 'ADD_ITEM', payload: { item: 'apple' } };
```

- **Reducers**: Pure functions that specify how the application's state changes in response to an action. Reducers take the current state and an action, and return a new state. Example:

```
function itemReducer(state = [], action) {
  switch (action.type) {
    case 'ADD_ITEM':
      return [...state, action.payload.item];
    default:
      return state;
  }
}
```

- **Store**: Holds the application state. It is created using `createStore()` and contains the current state, as well as methods like `dispatch()` to dispatch actions and `getState()` to retrieve the current state.

60. How does the useSelector Hook work in a React-Redux application?

- The **`useSelector`** hook is used to extract data from the Redux store. It takes a selector function as an argument, which specifies which piece of state you want to access.

  Example:

```
import { useSelector } from 'react-redux';

function Component() {
  const items = useSelector((state) => state.items);
  return <div>{items.length} items</div>;
}
```

61. What is middleware in Redux? Give an example.

- **Middleware** in Redux is a way to extend Redux's store functionality by adding custom logic between dispatching an action and the action reaching the reducer. Common use cases include logging, crash reporting, or handling asynchronous actions (like fetching data from an API).

  Example: Using `redux-thunk` for asynchronous actions:

```
// Action creator using thunk middleware for async action
const fetchData = () => {
  return (dispatch) => {
    dispatch({ type: 'FETCH_START' });
    fetch('/api/data')
      .then((response) => response.json())
      .then((data) => dispatch({ type: 'FETCH_SUCCESS', payload: data
}))
      .catch((error) => dispatch({ type: 'FETCH_ERROR', error }));
  };
};
```

62. Explain the difference between `useDispatch` and `useSelector`.

- **`useDispatch`** is a hook that gives access to the Redux **dispatch function**, which is used to send actions to the Redux store.

  Example:

```
const dispatch = useDispatch();
dispatch({ type: 'INCREMENT' });
```

- **`useSelector`** is a hook that allows you to **read** from the Redux store, i.e., to select a piece of state from the store.

63. What is the `connect()` function in Redux? How does it connect React to Redux?

- **`connect()`** is a higher-order component (HOC) that connects a React component to the Redux store. It allows the component to access the Redux state and dispatch actions. It replaces `useSelector` and `useDispatch` hooks in class-based components.

  Example:

```
import { connect } from 'react-redux';

function MyComponent({ items, dispatch }) {
  return (
    <div>
      <button onClick={() => dispatch({ type: 'ADD_ITEM', payload:
'banana' })}>Add Item</button>
      <ul>
        {items.map((item) => <li key={item}>{item}</li>)}
      </ul>
    </div>
  );
}

const mapStateToProps = (state) => ({
  items: state.items
});

export default connect(mapStateToProps)(MyComponent);
```

64. Explain how to manage asynchronous operations in Redux.

- **Redux-thunk** is a middleware that allows you to dispatch functions (thunks) instead of just actions. Thunks can be used to handle asynchronous operations like fetching data from APIs.
  - Example:

```
const fetchData = () => {
  return (dispatch) => {
    dispatch({ type: 'FETCH_REQUEST' });
    fetch('/api/data')
      .then((response) => response.json())
      .then((data) => dispatch({ type: 'FETCH_SUCCESS', payload:
data }))
      .catch((error) => dispatch({ type: 'FETCH_FAILURE', error
}));
  };
};
```

- **Redux-Saga** is another popular library for handling side effects, particularly complex asynchronous logic.

65. How can you combine multiple reducers in a Redux app?

- To combine multiple reducers, you can use **`combineReducers()`** from Redux. This function takes an object of reducers and returns a single reducer function.

  Example:

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

const rootReducer = combineReducers({
  items: itemsReducer,
  user: userReducer
});

export default rootReducer;
```

66. What is the purpose of Redux DevTools, and how do you set it up?

- **Redux DevTools** is a browser extension that provides powerful tools to inspect and debug Redux state changes, actions, and the history of state transitions. It allows you to time-travel through actions and states.

  To set it up, you need to apply the Redux DevTools extension to your store when creating it:

```
import { createStore } from 'redux';
import rootReducer from './reducers';

const store = createStore(
  rootReducer,
  window.__REDUX_DEVTOOLS_EXTENSION__ &&
window.__REDUX_DEVTOOLS_EXTENSION__()
);
```

  This will enable the DevTools extension in your browser, allowing you to track actions and state changes visually.
