React.js Theory Questions

Introduction to React.js

Question 1: What is React.js? How is it different from other JavaScript frameworks and libraries?

What is React.js? React.js is a JavaScript library developed by Facebook (now Meta) for building user interfaces, particularly single-page applications. It focuses on creating reusable UI components and efficiently updating the user interface when data changes.

Key Differences from Other Frameworks/Libraries:

- **Library vs Framework**: React is a library focused on the view layer, unlike Angular which is a full framework providing routing, HTTP client, forms, etc.
- **Virtual DOM**: React uses a virtual DOM for efficient updates, while frameworks like jQuery manipulate the real DOM directly
- **Component-Based**: Unlike traditional MVC frameworks, React follows a component-based architecture
- **Unidirectional Data Flow**: React enforces one-way data binding, unlike Angular's two-way data binding
- **JSX Syntax**: React uses JSX (JavaScript XML) which allows writing HTML-like syntax in JavaScript
- **Learning Curve**: React has a gentler learning curve compared to comprehensive frameworks like Angular
- **Ecosystem**: React requires additional libraries for complete functionality (routing, state management), while frameworks like Angular provide everything built-in

Question 2: Explain the core principles of React such as the virtual DOM and component-based architecture.

Core Principles of React:

1. Virtual DOM

- A JavaScript representation of the real DOM kept in memory
- When state changes occur, React creates a new virtual DOM tree
- Compares (diffs) the new virtual DOM with the previous version
- Updates only the parts of the real DOM that actually changed
- This process is called reconciliation and makes React applications fast

2. Component-Based Architecture

- Applications are built as a tree of components
- Each component manages its own state and renders UI
- Components can be composed together to build complex UIs
- Promotes reusability and maintainability
- Components are independent and can be tested in isolation

3. Unidirectional Data Flow

- Data flows down from parent to child components via props
- Child components cannot directly modify parent data
- Events flow up from child to parent components
- Makes debugging easier and applications more predictable

4. Declarative Programming

- You describe what the UI should look like for any given state
- React handles the how (DOM manipulation)
- Opposite of imperative programming where you specify step-by-step instructions

Question 3: What are the advantages of using React.js in web development?

Advantages of React.js:

Performance Benefits:

- Virtual DOM ensures efficient updates and rendering
- Minimizes expensive DOM operations
- Better user experience with faster load times

Development Experience:

- Reusable components reduce code duplication
- JSX makes code more readable and intuitive
- Strong developer tools (React DevTools)
- Hot reloading for faster development

Maintainability:

- Component-based architecture makes code modular

- Easier to debug due to predictable data flow
- Clear separation of concerns

Community and Ecosystem:

- Large, active community with extensive resources
- Rich ecosystem of third-party libraries
- Strong job market and career opportunities

Flexibility:

- Can be integrated into existing projects gradually
- Works well with other libraries and frameworks
- Suitable for both small and large-scale applications

SEO and Performance:

- Server-side rendering support with Next.js
- Better SEO capabilities compared to traditional SPAs
- Code splitting and lazy loading capabilities

Components (Functional & Class Components)

Question 1: What are components in React? Explain the difference between functional components and class components.

What are Components? Components are independent, reusable pieces of code that return JSX elements to be rendered to the screen. They serve the same purpose as JavaScript functions but work in isolation and return HTML via a render function.

Functional Components:

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

// Or with arrow function
const Welcome = (props) => {
```

```
  return <h1>Hello, {props.name}</h1>;
};
```

Class Components:

```
class Welcome extends React.Component {
  render() {
    return <h1>Hello, {this.props.name}</h1>;
  }
}
```

Key Differences:

Aspect	Functional Components	Class Components
Syntax	Simple function syntax	ES6 class syntax extending React.Component
State Management	Uses useState Hook (React 16.8+)	Uses this.state and this.setState()
Lifecycle Methods	Uses useEffect Hook	Built-in lifecycle methods (componentDidMount, etc.)
Performance	Slightly better performance	Slightly more overhead
Code Length	More concise and readable	More verbose
Testing	Easier to test	More complex to test
Current Trend	Preferred approach (modern React)	Legacy approach (still supported)

Question 2: How do you pass data to a component using props?

Props (Properties) Overview: Props are arguments passed into React components, similar to function parameters. They allow data to flow from parent to child components.

Passing Props - Parent Component:

```
function App() {
  return (
    <div>
```

```
      <Welcome name="John" age={25} isStudent={true} />
      <Welcome name="Jane" age={30} isStudent={false} />
    </div>
  );
}
```

Receiving Props - Child Component (Functional):

```
function Welcome(props) {
  return (
    <div>
      <h1>Hello, {props.name}</h1>
      <p>Age: {props.age}</p>
      <p>Student: {props.isStudent ? 'Yes' : 'No'}</p>
    </div>
  );
}

// With destructuring
function Welcome({ name, age, isStudent }) {
  return (
    <div>
      <h1>Hello, {name}</h1>
      <p>Age: {age}</p>
      <p>Student: {isStudent ? 'Yes' : 'No'}</p>
    </div>
  );
}
```

Receiving Props - Child Component (Class):

```
class Welcome extends React.Component {
```

```
  render() {
    return (
      <div>
        <h1>Hello, {this.props.name}</h1>
        <p>Age: {this.props.age}</p>
        <p>Student: {this.props.isStudent ? 'Yes' : 'No'}</p>
      </div>
    );
  }
}
```

Important Props Characteristics:

- Props are read-only (immutable)
- Props can be any data type (strings, numbers, objects, functions, etc.)
- Props flow down from parent to child (unidirectional)
- Components should never modify their props directly

Question 3: What is the role of render() in class components?

Role of render() Method:

The render() method is the most important method in a React class component. It's the only required method in a class component.

Key Responsibilities:

1. Returns JSX Elements:

```
class MyComponent extends React.Component {
  render() {
    return (
      <div>
        <h1>Welcome to React</h1>
        <p>This is rendered by the render method</p>
      </div>
```

```
    );
  }
}
```

2. Determines What Gets Displayed:

- Controls the component's output to the DOM
- Called automatically when component needs to re-render
- Must return a single JSX element (or React Fragment)

3. Access to Props and State:

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

  render() {
    return (
      <div>
        <h1>{this.props.title}</h1>
        <p>Count: {this.state.count}</p>
      </div>
    );
  }
}
```

4. Conditional Rendering:

```
class MyComponent extends React.Component {
  render() {
    const { isLoggedIn } = this.props;
```

```
    if (isLoggedIn) {
      return <h1>Welcome back!</h1>;
    } else {
      return <h1>Please log in</h1>;
    }
  }
}
```

Important render() Characteristics:

- Pure function - should not modify state
- Called every time component needs to update
- Should not contain side effects (API calls, etc.)
- Must return JSX, null, or false
- Cannot call setState() directly in render()

Props and State

Question 1: What are props in React.js? How are props different from state?

What are Props? Props (short for properties) are read-only data passed from parent components to child components. They allow components to communicate and share data.

What is State? State is mutable data that belongs to a component and can change over time. It represents the component's local data that can trigger re-renders when updated.

Key Differences:

Aspect	Props	State
Mutability	Immutable (read-only)	Mutable (can be changed)
Ownership	Owned by parent component	Owned by the component itself
Data Flow	Passed down from parent to child	Internal to component
Modification	Cannot be modified by receiving component	Can be modified using setState()

Aspect	Props	State
Purpose	Communication between components	Managing component's internal data
Initialization	Passed as attributes	Initialized in constructor or useState
Triggers Re-render	Changes in parent cause re-render	setState() triggers re-render

Props Example:

```
// Parent component
function App() {
  return <UserCard name="John" email="john@example.com" />;
}

// Child component - cannot modify props
function UserCard(props) {
  // props.name = "Jane"; // This would cause an error!
  return (
    <div>
      <h2>{props.name}</h2>
      <p>{props.email}</p>
    </div>
  );
}
```

State Example:

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

```
  }

  increment = () => {
    this.setState({ count: this.state.count + 1 }); // Can modify state
  };

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

Question 2: Explain the concept of state in React and how it is used to manage component data.

State Concept: State is a JavaScript object that holds dynamic data for a React component. It represents the component's memory and allows components to create dynamic and interactive user interfaces.

Key Characteristics of State:

1. Local and Encapsulated:

- Each component has its own state
- State is private to the component
- Other components cannot access it directly

2. Mutable:

- State can be changed during the component's lifecycle
- Changes trigger component re-rendering

3. Asynchronous Updates:

- State updates may be asynchronous
- React may batch multiple setState calls for performance

State in Class Components:

```
class WeatherApp extends React.Component {
  constructor(props) {
    super(props);
    // Initialize state
    this.state = {
      temperature: 20,
      city: 'New York',
      loading: false,
      error: null
    };
  }

  updateWeather = () => {
    this.setState({ loading: true });

    // Simulate API call
    setTimeout(() => {
      this.setState({
        temperature: 25,
        loading: false
      });
    }, 1000);
  };

  render() {
```

```
    const { temperature, city, loading } = this.state;

    return (
      <div>
        <h1>Weather in {city}</h1>
        {loading ? (
          <p>Loading...</p>
        ) : (
          <p>Temperature: {temperature}°C</p>
        )}
        <button onClick={this.updateWeather}>Update Weather</button>
      </div>
    );
  }
}
```

State in Functional Components (with Hooks):

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

function WeatherApp() {
  // Multiple state variables
  const [temperature, setTemperature] = useState(20);
  const [city, setCity] = useState('New York');
  const [loading, setLoading] = useState(false);

  const updateWeather = () => {
    setLoading(true);

    setTimeout(() => {
```

```
    setTemperature(25);
    setLoading(false);
  }, 1000);
 };

 return (
  <div>
   <h1>Weather in {city}</h1>
   {loading ? (
    <p>Loading...</p>
   ) : (
    <p>Temperature: {temperature}°C</p>
   )}
   <button onClick={updateWeather}>Update Weather</button>
  </div>
 );
}
```

Common Use Cases for State:

- Form input values
- Toggle states (show/hide, enabled/disabled)
- Loading states
- Error messages
- Counter values
- Fetched data from APIs
- UI state (selected items, active tabs)

Question 3: Why is this.setState() used in class components, and how does it work?

Why setState() is Used:

1. Triggers Re-rendering:

- Direct state mutation doesn't trigger re-render
- setState() tells React that state has changed
- React schedules a re-render of the component

2. Ensures React's Reconciliation:

- React needs to know when to update the Virtual DOM
- Direct mutation bypasses React's update mechanism
- setState() integrates with React's lifecycle

3. Maintains Component Consistency:

- Ensures proper component lifecycle execution
- Allows React to batch updates for performance
- Maintains predictable component behavior

Correct Way:

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

  // CORRECT - Using setState()
  increment = () => {
    this.setState({ count: this.state.count + 1 });
  };

  render() {
    return (
      <div>
        <p>Count: {this.state.count}</p>
```

```
      <button onClick={this.increment}>Increment</button>
    </div>
  );
 }
}
```

How setState() Works:

1. Object Merge:

```
this.state = { name: 'John', age: 25, city: 'NYC' };

// Only updates age, keeps name and city unchanged
this.setState({ age: 26 });
```

2. Functional Updates:

```
// For updates based on previous state
this.setState(prevState => ({
 count: prevState.count + 1
}));

// With callback to access props
this.setState((prevState, props) => ({
 count: prevState.count + props.increment
}));
```

3. Callback Function:

```
this.setState(
 { count: this.state.count + 1 },
 () => {
  // This callback runs after state is updated
  console.log('New count:', this.state.count);
 }
```

);

4. Batching and Asynchronous Nature:

// These may be batched together

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

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

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

// May only increment by 1, not 3!

// Use functional form for sequential updates:

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

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

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

Important setState() Rules:

- Never mutate state directly
- setState() calls are asynchronous
- React may batch multiple setState() calls
- Use functional form when new state depends on previous state
- setState() merges the new state with existing state
- Always treat state as immutable