2. React JS

Templating using JSX

Templating using JSX (JavaScript XML) in React.js is a way of creating dynamic and reusable components that define the structure and content of your user interface. JSX is a syntax extension for JavaScript that allows you to write HTML-like code within your JavaScript code, making it easier to describe the structure of your UI components. Here's a breakdown of how templating using JSX works in React.js, along with an example:

1. **JSX Syntax**: JSX allows you to write HTML-like code in your JavaScript files. It looks similar to HTML but is used to define React components. JSX elements resemble HTML tags, and you can also embed JavaScript expressions within curly braces `{}`.

2. **Creating Components**: In React, you create components using JSX. Components are the building blocks of your UI. You can create functional components or class-based components. These components can be reused throughout your application.

3. **Props**: Components can receive data from their parent components through props (short for properties). Props are like attributes in HTML and provide a way to pass data and configuration to your components.

4. **Example**:

Let's consider a simple example of creating a "Button" component using JSX in React.js:

```
import React from 'react';

// Functional Component
function Button(props) {
  return (
    <button
```

```
      className={props.className}
      onClick={props.onClick}
    >
      {props.label}
    </button>
  );
}

export default Button;
```

In this example, we've created a functional component named "Button". It receives three props: `className`, `onClick`, and `label`. The component renders a `<button>` element with the provided class, click handler, and label.

You can then use this "Button" component in other parts of your application:

```
import React from 'react';
import Button from './Button'; // Assuming the path is correct

function App() {
  const handleButtonClick = () => {
    alert('Button clicked!');
  };

  return (
    <div>
      <h1>My App</h1>
      <Button
        className="primary-button"
        onClick={handleButtonClick}
```

```
      label="Click Me"
    />
  </div>
 );
}

export default App;
```

In this usage example, we've imported the "Button" component and used it within the "App" component. The `handleButtonClick` function is passed as the `onClick` prop to the "Button" component, and the label "Click Me" is also provided.

Remember that JSX gets transpiled into JavaScript code that React understands, so while it looks like HTML, it's actually creating and manipulating React elements under the hood.

1. **Expressions in JSX**:

JSX allows you to embed JavaScript expressions within curly braces `{}`. This allows you to dynamically generate content or values within your JSX elements.

```
import React from 'react';

function App() {
  const greeting = 'Hello, React!';
  const sum = (a, b) => a + b;

  return (
    <div>
      <h1>{greeting}</h1>
      <p>The sum of 2 and 3 is {sum(2, 3)}.</p>
    </div>
```

```
  );
}

export default App;
```

2. **Functions in JSX**:

   You can call functions within JSX elements and use their return values to render content.

```
import React from 'react';

function getName() {
  return 'John Doe';
}

function App() {
  return (
    <div>
      <p>Hello, {getName()}!</p>
    </div>
  );
}

export default App;
```

3. **Attributes in JSX**:

   JSX elements can have attributes, similar to HTML elements. You can pass values from variables or functions to these attributes.

```
import React from 'react';

function App() {
  const imgUrl = 'https://example.com/image.jpg';

  return (
    <div>
      <img src={imgUrl} alt="Example" />
      <a href="https://example.com">Visit Example</a>
    </div>
  );
}

export default App;
```

Attributes can also accept JavaScript expressions, making it easy to conditionally set values:

```
import React from 'react';

function App() {
  const isLoggedIn = true;

  return (
    <div>
      <p>{isLoggedIn ? 'Welcome back!' : 'Please log in.'}</p>
    </div>
```

```
  );
}

export default App;
```

Remember that when embedding JavaScript expressions or calling functions in JSX, you should enclose them in curly braces `{}` to indicate that you're switching from JSX mode to JavaScript mode.

Props:

In React functional components, props are a way to pass data from parent components to child components. Props (short for "properties") are similar to function arguments in JavaScript, allowing you to customize the behavior and appearance of a component. Here's how you use props in a functional component:

1. **Passing Props from Parent Component**:

In the parent component, you can pass props to a child component by including attributes within the JSX element:

```
import React from 'react';
import ChildComponent from './ChildComponent'; // Assuming the path is correct

function ParentComponent() {
  const greeting = "Hello from Parent!";

  return (
    <div>
      <ChildComponent message={greeting} />
```

```
    </div>
  );
}

export default ParentComponent;
```

2. **Receiving Props in Child Component**:

In the child component, you can access the props by receiving them as a parameter to the functional component:

```
import React from 'react';

function ChildComponent(props) {
  return <p>{props.message}</p>;
}

export default ChildComponent;
```

In this example, the `ChildComponent` receives the `message` prop and renders its value within a `<p>` element.

3. **Using Props in Child Component**:

You can use the received props in your child component's JSX to render dynamic content:

```
import React from 'react';
```

```
function ChildComponent(props) {
  return (
    <div>
      <p>{props.message}</p>
      <p>Length of message: {props.message.length}</p>
    </div>
  );
}

export default ChildComponent;
```

Props are read-only, meaning that the child component should not modify the values of its props. They provide a way for parent components to communicate with their children in a one-way manner.

In summary, props allow you to pass data from parent components to child components, enabling you to create reusable and dynamic components in your React applications.

Event Management:

Event handling in React.js involves managing user interactions, such as clicks, inputs, and other actions. React provides a consistent and synthetic event system that abstracts away browser differences. Here's how you can manage events in React:

1. **Basic Event Handling**:

In JSX, you can attach event handlers to elements using camelCase syntax, such as `onClick`, `onChange`, etc.

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

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

```
  const handleButtonClick = () => {
    setCount(count + 1);
  };

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

export default App;
```

In this example, when the button is clicked, the `handleButtonClick` function is called, and it updates the `count` state.

2. **Event Object**:

When using event handlers, React provides a synthetic event object that is similar to the native DOM event. You can access it as the first argument of your event handler function.

```
import React from 'react';

function App() {
  const handleInputChange = (event) => {
    console.log('Input value:', event.target.value);
  };

  return (
```

```
    <div>
      <input type="text" onChange={handleInputChange} />
    </div>
  );
}

export default App;
```

3. **Preventing Default Behavior**:

You can prevent the default behavior of an event using the `preventDefault()` method on the event object. This is commonly used with form submissions and anchor tags.

```
import React from 'react';

function App() {
  const handleLinkClick = (event) => {
    event.preventDefault();
    console.log('Link clicked!');
  };

  return (
    <div>
      <a href="#" onClick={handleLinkClick}>Click Me</a>
    </div>
  );
}

export default App;
```

4. **Passing Parameters to Event Handlers**:

If you need to pass additional parameters to an event handler, you can use an arrow function or the `bind` method.

```
import React from 'react';

function App() {
  const handleButtonWithParams = (param) => {
    console.log('Button clicked with:', param);
  };

  return (
    <div>
      <button onClick={() => handleButtonWithParams('some parameter')}>
        Click with Params
      </button>
    </div>
  );
}

export default App;
```

State Management:

State management in React involves handling and managing the dynamic data within a component and across your application.

React provides a way to manage and update state using the `useState` hook for functional components or using the `state` property and class methods for class-based components. Additionally, you might use more advanced state management libraries like Redux or MobX for larger and more complex applications. Here's a breakdown of how state management works in React:

1. **Using `useState` Hook (Functional Components)**:

The `useState` hook is the most common way to manage state in modern React functional components.

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

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

  const increment = () => {
    setCount(count + 1);
  };

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

In this example, `count` is the state variable, and `setCount` is the function to update the state. The `useState` hook returns the current state and the updater function.

2. **Using `state` (Class-based Components)**:

In class-based components, state is managed using the `state` property. You can update state using the `setState` method.

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

class Counter extends Component {
```

```
  constructor() {
    super();
    this.state = {
      count: 0
    };
  }

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

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

Fetch() Api:

The `fetch()` API is a modern way to make network requests (HTTP requests) in JavaScript, including in React applications. It provides a standardized and native way to perform asynchronous requests and handle responses. The `fetch()` API returns a `Promise` that resolves to the `Response` representing the response to the request.

Here's a breakdown of how the `fetch()` API works in React:

1. **Making a GET Request**:

To make a GET request using the `fetch()` API, you provide the URL of the resource you want to fetch.

```
fetch('https://api.example.com/data')
  .then(response => {
    // Handle the response
  })
  .catch(error => {
    // Handle errors
  });
```

2. **Handling the Response**:

The `then()` method is used to handle the response from the request. Inside this method, you can parse the response using methods like `.json()` for JSON data, `.text()` for text data, and others depending on the content type of the response.

```
fetch('https://api.example.com/data')
  .then(response => response.json())
  .then(data => {
    // Handle the parsed data
  })
  .catch(error => {
    // Handle errors
  });
```

3. **Error Handling**:

You can use the `.catch()` method to handle errors that might occur during the request.

```
fetch('https://api.example.com/data')
```

```
  .then(response => response.json())
  .then(data => {
    // Handle the parsed data
  })
  .catch(error => {
    // Handle errors
  });
```

4. **Sending Data and Customizing Requests**:

For non-GET requests, you can provide additional options in the `fetch()` call, including specifying the HTTP method, headers, and the body of the request.

```
fetch('https://api.example.com/data', {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json',
    // Other headers if needed
  },
  body: JSON.stringify({ key: 'value' })
})
  .then(response => response.json())
  .then(data => {
    // Handle the response data
  })
  .catch(error => {
    // Handle errors
  });
```

The `fetch()` API provides a low-level way to work with HTTP requests. However, in more complex scenarios, you might prefer to use third-party libraries like `axios` for additional features, such as request cancellation, interceptors, and a more user-friendly API.

Example:

Here's an example of how to use the `fetch()` API to make an HTTP GET request in a React component:

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

function App() {
  const [data, setData] = useState([]);

  useEffect(() => {
    // Fetch data when the component mounts
    fetchData();
  }, []);

  const fetchData = async () => {
    try {
      const response = await fetch('https://jsonplaceholder.typicode.com/posts');
      if (!response.ok) {
        throw new Error('Network response was not ok');
      }
      const jsonData = await response.json();
      setData(jsonData);
    } catch (error) {
      console.error('Error fetching data:', error);
    }
  };

  return (
    <div>
      <h1>Fetch API Example</h1>
```

```
      <ul>
        {data.map(item => (
          <li key={item.id}>{item.title}</li>
        ))}
      </ul>
    </div>
  );
}

export default App;
```

In this example:

- We're using the `useState` and `useEffect` hooks to manage the state and fetch data.
- The `useEffect` hook fetches data when the component mounts.
- The `fetchData` function is defined as an `async` function that makes an HTTP GET request using the `fetch()` API.
- We're using `await` to handle the asynchronous nature of the request and response.
- We're handling potential errors by checking the response status and catching errors using a `try` and `catch` block.

Please note that this example uses the [JSONPlaceholder](https://jsonplaceholder.typicode.com/) API, which provides fake data for testing purposes. You can replace the API URL with your own API endpoint.

Another Example:

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

const DataFetching = () => {
  const [data, setData] = useState([]);
  const [loading, setLoading] = useState(true);
```

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

useEffect(() => {
  // Example API endpoint
  const apiUrl = 'https://jsonplaceholder.typicode.com/posts';

  fetch(apiUrl)
    .then(response => {
      if (!response.ok) {
        throw new Error('Network response was not ok');
      }
      return response.json();
    })
    .then(data => {
      setData(data);
      setLoading(false);
    })
    .catch(error => {
      setError(error);
      setLoading(false);
    });
}, []);

if (loading) return <p>Loading...</p>;
if (error) return <p>Error: {error.message}</p>;

return (
  <div>
    <h2>Data Fetching Example</h2>
```

```
    <ul>
      {data.map(item => (
        <li key={item.id}>
          <h3>{item.title}</h3>
          <p>{item.body}</p>
        </li>
      ))}
    </ul>
  </div>
 );
};

export default DataFetching;
```

Rendering List and Portals

Rendering lists and using portals are two important concepts in React. Let's explore how to render lists and use portals in React applications.

1. **Rendering Lists**:

Rendering lists is a common task in web development. In React, you can use the `.map()` function to loop through an array and render components for each item in the array. Here's an example of rendering a list of items:

```
import React from 'react';

function ListExample() {
  const items = ['Item 1', 'Item 2', 'Item 3'];

  return (
```

```
    <div>
      <h1>List Example</h1>
      <ul>
        {items.map((item, index) => (
          <li key={index}>{item}</li>
        ))}
      </ul>
    </div>
  );
}

export default ListExample;
```

In this example, the `.map()` function is used to iterate through the `items` array and render a `<li>` element for each item. The `key` attribute is important to provide a unique identifier for each list item, helping React efficiently update the DOM.

2. **Portals**:

Portals allow you to render a React component into a different part of the DOM, outside the parent hierarchy. This can be useful for rendering modals, tooltips, and other UI elements that need to "break out" of the normal component tree.

Here's an example of using a portal to render a modal dialog outside the normal component hierarchy:

Modal.js:

```
import React from 'react'
import ReactDOM from 'react-dom';
```

```
export const Modal = ({ isOpen, onClose, children }) => {
   if (!isOpen) return null;
   return ReactDOM.createPortal(
      <div className="modal">
         <div className="modal-content">
            <button onClick={onClose} className="modal-close">
               &times;
            </button>
            {children}
         </div>
      </div>,
      document.getElementById('modal-root')
   );
}
```

App.js:

```
import logo from './logo.svg';
import React, { useState } from 'react';
import './App.css';
import { Modal } from './Modal';

function App() {
  const [isModalOpen, setIsModalOpen] = useState(false);

  const openModal = () => setIsModalOpen(true);
  const closeModal = () => setIsModalOpen(false);
  return (
    <div className="App">
      <header className="App-header">
```

```
      <h1>Portal Example</h1>
      <button onClick={openModal}>Open Modal</button>
      <Modal isOpen={isModalOpen} onClose={closeModal}>
        <h2>Modal Content</h2>
        <p>This is a simple modal example using portals in React.</p>
      </Modal>
    </header>
  </div>
 );
}

export default App;
```

css for modal:

```
/* Modal background overlay */
.modal {
  position: fixed;
  top: 0;
  left: 0;
  right: 0;
  bottom: 0;
  background: rgba(0, 0, 0, 0.7);
  /* Semi-transparent black background */
  display: flex;
  align-items: center;
  /* Center modal vertically */
  justify-content: center;
  /* Center modal horizontally */
```

```
  z-index: 1000;
  /* High z-index to ensure it appears above other content */
}

/* Modal content */
.modal-content {
  background: white;
  /* White background for modal content */
  padding: 20px;
  /* Padding inside modal */
  border-radius: 4px;
  /* Rounded corners */
  max-width: 500px;
  /* Max width of modal */
  width: 100%;
  /* Ensure it does not exceed the width */
  box-shadow: 0 2px 10px rgba(0, 0, 0, 0.1);
  /* Subtle shadow for depth */
}
```

Add modal-root in public/index.html file:

```
<div id="modal-root"></div>
```

In this example, the `Modal` component uses `ReactDOM.createPortal()` to render its content into a separate DOM element with the ID `'modal-root'`. This allows the modal to appear on top of the rest of the page content.

Keep in mind that portals should be used sparingly and only when necessary, as they can make the component structure less clear and can lead to unexpected behavior if misused.

These examples demonstrate how to render lists and use portals in React applications. They are fundamental concepts that help you create dynamic and flexible user interfaces.