MERN: UNIT-3 Advanced Topics and Real World Project

Prepared By,

M.Gouthamm, Asst.Prof, CSE, MRUH

Advanced Topics and Real World Project

Topics Covered

1. **Forms and Form Handling**
2. **React and Http Requests**
3. **React and Authentication**

Forms

- Forms are a crucial part of React web applications. They allow users to enter and upload information directly in components from a login screen to a checkout page.
- As the majority of React applications are single-page apps or web apps that load one page dynamically displaying new information, you will not send the information to a server directly from the form.
- You will instead capture and submit or show the information in the form on the client-side using an additional JavaScript file.

React Forms

- In React Forms, all the form data is stored in the React's component state, so it can handle the form submission and retrieve data that the user entered.
- React forms are used to interact with the user and provides additional functionality such as preventing the default behavior of the form which refreshes the browser after the form is submitted.
- In React, forms are created using the HTML <form> tag just like in regular HTML.

```
import React from "react";

function App() {

  return <form>/* ... */</form>;

}
```

Forms

- Forms play an essential role in modern web applications. They enable users to share information, complete tasks and provide feedback.

- Without forms, many of the tasks that we take for granted on the web, such as logging in, signing up, or making purchases, would not be possible.

- As such, learning how to create effective and user-friendly forms is essential for developers looking to build engaging and interactive web applications.

- With its extensive collection of built-in hooks, React provides several features and techniques for creating and managing forms, including state management, event handling, and form validation.

ReactJS

Forms

Adding Forms in React

Forms in React can be easily added as a simple react element. Here are some examples

React Forms Text Input Example:

This example displays the text input value on the console window when the React onChange event triggers.

```
import React from "react";

const FormsComponent=()=>{

    const HandleInputChange=(event)=>{
        console.log(event.target.value);
    }
```

Contd….

Forms

```
return(
        <div>
            <h5> FormsComponent Section......</h5>

            <div className="card">
                <form>
                    <div className="card-title"> Login Form  </div>
                    <div className="">
                        <label>UserName:</label>
                        <input type="text" placeholder="Enter Username"
onChange={HandleInputChange}/>
                    </div>
                    <div className="card-text">
                        <label>Password:</label>
                        <input type="password" placeholder="Enter Password"
onChange={HandleInputChange}/>
                    </div>
                    <div className="card-text">
                        <button type="button" className="btn btn-success">Submit</button>
                    </div>

                </form>
            </div>

        </div>
    );
}
export default FormsComponent;
```

Forms

- In React, there are two ways of handling form data:

1. **Controlled Components**
2. **Uncontrolled Components**

Controlled Components

- In simple HTML elements like input tags, the value of the input field is changed whenever the user type.
- But, In React, whatever the value the user types we save it in state and pass the same value to the input tag as its value, so here DOM does not change its value, it is controlled by react state.
- In this approach, form data is handled by React through the use of hooks such as the **useState** hook.

Uncontrolled Components:

- Form data is handled by the Document Object Model (DOM) rather than by React.
- The DOM maintains the state of form data and updates it based on user input.

ReactJS

React Forms: Controlled

Handling React Forms

In HTML the HTML DOM handles the input data but in react the values are stored in state variable and form data is handled by the components.

React Form Handling Example:

This example shows updating the value of inputValue each time user changes the value in the input field by calling the setState() function.

Controlled Components in React

In React, a controlled component is a component where form elements derive their value from a React state.

When a component is controlled, the value of form elements is stored in a state, and any changes made to the value are immediately reflected in the state.

To create a controlled component, you need to use the value prop to set the value of form elements and the onChange event to handle changes made to the value.

The value prop sets the initial value of a form element, while the onChange event is triggered whenever the value of a form element changes. Inside the onChange event, you need to update the state with the new value using a state update function.

React Forms: Controlled

Example - 1: **Controlled Components**

```
import {useState} from 'react';
 export default function  ControlledComponent()  {
      const  [inputValue, setInputValue] =  useState('');

      const  handleChange = (event) => {
           setInputValue(event.target.value);
      };
return  (
<form>
      <label>Input Value:
      <input  type="text"  value={inputValue} onChange={handleChange} />
      </label>
      <p>Input Value: {inputValue}</p>
</div>
)};
```

React Forms: Uncontrolled

unControlled Components in React

Uncontrolled components in React refer to form elements whose state is not managed by React. Instead, their state is handled by the browser's DOM.

For instance, let's say you have a form that consists of a text input field, a select box, and a checkbox. In a controlled component, you would create a state for each form element and write event handlers to update the state whenever the user interacts with any of the form elements.

In contrast, an uncontrolled component allows the browser to handle the form elements' state. When a user enters text into a text input field or selects an option from a select box, the browser updates the DOM's state for that element automatically.

To get the value of an uncontrolled form element, you can use a feature called "ref". **"Refs" provide a way to access the current value of DOM elements.**

You can create a **"ref" using the useRef hook**, then attach it to the form element you want to access.

This allows you to retrieve the current value of an element at any time, without needing to manage its state in your React component.

React Forms: Uncontrolled

Here's an example of an uncontrolled component:

```
import { useRef } from "react";

export default function Uncontrolled() {
  const selectRef = useRef(null);
  const checkboxRef = useRef(null);
  const inputRef = useRef(null);

  function handleSubmit(event) {
    event.preventDefault();
    console.log("Input value:", inputRef.current.value);
    console.log("Select value:", selectRef.current.value);
    console.log("Checkbox value:", checkboxRef.current.checked);
  }
```

React Forms: Uncontrolled

```
return (
    <form onSubmit={handleSubmit}>
      <label>  <p>Name:</p>
        <input ref={inputRef} type="text" />
      </label>
      <label>  <p>Favorite color:</p>
        <select ref={selectRef}>
          <option value="red">Red</option>
          <option value="green">Green</option>
          <option value="blue">Blue</option>
        </select>
      </label>
      <label>  Do you like React?
        <input type="checkbox" ref={checkboxRef} />
      </label>
      <button type="submit">Submit</button>
    </form>
  );
}
```

React and Http Requests

- When developing web applications - we routinely access resources hosted on a server. Asking for, sending, or performing other operations on resources is accomplished through HTTP requests.

- These requests are sent from the client to a host on a server. When making HTTP request, the client employs a URL (Uniform Resource Locator) and a payload that contains the necessary information to access the server resources.

- The five commonplace HTTP request methods are **GET, PUT, POST, PATCH, and DELETE**. These methods allow us to perform standard CRUD operations.

Http Requests: GET

What is GET Request?

- Get method is an HTTP request method which aims to get resources from a server (here specifically API). It is a basic method which is going to help us to get data from server.

- **Axios and the Fetch API** are the two main methods for making HTTP requests.

- The Fetch API is a built-in promise-based JavaScript module for retrieving resources from a server or an API endpoint.

- Axios is a promise-based HTTP client library that makes it simple to send asynchronous HTTP requests to REST endpoints. This is an external library that must be installed.

How To Perform GET HTTP Request in React's Functional Components

- A functional component is created when a JavaScript function (either normal or ES6) returns a React element (JSX). We use React hooks when working with functional components, which are very important when performing GET requests.
- We'll use the useState() and useEffect() hooks. Because the useEffect() hook renders immediately when the app mounts we always perform GET requests within it, and we use the useState() hook to store our data/response.

Http Requests: GET

GET HTTP Request in React JS : Fetch API or Fetch()

The fetch() method is one of the common method to use in the ReactJS. In React, the fetch() method accepts **one mandatory argument - the URL** to the resource we want to fetch, **as well as an optional argument** that indicates the request method.

The default value for **that optional argument is GET**, so it is not necessary to set it when making a GET request.

Then Fetch API returns a Promise, so we can use the **then() and catch() methods** to handle success or failure with the data. Here **response.json()** is used to convert raw data to JSON formatted data.

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

Http Requests: GET

- Here is a basic example for the fetch method, As you can see we called the API in useEffect which aims to call the API before application render.
- Also, we used a useState to store data after every successful call. To store data from API call, we ideally use useState hooks.

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

const App = () => {
   const [posts, setPosts] = useState([]);

   useEffect(() => {
      fetch('https://jsonplaceholder.typicode.com/posts')
         .then((res) => res.json())
         .then((data) => {
            console.log(data);
            setPosts(data);
         })
         .catch((err) => {
            console.log(err.message);
         });
   }, []);

   return (
      // ... consume here
   );
};
```

Http Requests: GET

HTTP Get Method in React JS: axios()

- Axios is another way to work with get method, It looks cleaner than fetch() method, right? **Axios is a promise-based HTTP** client library that makes it simple to send asynchronous HTTP requests to REST endpoints.
- **Axios is an external library**, or you can say external package.
  So you need to install it in your React application using this line of code in terminal or cmd. **npm install axios**
- To use get method in **axios, we use axios.get()** which takes a mandatory argument (URL).
- Axios is pretty different than fetch, because axios provides more powerful and flexible APIs than fetch method.

```
import axios from 'axios';

axios.get('https://api.example.com/data')
  .then(response => {
    // Do something with the data
  })
  .catch(error => {
    // Handle the error
  });
```

Http Requests: GET

HTTP Get Method in React JS: Async/await

Async/await method is not a different way to get data from API, but you can say it is the way to use fetch and axios method. Here are basic examples:

Using Fetch API

```
function ExampleComponent() {
  const [data, setData] = useState(null);

  useEffect(() => {
    async function fetchData() {
      try {
        const response = await fetch('https://api.example.com/data');
        const data = await response.json();
        setData(data);
      } catch (error) {
        console.error(error);
      }
    }
    fetchData();
  }, []);

  return (
    <div>
      {data ? <p>{data.example}</p> : <p>Loading...</p>}
    </div>
  );
}
```

ReactJS

Http Requests: GET

Using axios

```
import axios from 'axios';

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

  useEffect(() => {
    async function fetchData() {
      try {
        const response = await axios.get('https://api.example.com/data');
        setData(response.data);
      } catch (error) {
        console.error(error);
      }
    }
    fetchData();
  }, []);

  return (
    <div>
      {data ? <p>{data.example}</p> : <p>Loading...</p>}
    </div>
  );
}
```

Http Requests: GET

Using axios

```
import axios from 'axios';

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

  useEffect(() => {
    async function fetchData() {
      try {
        const response = await axios.get('https://api.example.com/data');
        setData(response.data);
      } catch (error) {
        console.error(error);
      }
    }
    fetchData();
  }, []);

  return (
    <div>
      {data ? <p>{data.example}</p> : <p>Loading...</p>}
    </div>
  );
}
```

ReactJS

Http Requests: POST

- As the name implies, POST requests are used to post data to an endpoint - which then typically processes it and saves it in a database.

- This data could come from a form, be saved in an object, or be obtained in another way - but it's typically converted into a JSON representation for the REST API to consume.

- Sending HTTP requests with any verb is made simple by the Fetch API (built-in) and libraries such as Axios.

- The Fetch API is a built-in browser method for performing HTTP requests, whereas Axios is an external package we must install in our project before using.

Http Requests: POST

HTTP Post Method in React JS: Fetch()

- The fetch() is a built-in JavaScript function which allows us to make network related requests. It is simple to use, and it doesn't need **any external package** to use.
- It, basically, returns a **promise that resolves** to a response object, which is useful when accessing response data.

- Let's see an example, in the below example we have created a form with an input field and a button. When this form submits, then we call a function in which we have used **fetch() with initial URL of API.**
- Here, the **default method is Get()**, So we need to **mention method as POST**. Which makes a POST request to the specified URL with the data from the input field in the body of the request.

Http Requests: POST

The fetch() function returns a promise, so we need to use await to wait for the response. then we have just parsed the data to JSON and log the result to console.

Example:

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

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

  const handleSubmit = async (e) => {
    e.preventDefault();
    const response = await fetch('https://example.com/api/data', {
      method: 'POST',
      body: JSON.stringify(data),
      headers: {
        'Content-Type': 'application/json'
      }
    });
    const result = await response.json();
    console.log(result);
  }
  return (
    <form onSubmit={handleSubmit}>
      <input type="text" onChange={e => setData({...data, name: e.target.value})} />
      <button type="submit">Submit</button>
    </form>
  );
}
export default Example;
```

Http Requests: POST

HTTP Post Method in React JS: axios()

Axios is another popular library for making network requests in React.

It is basically an external library which need to be imported and install in the React application.

Axios is a promise based library that makes it easy when working with network requests. It is also much powerful and flexible method then fetch as well.

Here is a basic example for Post method with axios:

Now in the example, we have to import axios, in order to work with network requests.

To use **POST method in axios we can use axios.post()**. The **axios.post() method returns a promise**, so we use the await keyword to wait for the response. We then log the response data to the console.

Http Requests: POST

- One more thing to note that in order to use post method, we should use .catch(), in order to perform error handling and it is essential.

```
import axios from 'axios';

function Example() {
    const handleSubmit = async () => {
    try {
      const response = await axios.post('https://example.com/api/data', {
        name: "John Doe"
      });
      console.log(response.data);
    } catch (error) {
      console.error(error);
    }
  }

  return (
    <form onSubmit={handleSubmit}>
      <button type="submit">Submit</button>
    </form>
  );
}
export default Example;
```

Http Requests: PUT

- A PUT request is one of the four fundamental HTTP methods, along with GET, POST, and DELETE.
- A PUT request is used to update or replace an existing resource on a server. The request body usually contains the data that you want to update or replace.
- A PUT request is different from a POST request, which is used to create a new resource on a server. A POST request does not require an id or a specific URL, while a PUT request does. A POST request may result in a duplicate resource if it is repeated, while a PUT request is idempotent, meaning that it will not change the state of the server if it is repeated.
- A PUT request is also different from a PATCH request, which is used to partially update an existing resource on a server. A **PATCH request only sends the data that needs to be changed**, while a **PUT request sends the entire data that replaces the original resource**.
- A PATCH request is more efficient and flexible, but not all servers support it.

Http Requests: PUT

How to Make a Simple PUT Request with the Fetch API

Making a PUT request with the Fetch API is very easy and straightforward. You just need to call the fetch() method, pass in the URL to fetch from, and provide an options object with the request method, headers, and body. Then, you can handle the response when it resolves.

Example:

```
// Define the data to update
const data = {
  title: "How to Use Fetch to Make PUT Requests in React",
  content: "In this blog post, we will learn how to use the Fetch API to make PUT requests in React..."
};
// Define the request options
const requestOptions = {
  method: "PUT", // Specify the request method
  headers: { "Content-Type": "application/json" }, // Specify the content type
  body: JSON.stringify(data) // Send the data in JSON format
};
// Make the request
fetch("https://example.com/api/posts/1", requestOptions)
  .then(response => response.json()) // Parse the response as JSON
  .then(data => console.log(data)) // Do something with the data
  .catch(error => console.error(error)); // Handle errors
```

Http Requests: PUT

How to Use React Hooks to Make a PUT Request in a Functional Component

If you are using React, you may want to make a PUT request in a functional component and update the component state when the data is received. This allows you to fetch data from an API and display it in your UI.

To do that, you can use React hooks, which are special functions that let you use state and other React features in functional components. The most commonly used hooks are useState and useEffect.

The useState hook lets you declare a state variable and a function to update it. The initial state is passed as an argument to the hook, and the returned value is an array with the state variable and the updater function. You can use the state variable to store the data that you want to display in the UI, and the updater function to update the state when the data changes.

The useEffect hook lets you perform side effects, such as making HTTP requests, in functional components. The first argument is a function that runs after the component renders, and the second argument is an array of dependencies that determines when the effect runs. If you pass an empty array, the effect will only run once when the component mounts

Http Requests: PUT

Example:

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

function App() {
  // Declare a state variable to store the data
  const [data, setData] = useState(null);

  // Define the data to update
  const newData = {
    title: "How to Use Fetch to Make PUT Requests in React",
     content: "In this blog post, we will learn how to use the Fetch API to make PUT requests in
React..."
  };

  // Define the request options
  const requestOptions = {
    method: "PUT",
    headers: { "Content-Type": "application/json" },
    body: JSON.stringify(newData)
  };
```

Contd..

Http Requests: PUT

Example:

```
// Use the useEffect hook to make the request
  useEffect(() => {
    // Make the request
    fetch("https://example.com/api/posts/1", requestOptions)
      .then(response => response.json()) // Parse the response as JSON
      .then(data => setData(data)) // Update the state with the data
      .catch(error => console.error(error)); // Handle errors
  }, []); // Pass an empty array as the second argument to run the effect only once
  // Return the JSX to render
  return (
    <div className="App">
      <h1>How to Use Fetch to Make PUT Requests in React</h1>
      {data ? ( // Check if the data is available
        <div>
          <h2>{data.title}</h2> // Display the data
          <p>{data.content}</p>
        </div>
      ) : (       <p>Loading...</p> // Display a loading message    )}
    </div>
 ); }
```

Http Requests: DELETE

HTTP Delete Method in React JS: Fetch()

The fetch() is a built-in JavaScript function which allows us to make network related requests. It is simple to use, and it doesn't need any external package to use. It, basically, returns a promise that resolves to a response object, which is useful when accessing response data.

Let see a demo example, here we have a URL in fetch method. Notice that this URL pretty much different than normal API URL, because we want to delete specific resource and we are providing that resource's URL. Then after, we need to specify DELETE method in method section. And lastly we are just fetching rest of the data.

```
fetch('https://example.com/api/resources/123', {
  method: 'DELETE'
})
.then(res => res.json())
.then(data => console.log(data))
.catch(error => console.error(error));
```

Http Requests: DELETE

HTTP Delete Method in React JS: axios()

- Axios is another popular library for making network requests in React. It is basically an external library which need to be imported and install in the React application. Axios is a promise based library that makes it easy when working with network requests. It is also much powerful and flexible method then fetch as well. Here is a basic example for Post method with axios:

- Now in the below example, we have to import axios, in order to work with network requests. To use Delete method in axios we can use **axios.delete().** The **axios.delete()** method returns a promise, so we use the await keyword to wait for the response. and these data will automatically gets modified. we don't need to convert it to the JSON format, We then log the response data to the console.

- The Axios method is pretty simple and fewer code to work with, so axios looks much cleaner than fetch() method. One more thing to note that in order to use Delete method, we should use .catch(), in order to perform error handling and it is essential.

```
import axios from 'axios';

axios.delete('https://example.com/api/resources/123')
  .then(res => console.log(res.data))
  .catch(error => console.error(error));
```

React Authentication

Most modern web apps require individuals to verify their identity.

- **Authentication** is the process of verifying the identity of an individual. A user can interact with a web application using multiple actions. Access to specific resources can be restricted by using user levels.

- **Authorization** is the process of controlling user access via assigned roles and privileges.

React Authentication

Authentication

- Authentication is the process of verifying identity. A unique identifier is associated with a user, which is the username or user ID.
- In React applications typically involves managing user sessions, handling login and registration processes, and protecting routes that require authentication.
- Traditionally, we combine **username and password** to authenticate a user. The authentication logic has to be maintained locally. So we will term it local authentication. Apart from local authentication, we can use **OpenID, OAuth, and SAML as authorization providers.**

Local Authentication

The most common authentication technique is using a **username and password.** The usual flow while implementing is the following:

- The user registers using an identifier like a username, email, or mobile.
- The application stores user credentials in the database.
- The application sends a verification email or message to validate the registration.
- After successful registration, the user enters credentials for logging in.
- On successful authentication, the user is allowed access to specific resources.
- The user state is maintained via **Sessions or JWT.**

ReactJS

React Authentication

- Here's a general overview of how you can implement authentication in a React application:

1. Authentication Flow

- The user visits the application and attempts to access protected resources. If the user is not authenticated, they are redirected to the login page. After successful authentication, the user is redirected back to the originally requested page or to a default landing page.

a) **Login Component**

b) **Registration Component**

- **Logout Component**

2. Managing User Sessions:

- Once the user logs in successfully, you need to manage their session state. You can store session data in various ways:
- **Using Cookies:** Store a session token in a cookie that is sent with each request to identify the user's session.
- **Using Local Storage or Session Storage:** Store session data in the browser's local or session storage.

a) **Using JWT (JSON Web Tokens):** Upon successful authentication, the server issues a JWT containing user information, which is then stored in the client-side storage.

React Authentication

3. Protected Routes:

Define routes that require authentication, and ensure that unauthenticated users cannot access them.

You can achieve this by creating a higher-order component (HOC) or using React Router's Private Route component to wrap the protected routes.

4. Communicating with the Backend:

Your React application needs to communicate with a backend server for authentication and authorization.

Implement API endpoints on the server for user authentication, registration, and logout.

Use libraries like Axios or Fetch to make HTTP requests from your React components to these endpoints

5. Error Handling:

Handle authentication errors gracefully by displaying meaningful error messages to the

user. For example, if the user enters invalid credentials, inform them appropriately.

6. Redirects:

After successful login or logout, redirect the user to the appropriate page.

If the user tries to access a protected route without authentication, redirect them to the login page.

7. Session Management:

Implement features like session expiration and automatic logout to enhance security.

Refresh tokens can be used to obtain new access tokens without requiring the user to log in again after the access token expires.

End Of UNIT-3