**4.ReactJs-Hol**

**Objectives**

* Explain the need and Benefits of component life cycle
* Identify various life cycle hook methods
* List the sequence of steps in rendering a component

In this hands-on lab, you will learn how to:

* Implement componentDidMount() hook
* Implementing componentDidCatch() life cycle hook.

## **Prerequisites**

The following is required to complete this hands-on lab:

* Node.js
* NPM
* Visual Studio Code

## **Notes**

Estimated time to complete this lab: **60 minutes.**

1. Create a new react application using *create-react-app* tool with the name as “blogapp”
2. Open the application using VS Code
3. Create a new file named as **Post.js** in **src folder** with following properties



Figure 2: Post class

1. Create a new class based component named as **Posts** inside **Posts.js** file



Figure 3: Posts Component

1. Initialize the component with a list of Post in state of the component using the constructor
2. Create a new method in component with the name as **loadPosts()** which will be responsible for using Fetch API and assign it to the component state created earlier. To get the posts use the url (<https://jsonplaceholder.typicode.com/posts>)



Figure 4: loadPosts() method

1. Implement the **componentDidMount()** hook to make calls to **loadPosts()** which will fetch the posts



Figure 5: componentDidMount() hook

1. Implement the **render()** which will display the title and post of posts in html page using heading and paragraphs respectively.



Figure 6: render() method

1. Define a **componentDidCatch()** method which will be responsible for displaying any error happing in the component as alert messages.



Figure 7: componentDidCatch() hook

1. Add the Posts component to App component.
2. Build and Run the application using *npm start* command.

# ReactJS Hands-On Lab: Blog Application

This document provides the complete solution for the ReactJS hands-on lab. The goal of this lab is to build a simple application that fetches and displays blog posts, demonstrating the use of React component lifecycle methods.

### Overview of the Solution

The solution is divided into three main files, as is standard for a React application:

1. Post.js: A simple class that defines the structure of a single post object. This acts as a data model.
2. Posts.js: The main React component that handles fetching the post data from an API, managing the application's state, and rendering the list of posts. This is where the core logic and lifecycle methods (componentDidMount and componentDidCatch) are implemented.
3. App.js: The root component of the application, which renders the Posts component.

Below, you will find the code for each of these files.

### How to Use This Solution

1. **Set up your environment:** Make sure you have Node.js and NPM installed.
2. **Create a new React App:** Open your terminal or command prompt and run the following command to create a new React project named blogapp:
3. npx create-react-app blogapp
4. **Navigate to the project directory:**
5. cd blogapp
6. **Replace the code:** Open the src folder inside your blogapp project. Copy the code from the blocks below and paste it into the corresponding files (Post.js, Posts.js, and App.js). You will need to create Post.js and Posts.js.
7. **Run the application:** In your terminal, run the following command:
8. npm start

This will start the development server, and you can view your application in your web browser at http://localhost:3000.

### Code Files

Here is the complete code for each file required by the assignment.

Src/Post.js

// src/Post.js

/\*\*

 \* This class represents the data structure for a single blog post.

 \* It's not a React component, but a plain JavaScript class used as a model.

 \* Using models helps in maintaining a clear and consistent data structure throughout the application.

 \*/

class Post {

  constructor(id, title, body) {

    this.id = id;

    this.title = title;

    this.body = body;

  }

}

export default Post;

src/Posts.js

// src/Posts.js

import React from 'react';

// We are not directly using the Post model class in this file for instantiation,

// as the data from the API already matches the desired structure.

// However, in a larger application, you might import it for type checking or creating new posts.

// import Post from './Post';

/\*\*

 \* The Posts component is a class-based component responsible for:

 \* 1. Fetching a list of posts from a public API.

 \* 2. Storing the posts in its local state.

 \* 3. Handling loading and error states.

 \* 4. Rendering the list of posts to the UI.

 \*/

class Posts extends React.Component {

  constructor(props) {

    super(props);

    // Initialize the component's state.

    // - posts: an array to hold the blog post data.

    // - error: an object to hold any error information if fetching fails.

    // - isLoading: a boolean to track the loading state.

    this.state = {

      posts: [],

      error: null,

      isLoading: true,

    };

    console.log('[Posts.js] Constructor called');

  }

  /\*\*

   \* This is an async method to fetch post data from the JSONPlaceholder API.

   \* It uses the modern Fetch API with async/await for cleaner asynchronous code.

   \*/

  async loadPosts() {

    try {

      const response = await fetch('https://jsonplaceholder.typicode.com/posts');

      // Check if the network response was successful.

      if (!response.ok) {

        throw new Error(`HTTP error! status: ${response.status}`);

      }

      const data = await response.json();

      // Update the state with the fetched posts and set isLoading to false.

      this.setState({ posts: data, isLoading: false });

    } catch (e) {

      // If an error occurs during fetching, update the state with the error info.

      this.setState({ error: e, isLoading: false });

      console.error("Failed to load posts:", e);

    }

  }

  /\*\*

   \* componentDidMount is a lifecycle hook that runs after the component output has been

   \* rendered to the DOM. This is the perfect place to make API calls.

   \*/

  componentDidMount() {

    console.log('[Posts.js] componentDidMount called');

    // We call loadPosts() here to fetch the data as soon as the component is mounted.

    this.loadPosts();

  }

  /\*\*

   \* componentDidCatch is a lifecycle hook that gets called when an error is thrown

   \* by a descendant component. It's used for implementing Error Boundaries.

   \* Note: It does not catch errors within the same component, only in its children.

   \* @param {Error} error - The error that was thrown.

   \* @param {object} info - An object with a componentStack key containing information about which component threw the error.

   \*/

  componentDidCatch(error, info) {

    console.log('[Posts.js] componentDidCatch called');

    // Update state to display an error message to the user.

    this.setState({ error: { message: "An error occurred in a child component.", details: error, info } });

    console.error("Caught an error in a child component: ", error, info);

  }

  /\*\*

   \* The render method is responsible for describing the UI.

   \* It is called during the mounting and updating phases.

   \*/

  render() {

    console.log('[Posts.js] Render method called');

    const { posts, error, isLoading } = this.state;

    // Display a loading message while data is being fetched.

    if (isLoading) {

      return <div>Loading posts...</div>;

    }

    // Display an error message if the API call failed or a child component crashed.

    if (error) {

      return <div style={{ color: 'red' }}>Error: {error.message}</div>;

    }

    // If data is loaded successfully, render the list of posts.

    return (

      <div>

        <h1>Blog Posts</h1>

        {posts.map(post => (

          <article key={post.id} style={{ borderBottom: '1px solid #ccc', marginBottom: '20px', paddingBottom: '20px' }}>

            <h2 style={{textTransform: 'capitalize'}}>{post.title}</h2>

            <p>{post.body}</p>

          </article>

        ))}

      </div>

    );

  }

}

export default Posts;

src/App.js

// src/App.js

import React from 'react';

import Posts from './Posts'; // Import the Posts component

import './App.css';

/\*\*

 \* This is the main App component, which acts as the root of our application.

 \* It's responsible for the overall layout and for rendering the primary components.

 \*/

function App() {

  return (

    <div className="App">

      <header className="App-header">

        {/\* The Posts component is rendered here. \*/}

        <Posts />

      </header>

    </div>

  );

}

export default App;