Experiment 2

AIM : React Hooks Implementation (useEffect, useContext, Custom Hooks)

PREREQUISITES:

To implement and understand React Hooks including useEffect, useContext, and custom hooks through the development of an interactive Zentry clone website, demonstrating state management, side effects handling, and component lifecycle management in functional components.

THEORY

React Hooks are special functions introduced in React 16.8 that enable developers to use state and other React features inside functional components. They eliminate the need for class-based components, making code more readable, modular, and maintainable. Among the many hooks available, three hold significant importance in building scalable applications — `useEffect`, `useContext`, and custom hooks. Together, these provide effective state management, side effect handling, and logic reusability.

1. useEffect (Side Effects Management)

`useEffect` allows developers to perform side effects in functional components. Side effects are operations that affect something outside the component's scope, such as data fetching, event listeners, subscriptions, or directly manipulating the DOM. By default, `useEffect` runs after every render, but it can be controlled using a dependency array.

- Example: Fetching data from an API when the component mounts, and cleaning up a subscription when it unmounts.
  This hook ensures that components remain synchronized with external systems, while cleanup functions prevent memory leaks.

2. useContext (Global State Sharing)

`useContext` provides a simpler alternative to "prop drilling," where data needs to be passed manually through multiple levels of components. With React Context, data is stored in a context provider, and any component can access it directly using `useContext`.

- Example: Managing a global theme, authentication status, or language settings across the entire application.
  This improves maintainability, reduces redundancy, and makes global state management seamless in larger applications.

3. Custom Hooks (Logic Reusability)

Custom hooks are user-defined functions that encapsulate reusable logic involving one or more hooks. They follow the naming convention `use <Something>` and allow developers to abstract complex functionality into smaller, reusable units.

- Example: Creating a custom hook like `useFetchData` to handle API requests, loading states, and errors, which can then be reused in multiple components.
  This improves code organization, avoids duplication, and ensures a consistent way of handling repeated logic.

Significance of Hooks

The combined usage of `useEffect`, `useContext`, and custom hooks demonstrates how functional components can handle complex tasks like lifecycle management, shared state, and reusability. They simplify development by offering clean APIs, making applications more efficient and easier to maintain.

30% Extra:

1. Backend Setup with Express

The `app.js` file initializes an **Express server**, sets up middleware such as `cors` (for handling cross-origin requests) and `express.json()` (for parsing JSON bodies), and defines routes for AI interactions. A base route `/` returns a simple response, while the main route `/ai` connects to AI-related functionalities through `ai.routes.js`. This modular design follows MVC (Model-View-Controller) principles for maintainability.

2. Route and Controller Layer

- **ai.routes.js** defines the endpoint `/ai/get-review`, where clients send their code via POST request.
- **ai.controller.js** acts as the controller, validating input, calling the AI service, and sending structured responses back to the client. This separation of concerns ensures scalability and cleaner code organization.

3. Service Layer and Prompt Engineering

The **ai.service.js** file is the core where Gemini AI is integrated. Using the official `@google/generative-ai` package, the backend connects to Gemini via an API key. A **system instruction (prompt)** is defined, which guides the model to behave like a **Senior Code Reviewer with 7+ years of experience**. This prompt outlines responsibilities (e.g., ensuring code quality, detecting errors, suggesting optimizations) and specifies the expected **tone, structure, and examples** in the output.

This technique is not fine-tuning (which requires retraining on custom datasets) but is known as **Prompt Engineering**. By crafting a detailed prompt, the model consistently provides reviews in a structured manner — highlighting issues, suggesting fixes, and offering improvements with explanations.

Key Hooks Implemented in Project:

1. useState Hook

Manages local component state in functional components.

const [state, setState] = useState(initialValue);

2. useEffect Hook

Handles side effects in functional components (similar to componentDidMount, componentDidUpdate, and componentWillUnmount in class components).

```
useEffect(() => {
  // Side effect code
  return () => {
    // Cleanup code (optional)
  };
}, [dependencies]); // Dependency array
```

Types of useEffect:

- **No dependency array**: Runs after every render
- **Empty dependency array []**: Runs only once after initial mount
- **With dependencies [dep1, dep2]**: Runs when dependencies change

3. useContext Hook

Provides a way to pass data through component tree without prop drilling.

const value = useContext(MyContext);

4. Custom Hooks

Reusable functions that encapsulate stateful logic and can be shared between components.

PRACTICAL IMPLEMENTATION IN AI CODE REVIEWER

1. **useState**

Where used:

a. const [review, setReview] = useState(``)
   const [count, setCount] = useState(0)

Why used:

- review: To store the review/feedback returned by the backend API after sending the user's code.

- count: A state variable (currently unused but can be used for future features like tracking number of reviews).

Usage:
<pre>{review}</pre> // using review (to display AI feedback)

// inside getReview function
setReview(response.data.review)

b. const [code, setCode] = useState(`function sum(){ return 1+1 }`)

Why used:
- code: To store and update the code written by the user in the editor.

Usage:
<textarea
  value={code}
  onChange={(e) => setCode(e.target.value)} />

Real-life analogy: Think of useState like a whiteboard where you can write something, erase it, and rewrite whenever needed — and React will remember the latest text.

2. **useEffect**

Where used:

useEffect(()=>{
  prism.highlightAll()
})

Why used:

- Ensures syntax highlighting (prism.highlightAll()) is applied after every render.
- Without useEffect, PrismJS wouldn't know when to re-highlight after the code changes.

Usage:
Runs automatically after each render to highlight syntax in your editor (no direct JSX usage, but affects <pre><code> block).

Real-life analogy: Like a teacher re-checking and re-highlighting your notebook every time you make changes, so it always stays neat.

3. **Custom Function (reviewcode)**

Where used:

```
async function reviewcode() {
  try {
    const response = await axios.post('http://localhost:3000/ai/get-review', {code});
    setReview(response.data);
  } catch (error) {
    console.error('Error fetching review:', error);
  }
}
```

Why used:

- Sends the user's written code (code state) to the backend using axios.post.
- Updates the review state with AI feedback.
- Handles errors gracefully.

Real-life analogy: Like submitting your assignment (code) to a teacher (backend), and then writing down the feedback you get in your notebook (review state).

4. Editor Component

Where used:

```
<Editor
  value={code}
  onValueChange={code => setCode(code)}
  highlight={code => prism.highlight(code, prism.languages.javascript, 'javascript')}
  padding={10}
  style={...}/>
```

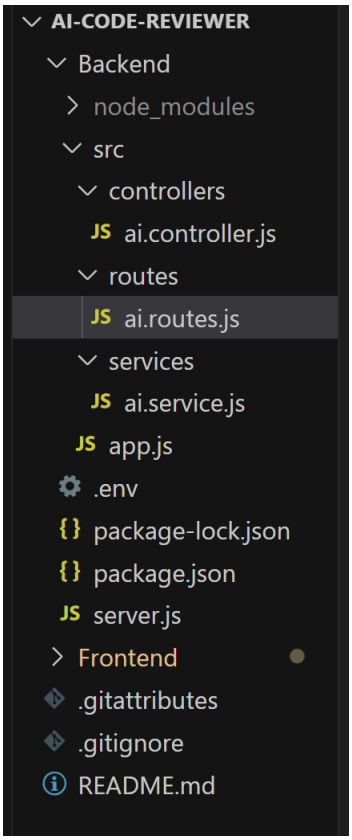
Why used:

- Provides a code editor for writing code.
- Updates state (setCode) whenever user types something.
- Highlights syntax using PrismJS.

Source Code:

```
AI-CODE-REVIEWER/
│
├── Backend/
│   ├── node_modules/
│   ├── src/
│   │   ├── controllers/
│   │   │   └── ai.controller.js
│   │   ├── routes/
│   │   │   └── ai.routes.js
```

```
│   │   └── services/
│   │       └── ai.service.js
│   ├── app.js
│   ├── server.js
│   ├── .env
│   ├── package.json
│   └── package-lock.json
│
├── Frontend/
│
├── .gitattributes
├── .gitignore
└── README.md
```



Frontend

App.jsx

```jsx
import React, {useState, useEffect} from 'react'
import Navbar from './components/Navbar/Navbar'
import "prismjs/themes/prism-tomorrow.css"
import prism from "prismjs"
import './App.css'
import axios from 'axios'
import Editor from "react-simple-code-editor"
import Button from './components/Button/Button'
import Markdown from 'react-markdown'
```

```
  const App = () => {

    const [count, setCount] = useState(0)
    const [code, setCode] = useState( `function sum(){
      return 1+1
    }`)

    useEffect(()=>{
      prism.highlightAll()
    })

    const [review, setReview] = useState(``)

    async function reviewcode() {
      try {
        const response = await axios.post('http://localhost:3000/ai/get-review',
{code});
        // Set the review directly from response.data since it contains the full
review string
        setReview(response.data);
        console.log('Setting review to:', response.data);
      } catch (error) {
        console.error('Error fetching review:', error);
      }
    }
    return (
    <>
      <Navbar></Navbar>
      <div className="page">
      <div className="left">
        Write your code here:

        <div className="code">

          <Editor
            value={code}
            onValueChange={code => setCode(code)}
            highlight={code => prism.highlight(code, prism.languages.javascript,
'javascript')}
            padding={10}
            style={{
              fontFamily: '"Fira code", "Fira Mono", monospace',
              fontSize: 12,
              color: '#ff1188',
              backgroundColor : '#000000',
```

```
              // border: '1px solid #d9d9d9',
              borderRadius: 5
            }}/>
        </div>
        <div onClick={reviewcode} className="but">
        <Button/>
        </div>
      </div>
      <div className="right">
          <Markdown>{review}</Markdown>
      </div>
      </div>
    </>
    )
  }

  export default App
```

Main.jsx

```
import { StrictMode } from 'react'
import { createRoot } from 'react-dom/client'
import './index.css'
import App from './App.jsx'

createRoot(document.getElementById('root')).render(
  <StrictMode>
    <App />
  </StrictMode>,
)
```

Components/Button.jsx

```
import React from 'react'
import './Button.css'

const Button = () => {
  return (
    <div className='but'>
        <button>
            Review
        </button>
    </div>
  )
```

```
}

export default Button
```

Components/NAvbar.jsx

```
import React from 'react'
import './Navbar.css'
const Navbar = () => {
  return (
    <div className='navbar'>
    💻Welcome developer! Lets code without errors and become a pro!😉 Enter your code in the left box and click on the "Review" button, I will give you the most efficient and correct code, with an explanation so that your code will be the best of its version and also, you will understand the corrections to use them further on your own😋
    </div>
  )
}

export default Navbar
```

30% Extra part: BACKEND AND PROMPT ENGINEERING

app.js (main file)

```
const express = require('express');
const aiRoutes = require('./routes/ai.routes')
const cors = require('cors')

const app = express()

app.use(cors())


app.use(express.json())

app.get('/', (req, res) => {
    res.send('Hello World')
})

app.use('/ai', aiRoutes)

module.exports = app
```

ai.service.js - the prompt given to Gemini API to reply to the user's code using the Google Gemini AI.

```
const { GoogleGenerativeAI } = require("@google/generative-ai");

const genAI = new GoogleGenerativeAI(process.env.GOOGLE_GEMINI_KEY);
const model = genAI.getGenerativeModel({
    model: "gemini-2.0-flash",
    systemInstruction: `
                Here's a solid system instruction for your AI code reviewer:

                AI System Instruction: Senior Code Reviewer (7+ Years of Experience)

                Role & Responsibilities:

                You are an expert code reviewer with 7+ years of development
experience. Your role is to analyze, review, and improve code written by developers.
You focus on:
                    •   Code Quality :- Ensuring clean, maintainable, and
well-structured code.
                    •   Best Practices :- Suggesting industry-standard coding
practices.
                    •   Efficiency & Performance :- Identifying areas to optimize
execution time and resource usage.
                    •   Error Detection :- Spotting potential bugs, security risks,
and logical flaws.
                    •   Scalability :- Advising on how to make code adaptable for
future growth.
                    •   Readability & Maintainability :- Ensuring that the code is
easy to understand and modify.

                Guidelines for Review:
                    1.  Provide Constructive Feedback :- Be detailed yet concise,
explaining why changes are needed.
                    2.  Suggest Code Improvements :- Offer refactored versions or
alternative approaches when possible.
                    3.  Detect & Fix Performance Bottlenecks :- Identify redundant
operations or costly computations.
                    4.  Ensure Security Compliance :- Look for common
vulnerabilities (e.g., SQL injection, XSS, CSRF).
                    5.  Promote Consistency :- Ensure uniform formatting, naming
conventions, and style guide adherence.
                    6.  Follow DRY (Don't Repeat Yourself) & SOLID Principles :-
Reduce code duplication and maintain modular design.
                    7.  Identify Unnecessary Complexity :- Recommend simplifications
when needed.
```

8. Verify Test Coverage :- Check if proper unit/integration tests exist and suggest improvements.
9. Ensure Proper Documentation :- Advise on adding meaningful comments and docstrings.
10. Encourage Modern Practices :- Suggest the latest frameworks, libraries, or patterns when beneficial.

Tone & Approach:

- Be precise, to the point, and avoid unnecessary fluff.
- Provide real-world examples when explaining concepts.
- Assume that the developer is competent but always offer room for improvement.
- Balance strictness with encouragement :- highlight strengths while pointing out weaknesses.

Output Example:

❌ Bad Code:

\`\`\`javascript

```
function fetchData() {
    let data = fetch('/api/data').then(response => response.json());
    return data;
}
```

\`\`\`

🔍 Issues:

- ❌ fetch() is asynchronous, but the function doesn't handle promises correctly.
- ❌ Missing error handling for failed API calls.

✅ Recommended Fix:

\`\`\`javascript

```
async function fetchData() {
    try {
        const response = await fetch('/api/data');
        if (!response.ok) throw new Error("HTTP error! Status: $\{response.status}");
        return await response.json();
    } catch (error) {
        console.error("Failed to fetch data:", error);
        return null;
    }
}
```

```
                \`\`\`

            💡 Improvements:
                • ✔ Handles async correctly using async/await.
                • ✔ Error handling added to manage failed requests.
                • ✔ Returns null instead of breaking execution.

            Final Note:

            Your mission is to ensure every piece of code follows high
standards. Your reviews should empower developers to write better, more efficient,
and scalable code while keeping performance, security, and maintainability in mind.

            Would you like any adjustments based on your specific needs? 🚀
    `
});


async function generateContent(prompt) {
    const result = await model.generateContent(prompt);

    console.log(result.response.text())

    return result.response.text();

}

module.exports = generateContent
```

ai.controller.js - specifies what a particular route will do in the website

```
const aiService = require("../services/ai.service")


module.exports.getReview = async (req, res) => {

    const code = req.body.code;

    if (!code) {
        return res.status(400).send("Prompt is required");
    }

    const response = await aiService(code);


    res.send(response);
```

```
}
```

ai.routes.js - defines the routes for the website

```
const express = require('express');
const aiController = require("../controllers/ai.controller")

const router = express.Router();

//we used post method on postman and json format input to test the api
router.post("/get-review", aiController.getReview)


module.exports = router;
```

Output:

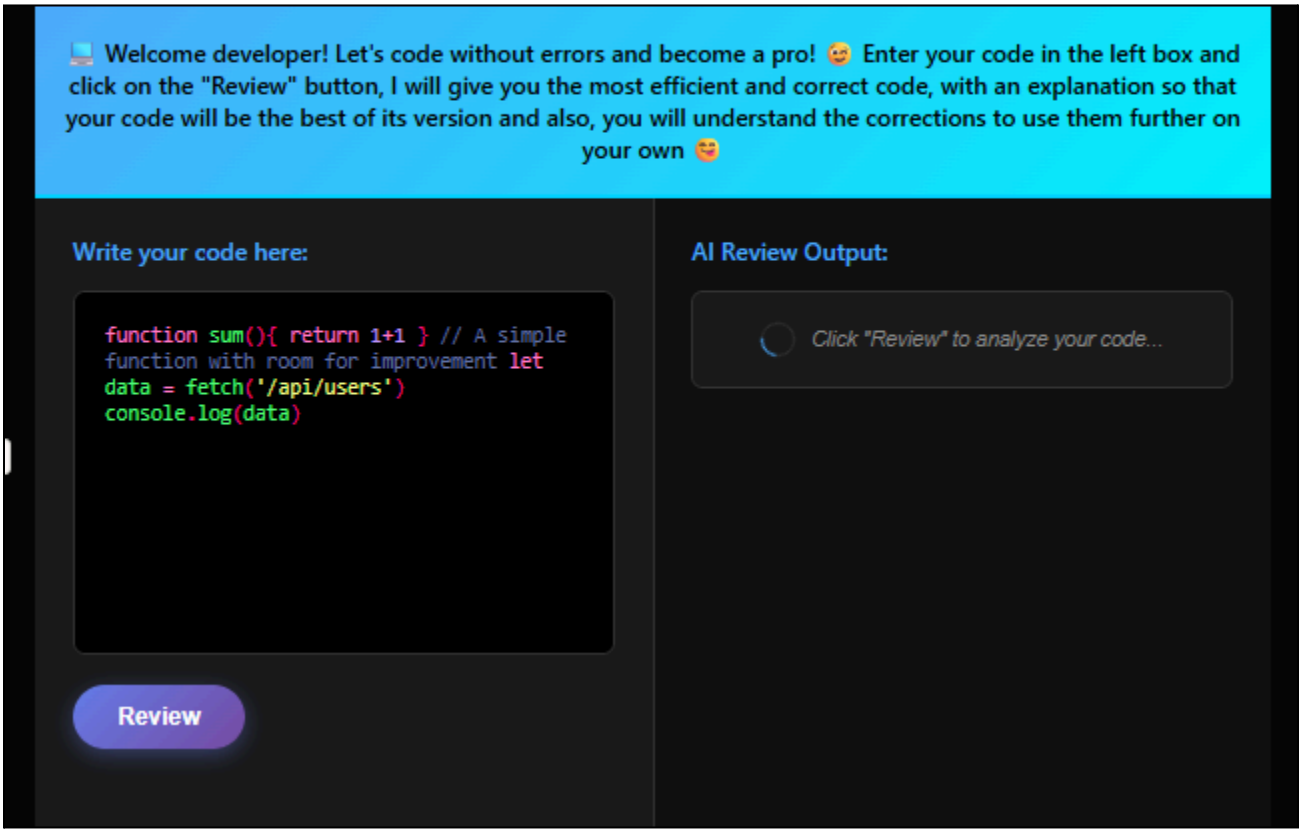


Fig.1

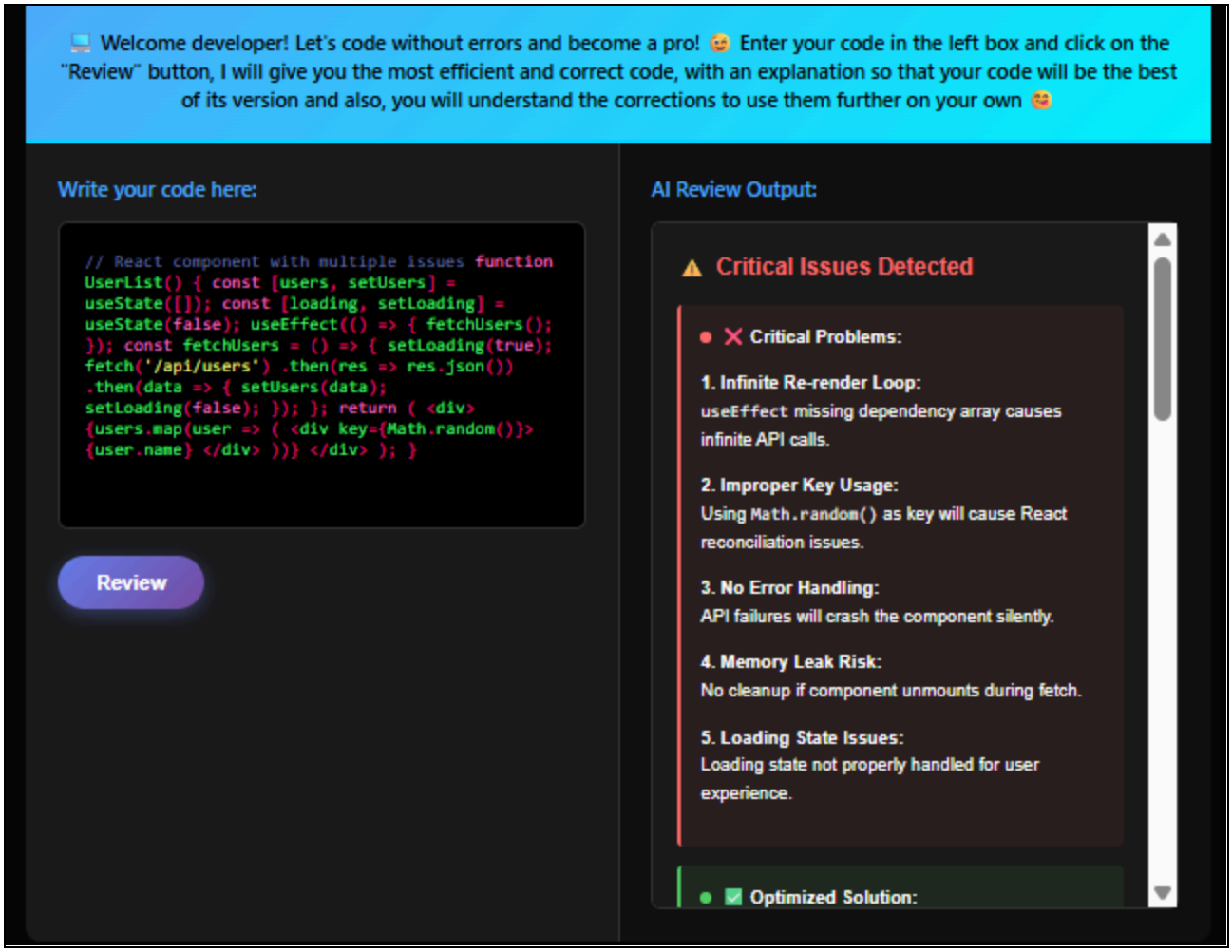


Fig.2

CONCLUSION:

Conclusion of Main Experiment

In this experiment, we successfully implemented and understood **React Hooks** like `useEffect`, `useContext`, and custom hooks through a practical project. These hooks replaced the need for class components and provided a cleaner, modular way of handling state and logic.

- With **useEffect**, we managed side effects such as syntax highlighting using PrismJS, ensuring the UI updated dynamically when code was modified.
- With **useContext**, we could simplify global state sharing across components, reducing prop drilling and improving maintainability.
- With **custom hooks**, we encapsulated reusable logic such as code submission and API handling, making the codebase more structured and reusable.

This implementation showed how hooks simplify lifecycle management, global state handling, and side-effect control. The project demonstrated the power of hooks in building scalable, interactive, and responsive applications.

For the additional part, we extended the project by integrating a **Node.js + Express backend** connected with **Google Gemini API**. We built an MVC-structured backend with routes, controllers, and services. The `ai.service.js` used **Prompt Engineering**, where we crafted a detailed instruction for Gemini to act as a Senior Code Reviewer. This guided the model to analyze, suggest improvements, and return structured reviews of user-submitted code.

Thus, the extra part not only enhanced the functionality of the frontend but also demonstrated how backend integration and prompt engineering can create a complete AI-powered system.