# Vidyavardhini's College of Engineering and Technology

## Department of Artificial Intelligence & Data Science

**AY: 2025-26**

| | | | |
|---|---|---|---|
| **Class:** | T.E. | **Semester:** | V |
| **Course Code:** | CSC502 | **Course Name:** | WC |

| | |
|---|---|
| **Name of Student:** | SHRUTI GAUCHANDRA |
| **Roll No. :** | 18 |
| **Assignment No.:** | 06 |
| **Title of Assignment:** | ADVANCED REACT |
| **Date of Submission:** | 09/10/25 |
| **Date of Correction:** | 09/10/25 |

## Evaluation

| Performance Indicator | Max. Marks | Marks Obtained |
|---|---|---|
| Completeness | 5 | 5 |
| Demonstrated Knowledge | 3 | 3 |
| Legibility | 2 | 2 |
| Total | 10 | 10 |

| Performance Indicator | Exceed Expectations (EE) | Meet Expectations (ME) | Below Expectations (BE) |
|---|---|---|---|
| Completeness | 5 | 3-4 | 1-2 |
| Demonstrated Knowledge Legibility | 3 | 2 | 1 |
| Legibility | 2 | 1 | 0 |

**Checked by**

**Name of Faculty** : Ms. KSHITIJA GHARAT

**Signature** : Gharat

**Date** : 6/10/25

CSC502.4 Using functional components of react develop back-end application.

Q1. You are building a React functional component that needs to fetch the data from an external API and display it. Use the useEffect hook to perform this side effect. Write a React component that fetches a list of users from https://jsonplaceholder.typicode.com/users and displays their names in a list. Explain how useEffect manages the side effect in your implementation.

→ Code:

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

function UsersList(){
  const [users, setUsers] = useState([]);
  const [loading, setLoading] =
useState(true);
  const [error, setError] =
useState(null);

  useEffect(()=>{
    const fetchUsers = async() => {
      try{
        const response = await
fetch('https://jsonplaceholder.typicode.code/users');

        if(!response.ok){
          throw new Error('Failed to fetch
```

```
      fetch users');
      }
      const data = await response.json();
      setUsers(data);
      setLoading(false);
    } catch (err) {
      setError(err.message);
      setLoading(false);
    }
  };
  fetchUsers();
}, []);

if(loading) return <div>Loading users...</div>;

if (error) return <div>Error: {error}</div>;
  return (
    <div>
    <h1>Users List</h1>
     <ul>
       {users.map(user => (
          <li key={user.id}>{user.name}</li>
     ))}
     </ul>
    </div>
  );
}
export default UsersList;
```

How the useEffect manages side effects:
The useEffect hook handles the side effect of fetching data by running after the component renders.
The empty dependency array [] ensures the fetch operation runs only once when the component mounts, preventing infinite re-renders.
The async function inside UseEffect manages the API call, updates state with fetched ~~code~~data, and handles errors appropriately.

Q2. You are developing a simple web application to manage a library of books. Using MVC architecture, implement the functionality to add a new book and display the list of books. Describe how you seperate the responsibilities between the Model, View, and Controller in your implementation.

→ Code:

```
// models/Book.js
const books = [];
let currentId = 1;
class Book {
    constructor(title, author, year) {
        this.id = currentId++;
        this.title = title;
        this.author = author;
        this.year = year;
```

```
}
static addBook(title, author, year) {
    const newBook = new Book(title, author, year);
    books.push(newBook);
    return newBook;
}
static getAllBooks() {
    return books;
}
static getBookById(id) {
    return books.find(book => book.id === parseInt(id));
}
}
module.exports = Book;

// controllers/bookController.js
const Book = require('../models/Book');
const bookController = {
    getBooks: (req, res) => {
        const books = Book.getAllBooks();
        res.render('books', {books});
    },
    showAddForm: (req, res) => {
        res.sender('addBook');
    },
    addBook: (req, res) => {
        const {title, author, year} = req.body;
        book.addBook(title, author, year);
        res.redirect('/books');
    }
```

```
};
module.exports = bookController;

// views/books.ejs

<!DOCTYPE html>
<html>
<head>
    <title> Library Books </title>
</head>
<body>
    <h1> Library Books </h1>
    <a href="/books/add"> Add New Book </a>
    <ul>
      <% books.forEach(book => { %>
        <li>
          <strong> <%= book.title %> </strong>
          by <%= book.author %>
          (<%= book.year %>)
        </li>
      <% }); %>
    </ul>
</body>
</html>

// app.js
const express = require('express');
const bodyParser = require('body-Parser');
const bookController = require('./controllers/
    bookController');
```

```
const app = express();
app.set('view-engine', 'ejs');
app.use(bodyParser.urlencoded({extended: true}));

app.get('/books', bookController.getBooks);
app.get('/books/add', bookController.showAddForm);
app.post('/books', bookController.addBook);

app.listen(3000, () => {
    console.log('Server running on port 3000');
});
```

The Model handles data logic and business rules for books, managing the in-memory storage and CRUD operations.
The View renders the user interface using EJS templates to display book information.
The Controller acts as an intermediary, processing HTTP requests, invoking model methods, and selecting appropriate views to render.

Q3. A Differentiate between MVC, FLUX and Redux

| Features | MVC | FLUX | Redux |
|---|---|---|---|
| Direction of data flow | Bidirectional flow | Unidirection-al flow | Unidirectional flow |
| Stores | No store concept | Multiple stores | Single store |
| Logic handling | Controller manages logic handling | Stores manages logic handling | Reducer manages logic handling. |
| Debugging | Debugging is difficult | Debugging is simpler with dispatcher | Debugging is faster with single store |
| Usage | Used for both server-side and client-side frame-works | Used for client-side frameworks | Used for client-side frameworks |
| Frontend frameworks supported. | AngularJS, Backbone, Sprout, etc. | React, Vue.js, Polymer, Angular. | Backbone, Sprout, Meteor, React, etc. |
| Back-end frameworks supported | Django, Ruby on Rails, Meteor | - | - |

Q4. You are building a React application that manages a shopping cart with multiple products. Use Redux to manage the state of the cart, including adding, removing, and updating items. Implement the Redux store, actions, and reducers, and explain how Redux helps in managing state across the application.

→ Code:

```
// redux/cartActions.js
export const ADD_TO_CART = 'ADD_TO_CART';
export const REMOVE_FROM_CART = 'REMOVE_FROM_CART';
export const UPDATE_QUANTITY = 'UPDATE_QUANTITY';

export const addToCart = (product) => ({
  type : ADD_TO_CART,
  payload:product
});
export const removeFromCart = (productId) => ({
  type: REMOVE_FROM_CART,
  payload : productId
});
export const updateQuantity = (productId, quantity)
  => ({
    type: UPDATE_QUANTITY,
    payload: {productId, quantity}
});

// redux/cartReducer.js
import {ADD_TO_CART, REMOVE_FROM_CART, UPDATE_QUANTITY
```

```
} from './cartActions';
const initialState = {
    items: []
    totalAmount: 0;
};
const cartReducer = (state = initialState, action) => {
    switch (action.type) {
        case ADD_TO_CART:
            const existingItem = state.items.find(
                item => item.id === action.payload.id
            );
        if (existingItem) {
            return {
                ...state,
                itenss:
    state.items.map(item => item.id === action.payload.id
        ? { item, quantity: item.quantity + 1 } : Item),
            totalAmount: state.totalAmount +
            action.payload.price };
    }
        return {
            ....state, items: [...state.items, { ...action.payload,
        quantity: 1 }],
        totalAmount: state.totalAmount + action.payload
            price.
        };
    case REMOVE_FROM_CART:
        const itemToRemove = state.items.find (item =>
        item.id === action.payload);
        return {
```

```
    ...state, items:
    state.items.filter(item => item.id !== action.payload),
    totalAmount:
    state.totalAmount -(itemToRemove.price + itemToRemo
    -ve.price *itemToRemove.quantity)
    };

    return {... state,
    items: state.items.map(item => item.id === action.
    payload.productId ? {... item, quantity: action.
      payload.quantity} : item
    ), totalAmount: state.totalAmount+ priceDifference
  };
  default:
    return state;
  }
};
export default cartReducer;

// redux/store.js
import {createStore} from 'redux';
import cartReducer from './cartReducer';

const store = createStore(
    cartReducer,
    window.__REDUX_DEVTOOLS_EXTENSION__ && window.__
    REDUX_DEVTOOLS_EXTENSION__()
);
export default store;
```

```
// React Component
import React from 'react';
import {useSelector, useDispatch} from 'react-redux';
import {addToCart, removeFromCart, updateQuantity}
 from './redux/cartActions';

function ShoppingCart() {
  const cart = useSelector(state => state);
  const dispatch = useDispatch();
  const handleAddToCart = (product) => {
     dispatch(removeFromCart(productId));
                 add TO
};

  const handleRemove = (productId) => {
     dispatch(removeFromCart(productId));
  };

  const handleUpdateQuantity = (productId, quantity)
   => {
     dispatch(updateQuantity(productId,quantity));
};
  return(
    <div>
       <h2> Shopping Cart </h2>
       <p> Total: $
  {cart.totalAmount.toFixed(2)} </p>
       <ul>
        {cart.items.map(item => (
          <li key={item.id}>{item.name} - $
    {item.price} x {item.quantity}
```

```
            <button onClick={() =>
              handleRemove(item.id)}>Remove</button>
            </li>
          ))}
        </ul>
      </div>
    );
}

export default ShoppingCart;
```

Redux provides a centralized store that maintains the entire application state in one place, making it accessible to any component.
The predictable state updates through pure reducers functions ensure debugging becomes easier and state changes are traceable.
Redux eliminates prop drilling by allowing any component to access state directly through the useSelector hook and dispatch actions using useDispatch.

Q5. Given a complex web application, propose a strategy using advanced React features like Refs and Hooks to manage component states across multiple layers. Explain the benefits of this approach.

→

```
import React, {createContext, useContext, useRef,
    useState, useCallback} from 'react';

const AppContext = createContext();
```

```
function useAppState () {
  const [globalState, setGlobalState] = useState({
    user: null,
    theme : 'light',
    notifications : []
  });

  const inputRef = useRef(null);
  const timerRef = useRef(null);

  const updateState = useCallback((key, value) => {
    setGlobalState ( prev => ({
      ...prev,
       [key] : value
    }));
}, []);

  const focusInput = useCallback(() => {
    if (inputRef.current) {
      inputRef.current.focus();
    }
}, []);
  return{
      globalState,
      updateState,
      inputRef,
      timerRef,
      focusInput.
  };
}
```

```
function AppProvider({ children}) {
  const appState = useAppState();
  return(
    <AppContext.Provider value={appState}>
      {children} </AppContext.Provider>
  );
}
function useApp() {
  const context = useContext(AppContext);
  if (! context) {
    throw new Error('useApp must be used within
      AppProvider');
  }
  return context;
}

function FormComponent() {
  const {globalState, updateState, inputRef, focusInput
  } = useApp();
  return(
    <div>
      <input
        ref={inputRef}
        type="text"
        placeholder="Enter text".
        value={globalState.user || ""}

        onChange={(e)=>
  updateState('user', e.target.value)} />
  <button onClick={focusInput}> Focus Input </button>
```

```
    </div>
  );
}
function App() {
  return(
    <AppProvider>
      <FormComponent/>
    </AppProvider>
  );
}
export default App;
```

Benefits of this approach:

1. Centralized state Management: context API with custom hooks provides, avoiding prop drilling.
2. useCallback memorizes functions to reduce unnecessary re-renders.
3. Refs enable direct DOM manipulation without causing re-renders, improving performance.
4. Custom hooks encapsulate logic for better code reuse and maintainability.