Third semester MCA Degree Examination, Dec.2025/Jan.2026

Rich Internet Application Development

Time: 3 hrs **Max.Marks: 100**

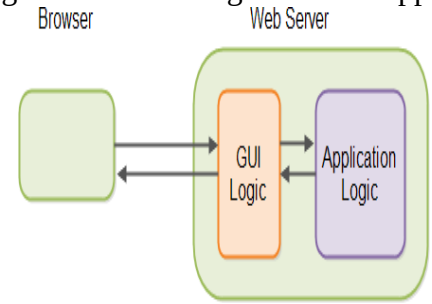
1. A. List the main stages in the evolution of Rich Internet Applications (RIA) and describe each stage briefly.

First Generation Web Applications

- First generation web applications were page oriented.
- All GUI logic and application logic inside the same web page
- Every action the application allowed was typically embedded in its own web page script.
- Each script was like a separate transaction which executed the application logic, and generated the GUI to be sent back to the browser after the application logic was executed.
- First generation web page technologies include Servlets (Java), JSP (JavaServer Pages), ASP, PHP and CGI scripts in Perl etc.

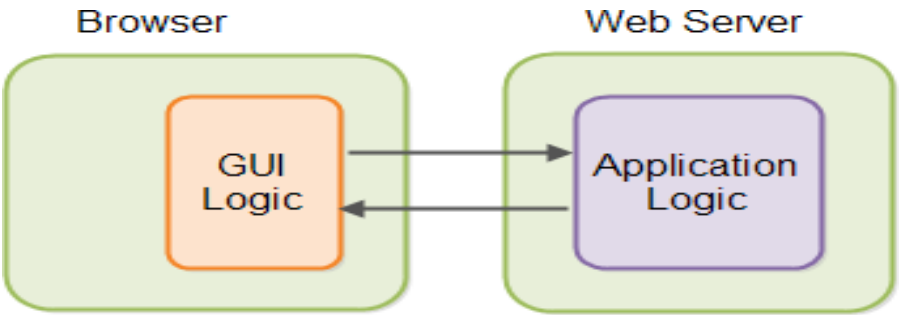
Second Generation Web Applications

- In second generation web applications developers found ways to separate the GUI logic from the application logic on the server shown in Figure 1.5.
- Web page scripts were used for the GUI logic
- Real classes and objects were used for the application logic
- Frameworks were developed to help make second generation web applications easier to develop.
- Examples of such frameworks are ASP.NET (.NET), Struts + Struts 2 (Java), Spring MVC (Java), JSF (JavaServer Faces), Wicket (Java) Tapestry (Java) and many others.
- GUI logic was written in the same language as the application logic, changing the programming language meant rewriting the whole application again.



RIA Web Applications

- RIA (Rich Internet Applications) web applications are the third generation of web applications.
- RIA technologies are typically executed in the browser using either JavaScript, Flash, JavaFX or Silverlight
- The GUI logic is now moved from the web server to the browser.
- GUI logic is executed in the browser, the CPU time needed to generate the GUI is lifted off the server, freeing up more CPU cycles for executing application logic.
- GUI state can be kept in the browser, thus further cleaning up the server side of RIA web applications.
- GUI logic is completely separated from the application logic is shown in Figure 1.6, it becomes easier to develop reusable GUI components
- No matter what server-side programming language is used for the application logic.



The main stages in the evolution of RIA are:

1. Static Web Era (Pre-RIA)

- **Description:** Initially, the web consisted of HTML-based pages that delivered static content. Interactions were synchronous—every action required a full page reload, causing latency, poor interactivity, and limited functionality, similar to a "thin-client" model.

2. Browser Plugin Era (Flash/Applets)

- **Description:** To overcome the limitations of standard HTML, browser plugins like Macromedia Flash (later Adobe Flash), Java Applets, and later Silverlight were used. These technologies provided rich, interactive graphical interfaces and multimedia capabilities that ran within a browser "sandbox," offering functionality similar to desktop applications.

3. AJAX and Dynamic Web 2.0

- **Description:** This stage introduced Asynchronous JavaScript and XML (AJAX), allowing web applications to send and receive data in the background without reloading the entire page. This enabled faster response times, dynamic UI updates (like Google Maps), and improved user experience, marking the transition to "Single Page Applications".

4. Rich Component Frameworks (Flex/Silverlight)

- **Description:** Specialized development frameworks were introduced (e.g., Adobe Flex, Microsoft Silverlight) to create advanced, enterprise-level RIA applications. These frameworks allowed developers to create complex, data-intensive interfaces with advanced controls, bridging the gap between web and desktop applications.

5. Native HTML5 and Modern JavaScript

- **Description:** The current, mature stage is defined by the decline of proprietary plugins in favor of HTML5, CSS3, and JavaScript libraries (e.g., React, Angular). This era leverages native browser technology to provide high-performance, rich, responsive, and mobile-friendly applications without the need for external plugins.

b. Demonstrate the use of JavaScript map() and filter() functions with short examples.

The map(), filter(), and reduce() methods are powerful JavaScript array functions that help transform and process data efficiently. They allow you to apply custom logic to arrays in a clean, functional programming style.

- map() creates a new array by applying a function to each element.
- filter() returns a new array containing only elements that meet a condition.
- reduce() combines all elements into a single value (like a sum or object).

1. JavaScript map() Method

The map() method in JavaScript creates a new array by applying a callback function to each element of the original array.

- Iterates through every element of the array
- Executes the callback function for each element
- Stores the returned value in the new array

```
let arr= [2, 4, 8, 10]

let updatedArr = arr.map(val=> val+2)

console.log(arr);

console.log(updatedArr);
```

2. JavaScript filter() Method

The filter() method in JavaScript creates a new array containing only the elements that satisfy a condition defined in a callback function.

- Iterates over each element of the array

- Includes elements when the callback returns true
- Excludes elements when the callback returns false

```
let arr = [2, 4, 8, 10];

let updatedArr = arr.slice().filter(val => val < 5);

console.log(arr);

console.log(updatedArr);
```

JavaScript reduce() Method

The reduce() in JavaScript is used to combine all elements of an array into a single value by applying a callback function to each element.

- **Accumulator:** stores the result after each iteration
- **currentValue:** the current element being processed
- **currentIndex:** index of the current element

```
let arr= [2,4,8,10]

let updatedArr = arr.reduce((prev, curr)=> curr= prev+curr)

console.log(arr);

console.log(updatedArr);
```

map()	filter()	reduce()
Returns a new array	Returns a new array	Returns a single element
Modifies each element of the array	Filter out the element which passes the condition	Reduces array to a single value
Uses value, index, array	Uses current value, index, array	Uses Previous value and current value

2.A. Identify the differences between var, let, and const in ES6 and state where each is preferred

JavaScript provides three ways to declare variables: var, let, and const, but they differ in scope, hoisting behaviour,

and re-assignment rules.

- **var:** Declares variables with function or global scope and allows re-declaration and updates within the same scope.

- **let:** Declares variables with block scope, allowing updates but not re-declaration within the same block.
- **const:** Declares block-scoped variables that cannot be reassigned after their initial assignment.

```
// var example

var x = 10;

var x = 20;   // Re-declaration allowed

x = 30;       // Update allowed

console.log(x); // Output: 30

// let example

let y = 10;

// let y = 20; // Re-declaration NOT allowed

y = 25;       // Update allowed

console.log(y); // Output: 25

// const example

const z = 10;

// z = 20;     // Re-assignment NOT allowed

console.log(z); // Output: 10

// Block scope demonstration

if (true) {

  var a = 1;

  let b = 2;

  const c = 3;

}

console.log(a); // Works (var is function/global scoped)

// console.log(b); //  Error (let is block scoped)

// console.log(c); //  Error (const is block scoped)
```

b.Illustrate the use of JavaScript classes by creating a simple class for a student and showing how objects are created.

In JavaScript, **classes** and **objects** are basic concepts of Object Oriented Programming (OOPs) that are used to
represent real-world concepts and entities.

- A class is a template to create objects having similar properties and behavior, or in other words, we can say that a class is a blueprint for objects.
- An object is an instance of a class. For example, the animal type Dog is a class, while a particular dog named Tommy is an object of the Dog class.

The `class` keyword is used to declare a new class. The `constructor` is a special method called automatically when a new object is created using the `new` keyword

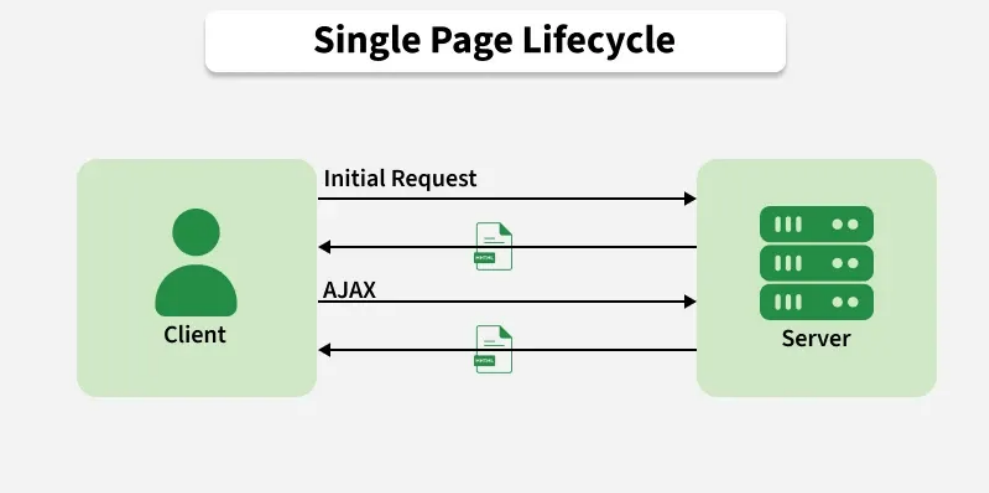
Example

```
class Student {
  constructor(name, grade, gpa) {
    this.name = name;
    this.grade = grade;
    this.gpa = gpa;
  }

  introduce() {
    console.log(`Hello, my name is ${this.name}. I am in grade ${this.grade} and my GPA is ${this.gpa}.`);
  }

  isOnHonorRoll() {
    return this.gpa >= 3.5;
  }
}

const student1 = new Student('Alice', 10, 3.8);
const student2 = new Student('Bob', 9, 3.2);

console.log("--- Student 1 Details ---");
console.log(`Name: ${student1.name}`);
console.log(`Grade: ${student1.grade}`);
student1.introduce();
console.log(`On Honor Roll: ${student1.isOnHonorRoll()}`);

console.log("\n--- Student 2 Details ---");
console.log(`Name: ${student2.name}`);
console.log(`Grade: ${student2.grade}`);
student2.introduce();
console.log(`On Honor Roll: ${student2.isOnHonorRoll()}`);
```

3.A. Summarize the key features of React that make it suitable for building Single Page Applications.

A Single Page Application (SPA) is a web app that loads content dynamically without refreshing the whole page. It uses JavaScript to update only the required parts of the screen based on user actions. This creates a smooth, fast, and app-like user experience.

- Eliminates full page reloads for better performance
- Sends and updates only the necessary data from the server
- Provides a more responsive and seamless interface



SPA Architecture and its Working

A Single Page Application (SPA) loads once and then updates content without reloading the page, giving a fast and smooth user experience.It includes three renderings :

Client-side rendering

- First browser requests HTML from the server.
- Then server swiftly responds with a basic HTML file and linked styles/scripts.
- Now user sees a blank page or loader while JavaScript executes.
- App fetches data, creates views, and injects them into the DOM.

Client-Side Rendering (CSR) can be slower for basic websites because it uses a lot of device resources. Yet, it's good for busy websites, making things faster for users. Just remember, if you want different social sharing options, you might need Server-Side Rendering (SSR) or Static Site Generation (SSG) instead.

Server-side rendering (SSR)

- The browser first asks the server for an HTML file.
- Now server gathers necessary data, builds the SPA, and creates an HTML file instantly.
- Now user sees the content ready to go.
- Single-page app structure adds events, makes a virtual DOM, and gets things ready.
- Now, the application is set for use.

HUSPI opts for server-side rendering to achieve a swift application experience, balancing the speed of single-page applications without burdening the user's browser, ensuring optimal app performance.

Static site generator (SSG)

- Browsers ask for HTML, SSGs quickly provide pre-made static pages.
- The server shows users the static page for fast loading.

- SPAs in the page fetch data and make dynamic changes to the page.
- SPA is ready for smooth user interaction after data is added.
- SSGs are great for fast static pages but may not be ideal for highly dynamic websites.

b.Construct a simple React functional component that displays a welcome 10 message and uses props.

In ReactJS**,** functional components are a core part of building user interfaces. They are simple, lightweight, and powerful tools for rendering UI and handling logic. Functional components can accept props as input and return JSX that describes what the component should render.

```
import React from 'react';

/**
 * A functional component that displays a personalized welcome message.
 *
 * @param {object} props - The props object.
 * @param {string} props.name - The name of the user.
 * @param {string} props.message - The welcome message to display.
 */
const WelcomeMessage = ({ name, message }) => {
  return (
    <div className="welcome-container">
      <h1>Hello, {name}!</h1>
      <p>{message}</p>
      <p>Welcome to our application.</p>
    </div>
  );
};

// Example usage in another component or the main App file:
// function App() {
//   return (
```

```
//    <div className="App">
//      <WelcomeMessage name="Jane Doe" message="We're glad to have you back." />
//      {/* You can reuse the component with different props: */}
//          <WelcomeMessage name="Guest User" message="Please sign in to access all features." />
//    </div>
//  );
// }
// Export the component for use elsewhere in the application
export default WelcomeMessage;
```

When a functional component receives input and is rendered, React uses props and updates the virtual DOM to ensure the UI reflects the current state.

- **Props:** Functional components receive input data through props, which are objects containing key-value pairs.
- **Processing Props:** After receiving props, the component processes them and returns a JSX element that defines the component's structure and content.
- **Virtual DOM:** When the component is rendered, React creates a virtual DOM tree that represents the current state of the application.
- **Re-rendering:** If the component's props or state change, React updates the virtual DOM tree accordingly and triggers the component to re-render.

Q.4 A. Describe how navigation occurs in a Single Page Application with the help of a basic routing example.

In a Single Page Application (SPA), navigation is managed on the client side using a routing library, which dynamically updates the user interface (UI) without requiring a full page reload from the server.

How Navigation Occurs in an SPA

The core process of SPA navigation involves several key steps:

1. **User Interaction**: The user clicks a link or uses browser navigation buttons (back/forward).
2. **Event Handling**: Instead of the browser performing its default action (making a new server request), the routing library intercepts the event.
3. **URL Modification**: The library uses the History API (pushState or replaceState) to change the browser's URL without a full page refresh.
4. **Component Rendering**: The router matches the new URL path to a specific UI component or view configured in its route definitions. It then dynamically renders the appropriate component into a designated "root" element in the HTML.

5. **View Update**: The content of the webpage changes instantly, giving the illusion of a traditional page navigation, all while remaining on the single initial HTML page

Basic Routing Example

Consider a simple SPA built with a JavaScript framework (like React, Angular, or Vue) and a hypothetical routing configuration:

Route Configuration (Conceptual JavaScript):

```
// A conceptual example of defining application routes

const routes = [

  { path: '/', component: HomeComponent },

  { path: '/about', component: AboutComponent },

  { path: '/products/:id', component: ProductDetailComponent }

];

// When the application loads, it mounts the root component.

// The router handles which specific view is visible.
```

Scenario: Navigating from Home to About Page

1. **Initial Load**: The user loads the website https://example.com/. The router matches the path / and renders the HomeComponent.
2. **User Clicks Link**: The user clicks a link defined as About Us.
3. **Router Intercepts**: The routing library stops the browser from navigating traditionally.
4. **URL Updates**: The URL in the address bar changes to https://example.com/about using the History API.
5. **Component Swap**: The router checks its configuration, finds the /about path, hides the HomeComponent, and renders the AboutComponent in its place. The user sees new content instantly without the browser ever leaving the original HTML page shell.

b.Write a small React program that updates a counter when a button is clicked.

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

import './App.css'; // Optional: for basic styling

function App() {

  // Declare a state variable 'count' and a function 'setCount' to update it

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

  // Function to handle the button click and increment the counter
```

```
  const incrementCounter = () => {
    setCount(count + 1);
  };
  return (
    <div className="App">
      <header className="App-header">
        <h1>Counter App</h1>
        {/* Display the current count */}
        <p>Current count: {count}</p>
        {/* Button that calls the incrementCounter function when clicked */}
        <button onClick={incrementCounter}>
          Click Me
        </button>
      </header>
    </div>
  );
}
export default App;
```

5.A. Present the steps involved in sending a request using XMIHttpRequest and receiving data from a server.

XMLHttpRequest is an object that is used to send a request to the webserver for exchanging data or transferring and manipulating to it and from the server behind the scenes. You can use the received data to update the data present on the web page without even reloading the page.
Below is the complete syntax to use XMLHttpRequest object.

Syntax:
At first, you have to invoke the **XMLHttpRequest()** method as shown below.

```
variable_name.onload = function () {
   // Content of callback function
   // after getting the response
}
```

Sending a request using the **open()** and **send()** methods as shown below.

```
variable_name.open("GET", "textFile.txt");
variable_name.send();
```

The XMLHttpRequest object has different functions and properties that are listed below.

XMLHttpRequest object methods:

- **new XMLHttpRequest():** It creates a new XMLHttpRequest object.
- **abort():** This method will cancel the current request to exchange data.
- **getAllResponseHeaders():** It will return a set of HTTP headers in form of a string.
- **getResponseHeader():** It will return the specified HTTP header information.
- **open(method, URL, async, userName, password):** This method specifies the method, URL, and other parameters of a request. In this function call, the method parameter defines the operation that has to be performed on the data like **GET, POST, HEAD,** and some other HTTP methods such as **PUT, DELETE.** The **async** parameter specifies the asynchronous behavior of the request. It holds two values *true* and *false***.** If the value is *true* then the script processing will continue after **send()** method without waiting for the response while *false* value means that the script will wait for a response before processing it.
- **send():** It will send the request to the server for data exchange. To GET the requests, **send()** is used.
- **send(string):** It also sends the request to the server for data exchange, but It is used to POST the requests.
- **setRequestHeader():** It will add the label and value pair to the header that has to be sent.

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content=
       "width=device-width, initial-scale=1.0">
    <title>Ready states of a request in ajax</title>
    <style>
       #container {
           display: flex;
           flex-direction: column;
           justify-content: center;
```

```
            align-items: center;
        }
    </style>
</head>
<body>
    <div id="container">
        <h1>Hey Geek,</h1>
        <h3>Welcome to GeeksforGeek!</h3>
        <button type="button" onclick="stateChange()">
            Update Text
        </button>
    </div>
    <script>
        function stateChange() {
            var state = new XMLHttpRequest();
            state.onload = function () {
                document.getElementById("container")
                    .innerHTML = state.getAllResponseHeaders();
            }
            state.open("GET", "gfgInfo.txt", true);
            state.send();
        }
    </script>
</body>
</html>
```

b.Show how JSON data is structured by creating a sample JSON object for a book record.

A JSON object for a book record is structured using key-value pairs, with each pair representing a specific property of the book, such as "title", "author", or "publication_year".

```
{
  "title": "The Hitchhiker's Guide to the Galaxy",
  "author": {
    "first_name": "Douglas",
    "last_name": "Adams"
  },
  "publication_year": 1979,
  "genres": [
    "Science Fiction",
    "Comedy"
  ],
  "is_available": true
}
```

Key Structural Elements

This example demonstrates several fundamental aspects of JSON structure:

- **Objects:** The entire record is enclosed within curly braces {}. Objects store unordered data as a collection of key-value pairs.
- **Key-Value Pairs:** Each piece of data has a key (a string, like "title") and a value (the data associated with that key, like "The Hitchhiker's Guide to the Galaxy"). A colon : separates the key from the value, and commas , separate individual pairs.
- **Nested Objects:** The "author" value is itself another JSON object, allowing for more complex, hierarchical data representation.
- **Arrays:** The "genres" value is an array, enclosed in square brackets []. Arrays are ordered lists of values (strings in this case).
- **Data Types:** JSON supports various data types, as seen in the example:
  - **Strings:** "title", "first_name", "Science Fiction"
  - **Numbers:** 1979
  - **Booleans:** true

- **Objects:** {"first_name": "Douglas", ...}
- **Arrays:** ["Science Fiction", "Comedy"]

6.A. Demonstrate how the Fetch API handles errors using try catch with a short code snippet.

The Fetch API can be effectively used with `try...catch` within an `async` function to handle network errors and errors from the API response. async function fetchData(url) {

```
try {
  // 1. Initiate the fetch request
  const response = await fetch(url);

  // 2. Check if the request was successful (status in the 200s)
  if (!response.ok) {
    // If not successful, throw an Error to be caught by the catch block
    // You can access response.status and response.statusText here
    throw new Error(`HTTP error! status: ${response.status}`);
  }

  // 3. Parse the response body as JSON
  const data = await response.json();
  console.log('Data received:', data);
  // You can return the data or process it further here

} catch (error) {
  // 4. Handle any errors that occurred during the fetch operation or in the 'if' block
  console.error('Fetch operation failed:', error.message);
  // Display an error message to the user, log the error, etc.
}
```

```
}


// Example usage with a placeholder URL

// fetchData('https://api.example.com/data'); Explanation of the Error Handling Steps:
```

- **try block:** This block contains the code that might throw an error. The await fetch(url) call will only throw a network error (like a DNS failure or the user being offline) which prevents the request from completing.
- **if (!response.ok) check:** The Fetch API's design considers status codes like 404 Not Found or 500 Server Error as successful network requests. To treat these HTTP errors as exceptions, you must explicitly check the response.ok property and manually throw new Error().
- **catch block:** This block executes if an error is thrown within the try block. This includes network failures, the manual error thrown for HTTP status issues, or errors during JSON parsing (e.g., if the response body is invalid JSON).

B. List the commonly used HTTP methods and state the purpose of each method.

HTTP (Hypertext Transfer Protocol) specifies a collection of request methods to specify what action is to be performed on a particular resource. The most commonly used HTTP request methods are **GET, POST, PUT, PATCH, and DELETE**. These are equivalent to the **CRUD operations (create, read, update, and delete)**.

HTTP Requests

HTTP Requests are the message sent by the client to request data from the server or to perform some actions. Different HTTP requests are:

GET

The GET method is used to retrieve data on a server. Clients can use the GET method to access all of the resources of a given type, or they can use it to access a specific resource. For instance, a GET request to the /products endpoint of an e-commerce API would return all of the products in the database, while a GET request to the /products/123 endpoint would return the specific product with an ID of 123. GET requests typically do not include a request body, as the client is not attempting to create or update data.

POST

The POST method is used to create new resources. For instance, if the manager of an e-commerce store wanted to add a new product to the database, they would send a POST request to the /products endpoint. Unlike GET requests, POST requests typically include a request body, which is where the client specifies the attributes of the resource to be created. For example, a POST request to the /products endpoint might have a request body that looks like this:

```
{
  "name": "Sneakers",
```

```
  "color": "blue",
  "price": 59.95,
  "currency": "USD"
}
```

PUT
The PUT method is used to replace an existing resource with an updated version. This method works by replacing the entire resource (i.e., the specific product located at the /products/123 endpoint) with the data that is included in the request's body. This means that any fields or properties not included in the request body are deleted, and any new fields or properties are added.

PATCH
The PATCH method is used to update an existing resource. It is similar to PUT, except that PATCH enables clients to update specific properties on a resource—without overwriting the others. For instance, if you have a product resource with fields for name, brand, and price, but you only want to update the price, you could use the PATCH method to send a request that only includes the new value for the price field. The rest of the resource would remain unchanged. This behavior makes the PATCH method more flexible and efficient than PUT.

DELETE
The DELETE method is used to remove data from a database. When a client sends a DELETE request, it is requesting that the resource at the specified URL be removed. For example, a DELETE request to the /products/123 endpoint will permanently remove the product with an ID of 123 from the database. Some APIs may leverage authorization mechanisms to ensure that only clients with the appropriate permissions are able to delete resources.

7.A. Use the useState hook to build a simple React component that manages a login form with two input fields.

React Hook Form is a popular third-party library that simplifies form management in React functional components by using hooks. It offers a complete set of tools to handle various aspects of forms, such as form state management**,** field handling**,** and form submission.
Some of the features of the React Hook Form Libraries are:

- Open-source
- Supports TypeScript
- Provides DevTool for inspecting form data
- Provides Form Builder – create forms by drag and drop
- Supports for React-native

```
import React from "react";
import { useForm } from "react-hook-form";
import "./App.css";
```

```
function Login() {
const {
 register,
handleSubmit,
                        formState: { errors },
                      } = useForm();

                      const onSubmit = (data) => {
                        const userData = JSON.parse(localStorage.getItem(data.email));
                        if (userData) {
                          if (userData.password === data.password) {
                            console.log(userData.name + " You Are Successfully Logged
                       In");
                          } else {
                            console.log("Email or Password is not matching with our
                       record");
                          }
                        } else {
                          console.log("Email or Password is not matching with our
                       record");
                        }
                      };

                      return (
                        <>
                          <h2>Login Form</h2>

                          <form className="App" onSubmit={handleSubmit(onSubmit)}>
                            <input
                              type="email"
                              {...register("email", { required: true })}
                              placeholder="Email"
                            />
                            {errors.email && <span style={{ color: "red" }}>*Email* is
                       mandatory</span>}

                            <input
                              type="password"
                              {...register("password", { required: true })}
                              placeholder="Password"
                            />
                            {errors.password && <span style={{ color: "red"
                       }}>*Password* is mandatory</span>}

                            <input type="submit" style={{ backgroundColor: "#a1eafb" }}
                       />
```

```
            </form>
        </>
    );
}
```

export default Login;

B. Outline the role of Material Ul in React development and mention four useful components.

MUI or Material UI is an **open-source React Components library** that is based on **Google's Material Design** and provides the **predefined UI components for React**.
Material UI is a **UI library** that **provides predefined react components** implementing **Google's Material Design**. Material UI is a **design language built by Google** in 2014 and works with various JavaScript frameworks apart from React such as Angular.js and Vue.js.

Why choose Material UI?

The main reason to choose Material UI is the quality of the inbuilt designs of Material UI. Its easy implementation makes it the first choice of most developers. The inbuilt components are also customizable so it helps in easily recreating the designs. The material design provided by The MUI is also SEO friendly. One of the various advantages of Material UI is that it has a good depreciation policy which allows easy upgrades to old deprecated components. Since the introduction of Material UI the development of React applications has become a lot faster.

Features of React Material UI

React Material UI combines powerful customization, responsiveness, fast loading, theming, backend integration, and seamless upgrades. The key features of Material UI are:

- **Highly customizable:** The templates provided are easily customizable and require very less coding to make changes to existing templates
- **Responsive:** The inbuilt components are by default responsive so it is easy to create web applications for different screen widths.
- **Fast loading:** Loading of the components is faster as compared to other frameworks
- **Themes:** Vast variety of themes and easy customization is possible.
- **Backend friendly:** It is easily integrable with the backend.
- **Depreciation policy:** Material UI provides easy upgrades for old deprecated methods.

Usage Practices of Material UI

There are some basic conventions which we must follow for effective usage of MUI components such as:

- **Globals:** All the MUI components are defined in a global scope and are isolated. These globals should only be used for website development
- **Mobile-first component:** The components codes are written for mobile first and then these components can be scaled up for bigger screen sizes
- **Responsive meta tag:** The meta tag should be made responsive to ensure touch zooming for all devices
- **CssBaseline:** The is another CssBaseline component which ensures consistency among different browsers

- **Default Font:** The default font used in material design is Roboto

Benefits of Material UI

The Material UI offers many benefits:

- MUI enhances User Experience and User Interface
- The customizable components helps to easily redesign the existing components
- The codes are easier to maintain and debugging is easier
- Apart from styles it improves the components functionality and accessibility

8.a. Describe how uscEffect can be used for loading data from an API when a 10 component is mounted.

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

function DataFetcher() {
  const [data, setData] = useState(null);
  const [isLoading, setIsLoading] = useState(true);
  const [error, setError] = useState(null);
  useEffect(() => {
    const fetchData = async () => {
      try {
        // Replace with your actual API endpoint
        const response = await fetch('https://api.example.com/data');
        if (!response.ok) {
          throw new Error('Network response was not ok');
        }
        const result = await response.json();
        setData(result);
      } catch (err) {
        setError(err.message);
      } finally {
        setIsLoading(false);
      }
```

```
    };
    fetchData();
  }, []); // The empty array ensures this runs only on mount

  if (isLoading) {
    return <div>Loading...</div>;
  }
  if (error) {
    return <div>Error: {error}</div>;
  }
  return (
    <div>
      <h1>Data Loaded:</h1>
      {/* Render your data here */}
      <pre>{JSON.stringify(data, null, 2)}</pre>
    </div>
  );
}

export default DataFetcher;
```

Explanation:

1. **useEffect Hook:** The useEffect hook is a function that allows you to perform side effects in functional components. API data fetching is a common side effect.
2. **Empty Dependency Array ([]):** This is the crucial part. When the dependency array is empty, React runs the effect callback only once after the initial render (when the component is "mounted"). If dependencies were included, the effect would re-run whenever those dependencies changed.
3. **Asynchronous Function (async/await):** Data fetching is an asynchronous operation. You define an async function inside the effect to use await for cleaner handling of the fetch request and the subsequent JSON parsing.
4. **State Management:** useState is used to manage the different states of the data fetching process:

1. data: stores the successfully retrieved data.

2. isLoading: a boolean to show a "Loading..." message while the request is in flight.
3. error: stores any error messages if the fetch fails.
5. **Calling the Function:** The fetchData function is defined within useEffect and immediately called.

B. Identify any three responsive design practices and state how they support mobile-friendly layouts.

Responsive design is essential for ensuring websites function effectively across various devices. Here are three key responsive design practices and how they support mobile-friendly layouts:

1. Fluid Grids and Relative Sizing

- **Definition:** Instead of using fixed pixel widths (e.g., width: 960px), fluid grids use relative units such as percentages (%), rem, or em to define the width of containers and elements.
- **Mobile Support:** This allows the layout to shrink or expand proportionally to fit any screen size, preventing content from being cut off or requiring horizontal scrolling on small screens.

2. CSS Media Queries and Breakpoints

- **Definition:** Media queries are CSS rules that act as filters, applying different styling rules based on device characteristics like screen width, orientation, or resolution. Breakpoints are specific points where the layout changes to accommodate the screen size.
- **Mobile Support:** They allow designers to switch from a multi-column desktop layout to a single-column layout on mobile, or to change font sizes and navigation styles to better suit mobile browsing.

3. Mobile-First Design Approach

- **Definition:** This approach involves designing for the smallest screen (smartphone) first, then progressively enhancing the layout and functionality for larger screens (tablets, desktops).
- **Mobile Support:** By prioritizing the essential content and features for small screens first, it ensures a faster-loading, less cluttered, and highly functional mobile experience, which is crucial for modern mobile users.

9.A. List the steps involved in preparing a React project for deployment on Vercel.

Steps for Deploying a React Project to Vercel

1. **Prepare Your React Project:**

- Ensure your project is complete and ready for production.
- Make sure your package.json file has a standard build script (e.g., "build": "react-scripts build", which is default for Create React App projects) that generates a production-ready static bundle .
- Verify your project runs locally using npm start (or yarn start) and that a production build can be generated successfully using npm run build .

2. **Push Your Code to a Git Repository:**

- Host your project on a supported Git provider like GitHub, GitLab, or Bitbucket .

3. **Link Your Git Repository to Vercel:**

- Sign up or log in to your Vercel account.
- On your Vercel Dashboard, click the **"New Project"** button .
- Import the Git repository where your React project is hosted. You may need to grant Vercel access to your repositories .

4. **Configure the Project Settings:**

- Vercel will automatically detect that it is a React project and pre-fill the build settings (e.g., Build Command: npm run build, Output Directory: build) .
- You can override these settings if necessary (e.g., if you use a different build tool or output directory) .
- Add any required **Environment Variables** under the "Environment Variables" section if your app needs them at build time or runtime [1].

5. **Deploy the Project:**

- Click the **"Deploy"** button. Vercel will start the build process, which involves installing dependencies, running the build command, and deploying the static assets to its global Edge Network .
- Once the deployment is complete, you will see a success page with a link to your live React application.

B. State the purpose of linting and mention common issues detected by linters in web applications.

Linting is a static code analysis process that automatically checks source code for bugs, syntax errors, and stylistic inconsistencies without executing it. Its purpose is to enhance code quality, enforce consistency, and prevent errors early in development. Common issues detected include undefined variables, syntax errors, code style violations, unused code, and security vulnerabilities.

Key Aspects of Linting:

- **Error Detection:** Identifies potential bugs such as typos, syntax errors, and undefined variables.
- **Consistency:** Enforces a uniform coding style (e.g., indentation, semicolon usage) across the project.
- **Best Practices:** Highlights code that ignores established best practices or is difficult to read.
- **Security & Optimization:** Flags potential security vulnerabilities and inefficient, "smelly" code.

Common Issues Detected in Web Applications:

- **JavaScript/TypeScript:** Undeclared variables, unused imports, missing semicolons, improper scoping, and console logs left in production.
- **CSS/HTML:** Invalid syntax, missing vendor prefixes, improper formatting, and accessibility issues (e.g., missing alt attributes).

- **General:** Inconsistent indentation, incorrect variable naming, and dead code.

  Common tools for this process include ESLint, Stylelint, and Webhint.
  Linting can detect errors in a code and errors that can lead to a security vulnerabilities. Linters Can Also detect formatting or styling issues and makes the code more readable for more secure code. Linters can suggest best practices. Also they can increases overall quality of the code.

  Linting tools offer a crucial benefit in error prevention by identifying issues as they are created, even before code execution. This proactive approach significantly improves code reliability and reduces the likelihood of issues ever turning into bugs.

  How Do Lint Tools Work?

  Here's how linting tools are typically fit into the development process.

  1. Write the code.
  2. Compile it.
  3. Analyze it with the linter.
  4. Review the bugs identified by the tool.
  5. Make changes to the code to resolve the bugs.
  6. Link modules once the code is clean.
  7. Analyze them with the linter.
  8. Do manual code reviews.

  Basic Linting Tools

  Linting tools are the most basic form of static analysis. Using lint tools can be helpful for identifying common errors, such as:

  - Indexing beyond arrays.
  - Dereferencing null pointers.
  - (Potentially) dangerous data type combinations.
  - Unreachable code.
  - Non-portable constructs

10.A. Describe how CRF tokens are used for securing form submissions in web applications.

Cross-Site Request Forgery (CSRF) is a critical web vulnerability that allows attackers to trick authenticated users into performing unintended actions, such as changing account details or even taking full control of their accounts.
Web applications typically rely on **cookies** to maintain user sessions, since HTTP is a stateless protocol and does not natively support persistent authentication. Without cookies, users would need to re-enter their credentials for every request, which would severely impact usability.

Cookies solve this by storing session information, but they also make applications susceptible to CSRF attacks if not properly protected.

Imagine you're logged into your online banking account to check your balance. Meanwhile, a malicious website silently tricks your browser into making unauthorized transactions on your behalf. This illustrates a Cross-Site Request Forgery (CSRF) attack, a serious web security vulnerability that impacts countless websites. Recognizing this threat is vital for both users and developers to build safer online experiences.

Conditions Required for a CSRF Attack

For a CSRF attack to succeed all three of the following must hold:

A Relevant Action: The application exposes an action the attacker wants to trigger, for example:

- Changing a user's email or password
- Transferring money
- Modifying account permissions

Cookie-Based Session Handling

- The app identifies users solely by a session cookie (or another credential the browser automatically includes).
- The browser automatically sends that credential with requests made from other sites.
- No additional server-side verification (e.g., CSRF tokens, same-site cookie policy, or re-authentication) is enforced.

Predictable Request Parameters

- The attacker can determine or guess all parameters required to perform the action.
- If a request requires a secret or random value (e.g., the current password, one-time token, or unpredictable nonce), the CSRF attempt will fail.

How CSRF Tokens Work

The process involves synchronizing a unique, unpredictable value (the token) between the server and the client's browser for each user session. This process generally follows these steps:

1. **Generation:** When a user first visits a secure page (like a login or payment form), the server generates a unique, cryptographically secure, and usually time-sensitive token.
2. **Transmission to Client:** The server typically embeds this token as a hidden field within the HTML form or sends it as a cookie to the user's browser .
3. **Submission:** When the user submits the form, their browser sends the form data, including the hidden token, back to the server.
4. **Verification:** The server then verifies that the token received in the request matches the original token associated with the user's session (the one stored server-side or in the cookie) .

Security Benefits

- **Same-Origin Policy Enforcement:** If an attacker attempts to trick a user into submitting a malicious form on a different website, their form would not include the correct, synchronized token .
- **Rejection of Malicious Requests:** The server's verification process will fail due to the missing or incorrect token, automatically rejecting the fraudulent request and preventing the attacker from performing unauthorized actions on the user's behalf .

B. Summarize the process of writing a basic unit test in Jest for a function that adds two numbers.

Jest is a JavaScript testing framework developed by Facebook, designed primarily for unit testing React applications. However, it can be used for testing any JavaScript codebase.

Jest is known for its simplicity, speed, and built-in features, making it one of the most popular choices for testing JavaScript applications.

```
//sum.test.js
const add=require('./sum.js')
test('first test',()=>{
   expect(add()).toBe(5)
})
```

Create the Test File

Next, create a test file in the same directory (or within a __tests__ folder) with a name matching the convention Jest looks for (e.g., adder.test.js or adder.spec.js).

3. Write the Test

Inside the test file, you will import the function and write your test case:

```
// adder.test.js

const add = require('./adder'); // Import the function to be tested

describe('add function', () => { // Optional: groups related tests

  test('adds 1 + 2 to equal 3', () => { // Defines a specific test case

    const result = add(1, 2);          // Call the function with inputs

    expect(result).toBe(3);            // Assert the expected output

  });

  // You can add more test cases:

  test('adds 5 + 10 to equal 15', () => {

    expect(add(5, 10)).toBe(15);

  });

});
```

Run the Tests

Assuming you have Jest installed and configured in your project (typically via npm install --save-dev jest and a configuration in package.json), you run the tests from your terminal using a command like:

```
bash
npm test
```

```
# or if globally installed:
jest
```

Jest will automatically discover and run the adder.test.js file, outputting the results to the console.