**Topic JavaScript:**

**Question 1:**

Create a simple **Interactive Quiz Application** where users can select answers and receive immediate feedback.

**Initial HTML Structure:**

<div id="quiz-container">

<h2>Web Engineering Quiz</h2>

<p id="question">What does DOM stand for?</p>

<div>

<button class="answer" data-correct="false">Data Object Model</button>

<button class="answer" data-correct="true">Document Object Model</button>

<button class="answer" data-correct="false">Dynamic Object Model</button>

</div>

<p id="feedback"></p>

</div>

**Requirements:**

1. **Answer Selection**:
   * Add an event listener to each button that checks if the answer is correct by reading the data-correct attribute.
   * Display "Correct!" or "Wrong!" in the #feedback paragraph.
2. **Highlight Correct Answer**:
   * Change the background color of the correct answer button to green when clicked and others to red if wrong.
3. **Disable Buttons After Answer**:
   * Once an answer is selected, disable all buttons to prevent further clicks.
4. **Implement Using**:
   * **Plain JavaScript**: Use querySelectorAll, dataset, addEventListener, and classList.
   * **jQuery**: Use .on(), .attr(), .data(), and .css().

**Question 2:**

Create a **User Registration Form** that validates user input and displays error messages dynamically.

**Initial HTML Structure:**

<form id="registration-form">

<input type="text" id="username" placeholder="Username">

<input type="email" id="email" placeholder="Email">

<input type="password" id="password" placeholder="Password">

<button type="submit" id="register-btn">Register</button>

</form>

<p id="error-messages"></p>

**Requirements:**

1. **Input Validation**:
   * Check if the username is at least 5 characters, email contains "@" and ".", and password is at least 8 characters.
2. **Display Error Messages**:
   * Display all error messages in the #error-messages paragraph without refreshing the page.
3. **Dynamic Error Styling**:
   * Apply a red border to invalid inputs and a green border to valid ones dynamically.
4. **Clear Errors on Success**:
   * If all fields are valid, clear the error messages and display "Registration Successful!" in green.
5. **Implement Using**:
   * **Plain JavaScript**: Use addEventListener, preventDefault, value, classList, and innerHTML.
   * **jQuery**: Use .on(), .val(), .html(), and .css().

**Question 3:**

Develop an **Image Gallery** where users can toggle captions on and off using buttons.

**Initial HTML Structure:**

<div id="gallery">

<div class="image-container">

<img src="image1.jpg" alt="Image 1">

<p class="caption">Caption for Image 1</p>

<button class="toggle-caption">Toggle Caption</button>

</div>

<div class="image-container">

<img src="image2.jpg" alt="Image 2">

<p class="caption">Caption for Image 2</p>

<button class="toggle-caption">Toggle Caption</button>

</div>

</div>

**Requirements:**

1. **Toggle Caption**:
   * When the "Toggle Caption" button is clicked, hide or show the corresponding caption using a smooth effect.
2. **Change Button Text**:
   * Change the button text between "Show Caption" and "Hide Caption" dynamically.
3. **Styling on Toggle**:
   * Add a border to the image when the caption is visible and remove it when hidden.
4. **Implement Using**:
   * **Plain JavaScript**: Use style.display, textContent, and addEventListener.
   * **jQuery**: Use .toggle(), .text(), and .css().

**Question 4:**

Create a **Countdown Timer** that allows users to start, pause, and reset the timer dynamically.

**Initial HTML Structure:**

<div id="timer-container">

<h2 id="timer-display">60</h2>

<button id="start-btn">Start</button>

<button id="pause-btn">Pause</button>

<button id="reset-btn">Reset</button>

</div>

**Requirements:**

1. **Start Timer**:
   * Start counting down from 60 seconds when the "Start" button is clicked.
2. **Pause Timer**:
   * Pause the countdown when the "Pause" button is clicked without resetting the current value.
3. **Reset Timer**:
   * Reset the timer to 60 seconds when the "Reset" button is clicked.
4. **Disable/Enable Buttons**:
   * Disable the "Start" button while the timer is running. Enable "Pause" and "Reset" as appropriate.
5. **Implement Using**:
   * **Plain JavaScript**: Use setInterval, clearInterval, innerHTML, and event listeners.
   * **jQuery**: Use .text(), .on(), .attr(), and setTimeout() equivalents.

**Question 5:**

You are required to create a **Dynamic Task List** for a project management tool using both plain JavaScript and jQuery. Each task should be associated with a status (e.g., Pending, In Progress, Completed).

**Initial HTML Structure:**

<div id="task-manager">

<input type="text" id="task-input" placeholder="Enter a new task">

<select id="status-selector">

<option value="pending">Pending</option>

<option value="in-progress">In Progress</option>

<option value="completed">Completed</option>

</select>

<button id="add-task-btn">Add Task</button>

<ul id="task-list"></ul>

</div>

**Requirements:**

1. **Add Task**:
   * On clicking the "Add Task" button, add a new task (<li> element) to the #task-list.
   * The task should display the input text and the selected status as a label, e.g., "Task 1 - [Pending]".
2. **Color Coding**:
   * Use CSS styles to apply specific colors to tasks based on their status:
     + Pending: Orange
     + In Progress: Blue
     + Completed: Green
3. **Delete Task**:
   * Add a "Remove" button next to each task. Clicking it should delete the task.
4. **Highlight on Hover**:
   * Increase the font size and add a background color when the mouse hovers over any task.
5. **Implement Using**:
   * **Plain JavaScript**: Use addEventListener, createElement, appendChild, and CSS selectors.
   * **jQuery**: Use .append(), .css(), .on(), and attribute selectors.

**Question 6:**

Create a **User Profile Card System** where each profile displays a user’s name, role, and status using both plain JavaScript and jQuery.

**Initial HTML Structure:**

<div id="user-profiles">

<input type="text" id="user-name" placeholder="Enter user name">

<input type="text" id="user-role" placeholder="Enter user role">

<button id="add-user-btn">Add User</button>

<div id="profiles"></div>

</div>

**Requirements:**

1. **Add Profile**:
   * Create a new div for each user when the "Add User" button is clicked.
   * Use data-role and data-status attributes to store the user's role and status.
2. **Display Profile Information**:
   * Display the user's name and role within the div.
   * Apply a class to each profile based on their role (e.g., "admin" for Admin, "developer" for Developer).
3. **Remove Profile**:
   * Add a "Remove Profile" button within each user card to delete the profile.
4. **Change Role**:
   * Add an option to click on a profile and change its role by updating the data-role attribute dynamically.
5. **Implement Using**:
   * **Plain JavaScript**: Use getAttribute, setAttribute, classList.add, and querySelector.
   * **jQuery**: Use attribute selectors ([data-role]), .attr(), .addClass(), .remove().

**Topic AJAX:**

**Question 1:**

**What will be the output in the #user-info div?**

Consider the following AJAX GET request using plain JavaScript:

<div id="user-info"></div>

<script>

const xhr = new XMLHttpRequest();

xhr.open('GET', 'https://jsonplaceholder.typicode.com/users/1', true);

xhr.onreadystatechange = function () {

if (xhr.readyState === 4 && xhr.status === 200) {

const user = JSON.parse(xhr.responseText);

document.getElementById('user-info').innerHTML = `Name: ${user.name}, Email: ${user.email}`;

}

};

xhr.send();

</script>

**Question 2:**

**What will be the output in the #response-message div after the POST request?**

Consider the following AJAX POST request using jQuery:

<div id="response-message"></div>

<script src="https://code.jquery.com/jquery-3.6.0.min.js"></script>

<script>

$.post('https://jsonplaceholder.typicode.com/posts', {

title: 'Web Engineering',

body: 'Learning AJAX with jQuery',

userId: 1

})

.done(function (data) {

$('#response-message').html(`Post ID: ${data.id}, Title: ${data.title}`);

})

.fail(function () {

$('#response-message').html('Error: Failed to submit data');

});

</script>

**Question 3:**

Identify and fix the errors in the following code:

<script>

const xhr = new XMLHttpRequest();

xhr.open('POST', 'https://jsonplaceholder.typicode.com/posts');

xhr.setRequestHeader('Content-Type', 'application-json'); // Error?

xhr.onload = function () {

if (xhr.status = 200) { // Error?

const response = JSON.parse(xhr.responseText);

console.log('Response ID:', response.id);

}

};

xhr.send(JSON.stringify({

title: 'Bug in the code',

body: 'Fix it!',

userId: 1

}));

</script>

**Question 4:**

The following jQuery AJAX request has an issue. Identify and correct it:

<div id="data"></div>

<script src="https://code.jquery.com/jquery-3.6.0.min.js"></script>

<script>

$.ajax({

type: 'GET',

url: 'https://jsonplaceholder.typicode.com/users/invalid',

success: function (response) {

$('#data').html(`Name: ${response[0].name}`);

},

error: function () {

console.log('Success');

}

});

</script>

**Question 5:**

What will be logged to the console when the following code runs?

<script>

async function fetchData() {

try {

const response = await fetch('https://jsonplaceholder.typicode.com/posts/1000'); // Non-existent

if (!response.ok) throw new Error('Data not found!');

const data = await response.json();

console.log('Post Title:', data.title);

} catch (error) {

console.error('Error:', error.message);

}

}

fetchData();

</script>

**Topic Node:**

You are tasked with creating a **Node.js application** that interacts with a **MongoDB database** to perform the following tasks:

1. **Insert a new user document** into a MongoDB collection.
2. **Retrieve the user by userId** from the MongoDB collection.
3. Use three different approaches to handle these asynchronous operations:
   * **Callback Version** (using MongoDB native driver).
   * **Promise Version** (using MongoDB native driver with Promises).
   * **Async/Await Version** (using async/await syntax).

**Topic Express:**

**Question 1:**

You are tasked with creating a **RESTful API** for managing a blogging platform using **Express.js**. The platform has the following functionality:

* **User Registration**: A user can register with their details (username, email, password).
* **User Login**: A user can log in using their credentials and receive a JWT token.
* **Create, Read, Update, Delete (CRUD)**: Users can create, read, update, and delete blog posts.

You need to implement the following:

1. **User Authentication Middleware**: Before allowing users to access the blog post CRUD routes, ensure they are authenticated using a JWT.
2. **Express Routing**: Implement all required routes for user registration, login, and blog post CRUD operations.
3. **Authorization Middleware**: Implement a middleware that checks if the logged-in user is the author of the blog post they want to update or delete.

**Requirements:**

1. Write the route handlers for:
   * POST /register: Register a new user.
   * POST /login: Log in and issue a JWT token.
   * GET /posts: Fetch all posts (only accessible by authenticated users).
   * POST /posts: Create a new post (only accessible by authenticated users).
   * PUT /posts/:id: Update a post (only the author of the post can update it).
   * DELETE /posts/:id: Delete a post (only the author of the post can delete it).
2. Implement the necessary middleware:
   * authMiddleware: This middleware should verify the JWT token from the request header and check if the user is authenticated.
   * authorizationMiddleware: This middleware should verify that the logged-in user is the owner of the blog post before updating or deleting it.
3. Make sure to handle errors properly using try/catch blocks in your route handlers.

**Question 2:**

You are developing an application with **JWT-based authentication** where users can log in and stay logged in by using **Refresh Tokens**. The application includes the following routes:

* **POST /login**: User logs in and gets an access token and a refresh token.
* **POST /refresh-token**: User uses the refresh token to get a new access token.

**Requirements:**

1. Create a JWT authentication mechanism that:
   * Issues a short-lived access token (expires in 15 minutes).
   * Issues a long-lived refresh token (expires in 7 days).
2. Implement a route /login where users authenticate by sending their username and password. If authentication is successful, issue both an **access token** and a **refresh token**. Store the refresh token securely in the database.
3. Implement a /refresh-token route that allows users to send their refresh token. The server should verify the refresh token and, if valid, issue a new access token. The refresh token should also be securely stored in the database.
4. Ensure proper error handling for cases where the refresh token is invalid or expired.
5. **Security Considerations**:
   * Implement token expiry for access tokens and refresh tokens.
   * Make sure to implement secure storage and handling of refresh tokens (e.g., store them securely in a database or in an HTTP-only cookie).

**Question 3:**

You are building a **production-grade application** using **Express.js**, and you need to implement custom **middleware** for:

* **Error Handling**: Catching errors and sending standardized error responses to the client.
* **Logging**: Logging every incoming request, including request path, HTTP method, and the status of the response.

**Requirements:**

1. Write a custom **error-handling middleware** that catches any errors thrown in your route handlers and sends a standardized error response with the status code, error message, and stack trace (only in development mode).
2. Implement a **request logging middleware** that logs each incoming request's method, path, and time of request, including the response status code after the request is processed.
3. Demonstrate how to use both middlewares in your Express app. For example:
   * Add a GET /hello route that deliberately throws an error to test your error handling middleware.
   * Add a POST /submit route that returns a success message to test your logging middleware.
4. Write tests for your middleware to verify that:
   * The error handling middleware correctly sends error responses.
   * The logging middleware logs requests correctly.

**Topic React:**

**Question 1.**

You are building a **Task Management Application** where users can manage tasks. The application allows users to add tasks, mark them as complete, and delete them.

**Requirements:**

1. **Parent Component (TaskApp)**:
   * The **TaskApp** component will hold the state of tasks using useState and initialize it as an empty array.
   * It should have an **input field** to add new tasks, and a **button** to add the task when clicked.
   * It should display a list of tasks, passing each task as a prop to a **TaskItem** component.
2. **TaskItem Component**:
   * The **TaskItem** component will receive the task and a function to mark the task as complete via props.
   * Display the task text along with a checkbox to mark it as complete.
   * When the checkbox is clicked, the **TaskItem** should call the passed function to mark it as complete, and visually update the item (e.g., strike-through the text).
   * It should also have a **Delete** button that calls a function passed from the parent to remove the task from the list.
3. **TaskInput Component**:
   * The **TaskInput** component will handle the user input for new tasks.
   * It should take the user input as a prop from the **TaskApp** component and call a function to add a new task when the user submits the form.

**Question 2.**

You are building a **Movie Search Application** where users can search for movies by title. When the user enters a movie title in the search input and clicks the search button, the app fetches movie data from the **OMDB API** (https://www.omdbapi.com/) and displays the search results.

**Requirements:**

1. **App Component**:
   * The **App** component should contain an **input field** where the user can type the title of a movie.
   * There should be a **Search Button** that triggers a search when clicked.
   * Display the list of movie results (movie poster, title, year of release, and a short description) after the user performs the search.
   * If no results are found, show a "No movies found" message.
2. **MovieSearch Hook**:
   * Create a custom hook useMovieSearch that:
     + Accepts the movie title as a parameter.
     + Fetches movie data from the OMDB API when the title changes (use useEffect to trigger the search whenever the title is updated).
     + Returns the results, loading state, and error state.
3. **Error Handling**:
   * Display an error message if there is an issue with the API request or if the movie title is not found.
4. **UI/UX**:
   * While the data is loading, display a loading spinner.
   * Show the list of movies once the data is successfully fetched.
   * If no movies match the search, show a "No movies found" message.

**Question 3.**

You are tasked with building a **multi-page React application** with **React Router** for navigation. The app should have the following pages:

* Home Page
* About Page
* Contact Page
* Not Found Page (for invalid routes)

**Requirements:**

1. **Set up React Router**:
   * Use BrowserRouter to wrap the entire app.
   * Create routes for:
     + /: Home page
     + /about: About page
     + /contact: Contact page
     + /\*: A catch-all route that will display a "Page Not Found" message for any invalid URL.
2. **HomePage Component**:
   * The **HomePage** should have a welcome message and buttons to navigate to the About and Contact pages.
3. **AboutPage Component**:
   * The **AboutPage** should contain information about your app or company.
4. **ContactPage Component**:
   * The **ContactPage** should contain a form where users can input their name and message and submit it (though no backend is required for this).
5. **Not Found Page**:
   * Display a **404** error page with a message that says "Page Not Found" when the user navigates to a non-existent route.

**Topic ContextApi, Zustand, Redux:**

**Question 1.**

You are tasked with building a **User Authentication** system in **React** using the **Context API** to manage user login status globally. The app should have the following functionality:

* A user can log in with their credentials (you can use a mock authentication function).
* Once logged in, the app should display a welcome message and the user's profile.
* The app should provide a button to log out, and when the user logs out, the welcome message should be replaced by a login form.

**Requirements:**

1. **UserContext**:
   * Create a **UserContext** using React.createContext() to hold the state of the user (logged in or logged out).
   * The context should also include a method to update the login status (login and logout functions).
2. **ContextProvider**:
   * Create a **UserProvider** component that will provide the context value to the rest of the app using Context.Provider.
3. **App Component**:
   * The **App** component should consume the **UserContext** and display either the login form or the user profile, depending on whether the user is logged in.
4. **Login Form**:
   * A form with an input for the username and a submit button. When submitted, the form should call the login function from the context and update the UI to show the user's profile.
5. **User Profile**:
   * Display a welcome message and a button to log out. When clicked, the logout function should be called, and the app should revert to showing the login form.

**Question 2.**

You are building a **theme switcher** app using the **Context API**. The app should allow the user to toggle between light and dark themes, and the theme should be applied globally to all components.

**Requirements:**

1. **ThemeContext**:
   * Create a **ThemeContext** using React.createContext() to manage the current theme (light or dark).
   * The context should have a function (toggleTheme) that toggles between light and dark themes.
2. **ContextProvider**:
   * Create a **ThemeProvider** component that will wrap the entire application and provide the current theme and the toggle function.
3. **App Component**:
   * The **App** component should consume the **ThemeContext** and apply the appropriate theme (e.g., className="light-theme" or className="dark-theme") to the body or a container div based on the current theme.
4. **Toggle Button**:
   * Add a button to toggle between light and dark themes. When clicked, it should call the toggleTheme function from the context.
5. **Styling**:
   * Use basic CSS to differentiate between the light and dark themes (e.g., change background color and text color).

**Question 3.**

You are tasked with building a **User Profile Management** app using **Zustand** for state management. The app should allow users to update their profile information, including their name, email, and bio. The profile information should be managed globally with Zustand, and users should be able to update and view the profile data.

**Requirements:**

1. **Store Setup**:
   * Create a Zustand store to manage the profile state. The store should include:
     + profile: An object to store the user's profile information (e.g., name, email, and bio).
     + updateProfile: A function that allows updating the profile information (e.g., setting the user's name, email, or bio).
2. **App Component**:
   * In the **App** component, consume the Zustand store to display the current profile information (name, email, bio).
   * Provide input fields for the user to update their name, email, and bio.
   * When the user updates any of the fields, the profile information in the Zustand store should be updated.
3. **Profile Form**:
   * Create a **ProfileForm** component that allows users to input their name, email, and bio. The component should:
     + Take the current profile data from the Zustand store and populate the input fields with the values.
     + Allow the user to edit the fields and update the profile by calling the updateProfile function.
4. **Display Updated Information**:
   * After the user updates their profile, the profile information displayed in the **App** component should be updated in real-time.

**Question 4.**

You are tasked with building a **Todo List** application using **Redux** for state management. The application should allow users to add, delete, and toggle the completion status of todo items. The state for the todo list will be managed globally using Redux, and you will use the **React-Redux** library to connect the state with the React components.

**Requirements:**

1. **State Structure**:
   * The Redux store should contain a todos array where each item is an object with the following properties:
     + id: A unique identifier for the todo (can be a string or number).
     + text: The text content of the todo.
     + completed: A boolean indicating whether the todo is completed or not.
2. **Actions**:
   * Define three actions:
     + ADD\_TODO: Adds a new todo to the list.
     + REMOVE\_TODO: Removes a todo from the list by its id.
     + TOGGLE\_TODO: Toggles the completed status of a todo by its id.
3. **Reducers**:
   * Create a **todoReducer** that handles the above actions. Ensure that the reducer updates the state immutably (i.e., without mutating the original state).
4. **React Components**:
   * **App Component**: The main component that renders the list of todos.
     + It should display the todos and show whether they are completed or not.
   * **TodoItem Component**: Renders individual todo items and allows toggling the completion status and deleting a todo.
   * **AddTodo Component**: Allows the user to input a new todo and add it to the list.
5. **Integration with Redux**:
   * Use the connect function from **React-Redux** to map the Redux state to props and dispatch the actions.
   * Use the **useDispatch** hook to dispatch actions like adding a new todo and toggling completion status.
6. **UI**:
   * The todos should be listed with a checkbox indicating whether they are completed.
   * Add a button next to each todo to allow it to be deleted.