#2

Prompt for ASP.NET Core Web Application with User Authentication and Role-Based Access
Objective: Write the code for an ASP.NET Core web application that implements a login system with role-based access control.
Requirements:
Home Page:
The home page should contain:
A textbox for the username.
A textbox for the password.
A login button.
Login Functionality:
When the user enters their credentials and clicks the login button:
Control should transfer to a C# method on the server side.
The method should check if the username and password are both "admin" (for admin privileges) or "guest" (for guest privileges).
If the credentials are "admin/admin", return "admin" to the client.
If the credentials are "guest/guest", return "guest" to the client.
If the credentials are incorrect, return an error message.
Dashboard Page:
Upon successful login, redirect the user to a dashboard page.
The dashboard should include:
A textbox for input.
Three buttons: Add, Edit, and Delete.
Button functionality based on user privileges:
If the user is an admin, all buttons (Add, Edit, Delete) should be enabled.
If the user is a guest, only the Add button should be enabled; Edit and Delete buttons should be disabled.

To implement an ASP.NET Core Web Application with user authentication and role-based access, follow the steps below. The application will check for predefined credentials (admin/guest) and adjust the dashboard functionality based on the user role.

Step 1: Create the ASP.NET Core Web Application

Start by creating a new ASP.NET Core Web Application project using Visual Studio or the .NET CLI.

Step 2: Set Up the Controllers

1. **HomeController**: This controller will handle the login logic.
2. **DashboardController**: This controller will manage the dashboard functionality, based on user roles.

Step 3: Home Page (Login Page)

In the `HomeController`, create an action to handle the login logic and return the appropriate response based on the credentials entered.

HomeController.cs:

```
csharp
```

```csharp
using Microsoft.AspNetCore.Mvc;

namespace RoleBasedAccess.Controllers
{
    public class HomeController : Controller
    {
        [HttpGet]
        public IActionResult Index()
        {
            return View();
        }

        [HttpPost]
        public IActionResult Login(string username, string password)
        {
            // Check if the username and password match the predefined values
            if (username == "admin" && password == "admin")
            {
                // Set the user role to 'admin'
                TempData["Role"] = "admin";
                return RedirectToAction("Dashboard", "Dashboard");
            }
            else if (username == "guest" && password == "guest")
            {
                // Set the user role to 'guest'
                TempData["Role"] = "guest";
                return RedirectToAction("Dashboard", "Dashboard");
            }
            else
            {
                // If credentials are incorrect, show an error message
                ViewBag.ErrorMessage = "Invalid credentials!";
                return View("Index");
            }
        }
    }
}
```

Step 4: Dashboard Page

The dashboard page will check the user's role (stored in `TempData`) and enable or disable buttons accordingly.

DashboardController.cs:

```
csharp
```

```csharp
using Microsoft.AspNetCore.Mvc;

namespace RoleBasedAccess.Controllers
{
    public class DashboardController : Controller
    {
        public IActionResult Dashboard()
        {
            // Get the role of the current user (admin or guest)
            string role = TempData["Role"]?.ToString();
```

```
                if (role == null)
                {
                    return RedirectToAction("Index", "Home");
                }

                ViewBag.Role = role;
                return View();
            }
        }
    }
```

Step 5: Create Views for Login and Dashboard

Views/Home/Index.cshtml (Login Page):

```html
@{
    ViewData["Title"] = "Login";
}

<h2>Login</h2>

<form method="post" action="/Home/Login">
    <div>
        <label for="username">Username</label>
        <input type="text" id="username" name="username" required />
    </div>
    <div>
        <label for="password">Password</label>
        <input type="password" id="password" name="password" required />
    </div>
    <button type="submit">Login</button>
</form>

@if (ViewBag.ErrorMessage != null)
{
    <div style="color: red;">@ViewBag.ErrorMessage</div>
}
```

Views/Dashboard/Dashboard.cshtml (Dashboard Page):

```html
@{
    ViewData["Title"] = "Dashboard";
    string role = ViewBag.Role?.ToString();
}

<h2>Dashboard</h2>

<div>
    <label for="inputText">Input</label>
    <input type="text" id="inputText" />
</div>

<div>
    <button id="btnAdd" @disabled="@(role == "guest" ? "disabled" : "")">Add</button>
    <button id="btnEdit" @disabled="@(role == "guest" ? "disabled" : "")">Edit</button>
    <button id="btnDelete" @disabled="@(role == "guest" ? "disabled" : "")">Delete</button>
</div>

<p>Logged in as: @role</p>
```

Step 6: Configure the Startup (Program.cs)

Make sure to set up routing and add the necessary services in the `Program.cs` file.

```csharp
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;

namespace RoleBasedAccess
{
    public class Program
    {
        public static void Main(string[] args)
        {
            CreateHostBuilder(args).Build().Run();
        }

        public static IHostBuilder CreateHostBuilder(string[] args) =>
            Host.CreateDefaultBuilder(args)
                .ConfigureWebHostDefaults(webBuilder =>
                {
                    webBuilder.UseStartup<Startup>();
                });
    }
}
```

Startup.cs (if needed):

```csharp
using Microsoft.AspNetCore.Builder;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;

namespace RoleBasedAccess
{
    public class Startup
    {
        public void ConfigureServices(IServiceCollection services)
        {
            services.AddControllersWithViews();
        }

        public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
        {
            if (env.IsDevelopment())
            {
                app.UseDeveloperExceptionPage();
            }
            else
            {
                app.UseExceptionHandler("/Home/Error");
                app.UseHsts();
            }

            app.UseHttpsRedirection();
            app.UseStaticFiles();

            app.UseRouting();

            app.UseEndpoints(endpoints =>
            {
                endpoints.MapControllerRoute(
                    name: "default",
                    pattern: "{controller=Home}/{action=Index}/{id?}");
            });
        }
    }
}
```

Explanation:

1. **Login Page** (`Index.cshtml`):
    - A form allows the user to enter their username and password.
    - Upon submission, it sends a `POST` request to the `Login` method in the `HomeController`, which checks the credentials.
    - If valid credentials are provided, the user is redirected to the `Dashboard` page; otherwise, an error message is displayed.
2. **Dashboard Page** (`Dashboard.cshtml`):
    - Displays a simple form with a textbox and three buttons (`Add`, `Edit`, and `Delete`).
    - The visibility and interactivity of the buttons are determined by the user's role (`admin` or `guest`). If the role is `guest`, only the `Add` button is enabled; the `Edit` and `Delete` buttons are disabled.
    - The `TempData["Role"]` value is used to store and retrieve the user's role during the session.

Step 7: Run the Application

1. Run the application and navigate to the login page.
2. Try logging in with the credentials:
    - `admin/admin` to get full access (all buttons enabled).
    - `guest/guest` to get limited access (only the Add button enabled).
3. The system will redirect you to the dashboard page based on the role after successful login.

This approach uses simple in-memory role-based logic for the demonstration. In production, you would typically use a more secure authentication mechanism, such as ASP.NET Core Identity, JWT, or OAuth, to manage users and roles.

Make this same as Asp.Net ReactJs

To implement the same functionality with ASP.NET Core as the backend and ReactJS as the frontend, you will need to create two separate projects: one for the ASP.NET Core Web API (backend) and one for the ReactJS application (frontend).

Here is how you can structure the solution:

Step 1: Set up the ASP.NET Core Web API (Backend)

This will handle the authentication logic (login) and role-based access control.

1.1 Create an ASP.NET Core Web API Project

In Visual Studio or using the .NET CLI, create a new **ASP.NET Core Web API** project.

```bash
dotnet new webapi -n RoleBasedAccessApi
```

1.2 Create a Login Controller

In `Controllers`, create a `AuthController` that will handle login functionality.

```csharp
using Microsoft.AspNetCore.Mvc;

namespace RoleBasedAccessApi.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class AuthController : ControllerBase
    {
        [HttpPost("login")]
        public IActionResult Login([FromBody] UserCredentials credentials)
        {
            if (credentials.Username == "admin" && credentials.Password == "admin")
            {
                return Ok(new { Role = "admin" });
            }
            else if (credentials.Username == "guest" && credentials.Password == "guest")
            {
                return Ok(new { Role = "guest" });
            }
            else
            {
                return Unauthorized(new { Message = "Invalid credentials!" });
            }
        }
    }

    public class UserCredentials
    {
        public string Username { get; set; }
        public string Password { get; set; }
    }
}
```

- The `AuthController` provides a `POST /api/auth/login` endpoint that accepts a `Username` and `Password` and returns the role (`admin` or `guest`).

1.3 Enable CORS

In `Startup.cs` or `Program.cs` (depending on your .NET version), enable CORS to allow the React frontend to make requests to the API.

```csharp
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddCors(options =>
        {
            options.AddPolicy("AllowAll", builder =>
            {
                builder.AllowAnyOrigin().AllowAnyMethod().AllowAnyHeader();
            });
        });

        services.AddControllers();
    }

    public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
    {
        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
        }

        app.UseCors("AllowAll");

        app.UseRouting();

        app.UseEndpoints(endpoints =>
        {
            endpoints.MapControllers();
        });
    }
}
```

Step 2: Set up the ReactJS Frontend

2.1 Create a React App

Create a React application using `` `create-react-app` ``.

```bash
npx create-react-app role-based-access-react
cd role-based-access-react
```

2.2 Install Axios for HTTP Requests

Install `` `axios` `` to handle HTTP requests from React to the ASP.NET Core API.

```bash
npm install axios
```

2.3 Create the Login Component

Create a login form where the user can enter their username and password.

`` `src/components/Login.js` ``:

```
javascript


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

  const Login = ({ onLoginSuccess }) => {
    const [username, setUsername] = useState('');
    const [password, setPassword] = useState('');
    const [errorMessage, setErrorMessage] = useState('');

    const handleLogin = async (event) => {
      event.preventDefault();

      try {
        const response = await axios.post('http://localhost:5000/api/auth/login', { username,
  password });
        const role = response.data.role;
        onLoginSuccess(role);
      } catch (error) {
        setErrorMessage('Invalid credentials!');
      }
    };

    return (
      <div>
        <h2>Login</h2>
        <form onSubmit={handleLogin}>
          <div>
            <label>Username</label>
            <input type="text" value={username} onChange={(e) => setUsername(e.target.value)}
  required />
          </div>
          <div>
            <label>Password</label>
            <input type="password" value={password} onChange={(e) => setPassword(e.target.valu
  e)} required />
          </div>
          <button type="submit">Login</button>
        </form>
        {errorMessage && <div style={{ color: 'red' }}>{errorMessage}</div>}
      </div>
    );
  };

  export default Login;
```

- This component contains a form with a username and password input field. When the user submits the form, it makes a `` `POST` `` request to the backend (`` `/api/auth/login` ``) to check the credentials.

2.4 Create the Dashboard Component

The dashboard will display buttons and functionalities based on the user's role (either `` `admin` `` or `` `guest` ``).

`` `src/components/Dashboard.js` ``:

```
javascript


  import React from 'react';

  const Dashboard = ({ role }) => {
    return (
      <div>
        <h2>Dashboard</h2>
        <div>
          <label>Input</label>
          <input type="text" />
        </div>
        <div>
          <button disabled={role === 'guest'}>Add</button>
          <button disabled={role === 'guest'}>Edit</button>
          <button disabled={role === 'guest'}>Delete</button>
        </div>
        <p>Logged in as: {role}</p>
      </div>
    );
  };

  export default Dashboard;
```

- If the role is `` `guest` ``, only the "Add" button will be enabled, while the "Edit" and "Delete" buttons will be disabled.

2.5 Combine Components in App.js

In the `` `App.js` `` file, manage the login state and display either the login form or the dashboard based on the user's role.

`` `src/App.js` ``:

```javascript
import React, { useState } from 'react';
import Login from './components/Login';
import Dashboard from './components/Dashboard';

function App() {
  const [role, setRole] = useState(null);

  const handleLoginSuccess = (role) => {
    setRole(role);
  };

  return (
    <div>
      {role ? (
        <Dashboard role={role} />
      ) : (
        <Login onLoginSuccess={handleLoginSuccess} />
      )}
    </div>
  );
}

export default App;
```

- When the user logs in successfully, the `` `role` `` state is set to `` `admin` `` or `` `guest` ``, and the appropriate component (`` `Dashboard` ``) is displayed.

Step 3: Run the Application

1. **Backend (ASP.NET Core API)**:
   - Navigate to the `` `RoleBasedAccessApi` `` directory and run the backend API.

```bash
dotnet run
```

The API will be running on `` `http://localhost:5000` ``.

2. **Frontend (React)**:
   - Navigate to the `` `role-based-access-react` `` directory and start the React development server.

```bash
npm start
```

The React app will be running on `` `http://localhost:3000` ``.

Step 4: Test the Application

1. Open your browser and go to `` `http://localhost:3000` ``.
2. Log in using the credentials:
   - **Admin**: `` `admin/admin` `` (All buttons enabled)
   - **Guest**: `` `guest/guest` `` (Only Add button enabled)
3. The system will display the dashboard based on the logged-in role and show the correct buttons.

Summary

- **Backend**: ASP.NET Core Web API handles the login and authentication logic, validating credentials and sending back the user role.
- **Frontend**: ReactJS displays the login form and the dashboard. The dashboard adapts based on the user's role (admin/guest) and enables/disables buttons accordingly.

This setup provides a simple implementation of role-based access control using ReactJS and ASP.NET Core.