**Front-end**

**Page Authorization**

* The code imports the necessary modules: React, jwt\_decode, and useNavigate from react-router-dom.
* The Home() function is defined and exported as the default module.
* The useNavigate hook is used to set up a navigation function to be called later.
* The localStorage method is used to get the value of the token key.
* The jwt\_decode module is used to decode the token, and the decoded value is stored in the decoded variable.
* An error is caught if the token is invalid.
* The handleLogout() function is defined, which removes the token from local storage and navigates the user back to the home page.
* An isAuthenticated variable is defined, which checks if the decoded object exists and if the UserName property is equal to 'admin'.
* Depending on the value of isAuthenticated, either a greeting and logout button or an access denied message, login prompt, and logout button are displayed to the user.

**Login page Authentication**

* This is a React component that renders a login form with different login options such as email and password login, Google OAuth login, and Facebook OAuth login.
* It first imports the required dependencies: React, useState hook, CSS stylesheet, Google and Facebook login components from respective libraries, react-router-dom, and axios.
* Then, it declares a functional component named "Login" that returns a JSX. It defines three state variables using the useState hook: "username", "password", and "token". "username" and "password" are set to empty strings as initial values, and "token" is set to null.
* The component then uses the "useNavigate" hook from the react-router-dom library to get a reference to the navigate function.
* Next, it declares a function named "ProceedLoginusiingAPI" that is triggered when the email and password form is submitted. This function calls an API endpoint using the fetch method and makes a POST request with the inputted username and password as parameters. It then saves the returned token in the local storage and navigates to the home page using the "navigate" function.
* The component then uses the "useGoogleLogin" hook from the "@react-oauth/google" library to declare a function named "login" that is triggered when the Google OAuth login button is clicked. This function calls the "axios.get" method to make a GET request to the Google OAuth userinfo API with the "tokenResponse" parameter. It then logs the returned data to the console.
* The component then declares a function named "responseFacebook" that is triggered when the Facebook OAuth login button is clicked. This function extracts the "accessToken" and "username" from the "response" object and makes a GET request to the Facebook API using the "axios.get" method with the extracted parameters. It then logs the returned data to the console.
* Finally, the component renders a login form with two input fields for email and password, and a "submit" button that triggers the "ProceedLoginusiingAPI" function when clicked. It also renders two OAuth login buttons for Google and Facebook, respectively.
* The Google login button is a custom button component that uses the "useGoogleLogin" hook and includes the "FcGoogle" icon from the "react-icons/fc" library. The Facebook login button is a component imported from the "react-facebook-login" library and includes the "fa-brands fa-facebook" icon.
* By using the code below user can register from fron-end.
* // Define register function that will be called when user clicks "Create Account" button
* const register = async () => {
* // Get the email and password input values from the HTML form
* const email = document.getElementById('email').value;
* const password = document.getElementById('password').value;
* try {
* // Send a POST request to the server with the email and password data
* const response = await fetch('/api/register', {
* method: 'POST',
* headers: {
* 'Content-Type': 'application/json'
* },
* body: JSON.stringify({ email, password })
* });
* // Parse the response as JSON
* const result = await response.json();
* // If the response is not OK (i.e. there was an error), throw an error with the error message from the server
* if (!response.ok) {
* const errorMessage = result.message || 'Failed to create account';
* throw new Error(errorMessage);
* }
* // Display success message
* alert('Account created successfully!');
* } catch (error) {
* // If there was an error (e.g. server error or network error), display an error message to the user
* const errorMessage = error.message || 'Failed to create account';
* alert(`Error: ${errorMessage}`);
* }
* };

Login Form-----------

<form onSubmit={register}>

                        <div className="form-group " >

                            <input type="email" id="email" name="email" required className="form-control login-info"  placeholder="Enter Email"  />

                        </div>

                        <div className="form-group ">

                            <input type="password" id="password" name="password" required className="form-control login-info"  placeholder="Password" />

                        </div>

                        <button className="btn btn-success font-weight-bold" style={{width:"240px", borderRadius:"8px"}} type="submit">Create Account</button>

                        </form>

**Index.js**

* The first two lines import the React and ReactDOM libraries. React is the core library for building user interfaces in React, and ReactDOM is the library that allows you to render React components in the DOM.
* The third line imports the index.css file, which contains CSS styles for the app.
* The fourth line imports the App component from the App.js file. This is the main component of the app and contains the logic for rendering the UI.
* The fifth line imports the reportWebVitals function, which can be used to measure performance metrics for the app.
* The sixth line imports the GoogleOAuthProvider component from the @react-oauth/google library. This component provides a way to authenticate users with their Google accounts.
* The seventh line creates a new root for the app using ReactDOM.createRoot() and passes in the root DOM element with an ID of root.
* The eighth line calls the render() method on the root object and renders the App component wrapped in a GoogleOAuthProvider component with a clientId prop of the Google OAuth client ID. This component will act as a parent for the App component and provide authentication capabilities.
* The ninth line calls the reportWebVitals() function, which can be used to measure performance metrics for the app.

**App.js**

* The first line imports several functions from the 'react-router-dom' library, which provides tools for creating routes in a React application.
* The next two lines import the CSS styles for the App component and two other components called Login and Home, respectively.
* The App function defines a React component that renders the overall structure of the application.
* Inside the App function, the BrowserRouter component is used to wrap the application, indicating that it will be using client-side routing.
* The Routes component is used to define the various routes in the application.
* The first Route component specifies that the Login component should be rendered when the path is '/', which is the default path for the application.
* The second Route component specifies that the Home component should be rendered when the path is 'home'.
* Finally, the App component is exported so that it can be used elsewhere in the application.

**Back-end**

**Auth Controller**

* This is a C# ASP.NET Core controller class that provides authentication functionality through the handling of login, register and register-admin requests. It uses Identity for managing users, roles and authentication.
* The code starts by importing the required namespaces for the classes used, including the ASP.NET Core Identity and JWT token classes.
* The AuthenticateController class is defined, which inherits from the ControllerBase class and is decorated with the [Route("api/[controller]")] and [ApiController] attributes. The controller implements two HTTP POST methods: Login and Register, which handle requests for logging in and registering a new user, respectively. Additionally, the controller implements another HTTP POST method: RegisterAdmin, which handles requests for registering a new administrator.
* In the constructor of the class, dependencies are injected using the UserManager<IdentityUser>, RoleManager<IdentityRole> and IConfiguration interfaces.
* The Login method accepts a LoginModel object as input, which contains the username and password of the user attempting to log in. It then attempts to find the user by the provided username using the UserManager object. If the user exists and the provided password is correct, the method retrieves the roles associated with the user, creates the claims for the JWT token, adds the user roles to the claims and creates a new JWT token using the GetToken method. The method then returns the token and its expiration date in the response if the login attempt is successful; otherwise, it returns a 401 Unauthorized error.
* The Register method accepts a RegisterModel object as input, which contains the email, username and password of the user attempting to register. It first checks if a user with the provided email already exists using the UserManager object. If a user already exists, the method returns a 500 Internal Server Error with an error message in the response. If no username is provided, the method sets a default username for the user. It then creates a new IdentityUser object with the provided user details, attempts to create the new user using the UserManager object and returns an error message in the response if the user creation fails. If the user creation is successful, the method returns a success message in the response.
* The RegisterAdmin method is similar to the Register method, but it additionally creates the admin and user roles if they do not already exist using the RoleManager object and adds the new user to both roles using the UserManager object.
* The GetToken method creates a new JwtSecurityToken object with the given claims and signing key and returns the token.

This class is responsible for handling user authentication and registration requests in an ASP.NET Core web application using Identity and JWT tokens.

private readonly UserManager<IdentityUser> \_userManager;

private readonly RoleManager<IdentityRole> \_roleManager;

private readonly IConfiguration \_configuration;

public AuthenticateController(

UserManager<IdentityUser> userManager,

RoleManager<IdentityRole> roleManager,

IConfiguration configuration)

{

\_userManager = userManager;

\_roleManager = roleManager;

\_configuration = configuration;

}

This code defines three dependencies that will be used throughout the class: a UserManager<IdentityUser>, a RoleManager<IdentityRole>, and an IConfiguration. These dependencies are injected into the constructor of the AuthenticateController class using the built-in dependency injection system in ASP.NET Core.

The UserManager is used to manage user accounts, while the RoleManager is used to manage roles (e.g., Admin, User) for the application. The IConfiguration is used to access configuration settings, such as the secret key used for signing JWT tokens and the issuer and audience values for the JWT token.

Handle login requests

[HttpPost]

[Route("login")]

public async Task<IActionResult> Login([FromBody] LoginModel model)

{

var user = await \_userManager.FindByNameAsync(model.Username);

if (user != null && await \_userManager.CheckPasswordAsync(user, model.Password))

{

var userRoles = await \_userManager.GetRolesAsync(user);

var authClaims = new List<Claim>

{

new Claim(ClaimTypes.Name, user.UserName),

new Claim("UserName", user.UserName.ToString()),

new Claim("Email", user.Email),

new Claim(JwtRegisteredClaimNames.Jti, Guid.NewGuid().ToString()),

};

foreach (var userRole in userRoles)

{

authClaims.Add(new Claim(ClaimTypes.Role, userRole));

}

var token = GetToken(authClaims);

return Ok(new

{

token = new JwtSecurityTokenHandler().WriteToken(token),

expiration = token.ValidTo

});

}

return Unauthorized();

}

This code handles login requests by checking if the provided username and password are valid. If so, it retrieves the roles associated with the user and creates a JWT token containing the user's information and roles. Finally, it returns the JWT token in the response.

[HttpPost]

[Route("register")]

public async Task<IActionResult> Register([FromBody] RegisterModel model)

{

var userExists = await \_userManager.FindByEmailAsync(model.Email);

if (userExists != null)

return StatusCode(StatusCodes.Status500InternalServerError, new Response { Status = "Error", Message = "User already exists!" });

model.SetDefaultUsername();

IdentityUser user = new()

{

Email = model.Email,

SecurityStamp = Guid.NewGuid().ToString(),

UserName = model.Username

};

var result = await \_userManager.CreateAsync(user, model.Password);

if (!result.Succeeded)

return StatusCode(StatusCodes.Status500InternalServerError, new Response { Status = "Error", Message = "User creation failed! Please check user details and try again." });

return Ok(new Response { Status = "Success", Message = "User created successfully!" });

}

This code handles user registration requests by checking if a user with the provided email already exists. If not, it creates a new IdentityUser object with the provided email, username, and password.

**Program.js**

This code sets up various services and middleware in an ASP.NET Core web application, including:

* Authentication and authorization using JSON Web Tokens (JWT) with the JwtBearer authentication scheme.
* Identity management using the Identity framework and Entity Framework for storage.
* Connection to a SQL Server database using Entity Framework.
* Controllers for handling HTTP requests.
* CORS (Cross-Origin Resource Sharing) configuration to enable cross-origin requests from a specific frontend URL.
* Swagger/OpenAPI documentation generation for the API endpoints.

using Microsoft.AspNetCore.Authentication.JwtBearer;

using Microsoft.AspNetCore.Identity;

using Microsoft.EntityFrameworkCore;

using Microsoft.IdentityModel.Tokens;

using System.Text;

using WebApplication1.DbContext;

using WebApplication1.Models;

var builder = WebApplication.CreateBuilder(args);

ConfigurationManager configuration = builder.Configuration;

This section imports the necessary namespaces and initializes the WebApplication builder with the provided command line arguments. It also retrieves the application's configuration settings.

builder.Services.AddDbContext<ApplicationDbContext>(options => options.UseSqlServer(configuration.GetConnectionString("ConnStr")));

This line adds a scoped ApplicationDbContext service to the dependency injection container using the AddDbContext method. It also configures the context to use a SQL Server database specified in the app's configuration settings.

builder.Services.AddIdentity<IdentityUser, IdentityRole>()

.AddEntityFrameworkStores<ApplicationDbContext>();

This section adds the Identity framework to the application's services, with support for the IdentityUser and IdentityRole types. It also configures the framework to use the ApplicationDbContext for storage. The commented-out line adds support for token generation, but it's not needed here since JWT tokens are used instead.

builder.Services.AddAuthentication(options =>

{

options.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme;

options.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;

options.DefaultScheme = JwtBearerDefaults.AuthenticationScheme;

})

.AddJwtBearer(options =>

{

options.SaveToken = true;

options.RequireHttpsMetadata = false;

options.TokenValidationParameters = new TokenValidationParameters()

{

ValidateIssuer = true,

ValidateAudience = true,

ValidateLifetime = true,

ValidateIssuerSigningKey = true,

ValidAudience = configuration["JWT:ValidAudience"],

ValidIssuer = configuration["JWT:ValidIssuer"],

IssuerSigningKey = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(configuration["JWT:Secret"]))

};

});

This section adds authentication to the application's services using the JWT Bearer authentication scheme. It configures the default authentication, challenge, and scheme to use JWT Bearer. It also sets up the JWT validation parameters, including the issuer, audience, and signing key, using values from the app's configuration settings.

builder.Services.AddControllers();

This line adds the default controller services to the dependency injection container.

builder.Services.AddEndpointsApiExplorer();

builder.Services.AddSwaggerGen();

These lines add the necessary services for generating Swagger/OpenAPI documentation for the application's API endpoints.

builder.Services.AddCors(Options =>

{

// Get the frontend URL from the configuration

var frondendUrl = configuration.GetValue<string>("frontend\_url");

Options.AddDefaultPolicy(builder =>

{

builder.WithOrigins(frondendUrl).AllowAnyOrigin().AllowAnyHeader(); // Set the allowed origin to the frontend URL

});

});

This section adds CORS configuration to the application's services to allow cross-origin requests from a specific frontend URL. It retrieves the URL from the app's configuration settings.

var app = builder.Build();

This line builds the application using the configured services