**Sakarya University Computer Engineering**

**Introduction to Cryptology Project**

****

**User Registration and Login with JWT Integration**

**Furkan Tataroğlu**

**G201210089**

**User Registration and Login with JWT Integration**

**Introduction**

This report provides an overview of the user registration and login functionalities implemented in a .NET-based application, with a focus on the integration of JWT (JSON Web Token) for secure authentication and authorization. Additionally, the utilization of Swagger for API testing is discussed, highlighting how JWT tokens are employed within this testing framework.

**User Registration and Login Processes**

The application offers two primary API endpoints to manage user interactions: register and login. These endpoints facilitate the creation of new user accounts and the authentication of existing users, respectively. Both processes utilize Microsoft Identity for user management and JWT for maintaining secure sessions.

A black and white striped background

Description automatically generated

**Registration Process**

**Endpoint:** POST /api/User/register

**How It Works:**

1. **Collecting User Information:** A user initiates the registration process by sending a UserAddDto object containing essential details such as email, username, and password to the register endpoint.
2. **Data Validation and Processing:** The UserController handles the incoming registration request through its Add method. This method leverages the IUserService to validate the provided information and securely store the new user in the database.
3. **Response:** Upon successful registration, the API responds with a Result object indicating success and includes relevant user information. If any issues arise during registration (e.g., email already in use), the API returns a BadRequest response with an appropriate error message.

A black and white rectangular object

Description automatically generated

**Login Process**

**Endpoint:** POST /api/User/login

**How It Works:**

1. **Submitting Credentials:** A user submits their email and password encapsulated in a LoginDto object to the login endpoint.
2. **Authentication:** The UserController's Login method processes the request by verifying the user's existence and validating the provided password using UserManager and SignInManager.
3. A black and white rectangular object with green and blue text

   Description automatically generated with medium confidence**JWT Token Generation:** Upon successful authentication, the TokenService generates a JWT token that includes the user's email, username, and roles as claims. This token is essential for maintaining secure sessions.

public string CreateToken(User user)

{

var claims = new List<Claim>

{

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

new Claim(JwtRegisteredClaimNames.GivenName, user.UserName)

};

var roles = \_userManager.GetRolesAsync(user).Result;

foreach (var role in roles)

{

claims.Add(new Claim(ClaimTypes.Role, role));

}

var creds = new SigningCredentials(\_key, SecurityAlgorithms.HmacSha512Signature);

var tokenDescriptor = new SecurityTokenDescriptor

Subject = new ClaimsIdentity(claims),

Expires = DateTime.Now.AddDays(7),

SigningCredentials = creds,

Issuer = \_config["JWT:Issuer"],

Audience = \_config["JWT:Audience"]

};

var tokenHandler = new JwtSecurityTokenHandler();

var token = tokenHandler.CreateToken(tokenDescriptor);

return tokenHandler.WriteToken(token);

**Code Explanation:**

* **Claims Setup:**
  + A list of claims is initialized with the user's email and username.
  + The user's roles are retrieved asynchronously using \_userManager.GetRolesAsync(user) and added to the claims list. This ensures that the token carries all relevant role information for authorization purposes.
* **Signing Credentials:**
  + SigningCredentials are created using a symmetric security key (\_key) and the HmacSha512Signature algorithm. This combination ensures that the token is securely signed, preventing unauthorized alterations.
* **Token Descriptor:**
  + The SecurityTokenDescriptor defines the properties of the JWT:
    - **Subject:** Encapsulates the claims identity.
    - **Expires:** Sets the token's expiration to 7 days from the creation time, enhancing security by limiting the token's validity period.
    - **SigningCredentials:** Associates the previously defined signing credentials with the token.
    - **Issuer & Audience:** Specifies the token issuer and intended audience, adding an extra layer of validation during token verification.
* **Token Generation:**
  + The JwtSecurityTokenHandler creates the token based on the descriptor and writes it as a string, which is then returned to the user. This JWT can now be used to authenticate and authorize subsequent requests to protected endpoints.

1. **Response:** The API responds with a NewUserDto object containing the user's email and the generated JWT token. If authentication fails (e.g., incorrect password), the API returns an Unauthorized response with an error message.

**Identity and Authentication Configuration**

To establish user management and implement JWT-based authentication, the following configurations are set up in the Program.cs file:

builder.Services.AddIdentity<User, IdentityRole<Guid>>(options =>

{

options.Password.RequireDigit = true;

options.User.AllowedUserNameCharacters = "";

options.User.RequireUniqueEmail = true;

}).AddEntityFrameworkStores<MyDbContext>();

builder.Services.AddAuthentication(options =>

{

options.DefaultAuthenticateScheme =

options.DefaultChallengeScheme =

options.DefaultForbidScheme =

options.DefaultScheme =

options.DefaultSignInScheme =

options.DefaultSignOutScheme = JwtBearerDefaults.AuthenticationScheme;

}).AddJwtBearer(options =>

{

options.TokenValidationParameters = new Microsoft.IdentityModel.Tokens.TokenValidationParameters

{

ValidateIssuer = true,

ValidIssuer = builder.Configuration["JWT:Issuer"],

ValidateAudience = true,

ValidAudience = builder.Configuration["JWT:Audience"],

ValidateIssuerSigningKey = true,

IssuerSigningKey = new SymmetricSecurityKey(

System.Text.Encoding.UTF8.GetBytes(builder.Configuration["JWT:SigningKey"]))

};

});

builder.Services.AddAuthorization(options =>

{

options.AddPolicy("AdminOnly", policy => policy.RequireRole("Admin"));

options.AddPolicy("UserOnly", policy => policy.RequireRole("User"));

});

**Explanation:**

* **AddIdentity Configuration:**
  + **User and Role Setup:** Configures Identity to use the User class for user entities and IdentityRole<Guid> for role management.
  + **Password Requirements:** Enforces that passwords must include at least one digit.
  + **Username and Email Rules:** Restricts allowed characters for usernames and ensures each email is unique.
  + **Entity Framework Stores:** Integrates Identity with Entity Framework by specifying MyDbContext as the data store.
* **AddAuthentication Configuration:**
  + **Default Schemes:** Sets JWT Bearer as the default scheme for authentication, challenges, forbids, sign-ins, and sign-outs.

**AddJwtBearer Configuration:**

* **TokenValidationParameters:** Defines the parameters for validating incoming JWT tokens.
  + **ValidateIssuer:** Ensures that the token's issuer matches the expected issuer defined in the configuration.
  + **ValidIssuer:** Specifies the valid issuer string, retrieved from the application's configuration settings (JWT:Issuer).
  + **ValidateAudience:** Ensures that the token's audience matches the expected audience.
  + **ValidAudience:** Specifies the valid audience string, also retrieved from the configuration (JWT:Audience).
  + **ValidateIssuerSigningKey:** Validates the signing key used to sign the token, ensuring its authenticity.
  + **IssuerSigningKey:** Provides the symmetric security key used to sign and validate the JWT. This key is retrieved from the configuration (JWT:SigningKey) and is essential for maintaining token integrity.

**AddAuthorization Configuration:**

* **Authorization Policies:** Defines policies that restrict access based on user roles.
  + **AdminOnly Policy:** Grants access only to users with the "Admin" role.
  + **UserOnly Policy:** Grants access only to users with the "User" role.

**Testing the API with Swagger**

Swagger is an invaluable tool for documenting and testing APIs. In this project, Swagger is integrated to simplify the testing of registration and login endpoints, as well as to manage JWT tokens during these tests.

**Swagger Configuration**

Swagger has been configured in the Program.cs file to include JWT authentication support.

//swagger

builder.Services.AddSwaggerGen(option =>

{

option.SwaggerDoc("v1", new OpenApiInfo { Title = "Demo API", Version = "v1" });

option.AddSecurityDefinition("Bearer", new OpenApiSecurityScheme

{

In = ParameterLocation.Header,

Description = "Please enter a valid token",

Name = "Authorization",

Type = SecuritySchemeType.Http,

BearerFormat = "JWT",

Scheme = "Bearer"

});

option.AddSecurityRequirement(new OpenApiSecurityRequirement

{

{

new OpenApiSecurityScheme

{

Reference = new OpenApiReference

{

Type=ReferenceType.SecurityScheme,

Id="Bearer"

}

},

new string[]{}

}

});

});

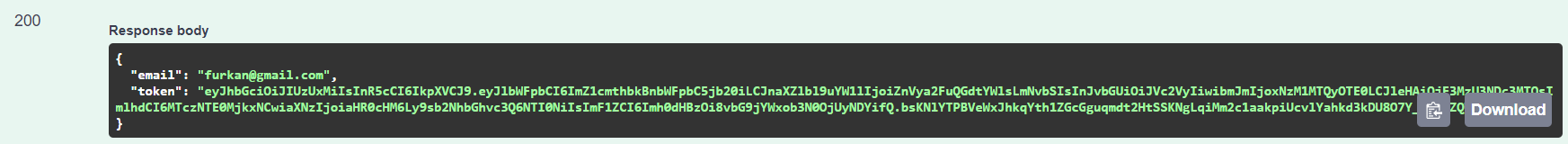
**Explanation:**

* **SwaggerDoc:** Defines the API documentation version and title.
* **SecurityDefinition:** Specifies the type of security (Bearer) and how the JWT token should be included in the request headers.
* **SecurityRequirement:** Ensures that all API endpoints require the specified security scheme (JWT Bearer token).

**Using JWT with Swagger**

To test protected endpoints using Swagger, follow these steps:

1. **Register a New User:**
   * Navigate to the POST /api/User/register endpoint in Swagger.
   * Provide the necessary user details (email, username, password) and execute the request.
   * A screenshot of a computer

     Description automatically generatedA successful response indicates that the user has been registered.
2. **Login with the Registered User:**
   * Go to the POST /api/User/login endpoint.
   * Enter the registered user's email and password.
   * Upon successful authentication, a JWT token will be received in the response.
3. **Authorize with the JWT Token:**
   * Click on the "Authorize" button in the Swagger UI.
   * In the dialog that appears, enter the token in the format: Bearer YOUR\_JWT\_TOKEN.
   * Click "Authorize" to apply the token to subsequent requests.

A screenshot of a computer

Description automatically generated

1. **Access Protected Endpoints:**
   * With the token authorized, access endpoints that require authentication, such as GET, PUT, or DELETE operations on user data.

A group of colored lines

Description automatically generated with medium confidence