Securing a REST API

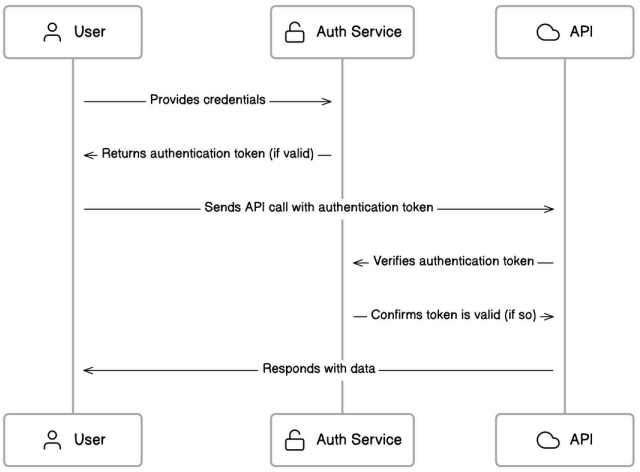
Securing a REST API involves implementing protective measures to safeguard the API's resources, endpoints, and data from unauthorized access, misuse, and malicious attacks. This encompasses employing authentication and authorization mechanisms, encrypting data transmission, validating input, managing tokens securely, monitoring and logging activities, and adhering to established security best practices. By fortifying the API against potential threats, organizations can ensure the confidentiality, integrity, and availability of their services while maintaining trust with users and stakeholders.

Ways to keep your REST API secured

In this project, two fundamental aspects of API security have been implemented: authentication with JSON Web Tokens (JWT) and authorization based on user roles.

Authentication with JWT Tokens - JSON Web Tokens (JWT) provide a stateless, secure way to authenticate users or clients accessing the API. JWT tokens are digitally signed, which ensures their integrity, and they can contain claims (such as user ID, roles, or other user-related data), which helps in establishing the user's identity and defining their permissions. By using JWT for authentication:

- Users can securely authenticate without the need to store session state on the server, making the API scalable and stateless.
- JWT tokens can include information about the user's roles or permissions, which simplifies subsequent authorization decisions.
- JWT tokens can be easily verified by the server, ensuring that only valid tokens from trusted sources are accepted.



Let's see an example and how to implement JWT Authentication.The code is written in C# and uses the .NET 6.0 framework. In a web API that uses JWT authentication we should implement:

- **UsersController class:** This class defines the endpoints for the API. It includes methods for:
  - Authenticating users (`/users/authenticate`) -used to authenticate users and return a JWT token.
  - Getting a list of all users (`/users`) - used to retrieve a list of all users in the application
- **UserService class:** This class contains the business logic for the API. It includes methods for:
  - Validating user credentials

    - Generating JWT tokens
    - Retrieving a list of users
- **JwtMiddleware class:** This class is responsible for authenticating users and generating JWT tokens. It does this by:
    - Validating the JWT token in the request header
    - Attaching the user object to the request if the token is valid
- **AppSettings class:** This class stores the configuration settings for the API, such as the secret key used to generate JWT tokens.

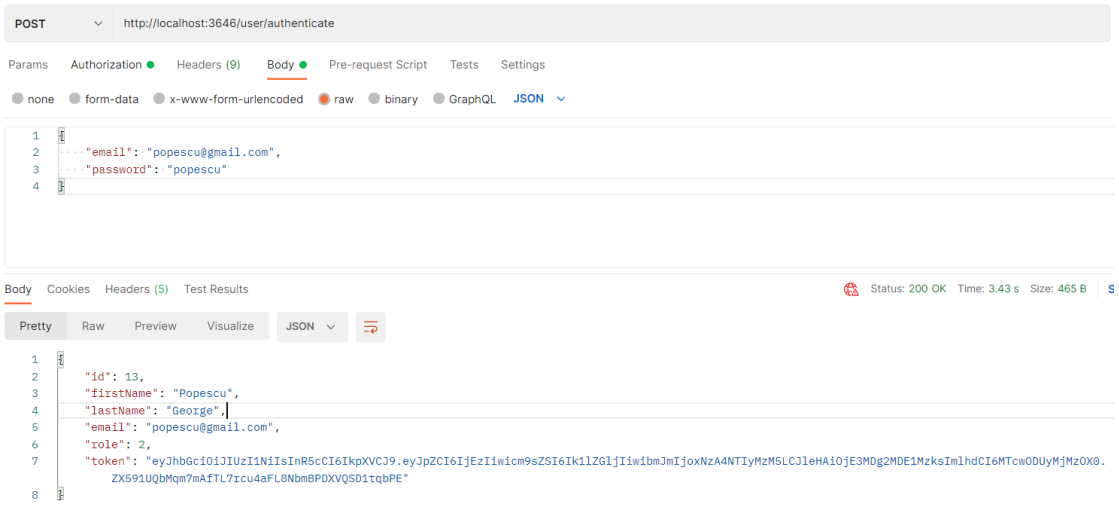
Here is an example of the code for the `UsersController` class:

```
[HttpPost("authenticate")]
0 references
public IActionResult Authenticate(AuthenticateRequest model)
{
    var response = _userService.Authenticate(model);

    if (response == null)
        return BadRequest(new { message = "Email or password is incorrect" });

    return Ok(response);
}

[Authorize(UserRole.Medic, UserRole.Patient,UserRole.SuperAdmin)]
[HttpGet]
0 references
public IActionResult GetAllUsers()
{
    var users = _userService.GetAllUsers();
    return Ok(users);
}
```

And UserService:

```
public string GenerateJwtToken(User user)
{
    var tokenHandler = new JwtSecurityTokenHandler();
    var key = Encoding.ASCII.GetBytes(_appSettings.Secret);
    var tokenDescriptor = new SecurityTokenDescriptor
    {
        Subject = new ClaimsIdentity(new[] { new Claim("id", user.Id.ToString()),
                new Claim(ClaimTypes.Role, user.Role.ToString()) }),
        Expires = DateTime.UtcNow.AddHours(22),
        SigningCredentials = new SigningCredentials(new SymmetricSecurityKey(key), SecurityAlgorithms.HmacSha256Signature)
    };
    var token = tokenHandler.CreateToken(tokenDescriptor);
    return tokenHandler.WriteToken(token);
}

2 references
public AuthenticateResponse Authenticate(AuthenticateRequest model)
{
    var user = _dbContext.Users.FirstOrDefault(user => user.Email == model.Email && user.Password == model.Password);

    if (user == null) return null;

    var token = GenerateJwtToken(user);

    return new AuthenticateResponse(user, token);
}
```

The API checks the username and password, and if they are correct, it generates a special access token. You received this token along with your user information. Keep this token safe, as it acts as your "key" to unlock other parts of the API in the future. Remember, whenever you want to do something in the API, include this token in your requests to show you're the authorised user.



Next, adding role-based authorization significantly enhances your API's security. Now, your API isn't just checking "who", but also "what they can do". Let's see how.

Authorization by Role - Role-based authorization allows you to control access to resources based on the roles assigned to users. By associating specific roles with different levels of access privileges, you can enforce the principle of least privilege, ensuring that users only have access to the resources and operations that are necessary for their role. With authorization by role:

- Access control becomes more manageable and scalable, as permissions are tied to predefined roles rather than individual users.
- Role-based authorization simplifies permission management and reduces the risk of misconfiguration or inconsistencies.
- It provides a flexible way to adapt access controls as the application evolves, by adding or modifying roles and their associated permissions.

```
89 references
public enum UserRole
{
    SuperAdmin = 0,
    ClinicAdmin = 1,
    Medic = 2,
    Patient = 3
}
```

```
[Authorize(UserRole.Medic, UserRole.ClinicAdmin, UserRole.SuperAdmin)]
[HttpGet("get_all_patients")]
0 references
```

The role enum defines all the available roles in the example api. I created it to avoid passing roles around as strings, so instead of 'Admin' we can use UserRole.Admin.

When a role is specified in the authorize attribute (e.g. [Authorize(Role.Admin)]) then the route is restricted to users in the specified role / roles.

```
public void OnAuthorization(AuthorizationFilterContext context)
{
    var user = (User)context.HttpContext.Items["User"];
    if (user == null)
    {
        context.Result = new JsonResult(new { message = "Unauthorized" })
        { StatusCode = StatusCodes.Status401Unauthorized };
    }
    else
    {
        if (!Roles.Contains(user.Role))
        {
            context.Result = new JsonResult(new { message = "Forbidden" })
            { StatusCode = StatusCodes.Status403Forbidden };
        }
    }
}
```

If a user object is found (indicating the user is authenticated), the code checks if the user's role is included in a predefined list called Roles. This list likely defines authorized roles for accessing the protected resource. If the user's role is not found in the Roles list, the code again sets the context.Result to a JsonResult object, similar to the unauthorized case. This time, the message is "Forbidden" and the status code is set to 403 (Forbidden). This indicates the user is authenticated but lacks the necessary permissions to access the resource. If the user's role is found in the Roles list, the code presumably allows the request to proceed without further intervention, as the user has passed both authentication and authorization checks.