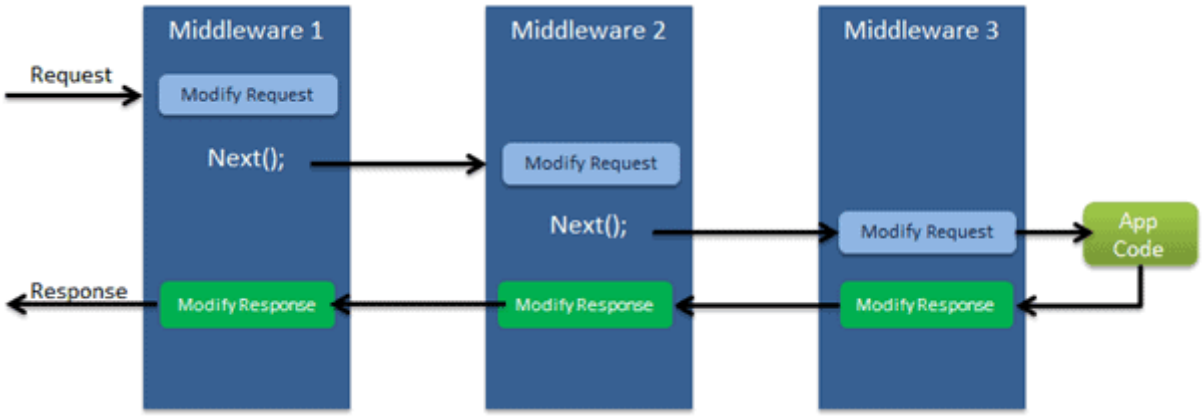
10. Custom middleware

Middleware in ASP.NET Core is software that's assembled into an application pipeline to handle requests and responses. Each component in the pipeline has the ability to process requests coming into the application and responses going out from the application. Middleware components can perform tasks such as authentication, logging, routing, and more.

10.1. How Middleware Works

1. **Request Handling**: Middleware components are executed in sequence as an HTTP request flows through the pipeline. Each component can perform actions on the request and decide whether to pass it to the next component.
2. **Response Handling**: After processing the request, middleware components handle the response in reverse order, potentially modifying the response before it is sent back to the client.

10.2. Creating custom middleware



To create custom middleware in ASP.NET Core, follow these steps:

1. **Create the Middleware Class**:
   - The middleware class must include a constructor that takes a `RequestDelegate` parameter.
   - It should also have an `Invoke` or `InvokeAsync` method that handles the request.
2. **Register the Middleware**:
   - Register the middleware in the `Configure` method of the `Startup` class using the `IApplicationBuilder.UseMiddleware<T>` method.

Example of custom logging middleware:

```
using Microsoft.AspNetCore.Http;
using System.Threading.Tasks;

public class CustomLoggingMiddleware
{
    private readonly RequestDelegate _next;

    public CustomLoggingMiddleware(RequestDelegate next)
    {
        _next = next;
    }

    public async Task InvokeAsync(HttpContext context)
    {
        // Code to execute before the next middleware
        Console.WriteLine($"Request: {context.Request.Method} {context.Request.Path}");

        await _next(context);

        // Code to execute after the next middleware
        Console.WriteLine($"Response: {context.Response.StatusCode}");
```

```
    }
}
```

Registering the middleware at startup:

```
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        // Add services to the container.
    }

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

        app.UseRouting();

        // Register custom middleware
        app.UseMiddleware<CustomLoggingMiddleware>();

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

For example we can use the middleware feature to implement a single place in which we catch all the exceptions - so that we are not polluting the endpoints logic with duplicated try-catches.

11. Authentication and authorization

In this chapter we will talk about various ways of implementing the authetnication and authorisation.
We will start with more simple ways and we will continue to more coplex one.
First let's differeiante between authoetnicatn and asuthrisation:

11.1. Authentication

Authentication is the process of verifying the identity of a user or entity. It ensures that the person or system is who they claim to be. Common methods of authentication include:

- **Passwords:** The user provides a secret password.
- **Biometrics:** The user provides a fingerprint, facial recognition, or other biological traits.
- **Two-Factor Authentication (2FA):** The user provides two types of evidence, such as a password and a code sent to their phone.

11.2. Authorization

Authorization is the process of determining what an authenticated user or entity is allowed to do. It involves granting or denying access to resources, actions, or services based on permissions or roles. For example:

- **Role-Based Access Control (RBAC):** Permissions are assigned to roles rather than individuals, and users are assigned roles.
- **Access Control Lists (ACLs):** Specific permissions are assigned to individual users or groups for various resources.

12. Basic authentication

We can leverage our knowledge of middleware to implement HTTP Basic Authentication. Basic authentication means that every request will contain the login and password of the user.

“Pasted image 20240607152336.png” could not be found.

Our middleware will check whether the credentials are valid. Only if they are correct, the request will be redirected to the next middleware. Usually, the data related to the login and password are passed in the Authorization HTTP header. They are usually encoded using Base64 encoding.

Example request:

```
GET /protected-resource HTTP/1.1
Host: example.com
Authorization: Basic dXNlcm5hbWU6cGFzc3dvcmQ=
```

In this example, `dXNlcm5hbWU6cGFzc3dvcmQ=` is the Base64 encoded form of `username:password`.

Why we use Base64 encoding?
HTTP headers can only contain text, not binary data. Base64 encoding converts binary data into a textual representation that can be safely included in HTTP headers.

Base64 encoding ensures that the encoded data contains only ASCII characters. This is important because it avoids issues related to character encoding or special characters that might not be handled properly by all systems.

Base64 encoding maintains the integrity of the data during transmission by representing it in a consistent and unambiguous format, reducing the risk of corruption or misinterpretation.

What is the difference between encoding and encryption?
Encdoing: Encoding is used to transform data into a different format using a scheme that is **publicly available**. The primary goal is to ensure that the data can be properly **consumed by different systems**.
Encryption: Encryption is used to **protect data by transforming it into an unreadable format using a specific algorithm and a key.** The primary goal is to secure data from unauthorized access.

Example of the middleware:

```
public class BasicAuthMiddleware
{
    private readonly RequestDelegate _next;
    private const string Realm = "My Realm";

    public BasicAuthMiddleware(RequestDelegate next)
    {        _next = next;
    }
    public async Task InvokeAsync(HttpContext context)
    {        if (context.Request.Headers.ContainsKey("Authorization"))
        {            var authHeader =
AuthenticationHeaderValue.Parse(context.Request.Headers["Authorization"]);

            if (authHeader.Scheme.Equals("basic", StringComparison.OrdinalIgnoreCase) &&
                authHeader.Parameter != null)
            {                var credentials =
Encoding.UTF8.GetString(Convert.FromBase64String(authHeader.Parameter)).Split(':');
                var username = credentials[0];
                var password = credentials[1];

                // Replace with your own user validation logic
                if (IsAuthorized(username, password))
                {                    await _next(context);
                    return;
                }            }        }
        // Return 401 Unauthorized if authentication fails
        context.Response.Headers["WWW-Authenticate"] = $"Basic realm=\"{Realm}\"";
        context.Response.StatusCode = StatusCodes.Status401Unauthorized;
    }
    private bool IsAuthorized(string username, string password)
    {        // Replace this with your own logic
```

```
        return username == "admin" && password == "password";
    }}
```

Then you can register the middleware:

```
app.UseMiddleware<BasicAuthMiddleware>();
```

13. Benefits and disadvantages of HTTP basic auth

Advantages of HTTP Basic Authentication

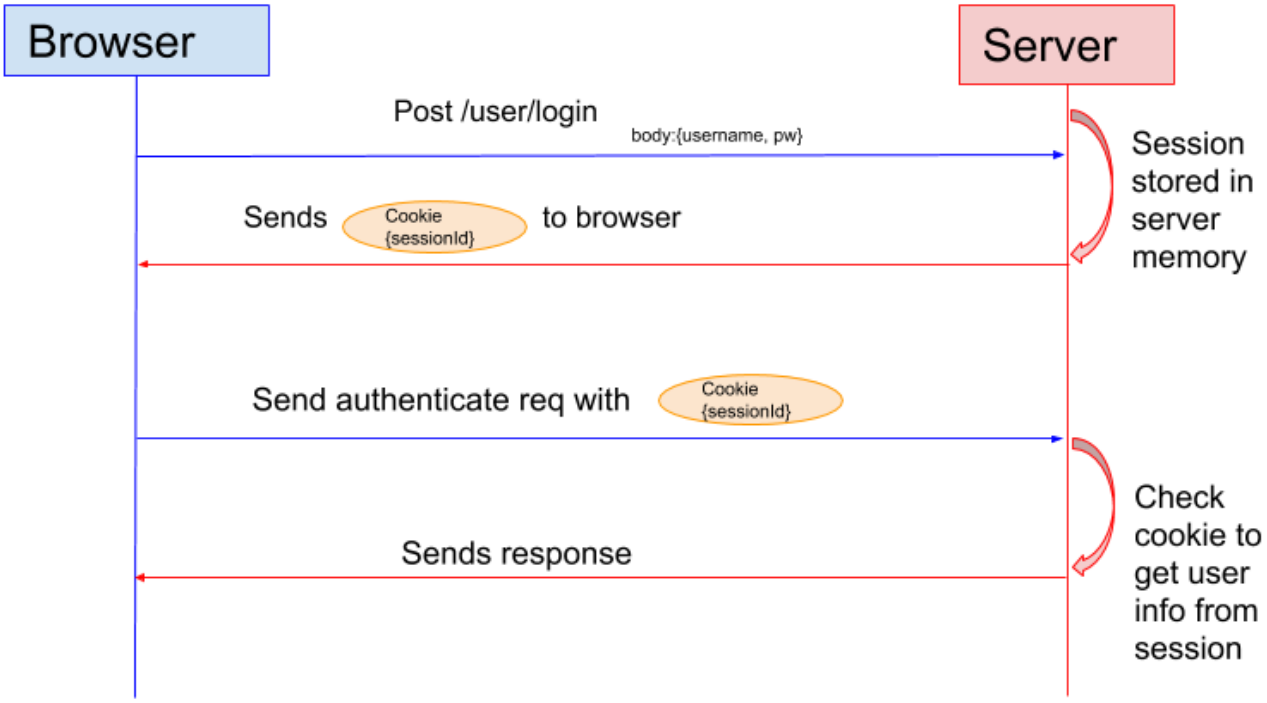
1. **Simplicity**:
   - Easy to implement and use, requiring minimal configuration.
2. **Wide Support**:
   - Supported by virtually all web servers and browsers, ensuring broad compatibility.
3. **Stateless**:
   - Since credentials are sent with each request, the server does not need to maintain session state, which simplifies server-side management.

Disadvantages of HTTP Basic Authentication

1. **Security**:
   - **Credentials are only Base64 encoded, not encrypted**, making them easily decodable if intercepted. Therefore, it must be used over HTTPS to protect the credentials in transit.
2. **No Expiration**:
   - Credentials do not expire with Basic Authentication, leading to potential security risks if not managed properly.
3. **User Experience**:
   - **Requires users to repeatedly send credentials** with each request, which can be less user-friendly and efficient compared to other authentication methods like token-based systems.
4. **Limited Functionality**:
   - Does not support advanced features such as multi-factor authentication or fine-grained access controls out of the box.

14. Using server session

In server session-based authentication, the server creates a session after a user logs in. The session is stored on the server, and a session ID is sent to the client. This session ID is then included in subsequent requests, allowing the server to identify the user and authorize their actions.



- Session IDs are random and difficult to guess, and they can be secured with HTTPS.
- The **server maintains the session state (usually in RAM memory)**, which can include user-specific data and permissions.
- Sessions can be configured to expire after a period of inactivity, enhancing security.

Disadvantages:**

- **Scalability**:
  - Managing sessions requires server resources and can become complex with a large number of users.
- **Statefulness**:
  - Requires maintaining state on the server, which can complicate load balancing and redundancy.
- **Session Hijacking**:
  - If an attacker obtains a session ID, they can impersonate the user unless additional measures are taken (e.g., regenerating session IDs, IP checks).

Sample communication:

```
//1. Login request

POST /login HTTP/1.1
Host: example.com
Content-Type: application/x-www-form-urlencoded

username=johndoe&password=securepassword

//2. Server response with session ID
HTTP/1.1 200 OK
Set-Cookie: sessionId=abc123; HttpOnly; Secure; Path=/

//3. Next request from client include the session ID

GET /protected-resource HTTP/1.1
Host: example.com
Cookie: sessionId=abc123

//4. Server response
HTTP/1.1 200 OK
Content-Type: application/json
```

```
{
  "data": "protected information"
}
```

15. Using JSON web token

In this chapter, we will talk about implementing JSON Web Token (JWT) Authentication. Before we do that, we will discuss a few important topics related to security.

15.1. Hashing function

A hashing function is a mathematical algorithm that transforms an input (or 'message') into a fixed-size string of characters, which is typically a sequence of numbers and letters. The output, known as the hash value or digest, is unique to each unique input.

15.1.1. Key characteristics of hashing functions

1. **Deterministic**:
   - The same input will always produce the same hash value.
2. **Fixed Output Size**:
   - Regardless of the input size, the output hash is of a fixed length. For example, SHA-256 always produces a 256-bit hash.
3. **Efficiency**:
   - Hash functions are designed to be fast (for password hashing we might use slower functions) and efficient, quickly processing large amounts of data.
4. **Pre-image Resistance**:
   - Given a hash value, it should be computationally infeasible to find the original input.
5. **Small Changes in Input**:
   - Any small change in the input should produce a significantly different hash, a property known as the **avalanche effect**.
6. **Collision Resistance**:
   - It should be extremely difficult to find two different inputs that produce the same hash value.

15.1.2. Common uses for hashing function

- **Data Integrity**:
  - Verifying that data has not been altered by comparing hash values before and after transmission or storage.
- **Password Storage**:
  - Storing hashes of passwords instead of the actual passwords for security purposes.
- **Digital Signatures**:
  - Ensuring the authenticity and integrity of a message or document.
- **Cryptographic Applications**:
  - Used in various cryptographic algorithms and protocols to ensure data security.

15.1.3. Examples of hashing functions

- **MD5 (Message Digest Algorithm 5)**: Produces a 128-bit hash value, though considered cryptographically broken and unsuitable for further use.
- **SHA-1 (Secure Hash Algorithm 1)**: Produces a 160-bit hash value, also considered broken and insecure.
- **SHA-256 (Secure Hash Algorithm 256-bit)**: Part of the SHA-2 family, widely used and considered secure.

15.2. Using hashing function for storing password in a secure way

15.2.1. Why Store Passwords Securely?

Storing passwords securely is crucial to protect user accounts from unauthorized access and prevent data breaches. Plain text passwords are vulnerable to theft and misuse, so secure storage methods are essential.

15.2.2. How to Store Passwords Securely

1. **Hashing Passwords**:

- Instead of storing plain text passwords, store hashed versions of passwords. A hashing function converts the password into a fixed-length string that cannot be easily reversed.

2. **Adding Salt**:
   - A salt is a random value added to the password before hashing. This ensures that even if two users have the same password, their hashes will be different, preventing attackers from using precomputed tables (rainbow tables) to crack passwords.
3. **Using PBKDF2 (Password-Based Key Derivation Function 2)**:
   - PBKDF2 is a key derivation function that applies a hash function multiple times to the password and salt. This process, known as key stretching, makes brute-force attacks significantly more difficult by increasing the computational effort required to hash each password.

15.2.3. Steps for Secure Password Storage

1. **Generate a Salt**:
   - Create a unique, random salt for each password.
2. **Hash the Password with Salt**:
   - Use a hashing algorithm combined with the salt. PBKDF2 is a commonly used function that applies a hash function (such as SHA-256) iteratively to the password and salt.
3. **Store the Hash and Salt**:
   - Store the resulting hash and the salt together. The salt does not need to be secret, but it must be unique for each user.

Example in CS:**

```
public static Tuple<string, string> GetHashedPasswordAndSalt(string password)
{
    byte[] salt = new byte[128 / 8];
    using (var rng = RandomNumberGenerator.Create())
    {        rng.GetBytes(salt);
    }
    string hashed = Convert.ToBase64String(KeyDerivation.Pbkdf2(
        password: password,
        salt: salt,
        prf: KeyDerivationPrf.HMACSHA1,
        iterationCount: 10000,
        numBytesRequested: 256 / 8));

    string saltBase64 = Convert.ToBase64String(salt);

    return new(hashed, saltBase64);
}

public static string GetHashedPasswordWithSalt(string password, string salt)
{
    byte[] saltBytes = Convert.FromBase64String(salt);

    string currentHashedPassword = Convert.ToBase64String(KeyDerivation.Pbkdf2(
        password: password,
        salt: saltBytes,
        prf: KeyDerivationPrf.HMACSHA1,
        iterationCount: 10000,
        numBytesRequested: 256 / 8));

    return currentHashedPassword;
}
```

15.2. JSON web token

15.2.1. What is a JWT?

A **JSON Web Token (JWT)** is an open standard (RFC 7519) for securely transmitting information between parties as a JSON object. This information can be verified and trusted because it is digitally signed.

15.2.2. Structure of a JWT

A JWT is composed of three parts:
Header:

- Contains metadata about the token, such as the type of token (JWT) and the signing algorithm used (e.g., HMAC SHA256).

Example:

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

Payload:

- Contains the claims. Claims are statements about an entity (typically, the user) and additional data. There are three types of claims: registered, public, and private.
- **Example:**

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "iat": 1516239022
}
```

Signature:

- Used to verify the token was not altered. The signature is created by taking the encoded header, the encoded payload, a secret, and the algorithm specified in the header, and then hashing them together.
- **Example (with HMAC SHA256):**

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  secret
)
```

HMACSHA256 stands for **Hash-based Message Authentication Code (HMAC) with Secure Hash Algorithm 256-bit (SHA-256)**. It is a specific type of HMAC that uses the SHA-256 hash function to ensure data integrity and authenticity.

HMAC is a mechanism that combines a cryptographic hash function with a secret key to produce a message authentication code (MAC). This ensures that the message has not been tampered with and verifies the sender's authenticity.

A JWT looks like this:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MD
IyfQ.SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c
```

15.2.3. Problems JWT solves

- **Stateless Authentication**:
  - Traditional session-based authentication requires the server to store session information, leading to scalability issues. JWT allows the server to authenticate users without storing session data on the server, as the token itself contains all the necessary information.
- **Cross-Domain Authentication**:
  - JWTs can be used across different domains, enabling Single Sign-On (SSO) scenarios.
- **Interoperability**:
  - As an open standard, JWT is widely supported across various platforms and technologies, making it a versatile solution for secure information exchange.

15.2.4. Why we use JWT?

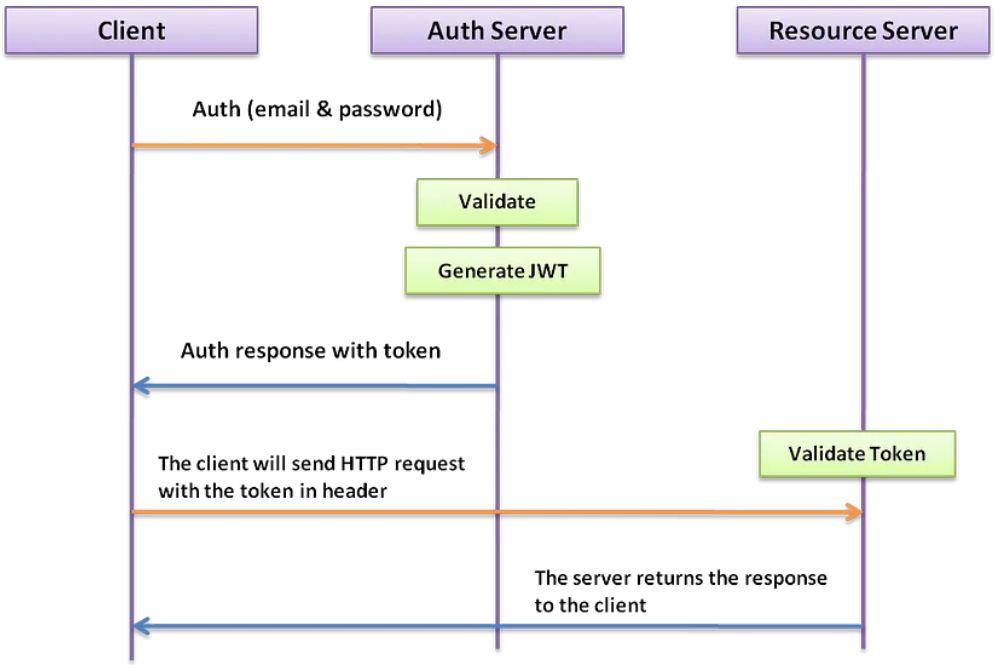
- **Security**:
  - JWTs are signed, ensuring the integrity and authenticity of the information they contain. Optionally, they can also be encrypted for confidentiality.
- **Scalability**:
  - By offloading session storage from the server, JWTs support stateless authentication, which is ideal for distributed systems and microservices architectures.
- **Performance**:
  - JWTs are self-contained and do not require database lookups for each request, which can improve performance, especially in large-scale applications.
- **Ease of Use**:
  - JWTs are easy to generate, distribute, and validate. They are transmitted as compact URL-safe strings, making them suitable for use in HTTP headers.

15.3. Using JWT for authentication/authorization

Ensuring secure and seamless user authentication and authorization is crucial JSON Web Tokens (JWT) and refresh tokens are commonly used to achieve this.

JWTs are short-lived tokens that carry user claims and are used to access protected resources once a user logs in and their credentials are verified the server issues a JWT along with a refresh token the JWT is then used by the client to authenticate subsequent requests while the refresh token is stored securely and used to obtain new JWTs when the current one expires this method provides a balance between security and user experience short-lived.

JWTs minimize the risk of token theft and refresh tokens eliminate the need for frequent re-authentication enabling continuous and secure access to resources without compromising performance or security.



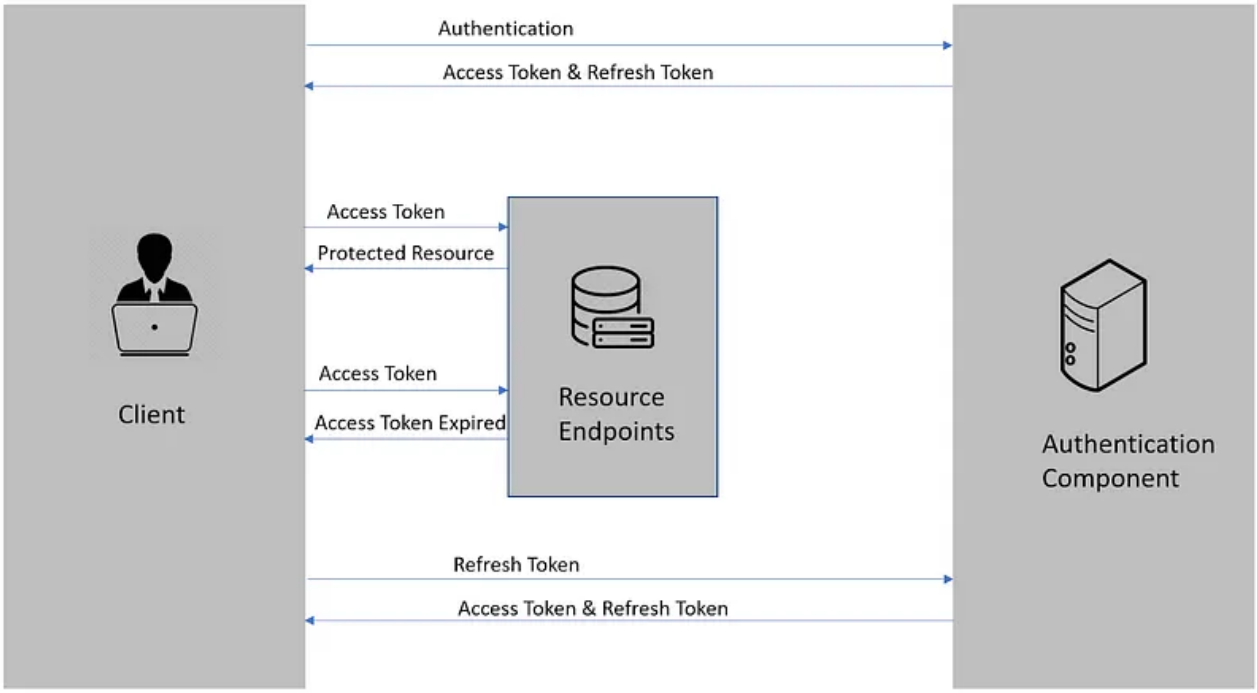
Client Authentication:

1. The client sends a login request with credentials.
2. The server validates the credentials and generates a JWT. Server generates the token using secret password known only to the server. Server generates also the refresh token (this is not a JSON web token). Refresh token is saved in the database along with the user's data (including the generated salt).
3. The server returns the JWT to the client along with the refresh token.

Client Authorization:

1. The client includes the JWT in the Authorization header of subsequent requests.
2. The server validates the JWT and processes the request if the token is valid.

Example of communication including the refresh token:



Good practices*

- When using JWT tokens, ensure they are transmitted over HTTPS to prevent interception.
- Keep the tokens short-lived to minimize the impact of potential theft.
- Store JWT tokens securely, avoiding local storage or session storage for sensitive applications; prefer HTTP-only cookies.
- Implement proper token expiration and revocation mechanisms to manage token lifecycles effectively.
- Regularly update and rotate signing keys to maintain security, and include only necessary information in the token payload to minimize exposure of sensitive data.

15. 4. Example of configuring the ASP.NET

1. First we will create endpoint which allows us to register new students

```
[AllowAnonymous]
[HttpPost("register")]
public IActionResult RegisterStudent(RegisterRequest model)
{
    var hashedPasswordAndSalt = SecurityHelpers.GetHashedPasswordAndSalt(model.Password);

    var user = new AppUser()
    {
        Email = model.Email,
        Login = model.Login,
        Password = hashedPasswordAndSalt.Item1,
        Salt = hashedPasswordAndSalt.Item2,
        RefreshToken = SecurityHelpers.GenerateRefreshToken(),
        RefreshTokenExp = DateTime.Now.AddDays(1)
    };
    _context.Users.Add(user);
    _context.SaveChanges();

    return Ok();
}
```

2. Then we will configure ASP.NET to validate the endpoints using the JWT token.

```
builder.Services.AddAuthentication(options =>
{
    options.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme;
```

```
    options.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;
}).AddJwtBearer(opt =>
{
    opt.TokenValidationParameters = new TokenValidationParameters
    {
        ValidateIssuer = true,   //by who
        ValidateAudience = true, //for whom
        ValidateLifetime = true,
        ClockSkew = TimeSpan.FromMinutes(2),
        ValidIssuer = "https://localhost:5001", //should come from configuration
        ValidAudience = "https://localhost:5001", //should come from configuration
        IssuerSigningKey = new
SymmetricSecurityKey(Encoding.UTF8.GetBytes(builder.Configuration["SecretKey"]))
    };
    opt.Events = new JwtBearerEvents
    {
        OnAuthenticationFailed = context =>
        {
            if (context.Exception.GetType() == typeof(SecurityTokenExpiredException))
            {
                    context.Response.Headers.Add("Token-expired", "true");
            }

            return Task.CompletedTask;
        }
    };
    }).AddJwtBearer("IgnoreTokenExpirationScheme",opt =>
{
    opt.TokenValidationParameters = new TokenValidationParameters
    {
        ValidateIssuer = true,   //by who
        ValidateAudience = true, //for whom
        ValidateLifetime = false,
        ClockSkew = TimeSpan.FromMinutes(2),
        ValidIssuer = "https://localhost:5001", //should come from configuration
        ValidAudience = "https://localhost:5001", //should come from configuration
        IssuerSigningKey = new
SymmetricSecurityKey(Encoding.UTF8.GetBytes(builder.Configuration["SecretKey"]))
    };
    });
```

We have setup the default policy for using the JWT access token. We have crated an additional policy called "IgnoreTokenExpiration" which is used for refresh token related endpoint. For this endpoint we do not want to check the expiration of the access token (but we still check the integrity of the token).

3. We now can protect specific endpoint using the [Attribute]. Remember that you have access to the User object which allows you to fetch the data about the user (from the access token's payload).

```
[Authorize]
[HttpGet]
public IActionResult GetStudents()
{
        var userId = User.Claims.FirstOrDefault(c => c.Type == "sub")?.Value;
        var roles = User.Claims.Where(c => c.Type == "role").Select(c => c.Value).ToList();

    return Ok("Secret data");
}
```

4. Now we will add the endpoint which is used to perform the login operation. As a result this endpoint should return access token and refresh token.

```
[AllowAnonymous]
[HttpPost("login")]
public IActionResult Login(LoginRequest loginRequest)
{
    AppUser user = _context.Users.Where(u => u.Login == loginRequest.Login).FirstOrDefault();
```

```
    string passwordHashFromDb = user.Password;
    string curHashedPassword = SecurityHelpers.GetHashedPasswordWithSalt(loginRequest.Password, user.Salt);

    if (passwordHashFromDb != curHashedPassword)
    {        return Unauthorized();
    }

    Claim[] userclaim = new[]
    {
        new Claim(ClaimTypes.Name, "pgago"),
        new Claim(ClaimTypes.Role, "user"),
        new Claim(ClaimTypes.Role, "admin")
        //Add additional data here
    };

    SymmetricSecurityKey key = new
SymmetricSecurityKey(Encoding.UTF8.GetBytes(_configuration["SecretKey"]));

    SigningCredentials creds = new SigningCredentials(key, SecurityAlgorithms.HmacSha256);

    JwtSecurityToken token = new JwtSecurityToken(
        issuer: "https://localhost:5001",
        audience: "https://localhost:5001",
        claims: userclaim,
        expires: DateTime.Now.AddMinutes(10),
        signingCredentials: creds
    );
    user.RefreshToken = SecurityHelpers.GenerateRefreshToken();
    user.RefreshTokenExp = DateTime.Now.AddDays(1);
    _context.SaveChanges();

    return Ok(new
    {
        accessToken = new JwtSecurityTokenHandler().WriteToken(token),
        refreshToken = user.RefreshToken
    });
```

5. And in the last step we will add the endpoint used with the refresh token.

```
[Authorize(AuthenticationSchemes = "IgnoreTokenExpirationScheme")]
[HttpPost("refresh")]
public IActionResult Refresh(RefreshTokenRequest refreshToken)
{
    AppUser user = _context.Users.Where(u => u.RefreshToken == refreshToken.RefreshToken).FirstOrDefault();

    if (user == null)
    {
            throw new SecurityTokenException("Invalid refresh token");
    }

    if (user.RefreshTokenExp < DateTime.Now)
    {
            throw new SecurityTokenException("Refresh token expired");
    }

    Claim[] userclaim = new[]
    {
        new Claim(ClaimTypes.Name, "pgago"),
        new Claim(ClaimTypes.Role, "user"),
        new Claim(ClaimTypes.Role, "admin")
        //Add additional data here
    };

    SymmetricSecurityKey key = new
SymmetricSecurityKey(Encoding.UTF8.GetBytes(_configuration["SecretKey"]));

    SigningCredentials creds = new SigningCredentials(key, SecurityAlgorithms.HmacSha256);

    JwtSecurityToken jwtToken = new JwtSecurityToken(
        issuer: "https://localhost:5001",
```

```
        audience: "https://localhost:5001",
        claims: userclaim,
        expires: DateTime.Now.AddMinutes(10),
        signingCredentials: creds
    );
    user.RefreshToken = SecurityHelpers.GenerateRefreshToken();
    user.RefreshTokenExp = DateTime.Now.AddDays(1);
    _context.SaveChanges();

    return Ok(new
    {
        accessToken = new JwtSecurityTokenHandler().WriteToken(jwtToken),
        refreshToken = user.RefreshToken
    });
}
```