Independent Skill Development Mission

Independent Skill Development Mission NXT
EMPOWERING FUTURES, SHAPING TOMORROW

ISDM (INDEPENDENT SKILL DEVELOPMENT MISSION)

SECURITY IN ASP.NET CORE

AUTHENTICATION (JWT, OAUTH) AND AUTHORIZATION

Authentication and **authorization** are two crucial aspects of application security, especially when dealing with user access in web applications and APIs. In ASP.Net Core, authentication is the process of verifying the identity of users, while authorization defines what actions authenticated users are allowed to perform. These two concepts work together to secure an application.

Authentication in ASP.Net Core
Authentication verifies the identity of a user, ensuring that the user is who they claim to be. ASP.Net Core offers multiple authentication mechanisms, including **JWT (JSON Web Tokens)** and **OAuth 2.0**. These methods are commonly used to authenticate users for APIs and web applications.

JWT Authentication
JWT is a compact and self-contained way to transmit information securely between parties as a JSON object. In the context of ASP.Net Core, JWT is commonly used for **stateless authentication** in which the server does not store the session. The user is authenticated via a token that contains encrypted information, such as user identity and claims, and is passed along with each request.

Here's how JWT works in ASP.Net Core:

1. The user submits their credentials (username and password) to the server.
2. If the credentials are valid, the server generates a JWT token, signs it with a secret key, and sends it back to the user.
3. The client stores the token (typically in local storage or session storage) and includes it in the Authorization header of subsequent requests.
4. The server validates the token, ensuring its authenticity, and grants access to protected resources.

Example of JWT Authentication in ASP.Net Core:

```
public class JwtAuthenticationService {

  private readonly string _secretKey = "my_secret_key_12345"; // This should be
stored securely


  public string GenerateJwtToken(string username) {
    var claims = new[] {
      new Claim(JwtRegisteredClaimNames.Sub, username),
      new Claim(JwtRegisteredClaimNames.Jti, Guid.NewGuid().ToString())
    };

    var key = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(_secretKey));
    var creds = new SigningCredentials(key, SecurityAlgorithms.HmacSha256);

    var token = new JwtSecurityToken(
      issuer: "http://myapp.com",
      audience: "http://myapp.com",
      claims: claims,
      expires: DateTime.Now.AddHours(1),
      signingCredentials: creds
    );

    return new JwtSecurityTokenHandler().WriteToken(token);
  }
}
```

In this example, the GenerateJwtToken method creates and returns a JWT token that contains user claims and an expiration time.

OAuth 2.0 Authentication
OAuth 2.0 is another popular authentication mechanism used in modern web applications. Unlike JWT, OAuth is a delegation protocol that allows users to grant third-party applications limited access to their resources without sharing their credentials. OAuth is widely used in scenarios where users authenticate using their social media accounts or other external services.

OAuth works through an authorization server that issues access tokens. These tokens are then used to access resources on behalf of the user. OAuth 2.0 flows include **Authorization Code Flow**, **Implicit Flow**, **Client Credentials Flow**, and **Resource Owner Password Credentials Flow**, with the Authorization Code Flow being the most common for web applications.

Example of OAuth Flow in ASP.Net Core: ASP.Net Core supports OAuth 2.0 via **IdentityServer** or middleware like **OpenIdConnect**. Here's a basic example using **Google OAuth**:

```
public void ConfigureServices(IServiceCollection services) {
  services.AddAuthentication(options => {
    options.DefaultScheme = CookieAuthenticationDefaults.AuthenticationScheme;
    options.DefaultChallengeScheme = GoogleDefaults.AuthenticationScheme;
  })
  .AddGoogle(options => {
    options.ClientId = Configuration["Google:ClientId"];
    options.ClientSecret = Configuration["Google:ClientSecret"];
  });
}
```

In this setup, users can authenticate via their Google account, and the application uses Google OAuth for secure access to user data.

Authorization in ASP.Net Core
Once a user is authenticated, **authorization** comes into play. Authorization is the

process of determining whether a user has permission to access a specific resource or perform a specific action.

ASP.Net Core uses **Roles** and **Claims-based** authorization. Roles are used to group users based on their permissions, while claims are key-value pairs that carry additional information about the user (e.g., user ID, email address, permissions).

You can implement role-based or claims-based authorization as follows:

```
[Authorize(Roles = "Admin")]

public IActionResult AdminDashboard() {

   return View();

}
```

In this example, the AdminDashboard action is protected, and only users with the Admin role are authorized to access it.

Securing APIs with SSL/TLS

Securing APIs is a critical aspect of web development, and **SSL/TLS** (Secure Sockets Layer / Transport Layer Security) is one of the most effective methods to ensure the security of data transmitted between the client and the server. SSL/TLS is a cryptographic protocol that provides secure communication over a network.

What is SSL/TLS?
SSL/TLS encrypts the data between the client (typically a web browser or mobile app) and the server, ensuring that no one can intercept or tamper with the data. SSL/TLS uses asymmetric cryptography to establish a secure connection, with the server providing a public certificate for encryption, and the client using the certificate to decrypt data.

SSL/TLS protects:

- **Confidentiality**: Ensures that sensitive data (e.g., passwords, personal information) is encrypted and protected.
- **Integrity**: Ensures that data is not altered during transmission.
- **Authentication**: Ensures that the server is who it claims to be.

Implementing SSL/TLS in ASP.Net Core
ASP.Net Core makes it easy to implement SSL/TLS by using the built-in support for

HTTPS. By default, ASP.Net Core applications can be configured to use HTTPS, ensuring that all communication between the client and server is encrypted.

Steps to Enable SSL in ASP.Net Core:

1. **Obtain an SSL Certificate**: You can obtain an SSL certificate from a trusted certificate authority (CA) or generate a self-signed certificate for development purposes.
2. **Configure the application to use HTTPS**:
    - In Program.cs (or Startup.cs in earlier versions of .NET Core), add the following line to enforce HTTPS redirection:
3. builder.Services.AddHttpsRedirection(options => {
4. options.HttpsPort = 5001;
5. });
    - Ensure that your web server is configured to listen on the HTTPS port (usually 443 for production and 5001 for development).
6. **Redirect HTTP to HTTPS**: It's essential to enforce secure connections by redirecting all HTTP traffic to HTTPS:
7. app.UseHttpsRedirection();

Securing APIs with SSL/TLS

To secure APIs, you need to ensure that API requests and responses are transmitted over HTTPS. All sensitive data, including authentication tokens (like JWTs or OAuth tokens), should be sent via HTTPS to protect them from man-in-the-middle attacks.

Example of securing an API with HTTPS:

```
[HttpGet]

[Route("api/secure-data")]

[Authorize]

public IActionResult GetSecureData() {

  return Ok("This is a secure response.");

}
```

In this example, the API endpoint is secured with both HTTPS and authentication. The Authorize attribute ensures that only authenticated users can access the endpoint, and HTTPS ensures the communication is encrypted.

SSL/TLS Best Practices:

- **Use Strong Encryption**: Ensure that your SSL/TLS configuration uses strong encryption algorithms (e.g., TLS 1.2 or higher).
- **Use Valid Certificates**: Always use a valid SSL certificate issued by a trusted certificate authority (CA) in production.
- **Disable Weak Ciphers**: Avoid using outdated or weak ciphers like SSLv3 and RC4 in your configuration.
- **Force HTTPS**: Always redirect users to HTTPS to ensure secure communication.
- **Regularly Update SSL/TLS Configurations**: Keep your SSL/TLS configurations updated to address vulnerabilities in old versions and weak ciphers.

Exercise

1. **Implement JWT Authentication**: Set up JWT authentication in an ASP.Net Core API. Implement the process of generating and validating JWT tokens, and secure API endpoints using the [Authorize] attribute.
2. **Enable HTTPS in ASP.Net Core**: Configure your ASP.Net Core application to use HTTPS. Ensure that all API calls are made over HTTPS and test with a self-signed certificate for local development.
3. **OAuth Integration**: Implement OAuth 2.0 in your ASP.Net Core application to authenticate users via a third-party service like Google or Facebook. Secure API endpoints using OAuth tokens.

Case Study: Building a Secure API for a Mobile Application

In this case study, you will create a secure **Mobile API** using ASP.Net Core. The API will handle authentication via JWT and OAuth, and all sensitive data will be transmitted securely using SSL/TLS. The mobile application will authenticate users and retrieve data from the API using tokens.

- **Authentication**: Implement JWT authentication for user login and OAuth for social login.
- **Authorization**: Implement role-based authorization for different types of users (e.g., admin and regular users).
- **API Security**: Secure the API endpoints using HTTPS, ensuring that all data exchanges between the mobile app and the server are encrypted.

MIDDLEWARE IN ASP.NET CORE

CREATING CUSTOM MIDDLEWARE

Middleware in ASP.Net Core is a key component in the request-response processing pipeline. It serves as a chain of handlers that are executed for every HTTP request made to the application. Middleware components have access to the request, can modify it, pass it along the pipeline, and can even short-circuit the pipeline by directly responding to the client.

ASP.Net Core allows developers to create **custom middleware** to fulfill specific application requirements. Custom middleware can perform tasks such as logging, error handling, authentication, or even modifying the HTTP response before it is sent to the client.

What is Middleware?

In ASP.Net Core, middleware is a software component that sits between the server and the application, handling requests and responses. Middleware can perform a wide variety of tasks, such as checking for valid authentication tokens, logging request details, or modifying the request or response. The middleware components are executed in the order they are registered in the pipeline.

Steps to Create Custom Middleware

1. **Define the Middleware Class**: The first step in creating custom middleware is to define a class that handles HTTP requests. The class should include a Invoke or InvokeAsync method, which will be executed for every request that passes through this middleware.

Here's an example of creating custom middleware to log the details of every incoming HTTP request:

```
public class RequestLoggingMiddleware

{

   private readonly RequestDelegate _next;



   public RequestLoggingMiddleware(RequestDelegate next)
```

```
    {
        _next = next;
    }

    public async Task InvokeAsync(HttpContext context)
    {
        Console.WriteLine($"Request Path: {context.Request.Path}");
        Console.WriteLine($"Request Method: {context.Request.Method}");

        await _next(context); // Pass control to the next middleware in the pipeline
    }
}
```

2. **Register the Middleware in the Pipeline**: After defining the middleware, it must be added to the request processing pipeline in the Configure method of the Startup.cs class. Middleware components are added in the order they should be executed.

```
3. public class Startup
4. {
5.     public void Configure(IApplicationBuilder app)
6.     {
7.         app.UseMiddleware<RequestLoggingMiddleware>(); // Register custom middleware
8.
9.         app.Run(async (context) =>
10.        {
11.            await context.Response.WriteAsync("Hello from ASP.Net Core!");
```

```
12.      });
13.  }
14. }
```

In this example, the custom middleware is registered using the UseMiddleware method. The RequestLoggingMiddleware class logs the request details, then calls the _next delegate to pass control to the next middleware in the pipeline.

Handling Responses in Custom Middleware

Custom middleware can also modify the response before it is sent to the client. For example, you can add custom headers or modify the response body:

```
public class ResponseModifyingMiddleware
{
  private readonly RequestDelegate _next;

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

  public async Task InvokeAsync(HttpContext context)
  {
    // Modify the response before calling the next middleware
    context.Response.OnStarting(() =>
    {
      context.Response.Headers.Add("X-Custom-Header", "CustomMiddleware");
      return Task.CompletedTask;
```

```
    });


    await _next(context); // Pass control to the next middleware in the pipeline
  }
}
```

In this example, the middleware adds a custom header to the response before it is sent to the client.

REQUEST PROCESSING PIPELINE

The **request processing pipeline** in ASP.Net Core is a series of middleware components through which an HTTP request passes. When a request is made, it is received by the web server, and each middleware component in the pipeline can either process the request, modify it, or terminate the request by sending a response directly to the client. The order of middleware registration determines the order in which they are executed.

The request processing pipeline is highly customizable, allowing developers to define the sequence of middleware based on the specific needs of the application. The sequence of middleware is crucial because the order in which middleware is registered impacts how requests are processed and responses are returned.

Structure of the Pipeline

When a request arrives at an ASP.Net Core application, it follows these basic steps in the pipeline:

1. **Request Handling**: The request is passed through the middleware components where each middleware can inspect, modify, or short-circuit the request.

2. **Execution of Middleware**: As the request moves through the pipeline, it encounters middleware components that perform specific tasks, such as authentication, logging, routing, or custom logic.

3. **Request Termination**: If a middleware component decides to terminate the request early, it can return a response to the client immediately, skipping the remaining middleware in the pipeline.

4. **Response Handling**: After the request has been processed, the response is passed back through the pipeline, where additional middleware can modify the response before sending it back to the client.

ASP.Net Core uses the concept of a **middleware pipeline**, which is configured in the Configure method of the Startup.cs file. Middleware components are registered in the order they should process requests.

Example: Request Processing Pipeline Configuration

```
public void Configure(IApplicationBuilder app)

{

  // Logging middleware

  app.UseMiddleware<RequestLoggingMiddleware>();


  // Authentication middleware

  app.UseAuthentication();


  // Authorization middleware

  app.UseAuthorization();


  // Routing middleware

  app.UseRouting();


  // Custom response modification middleware

  app.UseMiddleware<ResponseModifyingMiddleware>();


  // Final response handler

  app.Run(async (context) =>
```

```
    {
        await context.Response.WriteAsync("Request processed successfully!");
    });
}
```

In this example, the request goes through multiple stages:

1. **RequestLoggingMiddleware** logs the request details.
2. **AuthenticationMiddleware** handles user authentication.
3. **AuthorizationMiddleware** checks if the user is authorized to access the resource.
4. **UseRouting** sets up the routing mechanism to direct the request to the appropriate controller or endpoint.
5. **ResponseModifyingMiddleware** modifies the response before sending it to the client.
6. **Final handler** sends a response back to the client.

Understanding Middleware Order

The order in which middleware is configured in the pipeline is important. For example, UseRouting must be placed before UseEndpoints, as routing must occur before endpoint selection. Similarly, authentication must occur before authorization checks.

Exercises and Case Study

Exercise:

1. **Create Custom Middleware for Request Timing**:
    - Write a custom middleware that logs the time it takes to process each request. The middleware should calculate the request processing time and log it to the console.
2. **Modify Response Body with Middleware**:

- Create a custom middleware that modifies the response body. For instance, if the response body contains a message like "Hello World," change it to "Hello from Custom Middleware!"

3. **Order of Middleware**:

   - Create an ASP.Net Core application with at least five middleware components. Use them to log request details, authenticate users, process requests, and modify responses. Ensure that the order of middleware execution is logically structured.

Case Study:

Company ABC is building an e-commerce website using ASP.Net Core. The website needs to implement several middleware components to handle various tasks, such as:

- Logging requests and responses.
- Authenticating and authorizing users.
- Modifying responses based on user preferences (e.g., adding custom headers).

Your task is to:

1. Design a middleware pipeline that includes logging, authentication, authorization, and response modification.
2. Implement custom middleware to track the time taken to process requests.
3. Test and optimize the middleware pipeline to ensure requests are handled efficiently.

PERFORMANCE OPTIMIZATION

CACHING TECHNIQUES

In modern web applications, **performance optimization** is essential for ensuring fast response times and efficient use of system resources. One of the most effective ways to enhance performance in ASP.Net applications is through **caching**. Caching allows the application to temporarily store data in memory, reducing the need to repeatedly fetch data from slower sources, such as databases or external services.

What is Caching?
Caching involves storing frequently accessed data in a temporary storage (usually in memory) so that future requests can access it more quickly. The goal of caching is to improve application speed by reducing the number of requests made to resource-heavy processes, such as database queries or API calls. Caching can significantly reduce latency and improve response times for end-users.

Types of Caching in ASP.Net Core
ASP.Net Core offers several types of caching mechanisms, each suited for different scenarios:

1. **In-Memory Caching**:
   This is the simplest form of caching, where data is stored directly in the server's memory. It's suitable for smaller data that can be quickly accessed and doesn't require persistence across application restarts.

Example of in-memory caching:

```
public class WeatherService {

  private readonly IMemoryCache _cache;


  public WeatherService(IMemoryCache cache) {

    _cache = cache;

  }


  public string GetWeather(string city) {
```

```
    if (!_cache.TryGetValue(city, out string weather)) {
      weather = FetchWeatherFromApi(city);  // Call to external API or database
      _cache.Set(city, weather, TimeSpan.FromMinutes(5));  // Cache for 5 minutes
    }
    return weather;
  }
}
```

In this example, the weather data is fetched from an external API, but it's cached in memory for five minutes. The next time the same request is made, the data will be retrieved from the cache rather than the external service.

2. **Distributed Caching**:
   In scenarios where your application is hosted on multiple servers or requires scaling, **distributed caching** is used. This allows you to cache data across multiple machines, ensuring that all servers have access to the same cached data. Redis and SQL Server are common distributed caching providers.

Example of configuring Redis caching:

```
public void ConfigureServices(IServiceCollection services) {
  services.AddStackExchangeRedisCache(options => {
    options.Configuration = "localhost";
    options.InstanceName = "SampleInstance";
  });
}
```

With Redis caching, data can be shared across multiple instances of an application, which is essential for applications that need to scale horizontally.

3. **Output Caching**:
   Output caching involves caching the entire response generated by an action method. This is useful when the output of an action is consistent for a set period, such as static content or rarely changing data.

Example of output caching:

```
[ResponseCache(Duration = 60)]

public IActionResult Index() {

   return View();

}
```

Here, the result of the Index action is cached for 60 seconds, meaning that subsequent requests within this time frame will receive the cached version of the page, improving performance by reducing server load.

Benefits of Caching:

- **Faster Response Times**: By avoiding expensive operations (like database queries), caching can drastically reduce the time it takes to return a response to the user.
- **Reduced Server Load**: Caching reduces the need to perform operations that may involve accessing external systems or resources, reducing server load and improving overall system performance.
- **Scalability**: Distributed caching makes it easier to scale applications horizontally by enabling cache sharing across different instances.

Asynchronous Programming in ASP.Net

Asynchronous programming is an essential technique for improving the performance and scalability of web applications, particularly in ASP.Net Core. By allowing I/O-bound operations to execute in the background, asynchronous programming frees up the server to handle other requests concurrently, significantly improving throughput and responsiveness.

What is Asynchronous Programming?
Asynchronous programming is a method that allows certain tasks to be executed independently of the main thread. Instead of waiting for a time-consuming operation (such as reading a file, querying a database, or calling an external API) to finish, the program can continue executing other code. When the operation completes, a callback or continuation is triggered to handle the result.

In ASP.Net Core, asynchronous programming is typically implemented using the async and await keywords. These keywords enable non-blocking I/O operations, allowing multiple requests to be handled simultaneously without unnecessary delays.

Asynchronous I/O Operations

In web applications, I/O-bound operations (such as database queries, file access, or HTTP requests) are often the bottleneck. By making these operations asynchronous, you can avoid blocking the thread while waiting for these tasks to complete. Here's an example:

```
public class ProductService {

   private readonly HttpClient _httpClient;



   public ProductService(HttpClient httpClient) {

      _httpClient = httpClient;

   }



   public async Task<string> GetProductDetailsAsync(string productId) {

      HttpResponseMessage response = await
_httpClient.GetAsync($"https://api.example.com/products/{productId}");

      if (response.IsSuccessStatusCode) {

         return await response.Content.ReadAsStringAsync();

      }

      return null;

   }
}
```

In this example, the GetProductDetailsAsync method fetches product details from an external API asynchronously. The await keyword tells the system to perform the HTTP request asynchronously, and it does not block the main thread, allowing the server to process other requests while waiting for the external API response.

The Importance of Asynchronous Programming in ASP.Net Core

1. **Scalability**: Asynchronous programming helps the server handle more requests simultaneously without consuming additional threads, thus improving scalability.

2. **Improved Performance**: By making I/O-bound operations asynchronous, your application can continue processing other requests while waiting for the completion of a time-consuming task.

3. **Non-blocking Operations**: This allows your application to be more responsive to user input, enhancing the user experience.

Common Use Cases for Asynchronous Programming:

- **Database Queries**: When querying databases, especially large datasets, asynchronous operations can help avoid blocking the request thread, allowing the server to continue processing other requests.

- **File I/O**: Asynchronous file operations, such as reading and writing files, can be performed in the background, preventing delays in the main thread.

- **External API Calls**: Calling external APIs or web services often introduces latency. Asynchronous programming enables your application to remain responsive while waiting for the external service to return data.

Example of Async Database Query:

```
public async Task<List<Product>> GetAllProductsAsync() {

   return await _context.Products.ToListAsync();  // Asynchronous query to the database

}
```

In this example, the ToListAsync method fetches all products from the database asynchronously. While the database query is being processed, the server thread is free to handle other requests.

Best Practices for Asynchronous Programming:

- **Avoid Blocking Calls**: Always use await when calling asynchronous methods to ensure that the request is not blocked.

- **Use async for I/O-bound operations**: Asynchronous programming is most effective for I/O-bound operations. CPU-bound tasks can still benefit from parallelism or multi-threading techniques.

- **Use Cancellation Tokens**: When performing long-running tasks, such as external API calls, always implement cancellation tokens to allow users to cancel requests that are taking too long.

EXERCISE

1. **Implement Caching**: Add caching functionality to an ASP.Net Core application. Implement in-memory caching for frequently accessed data (e.g., user information or product data) to improve response time and reduce database load.

2. **Asynchronous Database Queries**: Refactor a database query method in your ASP.Net Core application to use asynchronous programming. Implement asynchronous GET, POST, and PUT requests to improve performance when interacting with the database.

CASE STUDY: OPTIMIZING AN E-COMMERCE APPLICATION

In this case study, you will optimize an **E-Commerce Application** by implementing both caching and asynchronous programming techniques. The application allows users to view products, add them to their cart, and make purchases.

1. **Caching**: Implement caching for frequently viewed products and category data to minimize database hits and improve load times for users.

2. **Asynchronous Programming**: Refactor database queries and external API calls (e.g., payment gateway integrations) to use asynchronous programming, ensuring the server can handle more concurrent requests and provide a faster response.

Assignment Solution: Secure a Web API using JWT Authentication and Optimize Application Performance using Caching

In this assignment, we will implement two key features in an ASP.Net Core Web API:

1. **JWT Authentication**: Secure the Web API by using JWT (JSON Web Token) for user authentication.
2. **Caching**: Optimize the application's performance by caching frequently accessed data.

We'll follow a step-by-step guide to complete this task.

Step 1: Set Up the ASP.Net Core Web API Project

1. **Create a New Web API Project**: Open Visual Studio and create a new **ASP.Net Core Web API** project.
    - Select **ASP.NET Core Web Application**.
    - Choose the **API** template.
    - Set the project name to SecureApiWithCaching.
2. **Install Necessary Packages**: To implement JWT authentication and caching, you need the following NuGet packages:
    - JWT Bearer authentication package.
    - Caching middleware (Distributed Caching, if needed).

To install these packages, use **NuGet Package Manager** or **Package Manager Console**:

Install-Package Microsoft.AspNetCore.Authentication.JwtBearer

Install-Package Microsoft.Extensions.Caching.Memory

STEP 2: IMPLEMENT JWT AUTHENTICATION

2.1: Configure JWT Authentication in Startup.cs

1. **Add JWT Authentication Services**: In the ConfigureServices method of Startup.cs, add the JWT authentication services and configure them to use your secret key for token validation.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllers();

    // Add JWT Authentication
    var key = Encoding.ASCII.GetBytes(Configuration["Jwt:SecretKey"]);
    services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
        .AddJwtBearer(options =>
        {
            options.TokenValidationParameters = new
TokenValidationParameters
            {
                ValidateIssuer = false,
                ValidateAudience = false,
                ValidateLifetime = true,
                IssuerSigningKey = new SymmetricSecurityKey(key),
                ClockSkew = TimeSpan.Zero
            };
        });
```

20. }

21. **Add JWT Configuration in appsettings.json**: In your appsettings.json, add a secret key for JWT.

22. {

23. "Jwt": {

24. "SecretKey": "your-very-secure-jwt-secret-key"

25. }

26. }

2.2: Create a User Authentication Controller

Create a UserController.cs to handle login and generate JWT tokens for authenticated users.

```
[ApiController]
[Route("api/[controller]")]
public class UserController : ControllerBase
{
  private readonly IConfiguration _configuration;

  public UserController(IConfiguration configuration)
  {
    _configuration = configuration;
  }

  // POST api/user/login
  [HttpPost("login")]
  public IActionResult Login([FromBody] UserLogin userLogin)
```

```
{
    if (userLogin.Username == "test" && userLogin.Password == "password") // Simulated authentication
    {
        var token = GenerateJwtToken();
        return Ok(new { token });
    }
    return Unauthorized("Invalid credentials");
}

private string GenerateJwtToken()
{
    var securityKey = new SymmetricSecurityKey(Encoding.ASCII.GetBytes(_configuration["Jwt:SecretKey"]));
    var credentials = new SigningCredentials(securityKey, SecurityAlgorithms.HmacSha256);
    var token = new JwtSecurityToken(
        issuer: null,
        audience: null,
        expires: DateTime.Now.AddHours(1),
        signingCredentials: credentials
    );

    return new JwtSecurityTokenHandler().WriteToken(token);
}
```

```
}


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

2.3: Protect Routes Using [Authorize]

In order to protect your API endpoints, use the [Authorize] attribute. This will ensure that only users with a valid JWT can access these endpoints.

Example of a protected API:

```
[ApiController]
[Route("api/[controller]")]
[Authorize]
public class DataController : ControllerBase
{
    // GET api/data
    [HttpGet]
    public IActionResult GetSecureData()
    {
        return Ok(new { Data = "This is a secure data response" });
    }
}
```

Now, the GetSecureData method is protected, and users must send a valid JWT token in the Authorization header to access it.

Step 3: Implement Caching to Optimize Application Performance

3.1: Add Caching Service

In Startup.cs, configure the in-memory cache service.

```
public void ConfigureServices(IServiceCollection services)
{
   services.AddControllers();

   // Add Memory Caching
   services.AddMemoryCache();

   // Add JWT Authentication
   var key = Encoding.ASCII.GetBytes(Configuration["Jwt:SecretKey"]);
   services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
       .AddJwtBearer(options =>
       {
          options.TokenValidationParameters = new TokenValidationParameters
          {
             ValidateIssuer = false,
             ValidateAudience = false,
             ValidateLifetime = true,
             IssuerSigningKey = new SymmetricSecurityKey(key),
             ClockSkew = TimeSpan.Zero
```

```
        };
    });
}
```

3.2: Use Caching in Controller

To use caching in your API, inject IMemoryCache into your controller and cache the data returned by API calls. Here's how to implement caching:

```
[ApiController]
[Route("api/[controller]")]
public class DataController : ControllerBase
{
    private readonly IMemoryCache _memoryCache;

    public DataController(IMemoryCache memoryCache)
    {
        _memoryCache = memoryCache;
    }

    // GET api/data
    [HttpGet]
    public IActionResult GetData()
    {
        var cacheKey = "data";
        if (!_memoryCache.TryGetValue(cacheKey, out string cachedData))
        {
            // Simulate some data retrieval from a slow service or database
```

```
        cachedData = "This is some data retrieved from an external source.";


        // Set the cache with a 5-minute expiration
        _memoryCache.Set(cacheKey, cachedData, TimeSpan.FromMinutes(5));
    }

    return Ok(new { Data = cachedData });
  }
}
```

In this example, when the GetData method is called:

- The method first checks if the data is available in the cache.
- If not, it retrieves the data (in this case, it's simulated), and caches it for 5 minutes.
- Subsequent calls will retrieve the data from the cache, improving performance.

3.3: Cache Expiration and Eviction Policies

You can set cache expiration policies such as absolute expiration, sliding expiration, or eviction.

Example of setting a sliding expiration:

```
_memoryCache.Set(cacheKey, cachedData, new MemoryCacheEntryOptions
{
  SlidingExpiration = TimeSpan.FromMinutes(5) // Cache will expire if not accessed in 5 minutes
});
```

STEP 4: TESTING AND VERIFYING THE APPLICATION

1. **Run the API**:
    - Press **Ctrl+F5** to run the application.
    - The API will be available at https://localhost:5001.
2. **Test JWT Authentication**:
    - Use **Postman** to send a POST request to https://localhost:5001/api/user/login with the following JSON body:
    - {
    - "Username": "test",
    - "Password": "password"
    - }
    - You will receive a JWT token in the response. Copy this token.
3. **Test Accessing Protected Route**:
    - Send a GET request to https://localhost:5001/api/data with the Authorization header:
    - Authorization: Bearer <your-jwt-token>
    - You should receive the protected data.
4. **Test Caching**:
    - Make multiple requests to the https://localhost:5001/api/data endpoint. After the first request, the subsequent requests should return the cached data for a period of 5 minutes.

Step 5: Conclusion

You have successfully secured your Web API with JWT authentication and implemented caching to optimize performance. By using JWT, users must authenticate before accessing protected routes. Caching helps improve performance by reducing the load on external services or databases.

EXERCISES

1. **Add Refresh Token Implementation**: Modify the authentication flow to implement refresh tokens, allowing users to get a new JWT token without logging in again.
2. **Use Distributed Caching**: Implement distributed caching using Redis to handle cache storage across multiple instances of the application.
3. **Cache Data in a Database**: Instead of caching hardcoded data, cache data retrieved from a database or an external API.

CASE STUDY

Company XYZ is building a Web API for managing user data and frequently accessed content. The company requires the API to be secured with JWT tokens and optimized for performance by caching data. Your task is to:

1. Implement JWT authentication to secure user data endpoints.
2. Optimize data retrieval using caching, ensuring that frequently accessed data is quickly available without making repeated calls to external services or databases.
3. Test and optimize the application to handle large-scale traffic effectively.