Independent Skill Development Mission

ISDM (INDEPENDENT SKILL DEVELOPMENT MISSION)

INTRODUCTION TO WEB APIS

RESTFUL APIS AND HTTP METHODS (GET, POST, PUT, DELETE)

Web APIs (Application Programming Interfaces) are a fundamental part of modern web development. They allow different systems, applications, or services to communicate with each other over the web using standard protocols. One of the most common and widely adopted styles for designing APIs is **REST** (Representational State Transfer). RESTful APIs adhere to principles that allow them to be scalable, simple to use, and stateless.

What is REST?

REST is an architectural style that relies on stateless communication between the client and the server. This means that every request from a client must contain all the information needed to understand and process the request. RESTful APIs follow a set of constraints that make them lightweight and easy to scale. These constraints include the use of standard HTTP methods, stateless communication, and the use of resources that are identified by URLs.

HTTP Methods in RESTful APIs

In RESTful APIs, the communication between the client and the server typically revolves around four key HTTP methods: **GET**, **POST**, **PUT**, and **DELETE**. These methods correspond to the basic operations of **CRUD** (Create, Read, Update, Delete), which are used to manage resources in an API.

1. **GET Method**:
    - The GET method is used to retrieve data from the server. It is a **read-only** operation and does not alter the data. For example, to retrieve a list of users from an API, you might send a GET request to /api/users.
    - Example:
    - GET /api/users

This request would retrieve a list of all users from the server.

2. **POST Method**:
    - The POST method is used to send data to the server to create a new resource. It is typically used when submitting data to the server, such as when creating a new user, adding an item to a shopping cart, or submitting a form.
    - Example:
    - POST /api/users
    - Content-Type: application/json
    - {
    - "name": "John Doe",
    - "email": "johndoe@example.com"
    - }

This request would create a new user with the provided data on the server.

3. **PUT Method**:
    - The PUT method is used to update an existing resource on the server. It is often used to modify data, such as updating a user's information. The client sends the updated data to the server, and the server replaces the existing resource with the new one.
    - Example:
    - PUT /api/users/1
    - Content-Type: application/json
    - {
    - "name": "John Doe Updated",
    - "email": "john.doe.updated@example.com"
    - }

This request would update the user with ID 1 on the server.

4. **DELETE Method**:
    - The DELETE method is used to remove a resource from the server. For example, when deleting a user from a system, a DELETE request might be sent to /api/users/{id}.
    - Example:
    - DELETE /api/users/1

This request would delete the user with ID 1 from the server.

Each of these methods allows you to perform operations on the resources exposed by a RESTful API, making it possible to interact with data in a standardized way.

Benefits of RESTful APIs:

- **Stateless**: Each request contains all the information needed to process it, making the server more scalable and reliable.
- **Scalability**: Because RESTful APIs rely on stateless communication, they can handle large numbers of clients and requests without affecting performance.
- **Simplicity**: RESTful APIs use standard HTTP methods, making them easy to understand and implement.
- **Flexibility**: RESTful APIs can work with a variety of data formats and transport protocols, but the most common is JSON.

JSON and XML in API Responses

When interacting with APIs, the data is typically exchanged in **JSON (JavaScript Object Notation)** or **XML (Extensible Markup Language)** formats. Both of these formats are human-readable and machine-readable, making them ideal for transmitting data over the web.

JSON (JavaScript Object Notation)
JSON is the most commonly used format for APIs due to its lightweight nature and ease of use. It is easy for humans to read and write, and easy for machines to parse and generate. JSON data consists of key-value pairs, where keys are strings and values can be strings, numbers, arrays, or objects.

Example of JSON:

{

"id": 1,

"name": "John Doe",

"email": "johndoe@example.com",

"isActive": true

}

In this example, the JSON response contains information about a user, including an id, name, email, and isActive status. JSON is highly efficient for sending data, especially in RESTful APIs, because it is easy to parse and integrate with JavaScript, which is commonly used in client-side development.

Why Use JSON?

- **Lightweight**: JSON is less verbose compared to XML, meaning it uses fewer resources when transmitting data.
- **Human-readable**: JSON is easy to read and understand, making it great for debugging and logging.
- **Easily parsed**: JSON is directly supported by most modern programming languages, making it easy to convert into usable objects or structures.

XML (Extensible Markup Language)
XML is another format used for API responses, although it is less common than JSON. XML uses tags to define data, and these tags can be nested to create complex structures. While XML is more flexible than JSON and can represent more complex data types, it is also more verbose and harder to parse.

Example of XML:

<user>

<id>1</id>

<name>John Doe</name>

<email>johndoe@example.com</email>

<isActive>true</isActive>

</user>

In this XML response, the same user data is returned as it is in the JSON example, but XML uses opening and closing tags for each piece of data, which makes it more verbose.

Why Use XML?

- **Flexibility**: XML allows for more complex data representations and can handle nested structures more naturally.
- **Platform independent**: XML is supported across different platforms and programming languages.
- **Metadata capabilities**: XML allows you to add metadata through attributes in the tags, which is useful for certain use cases.

JSON vs XML

While both JSON and XML are used for transmitting data, JSON has become the preferred choice for RESTful APIs due to its lightweight nature, simplicity, and ease of parsing. However, XML is still used in legacy systems or in cases where its features, like metadata and extensibility, are needed.

Example: API Response Formats

- **JSON Response**:
- {
- "userId": 101,
- "name": "Alice",
- "status": "active"
- }
- **XML Response**:
- <user>
- <userId>101</userId>
- <name>Alice</name>
- <status>active</status>
- </user>

EXERCISE

1. **Create an API to Retrieve Users**:
    - Create a RESTful API with ASP.Net Core that handles the following HTTP methods: GET, POST, PUT, and DELETE for managing a list of users.
    - Implement the GET method to retrieve all users in JSON format.
    - Implement the POST method to create a new user.
    - Implement the PUT method to update a user's information.
    - Implement the DELETE method to delete a user.
2. **Implement Data Serialization**:
    - Modify your API to support both **JSON** and **XML** responses based on the Accept header in the HTTP request.
    - Test your API using Postman or a similar tool by sending requests with the Accept: application/json and Accept: application/xml headers and examining the responses.

CASE STUDY: BUILDING A RESTFUL API FOR A BOOKSTORE

In this case study, you will build a **Bookstore API** that allows users to manage books in a system. The API will implement **CRUD** operations for books and return data in both JSON and XML formats.

1. **Entities**: The Book entity will have the following properties: Id, Title, Author, Genre, and Price.
2. **Endpoints**: Create RESTful endpoints to:
    - **GET** all books: /api/books
    - **POST** a new book: /api/books
    - **PUT** an existing book: /api/books/{id}
    - **DELETE** a book: /api/books/{id}

3. **Response Format**: Ensure the API returns responses in both JSON and XML formats depending on the request headers.

CREATING WEB APIS WITH ASP.NET CORE

SETTING UP A WEB API IN ASP.NET CORE

Web APIs are essential components for building web applications that expose data or functionality to be consumed by other services, client-side applications, or mobile apps. ASP.Net Core provides a powerful and flexible framework to create Web APIs, enabling developers to build RESTful services that follow HTTP standards and can interact with any platform or application that supports HTTP requests.

What is a Web API?

A Web API (Application Programming Interface) allows different software applications to communicate with each other over the internet. It enables one application to expose its data or functionality to be accessed and used by another. Web APIs are typically built following REST (Representational State Transfer) principles, where HTTP methods (GET, POST, PUT, DELETE) are used to interact with resources.

STEPS TO CREATE A WEB API IN ASP.NET CORE

Setting up a Web API in ASP.Net Core is straightforward and involves creating an ASP.Net Core application, configuring the necessary services, and defining the routes to handle API requests.

1. **Create a New Web API Project**: Open Visual Studio and create a new project. Choose **ASP.NET Core Web Application** and select the **API** template. This template sets up the basic project structure for an API, including a Controllers folder and the necessary dependencies for building and running an API.
2. **Configure Services**: In the Startup.cs file, you need to configure the services required for the Web API. This typically involves configuring dependency injection and enabling support for controllers. Here's a basic example of configuring services:
3. public class Startup
4. {
5. public void ConfigureServices(IServiceCollection services)
6. {
7. services.AddControllers(); // Adds the necessary services for controllers

```
8.    }
9.
10.   public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
11.   {
12.     if (env.IsDevelopment())
13.     {
14.       app.UseDeveloperExceptionPage();
15.     }
16.
17.     app.UseRouting();
18.
19.     app.UseEndpoints(endpoints =>
20.     {
21.       endpoints.MapControllers(); // Maps API controllers to routes
22.     });
23.   }
24. }
```

25. **Add Controllers**: In the Controllers folder, create a new controller class to define the endpoints for your Web API. A controller is a class that handles incoming HTTP requests, processes them, and returns a response. In Web APIs, controllers typically return data in formats like JSON or XML. Here's an example of a basic API controller:

```
26. [ApiController]
27. [Route("api/[controller]")]
28. public class BooksController : ControllerBase
29. {
30.   private static List<Book> books = new List<Book>
```

```
{
    new Book { Id = 1, Title = "C# Programming", Author = "John Doe" },
    new Book { Id = 2, Title = "ASP.Net Core", Author = "Jane Smith" }
};

[HttpGet]
public IEnumerable<Book> GetBooks()
{
    return books;
}

[HttpGet("{id}")]
public ActionResult<Book> GetBook(int id)
{
    var book = books.FirstOrDefault(b => b.Id == id);
    if (book == null) return NotFound();
    return book;
}
}
```

In this example, the BooksController defines two endpoints: one for fetching all books (GET /api/books) and another for fetching a single book by ID (GET /api/books/{id}). The [ApiController] attribute ensures that the controller responds to API requests, and the [Route] attribute defines the base route for the controller's actions.

4. **Run the API**: Once the controller is set up, you can run the Web API by pressing **F5** or **Ctrl+F5**. The API will be available at https://localhost:5001/api/books by default, depending on your project's configuration.

ROUTING AND CONTROLLERS IN WEB API

Routing is the process of mapping HTTP requests to appropriate controller actions based on the URL patterns. In ASP.Net Core, routing is defined by attributes that are applied to controller actions, and it plays a central role in the API's functionality. The controller is responsible for handling the requests and returning the appropriate data or responses. Let's explore both routing and controllers in detail.

Routing in Web API

Routing in ASP.Net Core Web APIs uses attribute routing, which means that routes are defined directly on the controller actions using attributes such as [Route], [HttpGet], [HttpPost], etc. The routing system allows for defining flexible and hierarchical URL patterns.

Basic Routing

The [Route] attribute is used to define a base route for the controller or action method. A route can include placeholders for dynamic data, such as the id in api/books/{id}.

Example of routing in ASP.Net Core:

```
[Route("api/[controller]")]
[ApiController]
public class BooksController : ControllerBase
{
    // GET: api/books
    [HttpGet]
    public IEnumerable<Book> GetBooks()
    {
        return books;
    }

    // GET: api/books/5
    [HttpGet("{id}")]
    public ActionResult<Book> GetBook(int id)
```

```
    {
        var book = books.FirstOrDefault(b => b.Id == id);
        if (book == null)
        {
            return NotFound();
        }
        return book;
    }
}
```

In this example, [Route("api/[controller]")] means that the route will be api/books, where [controller] is a placeholder for the controller's name, automatically replaced with the controller's name without the "Controller" suffix (in this case, Books). The [HttpGet("{id}")] attribute indicates that this action method will handle requests to GET /api/books/{id}, where {id} is a route parameter.

Route Constraints

Route parameters can also be constrained to specific types. For example, you can define a route that only accepts integers by adding a constraint to the route parameter:

```
[HttpGet("{id:int}")]
public ActionResult<Book> GetBook(int id)
{
    // Logic to fetch book by ID
}
```

This ensures that only valid integers are passed as id in the route.

Controllers in Web API

Controllers in Web APIs handle the HTTP requests and define the logic for interacting with data. A controller is a class that inherits from ControllerBase (instead of Controller used in MVC). The main purpose of a Web API controller is to process requests, perform the necessary operations, and return responses, typically in JSON format.

Controller Actions

Each action in a controller corresponds to an HTTP verb such as GET, POST, PUT, or DELETE. These actions are decorated with attributes like [HttpGet], [HttpPost], [HttpPut], and [HttpDelete] to specify which HTTP methods they respond to.

Example of Controller Actions:

```
[HttpPost]
public ActionResult<Book> CreateBook([FromBody] Book book)
{
   books.Add(book);
   return CreatedAtAction(nameof(GetBook), new { id = book.Id }, book);
}


[HttpPut("{id}")]
public IActionResult UpdateBook(int id, [FromBody] Book book)
{
   var existingBook = books.FirstOrDefault(b => b.Id == id);
   if (existingBook == null)
   {
      return NotFound();
   }
   existingBook.Title = book.Title;
   existingBook.Author = book.Author;
   existingBook.Price = book.Price;
   return NoContent();
}
```

In this example:

- The CreateBook method handles POST requests to create a new book and returns a 201 Created response with the location of the new resource.

- The UpdateBook method handles PUT requests to update an existing book, responding with 204 No Content if the operation is successful.

Action Results

ASP.Net Core Web API controllers typically return instances of ActionResult, which can represent various types of HTTP responses, such as Ok(), NotFound(), BadRequest(), or CreatedAtAction().

Exercises and Case Study

Exercise:

1. **Create a Web API for Managing Movies:**
    - Design a Movie model with properties like Id, Title, Director, and ReleaseDate.
    - Implement a controller with actions for:
        - Creating a new movie (POST).
        - Getting a list of movies (GET).
        - Updating a movie (PUT).
        - Deleting a movie (DELETE).
2. **Implement Route Constraints:**
    - Modify the GetBook action in the BooksController to accept only positive integers for the id route parameter.

Case Study:

A company XYZ is developing an online bookstore and needs to create a Web API to manage the books in their catalog. The API should allow users to:

1. Add new books to the catalog.
2. Get a list of all books.
3. Update the information of a specific book.
4. Delete a book from the catalog.

Your task is to:

- Set up a Web API project with ASP.Net Core.
- Design the Book model and the corresponding controller.
- Implement the necessary routes and controller actions for managing the book catalog.

INTEGRATING THIRD-PARTY APIS

CONSUMING EXTERNAL APIS IN ASP.NET APPLICATIONS

Third-party APIs allow ASP.Net developers to extend the functionality of their applications by integrating with external services. These APIs provide data or services from sources outside the application, such as payment gateways, social media platforms, weather services, and more. Consuming these external APIs in an ASP.Net application involves making HTTP requests, handling responses, and processing the data.

Making API Requests

ASP.Net Core provides several ways to consume external APIs. The most common method is to use the HttpClient class, which is part of the System.Net.Http namespace. This class allows you to send HTTP requests and receive HTTP responses from external APIs.

To start consuming an API, you typically need the following:

- **API endpoint URL**: The URL where the external service can be accessed.
- **API key or authentication token**: Some APIs require an authentication token or key to access their endpoints.
- **Request parameters**: Depending on the API, you may need to send additional parameters in the request, either in the URL or in the body of the request.

Here's an example of how to consume a third-party API using HttpClient in an ASP.Net Core application:

```
public class ExternalApiService {
  private readonly HttpClient _httpClient;

  public ExternalApiService(HttpClient httpClient) {
    _httpClient = httpClient;
  }

  public async Task<string> GetWeatherDataAsync(string city) {
```

```
    string apiUrl = $"https://api.openweathermap.org/data/2.5/weather?q={city}&appid=your_api_key";
    HttpResponseMessage response = await _httpClient.GetAsync(apiUrl);

    if (response.IsSuccessStatusCode) {
      string responseData = await response.Content.ReadAsStringAsync();
      return responseData;  // Return JSON data
    } else {
      return "Error retrieving data";
    }
  }
}
```

In this example, HttpClient is used to send a GET request to the **OpenWeather API** to get weather data for a specific city. The GetAsync() method sends the request, and the response.Content.ReadAsStringAsync() method reads the response.

Once the response is received, you can process the data, typically in JSON format, and extract the relevant information. In the case of the weather API, this could be temperature, humidity, wind speed, etc.

Handling JSON Data

External APIs often return data in **JSON format**, which is easy to parse and work with in .NET applications. You can use libraries like **Newtonsoft.Json** or the built-in **System.Text.Json** to deserialize the JSON response into C# objects.

Example of parsing JSON using **Newtonsoft.Json**:

```
public class WeatherResponse {
  public MainData Main { get; set; }
}

public class MainData {
  public double Temp { get; set; }
```

```
}

public async Task<double> GetTemperatureAsync(string city) {
  string responseData = await GetWeatherDataAsync(city);
  WeatherResponse weather = JsonConvert.DeserializeObject<WeatherResponse>(responseData);
  return weather.Main.Temp;
}
```

Here, we deserialize the JSON response from the weather API into a C# object (WeatherResponse) and extract the Temp property.

Error Handling

When consuming third-party APIs, it's crucial to handle errors such as network issues, invalid responses, or authentication errors. You can check for status codes like 404 (Not Found) or 500 (Internal Server Error) in the API response, and gracefully handle failures.

Best Practices for Consuming External APIs:

- **Error Handling**: Always check the HTTP status code and handle different types of errors (e.g., timeouts, not found).
- **Timeouts**: Set timeouts on API requests to prevent your application from hanging indefinitely.
- **Rate Limiting**: Some APIs impose rate limits, so it's important to manage and throttle requests if needed.
- **Async Programming**: Use asynchronous methods (async and await) to avoid blocking your application when making HTTP requests.

Authentication and Security for API Calls

When consuming third-party APIs, **authentication and security** are critical concerns. Many APIs require some form of authentication to ensure that only authorized users can access their data. This is typically done via **API keys**, **OAuth tokens**, or **JWT (JSON Web Tokens)**.

API Keys

An **API key** is a unique identifier sent with each request to authenticate the client application. Most APIs require you to register for an API key, which is then used in the header or URL of the request.

Here's how to include an API key in an HTTP request:

```
public async Task<string> GetDataWithApiKeyAsync(string apiUrl) {
  _httpClient.DefaultRequestHeaders.Add("Authorization", "Bearer your_api_key");
  HttpResponseMessage response = await _httpClient.GetAsync(apiUrl);
  if (response.IsSuccessStatusCode) {
    return await response.Content.ReadAsStringAsync();
  }
  return null;
}
```

In this case, the API key is added to the Authorization header as a Bearer token. The key is typically sent over HTTPS to ensure security during transmission.

OAuth Authentication

For more complex APIs, **OAuth** is a widely used protocol for authentication. OAuth allows the user to authorize a third-party application to access their data without sharing their credentials. It involves obtaining an access token through an authorization flow and using it to authenticate subsequent requests.

Here's a simplified flow of how OAuth works:

1. The user is redirected to the authorization server to grant access.
2. The authorization server issues an **access token**.
3. The application uses the access token to make authenticated API requests.

Example of using OAuth:

```
public async Task<string> GetProtectedDataAsync() {
  var client = new HttpClient();
  var requestMessage = new HttpRequestMessage(HttpMethod.Get, "https://api.thirdparty.com/data");
  requestMessage.Headers.Authorization = new AuthenticationHeaderValue("Bearer", "your_oauth_token");
  HttpResponseMessage response = await client.SendAsync(requestMessage);
```

```
    if (response.IsSuccessStatusCode) {
        return await response.Content.ReadAsStringAsync();
    }
    return null;
}
```

In this example, the OAuth token is included in the Authorization header as a Bearer token.

JSON Web Tokens (JWT)

JWT is another widely used method for securely transmitting information between a client and server. It's commonly used for user authentication in web applications and APIs. JWTs contain claims (user data or permissions) and are signed by a server to ensure integrity.

Example of including JWT in an API request:

```
public async Task<string> GetProtectedResourceAsync(string jwtToken) {
    _httpClient.DefaultRequestHeaders.Authorization = new AuthenticationHeaderValue("Bearer", jwtToken);
    HttpResponseMessage response = await _httpClient.GetAsync("https://api.thirdparty.com/resource");
    if (response.IsSuccessStatusCode) {
        return await response.Content.ReadAsStringAsync();
    }
    return null;
}
```

In this case, the JWT token is sent as a Bearer token in the Authorization header to authenticate the API request.

Security Best Practices:

- **HTTPS**: Always use HTTPS for API calls to protect sensitive data, such as API keys or tokens, from being intercepted.
- **Store API Keys Securely**: Never hard-code API keys in the source code. Use environment variables or configuration files to store sensitive information.

- **Limit Permissions**: Use the principle of least privilege when requesting access tokens. Only request the permissions necessary for the task.
- **Rate Limiting and Throttling**: Protect your application by implementing rate limiting to avoid overloading the API or triggering bans.
- **JWT Expiration**: Ensure JWT tokens have expiration times, and refresh tokens when necessary to maintain secure access.

EXERCISE

1. **Consume a Third-Party API**: Use the HttpClient in ASP.Net Core to consume an external weather API, such as the OpenWeather API. Extract and display the temperature, humidity, and wind speed from the JSON response.
2. **Implement Authentication**: Implement an OAuth authentication flow to interact with a third-party API. Use the provided OAuth credentials to get an access token and make authenticated requests.
3. **JWT Integration**: Integrate JWT-based authentication into an ASP.Net Core API. Secure your API endpoints with JWT and test them using tools like Postman.

CASE STUDY: BUILDING A SOCIAL MEDIA INTEGRATION IN ASP.NET CORE

In this case study, you will integrate a third-party social media API (e.g., Twitter or Facebook) into an ASP.Net Core application. The application will allow users to log in using their social media accounts and retrieve their profile information.

1. **OAuth Integration**: Implement OAuth 2.0 to allow users to log in using their social media credentials. Use the HttpClient to make authenticated requests to the social media API and retrieve the user's profile information.
2. **Secure API Calls**: Use JWT or OAuth tokens to secure the API calls and ensure that only authenticated users can access their profile data.
3. **Data Display**: Display the user's profile data (e.g., name, profile picture, recent posts) on a dashboard in your application.

Assignment Solution: Build a Simple API for User Authentication and Integrate it with an External API for Data Retrieval

In this assignment, we will build a simple API for user authentication using **ASP.Net Core** and integrate it with an external API to retrieve some data. The application will consist of two main components:

1. **User Authentication API**: To allow users to authenticate and receive a JWT (JSON Web Token) for further access.
2. **External API Integration**: The application will fetch data from an external API after successful authentication.

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

Step 1: Setting Up the ASP.Net Core Project

1. **Create a New Project**: Open Visual Studio and create a new **ASP.Net Core Web API** project.
    - Choose **ASP.NET Core Web Application**.
    - Select **API** as the project template.
    - Set the project name to UserAuthAPI.
2. **Configure Project Dependencies**:
    - To handle authentication, install **JWT Bearer Authentication** and **HttpClient** packages.
    - You can install these packages via **NuGet Package Manager** or **Package Manager Console**:
    - Install-Package Microsoft.AspNetCore.Authentication.JwtBearer
    - Install-Package System.Net.Http.Json

STEP 2: SET UP JWT AUTHENTICATION

1. **Configure JWT Authentication in Startup.cs**: To set up JWT Bearer authentication, modify the ConfigureServices method in the Startup.cs file. You will configure JWT bearer authentication to authenticate the users.

```
2. public void ConfigureServices(IServiceCollection services)
3. {
4.     services.AddControllers();
5.
6.     // JWT Authentication
7.     var key = Encoding.ASCII.GetBytes(Configuration["Jwt:SecretKey"]);
8.     services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
9.         .AddJwtBearer(options =>
10.        {
11.            options.TokenValidationParameters = new
TokenValidationParameters
12.            {
13.                ValidateIssuer = false,
14.                ValidateAudience = false,
15.                ValidateLifetime = true,
16.                IssuerSigningKey = new SymmetricSecurityKey(key),
17.                ClockSkew = TimeSpan.Zero
18.            };
19.        });
20. }
```

21. **Add Configuration in appsettings.json**: Add a secret key for JWT in the appsettings.json file.

```
22. {
23.   "Jwt": {
```

```
  "SecretKey": "your-very-secure-key-here"
}
}
```

Replace "your-very-secure-key-here" with your secret key. You will use this key to sign and verify JWT tokens.

STEP 3: CREATE USER AUTHENTICATION CONTROLLER

1. **Create a UserController to Handle Authentication**: In the Controllers folder, create a new controller named UserController.cs. This controller will handle user login and issue JWT tokens.

```
[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") // Simple check (You can integrate with DB here)
```

```
18.     {
19.         var token = GenerateJwtToken();
20.         return Ok(new { token });
21.     }
22.     return Unauthorized("Invalid credentials");
23. }
24.
25. private string GenerateJwtToken()
26. {
27.     var securityKey = new SymmetricSecurityKey(Encoding.ASCII.GetBytes(_configuration["Jwt:SecretKey"]));
28.     var credentials = new SigningCredentials(securityKey, SecurityAlgorithms.HmacSha256);
29.     var token = new JwtSecurityToken(
30.         null,
31.         null,
32.         expires: DateTime.Now.AddHours(1),
33.         signingCredentials: credentials
34.     );
35.
36.     return new JwtSecurityTokenHandler().WriteToken(token);
37. }
38. }
39.
40. public class UserLogin
41. {
```

```
42.  public string Username { get; set; }
43.  public string Password { get; set; }
44. }
```

Explanation:

- The Login action checks if the user credentials match hardcoded values ("test" and "password"). In a real application, this would be replaced with a database lookup.
- If valid, it generates a JWT token and returns it to the client.

STEP 4: CREATE EXTERNAL API INTEGRATION CONTROLLER

1. **Set Up the ExternalDataController to Fetch Data**: Now, we will create a controller to fetch data from an external API after successful authentication. We will use HttpClient to make HTTP requests to the external API.

In the Controllers folder, create a new controller named ExternalDataController.cs.

```
[ApiController]
[Route("api/[controller]")]
[Authorize] // This ensures authentication is required
public class ExternalDataController : ControllerBase
{
  private readonly IHttpClientFactory _httpClientFactory;

  public ExternalDataController(IHttpClientFactory httpClientFactory)
  {
    _httpClientFactory = httpClientFactory;
  }

  // GET api/externaldata
```

```
    [HttpGet]
    public async Task<IActionResult> GetExternalData()
    {
        var client = _httpClientFactory.CreateClient();
        var response = await client.GetAsync("https://api.publicapis.org/entries"); // Example external API

        if (response.IsSuccessStatusCode)
        {
            var data = await response.Content.ReadAsStringAsync();
            return Ok(data);
        }

        return BadRequest("Failed to retrieve data from external API");
    }
}
```

Explanation:

- The ExternalDataController is decorated with [Authorize], which ensures that only authenticated users (with a valid JWT) can access this endpoint.
- The GetExternalData action fetches data from an external public API (https://api.publicapis.org/entries) using HttpClient. You can replace the URL with any external API you wish to integrate.

STEP 5: REGISTER HTTP CLIENT IN STARTUP.CS

You need to register HttpClient in the ConfigureServices method so that it can be used in the ExternalDataController:

```
public void ConfigureServices(IServiceCollection services)
```

```
{
    services.AddControllers();

    // Register HttpClient for external API calls
    services.AddHttpClient();

    // 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
            };
        });
}
```

STEP 6: TEST THE APPLICATION

1. **Run the API**:
    - Press **Ctrl+F5** to run the application.
    - The API will be available at https://localhost:5001.

2. **Test User Login**:
    - Use a tool like **Postman** to send a POST request to https://localhost:5001/api/user/login with the following JSON body:
    - {
    - "Username": "test",
    - "Password": "password"
    - }
    - If the credentials are correct, you will receive a JWT token.
3. **Use the JWT Token to Access Data**:
    - Copy the token from the login response.
    - Send a GET request to https://localhost:5001/api/externaldata with the following **Authorization** header:
    - Authorization: Bearer <your-jwt-token>
    - You should receive data from the external API if the token is valid.

STEP 7: CONCLUSION

Congratulations! You have successfully built a simple API for user authentication and integrated it with an external API for data retrieval. This project demonstrated how to:

- Set up JWT authentication in ASP.Net Core.
- Create an authentication API for login and token generation.
- Integrate an external API using HttpClient to retrieve data after user authentication.

EXERCISE

1. Modify the authentication logic to check the user's credentials against a database (e.g., using Entity Framework).
2. Add a user registration feature to allow new users to sign up and receive JWT tokens.

3. Integrate a different external API (e.g., a weather API) and display relevant data based on the user's request.

Case Study

Company XYZ is building a Web API for user management, including authentication and integration with external data sources (e.g., financial data from an external service). The company requires an API that handles user login via JWT tokens and allows authenticated users to retrieve external data. Your task is to:

- Design and implement the API for user authentication.
- Integrate the API with an external data source (e.g., stock prices, weather information).
- Ensure that only authenticated users can access external data.