Problem Set 4 - Distributed Fault Tolerance

Argamosa, Daniel Cedric (S14)
Donato, Adriel Joseph (S12)

Key Implementation Steps

Network Setup

- In the distributed system, it is using both a host machine and a virtual machine. For our configuration, we opted to use the host machine which runs the **Frontend** and the **Broker** components, while the **virtual machine** (running Windows 11) hosts the **AuthService** and **CourseService**.
- To enable proper communication between these components, the configured virtual machine to use a Bridged Adapter. This setting allows the VM to have its own IP address on the local network. For example, my VM was assigned `192.168.100.64`. This IP was used in the Broker configuration to forward requests to the services running on the VM.
- Each service was configured to listen on a specific port. **AuthService** listens on port **4000**, and **CourseService** on port **5000**. The **Broker** uses these addresses to relay API calls from the **Frontend** to the appropriate backend service.

Schema: https://json.schemastore.org/appsettings.json

```
{
  "ServiceEndpoints": {
    "AuthService": "http://192.168.100.64:4000",
    "CourseService": "http://192.168.100.64:5000"
  },
  "Logging": {
    "LogLevel": {
      "Default": "Information"
    }
  },
  "AllowedHosts": "*"
}
```

Key Implementation Steps

OAuth2 and JWT Implementation

- User authentication is secured using **OAuth2** concepts with **JWT** tokens. When a user logs in via the **AuthService**, their credentials are verified against seeded dummy data in the database.
- If the login is successful, the **AuthService** generates a **JWT** token containing the user's username and role.
- This token is then sent back to the **Frontend**, where it is stored in the user session. The token is attached to every subsequent request as a **Bearer** token in the HTTP header, ensuring secure access to protected routes.This design allows services to authenticate and authorize requests without maintaining a centralized session, aligning with modern distributed systems practices.

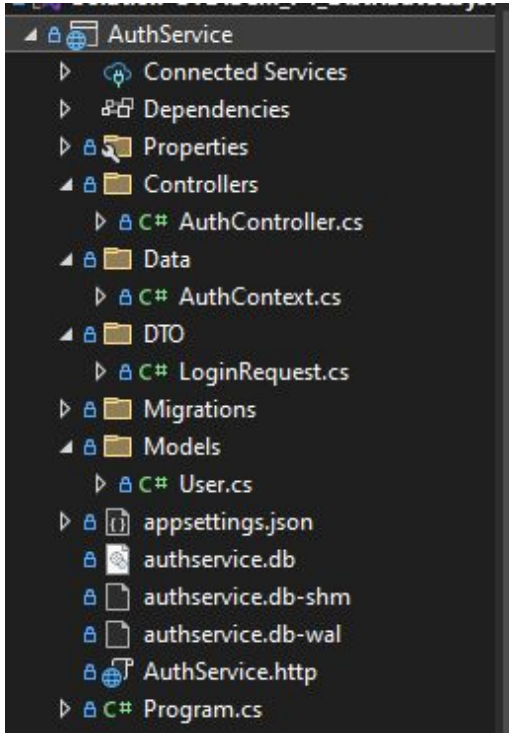
```
// POST /login
[HttpPost("login")]
0 references
public IActionResult Login([FromBody] LoginRequest request)
{
    // Query the database using only username and password.
    var user = _context.Users
        .FirstOrDefault(u =>
            u.Username.ToLower() == request.Username.ToLower()
            && u.Password == request.Password);

    if (user == null)
    {
        return Unauthorized(new { message = "Invalid credentials" });
    }

    var token = GenerateJwtToken(user);
    return Ok(new { token });
}
```

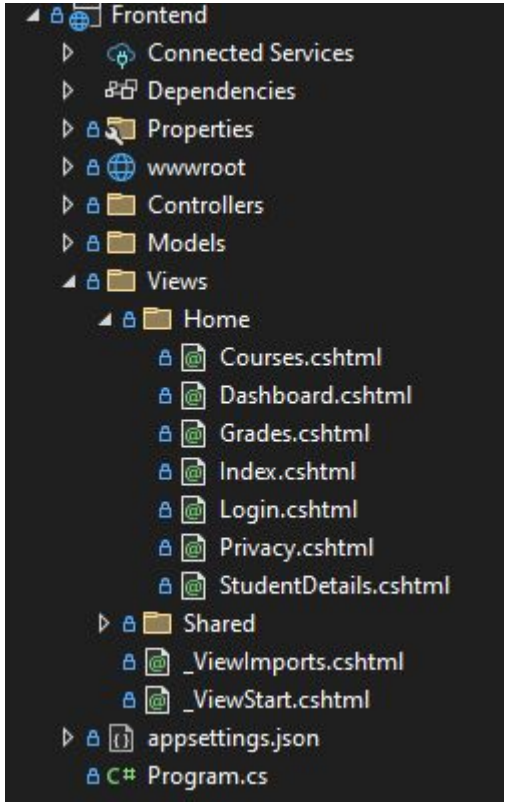
```
1 reference
private string GenerateJwtToken(User user)
{
    var key = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(_config["Jwt:Key"]));
    var creds = new SigningCredentials(key, SecurityAlgorithms.HmacSha256);
    var claims = new[]
    {
        new Claim(JwtRegisteredClaimNames.Sub, user.Username),
        new Claim("role", user.Role),
        new Claim(JwtRegisteredClaimNames.Jti, Guid.NewGuid().ToString())
    };

    var token = new JwtSecurityToken(
      issuer: _config["Jwt:Issuer"],
      audience: _config["Jwt:Audience"],
      claims: claims,
      expires: DateTime.Now.AddHours(1),
      signingCredentials: creds);

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

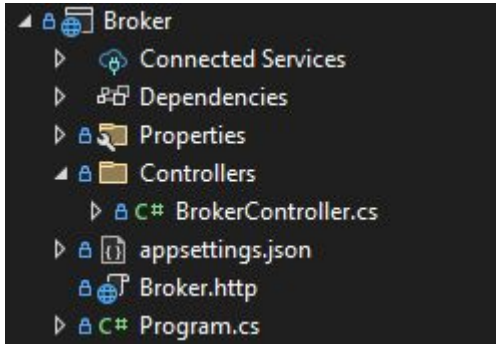
Key Implementation Steps

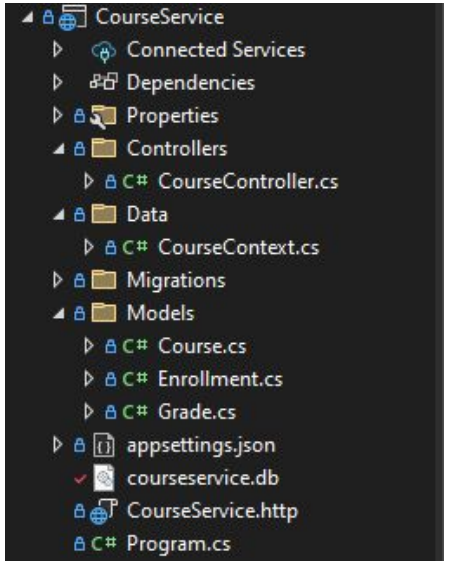
MVC Architecture

- The solution uses a clean MVC architecture across all components.
  - The **Frontend** is built with ASP.NET Core MVC using Razor views to render the UI.
  - The **Broker** acts as a middleware API router, receiving requests from the Frontend and forwarding them to either **AuthService** or **CourseService**.
  - The **AuthService** and **CourseService** are ASP.NET Core Web APIs that handle their respective business logic and database operations.
- Each layer is responsible for a distinct function:
  - The **View** layer handles UI interaction.
  - The **Controller** layer processes user inputs and communicates with models or services.
  - The **Model** layer interacts with the database.
- This structure supports maintainability, scalability, and clarity in the codebase.









Key Implementation Steps

Feature Demonstrations

- The system implements all required features along with additional enhancements:
  - **Login/Logout** – Users log in via the Frontend, and their JWT token is stored in session. Logout clears this session.
  - **View Courses** – The Frontend requests the list of available courses via the Broker, which forwards the call to the CourseService.
  - **Enroll in Courses** – Students can enroll in open courses. A duplicate enrollment check is enforced.
  - **View Grades** – Students can view their grades per course, retrieved from CourseService.Faculty
  - **Features** – Faculty can upload grades and view detailed student data, including enrolled courses and submitted grades.
- These features are fully integrated with role-based access control.

```
// POST: /Home/Enroll
[HttpPost]
0 references
public async Task<IActionResult> Enroll(string courseId)
{
    if (HttpContext.Session.GetString("Token") == null)
        return RedirectToAction("Login");

    var brokerUrl = _config["BrokerUrl"] + "/enroll";
    _client.DefaultRequestHeaders.Clear();
    _client.DefaultRequestHeaders.Add("Authorization", "Bearer " + HttpContext.Session.GetString("Token"));
    var payload = new { username = HttpContext.Session.GetString("Username"), courseId = courseId };
    try
    {
        var response = await _client.PostAsJsonAsync(brokerUrl, payload);
        if (!response.IsSuccessStatusCode)
        {
            var errorContent = await response.Content.ReadAsStringAsync();
            TempData["Error"] = JObject.Parse(errorContent)["message"]?.ToString() ?? "Error enrolling in course";
        }
        else
        {
            TempData["Message"] = JObject.Parse(await response.Content.ReadAsStringAsync())["message"]?.ToString();
        }
    }
    catch (System.Exception ex)
    {
        TempData["Error"] = ex.Message;
    }
    return RedirectToAction("Courses");
}
```

```
// GET: /Home/Grades
0 references
public async Task<IActionResult> Grades()
{
    if (HttpContext.Session.GetString("Token") == null)
        return RedirectToAction("Login");

    var username = HttpContext.Session.GetString("Username");
    var brokerUrl = _config["BrokerUrl"] + "/grades?username=" + username;
    _client.DefaultRequestHeaders.Clear();
    _client.DefaultRequestHeaders.Add("Authorization", "Bearer " + HttpContext.Session.GetString("Token"));
    JArray grades;
    try
    {
        var response = await _client.GetAsync(brokerUrl);
        var json = await response.Content.ReadAsStringAsync();
        grades = JArray.Parse(JObject.Parse(json)["grades"].ToString());
    }
    catch
    {
        ViewBag.Error = "Unable to connect to course service.";
        grades = new JArray();
    }
    return View(grades);
}
```

Explanations on How Fault Tolerance is Achieved

Fault Tolerance Overview

- One of the core strengths of this system is fault tolerance. Each node, **Frontend**, **Broker**, **AuthService**, and **CourseService**, is deployed independently. If any one node goes down, only the feature associated with that node becomes unavailable.
- For example, if **AuthService** is down, users cannot log in, but already-logged-in users can still access course-related features. If **CourseService** is down, course enrollment and grade viewing become unavailable, but users can still log in and navigate the dashboard.
- This is achieved through try-catch error handling in the **Broker**, and informative error messages passed back to the **Frontend**. These error messages are displayed directly on the UI, providing a clear user experience even when partial outages occur.

```
[HttpPost("enroll")]
0 references
public async Task<IActionResult> ForwardEnroll([FromBody] object payload)
{
    var courseUrl = _config["ServiceEndpoints:CourseService"] + "/enroll";
    Console.WriteLine("Forwarding login to: " + courseUrl);
    try
    {
        var response = await _client.PostAsJsonAsync(courseUrl, payload);
        var content = await response.Content.ReadAsStringAsync();
        Console.WriteLine("Response from Service: " + content);
        return StatusCode((int)response.StatusCode, content);
    }
    catch (Exception ex)
    {
        Console.WriteLine("Error in forwarding login: " + ex.Message);
        return StatusCode(503, new { message = "Course service is down" });
    }
}
```

Broker: ForwardEnroll with Try-Catch Fault Handling

```
// GET /health
[HttpGet("health")]
0 references
public IActionResult Health() => Ok(new { status = "OK" });
```

Health Check Endpoint (e.g., AuthService/Health):

```
<h1>Student Details</h1>
@if (ViewBag.Error != null)
{
    <p style="color:red;"><strong>Error:</strong> @ViewBag.Error</p>
}
```

frontend: Display Error Message in View

Explanations on How Fault Tolerance is Achieved

Code Snippet – Fault Isolation

- The following snippet from the **Broker** demonstrates fault isolation:

```
[HttpPost("enroll")]
0 references
public async Task<IActionResult> ForwardEnroll([FromBody] object payload)
{
    var courseUrl = _config["ServiceEndpoints:CourseService"] + "/enroll";
    Console.WriteLine("Forwarding login to: " + courseUrl);
    try
    {
        var response = await _client.PostAsJsonAsync(courseUrl, payload);
        var content = await response.Content.ReadAsStringAsync();
        Console.WriteLine("Response from Service: " + content);
        return StatusCode((int)response.StatusCode, content);
    }
    catch (Exception ex)
    {
        Console.WriteLine("Error in forwarding login: " + ex.Message);
        return StatusCode(503, new { message = "Course service is down" });
    }
}
```

- When **CourseService** is unreachable, the **Broker** catches the error and responds with a friendly message that the **Frontend** then displays.

Summary

Summary

- We used ASP.NET Core MVC and Web APIs to build a distributed enrollment system
- JWT-based authentication secures the system with role-based access
- Each feature is encapsulated in its own service node, enabling isolated failures
- The system uses a middleware Broker to ensure clean service-to-service communication.
- UI error feedback ensures a graceful user experience during node failures