Problem Set 4

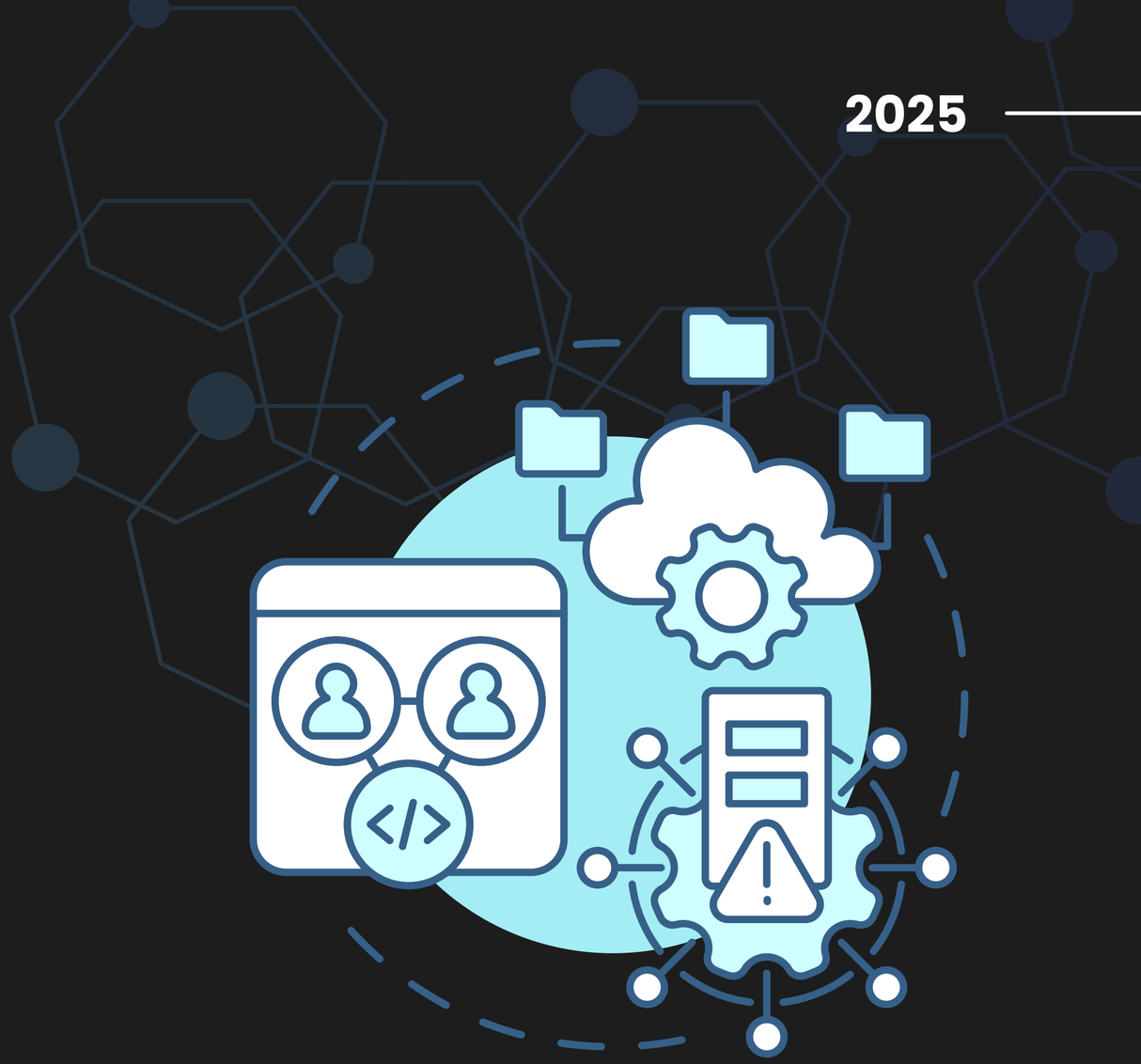
2025

DISTRIBUTED FAULT TOLERANCE

STDISCM S11 GROUP 5

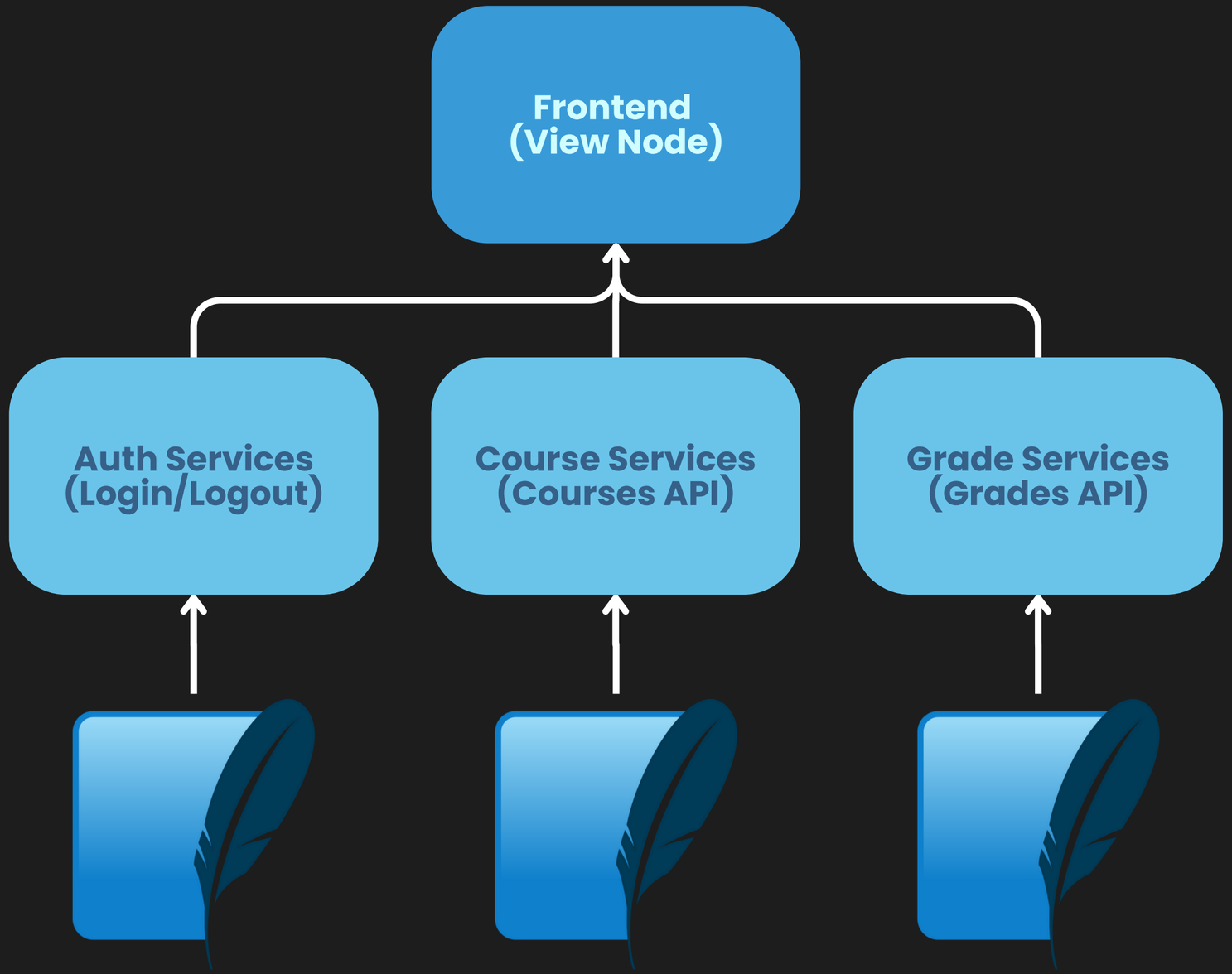
Project Overview

This project is a web-based **online enrollment system** built with a distributed architecture for **fault tolerance**. It uses the MVC pattern, with the View on a dedicated node and Controllers/APIs on separate nodes. Services run on networked virtual machines or bare-metal nodes, ensuring that a **node failure only impacts its specific function**, while the rest of the system remains operational.



Project Setup

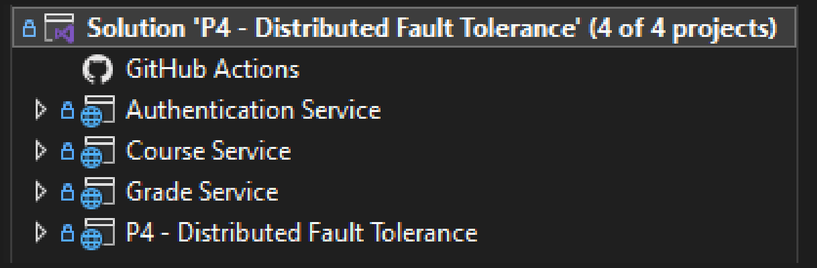
- 3 separate API services
  - Authentication Service
  - Courses Service
  - Grades Service
- 1 view node (frontend)
- 1 database per node
  - Use API calls to access data from other DB



Key Implementation Steps

Creation of 3 ASP.Net Core Web API and 1 ASP.Net Core Web App projects

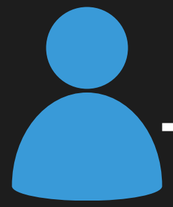
- Authentication Service – handles authentication, JWT, user sessions
- Course Service – handles course listing and enrollment
- Grade Service – handles grade viewing & uploading
- P4 - Distributed Fault Tolerance (frontend) – MVC Web App for the user interface



Roles

student →

- View available courses
- Enroll to courses
- View previous grades



teacher →

- View the grades of all their students
- Upload grades

Key Implementation Steps

Auth Service Implementation (JWT)

- JWT Token Generation
  - Token includes user ID, email, and roles as claims
  - Signed using symmetric key; expires in 1 hour

```
private async Task<string> GenerateJwtTokenAsync(ApplicationUser user)
{
    var jwtSettings = _config.GetSection("Jwt");
    var key = Encoding.UTF8.GetBytes(jwtSettings["Key"]!);

    var claims = new List<Claim>
    {
        new(ClaimTypes.NameIdentifier, user.Id),
        new(ClaimTypes.Email, user.Email),
        new(JwtRegisteredClaimNames.Jti, Guid.NewGuid().ToString())
    };

    var userRoles = await _userManager.GetRolesAsync(user);
    foreach (var role in userRoles)
    {
        claims.Add(new Claim(ClaimTypes.Role, role));
    }

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

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

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

Key Implementation Steps

Course Services and Grades Services Implementation

- Attribute Routing
  - Uses **[HttpGet], [HttpPost], [Route("...")]** to define accessible API routes.
- Data Access
  - Interacts with a centralized SQLite database using **DbContext** and strongly typed **models**.
- Error Handling
  - Checks for invalid inputs (e.g., course not found, duplicate enrollments) and returns proper HTTP status codes.

```
[HttpPost]
[Route("enrollStudent")]
0 references
public async Task<IActionResult> EnrollStudent([FromBody] EnrollRequest enrollRequest)
{
    var existingCourse = await _context.Courses
        .FirstOrDefaultAsync(c => c.CourseId == enrollRequest.CourseId);

    if (existingCourse == null)
    {
        return NotFound(new { message = "Course not found!" });
    }

    if (existingCourse.Students.Count >= existingCourse.Capacity)
    {
        return BadRequest(new { message = "Course is full!" });
    }

    if (existingCourse.Students.Contains(enrollRequest.IdNumber))
    {
        return BadRequest(new { message = "Student already enrolled!" });
    }

    existingCourse.Students.Add(enrollRequest.IdNumber);
    _context.Courses.Update(existingCourse);

    await _context.SaveChangesAsync();
    return Ok(new { message = "Student enrolled successfully!" });
}
```

Key Implementation Steps

Frontend MVC App (View Node)

- Created controllers that make HTTP calls to APIs
- Created Razor views

- View rendering process
  - Use HttpClient to call the respective backend service
  - Pass the result to the corresponding Razor view as a ViewModel
  - Render the HTML with dynamic data (course list, grades, etc.)

```
private async Task<List<Course>> GetCoursesAsync()
{
    List<Course> courses = new List<Course>();

    var token = GetUserAccessToken();
    if (string.IsNullOrEmpty(token))
    {
        Console.WriteLine("API Access Token is missing.");
        return courses;
    }
    _courseClient.DefaultRequestHeaders.Authorization =
        new AuthenticationHeaderValue("Bearer", token);

    try
    {
        HttpResponseMessage response = await _courseClient.GetAsync("getCourses");
        if (response.IsSuccessStatusCode)
        {
            var jsonData = await response.Content.ReadAsStringAsync();
            courses = JsonConvert.DeserializeObject<List<Course>>(isonData);
        }
    }
    catch (Exception ex)
    {
        ModelState.AddModelError("", "The course service is down. Please try again later.");
    }
    return courses;
}
```

How Fault Tolerance is Achieved

Service Isolation on Separate Nodes

- Each API service (Authentication, Course, Grade) is hosted independently
- Fault in one service does not bring down the entire system.
  - If the Grade Service fails, login, course viewing, and enrollment still work.
  - If the Auth Service goes down, existing sessions can remain active (JWT), and students can still browse courses if they're already authenticated.
    - Reduces dependency on a single point of failure

Fail-Fast

- It checks for known error messages in the API response right after the call.
  - If a failure pattern is detected (such as a node being down), it stops further processing and reports the error.