STDISCM

P4 - DISTRIBUTED FAULT TOLERANCE

GitHub: https://github.com/Fluffy-Neko/STDISCM-Problem-Set-4.git

Dane Chan, Raphael Tan Ai, Bruce Mercado, Christian Castillo

KEY IMPLEMENTATION STEPS

- Setting Up the file structure of the Enrollment System Codebase
- Coding the Views and UI for each features
  - Login/logout.
  - View available courses.
  - Students to enroll to open courses.
  - Students to view previous grades.
  - Faculty to be able to upload grades.
- Designing the Database and Data Models
- Creating the controllers that handle each features' logic
- Adding authentication (JWT)

SEPARATION OF RESPONSIBILITIES

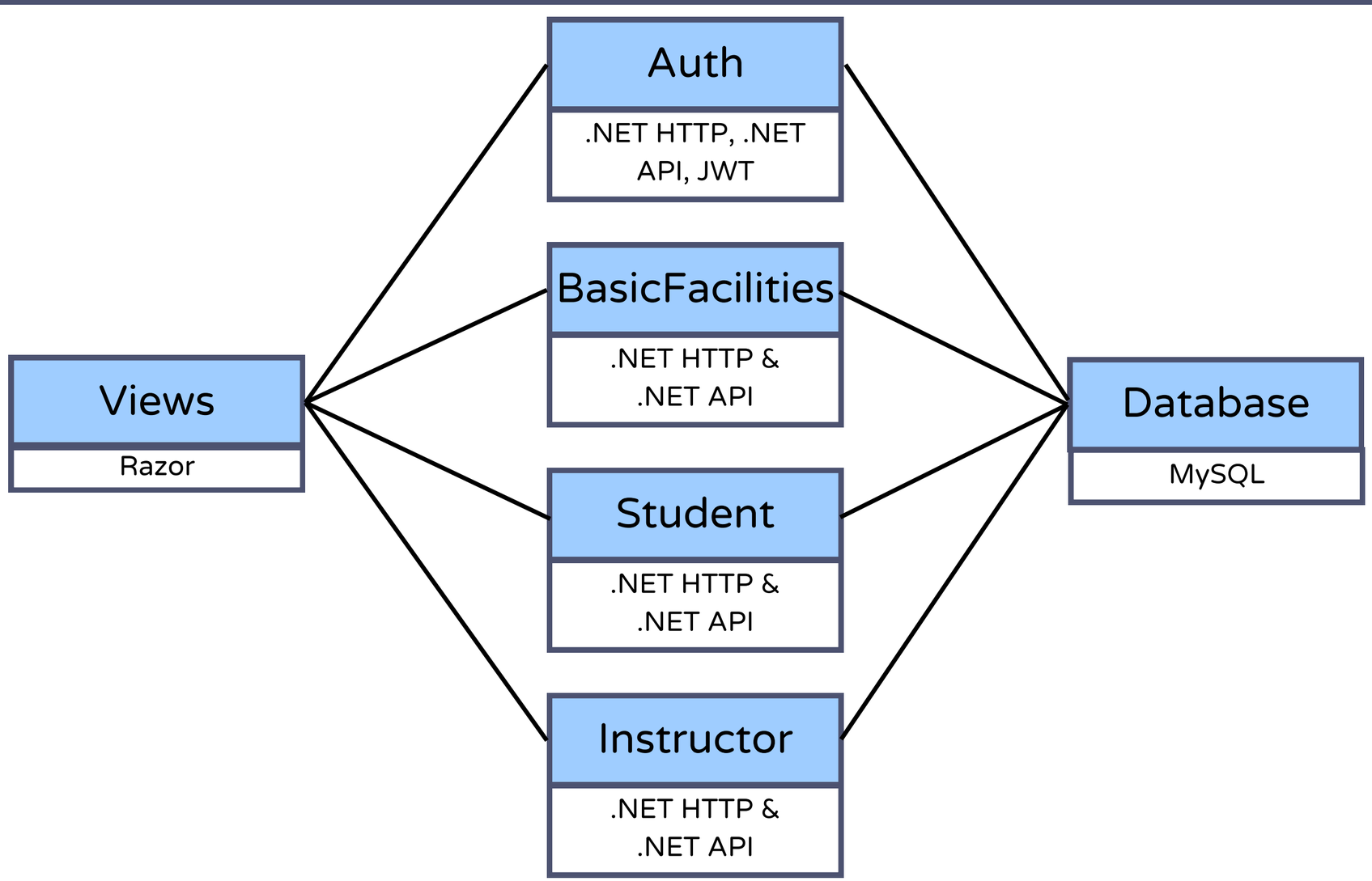
Student

- Can view courses
- Can enroll into courses
- Can see grades in enrolled courses

Instructor

- Can view courses
- Can grade students
- Can view student lists of courses

NODE ARCHITECTURE



NODE RESPONSIBILITIES

Node Component	Component Description
Controllers	Main backend logic of the node, typically handles API requests and methods, often interacts with database.
Models	Models for querying database as well as a wrapper for JSON/structs. Nodes only contain needed models.
Services	In this case, the JWT Token Service is the only service used, and is only used in Auth Node.
Views	Frontend (.cshtml) files for the GUI, only found in the Views Node.
Program.cs	Defines JWT authentication, controller routings, port number(s).

NODE RESPONSIBILITIES

Views	Handles frontend logic, API calls to other nodes, and routing; does not interact with the database.
Auth	Handles authentication logic, queries db for user information and then generates JWT token if successful.
BasicFacilities	Handles functionalities that can be performed by both student and instructor, validates based on JWT.
Instructor	Handles functionalities only usable by instructors, validates based on JWT.
Student	Handles functionalities only usable by students, validates based on JWT.
Database	Handles the database, hosted on MySQL workbench in its own node.

NODE RESPONSIBILITIES

Auth Node (Controller)

Queries database using login credentials.

```
var user = await _context.Users
    .FirstOrDefaultAsync(u => u.Id == loginRequest.Id && u.Password == loginRequest.Password);
```

Returns OK status and generated JWT token when successful.

JWT token is passed by ViewsNode to other nodes that need authentication.

```
if (user != null)
{
    string role;
    if (!user.Role)
    {
        role = "student";
    }
    else
    {
        role = "instructor";
    }

    var token = _tokenService.GenerateJwtToken(user.Id, role);

    return Ok(new
    {
        token,
        userId = user.Id,
        username = user.Username,
        role = user.Role ? "instructor" : "student"
    });
}
```

Also has logout, which clears session data.

NODE RESPONSIBILITIES

Auth Node (JWT)

Defines access token with parameters such as credentials, expiry period, etc.

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

var token = new JwtSecurityToken(
    issuer: _configuration["JwtSettings:Issuer"],
    audience: _configuration["JwtSettings:Audience"],
    claims: claims,
    expires: DateTime.Now.AddDays(1),
    signingCredentials: creds
);
```

Defines access token credentials.

```
"Jwt": {
  "SecretKey": "ThisIsTheSuperSecretKeyOfLilyThatIsOnlyHereForTestingPurposes",
  "Issuer": "http://localhost:5001",
  "Audience": "http://localhost:5002"
},
```

NODE RESPONSIBILITIES

Views Node (Auth Node API)

RESTful API: POST request for logging in.

Calls and waits for response from auth node.

Extracts and stores returned JWT from auth node in cookie.

Redirects to Courses/Index when successful.

```
// POST: /Home/Login
[HttpPost]
0 references
public async Task<IActionResult> Login(UserAuthModel loginRequest)
{
```

```
var response = await client.PostAsync("home/login", content);
```

```
// Parse the JWT token from the response
var jsonResponse = await response.Content.ReadAsStringAsync();
var responseObject = JsonSerializer.Deserialize<JsonElement>(jsonResponse);
var token = responseObject.GetProperty("token").GetString();

// Store the token in the session or a cookie
HttpContext.Session.SetString("JwtToken", token);
```

```
return RedirectToAction("Index", "Courses");
```

NODE RESPONSIBILITIES

Basic Facilities Node (Get Courses)

GET method for retrieving course data.

Authenticates JWT first before proceeding.

Queried courses different for Students and Instructors.

Returns relevant information for courses.

```
// GET: /Courses/Index
[Authorize]
[HttpGet]
0 references
public async Task<IActionResult> GetCourses()
{
    var userIdString = User.FindFirstValue(ClaimTypes.NameIdentifier);
    var role = User.FindFirstValue(ClaimTypes.Role);

    if (!int.TryParse(userIdString, out int userId) || string.IsNullOrEmpty(role))
        return BadRequest("Missing or invalid userId or role");

    var viewModel = new CourseListViewModel();
```

```
return new CourseViewModel
{
    Id = course.Id,
    CourseCode = course.CourseCode,
    Units = course.Units,
    Capacity = course.Capacity,
    SlotsTaken = courseEnrollments.Count(),
    IsEnrolled = enrollments.Any(e => e.CourseId == course.Id),
    Instructor = instructor?.Username ?? "Unknown",
};
```

NODE RESPONSIBILITIES

StudentNode (View Grades)

GET method for querying grades.

Filters grades based on a predefined list of valid grades.

Queries enrollments table for userId to retrieve grades.

Returns enrollment model.

```
// GET: api/Student/Grades
[HttpGet("Grades")]
0 references
public async Task<IActionResult> GetGrades()
{
    var userIdStr = User.FindFirstValue(ClaimTypes.NameIdentifier);
    if (!int.TryParse(userIdStr, out int userId))
        return Unauthorized("Invalid or missing user ID");

    var validGrades = new List<string> { "0.0", "1.0", "1.5", "2.0", "2.5", "3.0", "3.5", "4.0" };

    var grades = await _context.Enrollments
        .Where(e => e.StudentId == userId && validGrades.Contains(e.Grade))
        .ToListAsync();

    return Ok(grades);
}
```

NODE RESPONSIBILITIES

StudentNode (Enroll)

POST method for enrolling a student for a course.

Authenticates JWT and other user information first.

Makes new EnrollmentModel, defaults grade as “NGA” and adds a new enrollment table entry in the database.

Returns status OK if successful.

```
// POST: api/Student/Enroll
[HttpPost("Enroll")]
0 references
public async Task<IActionResult> Enroll([FromBody] EnrollmentRequestModel data)
{
    if (data == null || data.CourseId <= 0)
        return BadRequest("Invalid request payload.");

    var userIdStr = User.FindFirstValue(ClaimTypes.NameIdentifier);
    if (!int.TryParse(userIdStr, out int userId))
        return Unauthorized("Invalid or missing user ID");

    bool alreadyEnrolled = await _context.Enrollments
        .AnyAsync(e => e.StudentId == userId && e.CourseId == data.CourseId);

    if (alreadyEnrolled)
        return BadRequest("You're already enrolled in this course.");

    var course = await _context.Courses.FindAsync(data.CourseId);
    if (course == null)
        return NotFound("Course not found.");

var enrollment = new EnrollmentModel
{
    StudentId = userId,
    CourseId = data.CourseId,
    CourseCode = course.CourseCode,
    Grade = "NGA"
};

_context.Enrollments.Add(enrollment);
await _context.SaveChangesAsync();

return Ok("Successfully enrolled.");
```

NODE RESPONSIBILITIES

InstructorNode (Get Course Details)

GET method for viewing course details.

Validates user ID and role from session first.

Note: Grade defaults to NGA if the grade in db is invalid.

Queries relevant information (course and student info) and returns them along with an OK response if successful.

```
[HttpGet("View")]
0 references
public async Task<IActionResult> GetCourseDetails(int id)
{
    var userIdStr = User.FindFirstValue(ClaimTypes.NameIdentifier);
    if (!int.TryParse(userIdStr, out var userId))
        return Unauthorized("Invalid or missing user ID");

    var course = await _context.Courses.FirstOrDefaultAsync(c => c.Id == id && c.InstructorId == userId);
    if (course == null)
        return NotFound("Course not found or you are not the instructor.");

var studentViewModels = students.Select(s => new UserViewModel
{
    Id = s.Id,
    Username = s.Username,
    Grade = enrollments.FirstOrDefault(e => e.StudentId == s.Id)?.Grade ?? "NGA"
}).ToList();

var viewModel = new CourseViewModel
{
    Id = course.Id,
    CourseCode = course.CourseCode,
    Units = course.Units,
    Capacity = course.Capacity,
    SlotsTaken = enrollments.Count,
    Instructor = instructor?.Username ?? "Unknown",
    Students = studentViewModels
};

return Ok(viewModel);
```

NODE RESPONSIBILITIES

InstructorNode (Update Grades)

POST method to update a student's grade.

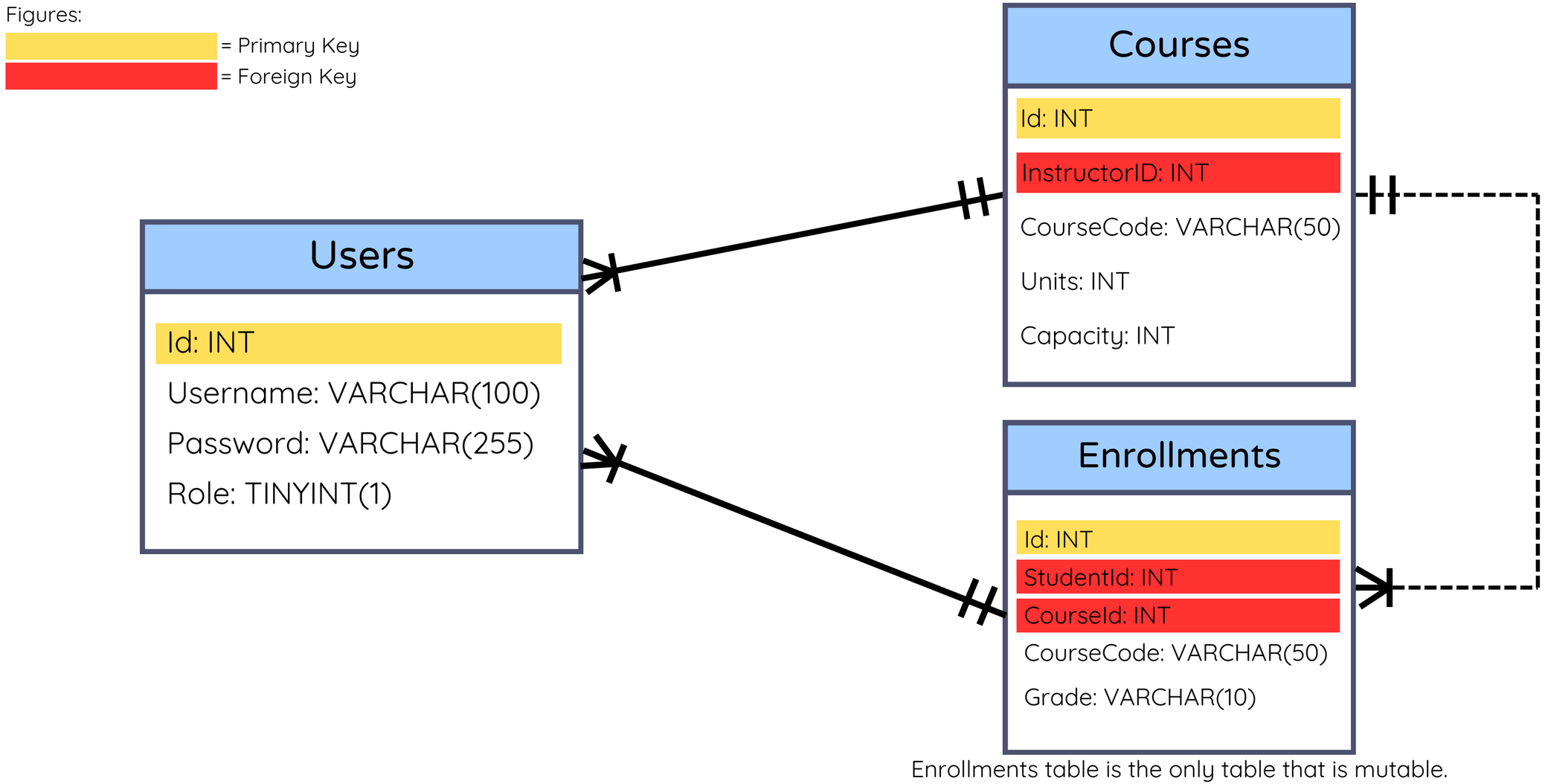
Authenticate with session info first.

Filter grade changes to valid grades only.

Makes changes to enrollment table in the db and returns status OK if successful.

```
[HttpPost("UpdateGrade")]
0 references
public async Task<IActionResult> UpdateGrade([FromBody] GradeUpdateRequest model)
{
    var userIdStr = User.FindFirstValue(ClaimTypes.NameIdentifier);
    if (!int.TryParse(userIdStr, out var userId))
        return Unauthorized("Invalid or missing user ID");

    var userRole = User.FindFirstValue(ClaimTypes.Role);
    if (userRole != "instructor")
        return Forbid("Only instructors can update grades.");

var validGrades = new List<string> { "NGA", "0.0", "1.0", "1.5", "2.0", "2.5", "3.0", "3.5", "4.0" };
if (!validGrades.Contains(model.Grade))
    return BadRequest("Invalid grade value.");

enrollment.Grade = model.Grade;
await _context.SaveChangesAsync();

return Ok("Grade updated successfully.");
```

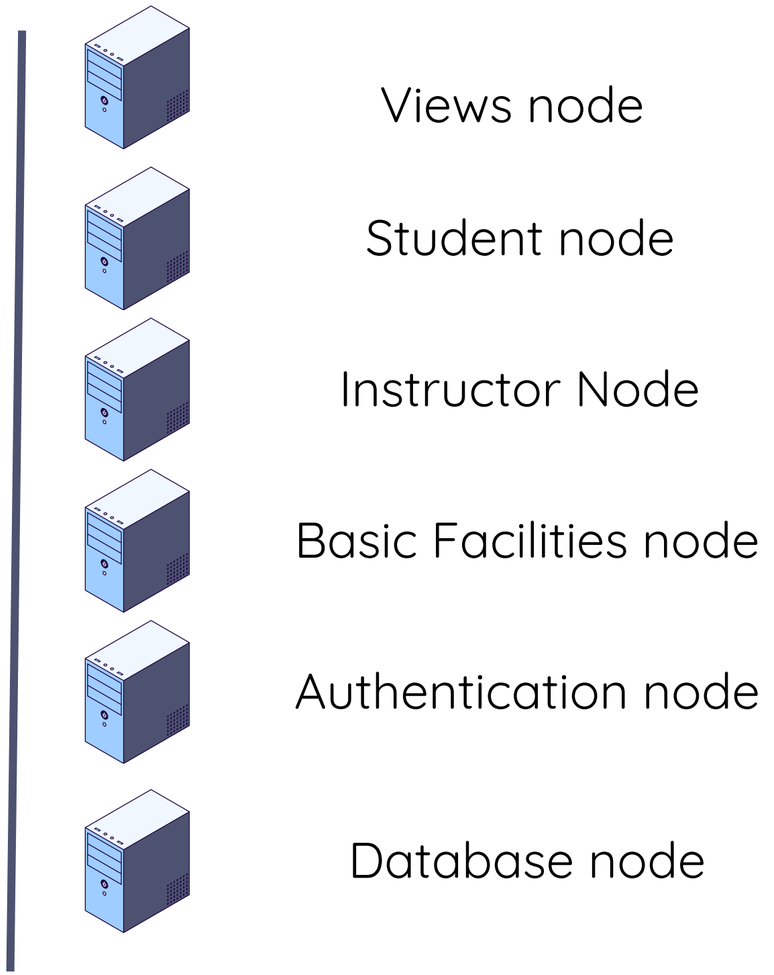
DATABASE SCHEMA



Enrollments table is the only table that is mutable.

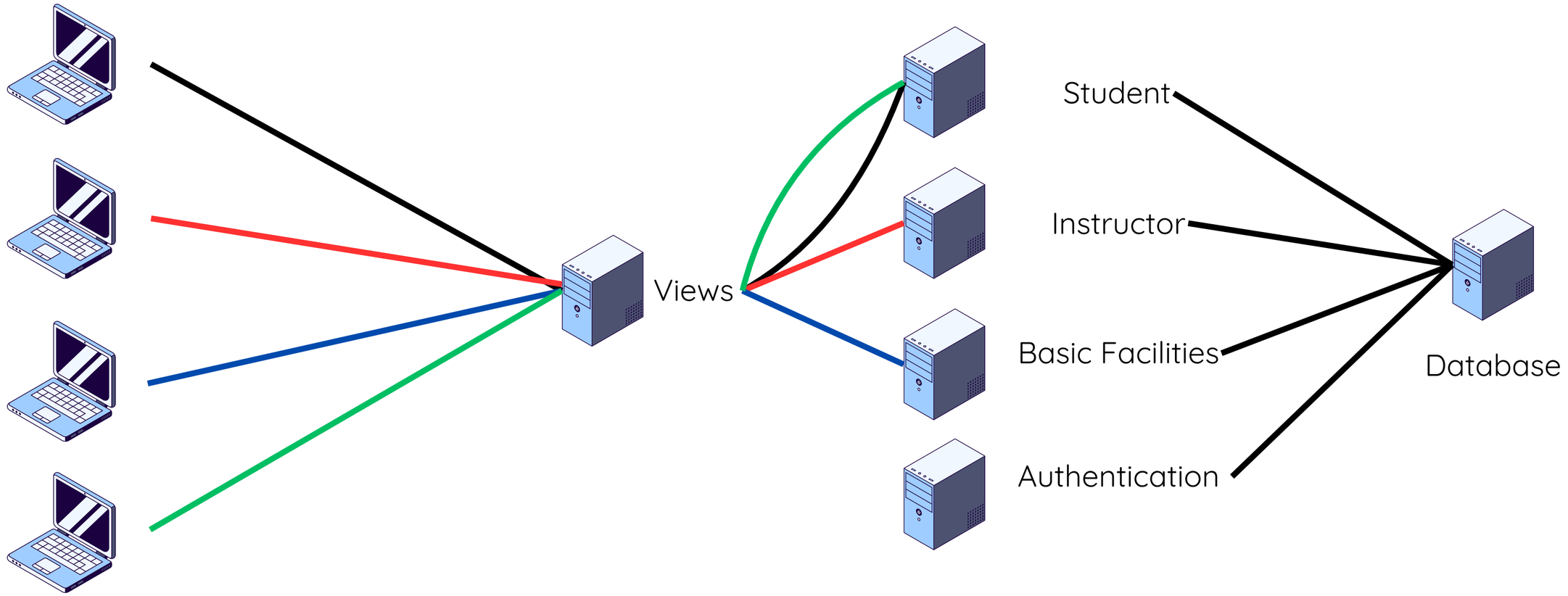
HOW FAULT TOLERANCE IS ACHIEVED

- The Views is in a separate node from the features
- The features accessible to Students are in a separate node from the features accessible to the Instructors.
- Authentication is handled in a separate node



- Multiple users can access different features of the distributed system
- With this setup, if one feature node is down, the other features are still accessible

---



- Multiple users can access different features of the distributed system
- With this setup, if one feature node is down, the other features are still accessible

---