Architectural Onboarding Report: The Tutorz Ecosystem

1. Executive Summary and Architectural Philosophy

1.1 The Imperative of Structural Integrity in Software Engineering

In the realm of modern software development, particularly within the context of the.NET ecosystem, the initial phase of any project often presents a deceptive simplicity. When a developer creates a new solution, the path of least resistance suggests a direct, linear approach: a user interface that speaks directly to a database. For a prototype or a trivial utility, such as a calculator or a simple to-do list, this monolithic, entangled approach—often derisively termed "spaghetti code"—may suffice. However, for an enterprise-grade platform like **Tutorz**, which is tasked with managing the complex, multi-faceted interactions of the Sri Lankan tuition industry, such an approach would lead to catastrophic technical debt.[1]

The Tutorz platform is not merely a data storage system; it is a dynamic ecosystem connecting Tutors, Students, Educational Institutes, and Administrators. It must handle intricate business rules ranging from custom student identification generation (e.g., "STU256800084") to complex fee commission calculations involving multiple stakeholders.[1] As new engineering personnel integrate into the Tutorz development team, understanding the architectural decisions that underpin this system is not optional—it is fundamental to contributing code that survives the test of time, scalability, and maintainability.

The chosen architectural pattern for Tutorz is **Clean Architecture**, a methodology championed by Robert C. Martin (Uncle Bob). This architecture is often visualized as an onion, consisting of concentric circles where the fundamental rule is that dependencies must only flow inward.[3] The core of the application—the business rules and domain entities—must remain pristine and unaffected by the chaotic, ever-changing external world of databases, user interfaces, and third-party frameworks. This report provides an exhaustive, "basic-to-pro" analysis of how Clean Architecture is implemented in the Tutorz.NET project, serving as the definitive onboarding manual for interns and junior developers.[5]

1.2 The Core Philosophy: Separation of Concerns

The primary objective of the Tutorz architecture is the **Separation of Concerns**. In a poorly architected system, a change in the user interface (e.g., switching from a web page to a mobile app) or a change in the database (e.g., switching from SQL Server to PostgreSQL) would require rewriting the business logic. This is unacceptable.

In the Tutorz solution, the codebase is physically separated into four distinct projects, each

representing a specific layer of the architectural "onion" [1]:

1. **Tutorz.Domain**: The Enterprise Business Rules. This layer contains the fundamental truths of the system—Entities like Student and Tutor. It has zero dependencies on any other project. It is the stable core.[7]
2. **Tutorz.Application**: The Application Business Rules. This layer defines "Use Cases"—what the system *does*. It handles registration, login, and ID generation. It depends only on the Domain.[8]
3. **Tutorz.Infrastructure**: The Interface Adapters and Frameworks. This layer handles the "plumbing"—database connections (Entity Framework Core), email sending (SMTP), and file storage. It depends on the Application and Domain layers.[9]
4. **Tutorz.Api**: The Presentation Layer. This is the entry point for the outside world (React frontend, mobile apps). It receives HTTP requests and returns JSON responses. It depends on the Application and Infrastructure layers (strictly for dependency injection).[10]

1.3 The Restaurant Analogy: A Conceptual Framework

To visualize these layers for a newcomer, one might employ the analogy of a high-end restaurant.[11]

Tutorz Layer	Restaurant Analogy	Description
Domain	**The Recipe**	The recipe for "Spicy Chicken Curry" is an abstract truth. It defines the ingredients and the steps. It does not care who cooks it, where the chicken was bought, or whether it is served on fine china or a paper plate. In Tutorz, the User entity is a recipe; it defines what a user is, regardless of the database.
Application	**The Chef**	The chef executes the recipe. They receive an order, coordinate the preparation, and ensure the rules (recipe) are followed. They do not grow the vegetables

		(Infrastructure) nor serve the table (Presentation). In Tutorz, the AuthService is the chef, orchestrating the login process.
Infrastructure	**The Supplier & Pantry**	The chef needs ingredients. The infrastructure layer provides them. The chef says, "I need eggs," and the infrastructure layer retrieves them from the fridge. In Tutorz, the UserRepository retrieves user data from the SQL database.
Presentation	**The Waiter**	The waiter takes the customer's request and brings the food. They do not cook. If the customer asks for a modification, the waiter relays it to the chef. In Tutorz, the AuthController receives the JSON request and passes it to the AuthService.

1.4 The Dependency Rule: The Golden Law

The most critical takeaway for any intern joining the project is the **Dependency Rule**: *Source code dependencies must always point inward*.[2]

The Domain project does not know that the Application project exists. The Application project does not know that the Infrastructure project exists (it only knows interfaces). The Infrastructure and Api projects depend on the inner layers. This ensures that the volatile outer layers (UI, Database) can change without affecting the stable inner layers (Business Logic). If an engineer attempts to reference the Tutorz.Infrastructure project from within Tutorz.Domain, they have violated the fundamental integrity of the architecture.[4]

2. The Domain Layer: The Heart of Tutorz

2.1 Defining the Enterprise Truth

The Tutorz.Domain project represents the "Platonic Ideal" of the Tutorz system. It contains the C# classes that define the concepts of Tutors, Students, and Institutes. This project references **nothing**. It is pure C#.[1]

2.1.1 The User Entity: The Root of Identity

At the center of the Tutorz domain is the User entity, found in Tutorz.Domain/Entities/User.cs. This class represents the fundamental identity of any actor in the system.

C#

```
public class User
{
    public Guid UserId { get; set; }
    public string RegistrationNumber { get; set; }
    public string Email { get; set; }
    public string PasswordHash { get; set; }
    public Guid RoleId { get; set; }
    //... audit fields like CreatedDate
}
```

Architectural Insight: Note that the User class is "anemic" in the current implementation—it contains properties but little behavior (methods). In a purely "Rich Domain Model" (a "Pro" level concept), this class would contain methods like ChangePassword() or UpdateEmail() to enforce rules internally.[14] However, the current structure separates state (Entities) from logic (Services), which is common in.NET architectures relying on Entity Framework Core.[15]

The RegistrationNumber property is critical. Unlike the UserId (a GUID, which is machine-friendly), the RegistrationNumber is human-friendly (e.g., STU-25-001). This reflects a specific business requirement from the SRS to allow searching institutes or students by number.[1]

2.1.2 Role-Specific Entities: Polymorphism via Composition

Rather than using inheritance (e.g., class Student : User), the Tutorz architecture uses composition. Specific profile data is stored in separate entities that link back to the User.

- **Student.cs**: Contains SchoolName, Grade, ParentName, DateOfBirth.
- **Tutor.cs**: Contains Bio, ExperienceYears, BankAccountNumber, BankName.
- **Institute.cs**: Contains InstituteName, Address, ContactNumber.[1]

This separation is vital for database normalization. A Tutor needs a bank account to receive payments; a Student does not. If all these fields were shoved into a single User table, the database would be riddled with null values. By separating them, the architecture enforces strict typing and cleaner data structures.

2.1.3 Value Objects and Enums

The Domain layer also contains **Value Objects** and **Enums**.

- **Enums (UserRole.cs, PaymentStatus.cs)**: These define the rigid constants of the system. UserRole likely contains Admin, Tutor, Student, Institute. Using enums prevents "magic strings" in the code, ensuring that a developer cannot accidentally type "Techer" instead of "Tutor".[1]
- **Value Objects (Money.cs)**: While the snippet listing shows Money.cs, its internal implementation is crucial. A Value Object differs from an Entity in that it has no identity. Five dollars is always five dollars. If Money is implemented correctly, it prevents arithmetic errors (e.g., adding 5 USD to 5 LKR) by encapsulating currency logic.[16]

2.2 The Anemic vs. Rich Domain Debate

New engineers should be aware of the "Anemic Domain Model" anti-pattern controversy. The current Tutorz codebase places most logic in AuthService (Application Layer) rather than inside the User class (Domain Layer). This is often called an "Anemic" model because the entities are just bags of getters and setters.[14]

Pro-Level Analysis:

- **Current State (Anemic):** Easy to map to database tables using Entity Framework. Great for simple CRUD (Create, Read, Update, Delete) apps.
- **Ideal State (Rich):** Business rules like "A student cannot be in Grade 13 if they are under 16 years old" should live *inside* the Student class. This ensures the rule is enforced everywhere, not just in one service. As the Tutorz project matures, refactoring towards a Rich Domain Model is a recommended path for long-term maintainability.[18]

---

3. The Application Layer: Orchestration and Use Cases

The Tutorz.Application project serves as the brain of the operation. It dictates *how* the entities interact to fulfill user requirements. It depends on Tutorz.Domain but has no knowledge of the database or the web API.[1]

3.1 The DTO (Data Transfer Object) Pattern

One of the first things an intern will notice is the duplication of data structures. We have a User entity in the Domain, but we also have LoginRequest, RegisterRequest, and AuthResponse classes in Tutorz.Application/DTOs/.[1]

Why do we do this?
This is a deliberate architectural pattern to separate the internal data model from the external API contract.19

1. **Security:** The User entity contains PasswordHash. If we returned the User object directly to the mobile app, we would be leaking sensitive security credentials. The AuthResponse DTO contains only safe fields: UserId, Email, Role, and Token.
2. **Versioning:** If we decide to rename the PhoneNumber column in the database to MobileNo, we can change the Entity. However, we can keep the DTO property as PhoneNumber, ensuring the mobile app doesn't break. The Application layer handles the mapping between the two.
3. **Validation:** DTOs are the perfect place to enforce input formatting rules (e.g., [EmailAddress], ``) without polluting the pure Domain entities with validation logic.[21]

3.2 The Service Layer: AuthService

The AuthService.cs file is the most critical component provided in the snippets. It acts as the orchestrator for the authentication use cases.[1] Let's analyze its workflow step-by-step.

3.2.1 Registration Workflow (RegisterAsync)

When a user registers, the AuthService performs a complex ballet of operations:

1. **Validation:** It first checks basic constraints.
    - if (request.Password.Length < 6): Enforces password strength.
    - Regex.IsMatch(request.PhoneNumber, @"^07\d{8}$"): Enforces Sri Lankan mobile number formats.
    - *Insight:* This logic currently resides in the Service. In a "Pro" refactoring, this should be moved to a FluentValidation validator to keep the service cleaner.[22]
2. **Uniqueness Check:** It calls _userRepository.GetAsync to ensure the email and phone number are not already taken.
3. **ID Generation:** This is a key business rule. It calls _idGeneratorService.GenerateNextIdAsync.
4. **Entity Construction:** It creates a User entity and hashes the password using BCrypt.
5. **Role Handling:** Based on the string role ("Tutor", "Student"), it creates the corresponding specific entity (Tutor or Student) and saves it.

Architectural Critique: The AuthService is currently coupled to multiple repositories (_tutorRepository, _studentRepository, _instituteRepository). This is a variation of the **Facade Pattern**, where the service hides the complexity of dealing with multiple tables from the

controller.[23]

3.2.2 The Custom ID Logic (IdGeneratorService)

The requirement for custom IDs (e.g., STU-25-6-00084) is handled by IdGeneratorService.cs.[1]

Mechanism:

1. It determines a prefix based on the role (STU, TUT, INS).
2. It calculates the current context (Year 25, Month 6).
3. For students, it extracts the class number (e.g., "Grade 10" -> 10).
4. It constructs a key: STU-25-6-10.
5. It calls _sequenceRepository.GetNextSequenceNumberAsync(key) to get the next incremental number (e.g., 84).
6. It formats the final string: STU256800084.

Data Flow Insight: This demonstrates how business logic (formatting the string) is separated from data access (getting the next number). The IdGeneratorService doesn't know *how* the number is stored; it just asks the repository for it.

3.3 Dependency Inversion via Interfaces

The Application layer defines **Interfaces** for everything it needs but cannot implement itself.

- IUserRepository: "I need to save a user."
- IEmailService: "I need to send an email."
- IIdGeneratorService: "I need a new ID."

These interfaces (Tutorz.Application/Interfaces/) are the "ports" in the Hexagonal Architecture concept. They allow the Application layer to remain independent. The actual implementation (the "adapters") resides in the Infrastructure layer.[24]

---

4. The Infrastructure Layer: Plumbing and Persistence

The Tutorz.Infrastructure layer is where the abstract concepts of the Application layer meet the concrete reality of servers and databases.

4.1 Entity Framework Core and the DbContext

The project uses **Entity Framework Core (EF Core)** as its Object-Relational Mapper (ORM). The TutorzDbContext class (Tutorz.Infrastructure/Data/TutorzDbContext.cs) acts as the bridge between C# objects and SQL Server tables.[25]

C#

```
public class TutorzDbContext : DbContext
{
    public DbSet<User> Users { get; set; }
    public DbSet<Role> Roles { get; set; }
    //...
}
```

Configuration:
The OnModelCreating method is used to configure database schema rules using the Fluent API.

- modelBuilder.Entity<User>().HasIndex(u => u.RegistrationNumber).IsUnique();
  This line is critical. It enforces data integrity at the database engine level. Even if the application logic fails and tries to insert a duplicate ID, SQL Server will reject it. This is a "defense in depth" strategy.1

4.2 The Repository Pattern: Implementation and Controversy

Tutorz implements the **Repository Pattern** using a generic base class (GenericRepository<T>) and specific implementations (UserRepository, StudentRepository).[26]

4.2.1 The Generic Repository

The GenericRepository<T> implements common operations (GetAsync, AddAsync) to avoid code duplication.

- *Intern Question:* "Why not just use the DbContext directly in the Service?"
- *Answer:* Using a Repository creates an abstraction. If we later decide to switch from EF Core to Dapper (a faster micro-ORM) for performance, or to a Mock for unit testing, we only have to change the Repository implementation. The Service code remains untouched.[27]

4.2.2 The Specific Repository (UserRepository)

While the generic repository handles basics, the UserRepository would handle complex, user-specific queries.

- *Data Flow:* AuthService calls _userRepository.AddAsync(user). The repository calls _context.Users.AddAsync(user). No SQL is written; EF Core generates the INSERT statement automatically.

4.2.3 The "Anti-Pattern" Counter-Argument

Advanced engineers might argue that EF Core's DbContext *is* already a repository and DbSet

is a Unit of Work, making the Repository pattern redundant.[29] This is a valid "Pro" level debate.

- *Rebuttal:* In Tutorz, the Repository pattern is still useful because it allows us to inject strict boundaries and enables easier mocking for Unit Tests. It prevents the "leaky abstraction" of EF-specific features (like IQueryable) bleeding into the Service layer.[27]

4.3 Database Migrations

The file 20251013092513_InitialCreate.cs represents a Database Migration.[1]

- **Concept:** In modern.NET, we don't write CREATE TABLE scripts manually. We define the C# Entity (User), and then run a command (dotnet ef migrations add). EF Core compares the C# model to the existing database and generates the C# migration code to bridge the gap.
- **Data Flow:** When the application starts, Program.cs calls DbInitializer.Initialize(context), which ensures the database is created and seeded with default roles.[1]

4.4 Handling Concurrency: UserSequenceRepository

The UserSequenceRepository manages the generation of unique IDs.

C#

```
public async Task<int> GetNextSequenceNumberAsync(string prefixKey)
{
    var sequence = await _context.UserSequences.FirstOrDefaultAsync(...);
    //... increment logic...
    await _context.SaveChangesAsync();
    return sequence.LastNumber;
}
```

- **Critical Risk:** If two users register at the *exact* same millisecond, they might both read the same LastNumber (e.g., 83) and both try to update it to 84.
- **Pro Solution:** The current code relies on EF Core's optimistic concurrency (or simple transactional saving). A more robust solution for high-scale systems would be to use a raw SQL query with a lock (UPDATE UserSequences SET LastNumber = LastNumber + 1 OUTPUT INSERTED.LastNumber...) or a distributed lock (Redis) to guarantee uniqueness.[27]

---

5. The Presentation Layer: The API

The Tutorz.Api project is the "face" of the application. It contains the **Controllers** that expose the business logic to the web via HTTP.[7]

5.1 The Controller's Role: Traffic Cop

The AuthController (Tutorz.Api/Controllers/AuthController.cs) is a perfect example of a "Thin Controller."

C#

```
[HttpPost("register")]
public async Task<IActionResult> Register(RegisterRequest request)
{
    var response = await _authService.RegisterAsync(request);
    return Ok(response);
}
```

Architectural Rule: Controllers should contain **zero** business logic. They should:

1. Receive the request.
2. Call the appropriate Service.
3. Return the result (200 OK, 400 Bad Request, 500 Internal Server Error).[31]

If an intern writes if (request.Age < 18) inside a Controller, they have violated the Separation of Concerns. That logic belongs in the Service or a Validator.

5.2 Dependency Injection Configuration (Program.cs)

The Program.cs file is the **Composition Root**. This is where the application is assembled.[32]

C#

```
builder.Services.AddDbContext<TutorzDbContext>(...);
builder.Services.AddScoped<IAuthService, AuthService>();
builder.Services.AddScoped<IUserRepository, UserRepository>();
```

This code configures the **Dependency Injection (DI) Container**. It tells.NET: "Whenever a constructor asks for IAuthService, please create an instance of AuthService and hand it

over".[33]

5.2.1 Service Lifetimes

Understanding the difference between AddScoped, AddTransient, and AddSingleton is mandatory for new developers [34]:

- **Scoped (AddScoped):** The service is created once per HTTP request. All components in that single request share the same instance. This is essential for TutorzDbContext to ensure that a "Unit of Work" (multiple database saves) remains consistent within one request. Tutorz uses this for almost all services.
- **Transient (AddTransient):** A new instance is created *every single time* it is requested. Useful for lightweight, stateless tools.
- **Singleton (AddSingleton):** Created once when the app starts and shared by *every* user. **Danger:** Never use this for Database Contexts, as it would cause threading issues where User A's data overwrites User B's data.

5.3 Global Exception Handling

The snippet shows an ExceptionMiddleware.cs file.[1] While the content is brief, its purpose is vital.

- **Current Anti-Pattern:** The AuthController currently has try-catch blocks in every method.
- **Pro Solution:** The middleware should wrap *all* requests. If an exception is thrown anywhere in the application, the middleware catches it, logs it, and returns a standardized JSON error response. This removes the need for repetitive try-catch blocks in controllers, cleaning up the code significantly.[10]

---

6. Data Flow Narratives: Tracing Execution

To solidify understanding, we will trace the execution path of three critical scenarios using the provided code.

6.1 Scenario A: Student Registration (The "Happy Path")

1. **Frontend:** Sends POST /register with { "role": "Student", "grade": "10",... }.
2. **API (AuthController):** Accepts JSON, deserializes to RegisterRequest. Calls AuthService.
3. **Application (AuthService):**
    - Validates inputs.
    - Calls IdGeneratorService.
        - **Application (IdGeneratorService):** Determines prefix "STU". Calls UserSequenceRepository.
        - **Infrastructure (UserSequenceRepo):** Reads DB, increments counter, returns "84".

        - **Application (IdGeneratorService):** Returns "STU-25-6-00084".
    - Creates User entity (hashed password).
    - Creates Student entity (linked to User).
    - Calls UserRepository.AddAsync.
4. **Infrastructure (UserRepository):** Adds entities to DbContext.
5. **Infrastructure (TutorzDbContext):** commits transaction to SQL Database.
6. **Application (AuthService):** Generates JWT Token. Returns AuthResponse.
7. **API (AuthController):** Returns HTTP 200 with Token.

6.2 Scenario B: Social Login (Google)

1. **Frontend:** User clicks "Sign in with Google" and sends the Google ID Token to POST /social-login.
2. **API:** Calls AuthService.SocialLoginAsync.
3. **Application:**
    - Calls ValidateGoogleAccessTokenAsync.
    - Uses HttpClient to call googleapis.com to verify the token is valid and get the email.
    - Checks if the email exists in the DB.
    - *Case 1 (User Exists):* Generates JWT for the existing user.
    - *Case 2 (New User):* Creates a new User entity (password is empty/null). **Critical Check:** If the frontend didn't send a Role, it throws an exception ("You should register first").
4. **Return:** Sends back the Tutorz JWT (not the Google token) to the frontend.

6.3 Scenario C: Fee Calculation (Future Implementation)

Based on the file FeeCalculation.sql [1] and the SRS [1], this flow represents a hybrid approach.

1. **API:** Admin requests "Calculate Monthly Fees".
2. **Application:** Calls FeeService.
3. **Infrastructure:** Instead of loading 10,000 attendance records into memory (which would be slow), the service calls the Stored Procedure FeeCalculation.
    - *Insight:* This moves the processing logic *to the database*. While this violates "pure" Clean Architecture (putting logic in the DB), it is a pragmatic "Pro" choice for performance in heavy data operations.

---

7. Advanced Concepts and Future Roadmap

As interns transition to junior developers, they must be aware of the "Pro" level concepts that are either missing or could improve the current Tutorz codebase.

7.1 CQRS (Command Query Responsibility Segregation) with MediatR

Currently, the API calls Services directly. A more advanced pattern uses **MediatR**.

- **Concept:** Split operations into **Commands** (Write: Register User) and **Queries** (Read: Get User Profile).[36]
- **Benefit:** The Controller just sends a message (_mediator.Send(new RegisterCommand(...))). It doesn't know who handles it. This decouples the API from the Service layer entirely.

7.2 FluentValidation

The validation logic inside AuthService is "imperative" (if (x < 5) throw...).

- **Improvement:** Use the **FluentValidation** library. Create a RegisterRequestValidator class that defines rules declaratively (RuleFor(x => x.Password).MinimumLength(6)). This removes clutter from the Service.[22]

7.3 Unit Testing and Mocking

The current structure is perfectly set up for Unit Testing.

- **How to Test AuthService:** Since AuthService depends on interfaces (IUserRepository), we can use a library like **Moq** to create a fake repository.
- **Test Scenario:** "When RegisterAsync is called, if UserRepository.GetAsync returns a user, verify that the service throws a 'User already exists' exception."
- This is impossible to do easily if we didn't use Dependency Injection and Repositories.[27]

7.4 Handling the "Anemic" Domain

To move toward a Rich Domain Model, logic should migrate from Services to Entities.

- *Example:* The Student entity should have a method PromoteGrade().
- *Logic:* public void PromoteGrade() { if (this.Grade == "13") throw new Exception("Graduated"); this.Grade = (int.Parse(this.Grade) + 1).ToString(); }
- This ensures that no Service can accidentally set a student's grade to 14 or "Banana". The Entity protects its own integrity.[15]

7.5 Implementation of SRS Features

The SRS outlines features like "Commission Calculation" and "Attendance Reports".[1]

- **Commission:** This requires a new Domain logic. When a Payment is recorded, an event (Domain Event) should be triggered to calculate the split (e.g., 80% to Tutor, 20% to Institute) and update a Ledger entity.
- **Attendance:** This requires high-performance insertion. If a class has 500 students, marking attendance might require a bulk-insert strategy in the Repository rather than a loop of AddAsync.

---

8. Conclusion

The Tutorz project is built upon a robust foundation of **Clean Architecture**. This is not merely a file structure; it is a philosophy of resilience. By separating the **Domain** (the business truth), the **Application** (the orchestration), the **Infrastructure** (the plumbing), and the **Presentation** (the interface), the system protects itself against the inevitable changes in technology and requirements.

For the new engineer, the path to mastery involves:

1. Respecting the **Dependency Rule** above all else.
2. Understanding that **DTOs** protect the API contract.
3. Utilizing **Dependency Injection** to decouple components.
4. Recognizing that **Repositories** abstract the database, facilitating testing and flexibility.

As you write code for Tutorz, ask yourself: "Does this logic belong to the core truth of a Student (Domain), the process of registering (Application), or the act of saving to SQL (Infrastructure)?" Answering this question correctly is the essence of professional software engineering. Welcome to the team.

Appendix A: Detailed Code Reference Map

This appendix maps the concepts discussed to the specific files provided in the codebase [1], serving as a quick reference for navigation.

Concept	Layer	Key File(s)	Functionality
User Identity	Domain	Entities/User.cs	Defines the core user schema and authentication fields.
Role Profiles	Domain	Entities/Student.cs, Tutor.cs	Stores role-specific data linked via UserId.
Input Models	Application	DTOs/Auth/Register Request.cs	Defines the data contract expected from the frontend.

Output Models	Application	DTOs/Auth/AuthResponse.cs	Defines the safe data contract returned to the client.
Business Logic	Application	Services/AuthService.cs	Orchestrates validation, ID generation, hashing, and saving.
ID Logic	Application	Services/IdGeneratorService.cs	Implements the custom STU-YY-MM-XXX string formatting.
Contracts	Application	Interfaces/IUserRepository.cs	Defines *what* persistence operations are available.
Database Map	Infrastructure	Data/TutorzDbContext.cs	Maps C# entities to SQL tables via EF Core.
Data Access	Infrastructure	Repositories/UserRepository.cs	Implements *how* to save data using EF Core.
ID Sequence	Infrastructure	Repositories/UserSequenceRepository.cs	Manages the atomic counter for ID generation in the DB.
API Endpoint	Presentation	Controllers/AuthController.cs	Receives HTTP requests and delegates to AuthService.

Wiring	Presentation	Program.cs	Configures Dependency Injection and Middleware.

Appendix B: Data Flow Diagrams (Textual Description)

B.1 Student Registration Flow

1. **Frontend** $\rightarrow$ POST /register JSON payload.
2. **AuthController** $\rightarrow$ Deserializes to RegisterRequest DTO.
3. **AuthController** $\rightarrow$ Calls IAuthService.RegisterAsync(dto).
4. **AuthService** $\rightarrow$ Validates constraints (password length, regex).
5. **AuthService** $\rightarrow$ Calls IIdGeneratorService.GenerateNextIdAsync.
6. **IdGeneratorService** $\rightarrow$ Calls IUserSequenceRepository.GetNextSequenceNumberAsync.
7. **UserSequenceRepository** $\rightarrow$ **SQL DB** (Increments/Locks Sequence Table) $\rightarrow$ Returns Integer (e.g., 55).
8. **IdGeneratorService** $\rightarrow$ Formats string "STU250600055" $\rightarrow$ Returns to Service.
9. **AuthService** $\rightarrow$ Creates User and Student Entities. Hashes Password.
10. **AuthService** $\rightarrow$ Calls IUserRepository.AddAsync & IStudentRepository.AddAsync.
11. **UserRepository** $\rightarrow$ DbContext.Add $\rightarrow$ DbContext.SaveChanges.
12. **EF Core** $\rightarrow$ **SQL DB** (INSERT INTO Users; INSERT INTO Students).
13. **AuthService** $\rightarrow$ Generates JWT. Creates AuthResponse.
14. **AuthController** $\rightarrow$ Serializes AuthResponse to JSON $\rightarrow$ Returns HTTP 200.

B.2 Login Flow

1. **Frontend** $\rightarrow$ POST /login (Identifier, Password).
2. **AuthController** $\rightarrow$ Calls IAuthService.LoginAsync.
3. **AuthService** $\rightarrow$ Detects if Identifier is Email or Phone.
4. **AuthService** $\rightarrow$ Calls IUserRepository.GetAsync (Find User).
5. **AuthService** $\rightarrow$ Checks BCrypt.Verify(password, hash).
6. **AuthService** $\rightarrow$ Calls IRoleRepository (Get Role Name).
7. **AuthService** $\rightarrow$ Generates JWT with Claims (ID, Email, Role).
8. **AuthService** $\rightarrow$ Returns AuthResponse (Token).
9. **AuthController** $\rightarrow$ Returns HTTP 200.

Works cited

1. Tutor.pdf
2. Clean Architecture in .NET: A Practical Guide with Examples | by Roshika Nayanadhara, accessed December 22, 2025, https://medium.com/@roshikanayanadhara/clean-architecture-in-net-a-practical-guide-with-examples-817568b3f42e
3. Clean Architecture in .NET: A Beginner's Guide | by Himanshu Chopra (.Net Guy) | Medium, accessed December 22, 2025, https://medium.com/@himanshuchopra81/clean-architecture-in-net-a-beginners-guide-5487383c557a
4. The Clean Architecture I Wish Someone Had Explained to Me | by Rafael C - Medium, accessed December 22, 2025, https://medium.com/@rafael-22/the-clean-architecture-i-wish-someone-had-explained-to-me-dcc1572dbeac
5. Clean Architecture with NET 8 - YouTube, accessed December 22, 2025, https://www.youtube.com/watch?v=lsmyqNrfQQw
6. The Beginner's Guide to Clean Architecture - YouTube, accessed December 22, 2025, https://www.youtube.com/watch?v=TQdLgzVk2T8
7. Understanding Clean Architecture in .NET — A Beginner-Friendly Guide, accessed December 22, 2025, https://akashjwork.medium.com/clean-architecture-275181cab655
8. Introduction to Clean Architecture with .NET - DEV Community, accessed December 22, 2025, https://dev.to/chinonsoike/introduction-to-clean-architecture-with-net-5ab4
9. Common web application architectures - .NET - Microsoft Learn, accessed December 22, 2025, https://learn.microsoft.com/en-us/dotnet/architecture/modern-web-apps-azure/common-web-application-architectures
10. Next-Level Boilerplate: An Inside Look Into Our .Net Clean Architecture Repo - ISE Developer Blog, accessed December 22, 2025, https://devblogs.microsoft.com/ise/next-level-clean-architecture-boilerplate/
11. Understanding Clean Architecture with a Chef Example | by Roshika Nayanadhara, accessed December 22, 2025, https://medium.com/@roshikanayanadhara/understanding-clean-architecture-with-a-chef-example-2878005abd4d
12. Why Your System is Like a Restaurant: A Guide to Layered Architecture - Get SDE Ready, accessed December 22, 2025, https://getsdeready.com/why-your-system-is-like-a-restaurant-a-guide-to-layered-architecture/
13. Clean Architecture: A Deep Dive into Structured Software Design | Spaceteams, accessed December 22, 2025, https://www.spaceteams.de/en/insights/clean-architecture-a-deep-dive-into-structured-software-design
14. Rich vs Anemic Domain Model [closed] - Stack Overflow, accessed December 22,

2025, https://stackoverflow.com/questions/23314330/rich-vs-anemic-domain-model
15. Refactoring From an Anemic Domain Model To a Rich Domain Model - Milan Jovanović, accessed December 22, 2025, https://www.milanjovanovic.tech/blog/refactoring-from-an-anemic-domain-model-to-a-rich-domain-model
16. List differences: DTO, VO, Entity, Domain, Model - Stack Overflow, accessed December 22, 2025, https://stackoverflow.com/questions/72025894/list-differences-dto-vo-entity-domain-model
17. Anemic Domain Model vs Rich Domain Model with Examples - The Valuable Dev, accessed December 22, 2025, https://thevaluable.dev/anemic-domain-model/
18. Rich vs. Anaemic Domain Models in Clean Architecture: Principles, Trade-offs, and Practical Guidance (Part 1) - Alex Tjahjana, accessed December 22, 2025, https://alextjahjana.medium.com/rich-vs-anaemic-domain-models-in-clean-architecture-principles-trade-offs-and-practical-8940b3b8d176
19. What is the use of DTO instead of Entity? - Software Engineering Stack Exchange, accessed December 22, 2025, https://softwareengineering.stackexchange.com/questions/373284/what-is-the-use-of-dto-instead-of-entity
20. DTO vs Entity: Why You Should Separate Concerns - DEV Community, accessed December 22, 2025, https://dev.to/lakmal_asela_8be4eb30d9db/dto-vs-entity-why-you-should-separate-concerns-15jd
21. Clean architecture - where to put input validation logic? - Stack Overflow, accessed December 22, 2025, https://stackoverflow.com/questions/57603422/clean-architecture-where-to-put-input-validation-logic
22. .NET Domain Driven Design implemented with Clean Architecture and Vertical slices : r/dotnet - Reddit, accessed December 22, 2025, https://www.reddit.com/r/dotnet/comments/1dmiv1h/net_domain_driven_design_implemented_with_clean/
23. Differences Between Repository and Service? | Baeldung on Computer Science, accessed December 22, 2025, https://www.baeldung.com/cs/repository-vs-service
24. Clean Architecture Mystery: Interface Adapters | by Plainionist - Medium, accessed December 22, 2025, https://medium.com/@plainionist/clean-architecture-mystery-interface-adapters-47ba66b7f26d
25. REPOSITORY DESIGN PATTERN IN ASP.NET CORE. | by Adeagbo Promise - Medium, accessed December 22, 2025, https://medium.com/our-internship-journey/repository-design-pattern-in-asp-net-core-bf5e7fd6a4b
26. Why SHOULD you use the Repository Pattern? And how to do it, the easy way... | by Mihai Blebea | Medium, accessed December 22, 2025,

https://medium.com/@mihaiblebea/why-should-you-use-the-repository-pattern-in-your-project-and-how-to-do-it-the-easy-way-2076e0889e2b
27. Designing the infrastructure persistence layer - .NET | Microsoft Learn, accessed December 22, 2025, https://learn.microsoft.com/en-us/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/infrastructure-persistence-layer-design
28. Repository and Unit of Work Patterns in ASP.NET Core: A Key to Cleaner C# Code and Architecture | by Uloma Okenyi | Medium, accessed December 22, 2025, https://medium.com/@ulomaobilookenyi/repository-pattern-in-asp-net-core-a-key-to-cleaner-c-code-and-architecture-37f76854c9fb
29. The Dark Side of Repository Pattern: A Developer's Honest Journey | by Mesut Atasoy, accessed December 22, 2025, https://medium.com/@mesutatasoy/the-dark-side-of-repository-pattern-a-developers-honest-journey-eb51eba7e8d8
30. Why shouldnt you use repository pattern : r/dotnet - Reddit, accessed December 22, 2025, https://www.reddit.com/r/dotnet/comments/14xxtwb/why_shouldnt_you_use_repository_pattern/
31. A Comprehensive Guide to JWT Authentication in .NET Core | by Emre Menekşe - Medium, accessed December 22, 2025, https://medium.com/@emreemenekse/a-comprehensive-guide-to-jwt-authentication-in-net-core-8e2d8859b1be
32. Understanding Program.cs in .NET. .NET 9 continue the “minimal hosting”... | by Karam Yaaqba | Medium, accessed December 22, 2025, https://medium.com/@kyaaqba/understanding-program-cs-in-net-78306fcb61e6
33. Dependency injection - Wikipedia, accessed December 22, 2025, https://en.wikipedia.org/wiki/Dependency_injection
34. Can someone explain when to use Singleton, Scoped and Transient with some real life examples? : r/csharp - Reddit, accessed December 22, 2025, https://www.reddit.com/r/csharp/comments/1acwtar/can_someone_explain_when_to_use_singleton_scoped/
35. Understanding Service Lifetimes in ASP.NET Core .NET 8: Transient, Scoped, and Singleton | by Ravi Patel | Medium, accessed December 22, 2025, https://medium.com/@ravipatel.it/understanding-service-lifetimes-in-asp-net-core-net-8-transient-scoped-and-singleton-fd48752fab4b
36. A Comprehensive Guide to Designing an API from Scratch with ASP.NET Core and Clean Architecture, Domain Driven Design (DDD), CQRS, accessed December 22, 2025, https://sayyedulawwab.com/blog/a-comprehensive-guide-to-designing-an-api-from-scratch-with-aspnet-core-and-clean-architecture-domain-driven-design-cqrs/
37. Should I bother unit testing my repository layer - Stack Overflow, accessed December 22, 2025, https://stackoverflow.com/questions/4452928/should-i-bother-unit-testing-my-repository-layer