

College Management System

Architecture Design Guide

PART 1

High-Level System Overview · Clean Architecture · CQRS · MediatR

Admission	Fees	Exam	Stock	Certificates
-----------	------	------	-------	--------------

Stack	.NET 8 · Clean Architecture · CQRS · MediatR · EF Core · Azure
Audience	Beginner-to-Intermediate .NET Developers
Based on	Real-world database schema (lakshits_admission) — ASP.NET C# project

1	High-Level System Overview
2	Clean Architecture — Layer by Layer
3	CQRS — Command vs Query (with real examples)
4	Why Clean Architecture + CQRS + MediatR together

1. High-Level System Overview

What are we building and how does it all fit together?

This College Management System is a cloud-based, modular platform built for Indian colleges. It handles the full student lifecycle — from the moment a student applies for admission, all the way through fee payment, exam form submission, and certificate generation. Instead of one big messy codebase, we split everything into clean, independent modules that talk to each other through well-defined boundaries.

Your existing database (lakshits_admission) already captures the core data — StudentRegistrationData, PersonalData, DocumentData, EducationalData, CourseStudentData, and so on. In the new architecture, each of these tables maps to a domain entity inside its own module. Nothing shares a database table across module boundaries unless absolutely necessary.

The Five Core Modules

Module	What it does	Key DB Tables (existing)
Admission	Online apply, document verify, merit list, confirm admission	StudentRegistrationData, PersonalData, DocumentData
Fees	Admission fees, hostel fees, prospectus fees, payment processing	ProspectusData, StudentRegistrationData (fees col)
Exam	Exam form, roll lists, admit card, fee collection	CourseStudentData, SemDataMaster, PercentageDataBSC
Stock	Consumable & dead stock entry and tracking	(New tables required)
Certificates	TC, Bonafide, Migration, Marksheet, ID card generation	PersonalData, EducationalData, CourseStudentData

End-to-End Student Journey



Each arrow above represents a Command in CQRS — a state change that goes through the Application Layer and gets handled by MediatR. The read operations (merit list, student report, roll list) are Queries.

2. Clean Architecture — Layer by Layer

How the project is divided and why each layer exists

Clean Architecture is about keeping your core business rules (domain logic) completely isolated from databases, frameworks, and external services. Think of it as rings — the inner rings don't know anything about the outer ones. This is why you can swap EF Core for Dapper, or change from SQL Server to PostgreSQL, without touching your business logic.

The Four Layers

API Layer	Controllers · Endpoints · JWT Auth · Swagger
Application Layer	Commands · Queries · Handlers · DTOs · Validators
Domain Layer	Entities · Aggregates · Domain Events · Interfaces
Infrastructure Layer	EF Core · SMS · Payment · File Storage · External APIs

Domain Layer (The Core)

- Contains your business entities: Student, Admission, Payment, Certificate, ExamForm.
- Has no NuGet package references — pure C# classes only.
- Defines interfaces like `IStudentRepository` but does NOT implement them.
- Real example: The Student entity has rules like 'a student cannot have two active admissions'.
- In your DB: `StudentRegistrationData` becomes the Student aggregate root here.

Application Layer (Use Cases)

- Where every feature of the system lives — as Commands or Queries.
- Uses MediatR to dispatch requests to the right handler.
- Example: `CreateAdmissionCommand` triggers `CreateAdmissionHandler`.
- Calls `IStudentRepository` (the interface) — not the actual database.
- Contains FluentValidation validators and DTO (Data Transfer Objects).

Infrastructure Layer (Technical Details)

- Implements all the interfaces defined in Domain.
- EF Core `DbContext` lives here — it knows about SQL Server and your tables.
- SMS service (Twilio / MSG91), Payment Gateway, Azure Blob Storage all live here.
- In your project: mapping `PersonalData`, `DocumentData` to EF entities happens here.

API Layer (Entry Point)

- ASP.NET Core Web API controllers.
- Controller receives HTTP request, creates a Command or Query, sends via MediatR.
- Handles JWT authentication, role-based authorization, Swagger docs.

- The controller knows NOTHING about the database — it just sends/receives messages.

Key Principle: Dependencies always point INWARD. Infrastructure depends on Application. Application depends on Domain. Domain depends on nothing. This is called the Dependency Rule.

3. CQRS — Command vs Query

Separating write operations from read operations (with real college examples)

CQRS stands for Command Query Responsibility Segregation. The idea is simple: anything that **CHANGES** data is a Command, and anything that **READS** data is a Query. They are handled separately, which makes the code easier to understand, test, and scale.

COMMANDS (Write — Changes Data)	QUERIES (Read — Returns Data)
• CreateAdmissionCommand — Student submits application	• GetStudentDetailsQuery — Fetch student profile
• VerifyDocumentsCommand — Admin approves uploaded docs	• GetMeritListQuery — Category-wise merit list
• ProcessPaymentCommand — Student pays admission fees	• GetAdmissionStatusQuery — Check application status
• ConfirmAdmissionCommand — Admission is finalized	• GetFeeReceiptQuery — Download payment receipt
• SubmitExamFormCommand — Student submits exam form	• GetRollListQuery — Exam roll list for department
• GenerateAdmitCardCommand — System creates admit card	• GetDepartmentReportQuery — Admission stats report
• IssueCertificateCommand — TC / Bonafide generated	• GetPendingDocumentsQuery — Docs not yet verified
• RejectAdmissionCommand — Admin rejects with reason	• GetStudentCertificateQuery — Fetch certificate PDF

Real Example: Student Submits Admission Application

Step	Detail
What happens	Student fills form online and hits Submit
Command created	CreateAdmissionCommand { StudentId, CourseId, Documents, PaymentInfo }
Sent via MediatR	mediator.Send(command) — MediatR finds the right handler
Handler runs	CreateAdmissionHandler validates data, calls repository, saves to DB
Side effects	SMS notification sent, registration number generated
Response	Returns AdmissionCreatedDto with registration number

Real Example: Admin Views Department-wise Merit List

This is a Query — no data changes, only reading. GetMeritListQuery { Department, Category, Year } goes through GetMeritListHandler, which queries the read side (could be a separate optimized view or just a filtered EF query),

and returns `MeritListDto[]` sorted by percentage. Notice that this handler does NOT go through the domain entities or business rules — it reads directly for performance.

CQRS tip: Commands go through domain entities and enforce business rules. Queries skip the domain and read directly from the database using DTOs for speed. This is why your reports (merit list, fee reports, gender-wise stats) are always fast.

4. Why Clean Architecture + CQRS + MediatR Together

Each one solves a specific problem — together they make the system production-ready

Pattern / Tool	Problem It Solves	College System Example
Clean Architecture	Prevents business logic from mixing with DB and UI code	AdmissionModule doesn't care if you use SQL Server or MongoDB
CQRS	Stops read and write logic from getting tangled together	RegistrationReport doesn't load Student aggregates — just raw SQL
MediatR	Removes direct dependencies between controller and handlers	AdmissionController doesn't import AdmissionService — just sends a message
Together	Each feature is self-contained, testable, and replaceable	ExamModule replaced ExamModule without touching AdmissionModule

The Request Pipeline — How a Feature Actually Runs

Every single feature in this system follows the same path. Once you understand this pipeline, you can build any feature — admission, fees, certificates — by just filling in the same template.

1	HTTP Request hits the Controller	↓
2	Controller creates a Command or Query object	↓
3	mediator.Send() dispatches it	↓
4	MediatR finds the correct Handler	↓
5	Handler validates, applies domain rules	↓
6	Handler calls Repository (interface)	↓
7	Infrastructure executes DB query	↓
8	Response flows back up the chain	↓
9	Controller returns HTTP 200 + DTO	✓

Your existing project (lakshits_admission) built in ASP.NET WebForms mixes data access, business logic, and UI all in one layer. The new architecture separates all three cleanly. Your StudentRegistrationData table becomes the Student entity in the Domain layer, the EF mapping lives in Infrastructure, and the admission logic lives in Application as Commands and Queries. Nothing else changes — you keep the same data, just organize the code better.

NEXT → PART 2 covers the complete folder structure with real examples from Admission and Fees modules.