* 1. Service design and architecture:

This section provides a comprehensive overview of the design phase, covering service design, architecture determination, detailed design, and modeling. It highlights the significance of these steps in creating a robust service capable of managing multiple functionalities efficiently.

The Base Management Service serves as the core of the system and functions as an API Gateway, facilitating communication between the frontend and backend services. It handles essential operations, processes API requests, and integrates with other system microservices (the Notification Service, AI Service, and Meeting Service.)

* Service Architecture:

System design involves defining the overall structure and architecture of the software, ensuring that all components interact and integrate seamlessly. This phase establishes the blueprint for an efficient data flow, interaction model, and integration strategy within the system.

One of the first constraints to consider when designing the architecture is non-functional requirements (quality attributes). Since the Base Management Service plays a central role in the system, it must meet the following key non-functional requirements:

* Extensibility: The system should allow new features or functionalities to be added with minimal changes to the existing codebase and without affecting core components. It should seamlessly communicate with new services that may be integrated in the future.
* Security: The system must be able to protect all sensitive user data, ensuring robust authentication, authorization, strong password policies, and a well-structured architecture to prevent vulnerabilities.
* Performance: The system should respond to any action in less than 3 seconds, ensuring a smooth and responsive user experience.
* Maintainability: The system should allow modifications, bug fixes, and feature updates with minimal effort, ensuring that changes do not disrupt other components.
* Given these strict requirements, selecting the right architectural pattern is crucial to achieving a scalable, maintainable, and extensible system.

To address these non-functional requirements while maintaining a modular, scalable, and efficient system, the **Clean Architecture pattern** has been adopted for the Base Management Service.

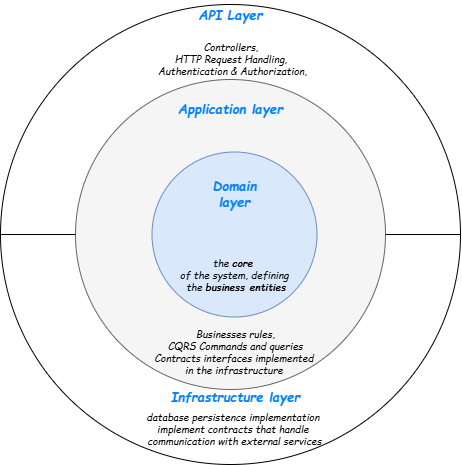


Figure 15 Base management service architecture

Benefits of Clean Architecture for This Service

1. Separation of Concerns – The system is divided into independent layers, each handling a specific concern, making the code clean, modular, and easier to maintain.
2. Scalability & Extensibility – New features can be added to the system without disrupting the existing architecture.
3. Security – Sensitive data is handled within a well-defined structure, and access control mechanisms are centralized within specific layers.
4. Testability – Since business logic is isolated from external dependencies, unit and integration testing become more effective.
5. Technology Independence – The core business logic remains decoupled from frameworks, databases, and external services, making future migrations and updates easier.

By implementing Clean Architecture, the Base Management Service maintains flexibility, robustness, and adaptability, making it ideal for microservice-based systems.

* CQRS + Mediator Pattern:

Since this service functions as an API Gateway and a core service, it handles a large number of requests and integrates with multiple external services. Using CQRS (Command Query Responsibility Segregation) alongside Mediator brings

several advantages:

* Separation of Read and Write Operations – Queries (read operations) and commands (write operations) are handled separately, optimizing database performance and scalability.
* Improved Performance – By separating read models from write models, queries can be optimized independently, reducing database load and improving response time.
* Scalability – Read-heavy operations and write-heavy operations can scale separately based on system needs.
* Decoupling Components – The Mediator pattern eliminates direct dependencies between handlers and controllers, making the system more modular and testable.

By combining Clean Architecture with CQRS + Mediator, the Base Management Service ensures a highly scalable, high-performance, and maintainable system.

CQRS Flow Diagram:

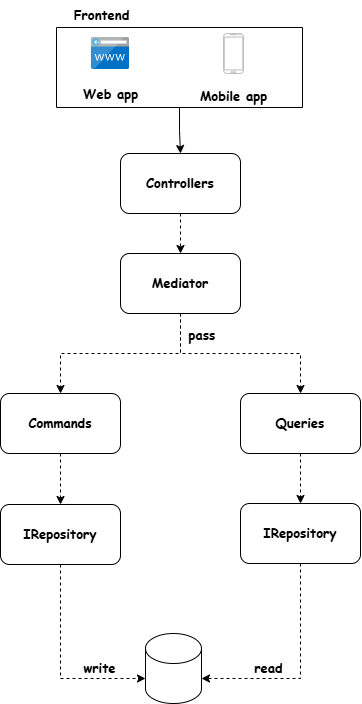


Figure 16 CQRS flow

Service Layers and Responsibilities:

The Base Management Service is structured into multiple layers, each responsible for specific tasks:

* 1. API Layer (Presentation Layer):
* Handles HTTP requests via controllers.
* Exposes RESTful APIs for communication between the frontend and backend.
* Implements authentication, authorization, and request validation.
* Routes API requests to the Application Layer.
  1. Application Layer
* Contains business logic and implements CQRS + Mediator.
* Handles commands and queries separately, improving performance and scalability.
* Defines service contracts (interfaces) for interacting with external services.
* Ensure that business rules are decoupled from infrastructure concerns.
  1. Domain Layer (Core Business Logic)
* Contains database entities, aggregates, and business rules.
  1. Infrastructure Layer
* The main functionality to manages and implement the external dependencies such as databases persistence, message brokers, and third-party APIs to communicate with other system services in our system (AI service , meeting service , Notifications service).

By utilizing Clean Architecture, CQRS, and Mediator, the Base Management Service achieves:

* + - High maintainability through modular design
    - Improved performance by optimizing read/write operations separately.
    - Seamless extensibility to integrate new services without major changes.

This approach ensures that the Base Management Service remains reliable, flexible, and efficient, serving as a robust foundation for the entire system.

* Used design patterns:
  1. **Strategy design pattern – define a family of file parser:**

The **Strategy Pattern** is used to define a family of file parsers, encapsulate them, and make them interchangeable. Instead of handling file parsing logic in a **single class**, each file format (**CSV, Excel, etc.**) has its **own dedicated strategy**.

### **The use of the strategy design pattern improves the system:**

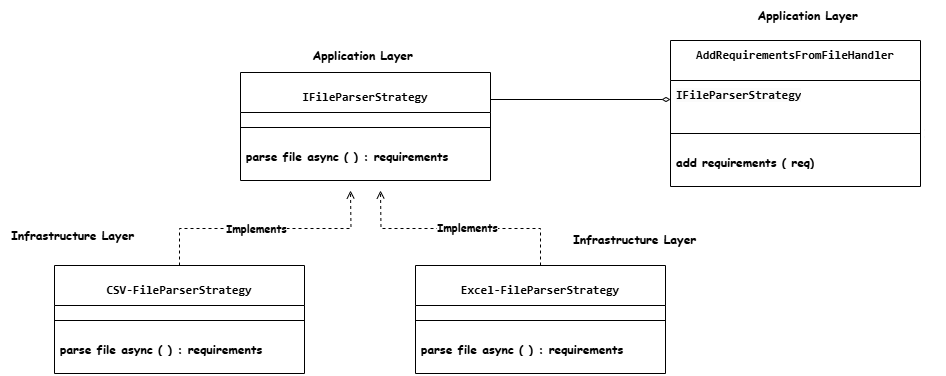
* **Encapsulation of Parsing Logic** – Each file format has a separate class, making the code more modular.
* **Easier Maintenance & Extensibility** – New file formats (e.g., JSON, XML) can be added without modifying existing logic.
* **Single Responsibility Principle (SRP)** – Each class handles only one file type, improving clarity and maintainability.

Figure 17 strategy design pattern

* IFileParserStrategy **Interface** defines the contract for all file parsers.
* ExcelFileParserStrategy **&** CsvFileParserStrategy implement the interface to parse specific file formats.
* The **application layer** interacts with these parsers through the interface, without worrying about file-specific logic.
  1. **Factory design pattern - dynamically select the appropriate file parser strategy:**

The **Factory Pattern** is used to dynamically select the appropriate **file parser strategy** based on the file type. Instead of manually choosing a parser, the **Factory Pattern automates the selection process**.

### **The use of the factory design pattern improves the system:**

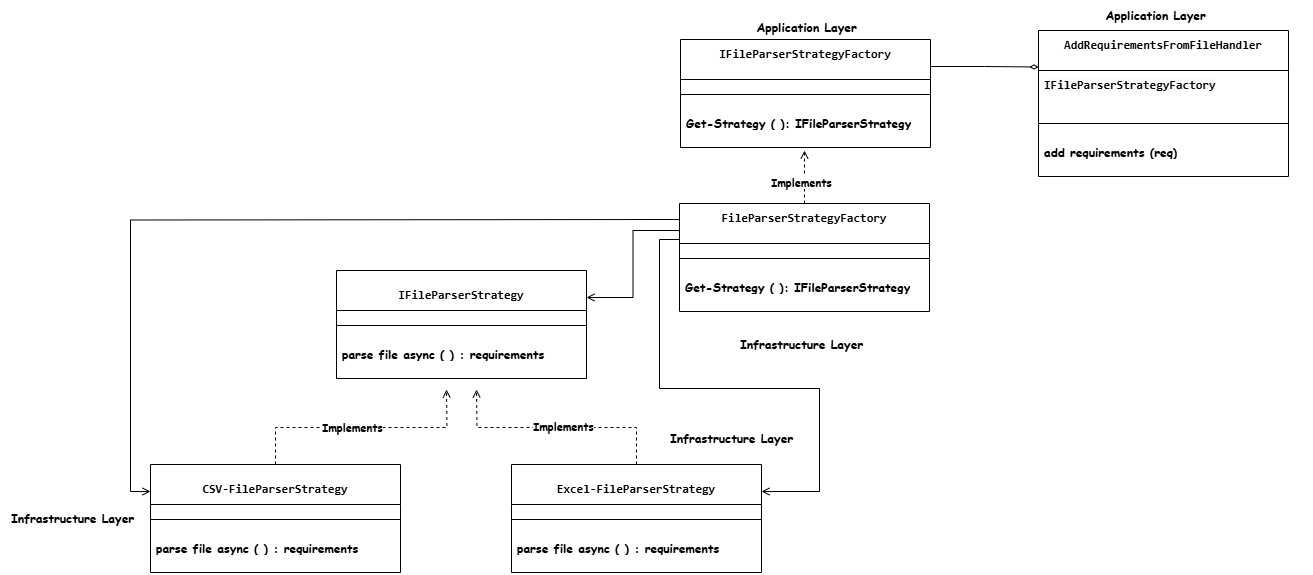
* **Centralized Decision-Making** – The factory decides which parser to use, keeping the main application logic clean.
* **Reduces Code Duplication** – No need for if-else conditions scattered across the codebase.
* **Easy to Extend** – Adding support for a new file type requires **only a new parser class** and a small update in the factory.
* The file parsing process after using factory design pattern:

Figure 18 Factory design pattern

* FileParserStrategyFactory determines the correct strategy based on file extension (.csv, .xlsx).
* **Returns the corresponding parser (**CsvFileParserStrategy**,** ExcelFileParserStrategy**).**
* The **application layer only interacts with** IFileParserStrategy, ensuring loose coupling.
* Authorization solution - design:

The Base Management Service follows a privilege-based authorization model that allows users to have different access levels at both the organization-wide and project-specific levels. This approach ensures that users can only access resources and actions based on their assigned privileges.

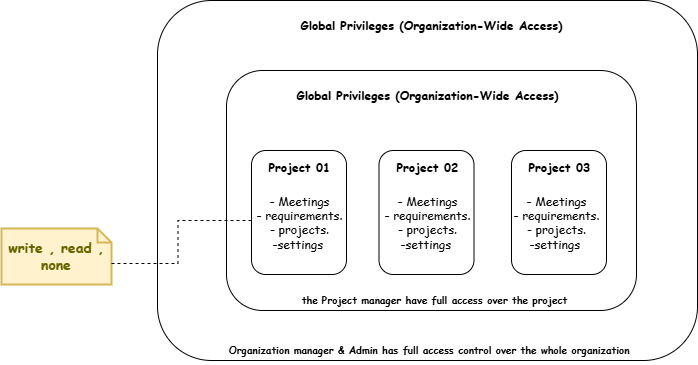


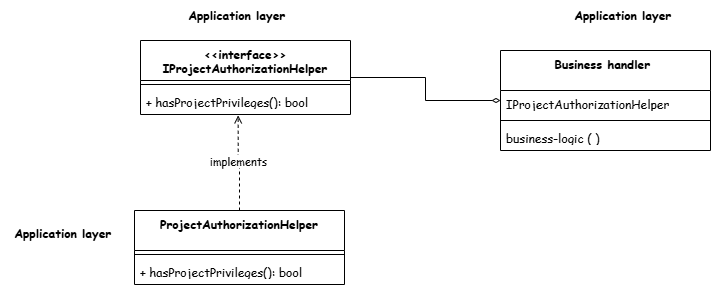
Figure 19 privileges design

**Global Privileges (Organization-Wide Access):**

* Admin & Organization Manager: Have full access across all projects and organization settings.
* They do not need per-project privileges, as they have automatic access to everything.
* Members have no privileges before adding to a project.

**Local Privileges (Project-Specific Access):**

* Organization Members
* Their access is restricted to specific projects.
* They have different privilege levels for each aspect of a project.

In our system the authorization logic handled in the business layer more manually to achieve domain related privileges and authorization

To enforce this privilege-based authorization, the ProjectAuthorizationHelper class is used. It follows the Authorization Helper Pattern, ensuring that authorization logic remains centralized and reusable across the application.

Since authorization is an application concern, the ProjectAuthorizationHelper is implemented in the Application Layer and used in Command & Query Handlers (CQRS).

* Data base schema design – data modeling:

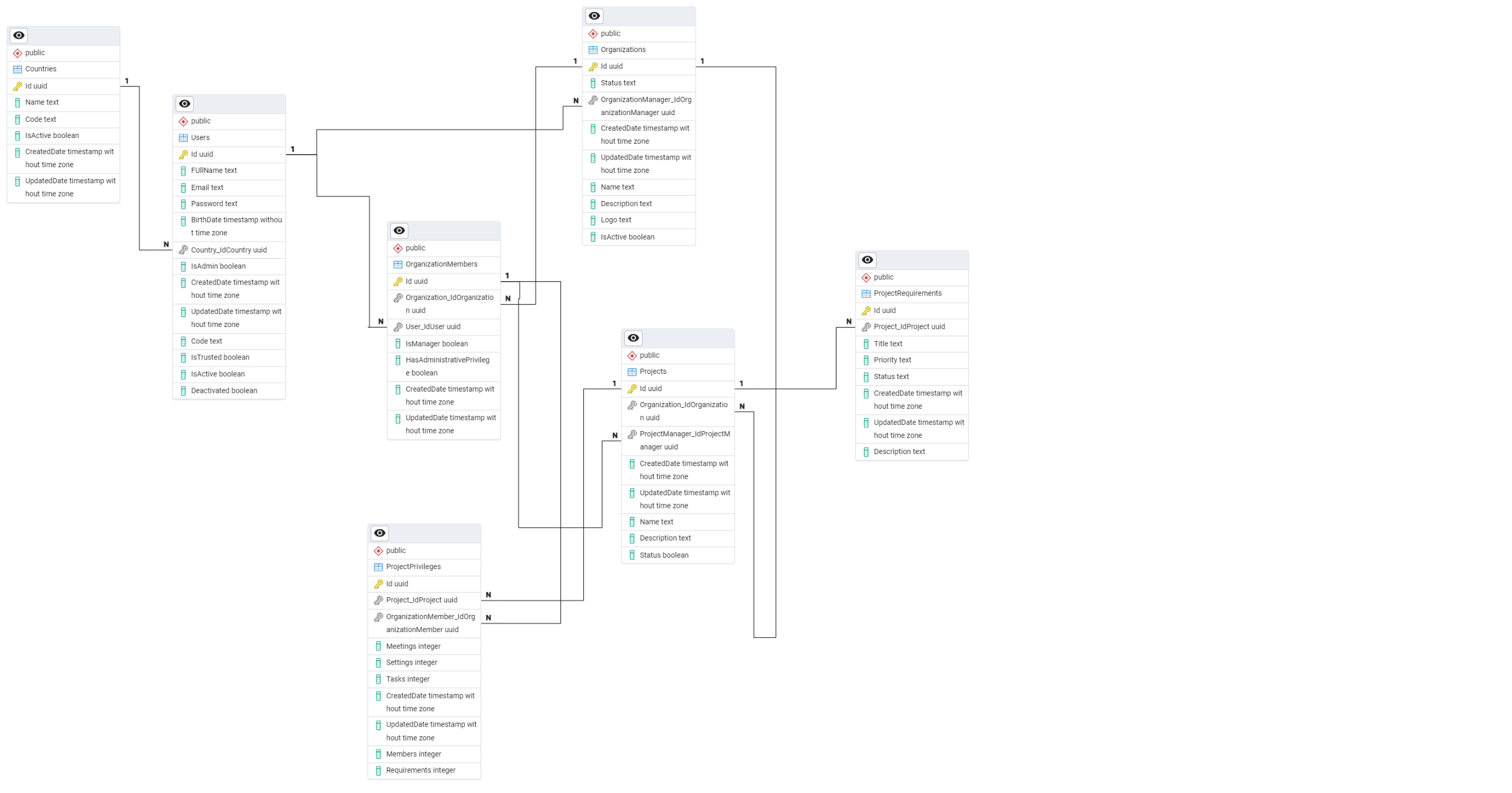


Figure 20 data modeling