**Service Technical Documentation**

**Table of Contents**

1. Architecture Overview
2. Project Structure
3. Technology Stack
4. Implementation Details
5. Messaging and Event Handling
6. Logging and Monitoring
7. Best Practices
8. Getting Started Guide

**1. Architecture Overview**

**Clean Architecture Implementation**

The service follows Clean Architecture principles as popularized by Jason Taylor, organizing code into concentric circles of dependency:

* Core (Domain & Application layers)
* Infrastructure
* API (Presentation layer)

ServiceName/

├── src/

│ ├── ServiceName.Domain/ # Enterprise business rules

│ ├── ServiceName.Application/ # Application business rules

│ ├── ServiceName.Infrastructure/ # External concerns

│ └── ServiceName.API/ # Entry point

└── tests/

├── ServiceName.UnitTests/

└── ServiceName.IntegrationTests/

**CQRS Pattern**

The service implements Command Query Responsibility Segregation (CQRS) pattern to separate read and write operations:

* Commands: Handle state changes (Create, Update, Delete)
* Queries: Handle data retrieval (Read)

This separation allows for:

* Independent scaling of read and write workloads
* Optimized data schemas for each operation type
* Simplified business logic
* Better performance monitoring and optimization

**2. Project Structure**

**Domain Layer**

Domain/

├── Entities/ # Business entities

├── Enums/ # Enumeration types

├── Events/ # Domain events

└── ValueObjects/ # Value objects

Application Layer

Application/

├── Common/

│ ├── Behaviors/ # Pipeline behaviors

│ ├── Exceptions/ # Custom exceptions

│ └── Interfaces/ # Core interfaces

├── Commands/ # Write operations

│ ├── CreateEntity/

│ │ ├── CreateEntityCommand.cs

│ │ └── CreateEntityCommandHandler.cs

│ └── UpdateEntity/

├── Queries/ # Read operations

│ └── GetEntity/

├── DTOs/ # Data transfer objects

└── Mappings/ # AutoMapper profiles

Infrastructure Layer

Infrastructure/

├── Persistence/

│ ├── Configurations/ # Entity configurations

│ ├── Migrations/ # Database migrations

│ └── Repositories/ # Repository implementations

├── Messaging/

│ ├── Consumers/ # RabbitMQ consumers

│ └── Publishers/ # RabbitMQ publishers

└── Logging/ # Seq logging setup

**3. Technology Stack**

**Core Technologies**

* **.NET Core 8**: Base framework
* **MediatR**: CQRS and Mediator pattern implementation
* **AutoMapper**: Object-to-object mapping
* **PostgreSQL**: Primary database
* **RabbitMQ**: Message broker
* **Seq**: Structured logging

**Storage Microservice Design Document**

1. Requirements

### Functional Requirements

- Upload files

- Download files using unique identifiers

- Delete files

- Retrieve file paginated

### Non-Functional Requirements

- High availability (99.9% uptime)

- Scalability to handle large files and high concurrent access

- Low latency for file retrieval (<200ms)

- Security (encryption at rest and in transit)

- Data durability and backup

- Cost-effective storage utilization

- Monitoring and logging capabilities

- Access control and authentication

- Compliance with data protection regulations

## 2. System Design

### High-Level Design

```

[Client Services] --> [API Gateway]

|

[Storage Service]

|

--------------------------------

| | |

[Object Storage] [Database]

### Low-Level Design Components

* 1. API Layer

A computer screen shot of a program code

Description automatically generated

* 1. Application Layer

A screenshot of a computer program

Description automatically generated

* 1. Domain Models

A screen shot of a computer

Description automatically generated

## 3. Storage Implementation

### Storage Solution: Hybrid Approach

1. Object Storage (Primary)

- Google Cloud Storage

- Reasons:

- Cost-effective for large files

- Highly durable and available

- Built-in versioning

- Geographic replication

- Pay-per-use model

2. Database (Metadata)

- PostgreSQL

- Stores file metadata, paths, and relationships

- Enables quick searches and filtering

- Maintains relationships between files and services

## 3. Inter-Service Communication

### Communication Patterns

* 1. Synchronous (REST API) using “StorageController”
  2. Asynchronous (Message Queue RabbitMQ)