**Order Management System - Technical Approach & Architecture**

**Executive Summary**

This document outlines the technical approach, architectural decisions, and design patterns implemented in the Order Management System. The solution demonstrates modern .NET development practices, emphasizing maintainability, testability, and scalability through Clean Architecture principles.

**1. Architectural Foundation & Philosophy**

**1.1 Clean Architecture Selection**

**Decision**: Implemented Clean Architecture with clear separation of concerns across five distinct layers.

**Rationale**:

* **Dependency Inversion**: Core business logic remains independent of external concerns
* **Testability**: Each layer can be tested in isolation without external dependencies
* **Maintainability**: Changes in infrastructure don't ripple through business logic
* **Team Scalability**: Clear boundaries enable parallel development across different layers

**1.2 Layer Responsibilities**

┌─────────────────┐

│ Presentation │ ← API Controllers, Swagger, Validation

├─────────────────┤

│ Application │ ← CQRS Handlers, DTOs, Business Services

├─────────────────┤

│ Domain │ ← Entities, Value Objects, Business Rules

├─────────────────┤

│ Infrastructure │ ← External Services, Caching, Events

├─────────────────┤

│ Persistence │ ← EF Core, Repositories, Database Config

└─────────────────┘

**Key Principle**: Dependencies flow inward - outer layers depend on inner layers, never the reverse.

**2. Domain Modeling & Business Logic Implementation**

**2.1 Domain-Driven Design Approach**

**Decision**: Rich domain models with encapsulated business logic rather than anemic data models.

**Implementation Highlights**:

* **Aggregate Roots**: Order and Customer manage their own consistency boundaries
* **Value Objects**: Money type with proper arithmetic operations and currency validation
* **Domain Events**: Status changes trigger events for cross-cutting concerns
* **Invariant Enforcement**: Business rules validated within domain entities

**Example - Order Status Validation**:

private static bool IsValidStatusTransition(OrderStatus current, OrderStatus target)

{

return current switch

{

OrderStatus.Pending => target is OrderStatus.Confirmed or OrderStatus.Cancelled,

OrderStatus.Confirmed => target is OrderStatus.Processing or OrderStatus.Cancelled,

// ... business rules encoded in domain

};

}

**Benefits**:

* Business logic centralized and easily testable
* Impossible to create invalid domain states
* Clear expression of business requirements in code

**2.2 Event-Driven Architecture**

**Decision**: Implemented domain events for decoupled cross-cutting concerns.

**Pattern Used**: Domain Events with MediatR notification publishing

* Events generated during business operations
* Handlers process notifications asynchronously
* Cache invalidation, analytics updates, and audit trails triggered by events

**Assumption**: Events are processed immediately after persistence for consistency, with future consideration for eventual consistency in distributed scenarios.

**3. Application Layer & CQRS Implementation**

**3.1 CQRS Pattern Selection**

**Decision**: Command Query Responsibility Segregation for clear separation of read and write operations.

**Rationale**:

* **Read Optimization**: Queries optimized with caching and projections
* **Write Optimization**: Commands focus on business rule validation and persistence
* **Scalability**: Different optimization strategies for reads vs writes
* **Maintainability**: Clear separation of concerns between data modification and retrieval

**3.2 MediatR Pipeline Architecture**

**Implementation**: Request/Response pattern with pipeline behaviors for cross-cutting concerns.

Request → Validation → Business Logic → Response

↓ ↓ ↓ ↓

Pipeline Pipeline Handler Pipeline

Behavior Behavior (Command/Query) Behavior

**Pipeline Behaviors Implemented**:

* **Validation Behavior**: FluentValidation integration for automatic request validation
* **Future Extension Points**: Logging, caching, authorization behaviors can be added

**Benefits**:

* Consistent request handling across all operations
* Cross-cutting concerns handled declaratively
* Easy to add new behaviors without modifying handlers

**3.3 Service Layer Strategy**

**Decision**: Service Manager pattern coordinating multiple specialized services.

**Services Implemented**:

* **DiscountService**: Orchestrates multiple discount strategies
* **CacheInvalidationService**: Manages cache consistency
* **ServiceManager**: Facade providing unified access to all services

**Rationale**: Separation of concerns while maintaining cohesive business operations.

**4. Discounting Architecture**

**4.1 Strategy Pattern Implementation**

**Decision**: Strategy pattern for flexible discount calculation system.

**Strategies Implemented**:

1. **VipDiscountStrategy** (Priority 1): 15% for VIP customers
2. **LoyaltyDiscountStrategy** (Priority 2): 10% for customers with 5+ orders
3. **BulkOrderDiscountStrategy** (Priority 3): 5% for orders > $500

**Key Design Choice**: **Cumulative discounts** rather than exclusive application.

**Business Assumption**: Multiple discounts can be applied simultaneously, providing maximum customer benefit while remaining capped at order total.

**4.2 Discount Calculation Flow**

Order Creation Request

↓

Evaluate All Strategies

↓

Apply Cumulative Discounts

↓

Validate Business Rules

↓

Persist with Final Amount

**Safety Mechanisms**:

* Discount cannot exceed order total
* No negative discount amounts
* Validation at both domain and application levels

**5. Data Access & Persistence Strategy**

**5.1 Repository Pattern with Unit of Work**

**Decision**: Repository pattern for data access abstraction with Unit of Work for transaction management.

**Implementation Approach**:

* **Generic Base Repository**: Common CRUD operations
* **Specialized Repositories**: Domain-specific queries (analytics aggregations)
* **Unit of Work**: Coordinated saves and transaction boundaries

**Benefits**:

* Clean separation between domain and data access
* Easy to mock for testing
* Centralized transaction management

**5.2 Entity Framework Configuration Strategy**

**Decision**: Configuration-driven entity mapping with value object support.

**Key Configurations**:

* **Value Object Mapping**: Money type mapped as owned entities
* **Index Strategy**: Performance indexes on frequently queried columns
* **Relationship Mapping**: Proper foreign key constraints and cascade behaviors

**Assumption**: In-memory database for development/testing with easy migration path to SQL Server for production.

**5.3 Database Seeding Architecture**

**Decision**: Configurable, idempotent seeding system.

**Features**:

* **Environment-based control**: Seeding only in appropriate environments
* **Idempotent operations**: Safe to run multiple times
* **Configurable scope**: Control what data gets seeded

**6. Performance Optimization Strategy**

**6.1 Multi-Tier Caching Architecture**

**Decision**: Intelligent caching with different strategies based on data characteristics.

**Caching Strategy**:

Data Type | Cache Duration | Invalidation Strategy

Order Analytics | 5 minutes | Immediate on order changes

Date Range Analytics| 1 hour | Extended for historical data

Individual Orders | 10 minutes | Targeted cache key removal

Customer Orders | 5 minutes | Customer-specific invalidation

**Rationale**: Balance between performance and data freshness based on business requirements.

**6.2 Query Optimization Techniques**

**Implementations**:

* **AsNoTracking()**: Read-only queries for better performance
* **Repository-level aggregations**: Analytics calculations pushed to database
* **Efficient indexing**: Strategic indexes on query-heavy columns
* **Date range normalization**: Consistent date handling for cache efficiency

**7. API Design & Documentation Strategy**

**7.1 RESTful API Design**

**Decision**: REST principles with proper HTTP semantics and status codes.

**Design Principles**:

* **Resource-based URLs**: Clear resource identification
* **HTTP Verbs**: Proper use of GET, POST, PUT for operations
* **Status Codes**: Meaningful HTTP status responses
* **Versioning**: URL-based versioning for backward compatibility

**7.2 Comprehensive API Documentation**

**Implementation**: OpenAPI/Scalar with detailed annotations.

**Documentation Features**:

* **Rich XML comments**: Detailed operation descriptions with examples
* **Request/Response models**: Complete schema documentation
* **Business rule explanations**: Discount logic and status transitions explained
* **Interactive testing**: Scalar UI for live API testing

**8. Error Handling & Resilience**

**8.1 Layered Exception Handling**

**Strategy**: Multiple exception handling layers with specific responsibilities.

**Layers**:

1. **Global Exception Handler**: Converts exceptions to proper HTTP responses
2. **Validation Pipeline**: FluentValidation with detailed error messages
3. **Domain Exceptions**: Business rule violations with meaningful messages
4. **Repository Level**: Data access error handling

**8.2 Response Standardization**

**Decision**: Consistent response wrapper pattern for all API operations.

**Benefits**:

* Predictable response structure for clients
* Comprehensive error information
* Timestamp and trace ID for debugging

**9. Testing Strategy & Quality Assurance**

**9.1 Testing Architecture**

**Approach**: Comprehensive testing pyramid with different testing levels.

**Testing Levels**:

* **Unit Tests**: Domain logic, discount strategies, business rules
* **Integration Tests**: API endpoints, database operations
* **Repository Tests**: Data access layer validation

**9.2 Testability Design Decisions**

**Implementations**:

* **Dependency Injection**: All dependencies injected for easy mocking
* **Interface Abstractions**: Clean contracts for test doubles
* **InternalsVisibleTo**: Strategic access for testing internal logic

**10. System Flow & Request Processing**

**10.1 Order Creation Flow**

API Request

↓

Controller Validation

↓

MediatR Command

↓

Validation Pipeline

↓

Command Handler

├─ Customer Validation

├─ Order Creation

├─ Discount Calculation

│ ├─ VIP Strategy Check

│ ├─ Loyalty Strategy Check

│ └─ Bulk Strategy Check

├─ Domain Rule Validation

├─ Persistence

└─ Cache Invalidation

↓

Response Generation

**10.2 Analytics Processing Flow**

Analytics Request

↓

Cache Check

├─ Cache Hit → Return Cached Data

└─ Cache Miss → Continue

↓

Repository Aggregation

↓

Business Logic Application

↓

Cache Storage

↓

Response Return

**11. Assumptions & Design Decisions**

**11.1 Business Assumptions**

1. **Discount Stacking**: Multiple discounts can be applied cumulatively
2. **Customer Progression**: Automatic segment upgrades based on order history
3. **Currency Scope**: Initial USD-only support with extensible design
4. **Order Finality**: Delivered and Cancelled are terminal states

**11.2 Technical Assumptions**

1. **Event Processing**: Domain events processed synchronously for consistency
2. **Cache Consistency**: Acceptable brief inconsistency for performance gains
3. **Database Choice**: In-memory for development with production SQL Server path
4. **Scalability Target**: Moderate concurrent load with infrastructure scaling options

**11.3 Future Scalability Considerations**

**Identified Extension Points**:

* **Event Sourcing**: Domain events provide foundation for event sourcing
* **Distributed Caching**: Interface-based caching enables Redis migration
* **Multi-tenancy**: Clean architecture supports tenant isolation

**12. Conclusion**

The combination of Clean Architecture, Domain-Driven Design, CQRS, and strategic design patterns creates a maintainable, testable, and scalable foundation.

**Key Strengths**:

* **Business Logic Centralization**: Domain-driven approach ensures business rules are clearly expressed and maintained
* **Performance Optimization**: Multi-tier caching and query optimization provide excellent performance characteristics
* **Extensibility**: Strategy patterns and clean interfaces enable easy feature additions
* **Quality Assurance**: Comprehensive testing strategy ensures reliability

The architecture successfully balances immediate requirements with long-term maintainability, demonstrating thoughtful consideration of both business needs and technical constraints.