**Order Management System API**

A .NET 8 Web API implementing an order management system with advanced discounting strategies, analytics, and state management following Clean Architecture principles.

**Features Implemented**

**Core Requirements Met**

**1. Discounting System**

* **VIP Customer Discount**: 15% for VIP segment customers
* **Loyalty Discount**: 10% for customers with 5+ orders
* **Bulk Order Discount**: 5% for orders exceeding $500
* **Discount Stacking**: Multiple discounts are applied cumulatively
* **Safety Constraints**: Discounts capped at order total, no negative discounts

**2. Order Status Tracking**

* **State Machine Implementation**: Enforced valid status transitions
* **Status Flow**: Pending → Confirmed → Processing → Shipped → Delivered
* **Cancellation Support**: Orders can be cancelled at any stage (except final states)
* **Domain Events**: Status changes trigger events for notifications and analytics

**3. Order Analytics Endpoint**

* **Comprehensive Metrics**: Average order value, fulfillment time, status distribution
* **Date Range Filtering**: Analytics for specific time periods
* **Performance Optimized**: Intelligent caching strategy (5 min general, 1 hour for date ranges)
* **Real-time Data**: Automatic cache invalidation on order changes

**Architecture**

**Clean Architecture Implementation**

├── OM.Api # Presentation Layer

├── OM.Application # Application Layer

├── OM.Domain # Domain Layer

├── OM.Infrastructure # Infrastructure Layer

├── OM.Persistence.SQLServer # Data Access Layer

└── Tests/ # Testing Projects

**Key Design Patterns**

* **CQRS with MediatR**: Clear separation of commands and queries
* **Repository Pattern + UnitOfWork**: Data access abstraction
* **Strategy Pattern**: Flexible discount calculation system
* **Domain Events**: Event-driven architecture for cross-cutting concerns
* **Value Objects**: Money type with proper arithmetic operations
* **Aggregate Roots**: Order and Customer with proper boundaries

**Technical Stack**

* **.NET 8**: Latest LTS framework
* **Entity Framework Core**: ORM with In-Memory database
* **MediatR**: CQRS and messaging
* **FluentValidation**: Input validation
* **Swagger/OpenAPI**: API documentation
* **Memory Caching**: Performance optimization
* **Domain Events**: Event-driven architecture

**Prerequisites**

* .NET 8 SDK
* Visual Studio 2022 or VS Code
* SQL Server (optional - uses In-Memory DB by default)

**Getting Started**

**1. Clone and Setup**

git clone [your-repo-url]

cd OrderManagement

dotnet restore

**2. Configuration**

Update appsettings.json for your environment:

{

"ConnectionStrings": {

"OMDB": "your-connection-string-here"

},

"DatabaseSeeding": {

"Enabled": true,

"SeedCustomers": true,

"SeedSampleOrders": false,

"Environment": "Development"

}

}

**3. Run the Application**

dotnet run --project OM.Api

**4. Access API Documentation**

* **Swagger UI**: https://localhost:7208/scalar/v1
* **OpenAPI Spec**: https://localhost:5170/openapi/v1.json

**API Endpoints**

**Orders Management**

POST /api/v1/order/create # Create new order

GET /api/v1/order/getAll # Get all orders

GET /api/v1/order/getbyId/{id} # Get order by ID

PUT /api/v1/order/updateOrderStatus/{id} # Update order status

**Analytics**

GET /api/v1/analytics/order-analytics # Overall analytics

GET /api/v1/analytics/order-analytics/date-range # Analytics by date range

**Key Business Rules**

**Discount Application**

* **Discount Priority**: All applicable discounts are combined
* **VIP**: 15% (highest priority, CustomerSegment.VIP)
* **Loyalty**: 10% (customers with TotalOrders > 5)
* **Bulk**: 5% (orders with Amount > $500)
* **Example**: VIP customer with 6 orders placing $600 order gets 30% total discount

**Customer Segment Progression**

// Automatic progression based on order history

0-2 orders: CustomerSegment.New

3-9 orders: CustomerSegment.Regular

10+ orders: CustomerSegment.VIP

**Order Status Transitions**

Pending ──┬──> Confirmed ──> Processing ──> Shipped ──> Delivered

└──> Cancelled └──> Cancelled └──> Cancelled

**Configuration Options**

**Database Seeding**

{

"DatabaseSeeding": {

"Enabled": true, // Enable/disable seeding

"SeedCustomers": true, // Seed sample customers

"SeedSampleOrders": false, // Seed sample orders

"Environment": "Development" // Environment restriction

}

}

**Caching Strategy**

* **Order Analytics**: 5 minutes (frequent updates)
* **Date Range Analytics**: 1 hour (stable historical data)
* **Individual Orders**: 10 minutes (moderate update frequency)
* **Customer Orders**: 5 minutes (dynamic data)

**Testing**

**Run Unit Tests**

dotnet test OM.UnitTests

**Run Integration Tests**

dotnet test OM.IntegrationTests

**Test Coverage**

* **Discount Calculation Logic**: Comprehensive unit tests for all strategies
* **Order Status Transitions**: Validation of business rules
* **API Endpoints**: Integration tests for order management
* **Analytics Calculations**: Accuracy and performance testing

**Performance Optimizations**

**1. Caching Strategy**

* **Multi-tier caching** with different expiration policies
* **Smart cache invalidation** on order updates
* **Cache-aside pattern** implementation

**2. Database Optimizations**

* **AsNoTracking()** for read-only queries
* **Efficient indexing** on frequently queried columns
* **Repository-level aggregation** for analytics

**3. Query Optimizations**

* **Date range normalization** for consistent querying
* **Batch operations** for bulk updates
* **Lazy loading** prevention with explicit includes

**Architecture Decisions**

**Why Clean Architecture?**

* **Separation of Concerns**: Each layer has distinct responsibilities
* **Testability**: Easy to unit test business logic in isolation
* **Maintainability**: Changes in one layer don't affect others
* **Scalability**: Can easily switch infrastructure components

**Why CQRS?**

* **Performance**: Optimized read and write operations
* **Scalability**: Separate scaling of reads vs writes
* **Complexity Management**: Clear separation of data modification and retrieval

**Why Strategy Pattern for Discounts?**

* **Flexibility**: Easy to add new discount types
* **Maintainability**: Each discount rule is self-contained
* **Testability**: Individual strategies can be tested in isolation
* **Business Rule Evolution**: Supports changing business requirements

**Error Handling**

**Global Exception Handling**

* **API Exception Handler**: Converts exceptions to proper HTTP responses
* **Validation Pipeline**: FluentValidation with detailed error messages
* **Domain Exception Types**: Specific exceptions for business rule violations

**Response Patterns**

{

"successful": true,

"message": "Order created successfully",

"data": { ... },

"errors": [],

"timestamp": "2024-01-15T10:30:00Z"

}

**Development Assumptions**

**Business Rules Assumptions**

1. **Discount Stacking**: All applicable discounts are combined (not exclusive)
2. **Currency**: All monetary values use USD (extensible via Money value object)
3. **Customer Progression**: Automatic segment upgrades based on order count
4. **Order Fulfillment**: Status "Delivered" automatically sets fulfillment timestamp

**Technical Assumptions**

1. **Single Currency**: Initially supporting USD only
2. **In-Memory Database**: Using EF InMemory for development/testing
3. **Synchronous Discounts**: Discount calculation is synchronous operation
4. **Event Reliability**: Domain events are processed immediately after persistence

**Performance Assumptions**

1. **Cache Duration**: Analytics cached for 5 minutes (adjustable per business needs)
2. **Concurrent Users**: Designed for moderate concurrent load (scalable with infrastructure changes)
3. **Data Volume**: Optimized for typical e-commerce order volumes

**Future Enhancements**

**Potential Extensions**

* **Multiple Currencies**: Extend Money value object for international support
* **Advanced Analytics**: Time-series analysis, customer behavior insights
* **Notification System**: Email/SMS notifications for status changes
* **Inventory Integration**: Stock management and reservation
* **Payment Processing**: Integration with payment gateways
* **Audit Trail**: Complete order modification history
* **Real-time Updates**: SignalR for live order status updates

**Scalability Considerations**

* **Microservices**: Split into Order, Customer, Analytics services
* **Distributed Caching**: Redis for multi-instance caching

**Support**

For questions or issues:

1. Check API documentation at /scalar/v1
2. Review test cases for usage examples
3. Examine domain events for business rule understanding

**Built with ❤️ using Clean Architecture principles and modern .NET practices**