Enterprise Inventory and Workflow Management System: A Comprehensive Architectural and Implementation Guide

Executive Summary

The development of an Enterprise Inventory and Workflow Management System (EIWMS) requires a rigorous engineering approach that balances transactional integrity, architectural scalability, and user-centric workflow optimization. Unlike standard data-entry applications, an inventory system serves as the operational backbone of an organization, directly influencing financial reporting, supply chain efficiency, and customer satisfaction. The request to build such a system from scratch using ASP.NET Core MVC and Microsoft SQL Server represents a strategic decision to leverage the robustness, type safety, and performance of the Microsoft ecosystem. This report provides an exhaustive, step-by-step guide to constructing this system, moving from high-level architectural paradigms to granular database normalization strategies, complex stored procedure logic, and dynamic security implementations.

The proposed solution adopts **Clean Architecture** as its structural foundation, ensuring that the core business logic—such as stock valuation algorithms and workflow state transitions—remains decoupled from external frameworks and database concerns.[1] This separation is critical for enterprise systems where business rules evolve independently of user interface technologies. Furthermore, the system incorporates a deeply **normalized database schema** designed to handle multi-warehouse operations, complex product variants, and strict audit trails, mitigating the risks of data redundancy and anomalies.[3]

To address the concurrency challenges inherent in inventory management—such as preventing overselling during high-traffic periods—the architecture utilizes **Transactional Stored Procedures** with explicit locking mechanisms.[5] Security is managed through a customized implementation of **ASP.NET Core Identity**, extended with a database-driven Role-Based Access Control (RBAC) system that allows for dynamic permission assignment without code redeployment.[7] Finally, the reporting module integrates **Chart.js** for real-time visualization, powered by optimized SQL views to ensure performance does not degrade as transaction volumes grow.[9]

This document serves as a definitive blueprint for software architects and senior developers, detailing the "why" and "how" of every critical component required to deliver a

production-ready enterprise inventory system.

---

1. Architectural Strategy and Solution Design

1.1 The Imperative for Clean Architecture

In the domain of enterprise software, the primary enemy of longevity is coupling. Traditional N-Tier architectures, while simple to establish, often encourage a dependency structure where business logic is inextricably linked to data access code or UI controllers. This makes testing difficult and refactoring hazardous. For an inventory system, where a bug in stock calculation can lead to financial discrepancies, testability and isolation are paramount. Therefore, this guide mandates the use of **Clean Architecture** (also known as Onion or Hexagonal Architecture).[1]

Clean Architecture inverts the traditional dependency graph. Instead of the core domain depending on the database, the database implementation depends on the core domain. The **Domain Layer** sits at the center, containing enterprise-wide logic and entities (e.g., Product, Order) and having no external dependencies. Surrounding this is the **Application Layer**, which orchestrates user intent (e.g., PlaceOrderCommand, ReceiveGoodsQuery). The outermost layers—**Infrastructure** and **Presentation**—depend only on the inner layers. This ensures that the business rules, the most valuable asset of the software, remain pristine and unaffected by changes in UI frameworks or database drivers.[2]

1.1.1 Solution Composition

A robust folder and project structure is the physical manifestation of this architectural philosophy. The recommended solution layout for this EIWMS is designed to enforce separation of concerns at the compiler level.

Project / Layer	**Responsibility**	**Dependencies**
Inventory.Domain	Defines the core entities, value objects, domain events, and repository interfaces. This is pure C# logic.	None (Zero external NuGet dependencies)
Inventory.Application	Contains the business workflows, DTOs (Data Transfer Objects), validation rules, and service interfaces.	Inventory.Domain
Inventory.Infrastructure	Implements the interfaces defined in Domain/Application.	Inventory.Application, Inventory.Domain

	This includes EF Core DbContext, SQL repositories, and email services.	
Inventory.Web	The ASP.NET Core MVC application. Handles HTTP requests, controllers, views, and client-side assets.	Inventory.Application, Inventory.Infrastructure
Inventory.Tests	Unit and Integration tests to verify logic without spinning up the full UI.	All Layers

This structure ensures that a developer cannot accidentally use an Entity Framework class inside the Domain layer, as the reference simply does not exist. It enforces discipline by design.[12]

1.2 Technology Stack Justification

The choice of technologies is driven by the requirements for stability, performance, and enterprise support.

- **ASP.NET Core MVC (.NET 8/9):** Selected for its mature pattern for building server-side rendered applications. While SPAs (Single Page Applications) are popular, MVC offers simpler SEO, faster initial load times for internal tools, and tighter integration with server-side validation logic.[1]
- **Microsoft SQL Server:** Chosen for its powerful relational engine, support for complex stored procedures, and robust locking mechanisms required to handle inventory concurrency.[3]
- **Entity Framework Core (EF Core):** utilized as an Object-Relational Mapper (ORM) for standard CRUD operations to speed up development. However, for performance-critical paths (like stock allocation), the system bypasses EF Core in favor of raw SQL or Dapper to execute optimized stored procedures.[14]
- **jQuery & Chart.js:** For the client-side, we avoid heavy frameworks like Angular or React to reduce build complexity. jQuery handles DOM manipulation for dynamic forms, while Chart.js provides reporting visualizations.[9]

---

2. Database Engineering: Normalization and Schema Design

The database is the source of truth for the inventory system. A poorly designed schema will lead to data anomalies—such as a product having two different prices in two different tables—that are impossible to reconcile. We apply **Third Normal Form (3NF)** to ensure that every non-key attribute is dependent on the primary key, the whole key, and nothing but the key.[3]

2.1 Product Data Modeling

Modeling products in an enterprise system is complex due to the need for **variants**. A "T-Shirt" is not a sellable item; a "Red T-Shirt, Size L" is. To handle this without creating a separate table for every product type, we employ a hybrid relational model.

2.1.1 The Master-Variant Pattern

- **Products (Master Table):** Represents the abstract product.
    - Id (PK, INT): Unique identifier.
    - Name (NVARCHAR(200)): The general name (e.g., "Cotton T-Shirt").
    - Description (NVARCHAR(MAX)): Marketing copy.
    - CategoryId (FK): Link to the hierarchical category tree.
    - BrandId (FK): Normalizing brands prevents "Nike" vs "Nike Inc" duplication.[16]
    - BaseUOMId (FK): Base Unit of Measure (e.g., "Each", "Kg").
    - TaxGroupId (FK): Critical for financial compliance.
- **ProductVariants (Sellable Entity):** Represents the specific SKU.
    - Id (PK, INT): The actual foreign key used in transactions.
    - ProductId (FK): Link to parent.
    - SKU (NVARCHAR(50)): The unique stock keeping unit code (e.g., "TSHIRT-RED-L"). **Indexed Unique Constraint**.
    - Barcode (NVARCHAR(100)): UPC/EAN code for scanning.
    - CostPrice (DECIMAL(18,4)): Current moving average cost.
    - SalesPrice (DECIMAL(18,4)): Standard selling price.
    - ReorderPoint (INT): Threshold for low-stock alerts.

2.1.2 Handling Dynamic Attributes (EAV vs. JSON)

Products often have attributes specific to their category (e.g., "Screen Size" for laptops, "Fabric" for clothes). Creating columns for all of these leads to a sparse table. While the Entity-Attribute-Value (EAV) model is a traditional solution, it complicates querying. A modern approach using SQL Server is to store these in a **JSONB** column or a normalized attribute structure if strict filtering is needed.[17]
For this guide, we recommend a **Normalized Attribute** approach for searchability:

- **Attributes**: Definitions (e.g., "Color", "Size").
- **ProductAttributeValues**: The link.
    - ProductVariantId (FK)
    - AttributeId (FK)
    - Value (NVARCHAR(100)) - e.g., "Red".

2.2 Inventory and Warehousing Schema

Inventory is not a static number; it is a calculation of location and movement.

- **Warehouses**: Physical buildings.
    - Id, Name, Address.
    - IsNettable (BIT): Determines if stock here counts towards "Available to Promise".
- **Locations (Bin Management)**: Specific spots within a warehouse.
    - Id (PK)
    - WarehouseId (FK)
    - Code (NVARCHAR(20)): e.g., "A-01-02" (Aisle A, Bay 1, Shelf 2).
    - ZoneType (ENUM): Picking, Bulk Storage, Receiving, Quarantine.
- **InventoryStock**: The current snapshot.
    - ProductVariantId (FK)
    - LocationId (FK)
    - QuantityOnHand (DECIMAL): Physical count.
    - QuantityAllocated (DECIMAL): Reserved for sales orders.
    - QuantityAvailable (Computed Column): QuantityOnHand - QuantityAllocated.
- **InventoryTransactions**: The immutable audit trail.[19]
    - Id (PK, BIGINT)
    - Date (DATETIME2)
    - Type (ENUM): Purchase, Sale, Adjustment, Transfer.
    - ProductVariantId (FK)
    - LocationId (FK)
    - QuantityChange (DECIMAL): +/- value.
    - ReferenceDocType (ENUM): Order, PO, Count.
    - ReferenceDocId (INT): Link to the source document.
    - UnitCost (DECIMAL): Cost at the time of movement (essential for FIFO).

2.3 Vendor and Procurement Schema

Managing external suppliers requires capturing their details and the lifecycle of purchase orders.

- **Vendors**:
    - Id, Name, TaxId.

    - PaymentTerms (INT): Days to pay (e.g., 30, 60).
    - Rating (INT): Calculated based on delivery performance.[21]
- **PurchaseOrders (PO)**:
    - Id, VendorId, Date.
    - Status (Draft, Submitted, Partial, Closed).
    - TotalAmount.
- **GoodsReceiptNotes (GRN)**: Represents the physical arrival of goods.
    - Id (PK)
    - PurchaseOrderId (FK)
    - DeliveryNoteNumber: Vendor's document reference.
    - ReceivedDate.

---

3. Core Domain Logic and Workflow Management

The "Workflow" aspect of the system dictates how entities move through their lifecycle. Hardcoding these transitions in controllers leads to "spaghetti code." Instead, we implement a **Finite State Machine (FSM)** pattern within the Domain Layer.

3.1 Order Workflow State Machine

An Order is not just a data row; it is a process. The valid states might be:
Draft -> Confirmed -> Allocated -> Picked -> Shipped -> Invoiced.
We enforce this via a Domain Service or within the Entity itself.

C#

```
public class Order : BaseEntity
{
   public OrderStatus Status { get; private set; }
   public ICollection<OrderLine> Lines { get; private set; }

   public void Confirm()
   {
      if (Status!= OrderStatus.Draft)
         throw new DomainException("Only draft orders can be confirmed.");

      if (!Lines.Any())
         throw new DomainException("Cannot confirm an empty order.");
```

```
        Status = OrderStatus.Confirmed;
        AddDomainEvent(new OrderConfirmedEvent(this));
    }

    public void Ship()
    {
        if (Status!= OrderStatus.Packed)
            throw new DomainException("Order must be packed before shipping.");

        Status = OrderStatus.Shipped;
        AddDomainEvent(new OrderShippedEvent(this));
    }
}
```

This ensures that no developer can accidentally set an order to "Shipped" without it first being "Packed," preserving process integrity.

3.2 Domain Events

Domain Events decouple side effects. When an order is confirmed, multiple independent things must happen:

1. Inventory must be reserved.
2. An email confirmation must be sent.
3. A notification must appear on the dashboard.

Instead of the OrderService calling EmailService and InventoryService directly, it simply publishes an OrderConfirmedEvent. Handlers in the Application Layer subscribe to this event and execute the logic. This is key to keeping the application maintainable.[1]

---

4. Data Access, Stored Procedures, and Concurrency

While Entity Framework Core is excellent for general data retrieval, high-concurrency inventory updates require the atomic guarantees of SQL transactions. If two users try to purchase the last item simultaneously, a standard "Read-Modify-Write" pattern in C# code will typically result in a race condition, causing the stock to drop below zero (Overselling).[23]

4.1 The Concurrency Problem

Scenario:

1. User A requests item X. System reads Stock = 1.

2. User B requests item X. System reads Stock = 1.
3. User A confirms. System sets Stock = 0.
4. User B confirms. System sets Stock = -1.

4.2 Solution: Transactional Stored Procedures with Locking

To solve this, we use a SQL Stored Procedure with **Pessimistic Locking** via the UPDLOCK hint. This forces the database to lock the inventory row for the duration of the transaction, queuing any other requests for that same row.

4.2.1 Stored Procedure: sp_AllocateInventory

SQL

```
CREATE PROCEDURE [Inventory].[sp_AllocateInventory]
   @OrderId INT,
   @WarehouseId INT
AS
BEGIN
   SET NOCOUNT ON;
   SET XACT_ABORT ON; -- Ensures auto-rollback on error [20]

   BEGIN TRANSACTION;

   BEGIN TRY
      -- Declare variables to hold line item details
      DECLARE @LineId INT, @ProductId INT, @Qty DECIMAL(18,2);

      -- Cursor or loop logic would go here to iterate over order lines.
      -- For simplicity, let's assume a single-line logic or bulk update.

      -- PESSIMISTIC LOCK: Lock the inventory rows for products in this order
      -- We select into a temp table to verify availability first
      SELECT
         i.ProductVariantId,
         i.QuantityAvailable
      INTO #StockCheck
      FROM InventoryStock i WITH (UPDLOCK, ROWLOCK) -- CRITICAL: Holds lock until commit
[6]
```

```
        JOIN OrderLines ol ON i.ProductVariantId = ol.ProductVariantId
        WHERE ol.OrderId = @OrderId AND i.WarehouseId = @WarehouseId;

        -- Validation: Check if any item lacks stock
        IF EXISTS (
          SELECT 1
          FROM OrderLines ol
          LEFT JOIN #StockCheck sc ON ol.ProductVariantId = sc.ProductVariantId
          WHERE ol.OrderId = @OrderId
          AND (sc.QuantityAvailable IS NULL OR sc.QuantityAvailable < ol.Quantity)
        )
        BEGIN
          -- Logic for partial allocation or failure
          ROLLBACK TRANSACTION;
          THROW 51000, 'Insufficient stock for one or more items.', 1;
        END

        -- Execute the Update
        UPDATE i
        SET
          i.QuantityAllocated = i.QuantityAllocated + ol.Quantity
        FROM InventoryStock i
        JOIN OrderLines ol ON i.ProductVariantId = ol.ProductVariantId
        WHERE ol.OrderId = @OrderId AND i.WarehouseId = @WarehouseId;

        -- Insert Audit Log
        INSERT INTO InventoryTransactions (Date, Type, ProductVariantId, QtyChange, RefId)
        SELECT GETDATE(), 'Allocation', ProductVariantId, 0, @OrderId
        FROM OrderLines WHERE OrderId = @OrderId;

        -- Update Order Status
        UPDATE Orders SET Status = 'Allocated' WHERE Id = @OrderId;

        COMMIT TRANSACTION;
    END TRY
    BEGIN CATCH
        IF @@TRANCOUNT > 0 ROLLBACK TRANSACTION;
        THROW;
    END CATCH
END
```

This procedure effectively serializes inventory movements for the same product, guaranteeing that stock can never be oversold. The UPDLOCK ensures that once the stock is read, no other transaction can modify it until this one completes.[24]

---

5. Security Architecture: Dynamic Role-Based Access Control (RBAC)

Enterprise security requires more flexibility than hardcoded `` attributes. Organizations need to create custom roles (e.g., "Senior Purchaser") and assign specific granular permissions (e.g., "Can Approve POs > $10k"). This requires a **Data-Driven RBAC** system.[8]

5.1 Extended Identity Schema

We extend ASP.NET Core Identity to include a Permissions concept.

- **ApplicationUser**: Inherits from IdentityUser. Adds FirstName, LastName, IsActive.
- **ApplicationRole**: Inherits from IdentityRole.
- **Permissions Table**:
  - Id (PK)
  - ClaimValue (NVARCHAR): e.g., "Permissions.Orders.Create".
  - GroupName: e.g., "Orders".
- **RolePermissions Junction Table**: Links Roles to Permissions.

5.2 Dynamic Policy Provider

To enforce these permissions without creating hundreds of policies in Program.cs, we implement a custom IAuthorizationPolicyProvider.

1. **Permission Attribute**: We create a custom attribute [AuthorizePermission("Permissions.Orders.Create")].
2. **Policy Provider**: When the app starts up, or when a request comes in, the provider intercepts the policy name. If it starts with "Permissions.", it dynamically builds a policy that requires that specific Claim.
3. **User Claims Transformation**: On login (in UserClaimsPrincipalFactory), we load all permissions associated with the user's roles and add them as Claims to the ClaimsPrincipal.

This architecture allows an Administrator to go into the UI, check a box for "Can Delete Users" for the "Manager" role, and immediately (after re-login) all Managers have that capability, with zero code changes or deployments.[7]

---

6. Implementation Module I: Vendor & Procurement

Operations

This module manages the "Inbound" supply chain. It is not just about creating POs; it is about managing the relationship and financial liability with suppliers.

6.1 Vendor Onboarding and Management

Before a PO can be raised, a vendor must be onboarded. This process involves capturing tax details, payment terms, and contact information. The schema supports multiple contacts and addresses per vendor.

- **Workflow**: New -> Pending Approval -> Active.
- **Rating System**: The system tracks OnTimeDeliveryRate and DefectRate based on GRN data to auto-calculate a vendor score (0-100).[21]

6.2 Purchase Order Workflow

1. **Requisition**: Users request items.
2. **Consolidation**: Buyers combine requisitions into a PO for a specific vendor.
3. **Pricing Engine**: The system pulls the last purchase price or a specific VendorPriceList entry to populate costs.
4. **Approval Chain**: If the PO total exceeds a threshold (defined in SystemConfigurations), the status sets to PendingApproval. A notification is sent to the ApproverRole.

6.3 Goods Receipt (GRN) and Cost Averaging

The Goods Receipt Note is the most financially significant transaction in the system. It converts "Expected Stock" into "Physical Stock" and establishes the **Cost of Goods**.
When a GRN is processed:

1. **Quantity Update**: InventoryStock.QuantityOnHand increases.
2. **Status Update**: PO Line updates to Received. If Received < Ordered, the PO remains Open (Backorder).
3. Cost Recalculation: The system updates the ProductVariant.CostPrice using the Moving Weighted Average formula:
   $$ NewCost = \frac{(CurrentQty \times CurrentCost) + (ReceivedQty \times PurchasePrice)}{CurrentQty + ReceivedQty} $$
   This ensures that inventory value on the balance sheet is accurate.27

---

7. Implementation Module II: Order Management & Fulfillment

This module manages the "Outbound" flow. It is designed to minimize friction while preventing errors like shipping the wrong item.

7.1 Allocation Strategy

Allocation is the process of reserving specific stock for an order.

- **Soft Allocation**: The stock is logically reserved but not physically moved.
- **Hard Allocation**: Specific bin locations are identified.

The sp_AllocateInventory (described in Section 4.2) handles the logic. It prioritizes stock based on:

1. **FIFO (First-In-First-Out)**: Allocates stock from the oldest receipt date to prevent obsolescence.
2. **Bin Efficiency**: Prioritizes picking from a single bin to minimize travel time.

7.2 Picking and Packing

Once allocated, the Order transitions to Picking.

1. **Pick List Generation**: The system generates a PDF or digital list sorted by BinCode (Z-path sorting) to optimize the warehouse walker's route.
2. **Packing Validation**: As items are packed, the user scans the barcode. The system validates ScannedSKU == OrderLineSKU. This step eliminates shipping errors.

7.3 Shipping and Invoicing

When the order is marked Shipped:

1. **Inventory Decrement**: The QuantityOnHand and QuantityAllocated are reduced.
2. **COGS Entry**: A transaction record is created logging the Cost of Goods Sold for financial reporting.
3. **Invoice Generation**: An invoice is generated and emailed to the customer.

---

8. Implementation Module III: Inventory Control

Beyond buying and selling, inventory requires maintenance.

8.1 Stock Adjustments and Cycle Counting

Physical reality often diverges from system records due to theft or damage.

- **Adjustments**: A manual entry (Write-off/Write-on). Requires a ReasonCode (e.g., "Damaged", "Found").
- **Cycle Counting**: Instead of shutting down for a week for annual stocktake, the system generates daily count tasks for a subset of products (e.g., "Count all A-Grade items every month, C-Grade every year").

8.2 Transfer Orders

Moving stock between Warehouses (e.g., Main Warehouse to Retail Store) uses a TransferOrder.

- **Transit Warehouse**: To prevent stock appearing in two places at once, items are moved to a virtual "Transit" location during the move.
    - Step 1 (Ship): Warehouse A -> Transit.
    - Step 2 (Receive): Transit -> Warehouse B.

---

9. Reporting and Analytics: Visualization and Insights

An EIWMS accumulates vast amounts of data. The Reporting module transforms this into actionable insights.

9.1 Technical Strategy for Reporting

Directly querying the transactional tables (like InventoryTransactions) for aggregate reports (e.g., "Sales by Month for the last 5 years") will kill performance as the dataset grows to millions of rows.

- **SQL Indexed Views**: We create views that pre-aggregate data. SQL Server maintains these physical indexes automatically.

  SQL
  ```sql
  CREATE VIEW dbo.v_MonthlySales WITH SCHEMABINDING AS
  SELECT
     YEAR(OrderDate) as Year,
     MONTH(OrderDate) as Month,
     SUM(TotalAmount) as Revenue,
     COUNT_BIG(*) as OrderCount
  ```

```
FROM dbo.Orders
GROUP BY YEAR(OrderDate), MONTH(OrderDate)
```

- **Read Replicas**: In a production environment, reports should target a secondary Read-Only node of the database to avoid locking the transactional tables used by the Order/PO workflows.[12]

9.2 Dashboard Implementation with Chart.js

For the UI, we utilize **Chart.js** to render interactive graphs. This is lightweight and avoids the complexity of PowerBI embedding for simple internal dashboards.
Workflow:

1. **Controller**: ReportsController has an action GetSalesData() that queries the v_MonthlySales view via Dapper for speed.
2. **JSON Response**: Returns a clean JSON object { labels: [...], datasets: [...] }.
3. **View**: A Razor view initializes the Chart.js canvas.

JavaScript

```
// Example Client-Side Code
var ctx = document.getElementById('revenueChart').getContext('2d');
var chart = new Chart(ctx, {
   type: 'line',
   data: {
      labels: data.labels, // ["Jan", "Feb", "Mar"]
      datasets:
      }]
   }
});
```

.[9]

10. Infrastructure, Deployment, and Best Practices

10.1 Dependency Injection Configuration

ASP.NET Core's DI container is the glue holding the Clean Architecture together.

- **Repositories**: Registered as Scoped (e.g., services.AddScoped<IInventoryRepository, InventoryRepository>()). This ensures the DbContext is reused within a single HTTP request but not shared across requests.[12]
- **Domain Services**: Registered as Transient or Scoped.
- **Infrastructure Services**: IEmailService might be Transient or Singleton depending on the implementation (e.g., HttpClient factory usage).

10.2 Audit Trails with EF Core Interceptors

To meet compliance requirements, every change to a database record must be logged. We implement a SaveChangesInterceptor in EF Core.

- **Mechanism**: Before SaveChanges commits, the interceptor inspects the ChangeTracker.
- **Action**: For every Added, Modified, or Deleted entity, it serializes the OriginalValues and CurrentValues to JSON and inserts a row into the AuditLogs table with the UserId and Timestamp.
- **Benefit**: This guarantees auditing works even if a developer forgets to write audit code in their service.[28]

10.3 Testing Strategy

- **Unit Tests (xUnit + Moq)**: Focus on the Domain and Application layers. Test that Order.Confirm() throws an exception if lines are empty. Mock the Repositories.
- **Integration Tests**: Use WebApplicationFactory to spin up the full MVC stack in memory. Use a real SQL Server (Docker container) for these tests to verify the Stored Procedures and unique constraints, as the "InMemory" provider behaves differently than SQL Server.[30]

Conclusion

Building an Enterprise Inventory and Workflow Management System is a significant undertaking that demands strict adherence to architectural principles. By following this guide, the resulting system will possess:

1. **Structural Integrity**: Through Clean Architecture, ensuring the business logic remains isolated and testable.
2. **Data Reliability**: Through rigorous normalization and the use of Stored Procedures for concurrent transaction handling.
3. **Operational Flexibility**: Through a workflow engine and dynamic RBAC system that adapts to organizational changes.

4. **Actionable Intelligence**: Through performant reporting views and integrated visualizations.

This blueprint provides the "North Star" for the development team. The implementation should proceed iteratively, starting with the Domain Entities and Database Schema (the foundation), followed by the Repositories and Application Logic (the structure), and finally the MVC Interface (the skin). This specific order of operations minimizes rework and ensures that the system's core is solid before the first user interface element is rendered.

Works cited

1. The Best Way To Structure Your .NET Projects with Clean Architecture and Vertical Slices, accessed on November 23, 2025, https://antondevtips.com/blog/the-best-way-to-structure-your-dotnet-projects-with-clean-architecture-and-vertical-slices
2. Clean Architecture in .NET 10: Patterns That Actually Work in Production (2025 Guide), accessed on November 23, 2025, https://dev.to/nikhilwagh/clean-architecture-in-net-10-patterns-that-actually-work-in-production-2025-guide-36b0
3. SQL Server Database Normalization Techniques - DbSchema, accessed on November 23, 2025, https://dbschema.com/blog/sql-server/database-normalization/
4. What Is Database Normalization? - IBM, accessed on November 23, 2025, https://www.ibm.com/think/topics/database-normalization
5. Using Stored Procedures in Transaction Management in SQL - SheCodes, accessed on November 23, 2025, https://www.shecodes.io/athena/264543-using-stored-procedures-in-transaction-management-in-sql
6. Can you force a SQL Stored Procedure to take locks in a given order - Stack Overflow, accessed on November 23, 2025, https://stackoverflow.com/questions/1397487/can-you-force-a-sql-stored-procedure-to-take-locks-in-a-given-order
7. Building Secure APIs with Role-Based Access Control in ASP.NET Core - Milan Jovanović, accessed on November 23, 2025, https://www.milanjovanovic.tech/blog/building-secure-apis-with-role-based-access-control-in-aspnetcore
8. Dynamic Authorization Policies in ASP | by iamprovidence | Medium, accessed on November 23, 2025, https://medium.com/@iamprovidence/dynamic-authorization-policies-in-asp-70019dc69e7d
9. Step-by-step guide - Chart.js, accessed on November 23, 2025, https://www.chartjs.org/docs/latest/getting-started/usage.html
10. Creating Charts With ASP.NET Core, accessed on November 23, 2025, https://www.c-sharpcorner.com/article/creating-charts-with-asp-net-core/
11. Clean Architecture in .NET Core | Scalable C# Design | by Jenil Sojitra - Medium, accessed on November 23, 2025,

https://medium.com/@jenilsojitra/clean-architecture-in-net-core-e18b4ad229c8
12. Best Practices for Structuring Large ASP.NET Projects: A Simple Guide - C# Corner, accessed on November 23, 2025, https://www.c-sharpcorner.com/article/best-practices-for-structuring-large-asp-net-projects-a-simple-guide/
13. Management data warehouse - SQL Server | Microsoft Learn, accessed on November 23, 2025, https://learn.microsoft.com/en-us/sql/relational-databases/data-collection/management-data-warehouse?view=sql-server-ver17
14. Using stored procedures for repository pattern with Service Class - Stack Overflow, accessed on November 23, 2025, https://stackoverflow.com/questions/51110410/using-stored-procedures-for-repository-pattern-with-service-class
15. .net core API and stored procedures : r/dotnet - Reddit, accessed on November 23, 2025, https://www.reddit.com/r/dotnet/comments/1cf9fjc/net_core_api_and_stored_procedures/
16. Best structure for inventory database - Stack Overflow, accessed on November 23, 2025, https://stackoverflow.com/questions/4380091/best-structure-for-inventory-database
17. Database design for an online store with highly variable product attributes? : r/SQL - Reddit, accessed on November 23, 2025, https://www.reddit.com/r/SQL/comments/1n4yrg8/database_design_for_an_online_store_with_highly/
18. E-Commerce Database Design: Managing Product Variants for Multi-Vendor Platforms Using the EAV Model - Amit Singh, accessed on November 23, 2025, https://np4652.medium.com/e-commerce-database-design-managing-product-variants-for-multi-vendor-platforms-using-the-eav-01307e63b920
19. Weekly DB Project #1: Inventory Management DB Design & Seed -From Schema Design to Performance Optimization | by Bhargava Koya - Fullstack .NET Developer | Medium, accessed on November 23, 2025, https://medium.com/@bhargavkoya56/weekly-db-project-1-inventory-management-db-design-seed-from-schema-design-to-performance-8e6b56445fe6
20. The Art Of The SQL Server Stored Procedure: Transactions - Darling Data, accessed on November 23, 2025, https://erikdarling.com/the-art-of-the-sql-server-stored-procedure-transactions/
21. Section 5 – Vendor and System Requirements - CivicLive, accessed on November 23, 2025, https://cdnsm5-hosted.civiclive.com/UserFiles/Servers/Server_17385004/File/Departments/Finance/Section%205_Vendor%20and%20System%20Requirements.pdf
22. Appendix A - Functional Specs, accessed on November 23, 2025, https://www.ndb.int/wp-content/uploads/2018/01/NDB-e-Procurement-system-R

FP-Functional-Requirement.pdf
23. How to Handle Concurrent Order Placement Requests | by Suresh kumar | Medium, accessed on November 23, 2025, https://medium.com/@veerdhakad4568/how-to-handle-concurrent-order-placement-requests-32810b50f4cb
24. Stored procedure to constantly check table for records and process them - Stack Overflow, accessed on November 23, 2025, https://stackoverflow.com/questions/71300324/stored-procedure-to-constantly-check-table-for-records-and-process-them
25. Protecting your API endpoints with dynamic policies in ASP.NET ..., accessed on November 23, 2025, https://blog.joaograssi.com/posts/2021/asp-net-core-protecting-api-endpoints-with-dynamic-policies/
26. Implement role-based access control in applications - Microsoft identity platform, accessed on November 23, 2025, https://learn.microsoft.com/en-us/entra/identity-platform/howto-implement-rbac-for-apps
27. Product receipt against purchase orders - Supply Chain Management | Dynamics 365, accessed on November 23, 2025, https://learn.microsoft.com/en-us/dynamics365/supply-chain/procurement/product-receipt-against-purchase-orders
28. EF Core Interceptors: SaveChangesInterceptor for Auditing Entities in .NET 8 Microservices, accessed on November 23, 2025, https://mehmetozkaya.medium.com/ef-core-interceptors-savechangesinterceptor-for-auditing-entities-in-net-8-microservices-6923190a03b9
29. Tracking Every Change: Using SaveChanges Interception for EF Core Auditing, accessed on November 23, 2025, https://www.woodruff.dev/tracking-every-change-using-savechanges-interception-for-ef-core-auditing/
30. Testing ASP.NET Core services and web apps - Microsoft Learn, accessed on November 23, 2025, https://learn.microsoft.com/en-us/dotnet/architecture/microservices/multi-container-microservice-net-applications/test-aspnet-core-services-web-apps