GitHub Copilot Instructions

This document provides comprehensive guidance for GitHub Copilot to understand and work effectively with the Demo Inventory Microservice project.

Project Overview

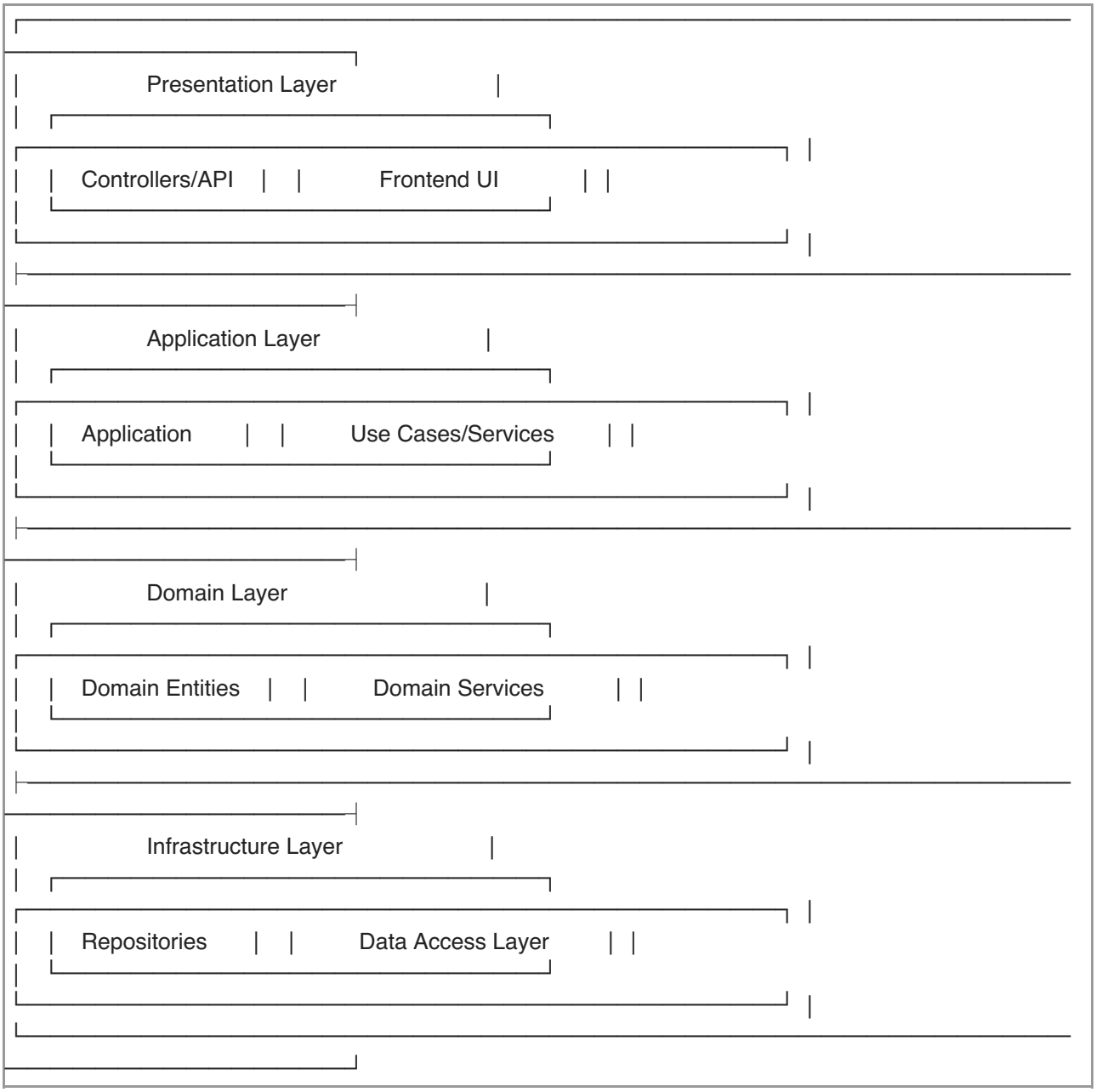
This is a **modern full-stack inventory management microservice** demonstrating Clean Architecture principles with:

- **.NET 9** backend with Clean Architecture layers
- **React 19 + TypeScript** frontend with modern tooling
- **PostgreSQL** database with Entity Framework Core
- **Comprehensive testing** strategy (Unit, API, E2E)
- **Docker containerization** and CI/CD pipeline

Architecture & Design Patterns

Clean Architecture Implementation

The project follows **Clean Architecture** with strict separation of concerns:



Layer Responsibilities:

- **Domain**: Core business logic, entities, value objects, domain services
- **Application**: Use cases, DTOs, application services, interfaces
- **Infrastructure**: Data access, external APIs, persistence, EF Core
- **Presentation**: Controllers, API endpoints, frontend UI components

Key Design Principles

1. **Dependency Inversion**: Dependencies flow inward toward the domain
2. **Interface Segregation**: Use specific interfaces for different concerns
3. **Single Responsibility**: Each class/method has one reason to change
4. **Domain-Driven Design**: Rich domain models with business logic
5. **CQRS Pattern**: Separate read and write operations where appropriate

Technology Stack

Backend (.NET 9)

- **ASP.NET Core Web API**: RESTful API with OpenAPI/Swagger
- **Entity Framework Core**: ORM with PostgreSQL provider
- **Dependency Injection**: Built-in .NET DI container
- **Logging**: Built-in ILogger with structured logging
- **Validation**: FluentValidation for input validation
- **Testing**: xUnit, NSubstitute, FluentAssertions

Frontend (React 19)

- **React 19**: Modern functional components with hooks
- **TypeScript**: Strict type safety and modern ES features
- **Vite**: Fast build tool and dev server
- **Axios**: HTTP client for API communication
- **React Router**: Client-side routing
- **Testing**: Vitest, React Testing Library

Database & Infrastructure

- **PostgreSQL 13+**: Primary database
- **Docker**: Containerization with multi-stage builds
- **Docker Compose**: Multi-container orchestration
- **GitHub Actions**: CI/CD pipeline with automated testing

Code Standards & Best Practices

Backend (.NET/C#) Standards

Naming Conventions

- **PascalCase**: Classes, methods, properties, interfaces
- **camelCase**: Private fields, local variables, method parameters
- **Interface prefix**: Use I prefix (e.g., IProductRepository)
- **Async suffix**: Add Async to async method names

Code Style Example

```
public class ProductService : IProductService
{
    private readonly IProductRepository _productRepository;
    private readonly ILogger<ProductService> _logger;

    public ProductService(
        IProductRepository productRepository,
        ILogger<ProductService> logger)
    {
        _productRepository = productRepository ?? throw new
ArgumentNullException(nameof(productRepository));
        _logger = logger ?? throw new ArgumentNullException(nameof(logger));
    }

    /// <summary>
    /// Retrieves all products from the inventory.
    /// </summary>
    /// <returns>A collection of products.</returns>
    public async Task<IEnumerable<ProductDto>> GetAllProductsAsync()
    {
        _logger.LogInformation("Retrieving all products");

        var products = await _productRepository.GetAllAsync();

        return products.Select(p => new ProductDto
        {
            Id = p.Id,
            Name = p.Name,
            SKU = p.SKU,
            Price = p.Price,
            StockQuantity = p.StockQuantity
        });
    }
}
```

Architecture Guidelines

- **Domain purity**: Keep domain layer free of external dependencies
- **Async/await**: Use for all I/O operations
- **Error handling**: Use Result pattern or proper exception handling
- **Dependency injection**: Register services with appropriate lifetimes
- **Repository pattern**: Abstract data access behind interfaces

Frontend (React/TypeScript) Standards

Component Standards

- **Functional components**: Use with hooks, avoid class components
- **TypeScript interfaces**: Define props and state types
- **Error boundaries**: Implement for error handling
- **Custom hooks**: Extract reusable logic
- **Proper state management**: Use React hooks appropriately

Code Style Example

```
interface ProductListProps {
  searchTerm?: string;
  onProductSelect: (product: Product) => void;
}

export const ProductList: React.FC<ProductListProps> = ({
  searchTerm,
  onProductSelect
}) => {
  const [products, setProducts] = useState<Product[]>([]);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState<string | null>(null);

  useEffect(() => {
    const fetchProducts = async () => {
      try {
        setLoading(true);
        const response = await productApi.getProducts(searchTerm);
        setProducts(response.data);
      } catch (err) {
        setError('Failed to fetch products');
      } finally {
        setLoading(false);
      }
    };

    fetchProducts();
  }, [searchTerm]);

  if (loading) return <div>Loading...</div>;
  if (error) return <div>Error: {error}</div>;

  return (
    <div className="product-list">
      {products.map(product => (
        <ProductCard
          key={product.id}
          product={product}
          onClick={() => onProductSelect(product)}
        />
      ))}
    </div>
  );
};
```

Build, Test & Validation

Building the Application

Backend Build

```
# Restore dependencies
dotnet restore

# Build solution
dotnet build

# Build specific project
dotnet build backend/src/DemoInventory.API

# Build for production
dotnet publish -c Release -o ./publish
```

Frontend Build

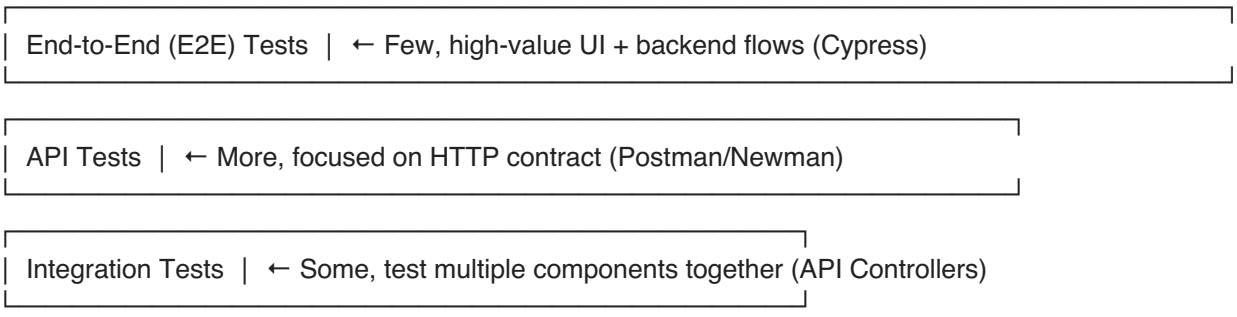
```
# Install dependencies
npm install

# Development build
npm run dev

# Production build
npm run build

# Lint code
npm run lint

# Preview production build
npm run preview
```

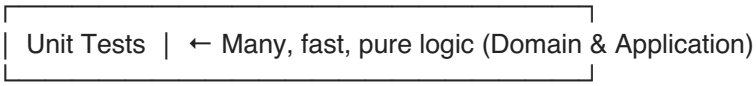
Testing Strategy

The project implements a comprehensive testing pyramid:

Testing Pyramid

The project uses a layered testing strategy, often called the "testing pyramid", to ensure quality and reliability at all levels:





- **Unit Tests**: Test individual functions/classes in isolation (most numerous, fastest).
- **Integration Tests**: Test how multiple components work together (e.g., controller + service + repo).
- **API Tests**: Test the HTTP API contract and business flows.
- **E2E Tests**: Simulate real user scenarios through the UI, covering the full stack.

This approach ensures fast feedback for core logic, confidence in integration, and real-world validation of the system.

Unit Testing (Backend)

```
# Run all tests
dotnet test

# Run specific test project
dotnet test backend/tests/DemoInventory.Domain.Tests

# Run with coverage
dotnet test --collect:"XPlat Code Coverage"

# Run tests in watch mode
dotnet watch test
```

Unit Test Guidelines:

- Use **xUnit** as testing framework
- Use **NSubstitute** for mocking
- Use **FluentAssertions** for readable assertions
- Follow **AAA pattern** (Arrange, Act, Assert)
- Test behavior, not implementation details
- Achieve high coverage for domain and application layers

Frontend Testing

```
# Run unit tests
npm test

# Run tests with UI
npm run test:ui

# Run tests with coverage
npm run test:coverage

# Run tests in watch mode
npm test -- --watch
```

API Testing

```
# Auto-detect environment and run
cd tests/postman && ./run-newman.sh

# Run against local development
./run-newman.sh local

# Run against Docker environment
./run-newman.sh docker
```

End-to-End Testing

```
# Local development (requires API and frontend running)
cd tests/e2e && npm run test:e2e

# Docker environment
npm run test:e2e:docker

# Interactive mode
npm run cypress:open
```

Development Environment Setup

Local Development

1. **Prerequisites**: .NET 9 SDK, Node.js 20+, PostgreSQL 13+
2. **Database**: Create demo_inventory database
3. **Backend**: dotnet run --project backend/src/DemoInventory.API
4. **Frontend**: cd frontend && npm run dev

Docker Development

```
# Start complete stack
docker-compose up -d

# Start only database
docker-compose up -d db

# View logs
docker-compose logs -f

# Rebuild specific service
docker-compose build api && docker-compose up -d api
```

Common Development Tasks

Adding New Features

1. Domain-First Approach

```
// 1. Create domain entity
public class Category : BaseEntity
{
    public string Name { get; private set; }
    public string Description { get; private set; }

    public Category(string name, string description)
    {
        Name = name ?? throw new ArgumentNullException(nameof(name));
        Description = description;
    }
}

// 2. Add repository interface
public interface ICategoryRepository : IRepository<Category>
{
    Task<Category?> GetByNameAsync(string name);
}

// 3. Create application service
public class CategoryService : ICategoryService
{
    // Implementation with business logic
}

// 4. Add API controller
[ApiController]
[Route("api/[controller]")]
public class CategoriesController : ControllerBase
{
    // RESTful endpoints
}
```

2. Frontend Feature Development

```
// 1. Define types
interface Category {
  id: string;
  name: string;
  description: string;
}

// 2. Create API service
export const categoryApi = {
  getCategories: () => axios.get<Category[]>('/api/categories'),
  createCategory: (category: CreateCategoryDto) =>
    axios.post<Category>('/api/categories', category),
};

// 3. Create React components
export const CategoryList: React.FC = () => {
  // Component implementation
};
```

Database Migrations

```
# Create migration
dotnet ef migrations add AddCategoryTable --project backend/src/DemoInventory.Infrastructure

# Update database
dotnet ef database update --project backend/src/DemoInventory.Infrastructure

# Generate migration script
dotnet ef migrations script --project backend/src/DemoInventory.Infrastructure
```

Commit Message Standards

Use **Conventional Commits** format:

```
<type>[optional scope]: <description>

[optional body]

[optional footer(s)]
```

Types:

- feat: New feature
- fix: Bug fix
- docs: Documentation changes
- style: Code style changes
- refactor: Code refactoring
- perf: Performance improvements
- test: Test changes
- chore: Build/tool changes

Examples:

```
feat(api): add product categories endpoint

Add new endpoint for managing product categories including
CRUD operations and category assignment to products.

Closes #123
```

Tools & Libraries Reference

Backend Dependencies

- **Microsoft.EntityFrameworkCore**: ORM framework
- **Npgsql.EntityFrameworkCore.PostgreSQL**: PostgreSQL provider
- **Swashbuckle.AspNetCore**: OpenAPI/Swagger generation
- **FluentValidation**: Input validation
- **Serilog**: Structured logging
- **AutoMapper**: Object-to-object mapping

Frontend Dependencies

- **React**: UI library (functional components + hooks)
- **TypeScript**: Type safety and modern JavaScript
- **Axios**: HTTP client for API calls
- **React Router**: Client-side routing
- **Vite**: Build tool and development server

Development Tools

- **Visual Studio 2022** or **VS Code**: Primary IDEs
- **Docker Desktop**: Containerization
- **Postman**: API testing and documentation
- **pgAdmin**: PostgreSQL administration
- **Newman**: Command-line Postman runner
- **Cypress**: End-to-end testing

Error Handling & Logging

Backend Error Handling

```
// Use Result pattern for business logic
public async Task<Result<ProductDto>> GetProductByIdAsync(Guid id)
{
    try
    {
        var product = await _repository.GetByIdAsync(id);
        if (product == null)
            return Result<ProductDto>.Failure("Product not found");

        return Result<ProductDto>.Success(MapToDto(product));
    }
    catch (Exception ex)
    {
        _logger.LogError(ex, "Error retrieving product {Id}", id);
        return Result<ProductDto>.Failure("Internal server error");
    }
}

// Global exception handling in controllers
[HttpGet("{id}")]
public async Task<ActionResult<ProductDto>> GetProduct(Guid id)
{
    var result = await _productService.GetProductByIdAsync(id);

    return result.IsSuccess
        ? Ok(result.Value)
        : BadRequest(result.Error);
}
```

Frontend Error Handling

```
// Error boundary for React components
export class ErrorBoundary extends React.Component {
  // Implementation
}

// API error handling
const handleApiError = (error: AxiosError) => {
  if (error.response?.status === 404) {
    // Handle not found
  } else if (error.response?.status >= 500) {
    // Handle server errors
  }
};
```

Performance & Optimization

Backend Optimization

- Use **async/await** for all I/O operations
- Implement **caching** for frequently accessed data

- Use **pagination** for large data sets
- Optimize **database queries** with proper indexing
- Enable **response compression**

Frontend Optimization

- Use **React.memo** for expensive components
- Implement **code splitting** with lazy loading
- Optimize **bundle size** with tree shaking
- Use **React Query** or similar for data fetching
- Implement **virtual scrolling** for large lists

Security Considerations

Backend Security

- **Input validation**: Validate all user inputs
- **SQL injection prevention**: Use parameterized queries
- **CORS configuration**: Configure appropriate CORS policies
- **HTTPS**: Use HTTPS in production
- **Authentication**: Implement JWT or similar

Frontend Security

- **XSS prevention**: Sanitize user inputs
- **CSRF protection**: Use appropriate tokens
- **Content Security Policy**: Implement CSP headers
- **Dependency scanning**: Regular security audits

Troubleshooting Guide

Common Backend Issues

- **Port conflicts**: Check if port 5126 is available
- **Database connection**: Verify PostgreSQL connection string
- **Missing dependencies**: Run dotnet restore
- **Build errors**: Check .NET version compatibility

Common Frontend Issues

- **Node version**: Ensure Node.js 20+ is installed
- **Dependency conflicts**: Delete node_modules, run npm install
- **CORS errors**: Verify API CORS configuration
- **Environment variables**: Check .env file setup

Docker Issues

- **Container startup**: Check Docker daemon status
- **Port conflicts**: Ensure ports aren't already in use
- **Volume mounts**: Check file permissions and paths

Additional Resources

- **Architecture Guide (../docs/ARCHITECTURE.md)**: Detailed system architecture
- **Development Guide (../docs/DEVELOPMENT.md)**: Setup and development workflow
- **Testing Guide (../docs/TESTING.md)**: Testing strategy and best practices
- **Contributing Guidelines (../docs/CONTRIBUTING.md)**: How to contribute

---

Remember: Always follow the existing patterns and conventions in the codebase. When in doubt, refer to similar implementations within the project or ask for clarification in the issue or pull request.