**Lab 2 Objective Questions and Answers**

**Question 1: What is the purpose of a multi-layered architecture in EF Core projects?**

**Answer:**  
A multi-layered architecture separates concerns into distinct projects, each with specific responsibilities[[1]](#fn1)[[2]](#fn2):

* **Domain Layer (RetailInventory.Domain)**: Contains entity models, business logic, and domain rules
* **Data Layer (RetailInventory.Data)**: Contains DbContext, configurations, migrations, and data access logic
* **Application Layer (RetailInventory.App)**: Contains the entry point, dependency injection setup, and application orchestration

**Benefits:**

* **Separation of Concerns**: Each layer has a single responsibility
* **Maintainability**: Changes in one layer don't affect others
* **Testability**: Each layer can be unit tested independently
* **Reusability**: Domain models can be used across different applications
* **Scalability**: Easy to add new features without affecting existing code

**Question 2: How do project references work in a multi-layered solution?**

**Answer:**  
Project references establish dependencies between projects in a solution:

**Reference Hierarchy:**

* **App Layer** → References both Domain and Data layers
* **Data Layer** → References Domain layer only
* **Domain Layer** → Has no dependencies (pure business logic)

**Implementation:**

# App references both Domain and Data  
dotnet add reference ../RetailInventory.Domain/RetailInventory.Domain.csproj  
dotnet add reference ../RetailInventory.Data/RetailInventory.Data.csproj  
  
# Data references Domain  
dotnet add reference ../RetailInventory.Domain/RetailInventory.Domain.csproj

**Benefits:**

* **Dependency Inversion**: Higher-level modules don't depend on lower-level modules
* **Compile-time Safety**: References are validated at build time
* **IntelliSense Support**: Full IDE support across projects
* **Namespace Organization**: Clear separation of concerns through namespaces

**Question 3: What is the role of dependency injection in EF Core applications?**

**Answer:**  
Dependency Injection (DI) is a design pattern that provides dependencies to classes rather than having them create dependencies themselves:

**In EF Core Context:**

* **DbContext Registration**: Register DbContext with connection string in DI container
* **Service Lifetime Management**: Control how long DbContext instances live
* **Configuration Management**: Inject configuration services for connection strings
* **Testability**: Easy to mock DbContext for unit testing

**Implementation Example:**

var services = new ServiceCollection();  
services.AddDbContext<RetailDbContext>(options =>  
 options.UseSqlServer(configuration.GetConnectionString("DefaultConnection")));  
  
var serviceProvider = services.BuildServiceProvider();  
using var context = serviceProvider.GetRequiredService<RetailDbContext>();

**Benefits:**

* **Loose Coupling**: Classes don't directly instantiate their dependencies
* **Configuration Centralization**: All service configuration in one place
* **Lifetime Management**: Automatic disposal of resources
* **Testing Support**: Easy to substitute mock implementations