Goal

To **securely configure your database connection** without using appsettings.json, instead leveraging **environment variables**, in a way that is:

- Clean Architecture–compliant
- Testable
- Scalable
- Suitable for modern deployments like Docker, Azure, and Kubernetes

Project Structure Overview

This implementation replaces traditional `appsettings.json`-based configuration using the following components:

Component	Responsibility
IConnection	Interface defining the contract for fetching connection strings
Connection	Concrete implementation that reads environment variables
ConnectionString	Static helper to build a valid SQL Server connection string
MigrationConfig	Fallback configuration for development/test defaults
InfrastructureConfigurator	Registers DB context and services in IServiceCollection

Implementation Details

1. `IConnection` Interface

```
public interface IConnection
{
    string GetUserManagementConnectionString();
}
```

Responsibility:

Defines the contract for retrieving the **User Management Database** connection string. This promotes dependency injection and testability.

2. `Connection` Class (Environment-Based Provider)

```
public class Connection : IConnection
{
    private readonly string? _server;
    private readonly string? _database;
    private readonly string? _userId;
    private readonly string? _password;

    public Connection()
    {
        _server = Environment.GetEnvironmentVariable("DB_SERVER", EnvironmentVariableTarget.Process);
        _database = Environment.GetEnvironmentVariable("DB_NAME", EnvironmentVariableTarget.Process);
        _userId = Environment.GetEnvironmentVariable("DB_USER", EnvironmentVariableTarget.Process);
        _password = Environment.GetEnvironmentVariable("DB_PASSWORD",
EnvironmentVariableTarget.Process);
    }

    public string GetUserManagementConnectionString()
    {
        return ConnectionString.GetConnectionString(
            _server ?? MigrationConfig.Server,
            _database ?? MigrationConfig.UserManagement,
            _userId ?? MigrationConfig.UserId,
            _password ?? MigrationConfig.Password
        );
    }

}
```

Responsibility:

- Fetch connection values from **environment variables**.
- Fall back to values in `MigrationConfig` during development/testing.
- Returns a formatted and validated SQL Server connection string.

3. `ConnectionString` Helper

```
public static class ConnectionString
{
    public static string GetConnectionString(string server, string dbName, string userId, string
password)
    {
        server = string.IsNullOrEmpty(server) ? MigrationConfig.Server : server;
        dbName = string.IsNullOrEmpty(dbName) ? MigrationConfig.UserManagement : dbName;
        userId = string.IsNullOrEmpty(userId) ? MigrationConfig.UserId : userId;
        password = string.IsNullOrEmpty(password) ? MigrationConfig.Password : password;

        return Build(server, dbName, userId, password);
    }
    private static string Build(string server, string dbName, string user, string password)
    {
        try
        {
            if (!string.IsNullOrEmpty(user) && !string.IsNullOrEmpty(password))
            {
                // SQL Authentication
                return $"Server={server};Database={dbName};User
Id={user};Password={password};Trusted_Connection=False;MultipleActiveResultSets=True;TrustServerCertif
icate=True;";
            }
            else
            {
                // Windows/Integrated Authentication
                return
$"Server={server};Database={dbName};Trusted_Connection=True;MultipleActiveResultSets=True;TrustServerC
ertificate=True;";
            }
        }
        catch (Exception ex)
        {
            throw new Exception($"Error creating connection string: {ex.Message}", ex);
        }
    }

}
```

Responsibility:

- Centralized helper to build SQL Server connection strings.
- Supports both SQL and Windows Authentication.
- Handles errors clearly with exception feedback.

4. `MigrationConfig` Fallback (Optional)

```
public static class MigrationConfig
{
    public static string Server = ".";
    public static string UserManagement = "UserManagement";
    public static string UserId = "";
    public static string Password = "";

}
```

Responsibility:

- Provides fallback/default values **only during development**.
- Used when environment variables are **not set**.
- Keeps sensitive data out of your codebase or config files.

5. InfrastructureConfigurator

```
public static class InfrastructureConfigurator
{
    public static void ConfigureServices(IServiceCollection services)
    {
        services.AddScoped<IConnection, Connection>();

        RegisterUserManagementDbContext<UserManagementDbContext>(services);

        services.AddScoped<IUserService<ApplicationUser>, UserService>();

    }

    private static void RegisterUserManagementDbContext<TContext>(IServiceCollection services) where 
TContext : DbContext
    {
        services.AddDbContext<TContext>((serviceProvider, options) =>
        {
            var connectionProvider = serviceProvider.GetRequiredService<IConnection>();
            var connectionString = connectionProvider.GetUserManagementConnectionString();
            options.UseSqlServer(connectionString);
        });
    }
}
```

Responsibility:

- Centralized configuration of services and database context.
- Isolates all infrastructure registrations from API/Application layers.
- Fully supports **Clean Architecture** via **Dependency Injection**.

Use in Program.cs (API Layer Only)

In your **API project**, call:

```
InfrastructureConfigurator.ConfigureServices(builder.Services);
```

Note: Do **not** put this in Application or Domain layers.

Advantages of This Approach

1. Security

- Credentials are **never hardcoded** or stored in `appsettings.json`.
- Environment variables are **CI/CD and DevOps–friendly**.
- Reduces risk of leaking secrets into source control.

2. Deployment-Friendly

- Seamless integration with:
  - **Azure App Service (App Settings)**
  - **Docker (`ENV` variables)**
  - **Kubernetes (Secrets and ConfigMaps)**

3. Clean Architecture–Compatible

- Infrastructure concerns (e.g., DB connection) are **completely separated** from:
  - Domain Layer
  - Application Layer
- Promotes **testability, modularity, and scalability**.

4. Multi-Database Ready

- You can easily extend `IConnection` for:
  - `GetAuditDbConnectionString()`
  - `GetReportingDbConnectionString()`
  - `GetTenantDbConnectionString(string tenantId)`

5. Testability & Maintainability

- Use **mock implementations** of `IConnection` for testing.
- No config files or local secrets needed for unit tests.
- Supports full **inversion of control** (IoC).

How to Set Environment Variables

PowerShell (Local Dev):

```
powershell

$env:DB_SERVER = "localhost"
$env:DB_NAME = "UserManagement"
$env:DB_USER = "sa"
$env:DB_PASSWORD = "YourStrongPassword123"
```

Dockerfile:

```
dockerfile

ENV DB_SERVER=sql-server
ENV DB_NAME=UserManagement
ENV DB_USER=sa
ENV DB_PASSWORD=YourStrongPassword123
```