# Infos

If you need flexibility to work with a wide range of data types but are willing to trade off type safety and compile-time checking, you might use **object.**

If you want to write type-safe and reusable code that can work with different data types without sacrificing the benefits of compile-time checking, you should use **generics (T).**

// Creating a tuple with two elements (int and string)  
Saving to DataType in one Parameter

Tuple<int, string> myTuple = new Tuple<int, string>(42, "Hello");

# Topics

## Reading Json Datei:

**Installtion:**

1. Microsoft.Extensions.Configuration
2. Microsoft.Extensions.Configuration.Json

**Code Snpits:**

public class DataBase

{

public string ConnectionString { get; set; }

public DataBase()

{  
 IConfigurationRoot configuration = new ConfigurationBuilder()

.SetBasePath(Directory.GetCurrentDirectory())

.AddJsonFile("appsettings.json")

.Build();

ConnectionString = configuration.GetConnectionString("DBCredential");

}

}

## Working with Entity Framework Core (Mysql Server):

**Step 1 Installtion:**

1. MySql.Data.EntityFrameworkCore
2. Pomelo.EntityFrameworkCore.MySql
3. Microsoft.EntityFrameworkCore.Tools

**Step 2** Define Your Data Model (Code-First Approach)

And then Create a Database Diractory

**Step 3** Create a DbContext

**Code Snpits:**  
public class ModelDbContext: DbContext

{

public string ConnectionString { get; set; }

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

IConfigurationRoot configuration = new ConfigurationBuilder()

.SetBasePath(Directory.GetCurrentDirectory())

.AddJsonFile("appsettings.json")

.Build();

ConnectionString = configuration.GetConnectionString("DBCredentialEFW");

ServerVersion serverVersion = ServerVersion.AutoDetect(ConnectionString);

//Inject TrustServerCertificate=True to

optionsBuilder.UseMySql(ConnectionString, serverVersion);

}

}

**Step 4** Create Database Migrations (Code-First Only)  
  
Add-Migration InitialCreate

Update-Database

# Code Snippets

To set a Property as a Primary Key be only adding a Attribute EFW:

public class ArticleModel

{

[Key]

public int Number { get; set; }

}

# Directory Architecture

Console Applection  
  
MyConsoleApp/

├── MyConsoleApp.csproj # Project file

├── Program.cs # Main entry point

├── Controllers/ # Custom classes or modules (optional)

├── Models/ # Data models (optional)

├── Services/ # Business logic or services (optional)

├── Data Access/ # Data storage (e.g., database files)

├── Utilities/ # Utility classes and helper methods

├── Config/ # Configuration settings (e.g., appsettings.json)

├── Tests/ # Unit tests (if using a testing framework)

├── Output/ # Directory for storing generated files or logs

├── Dependencies/ # External libraries and dependencies

├── Documentation/ # Documentation files (e.g., README.md)

└── bin/ # Compiled output (generated by build)

# Unit Test

Step 1: Create a unit test project  
Step 2: Add a Reference to the Project to Be Tested  
Step 3: Write Your Test Class/Methods [TestClass]/ [TestMethod]  
Step 4: Run Your Tests  
  
  
**Test method requirements**

A test method must meet the following requirements:

* It's decorated with the [TestMethod] attribute.
* It returns void.
* It cannot have parameters.

Die SOLID-Prinzipien   
sind eine Reihe von fünf grundlegenden Designprinzipien in der objektorientierten Softwareentwicklung, die entwickelt wurden, um den Aufbau und die Wartung von Software zu erleichtern. Die SOLID-Prinzipien helfen dabei, Code flexibler, verständlicher und wartbarer zu gestalten. Hier sind die SOLID-Prinzipien auf einfache Weise erklärt:

S - Single Responsibility Principle (SRP - Prinzip der einzelnen Verantwortung): Eine Klasse sollte nur einen Grund zur Änderung haben. Das bedeutet, dass eine Klasse nur eine Aufgabe oder Verantwortlichkeit haben sollte. Wenn eine Klasse zu viele Dinge tut, wird der Code schwerer verständlich und wartbar.

O - Open/Closed Principle (OCP - Prinzip der offenen/geschlossenen Erweiterung): Softwareeinheiten (Klassen, Module, Funktionen usw.) sollten offen für Erweiterungen, aber geschlossen für Änderungen sein. Das bedeutet, dass Sie neue Funktionalitäten hinzufügen können, ohne den bestehenden Code zu ändern. Dies wird oft durch Verwendung von Abstraktion und Schnittstellen erreicht.

L - Liskov Substitution Principle (LSP - Prinzip der Liskovschen Substitution): Objekte einer abgeleiteten Klasse sollten in der Lage sein, anstelle von Objekten der Basisklasse verwendet zu werden, ohne dass die Integrität des Programms gefährdet wird. Kurz gesagt, abgeleitete Klassen sollten die Verhalten der Basisklasse nicht verletzen oder ändern.

I - Interface Segregation Principle (ISP - Prinzip der Schnittstellentrennung): Es ist besser, viele spezifische Schnittstellen zu haben, als eine allgemeine, um sicherzustellen, dass Klienten nur die Methoden sehen, die sie benötigen. Dies verhindert, dass Klassen unnötige Abhängigkeiten von ungenutzten Methoden haben.

D - Dependency Inversion Principle (DIP - Prinzip der Abhängigkeitsumkehr): Hochrangige Module sollten nicht von niedrigrangigen Modulen abhängen, sondern beide sollten von abstrakten Schnittstellen abhängen. Dies fördert die Entkopplung und Flexibilität im Code. Abstraktionen sollten nicht von Details abhängen, sondern Details sollten von Abstraktionen abhängen.

Code First Approch  
  
**Step 1 Installtion:**

1. MySql.Data.EntityFrameworkCore
2. Pomelo.EntityFrameworkCore.MySql
3. Microsoft.EntityFrameworkCore.Tools

Step 2 :

Write this code in your Console and change the Values

For Only One Table :  
Scaffold-DbContext "data source=116.202.213.29;initial catalog=weinmann\_faktura ; persist security info=True;user id=techsample; password=7w6Xvm4cH2cXX7q8EeBU4t; TrustServerCertificate=True" Microsoft.EntityFrameworkCore.SqlServer -OutputDir DataAccess\Weinmann\ScaffoldingTables -Tables Mitarbeiter\_Daten

For many One Table :

Scaffold-DbContext "data source=116.202.213.29;initial catalog=weinmann\_faktura ; persist security info=True;user id=techsample; password=7w6Xvm4cH2cXX7q8EeBU4t; TrustServerCertificate=True" Microsoft.EntityFrameworkCore.SqlServer -OutputDir DataAccess\Weinmann\ScaffoldingTables -Tables Mitarbeiter\_Daten,Plugin\_10340\_service\_repair\_reports  
  
  
Scaffold-DbContext "data source=192.168.110.182;initial catalog=Sami\_Test ; persist security info=True; persist security info=True;user id=sa; password=onemorebyte2021!?; TrustServerCertificate=True"

Microsoft.EntityFrameworkCore.SqlServer -OutputDir DataAccess\ScaffoldingTables\ArtikelBeschaffung -Tables artikel,artikel\_beschaffung

# decompile the DLL

One common tool for this purpose is **dotPeek**, a free .NET decompiler developed by JetBrains. Here's how you can use dotPeek to decompile and inspect a DLL:

1. **Download and Install dotPeek**:
   * You can download dotPeek from the JetBrains website (<https://www.jetbrains.com/decompiler/>).
   * Install the software on your machine.
2. **Open the DLL with dotPeek**:
   * Launch dotPeek.
   * Go to "File" > "Open" and select the DLL you want to inspect (the one you placed in the "bin" folder).
3. **Inspect the DLL**:
   * dotPeek will decompile the DLL and display its contents, including classes, methods, and code.
   * You can navigate through the decompiled code to inspect the DLL's functionality.

# Code Snippets:

public IEnumerable<int> Test(int n)

{

for (int i = 0; i < n; i++)

{

yield return i;

Console.WriteLine(i);

}

}  
  
IEnumerable<int> List = Test(10);

Clean Architecture

is a software architectural pattern that encourages the separation of concerns in your application. It defines layers and dependencies in a way that promotes maintainability, scalability, and testability. Each project or assembly in a Clean Architecture project has a specific role, and their relationships are organized in a way that ensures a clean separation of concerns.

Here's a typical project structure for a .NET-based Clean Architecture application, along with a brief description of each project:

## Core Project (Innermost Layer):

Responsibility: This is the most central project, where you define your domain models, business logic, and application-specific rules.

Reference: Core does not reference any other project. It's dependency-free and should contain your most essential application logic.

## Application Project (Use Cases):

Responsibility: This project contains application-specific use cases, which are the application's high-level services. These use cases orchestrate the core domain logic.

Reference: It references the Core project.

## Application.Contracts Project (Interfaces):

Responsibility: Define interfaces for services and repositories that your application requires. These contracts allow for the inversion of control, making it possible to implement the actual services and repositories in the Infrastructure project.

Reference: It references the Core project. Other projects, including Infrastructure, will reference this project.

## Infrastructure Project (Outermost Layer):

Responsibility: Contains implementations for infrastructure concerns, such as data access (e.g., Entity Framework or Dapper), external services (e.g., APIs, email services), and cross-cutting concerns (e.g., logging, authentication).

Reference: It references the Core and Application.Contracts projects, as it implements the interfaces defined in Application.Contracts.

Web (or Presentation) Project:

Responsibility: This is the project where you implement the user interface and interaction with the application. In the case of a web application, this could be a Blazor or ASP.NET Core application.

Reference: It references the Core, Application, Application.Contracts, and Infrastructure projects.

Tests (Optional):

Responsibility: This project contains your unit tests, integration tests, and end-to-end tests. It ensures that your application behaves correctly and maintains its integrity.

Reference: Typically, it references the Core and Application projects for testing business logic.

Database Migrations (Optional):

Responsibility: If you're using a database, this project contains database migration scripts. It's commonly used with Entity Framework Core or other ORM tools.

Reference: It references the Core, Application, and Infrastructure projects for applying migrations to the database.

Shared Kernel (Optional):

Responsibility: This project contains shared domain concepts, value objects, and utility functions used across multiple bounded contexts in your application.

Reference: It may reference the Core project and be referenced by the Core or other projects as needed.

Identity (Optional):

Responsibility: If your application involves user authentication and authorization, this project handles identity-related concerns.

Reference: It references the Core, Application, and Infrastructure projects for user management and authentication.

External Services (Optional):

Responsibility: If your application integrates with external services (e.g., APIs, message queues), you can have a project specifically for handling these integrations.

Reference: It references the Core and Infrastructure projects, depending on how these integrations are used.

Please note that Clean Architecture provides a clear separation of concerns, but the actual project structure and names can vary based on your application's specific requirements and technology stack. Additionally, Clean Architecture promotes the use of Dependency Injection (DI) to manage the dependencies between the layers. This allows for flexibility and testability, as well as easy swapping of components without affecting the core business logic.

Core: Hier definieren Sie Ihre Geschäftslogik und Modelle.

Application: Dieses Projekt enthält Anwendungslogik und Use Cases.

Application.Contracts: Hier legen Sie Schnittstellen für Services und Repositories fest.

Infrastructure: Implementiert Datenzugriff, externe Dienste und Infrastrukturdetails.

Web (oder Präsentation): Hier erstellen Sie die Benutzeroberfläche und Interaktion

Ein Bild, das Screenshot, Text, Kreis, Grafiken enthält.

Automatisch generierte Beschreibung

# Onion Architecture

The Onion Architecture is a software architectural pattern that promotes a separation of concerns within an application. It's often used in combination with Domain-Driven Design (DDD) principles. The architecture is divided into layers, and each layer has a specific responsibility. The layers are usually arranged in a way that dependencies flow inward toward the core, hence the name "Onion Architecture."

## Core Layer (Innermost Layer):

This layer contains the core domain logic and entities.  
  
// Core Layer

namespace OnionArchitecture.Core.Entities

{

public class Product

{

public int Id { get; set; }

public string Name { get; set; }

// Other properties...

}

// More entities, value objects, and domain logic...

}

## Infrastructure Layer (Middle Layer):

This layer contains implementations for external concerns like databases, file systems, etc.  
// Infrastructure Layer

namespace OnionArchitecture.Infrastructure.Persistence

{

public class ProductRepository

{

// Methods for database operations related to products

}

// More infrastructure-related classes...

}

## Application Layer (Outermost Layer):

This layer contains the application-specific logic, such as services and use cases.

// Application Layer

using OnionArchitecture.Core.Entities;

using OnionArchitecture.Infrastructure.Persistence;

namespace OnionArchitecture.Application.Services

{

public class ProductService

{

private readonly ProductRepository \_productRepository;

public ProductService(ProductRepository productRepository)

{

\_productRepository = productRepository;

}

public void AddProduct(Product product)

{

// Business logic

// Validate, process, etc.

// Save to the database using the repository

\_productRepository.Add(product);

}

// Other application services...

}

// More application-related classes...

}

Ein Bild, das Text, Kreis, Screenshot enthält.

Automatisch generierte Beschreibung