**C# Coding Standards and Naming Conventions**

Sept 2021

Content

[Purpose 2](#_Toc82780903)

[C# Naming conventions 2](#_Toc82780904)

[Indentation and Spacing 11](#_Toc82780905)

[Keep private member variables, properties, and methods in the top of the file and public members in the bottom. 13](#_Toc82780906)

[Good Programming practices 13](#_Toc82780907)

[Principles and Recommendations in Software Architecture 21](#_Toc82780908)

[• Independent from third-party bookstores 21](#_Toc82780909)

[• Testable 21](#_Toc82780910)

[• Independent of sight 21](#_Toc82780911)

[• Independent of database technology 21](#_Toc82780912)

[• Independent from outside agents 21](#_Toc82780913)

[Common abstractions in an architecture 22](#_Toc82780914)

[Clean Architecture N Layers 23](#_Toc82780915)

[Technological proposal 26](#_Toc82780916)

[Appendix 27](#_Toc82780917)

## Purpose

To describe in detail the requirements of a standard Policy for backend software development.

As well as suggesting a minimum set of requirements and technical recommendations that standardize the software development process in the phases defined by the application methodologies.

It recommends a series of standardized and consistent work instructions in such a process and serves as a common vocabulary and terminology framework for software development.

## C# Naming conventions

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Name Object | Notation | Length | Plural | Prefix | Suffix | Abbreviation | Char Mask | Underscores |
| Namespace | Pascal Case | 128 | Yes | Yes | No | No | [A-z] [0-9] | No |
| Class name | Pascal Case | 128 | No | No | Yes | No | [A-z] [0-9] | No |
| Constructor name | Pascal Case | 128 | No | No | Yes | No | [A-z] [0-9] | No |
| Method name | Pascal Case | 128 | Yes | No | No | No | [A-z] [0-9] | No |
| Method arguments | camel Case | 128 | Yes | No | No | Yes | [A-z] [0-9] | No |
| Local variable | camel Case | 50 | Yes | No | No | Yes | [A-z] [0-9] | No |
| Constants name | Pascal Case | 50 | No | No | No | No | [A-z] [0-9] | No |
| Field name | camel Case | 50 | Yes | No | No | Yes | [A-z] [0-9] | Yes |
| Properties name | Pascal Case | 50 | Yes | No | No | Yes | [A-z] [0-9] | No |
| Delegate name | Pascal Case | 128 | No | No | Yes | Yes | [A-z] | No |
| Enum type name | Pascal Case | 128 | Yes | No | No | No | [A-z] | No |

Do use PascalCasing for class names and method names:

|  |
| --- |
| public class ClientActivity  {  public void ClearStatistics()  {  //...  }  public void CalculateStatistics()  {  //...  }  } |

why: consistent with the Microsoft's .NET Framework (see [Appendix](#_Appendix_1)) and easy to read.

Do use camel Casing for method arguments and local variables:

|  |
| --- |
| public class UserLog  {  public void Add (LogEvent logEvent)  {  int itemCount = logEvent.Items.Count;  // ...  }  } |

Why: consistent with the Microsoft's .NET Framework(see [Appendix](#_Appendix_1)) and easy to read.

Do not use Hungarian notation or any other type identification in identifiers

|  |
| --- |
| // Correct  int counter;  string name;  // Avoid this  int iCounter;  string strName; |

Why: consistent with the Microsoft's .NET Framework(see [Appendix](#_Appendix_1)) and Visual Studio IDE makes determining types very easy (via tooltips). In general, you want to avoid type indicators in any identifier.

Do not use Screaming Caps for constants or read only variables:

|  |
| --- |
| // Correct  public const string ShippingType = "Dropship";  // Avoid this  public const string SHIPPINGTYPE = "Dropship"; |

Why: consistent with the Microsoft's .NET Framework(see [Appendix](#_Appendix_1)). Upper case grab too much attention

Use meaningful names for variables. The following example uses seattleCustomers for customers who are in Seattle:

|  |
| --- |
| var puertoricoCustomers = from customer in customers  where customer.City == "Seattle"  select customer.Name; |

Why: consistent with the Microsoft's .NET Framework(see [Appendix](#_Appendix_1)) and easy to read.

Avoid using Abbreviations. Exceptions: abbreviations commonly used as names, such as Id, Xml, Ftp, Uri.

|  |
| --- |
| // Correct  UserGroup userGroup;  Assignment employeeAssignment;  // Avoid this  UserGroup usrGrp;  Assignment empAssignment;  // Exceptions  CustomerId customerId;  XmlDocument xmlDocument;  FtpHelper ftpHelper;  UriPart uriPart; |

Why: consistent with the Microsoft's .NET Framework(see [Appendix](#_Appendix_1)) and prevents inconsistent abbreviations.

Do use PascalCasing or camelCasing (Depending on the identifier type) for abbreviations 3 characters or more (2 chars are both uppercase when PascalCasing is appropriate or inside the identifier).:

|  |
| --- |
| HtmlHelper htmlHelper;  FtpTransfer ftpTransfer, fastFtpTransfer;  UIControl uiControl, nextUIControl; |

Do not use Underscores in identifiers. Exception: you can prefix private fields with an underscore:

|  |
| --- |
| // Correct  public DateTime clientAppointment;  public TimeSpan timeLeft;  // Avoid this  public DateTime client\_Appointment;  public TimeSpan time\_Left;  // Exception (Class field)  private DateTime \_registrationDate; |

Why: consistent with the Microsoft's .NET Framework(see [Appendix](#_Appendix_1)) and makes code more natural to read (without 'slur'). Also avoids underline stress (inability to see underline).

Do use predefined type names (C# aliases) like int, float, string for local, parameter and member declarations. Do use .NET Framework names like Int32, Single, String when accessing the type's static members like Int32.TryParse or String.Join.

|  |
| --- |
| // Correct  string firstName;  int lastIndex;  bool isSaved;  string commaSeparatedNames = String.Join(", ", names);  int index = Int32.Parse(input);  // Avoid this  String firstName;  Int32 lastIndex;  Boolean isSaved;  string commaSeparatedNames = string.Join(", ", names);  int index = int.Parse(input); |

Why: consistent with the Microsoft's .NET Framework(see [Appendix](#_Appendix_1)) and makes code more natural to read.

Do use implicit type var for local variable declarations. Exception: primitive types (int, string, double, etc) use predefined names.

|  |
| --- |
| var stream = File.Create(path);  var customers = new Dictionary();  // Exception  int index = 100;  string timeSheet;  bool isCompleted; |

Why: removes clutter, particularly with complex generic types. Type is easily detected with Visual Studio tooltips.

Do use noun or noun phrases to name a class.

|  |
| --- |
| public class Employee  {  }  public class BusinessLocation  {  }  public class DocumentCollection  {  } |

Why: consistent with the Microsoft's .NET Framework(see [Appendix](#_Appendix_1)) and easy to remember.

Do prefix interfaces with the letter I. Interface names are noun (phrases) or adjectives.

|  |
| --- |
| public interface IShape  {  }  public interface IShapeCollection  {  }  public interface IGroupable  {  } |

Why: consistent with the Microsoft's .NET Framework(see [Appendix](#_Appendix_1)).

Do name source files according to their main classes. Exception: file names with partial classes reflect their source or purpose, e.g., designer, generated, etc.

|  |
| --- |
| // Located in Task.cs  public partial class Task  {  }  // Located in Task.generated.cs  public partial class Task  {  } |

Why: consistent with the Microsoft practices. Files are alphabetically sorted, and partial classes remain adjacent.

Do organize namespaces with a clearly defined structure:

|  |
| --- |
| // Example  namespace Company.Technology.Feature.Subnamespace  {  }  namespace Company.Product.Module.SubModule  {  }  namespace Product.Module.Component  {  }  namespace Product.Layer.Module.Group  {  } |

Why: consistent with the Microsoft's .NET Framework(see [Appendix](#_Appendix_1)). Maintains good organization of your code base.

Do vertically align curly brackets:

|  |
| --- |
| // Correct  class Program  {  static void Main (string [] args)  {  //...  }  } |

Why: Microsoft has a different standard, but developers have overwhelmingly preferred vertically aligned brackets.

Do declare all member variables at the top of a class, with static variables at the very top.

|  |
| --- |
| // Correct  public class Account  {  public static string BankName;  public static decimal Reserves;  public string Number { get; set; }  public DateTime DateOpened { get; set; }  public DateTime DateClosed { get; set; }  public decimal Balance { get; set; }  // Constructor  public Account ()  {  // ...  }  } |

Why: generally accepted practice that prevents the need to hunt for variable declarations.

Do use singular names for enums. Exception: bit field enums.

|  |
| --- |
| // Correct  public enum Color  {  Red,  Green,  Blue,  Yellow,  Magenta,  Cyan  }  // Exception  [Flags]  public enum Dockings  {  None = 0,  Top = 1,  Right = 2,  Bottom = 4,  Left = 8  }  // Exception  [Flags]  public enum LedStatus  {  BloqMays=1,  BloqNum=2,  BloqDespl=4,  All=BloqMays | BlogNum | BloqDespl,  None = 0  } |

Why: consistent with the Microsoft's .NET Framework(see [Appendix](#_Appendix_1)) and makes the code more natural to read. Plural flags because enum can hold multiple values (using bitwise “or”).

Do not explicitly specify a type of an enum or values of enums (except bit fields):

|  |
| --- |
| // Avoid this  public enum Direction: long  {  North = 1,  East = 2,  South = 3,  West = 4  }  // Correct  public enum Direction  {  North,  East,  South,  West  } |

Why: It is consistent with the .NET Framework (see Appendix) and makes the code more natural to read.

Do not use an "Enum" suffix in enum type names:

|  |
| --- |
| // Avoid this  public enum Coin**Enum**  {  Penny,  Nickel,  Dime,  Quarter,  Dollar  }  // Correct  public enum Coin  {  Penny,  Nickel,  Dime,  Quarter,  Dollar  } |

Why: consistent with the Microsoft's .NET Framework(see [Appendix](#_Appendix_1)) and consistent with prior rule of no type indicators in identifiers.

Do not use "Flag" or "Flags" suffixes in enum type names:

|  |
| --- |
| // Avoid this  [Flags]  public enum DockingsFlags  {  None = 0,  Top = 1,  Right = 2,  Bottom = 4,  Left = 8  }  // Correct  [Flags]  public enum Dockings  {  None = 0,  Top = 1,  Right = 2,  Bottom = 4,  Left = 8  } |

Why: consistent with the Microsoft's .NET Framework(see [Appendix](#_Appendix_1)) and consistent with prior rule of no type indicators in identifiers.

Do use suffix EventArgs at creation of the new classes comprising the information on event:

|  |
| --- |
| // Correct  public class BarcodeReadEventArgs : System.EventArgs  {  } |

Why: consistent with the Microsoft's .NET Framework(see [Appendix](#_Appendix_1)) and easy to read.

Do name event handlers (delegates used as types of events) with the "EventHandler" suffix, as shown in the following example:

|  |
| --- |
| // Correct  public delegate void ReadBarcodeEventHandler(object sender, ReadBarcodeEventArgs e); |

Why: consistent with the Microsoft's .NET Framework(see [Appendix](#_Appendix_1)) and easy to read.

Do not create names of parameters in methods (or constructors) which differ only by the register:

|  |
| --- |
| // Avoid this  private void MyFunction(string name, string **Name**)  {  //...  } |

Why: consistent with the Microsoft's .NET Framework(see [Appendix](#_Appendix_1)) and easy to read, and also excludes possibility of occurrence of conflict situations.

DO use two parameters named sender and e in event handlers. The sender parameter represents the object that raised the event. The sender parameter is typically of type object, even if it is possible to employ a more specific type.

|  |
| --- |
| public void ReadBarcodeEventHandler(object sender, ReadBarcodeEventArgs e)  {  //...  } |

Why: consistent with the Microsoft's .NET Framework(see [Appendix](#_Appendix_1)) and consistent with prior rule of no type indicators in identifiers.

Do use suffix Exception at creation of the new classes comprising the information on exception:

|  |
| --- |
| // Correct  public class BarcodeReadException : System.Exception  {  } |

Why: consistent with the Microsoft's .NET Framework(see [Appendix](#_Appendix_1)) and easy to read.

Do use prefix Any, Is, Have or similar keywords for Boolean identifier:

|  |
| --- |
| // Correct  public static bool IsNullOrEmpty(string value) {  return (value == null || value.Length == 0);  } |

Why: consistent with the Microsoft's .NET Framework(see [Appendix](#_Appendix_1)) and easy to read.

Use Named Arguments in method calls:

*When calling a method, arguments are passed with the parameter name followed by a col on and a value.*

|  |
| --- |
| // Method  public void DoSomething (string foo, int bar)  {  ...  }  // Avoid  DoSomething ("someString", 1);  // Correct  DoSomething (foo: "someString", bar: 1); |

Why: In Named Arguments, we do not need to pass the parameters in order as defined on method definition, so we can pass the arguments in any order on method calling.

## Indentation and Spacing

Use TAB for indentation. Do not use SPACES. Define the Tab size as 4.

Comments should be in the same level as the code (use the same level of indentation).

|  |
| --- |
| // Correcto  // Format a message and display  string fullMessage = "Hello " + name;  DateTime currentTime = DateTime.Now;  string message = fullMessage + ", the time is : " +currentTime.ToShortTimeString();  MessageBox.Show ( message ); |

|  |
| --- |
| // Incorrect  // Format a message and display  string fullMessage = "Hello " + name;  DateTime currentTime = DateTime.Now;  string message = fullMessage + ", the time is : " +currentTime.ToShortTimeString();  MessageBox.Show ( message ); |

Curly braces ({} ) should be in the same level as the code outside the braces.Imagen que contiene Texto

Descripción generada automáticamente

Use one blank line to separate logical groups of code.

|  |
| --- |
| // Correct  bool SayHello ( string name )  {  string fullMessage = "Hello " + name;  DateTime currentTime = DateTime.Now;  string message = fullMessage + ", the time is : " +currentTime.ToShortTimeString();  MessageBox.Show ( message );    if ( ... )  {  // Do something  // ...  return false;  }  return true;  } |
| // Incorrect  bool SayHello (string name)  {  string fullMessage = "Hello " + name;  DateTime currentTime = DateTime.Now;  string message = fullMessage + ", the time is : " +currentTime.ToShortTimeString();  MessageBox.Show ( message );  if ( ... )  {  // Do something  // ...return false;  }  return true;  } |

There should be one and only one single blank line between each method inside the class.

The curly braces should be on a separate line and not in the same line as if, for, etc.

|  |
| --- |
| // Correct  if ( ... )  {  // Do something  } |
| // Incorrect  if ( ... ) {  // Do something  } |

Use a single space before and after each operator and brackets.

|  |
| --- |
| // Correct  if (showResult == true)  {  for (int i = 0; i <10; i++)  {  //  }  } |
| // Incorrect  if (showResult == true)  {  for (int i = 0; i <10; i ++)  {  //  }  } |

Use #region to group related pieces of code together. If you use proper grouping using

#region, the page should like this when all definitions are collapsed.:

Texto

Descripción generada automáticamente con confianza media

Keep private member variables, properties, and methods in the top of the file and public members in the bottom.

## Good Programming practices

Avoid writing very long methods. A method should typically have 1~25 lines of code. If a method has more than 25 lines of code, you must consider re factoring into separate methods.

Method name should tell what it does. Do not use mis-leading names. If the method name is obvious, there is no need of documentation explaining what the method does.

|  |
| --- |
| // Correct  void SavePhoneNumber (string phoneNumber )  {  // Save the phone number.  } |

|  |
| --- |
| // Incorrect  // This method will save the phone number.  void SaveDetails ( string phoneNumber )  {  // Save the phone number.  } |

A method should do only 'one job'. Do not combine more than one job in a single method, even if those jobs are very small.

|  |
| --- |
| // Save the address.  SaveAddress ( address );  // Send an email to the supervisor to inform that the address is updated.  SendEmail ( address, email ); |

|  |
| --- |
| //Incorrect  //Save address and send an email to the supervisor to inform that  // the address is updated.  // Save the address.  SaveAddress ( address, email );  void SaveAddress ( string address, string email )  {  // Job 1.  // Save the address.  // ...  // Job 2.  // Send an email to inform the supervisor that the address ischanged.  // ...  } |

Use the C# specific types (aliases), rather than the types defined in System namespace.

|  |
| --- |
| //Correct  int age;  string name;  object contactInfo;  //Incorrect  Int16 age;  String name;  Object contactInfo; |

Always watch for unexpected values. For example, if you are using a parameter with 2 possible values, never assume that if one is not matching then the only possibility is the other value.

|  |
| --- |
| //Correct  If ( memberType == eMemberTypes.Registered )  {  // Registered user... do something...  }  else if ( memberType == eMemberTypes.Guest )  {  // Guest user... do something...  }  else  {  // Un expected user type. Throw an exception  throw new Exception (Un expected value” + memberType.ToString() + “’.”)  // If we introduce a new user type in future, we can easilyfind  // the problem here.  } |
| //Incorrect  If ( memberType == eMemberTypes.Registered )  {  // Registered user... do something...  }  else if ( memberType == eMemberTypes.Guest )  {  // Guest user... do something...  }  else  {  // Guest user... do something...  // If we introduce another user type in future, this code will  // fail and will not be noticed.  } |

|  |
| --- |
| //Correct  If ( memberType == eMemberTypes.Registered )  {  // Registered user... do something...  }  else if ( memberType == eMemberTypes.Guest )  {  // Guest user... do something...  }  else  {  // Un expected user type. Throw an exception  throw new Exception (Un expected value” + memberType.ToString() + “’.”)  // If we introduce a new user type in future, we can easilyfind  // the problem here.  } |
| //Incorrect  If ( memberType == eMemberTypes.Registered )  {  // Registered user... do something...  }  else if ( memberType == eMemberTypes.Guest )  {  // Guest user... do something...  }  else  {  // Guest user... do something...  // If we introduce another user type in future, this code will  // fail and will not be noticed.  } |

Do not hardcode numbers. Use constants instead. Declare constant in the top of the file and use it in your code.

However, using constants are also not recommended. You should use the constants in the config file or database so that you can change it later. Declare them as constants only if you are sure this value will never need to be changed.

Do not hardcode strings. Use resource files.

Convert strings to lowercase or upper case before comparing. This will ensure the string will match even if the string being compared has a different case.

|  |
| --- |
| if (name.ToLower() == “john” )  {  //...  } |

Use String.Empty instead of “”

|  |
| --- |
| //Correct  If(name == String.Empty )  {  // do something  } |
| //Incorrect  If (name == “” )  {  // do something  } |

Avoid using member variables. Declare local variables wherever necessary and pass it to other methods instead of sharing a member variable between methods. If you share a member variable between methods, it will be difficult to track which method changed the value and when.

|  |
| --- |
| //Incorrect  class Rectangle  {  //member variables  private double length;  private double width;  public void Acceptdetails()  {  length = 10;  width = 14;  }  public double GetArea() {  return length \* width;  } |

|  |
| --- |
| //Correct  class Rectangle  {  //member variables  private double length;  private double width;  public double GetArea(length, width)  {  //…//  }  } |

Use enum wherever required. Do not use numbers or strings to indicate discrete values.

|  |
| --- |
| //Correct  enum MailType  {  Html,  PlainText,  Attachment  }  void SendMail (string message, MailType mailType)  {  switch ( mailType )  {  case MailType.Html:  // Do somethingbreak.  case MailType.PlainText:  // Do somethingbreak;  case MailType.Attachment:  // Do something  break;  default:  // Do somethingbreak;  }  } |

|  |
| --- |
| //Incorrect  void SendMail (string message, MailType mailType)  {  switch ( mailType )  {  Case “Html”:  // Do somethingbreak.  break;  case “PlainText”:  // Do somethingbreak;  break;  case “Attachment”:  // Do something  break;  default:  // Do somethingbreak.  break;  }  } |

Do not make the member variables public or protected. Keep them private and expose public/protected Properties.

The event handler should not contain the code to perform the required action. Rather call another method from the event handler.

Do not programmatically click a button to execute the same action you have written in the button click event. Rather, call the same method which is called by the button click event handler.

Never hardcode a path or drive name in code. Get the application path programmatically and use relative path.

Never assume that your code will run from drive "C:". You may never know; some users may run it from network or from a "Z:".

In the application start up, do some kind of "self-check" and ensure all required files and dependencies are available in the expected locations. Check for database connection in startup, if required. Give a friendly message to the user in case of any problems.

If the required configuration file is not found, application should be able to create one with default values.

If a wrong value found in the configuration file, application should throw an error or give a message and should tell the user what the correct values are.

Error messages should help the user to solve the problem. Never give error messages like "Error in Application", "There is an error" etc. Instead give specific messages like "Failed to update database. Please make sure the login id and password are correct."

When displaying error messages, in addition to telling what is wrong, the message should also tell what the user should do to solve the problem. Instead of message like "Failed to update database.", suggest what should the user do: "Failed to update database. Please make sure the login id and password are correct.".

Do not have more than one class in a single file.

Avoid having very large files. If a single file has more than 1000 lines of code, it is a good candidate for refactoring. Split them logically into two or more classes.

Avoid public methods and properties unless they really need to be accessed from outside the class. Use “internal” if they are accessed only within the same assembly

If you have a method returning a collection, return an empty collection instead of null, if you have no data to return. For example, if you have a method returning an ArrayList, always return a valid ArrayList. If you have no items to return, then return a valid ArrayList with 0 items. This will make it easy for the calling application to just check for the “count” rather than doing an additional check for “null”.

Use the AssemblyInfo file to fill information like version number, description, company name, copyright notice etc.

Logically organize all your files within appropriate folders. Use 2 level folder hierarchies. You can have up to 10 folders in the root folder and each folder can have up to 5 sub folders. If you have too many folders than cannot be accommodated with the above mentioned 2 level hierarchy, you may need re factoring into multiple assemblies.

Make sure you have a good logging class which can be configured to log errors, warning, or traces. If you configure to log errors, it should only log errors. But if you configure to log traces, it should record all (errors, warnings, and trace). Your log class should be written such a way that in future you can change it easily to log to Windows Event Log, SQL Server, or Email to administrator or to a File etc without any change in any other part of the application. Use the log class extensively throughout the code to record errors, warning and even trace messages that can help you trouble shoot a problem.

If you are opening database connections, sockets, file stream, etc., always close them in the finally block. This will ensure that even if an exception occurs after opening the connection, it will be safely closed in the finally block.

Declare variables as close as possible to where it is first used. Use one variable declaration per line.

Use StringBuilder class instead of String when you must manipulate string objects in a loop. The String object works in weird way in .NET. Each time you append a string, it is discarding the old string object and recreating a new object, which is a relatively expensive operation.

Consider the following example:

|  |
| --- |
| //Incorrect  public string ComposeMessage (string[] lines)  {  string message = String.Empty;  for (int i = 0; i < lines.Length; i++)  {  message += lines [i];  }  return message;  } |

In the above example, it may look like we are just appending to the string object ‘message’. But what is happening in reality is, the string object is discarded in each iteration and recreated and appending the line to it.

If your loop has several iterations, then it is a good idea to use StringBuilder class instead of String object.

See the example where the String object is replaced with StringBuilder.

|  |
| --- |
| //Correct  public string ComposeMessage (string [] lines)  {  StringBuilder message = new StringBuilder();  for (int i = 0; i < lines. Length; i++)  {  message. Append(lines[i]);  }  return message.ToString();  } |

# 

# Principles and Recommendations in Software Architecture

There is no single clean architecture, there are several alternatives, but they are all based on the same software design principle: Separation of responsibilities. In this type of architecture, layers are established, each with a specific responsibility. More information can be found in Uncle Bob's article. Thanks to this separation, systems are achieved:

## • Independent from third-party bookstores

## • Testable

## • Independent of sight

## • Independent of database technology

## • Independent from outside agents

**Advantages in establishing a clean architecture**

A clean architecture allows, leaving the business logic intact and testable, to replace all the ends of the system (database, remote APIs, views, etc.…) with mocks or other specific implementations.

In the case of apps for iOS and Android, there are other advantages such as:

1. Iterate / evolve your product faster. Each module takes responsibility for one or more related use cases. Is there a problem in a specific module? The problem is simply searched for in the files related to that module (specifically in its tests).

2. Activating and deactivating functionalities and modules is faster. By having all the files of a module located in a single point in your project, the activation and deactivation of certain functionalities is faster (it can even be done with a flag on the server).

3. Enabling unit test, and code analysis techniques, allows:

Gain confidence. After each change uploaded to the server, the technical manager feels supported, knowing that the new changes do not cause failures in what was already working.

Find and fix bugs faster. A test case should be the functional specification of a module. If there is a problem in a module, the first place to look is the files and folders in that test case.

Establish regressions. Tests can be added as errors are found in the code to make sure they don't happen again.

4. Achieve a more manageable, maintainable and flexible code in the long term.

5. There are clear standards and conventions on how to program, but the architectures set the guidelines in which a programmer must adhere to introduce new functionalities in the application. It is relatively easy to get a new developer to learn these guidelines if you have a defined architecture.

6. Easily allows an application to evolve, (solution) initially created as Monolithic to various solutions focused on microservices, migrating specific modules (use cases) to these new solutions, even considering different alternatives in data handling, and front end.

7. Immediate implementation

It can be implemented with any programming language, among which we mention: Java, .Net, Php, Node.js.

8. The primary focus of the project is placed on the core and logic of the domain.

9. This architecture allows important changes to the application, without major impacts:

You could change the framework used, if necessary, since everything is decoupled, you could replace the database or add some other if needed.

10. The Result is an optimal, solid, quality, and scalable product.

## Common abstractions in an architecture

Imagen de la pantalla de un celular con letras

Descripción generada automáticamente con confianza media

Plato con pasta y vegetales

Descripción generada automáticamente The way to solve these abstractions can lead us to an architecture like this:

• Complex

• Inconsistent

• Rigid

• Not maintainable

• Not testable

Comida en un plato

Descripción generada automáticamente It is suggested in this guide to reach a clear level of abstraction, as in the following graphic:

• Simple

• Understandable

• Flexible

• Maintainable

• Testable

• Scalable

## Clean Architecture N Layers

Un pastel decorado

Descripción generada automáticamente con confianza media The clean architecture presents distribution of software components, where logical levels with different roles are distinguished enclosed in rings in the following graphic:

Gráfico, Gráfico de proyección solar

Descripción generada automáticamente

Each color is:

a unique responsibility.

a different level of abstraction.

Different developer roles

**Explanation of each layer:**

* **Domain** - Contains types and business logic
* Application - Contains business logic and types
* **Infrastructure**: (including persistence) contains all external abstractions, and transversal to development
* **Presentation**: It is the visual presentation layer.

Diagrama

Descripción generada automáticamente con confianza media

Graphic explanation. Infrastructure and presentation components can be replaced with minimal effort.

**Gráfico, Gráfico de proyección solar

Descripción generada automáticamenteDomain**:

It has the Entities, Value Objects, Enumerations and Logical Exceptions, Infrastructure accesses this layer and IoC (Inversion of Control) will help us to inject dependencies.

Imagen de la pantalla de un celular con letras

Descripción generada automáticamente con confianza baja **Application**:

This layer has the interfaces, models, logical commands and queries, validations and exceptions, it is the level of logical abstraction of the application that solves the Use cases. It implements the principle of single responsibility, multiple implementations.

Gráfico, Gráfico de proyección solar

Descripción generada automáticamente **Persistence**:

This layer has the DbContext, migrations, configurations, initial data seeding, and data abstractions.

Gráfico, Gráfico de proyección solar

Descripción generada automáticamente **Infrastructure**:

This layer implements Inversion of Control, Implementations as external APIs, File System, Email / SMS, any external abstraction, or transversal to the system.

Imagen de la pantalla de un celular con letras

Descripción generada automáticamente con confianza baja **Presentation**:

This layer contains the presentation layer developed in MVC .NET CORE, SPA: Angular or React, Web API, Razor Pages, MVC, Web Forms, etc.

Includes controllers, filters, views, view models, and application startup.

## Technological proposal

• ASP.NET Core 5.0 WebAPI

• Entity Framework Core 5.0

**Open Source:** This Project will mainly use nugget packages licensed under the MIT license.

**Features**

* Clean architecture
* CQRS with MediatR library
* Entity Framework Core: code first
* Repository master
* MediatR Pipeline Logging, Validation NLOG and Serilog
* Swagger UI
* Response Wrappers, Pagination
* In-memory database
* OAUTH or another authentication
* Role-based authorization
* Identity initialization, database initialization
* Custom exception handling middleware
* API version control
* Fluent Validation and Automapper
* SMTP / Mailkit / Sendgrid email service

**Solution proposal in Visual Studio**

Interfaz de usuario gráfica

Descripción generada automáticamente con confianza bajaDiagrama

Descripción generada automáticamente con confianza mediaTexto

Descripción generada automáticamente

## Appendix

Microsoft standard web link:

<https://docs.microsoft.com/en-us/dotnet/standard/design-guidelines/>

<https://docs.microsoft.com/en-us/dotnet/standard/design-guidelines/naming-guidelines>