Basics

21 January 2022 09:06

.NET: It is a software environment for building and executing *runtime managed applications* on multiple platforms. It offers support for

1. **Common Language Runtime** (CLR) - It specifies how data-types are implemented and how code is compiled for .NET applications and handles execution of such compiled (managed) code on top of services offered by its host. It includes support for
   (a) **Common Type System** - A .NET data-type is either a *value type* which provides direct access to the data or a *reference type* which provides access to the data through an indirection. The CLR's type system provides common implementations for *primitive value types* and a *unified object oriented model* for implementing user-defined reference and value types.
   (b) **Virtual Execution System** - A set of related .NET data-types are compiled together into a deployment unit called an *assembly* which contains *meta-data* (machine-readable description) for those types and *intermediate language* (IL) *opcodes* (machine neutral instructions) for their implemented methods. The CLR's execution system loads the assemblies required by the application at runtime and translates the IL opcodes of a method to their equivalent native machine instruction just-in-time of its invocation.
2. **Base Class Library** (BCL) - It is a framework (Microsoft.NETCore.App) of assemblies containing types required by a .NET application for consuming services offered by the following in a portable manner
   (a) **Runtime** which includes support for built-in data types, reflection and interoperation
   (b) **Platform** which includes support for concurrency, file I/O and communication.
3. **C# Programming Language** - It is a high-level programming language (pronounced as See Sharp) designed specifically for implementing applications which target the CLR. It has following important characteristics
   (a) It offers C++ like but more expressive syntax based on the CLR's

type system with opt-in support (unsafe blocks) for pointers.
(b) It is primarily object oriented based on common root single class inheritance model with added support for generic and functional programming.

C# Built-in Types

Type	**Data**
bool	Primitive *true*/*false* option value
char	Primitive character Unicode value
byte/sbyte	Primitive 8-bit unsigned/signed integer value
short/ushort	Primitive 16-bit signed/unsigned integer value
int/uint	Primitive 32-bit signed/unsigned integer value
long/ulong	Primitive 64-bit signed/unsigned integer value
nint/unint	Primitive native-size signed/unsigned integer value
float	Primitive 32-bit single-precision floating-point real value
double	Primitive 64-bit single-precision floating-point real value
decimal	128-bit high-precision fixed-point real value
string	reference to an immutable sequence of characters
object	reference to an instance of any type

struct	**class**
Defines a *non-nullable* type which is implicitly passed by *value*.	Defines a *nullable* type which is always passed by *reference*.
Memory is *automatically* allocated for its instance and *new* operator may be used for its initialization.	Memory is *explicitly* allocated for its instance using *new* operator which also performs its initialization.
Memory is assigned to its locally identified instance on the *stack* (fast) and it is automatically reclaimed after the identifying method of this instance returns.	Memory is assigned for any instance on the *heap* (slow) and it is automatically reclaimed during *garbage collection* that occurs after this instance is no longer reachable from any method.

Instance fields can be accessed using *direct addressing* (fast)	Instance fields can only be accessed through an *indirection* (slow)
Parameter-less constructor is always supported which is either explicitly defined (C# 10) or provided by the runtime.	Parameter-less constructor is only supported if it is either explicitly defined or no constructor is defined at all in which case it is provided at compile time.
Cannot explicitly extend any other type because it always extends System.ValueType which itself extends System.Object.	Can explicitly extend any one other class type otherwise it implicitly extends System.Object.
Implicit conversion to a compatible type requires boxing (copying) of instance data on the heap.	Implicit conversion to a compatible type does not require any copying of instance data.
Suitable for abstraction of complex data which is small and requires high performance data access.	Suitable for abstraction of complex data which is large or requires extensibility through subtyping.

Inheritance

25 January 2022 09:31

Object Identity: Two objects are considered to be *identical* if they *refer* to the same instance in the memory. In .NET whether an object x is identical to object y is indicated by the expression:

System.Object.ReferenceEquals(x, y)

Object Equality: Two objects are considered to be *equal* if they are instances of same class with matching data in the memory. In .NET whether an object x is equal to an object y is indicated by the expression:

x.GetHashCode() == y.GetHashCode() && x.Equals(y)

C# Member Access Outside of Defining Type

Access Modifier	**Current Assembly**	**External Assembly**
private (default)	none	none
internal	all	none
protected	derived	derived
internal protected	all	derived
public	all	all

Abstract Class	**Interface**
It is a non-activatable class which can define *instance fields.*	It is a non-activatable reference type which cannot define *instance fields.*
It can define a pure (unimplemented) instance method using the *abstract* modifier.	Its instance method is implicitly pure unless it is defined with a specific implementation.
Members are private by default.	Members are public by default.

Supports an instance constructor which is called by derived classes.	Does not support an instance constructor.
It can extend exactly one other class which may or may not be abstract.	It can extend multiple other interfaces.
A class can inherit from a single abstract class and it must override pure methods of that class otherwise it must be declared abstract.	A class can inherit from multiple interfaces and it must either implement their pure methods or define them as abstract methods in which case this class must be declared abstract.
A struct cannot inherit from an abstract class.	A struct can inherit from multiple interfaces and it must implement their pure methods.
Generally defined for specifying common type of state supported by objects of different inheriting classes.	Generally defined for specifying common type of behavior supported by objects of different inheriting types.

Multiple Inheritance in .NET - The type-system of CLR does not allow a class to inherit from multiple classes because an instance of such a class will require a layout with multiple sub-objects out of which only first one can be referenced in a safe manner and without complicating runtime access to the type of that instance (for casting, reflection etc) which is essential in .NET. Since interface cannot define instance fields it does not require a sub-object within an instance of its inherited class and as such a class can inherit from multiple interfaces.

Generics

28 January 2022 09:28

CLR Generics: It is syntactical support offered by the intermediate language for implementing *type-safe* code patterns which can be reused with different *data-types*. It enables a high-level language (such as C#) compiler to identify matching data-types in a declaration and to eliminate any unnecessary type conversion (such as casting, boxing and unboxing).
A generic declaration contains at least one *type argument* which is open to substitution with a known type and it is replaced by this type at runtime (reification). A type argument T is treated as System.Object type at compile time and as such it can be substituted by any type unless it appears in the declaration with following constraints.

1. **T: struct** - T can only be substituted by a value type
   T: class - T can only be substituted by a reference type
2. **T: R** - T can only be substituted by a type which supports implicit conversion to (inherits from) reference type R and as such members of R can be applied to T.
3. **T: new()** - T can only be substituted by a type which supports a parameter-less constructor and as such new operator can be applied to T with zero arguments.

Variance in Generics: By default a generic type G *is invariant* over its type argument T meaning G<V> cannot be converted to G<U> irrespective of any relationship between types U and V. A generic interface I with type argument T can be defined in

1. **Covariant form as I<out T>** to indicate that T does not appear as a parameter type in members of I and as such I<V> can be converted to I<U> if U and V are reference types such that V supports implicit conversion to (inherits from) U.
2. **Contravariant form as I<in T>** to indicate that T does not appear as a return type, ref or out parameter type in members of I and as such I<V> can be converted to I<U> if U and V are reference types such that U supports implicit conversion to (inherits from) V.

Generic Collections: The BCL includes System.Collections.Generic namespace which provides support for generic collections. It defines **ICollection**<V> interface which extends **IEnumerable**<V> interface and is extended by

1. **IList**<V> - it specifies support for collecting indexed values and for retrieving them is a sequential manner. It is implemented by List<V>
2. **ISet**<K> - it specifies support for collecting unique keys and for retrieving them in order of their values. It is implemented by HashSet<K> and SortedSet<K>
3. **IDictionary**<K, V> - it specifies support for collecting pairs each containing a unique key and a value mapped to that key. It is implemented by Dictionary<K, V>, SortedList<K, V> and SortedDictionary<K, V>

Runtime

29 January 2022 09:30

Delegate: It is a reference type which in-directs a call to a method with a particular list of parameter types and a particular return type.
A delegate object is actually an instance of a (compiler generated) class derived from System.MulticastDelegate and it has following characteristics

1. A delegate object can refer to one (through assignment) or more (through addition) methods whose return type and parameter types are compatible with those specified in the type of that delegate object.
2. The type of delegate object implements a compatible *Invoke* method, a call to which is equivalent to sequentially invoking the methods referred by that delegate object.

Event: It is a notification relayed by an object called an *event source* describing a certain change in its state to other objects called *event sinks* which are interested in taking some action when that change occurs. A .NET application can support events using following steps

1. In the event source class define an event member of System.EventHandler<E> delegate type where E is a class derived from System.EventArgs. Invoke this delegate to relay the event.
2. In the event sink class define a method compatible with above System.EventHandler<E> delegate type and add this method to the event member of the event source. When the event is relayed by the event source this method will be invoked.

Language INtegrated Queries (LINQ)**:** It is a feature of .NET runtime which enables a high-level language such as C# to provide syntactical support for querying different sources of data using data-source independent declarative statements.
LINQ specifies a set of *functional* (chainable or fluent) methods called *query operators* which can be applied to a compatible data-source. The BCL provides built-in implementations for query operators in form of extension methods defined in following static classes of System.Linq

namespace

1. **Enumerable** whose methods target *System.Collections.Generic.IEnumerable*<V> interface and pass query operations to the implementation of this interface in form of *delegates* (containing references to the instructions of those operations).
2. **Queryable** whose methods target *System.Linq.IQueryable*<V> interface and pass query operation to the LINQ provider exposed by the implementation of this interface in form of *expression-trees* (which can be decomposed into the instructions of those operations)

Reflection in .NET: Under CLR, the meta-data (type information) of any type is loaded into the memory when required as an instance of System.Type whose reference can be obtained using following methods:

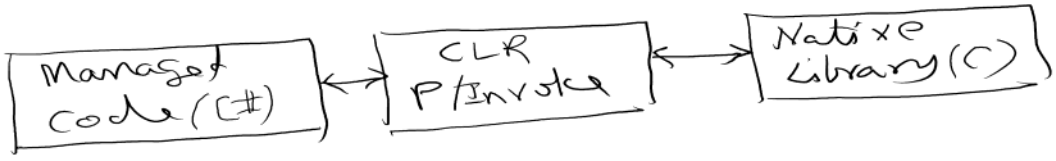
1. From the literal name of type T which is known at compile time
   Type t = typeof(T);
2. From an instance obj of the type
   Type t = obj.GetType();
3. From fully qualified name N (Namespace.Type,assembly) of the type which is discovered at runtime
   Type t = Type.GetType(N);

Attribute: It is a *programming language neutral* modifier which can be applied to a *declaration target* (such as class, field, property, method etc) in order to extend its meta-data. There are two types of attributes.

1. **Custom Attribute** which is inserted into the meta-data of its declaration target as an instance of a class derived from Sytstem.Attribute.
2. **Pseudo Attribute** which is inserted into the meta-data of its declaration target as a built-in IL modifier.

Platform Invocation (P/Invoke)**:** It is a mechanism built within the CLR that allows managed (IL) code to *invoke* unmanaged (native) functions exported by a *platform* specific library which supports *dynamic linking*. Platform invocation can be applied in C# using following steps:

1. Define an *extern static* method with System.Runtime.InteropServices.DllImportAttribute which identifies the native library along with its unmanaged entry-point function.
2. Call the above method and the CLR will automatically invoke the specified entry-point function performing required conversions between managed (C#) and unmanaged (C) data-types.



Database

04 February 2022 15:30

.NET Data Provider: It is an add-on (nuget) package which enables a .NET application to consume relational data managed by a particular data-base system using SQL. It includes support for following objects

1. **Connection** - It opens a communication session with the database system using the information provided in a (connection) string.
2. **Command** - It executes an SQL statement on a database system using the Connection object.
3. **DataReader** - It fetches rows resulting from execution of a query statement using the Command object.

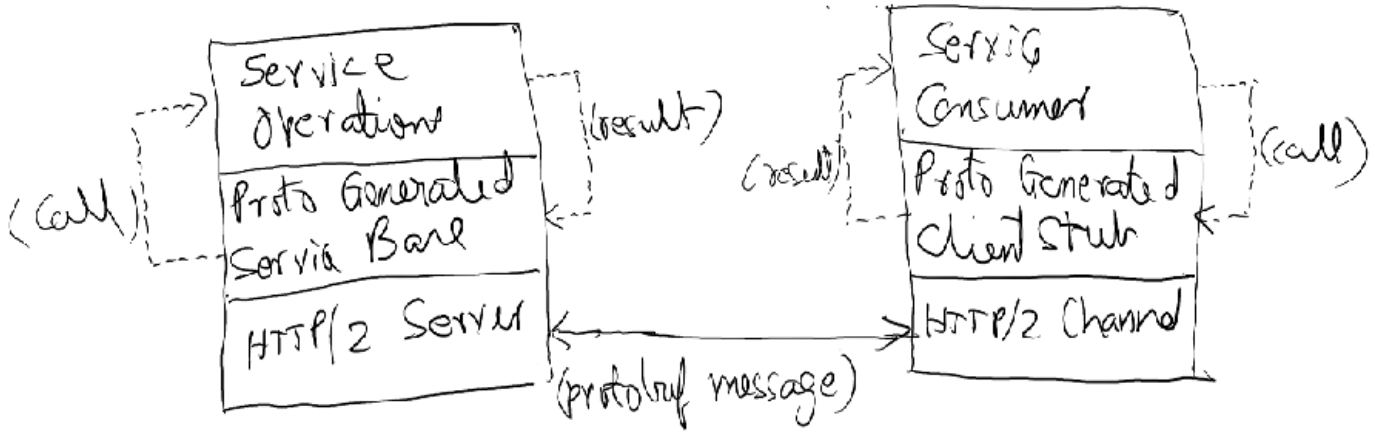
Entity Framework Core: It is an add-on package which enables a .NET application to consume relational data managed by a particular database system using LINQ queryable sets of uniquely identifiable *plain old CLR object* (POCO) type instances known as *entities*.
Under entity framework the *conceptual model* of application data is as specified by entities is associated with the *storage model* of that data using

1. **Mapping Conventions** - The Entity class is mapped to the database table whose name matches with the name of *DbSet<Entity>* type property in the *DbContext* derived class and its each property is mapped to the column with matching name in that table. The column mapped to *Id* or *EntityId* property is constrained to be the *primary-key* and the column mapped to *ParentEntityId* property is constrained to be the *foreign-key*.
2. **Data Annotations** - The conventional mapping is overridden by applying database schema specifying attributes to the entity class and its properties.
3. **Fluent API**: The conventional mapping is overridden by passing the meta-data of the entity class and its properties along with database schema specifiers through a chain of methods exposed by model builder objects.

gRPC: It is a *light-weight*, *high-performance* generic *remote-procedure call* framework for implementing *cross-platform*, *service-oriented*

distributed applications using different high-level programming languages. It includes support for

1. **Protobuf** - It is a standard specified by Google consisting of an *interface definition language* known as Proto for describing service operations along with the structures of messages exchanged by those operations and a *binary buffer format* for serializing and deserializing those messages.
2. **Channel** - It allows the client which is a consumer of service operations to *connect* to the server which publishes those operation while enabling each of them to *stream* a protobuf message or a sequence of such messages to other using *HTTP/2*.

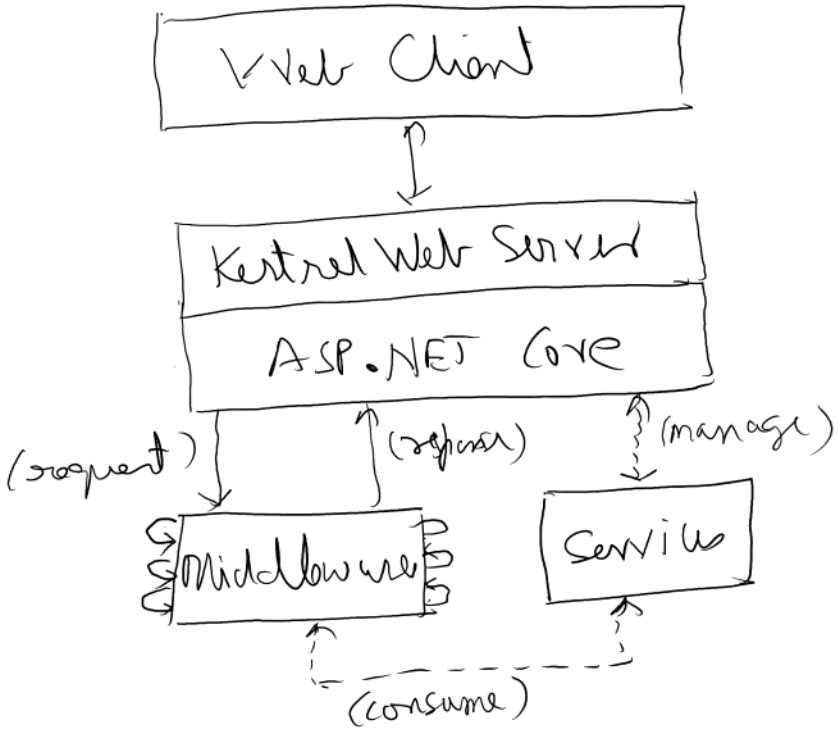


Web

08 February 2022 15:31

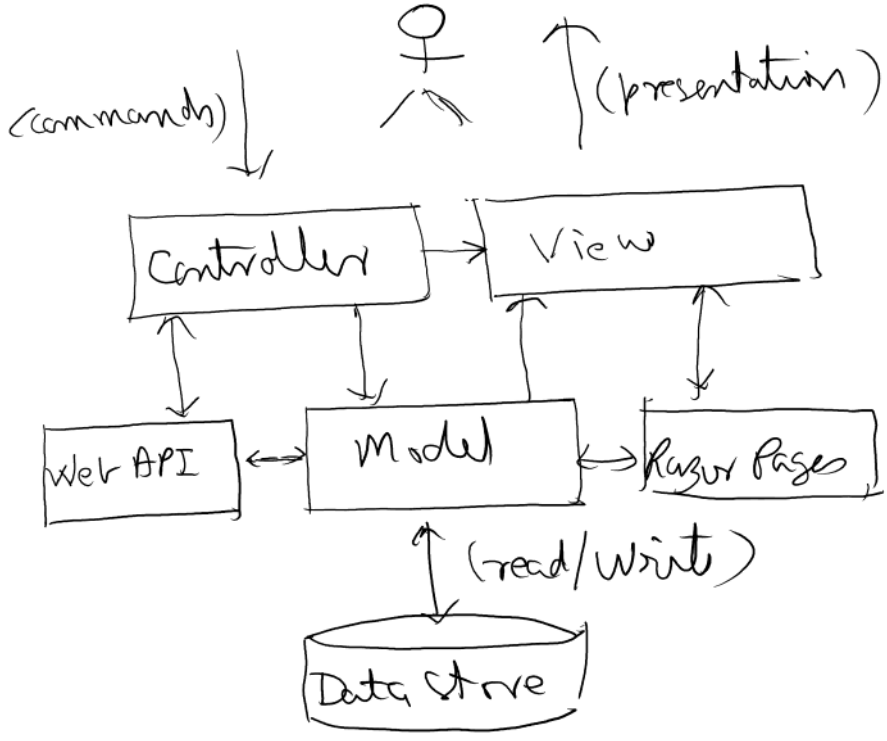
ASP.NET Core: It is a framework for building *cross-platform* HTTP *server-side* applications using .NET. It provides the *Kestrel* web-server and uses it by default for hosting applications with support for

1. **Request Pipeline** enabling the application to generate the response for a request received by the server by passing this request through a sequence of operations known as the *middle-wares*.
2. **Dependency Injection** enabling the application to manage the *life-times* of services required by its components and allowing these components to consume such services in a *loosely coupled* manner.



ASP.NET MVC: It is a set of built-in middle-wares and services required for dividing the implementation of a complex web-application into a *model* which handles data access, a *view* which handles data presentation and a *controller* which passes the requested commands to the other two. It also serves as the basis for

1. **Razor Pages** enabling a web-application to publish dynamic content (using model and view) through a web-page composed of HTML and C# code which is executed on the server.
2. **Web API** enabling a web-application to publish a RESTful service (using model and controller) which client-side code can consume to create, read, update and delete data-objects persisted on the server.



Windows

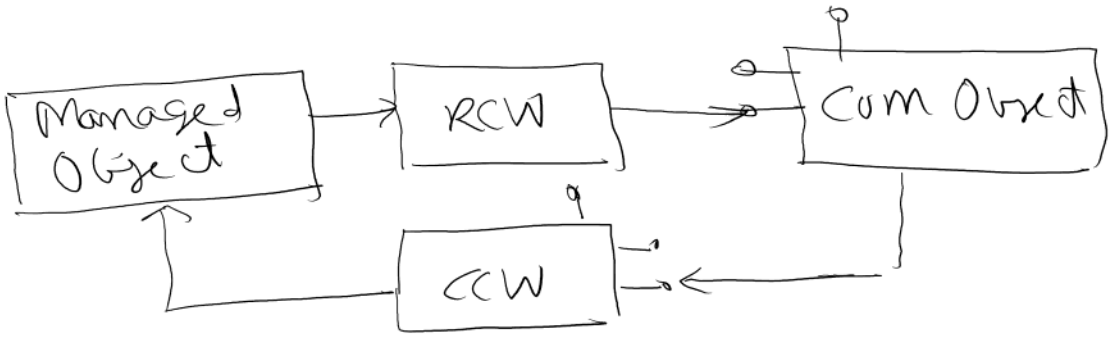
11 February 2022 09:30

Component Object Model (COM)**:** It is a Windows specific binary standard for implementing a programming language neutral object known as a component which supports dynamic activation, manages its own lifetime and whose interfaces can be discovered at runtime.
A COM object is activated by the Windows platform

1. from a class identified by a *GUID* (a 128-bit globally unique identifier) mapped (in Windows registry) to the path of its factory server (DLL or EXE). A COM object can only be consumed using a GUID identified interface implemented by its class and each such interface extends the standard *IUnknown* interface which specifies support for *querying interfaces* and *reference counting*.
2. within a group known as an *apartment* so that its methods can be directly invoked only by a thread belonging to that apartment. A COM object provided by an *in-process* (DLL) server with threading model set (in Windows registry) to 'Apartment' can only be activated within a *single-threaded apartment* (STA) containing exactly one thread which can also handle method-invocation messages sent by other threads (through a proxy of that object).

COM Interop: It is runtime support offered by .NET on Windows platform for reusing COM objects within managed code with help of following CLR generated objects:

1. **Runtime Callable Wrapper** - for consuming COM interface exposed by an unmanaged COM object from managed code
2. **COM Callable Wrapper** - for passing interface of a managed object to unmanaged code as a COM interface.



Character User Interface (CUI)	**Graphical User Interface** (GUI)
Smallest unit of output is a *character* and as such it can only present plain text.	Smallest unit of output is a *pixel* and as such it can present graphical shapes in addition to plain text.
Input can only be received using keyboard.	Input can be received using a pointing device (mouse) in addition to keyboard.
User interacts by typing and entering unfriendly command which are different for different applications.	User interacts by pointing and clicking on friendly graphical elements which are consistent across applications.
User's input is received through *pull model* which requires simple programming.	User's input is received through *event-driven* model which requires complex programming.
Implementation is generally independent of the underlying platform.	Implementation is generally specific to the underlying platform.
Commonly used by *console applications* which receive most of their input from	Commonly used by *desktop applications* which receive most of their input interactively at

command line at startup time.	runtime.

Window: It is a rectangular area of graphical screen which can react to user's input. On Windows platform the state of a window on the screen is managed by a shared memory structure called a *window object* which has following characteristics.

1. It is a single-threaded object which can only be accessed directly by its *owner* (creating) thread.
2. It provides a *message-driven* interface for controlling the appearance and the behavior of its on-screen window.

Windows Forms: It is runtime support offered by .NET for implementing basic UI for desktop applications on Windows platform. It hosts a form based graphical user interface composed of *controls* each with following characteristics:

1. It wraps a window-object and exposes its message-driven interface using properties and events.
2. It provides a *graphics* object for painting shapes and text on a window it wraps using drawing elements such as a *brush, pen* and *font*.

Data Binding: It is a mechanism of attaching a component of user-interface to an element of data so that any change in the state of this component automatically updates that element. In *bidirectional* (two-way) data binding a change in the data element is also automatically reflected in the component bound to that element.
In Windows Forms, data-binding is handled for controls by the *BindingSource* component which includes a list of data elements and indicates the element at current position. It supports

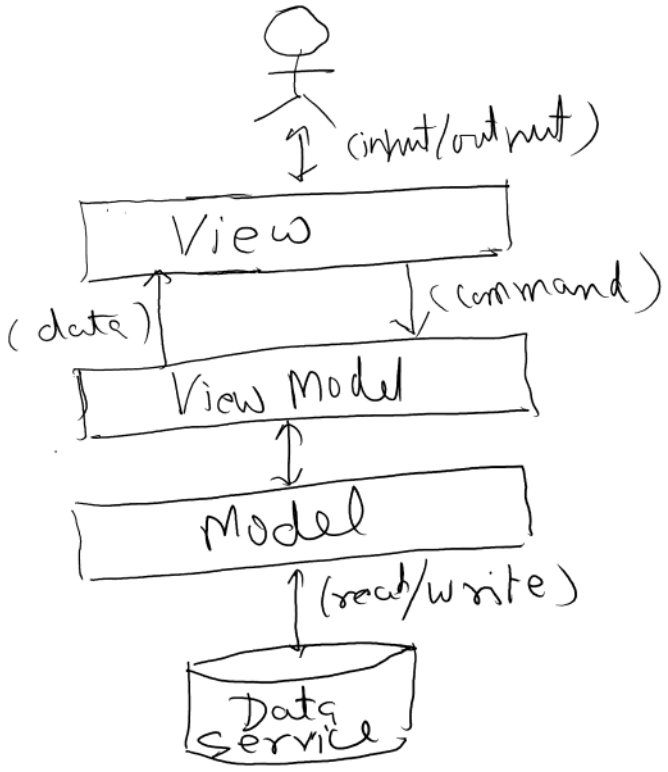
1. **Simple Data Binding** in which the control (such as TextBox) is bound to the data element at the current position within the BindingSource.
2. **Complex Data Binding** in which the control (such as ComboBox) is bound to the entire list of data elements included in the BindingSource.

Windows Presentation Foundations (WPF)**:** It is runtime support offered by .NET for implementing advanced UI for desktop applications. It hosts a windows based graphical user interface composed of *UI Elements* each with following characteristics:

1. It is a *single-threaded* object with ability to handle cross-thread invocations and to support *dependency properties* which allow bidirectional data-binding for their sparsely stored values.
2. It is a *visual* object which provides drawing instructions required for rendering it on the graphical screen and for receiving user's input directed to its *layout*.

Windows Forms	**WPF**
Output is rendered as *raster* graphics which is composed of pixel coordinates and colors.	Output is rendered as *vector* graphics which is composed of drawing instructions and meta-data.
Input is directly received by the child which is a *heavy-weight* window object placed on top of the parent.	Input is routed by the parent to the child which is a *light-weight* visual object drawn on the parent.
The layout of the UI is implemented within its *designer* code.	The layout of the UI is specified using a markup language known as *XAML*.
The UI only supports 2D graphics.	The UI supports 2D/3D graphics along with animation.
Offers a simple programming model based on *event-handling*.	Prefers a complex programming model based on *property-binding.*

MVVM Pattern: It is an *architectural style* for dividing the implementation of an application into a *model* which handles reading and writing of data, a *view* which presents the user-interface and the *view-model* which exposes bindable properties for passing data from model to view and commands from view to model.


(input/output)
View
(data)
(command)
View Model
Model
(read/write)
Data Service