# Basics

18 June 2022 09:28

**.NET:** It is a software *platform* for building and executing runtime *managed* applications on different operating systems (Windows, Linux, macOS, Android and iOS). The programing environment of .NET consists of

1. **Common Language Runtime** (CLR) - It specifies how data is structured and code is compiled for a .NET application and it handles execution of such compiled (managed) code on top of the services provided by its host. It implements
   (a) **Common Type System** - A .NET data type is either a *value type* whose data can be directly accessed from its identifier or a *reference type* whose data can only be accessed through an indirection from its identifier. The CLR supports (fourteen) *primitive* value types and offers a *unified object-oriented* model for implementing *user-defined* reference and value types.
   (b) **Virtual Execution System** - A group of related .NET data types are compiled into a single unit of deployment called an *assembly* (DLL file) which contains the *meta-data* (machine readable description) of those types along with *intermediate language opcodes* (machine neutral instructions) of their implemented methods. The CLR loads the assemblies required by the executing application and translates the IL opcodes of a method into its equivalent machine instructions *just-in-time* of its invocation.

2. **Base Class Library** (BCL) - It is a framework (Microsoft.NETCore.App) of assemblies consisting of types required by a .NET application for consuming the services offered by the following in a portable manner
   (a) **Runtime** which includes support for built-in types, reflection and native interop.
   (b) **Platform** which includes support for multithreading, file i/o and communication sockets.

3. **C# Programming Language** - It is a high-level programming language (pronounced as See Sharp) designed specifically for coding applications and libraries which target the CLR. It has following important features
   (a) It offers a C++ like but more expressive syntax based on the common type system with opt-in support for pointers and fixed-size buffers.
   (b) It is primarily object oriented based on common root single state inheritance model with added support for generic and functional programming.

**C# Built-in Types**

| **Type** | **Data** |
|---|---|
| bool | Primitive *true*\|*false* option value |
| char | Primitive *unicode character* type 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 *double-precision floating-point real* value |
| decimal | 96-bit *high-precision fixed-point real* value |
| string | Reference to an *immutable sequence* of *char* type values |
| 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 instance and new operator is | Memory is *explicitly* allocated for instance using new operator which |

| | |
|---|---|
| used for calling the constructor | also calls the constructor |
| Memory is assigned on *stack* (fast) for a local instance which is automatically deallocated when its identifying method returns | Memory is always assigned on *heap* (slow) for any instance which is automatically deallocated during garbage collection if it is no longer reachable from any method |
| Instance fields are *directly* accessible (fast) from the identifier | Instance fields are only accessible through an *indirection* (slow) from the identifier |
| Always supports a parameter-less constructor which is equivalent to default field initializer if it is not explicitly defined | Only supports a parameter-less constructor if it is explicitly defined or implicitly defined in absence of any explicitly defined constructor |
| Cannot explicitly extend any other type because it always directly 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 copying of instance data into an object on heap (boxing) | Implicit conversion to a compatible type does not require copying of instance data |
| Suitable for abstraction of data which is *small* and requires *high performance* access | Suitable for abstraction of data which is *large* or requires *extensibility through sub-typing* |

# Inheritance

20 June 2022 17:58

**Object Identity:** An object **x** is *identical* to object **y** if they refer to the same instance in the memory which can be determined from expression: **System.Object.ReferenceEquals(x, y)**

**Object Equality:** An object **x** is *equal* to object **y** if they refer to instances of same type with matching data in the memory which can be determined from expression:

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

**Visibility of members outside of its 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 reference type which can define instance fields | It is a non-activatable reference type which cannot define instance fields |
| Can define a pure (unimplemented virtual) instance method using abstract modifier | Instance method is implicitly pure (but not virtual) unless defined with an implementation |
| Can define a constructor which is called by derived classes | Cannot define a constructor |
| Members are private by default | Members are public by default |

| | |
|---|---|
| Can extend exactly one other abstract or non-abstract class | Can extend multiple other interfaces |
| A class can only inherit from a single abstract class | A class can inherit from multiple interfaces |
| A struct cannot inherit from an abstract class | A struct can inherit from multiple interfaces |
| Suitable for segregation of common methods exposed by different types of objects which are associated with their common type of state | Suitable for segregation of common methods exposed by different types of objects which are independent of the type of their state |

**Multiple Inheritance in C#:** CLR's type system does not allow a class to inherit from multiple other classes because an instance of such a class will require multiple sub-objects (to store values of instance fields defined by corresponding base-class) which complicates its runtime type access required for *safe-casting* and *reflection*. Since interface cannot define instance fields it does require a sub-object within an instance of its inheriting class and as such CLR allows a class to inherit from multiple interfaces.

# Generics

22 June 2022 17:38

**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 the C# compiler to identify matching data-types in a declaration and to eliminate unnecessary type conversions (like casting and boxing).

A generic declaration contains at least one *type parameter* which can be substituted with any known data-type by default and it is replaced by that data-type at runtime (reification) but it is treated as System.Object at compile time.

A type parameter T can be included in a generic declaration with following compile-time restrictions called *constraints*

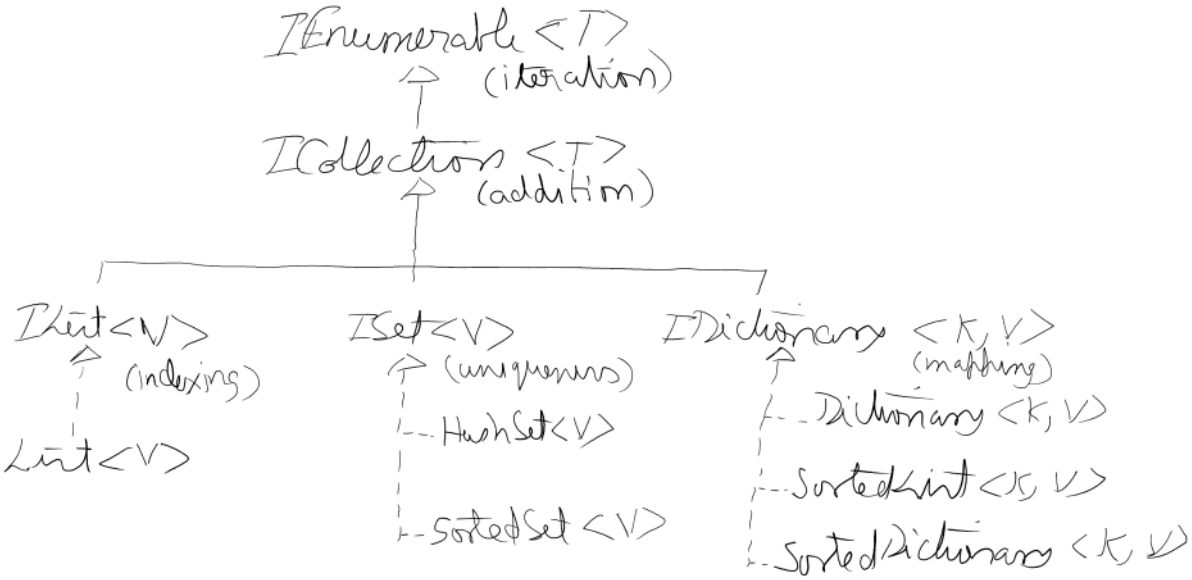
1. **T: struct|class** - to indicate that T can only be substituted by a value|reference type.
2. **T: R** - to indicate that T can only be substituted by a type which supports implicit conversion to the reference type R and such members of R can be applied to T.
3. **T: new()** - to indicate that T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

**Variance in Generics:** A generic type **G** is *invariant* over its type parameter **T** meaning G<U> cannot be substituted by G<V> irrespective of relationship between U and V. However a generic interface **I** with type parameter **T** can be defined in a

1. **Covariant form I<out T>** - to indicate that T does not appear as a parameter type in members of I and as such I<U> can be substituted by I<V> if U and V are reference types such that V supports implicit conversion to U.
2. **Contravariant form I<in T>** - to indicate that T only appears as a parameter type without *ref* or *out* modifiers in members of I and as such I<U> can be substituted by I<V> if U and V are reference types such that U supports implicit conversion to V.

**Generic Collection:** It is an object of a generic type which supports grouping of elements of a given type and provides operations to access those elements in a type-safe manner. The System.Collections.Generic namespace of BCL includes ICollection<T> interface which extends its IEnumerable<T> interface (which defines GetEnumerator method) to define standard methods for adding/removing elements to/from a collection. ICollection<T> interface is further extended by

1. **IList<T**> - It is implemented by a sequential collection which supports indexer for randomly accessing its elements. This interface is implemented by List<T> and also by System.Array (base of all CLR arrays).
2. **ISet<T>** - It is implemented by a collection which does not allow duplicate elements whose uniqueness is determined from their behavior. This interface is implemented by HashSet<T> (less memory, slow operations) and SortedSet<T> (more memory, fast operations).
3. **IDictionary<K, V>** - It is implemented by a collection whose each entry refers to a pair consisting of a unique key and its associated value. This interface is implemented by Dictionary<K, V> (less memory, slow addition, slow search), SortedList<K, V> (less memory, slow addition, fast search) and SortedDictionary<K, V> (more memory, fast addition, fast search).


IEnumerable<T>
(iteration)
ICollection<T>
(addition)
IList<V>
(indexing)
List<V>
ISet<V>
(uniqueness)
HashSet<V>
SortedSet<V>
IDictionary<K, V>
(mapping)
Dictionary<K, V>
SortedList<K, V>
SortedDictionary<K, V>

# Runtime

24 June 2022 10:50

**Callback Function:** It is a mechanism of passing a function of a specific type to some code in order to allow this code to indirectly call that function. C# provides syntactical support for implementing callbacks using *delegate* type (a System.MulticastDelegate derived class auto-generated by the compiler) which has following characteristics:

1. A method whose return type and list of parameter types are compatible with the corresponding types specified in the delegate definition can be assigned (or added) to an identifier of that delegate type.
2. A method (or methods) assigned (or added) to an identifier of a delegate type can be invoked (sequentially) by calling the compatible *Invoke* method exposed by that delegate type.

**Functional Programming:** It is *a declarative style* of processing data in which this data is passed through a *chain of calls* to functions which do not have side effects and accept other functions as their arguments when required. .NET offers language integrated queries (LINQ) for retrieval and *functional* processing of data delivered by *any data-source* in an *identical* manner using

1. **Standard Query Operators** - these are common *fluent functions* which perform specific operations on the data yielded by a data-source. The *System.Linq* namespace of BCL includes the static Enumerable/Queryable class which defines query operators as extension methods for IEnumerable<T>/IQueryable<T> interface to perform operations passed to them in form of delegates/expression-trees.
2. **C# Query Syntax** - it includes *context-sensitive keywords* for composing a declarative statement describing how data is obtained from a data-source and processed. The compiler translates a query statement into a chain of calls to the corresponding standard query operators beginning with the target data-source.

**Reflection:** It is a mechanism which enables a program to examine the

structure of its own data at runtime. The CLR provides meta-data (description) of a type **T** through an instance of **System.Type** whose reference can be obtained at compile time in C# using expression **typeof(T)** and it can be discovered at runtime from

1. An instance **obj** of the target type using expression **obj.GetType()**
2. Fully qualified name **N** (*Namespace.Type,assembly*) of the target type using expression **System.Type.GetType(N)**
   Note: The assembly identified in name N is loaded from the directory containing the assembly of executing program and this assembly is also used to obtain the target type if N does not include its assembly name.

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

1. **Custom Attribute** - this type of attribute is inserted into the meta-data of its declaration target as an instance of a class derived from System.Attribute.
2. **Pseudo Attribute** - this type of attribute is inserted into the meta-data of its declaration target as a built-in intermediate language modifier.

**Unsafe Context:** It is a statement block which generates managed (IL) code whose safe execution cannot be ensured by the CLR. In C#, unsafe context must be marked with *unsafe* keyword and it must be compiled by passing /unsafe option to the compiler (by setting *AllowUnsafeBlocks* project property to true). It is commonly used for including a *pointer* type in the code which may be required for

1. Improving performance of the program because accessing data using a pointer cannot be prechecked and hence does not execute extra verification instructions.
2. Calling a native function exported by a platform specific dynamically linked library using *platform invocation* mechanism supported by the CLR.

# Platform

28 June 2022 09:31

**Concurrency:** It is programming support offered by the platform for simultaneously executing multiple blocks of code. It is used for

1. **Asynchrony** - in which the caller of a procedure can resume execution before that procedure return in order to increase the responsiveness of the running program.
2. **Parallelism** - in which different iterations of a long running loop is executed on separate processors (or cores) available on the hardware in order to increase performance of the running program.

**Thread:** It is the basic unit of concurrency within an executing program and it has following characteristics.

1. A thread refers to a function which it executes concurrently with other threads including the main thread which is started automatically when the program is executed.
2. A thread shares values of variables with other threads of the program except for the local variables of the functions it calls.

**Object Serialization:** It is a mechanism of converting the entire state of an object (including the state of each object it references) into a stream (series) of bytes from which it can be deserialized (reconstructed). It is commonly used for

1. **Persistence** - in which the object is transferred to a storage medium.
2. **Marshalling** - in which the object is transported across process boundary.

**Object Serialization in .NET:** The BCL supports *text-based serialization* for serializing/deserializing an object by getting/setting values of its accessible properties. It provides

1. **System.Xml.Serialization.XmlSerializer** - It serializes the target object in *XML* format and during deserialization instantiates the type of target object using its parameter-less constructor.

2. **System.Text.Json.JsonSerializer** - It serializes the target object in *JSON* format and during deserialization instantiates the type of target object using its parameter-less constructor or a constructor whose parameter names match with the names of its accessible properties.

**Graphical User Interface** (GUI)**:** It is a mechanism that enables a software known as a *desktop application* to interact with a human through a more visual and friendly manner with support for

1. Output of rich pixel-based content that includes graphical objects in addition to character based text.
2. Input of commands and data using a pointing device such as a mouse in addition to the keyboard.

**Window:** It is an *overlappable rectangular* area of graphical screen which can react to user's input. On Windows system, each on-screen window is managed using a shared memory-structure called a *window object* which has following characteristics:

1. It is a *single-threaded* object which is directly accessible only to its *owner thread* responsible for creating it.
2. It provides a *message-driven* interface for controlling the appearance and the behavior of its on-screen window.

**Windows Forms:** It is the programming support offered by .NET for implementing a basic graphical UI for an application on the Windows platform. It enables composition of a *form-based* UI using interactive components called *controls* each of which wraps a corresponding window object to support

1. Properties for sending messages to the window object and events for handling messages sent to its owner thread.
2. Painting geometrical shapes, pictures and text on the window managed by the window object using its graphics object.
3. Activating objects and initializing their properties in a visual manner within the designer provided by the development environment.
4. Binding properties to data exposed by another object so that they are automatically initialized with that data and when they are

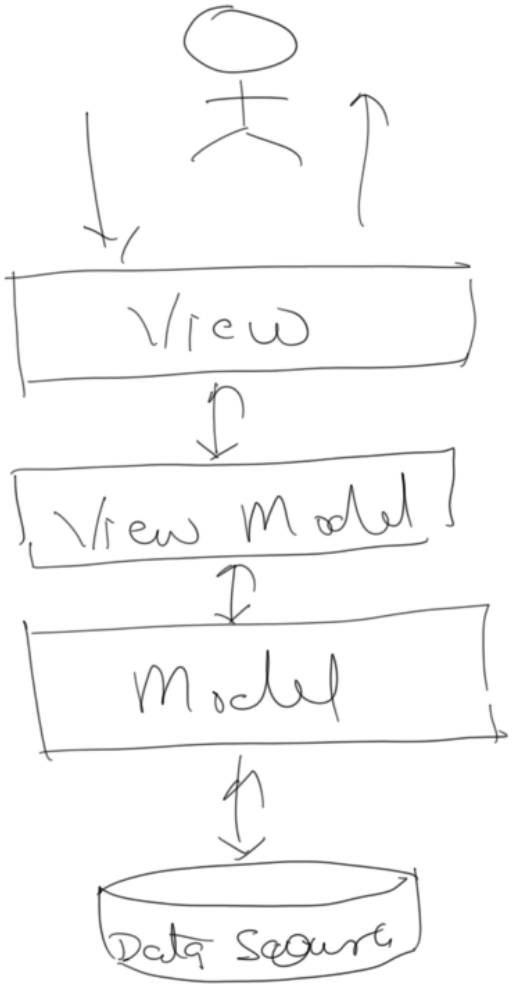
changed that data is automatically updated.

**Windows Presentation Foundation** (WPF)**:** It is the programming support offered by .NET for implementing an advances graphical UI for an application on the Windows platform. It enables composition of a *window-based* UI using interactive components called *controls* each with following characteristics:

1. It is a *dispatcher object* i.e. it is a single threaded object which supports *cross-thread* invocations.
2. It is a *dependency object* i.e. it is a dispatcher object which can expose *sparsely stored* properties with support for *bidirectional* data-binding.
3. It is a *visual* object i.e. it is a dependency object which can provide drawing instructions required for rendering it on the screen.
4. It is a *UI element* i.e. is a visual object which can receive user's inputs 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 control which is a *heavy-weight* object placed on the parent window. | Input is routed by the parent window to the control which is a *light-weight* object drawn on that window. |
| The *layout* of the UI is implemented within its *designer* code. | The *layout* of the UI is described using a markup language called *XAML.* |
| The UI only supports 2D graphics. | The UI supports 2D/3D graphics along with animation. |
| It offers a simple programming model based on *handling* of *events*. | It prefers a complex programming model based on *binding* of *properties**. |

**MVVM Pattern* enables division of the implementation of an application into the *model* which handles reading/writing of data from/to its source, the *view* which handles input/output of data from/to the user and the *view-model* which exposes bindable properties for passing data supplied by the model to the view and commands received by the view to the model.



# Communication

29 June 2022 17:10

**Socket:** It is a *common logical interface* offered by the platform for transporting data between separate processes using different *communication schemes* supported by the operating system. Each socket is bound to a unique *endpoint* identifying its communication type specific *address* and provides operations to *send/receive* data to/from other such endpoints. A *connection oriented* socket which supports *streaming* of data between endpoints is used for implementing

**(A) Server** - It provides a service for processing or sharing some data over a *well-known endpoint address* using following steps

1. Open a listener socket and bind it to a particular endpoint address.
2. Use the above listener socket to accept the socket connected to the endpoint which has requested this connection.
3. Use the above accepted socket to exchange data with its connected endpoint.
4. Close the above accepted socket and go to step 2.

**(B) Client** - It consumes the service provided by the server from a *random endpoint address* using following steps

1. Open a socket and connect it to the endpoint address of the server.
2. Use the above socket to exchange data with its connected endpoint
3. Close the above socket.

**Network Communication:** It is a mechanism for supporting transfer of data between different processes running on separate machines linked to each other through some type of networking hardware. The platform provides sockets built on top of the network communication layer (protocol stack) of the operating system which commonly includes implementations for

1. **Internet Protocol** (IP v4/6) - It is *a network protocol* which provides a network linkage type (WiFi, 4G) independent scheme for identifying each *host* (communicating machine) on the network using a unique (32/128-bit) integer known as its *IP address* and for supporting transfer of data between such hosts using structured

blocks (containing a maximum of 65535 bytes each) known as *IP packets* each including the IP addresses of source and destination along with other headers.

2. **Transmission Control Protocol** (TCP) - It is a *transport protocol* which provides an IP based scheme for identifying each *peer* (communicating process) running on a host using a unique 16-bit integer called *port address* and for supporting a connection oriented mechanism for reliable streaming of data between two peers using IP packets which include synchronization and acknowledgement headers.

**Distributed Computing:** Dividing the implementation of a software into multiple parts so that each part executes within its own process on a separate machine and interacts with other parts using network communication is called distributed computing. It is commonly used for

1. Centralization of resources over the network by running some common code on a single machine which otherwise is supposed to execute on multiple machines. Large scale centralization of resources over the internet is called *cloud computing*.
2. Decentralization of resources over the network by running different parts of some code on multiple machines which otherwise is supposed to execute on a single machine. Large scale decentralization of resources over the internet is called *grid computing*.

**Hyper-Text Transport Protocol** (HTTP)**:** It is a standard TCP/IP based scheme (application protocol) for sharing a resource over the network using its *uniform resource identifier* (URI) containing the path of that resource along with the endpoint of its provider (server). HTTP offers a simple *request-response* based model for exchanging data between endpoints in a *stateless* manner (one request per connection) which involves

1. The client sends a request based on a URI in following form
   <VERB> <path> HTTP/<version>\r\n
   <Header-Name>: <value>\r\n
   …

\r\n
<body-content>

Standard HTTP Request verbs
**POST** - *Create* a new resource from the content of the request body.
**GET** - *Read* the resource identified in the requested path and send its content in the response body.
**PUT** - *Update* the resource identified in the requested path from the content sent in the request body.
**DELETE** - *Delete* the resource identified in the requested path.

1. The server sends the response based on the request in following form
   HTTP/<version> <status-code> <status-message>\r\n
   <Header-Name>: <value>\r\n
   …
   \r\n
   <body-content>

   Standard HTTP Response Status Codes
   **2nn** - The request was handled successfully and the content of the requested resource is in the response body.
   **3nn** - Redirect the request to the URI in the status message.
   **4nn** - The request has an error such as invalid path or verb or body content.
   **5nn** - The request cannot be handled because of some internal error.

# Database

01 July 2022 09:35

**Relational Data:** It is a form of persistent data, logically structured into *tables* each created with its own set of *columns* which can be referenced by columns of another table to support *parent-child* relationship between those tables. The data is stored in the *rows* of the table whose columns are defined to specify

1. **Schema** - to indicate the type of the values that can be stored in the rows of that table.
2. **Constraints** - to indicate the restrictions on the values that can be stored in the rows of that table.

**Transactional Data:** It is a form of persistent data whose different parts can be updated within a single unit of work known as a *transaction* with support for a *commit* operation to persists the new state of data and a *rollback* operation to restore the data to its original state. A well-behaved transaction has following ACID characteristics

1. **Atomic** - A transaction should only perform an update whose effect can be cancelled by the rollback operation.
2. **Consistent** - A transaction should rollback as soon as any of its update violates a constraint on the target data.
3. **Isolated** - A transaction should not allow any data it accesses to be updated by an operation that does not belong to its scope.
4. **Durable** - A transaction should always end either with a commit or a rollback operation.

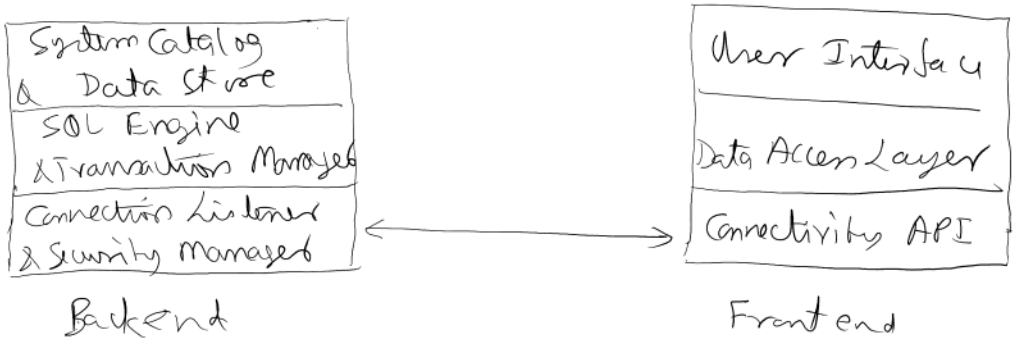
**Relational Database System:** It is a software that manages persistence of relational and transactional data with support for

1. Consuming this data using an API based on (a fourth generation declarative) *structured query language* (SQL).
2. Sharing this data in a secure manner based on *authentication* (confirming identity of a user) and *authorization* (granting permission to a user) schemes.

**Database Application:** It is software that automates a certain business

process by performing a set of transactional operations known as the *business logic* on some persistent data. Its implementation is generally divided to execute as at least two separate but communicating processes with following responsibilities

1. **Backend** - It manages persistence of transactional data for the application and it can optionally implement the business logic for that application to increase its *maintainability*. It is commonly supported using a relational database server installed on a central machine over the network.
2. **Frontend** - It provides a user-interface for the application and it can optionally implement the business logic for that application to increase its *scalability*. It is commonly supported using a client program which is installed on each of its user machine over the network.




**Middle Tier:** It is a separate process which hosts the implementation of a particular business logic for an application and is inserted between the frontend (presentation tier) and the backend (data tier) of that application to improve its maintainability as well as its scalability. It is commonly designed using *service oriented architecture* in which its implementation is divided into (small) sets of operations known as (micro)services each with following characteristics

1. A service is an *autonomous* software and as such it can be deployed

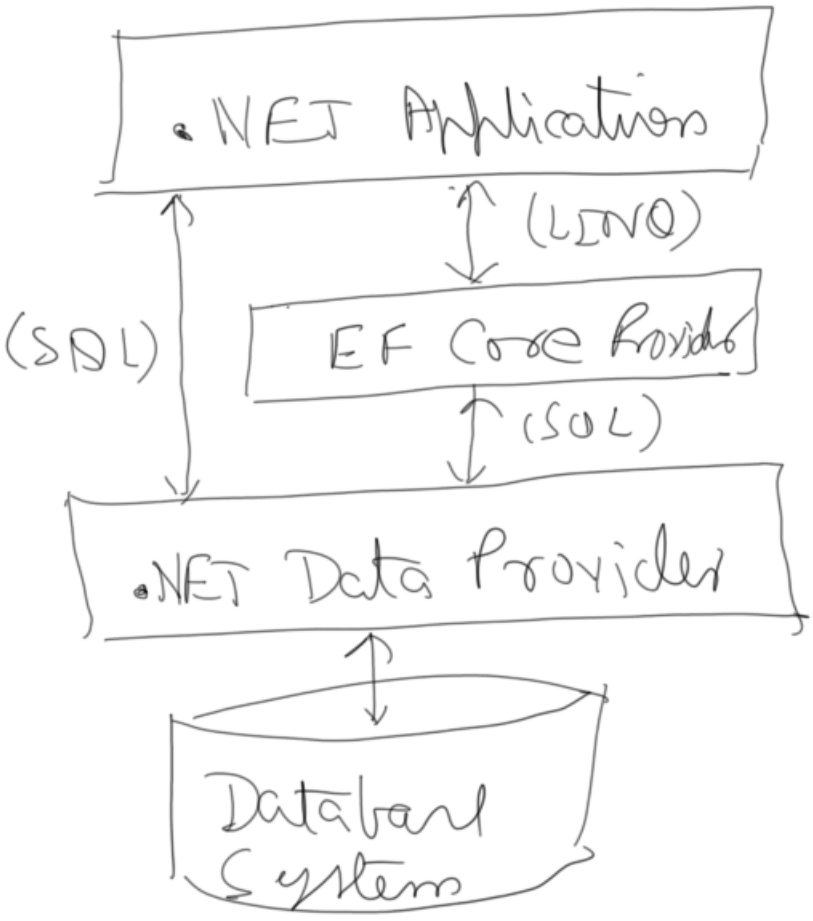
(and maintained) independently of other services.
2. A service only shares the *description* of its operations and the *schema* of the data exchanged by those operations.
3. A service has an *explicit boundary* and as such it can only be consumed by exchanging messages with the endpoint published by its host process which can execute on a separate machine (or in a separate container).

**.NET Data Provider:** It is a database specific package which allows a .NET application to consume the relational data managed by that database system. It includes support for following object types

1. **Connection** - It opens a communication session with the database system using the information specified in a connection-string.
2. **Command** - It sends an SQL statement to the database system for execution using the Connection object.
3. **DataReader** - It fetches the rows resulting from the execution of a query statement sent to database system using the Command object.

**Entity Framework Core:** It is a database specific package which allows a .NET application to consume relational data managed by that database system as objects whose types are unaware of their data-source. It includes support for

1. **Entity** - It is an instance of a *plain old CLR object* (POCO) type with a *unique identity* and whose state can be synchronized with a row of a database table mapped to its type (using *Standard-Conventions* or *Data-Annotations* or *Fluent-API*).
2. **DbSet** - It is a LINQ queryable set of entities of a particular type which is loaded when those entities are retrieved for the first time from the database table mapped to their type.
3. **DbContext** - It provides basic support for connecting to the database system, for defining DbSet type properties which expose entities and to save any change to such a DbSet into the database table mapped to its entity type.


.NET Application
(LINQ)
(SQL)
EF Core Provider
(SQL)
.NET Data Provider
Database System

# Web

02 July 2022 09:29

**Web Application:** It is a software executed by a web-server to publish *dynamic content* over its HTTP endpoint so that this content can be presented to the user through a web-browser.

| **Classic Web Application** | **Modern Web Application** |
|---|---|
| The data is acquired and output is rendered by the server-side code and this output is transported to the browser for presentation. | The data is acquired by the server side code and it transported to the browser where it is rendered by the client-side code for presentation. |
| Input is received by the browser and transported to the server where it is handled by the server side code. | Input is received by the browser where it is directly handled by client side code which calls server-side code if required. |
| Implementation requires familiarity with server-side frameworks such as Spring (Java) and ASP.NET (C#). | Implementation requires familiarity with server-side frameworks as well as client side frameworks such as Angular and React. |
| Application is more secure and is independent of type/version of the browser. | Application is less secure and may depend upon the type/version of the browser. |
| UI is less responsive because every interaction requires its re-rendering on the server. | UI is more responsive because interactions do not require any re-rendering. |
| It uses patterns suitable for large applications with multiple web-pages | It uses patterns suitable form a *single page application* (SPA) |

**REST** (REpresentational State Transfer)**:** It is an *architectural style* applied for enabling the *exchange* (transfer) of the *content* (state) of

some *resource* between *HTTP endpoints* using a mutually agreed upon *format* (representational form). A RESTful service (API) uses REST to publish methods of a *stateless* object so that each such method

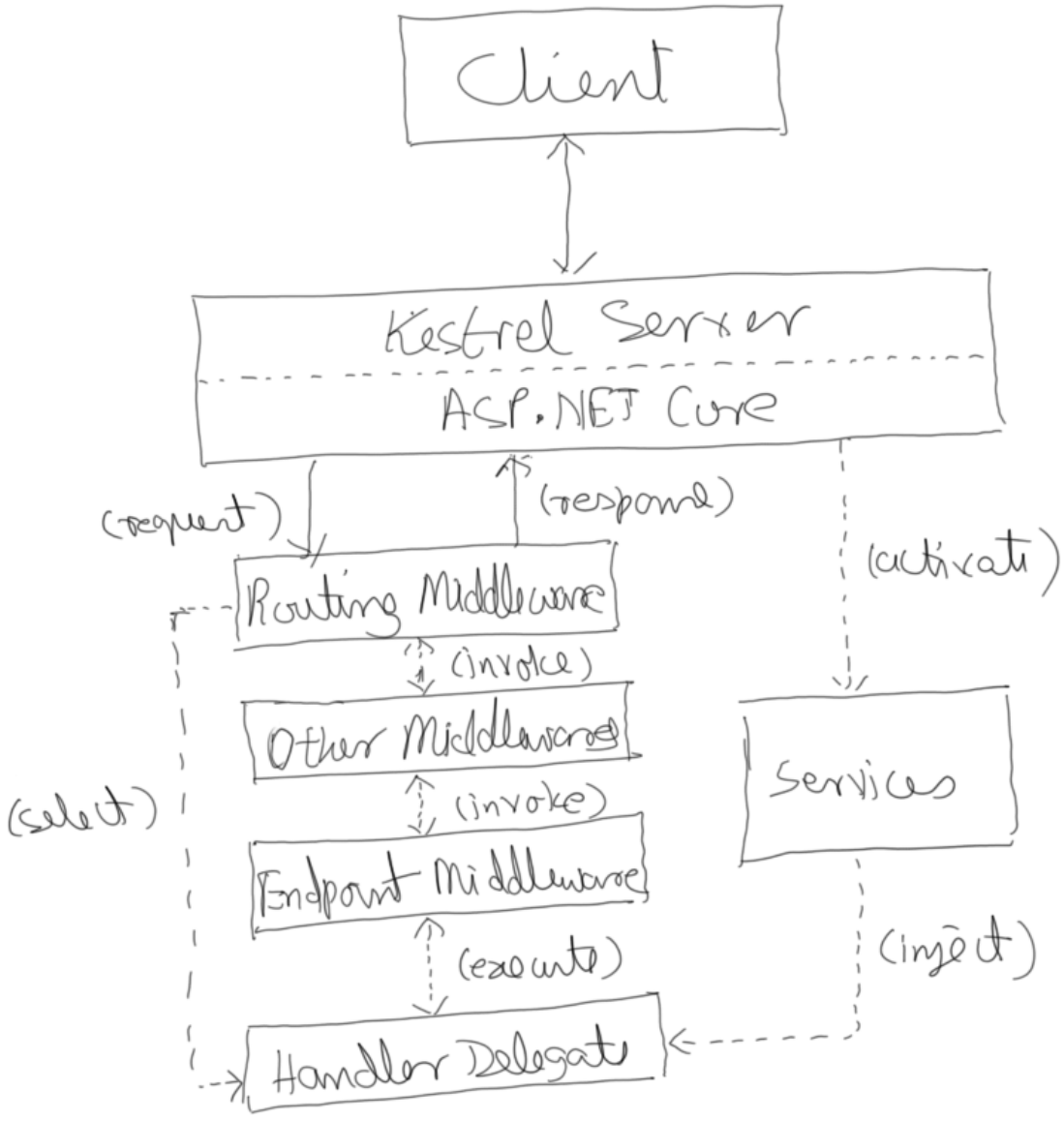
1. Performs a *create, read, update* or *delete* operation indicated by the requested verb (POST, GET, PUT or DELETE) on a data-object identified within the requested path.
2. *Produces* (in response) or *Consumes* (from request) a data-object using a standard *media-format* (such as application/json) and returns a standard HTTP status code to indicate the success or the failure of its operation.

**ASP.NET Core:** It is an infrastructure for building *cross-platform server-side* applications using .NET. It includes a *highly configurable multi-protocol* server known as *Kestrel* which it uses by default to host applications with support for

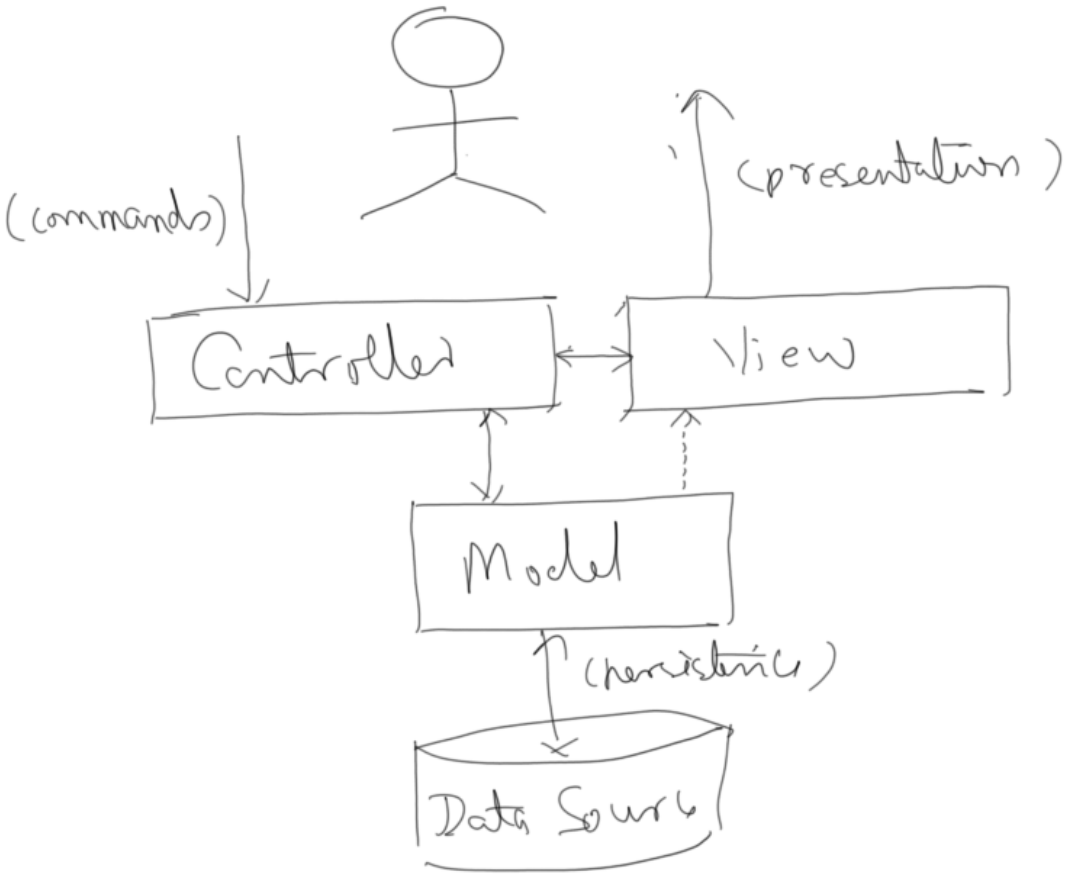
1. **Request Pipeline** - to generate a response to the request received by the server by passing this request through a chain of delegates known as *middle-wares* allowing each of them to process an incoming request and optionally invoking the next one in the chain. The request pipeline includes following middle-wares by default:
   (a) **Routing** - It is the *initial* middle-ware which uses the *verb* and the *path* of the current request to select the delegate whose method handles this request.
   (b) **Endpoint** - It is the *terminal* middle-ware which *executes* the delegate selected by the routing middle-ware for handling the current request.

2. **Service Container** - to manage the *life-times* of commonly required services and inserting the interfaces exposed by such services into their consumer using a loosely-coupled mechanism known as *dependency-injection*. The service container supports following life-time modes for services
   (a) **Singleton** - Exactly one instance of service type is activated and this instance is shared by all of its consumers across all requests.
   (b) **Scoped** - A new instance of service type is activated for each

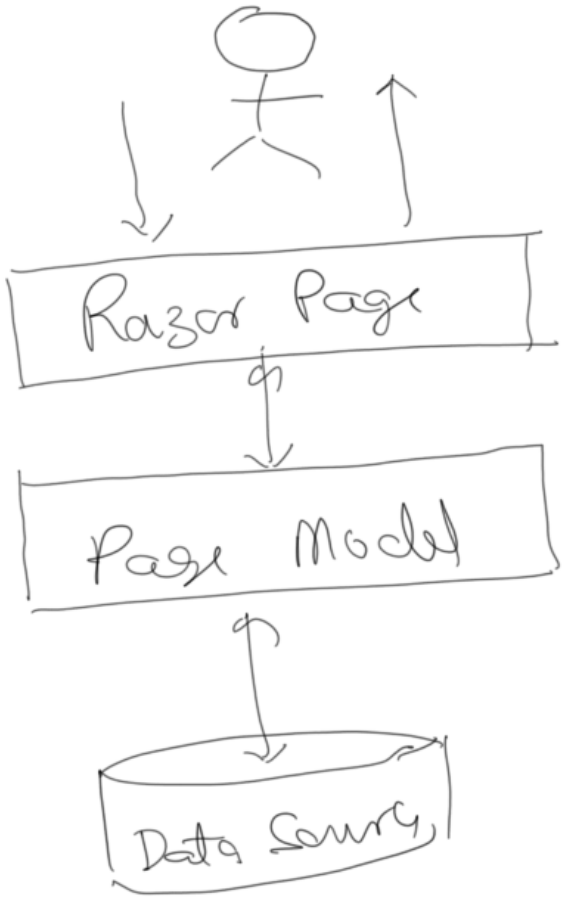
request and this instance is shared by all of its consumers.
(c) **Transient** - A new instance of service type is activated every time it is required by any of its consumer.



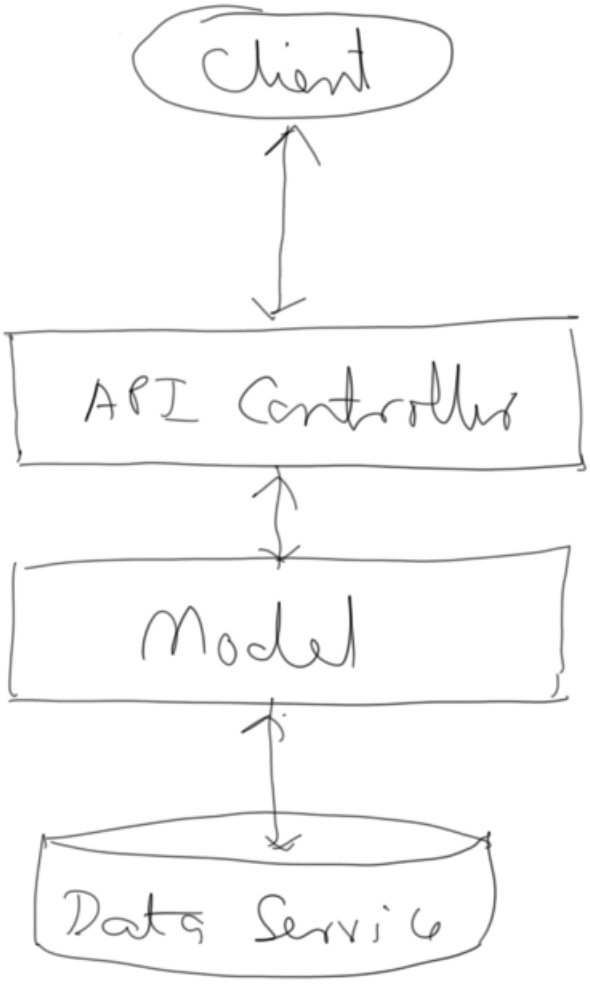

**ASP.NET Core MVC:** It is a framework built on top of ASP.NET Core for dividing the implementation of a large web application into a *model* which provides access to data, a *view* which provides a user-interface and a *controller* which passes user's commands to the other two.




**Razor Pages:** It is a framework built on top of ASP.NET Core MVC for publishing *dynamic content* through a web-page implemented using *razor-syntax* which allows mixing of HTML with C# code whose execution is performed on the server.

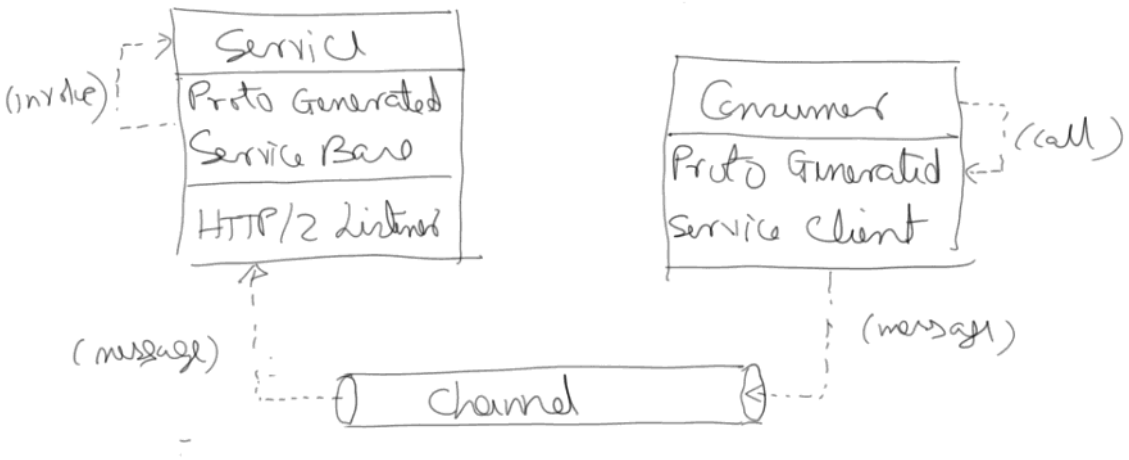



**Web-API:** It is a framework built on top of ASP.NET Core MVC for publishing a RESTful service which client-side code can consume to create, read, update or delete a data-object whose persistence is managed on the server.




**gRPC:** It is a *light-weight* generic *remote procedure call* framework for implementing *cross-platform service-oriented* distributed applications in multiple high-level languages (like C++, Java, C#, Go, Python). It includes support for

1. **Protobuf** - It specifies a standard *syntax* for describing operations exposed by a service along with the structures of messages exchanged by those operations and provides a standard *binary format* for serializing such messages.
2. **Channel** - It opens an HTTP/2 connection between network endpoints enabling one to send invocation request to the other while allowing them to stream a protobuf serialized message or a sequence of such messages to each other.


Servic
Proto Generated
Service Base
HTTP/2 Listener
(invoke)
Consumer
Proto Generatid
service client
(call)
(message)
(message)
Channel