# What is RESTful api?

REST is a set of architectural constraints, not a protocol or a standard.

When a client request is made via a RESTful API, it transfers a representation of the state of the resource to the requester or endpoint. This information, or representation, is delivered in one of several formats via HTTP: JSON (Javascript Object Notation), HTML, XLT, Python, PHP, or plain text.

For an API to be considered RESTful, it must conform to these criteria:

* A client-server architecture made up of clients, servers, and resources, with requests managed through HTTP.
* Stateless client-server communication, meaning no client information is stored between get requests and each request is separate and unconnected.

# Parameter Binding in ASP.NET Web API

Query string parameter name and action method parameter name must be the same (case-**insensitive**). If names do not match, then the values of the parameters will not be set. The order of the parameters can be different.

public class StudentController : ApiController

{

public Student Get(int id)

{

}

}

http://localhost/api/student?id=1

http://localhost/api/student?ID=1

Example: Post Method with Primitive and Complex Type Parameters

 Copy

public class Student

{

public int Id {get; set; }

public string Name {get; set; }

}

public class StudentController: ApiController

{

public Student Post (int age, Student student)

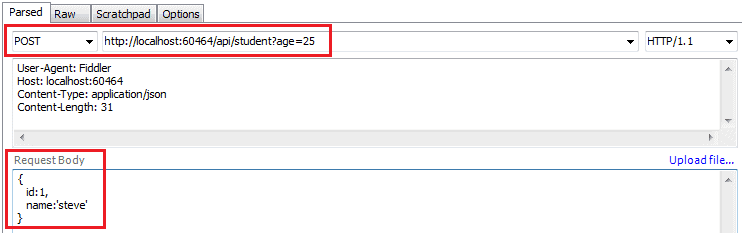
{

}

}

The above Post method includes both primitive and complex type parameters. So, by default, Web API will get the age parameter from query string and student parameter from the request body.

Following is a valid HTTP POST request in the fiddler for the above action method.

[](https://www.tutorialsteacher.com/Content/images/webapi/post-request-fiddler.png)Parameter Binding

 Note:

Post action method cannot include multiple complex type parameters because, at most, one parameter can be read from the request body.

Parameter binding for Put and Patch method will be the same as the POST method in Web API.

# [FromUri] and [FromBody]

Use [FromUri] attribute to force Web API to get the value of complex type from the query string and [FromBody] attribute to get the value of primitive type from the request body, opposite to the default rules.

# Action Method Return Type

The Web API action method can have following return types.

1. Void
2. Primitive type or Complex type
3. HttpResponseMessage
4. IHttpActionResult

| **ApiController Method** | **Description** |
| --- | --- |
| BadRequest() | Creates a BadRequestResult object with status code 400. |
| Conflict() | Creates a ConflictResult object with status code 409. |
| Content() | Creates a NegotiatedContentResult with the specified status code and data. |
| Created() | Creates a CreatedNegotiatedContentResult with status code 201 Created. |
| CreatedAtRoute() | Creates a CreatedAtRouteNegotiatedContentResult with status code 201 created. |
| InternalServerError() | Creates an InternalServerErrorResult with status code 500 Internal server error. |
| NotFound() | Creates a NotFoundResult with status code404. |
| Ok() | Creates an OkResult with status code 200. |
| Redirect() | Creates a RedirectResult with status code 302. |
| RedirectToRoute() | Creates a RedirectToRouteResult with status code 302. |
| ResponseMessage() | Creates a ResponseMessageResult with the specified HttpResponseMessage. |
| StatusCode() | Creates a StatusCodeResult with the specified http status code. |
| Unauthorized() | Creates an UnauthorizedResult with status code 401. |

# Create Custom Result Type

You can create your own custom class as a result type that implements IHttpActionResult interface.

The following example demonstrates implementing IHttpActionResult class.

Example: Create Custom Result Type

public class TextResult : IHttpActionResult

{

string \_value;

HttpRequestMessage \_request;

public TextResult(string value, HttpRequestMessage request)

{

\_value = value;

\_request = request;

}

public Task<HttpResponseMessage> ExecuteAsync(CancellationToken cancellationToken)

{

var response = new HttpResponseMessage()

{

Content = new StringContent(\_value),

RequestMessage = \_request

};

return Task.FromResult(response);

//Creates a [Task<TResult>](https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task-1?view=net-6.0) that's completed successfully with the specified result.

}

}

Now, you can return TextResult object from the action method as shown below.

Example: Return Custom Result Type

public IHttpActionResult GetName(int id)

{

string name = GetStudentName(id);

if (String.IsNullOrEmpty(name))

{

return NotFound();

}

return new TextResult(name, Request);

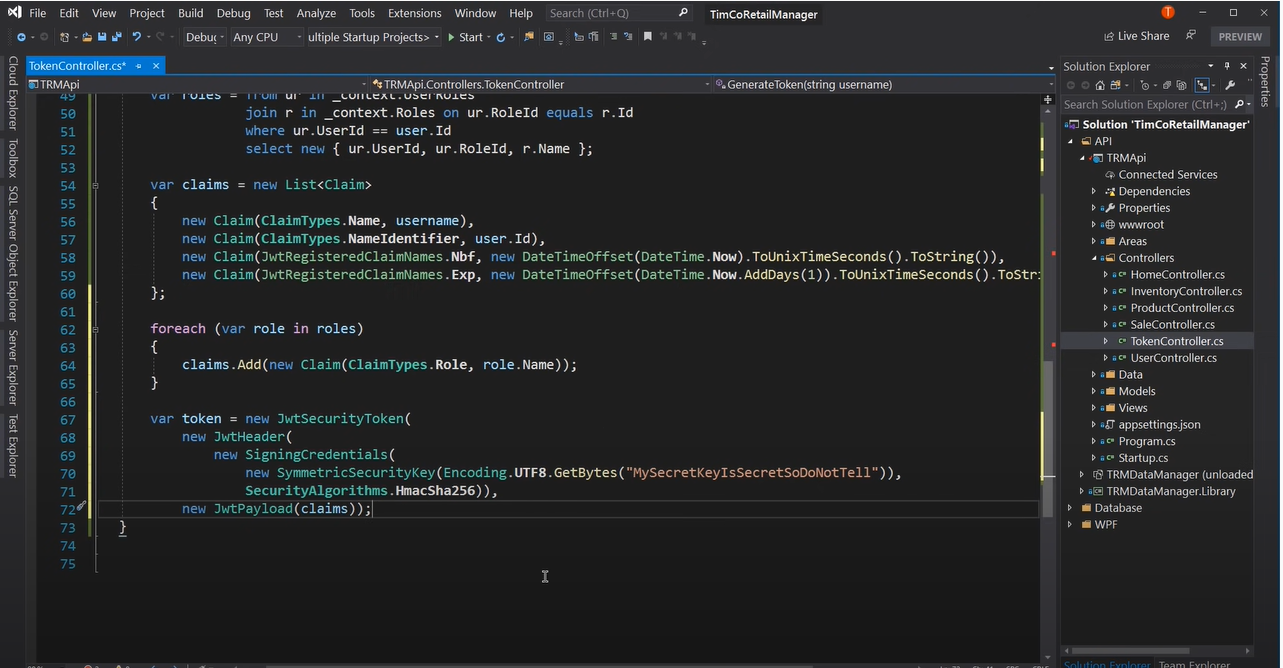
}

# Web API Request/Response Data Formats:

Media type (aka MIME type) specifies the format of the data as type/subtype e.g., text/html, text/xml, application/json, image/jpeg etc.

In HTTP request, MIME type is specified in the request header using Accept and Content-Type attribute. The Accept header attribute specifies the format of response data which the client expects, and the Content-Type header attribute specifies the format of the data in the request body so that receiver can parse it into appropriate format.

# JWT TOKEN:



Nbf: Token expire not Before

Exp: Expires

## Jwt token requires:

1. Name - UserName
2. UserID
3. Roles
4. SymetricSecurityKey
5. Security Algorithm - HmacSha256

# C#

C# is a general purpose, object-oriented, component-based programming language.

# .Net

.NET is a platform that includes languages, a runtime, and framework libraries, allowing developers to create many types of applications.

# .Net Web Api design

https://www.freecodecamp.org/news/an-awesome-guide-on-how-to-build-restful-apis-with-asp-net-core-87b818123e28/

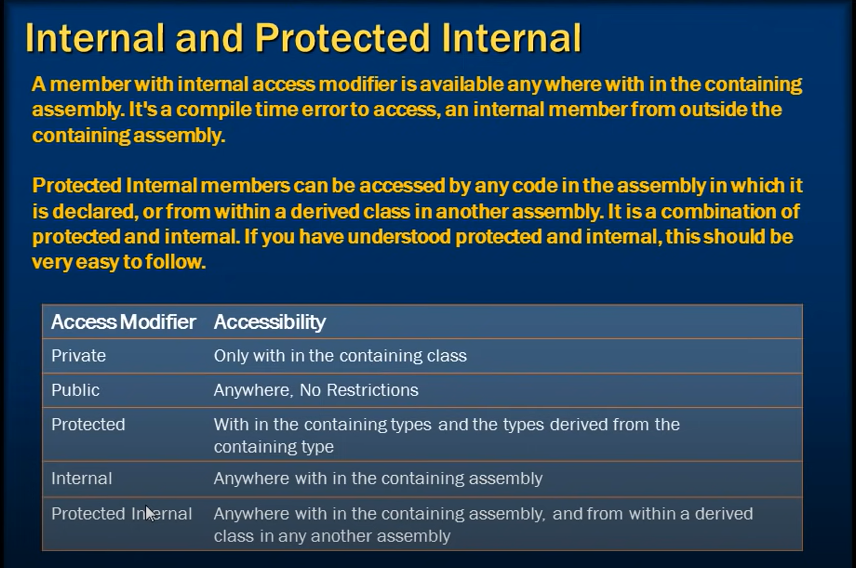
# Collections

NET collection classes let you work with data in many ways. Instead of an array, you

can use a List. If you need a first-in first-out set of items, you can use a Queue. If you need to

work with items that have unique IDs, you can use a Dictionary.

# Access Modifiers



# IOC

Not Started

# using

using (var calc4 = new ScientificCalculator()) // accepts any that implements IDisposable

{

// Do stuff.

// Call dispose automatically after execution

}

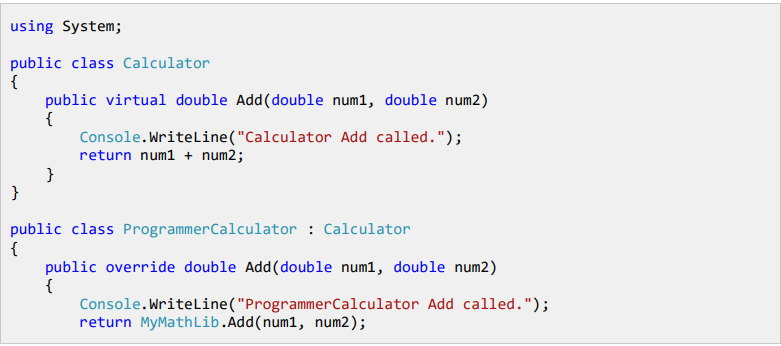
The using statement accepts parameters with any type that implements **IDisposable**. It takes care of calling Dispose () after the block completes execution.

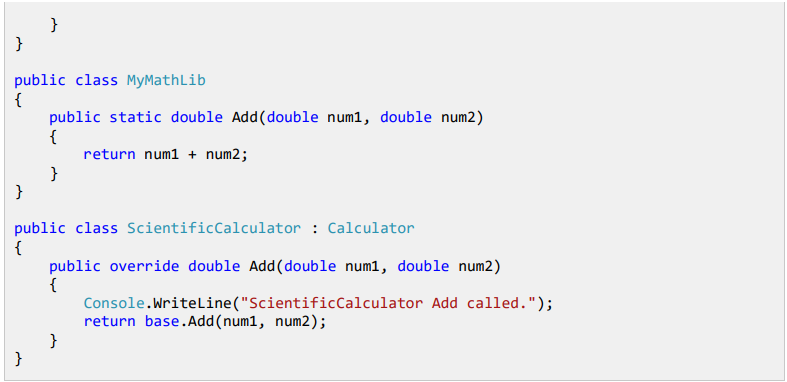
# Polymorphism

Polymorphism can be done through

1. virtual and override method.
2. Abstract Classes
3. Interface

Polymorphism lets derived classes specialize a base class implementation. The mechanism to allow polymorphism is to decorate a base class method with the virtual modifier and decorate the derived class method with the override modifier.

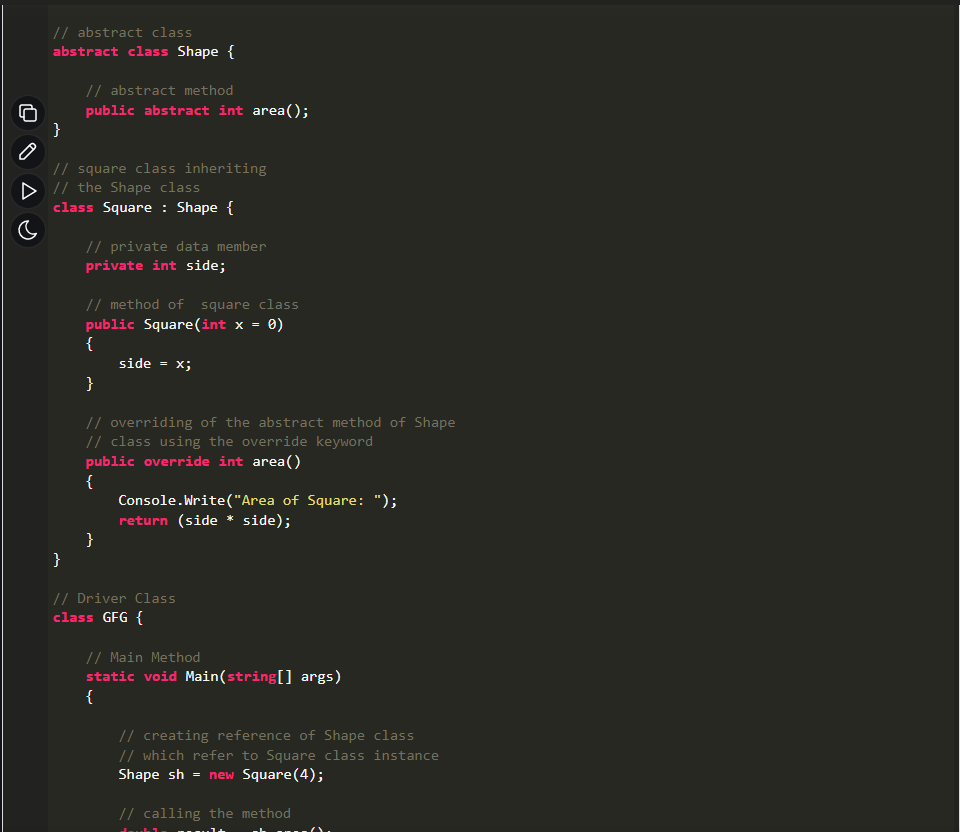




# Abstraction

Abstract class cannot be instantiated. We define class as an abstract class if we don’t want class to be instantiated.

If the class is defined as an abstract and the method is also defined as an abstract method, it forces the class inheriting the abstract class to implement the abstract method.



# Inheritance

You cannot inherit from multiple parent class.

# Encapsulation

* Technically in encapsulation, the variables or data of a class are hidden from any other class and can be accessed only through any member function of own class in which they are declared.
* As in encapsulation, the data in a class is hidden from other classes, so it is also known as data-hiding.
* Encapsulation can be achieved by: Declaring all the variables in the class as private and using C# Properties in the class to set and get the values of variables.

# Encapsulation vs Data Abstraction

* Encapsulation is data hiding (information hiding) while Abstraction is detail hiding(implementation hiding).

# Interfaces

Interface is a contract, what ever class that implements that interface/contract must have those certain things in contract.

We cannot instantiate interface.

# Dependency Injection

Use DI in .Net Core

.NET Core provided a built-in IoC Container that facilitates DI management. The IoC Container is responsible for helping automatic DI. It includes:

* **Registration**: The IoC Container ought to know which type of object to make for a specific dependency so, it delivers a way to map a type to a class so that it can make the correct dependency instance.
* **Resolution**: The IoC Container helps to resolve a dependency by making an object and injecting it into the requesting class. We do not have to instantiate objects manually to handle dependencies.
* **Disposition**: The IoC Container manages the lifetime of the dependencies.

1. Loosely coupled code
2. Maintainability
3. Testability
4. Re-usability

# IOC (Inversion of Control)

Inversion of Control (IoC) is a design principle (although, some people refer to it as a pattern).

# IQueryable vs IEnumerable

IQueryable inherits IEnumerable, so IQueryable does everything that IEnumerable does. IQueryable extends the IEnumerable with logic for querying data.

Yes, both will give you deferred execution.

The difference is that [IQueryable<T>](https://msdn.microsoft.com/en-us/library/bb351562.aspx) is the interface that allows LINQ-to-SQL (LINQ.-to-anything really) to work. So, if you further refine your query on an [IQueryable<T>](https://msdn.microsoft.com/en-us/library/bb351562.aspx), that query will be executed in the database, if possible.

For the [IEnumerable<T>](https://msdn.microsoft.com/en-us/library/9eekhta0.aspx) case, it will be LINQ-to-object, meaning that all objects matching the original query will have to be loaded into memory from the database.

In code:

IQueryable<Customer> custs = ...;

// Later on...

var goldCustomers = custs.Where(c => c.IsGold);

That code will execute SQL to only select gold customers. The following code, on the other hand, will execute the original query in the database, then filtering out the non-gold customers in the memory:

IEnumerable<Customer> custs = ...;

// Later on...

var goldCustomers = custs.Where(c => c.IsGold);

This is quite an important difference and working on [IQueryable<T>](https://msdn.microsoft.com/en-us/library/bb351562.aspx) can in many cases save you from returning too many rows from the database. Another prime example is doing paging: If you use [Take](https://msdn.microsoft.com/en-us/library/bb300906.aspx) and [Skip](https://msdn.microsoft.com/en-us/library/bb357513.aspx) on [IQueryable](https://msdn.microsoft.com/en-us/library/system.linq.iqueryable.aspx), you will only get the number of rows requested; doing that on an [IEnumerable<T>](https://msdn.microsoft.com/en-us/library/9eekhta0.aspx) will cause all of your rows to be loaded in memory.

# Delegates

A delegate is a type safe functional pointer. A delegate will point to a function and when the delegate is called upon the function will be executed.

Delegate bool **IsPromotable**(Employee emp)

**IsPromotable** ispromo = new **IsPromotable**(IsEligible);

Public bool **IsEligible** (Employee emp){

If (emp.Salary > 4000){

Return true;

}

Else {

Return false;

}

}

**isPromo** (new Employee () {

Salary = 5000,

Name = “Anish Shrestha”

})

Multicast delegate is a delegate that casts to more than one function.

# Linq Queries

LINQ queries use deferred execution. This means that the query doesn’t execute until you execute a foreach loop or call one of the standard query operators, like ToList, that requests the data.

# Async/Await: ConfigureAwait(false)

# Out vs Ref

|  |  |
| --- | --- |
| ref keyword | out keyword |
| It is necessary the parameters should initialize before it passes to ref. | It is not necessary to initialize parameters before it passes to out. |
| It is not necessary to initialize the value of a parameter before returning to the calling method. | It is necessary to initialize the value of a parameter before returning to the calling method. |
| The passing of value through ref parameter is useful when the called method also need to change the value of passed parameter. | The declaring of parameter through out parameter is useful when a method returns multiple values. |
| When ref keyword is used the data may pass in bi-directional. | When out keyword is used the data only passed in unidirectional. |

# Sealed Class:

* After a class is defined as sealed it cannot be inherited.
* After a method is defined as sealed it cannot be override.

## Websockets