|  |  |
| --- | --- |
| HTTP Request And it’s components | HTTP Response And it’s components |
| URL, Verbs, Headers, Body | Status code, |

**Verbs:**

**Get**

**Post**

**Put**: To update all properties(columns) e.g. update all info of a book

**Patch**: To update some of the Properties(columns) e.g. update all some of the info of a book

**Delete**: Two type:

**Soft Delete**: We generally delete one column of data using Patch

**Hard Delete**: Delete the entire record of data

**Status Code:** issued from the server and they gives the info about the response

**Note:** All Http response status code divide into 5 categories

* 1xx: info response
* 2xx: successful
* 3xx: Redirection
* 4xx: Client Error
* 5xx: Server error

**Frequently used status code:**

* 200 Ok
* 201 New Resource created
* 204 No Content (Server processed successfully and it is not returning any content)
* 301 Moved permanently ( the resource you are looking for is moved permanently)
* 302 Moved temporarily
* 400 Bad requests
* 401 Unauthorized
* 404 Not Found
* 405 Method not allowed( like if we have web api which accepts get method if we used put post or other it will show error)
* 500 Internal Server error (It could be of any reason line Db connection is not working or unhandled exception)
* 503 Service Unavailable (ex- down for maintenance)

**Steps to create Web API using command line Interface (.Net core 5):**

* To see the list of all type of projects and languages used: **dotnet new**
* To create a new api project with name myAPIProject: **dotnet new webapi --name myAPIProject**
* To built the project use the command: **dotnet build**
* To run the project use the command**: dotnet run**
* Now it will show the url for Project type the same in the browser:

e.g. <http://localhost:5000>

now use <http://localhost:5000/swagger> it will show some uI which Is swagger use rto test api like postman it is by default installed

**Route Constrains For the Url :**

We can Type of parameters

e.g. Route(“{id:int:min(10)}”)

Route(“{name:string:minlength(10)}”)

**Some of the Main Constrians Type**

* Type: int, bool, datetime, double, float etc;
* Min: min(number)
* Max: max(number)
* MinLength: minlength(20)
* MaxLength: maxlength(10)
* Length:length(20) only 20 characters are allowed
* Range: range(10,30)
* Alpha: Used to validate only alphabetical characters.

e.g. Route(“{id:length(10):alpha}”)

* Required: required
* Regex: regex(expression) e.g. Route(“{id:regex(a(b|c)) }”)

**Action Method return type:**

**Specific type** like : bool, int, string etc.

**Complex Data:** e.g. List, Employee, IActionResult, IActionResult<T>(it is Action Result With type)

**IActionResult:** it is an interface and used to return multiple type of data

**e.**g. if( data not found) return NotFound();

else return Ok(new List<Employee>(){

new Employee(){id =1, Name= “Emp1”},

new Employee(){id =2, Name= “Emp2”}

});

**IActionResult<T>:** if( data not found) return NotFound();

else return new List<Employee>(){

new Employee(){id =1, Name= “Emp1”},

new Employee(){id =2, Name= “Emp2”}

};

**Format Response with Status Code:**

**OK**(): returns retuns Ok status that is 200

**e.g.:** return Ok() //Only send the Ok status ie: 200

return Ok(“All Employee”) // Send data with Ok status

**Accepted():** status that is 202

**e.g.:** return Accepted() //Only send the Accepted status ie: 200

return Accepted (“~/api/test”) // Send data with Accepted status

**AcceptedAction():** status that is 202

**e.g.:** return AcceptedAction() //Only send the Accepted status ie: 200

return AcceptedActoion (“Name of Action Method”) // Send Complet url for the request with Accepted status

**AcceptedAtRoute():** We can give name to the route and it will show the name of route in response

**e.g.:** return AcceptedAtRoute() //Only send the Accepted status ie: 200

return AcceptedActoion (“Name of Action Method”) // Send Complet url for the request with Accepted status

**BadRequest():** It is used when there is an error in client data passed in url or other way

**e.g.:** return BadRequest () //400

**Conflict():** Is used to return 409 Status report

**Content():** is used to when we have formatted data in controller and you don’t want to format it aging then we use it.

**Created():** to return 201 when we created new record

**Dependency Injection:**

* It is used to implement IOC(Inversion of control)
* IOC means to have loosely coupling in the code
* Easy for unit testing

If we create object of class using new like a service of Email which contains Email related things

And we want to use that is all controller and if in future we make any change in Email class we have to make that change in all Controller methods

So the solution is Dependency Injection

Asp.Net Core framework provides the built- in support for DI

Dependency are register in container, and the container in asp.net core is IServiceProvider

**Singleton Service:**

* It is created by using AddSingleton<> method inside ConfigureServices() method of Startup.cs

Syntax to add service

services.AddSc<IProductRepository, ProductRepository>();

* There will be only one instance of the singleton service throughout the app
* For all the Http Request It will not create new instance all time it will use the same instance
* If we use the Singleton service then as the same instance is sheared so if we used some in memory data than it will not vanish old request data in new one

e.g.

//To stablished communication between controller and repository

private readonly IProductRepository \_productRepository;

private readonly IProductRepository \_productRepository1;

public ProductController(IProductRepository productRepository, IProductRepository productRepository1) //This is the constroctor of controller

//public ProductController()

{

//Here we created two different instance for IProductRepository

\_productRepository = productRepository;

\_productRepository1 = productRepository1;

//\_productRepository = new ProductRepository();

}

[HttpPost("")]

public IActionResult AddProduct([FromBody]ProductModel product)

{

\_productRepository.AddProduct(product); //Add data by one instance

var products = \_productRepository1.GetAllProducts(); //Get the data from another instance

return Ok(products);

}

**Scoped:**

* Scoped service can be registered using AddScoped<> method
* A new instance of the service will be created for new Http Request.
* It is opposite to Singleton Service
* In this each request new instance for each created instance for the particular object are same.

Syntax to add service

services.AddScoped<IProductRepository, ProductRepository>();

**Transient:**

* Transient service can be registered using AddTransient <> method

Syntax to add service

services. AddTransient <IProductRepository, ProductRepository>();

* In this each request new instance for each created instance for the particular object.
* A new Instance of the service is created every-time it is requested

e.g.

private readonly IProductRepository \_productRepository;

private readonly IProductRepository \_productRepository1;

public ProductController(IProductRepository productRepository, IProductRepository productRepository1)

//public ProductController()

{

//Two instance is created

\_productRepository = productRepository;

\_productRepository1 = productRepository1;

//\_productRepository = new ProductRepository();

}

[HttpPost("")]

public IActionResult AddProduct([FromBody]ProductModel product)

{

\_productRepository.AddProduct(product); //Add data by one instance

//It will not show data as both instance are different

var products = \_productRepository.GetAllProducts(); //Get the data from another instance

return Ok(products);

}

**Entity FrameWork:**

* Is a open source framework for Microsoft that works as a bridge in between the application and the database.
* The Entity framework core is compatible with any dot net application and any RDBMS.
* It is the Microsoft’s official technology to interact with the RDBMS
* It is an O/RM and work on the object-oriented perspective
* This means all the tables in Db are converted into c# classes and the corresponding columns are converted as the properties of the classes

**Two ways to add Entity**

1. **Code First:** In this we first create Classes in C# and table in Db automatically added automatically
2. **Database First:** IN this we have database and we creates all classes related to database in the c# using some commands of Entity framework

**To interact with Database**

We have to use add command in nuget package console

i.e. add-migration init //Note Init is name we can use any name here it will create the same Migration class name in solution

It will create a new folder Migration in Solution and add class with the name init

Whenever we make some change in code that is ready to go in Database evertime we have to generate Migration to update in Database

**To update we have to run one more command**

i.e. update-database

**HTTP Put Vs Patch:**

* **Put** Is used to update the entire record of Data Like If there is a book record {id: 1, Name: Java, Description: This is for Java} and we want to updata all every thing of the book then we use Put and end result will become like {id: 1, Name: Javas Updated, Description: This is for Javas Updated}
* But if we want to update only book description we only pass new description then it will put null for the Name
* **Patch** So in this scenario Patch is better approach to updata record partially based on send data in query then it will not change Name of book to the null

**To use Patch In MVC Core App**

We need to install to packages

1. Microsoft.AspNetCore.JsonPatch
2. Microsoft.AspNetCore.Mvc.NewtonSoftJson

**Once Http Patch Is installed**

We have to use AddNewtonsoftJson() in the startup class

For Patch to send data we have to use some different way we have pass Array of field data with new value

URL: <https://localhost:44350/api/Books/4> //4 is the id of book for which we want to update data

[

    {

        "op": "replace", //It is operatio

        "path":"title", //What you want to change (column Name)

        "value": "Javascript" //New value

    },

{

        "op": "replace", //It is operatio

        "path":"column2", //What you want to change (column Name)

        "value": "Value for column2" //New value

    },

{

        "op": "remove", // operation to remove from Author column

        "path":"Author"

    },

]

**Automapper:**

1. It is used to map properties of one type to another type
2. Like we have many scenario when we want to Map one class of data to another type then we have to do it manually but if we have many properties to map than it the chances of error increases as we may left few properties to map or map one properte to wrong one

e.g.

In the following example we get BookModel type data and convert to Book type data for that we have to map each property one by one

public async Task<int> AddBookAsync(BookModel bookModel)

{

//Getting all books Data and converting into BookModel type by selecting each Book Data

var book = new Books()

{

Title = bookModel.Title,

Description = bookModel.Description

};

\_context.Books.Add(book);

await \_context.SaveChangesAsync();

return book.Id;

}

**To Solve this problem we use Automapper**

It is a package that will do all mapping thing automatically we don’t need to map one by one

We have to install

AutoMapper.Extensions.Microsoft.DependencyInjection

**In startup file add**

services.AddAutoMapper(typeof(Startup));

**Now Create one class to define what are the mapping Table**

public class ApplicationMapper : Profile

{

public ApplicationMapper()

{

//Note : Reverse method is used to we can map Books to BookModel as well as BookModel to Books

CreateMap<Books, BookModel>().ReverseMap();

}

}

**Add Dependency Injection for mapper in the constructor**

private readonly BookStoreContext \_context;

private readonly IMapper \_mapper;

public BookRepository(BookStoreContext context, IMapper mapper)

{

\_context = context;

\_mapper = mapper;

}