Contents

[Application Architecuture 2](#_Toc109832499)

[Creating Web API Project and Exploring default files 3](#_Toc109832500)

[launchSettings.json 4](#_Toc109832501)

[.csproj file 5](#_Toc109832502)

[Converting an Console Core Application to Web API project 6](#_Toc109832503)

[Project .csproj file level changes 6](#_Toc109832504)

[Program.cs file level changes 6](#_Toc109832505)

[Startup.cs file level changes 6](#_Toc109832506)

[Host Builder 7](#_Toc109832507)

[CreateDefaultBuilder 7](#_Toc109832508)

[ConfigureWebHostDefaults 7](#_Toc109832509)

[Startup Class 7](#_Toc109832510)

[Core Concepts 7](#_Toc109832511)

[Controller 7](#_Toc109832512)

[Middleware and HTTP Request Pipeline 8](#_Toc109832513)

[Http Request Pipeline 8](#_Toc109832514)

[Middleware 9](#_Toc109832515)

[Middleware methods 9](#_Toc109832516)

[Implementing Custom Middleware 11](#_Toc109832517)

[Routing 11](#_Toc109832518)

[How to Enable Routing 11](#_Toc109832519)

[Working with Variables in Routing 12](#_Toc109832520)

[Query String in Routing 12](#_Toc109832521)

[Multiple URLs for Single Resource 12](#_Toc109832522)

[Token Replacement in Routing 13](#_Toc109832523)

[Set the Base Route at the Controller level 13](#_Toc109832524)

[Route Constraint 14](#_Toc109832525)

[Return Types 15](#_Toc109832526)

[Specific Return Type 15](#_Toc109832527)

[IActionResult 15](#_Toc109832528)

[Model Binding 15](#_Toc109832529)

[Bind Property 16](#_Toc109832530)

[BindProperties 17](#_Toc109832531)

[Dependency Injection (DI) 17](#_Toc109832532)

[Lifetime of services in DI 18](#_Toc109832533)

[Singleton 18](#_Toc109832534)

[Scope 18](#_Toc109832535)

[Transient 18](#_Toc109832536)

[TryAddScoped, TryAddSingleton, TryAddTransient 19](#_Toc109832537)

[Use Case 19](#_Toc109832538)

[Resolving Service Dependency in Action Method 19](#_Toc109832539)

[Project: Grocery Store 20](#_Toc109832540)

[ASP.NET Core Web APP 20](#_Toc109832541)

[GroceryDB 20](#_Toc109832542)

[Entity Framework Core 20](#_Toc109832543)

[Code First Approach 21](#_Toc109832544)

[Context Class 21](#_Toc109832545)

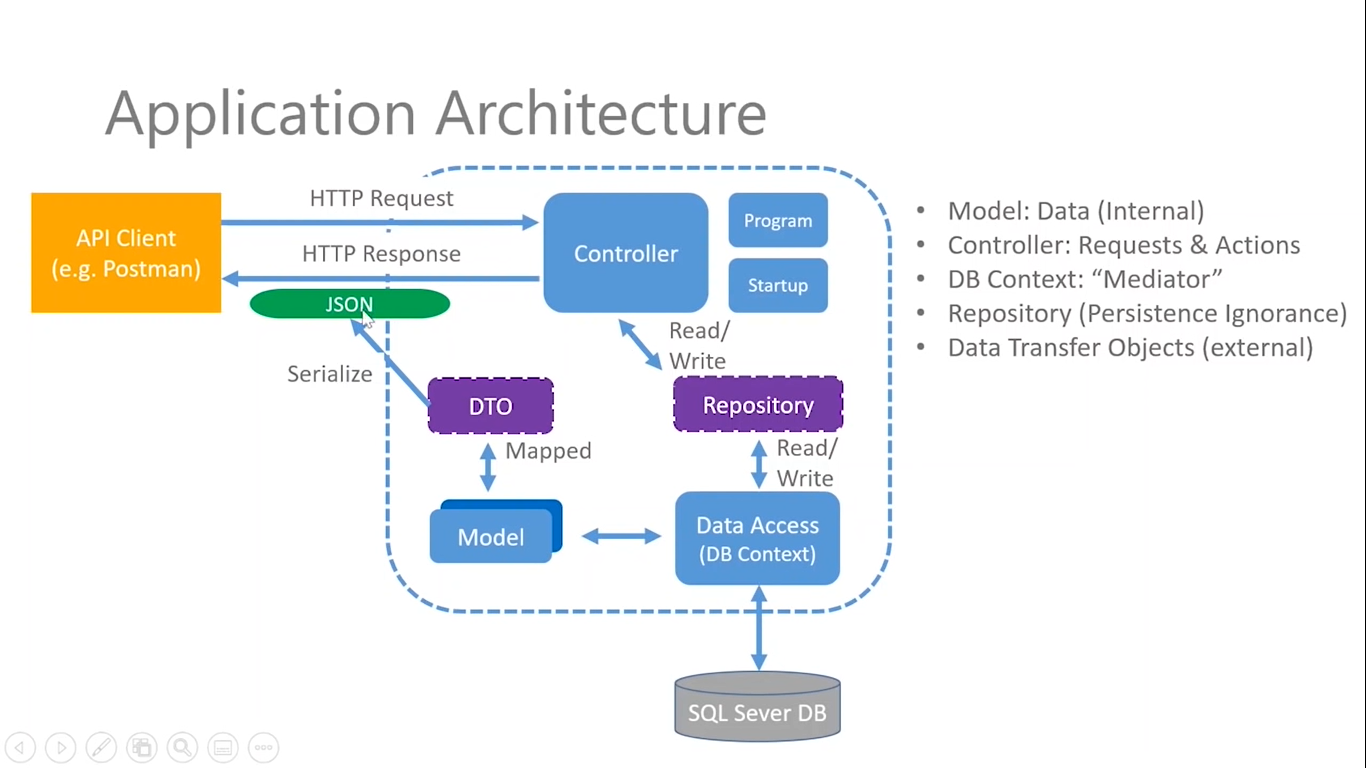
[Connection String 21](#_Toc109832546)

[Register the Context as service 21](#_Toc109832547)

[Migration 21](#_Toc109832548)

[Adding Auto Mapper 22](#_Toc109832549)

# Application Architecuture



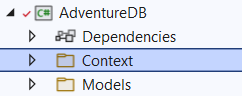
**Entity Framework Core**

|  |
| --- |
| PM> Scaffold-DbContext "Data Source=DESKTOP-PD9SE9U\SQLEXPRESS01;Initial Catalog=AdventureWorks2019;Integrated Security=True" Microsoft.EntityFrameworkCore.SqlServer -OutputDir Models -ContextDir Context -Context AdventureContext -force |

Replacing above hard coded connection string with Appsettings.json connection

|  |
| --- |
| "ConnectionStrings": {  "AdventureDb": "Data Source=DESKTOP-PD9SE9U\\SQLEXPRESS01;Initial Catalog=AdventureWorks2019;Integrated Security=True"  }, |

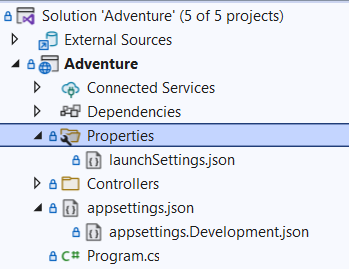
|  |
| --- |
| Scaffold-DbContext -Connection Name=AdventureDb Microsoft.EntityFrameworkCore.SqlServer -OutputDir Models -ContextDir Context -Context AdventureContext -Force |



# Creating Web API Project and Exploring default files

Creating a web api project. Below is the default project structure will get created

Let’s discuss about one by one



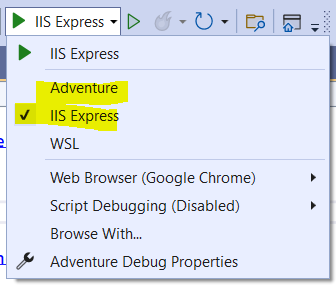
## launchSettings.json

As .net core is capable of running on any OS, below configuration will useful to run the web api.

If it is running on windows application, system will use IIS express setting, if it is running on other OS, system will use kestrel web server.

|  |
| --- |
| {  "$schema": "https://json.schemastore.org/launchsettings.json",  "iisSettings": {  "windowsAuthentication": false,  "anonymousAuthentication": true,  "iisExpress": {  "applicationUrl": "http://localhost:46176",  "sslPort": 44369  }  },  "profiles": {  "Adventure": {  "commandName": "Project",  "dotnetRunMessages": true,  "launchBrowser": true,  "launchUrl": "swagger",  "applicationUrl": "https://localhost:7209;http://localhost:5209",  "environmentVariables": {  "ASPNETCORE\_ENVIRONMENT": "Development"  }  },  "IIS Express": {  "commandName": "IISExpress",  "launchBrowser": true,  "launchUrl": "swagger",  "environmentVariables": {  "ASPNETCORE\_ENVIRONMENT": "Development"  }  }  }  } |

We can switch the web server using below



## .csproj file

When we double click on project, it will open a file call .csproj file (other way to open is Right Click on project and select Edit project file). It will gives the information about

* Targeting SDK
* Packages installed
* Project references and other project specific details

|  |
| --- |
| <Project Sdk="Microsoft.NET.Sdk.Web">  <PropertyGroup>  <TargetFramework>net6.0</TargetFramework>  <Nullable>enable</Nullable>  <ImplicitUsings>enable</ImplicitUsings>  <UserSecretsId>sdfdsf-ewrwe-rtytu-</UserSecretsId>  </PropertyGroup>  <ItemGroup>  <PackageReference Include="Microsoft.EntityFrameworkCore.Design" Version="6.0.5">  <PrivateAssets>all</PrivateAssets>  <IncludeAssets>runtime; build; native; contentfiles; analyzers; buildtransitive</IncludeAssets>  </PackageReference>  <PackageReference Include="Swashbuckle.AspNetCore" Version="6.2.3" />  </ItemGroup>  <ItemGroup>  <ProjectReference Include="..\AdventureRepository\AdventureRepository.csproj" />  <ProjectReference Include="..\AdventureService\AdventureService.csproj" />  </ItemGroup>  </Project> |

# Converting an Console Core Application to Web API project

## Project .csproj file level changes

* Change the Project SDK to web
* Remove the Output Type
* Verify the target framework moniker (TFM)

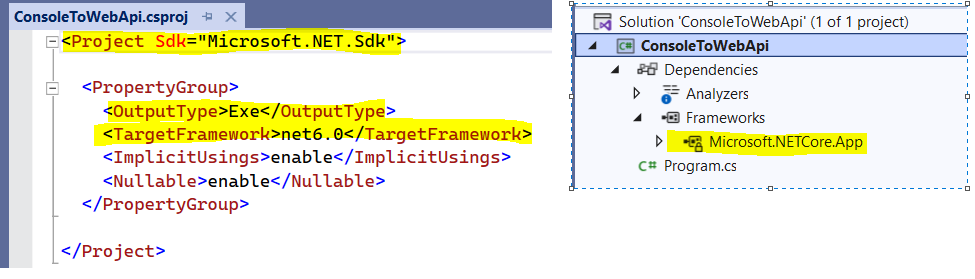
## Program.cs file level changes

* Add the Web Host Builder
* Configure the startup class

## Startup.cs file level changes

* Add Routing
* Set Default Route

Step#1: Below is the default project.config file and framework from console app



Step#2: lets modify the project sdk, TargetFramework and remove the OutypeType file.

After modification will observe another framework will get added with the name Microsoft.ASPNETCore.App



Program.cs file changes

Modified Class

|  |
| --- |
| static void Main(String[] arrgs)  {  CreateHostBuilder(arrgs);  }  public static IHostBuilder CreateHostBuilder(String[] arrgs) =>  Host.CreateDefaultBuilder(arrgs).ConfigureWebHostDefaults(webBuilder =>  {  webBuilder.UseStartup<Startup>();  }); |

### Host Builder

Host Builder is an object that is used to add some default features in the application

### CreateDefaultBuilder

Below are some of the responsibilities of CreateDefaultBuilder

* Enable scope validation on the **Dependency Injection (DI)** container
* Set the **ContentRootPath** to the result of System.IO **GetCurrentDirectory**
* Load app configuration from **appsettings.json,command line args, Environment variables**
* Load app configuration from **User Secrets** when environment set to “**Development**”
* Configure the **ILoggerFactory** to **log** the console, debug and event source output

### ConfigureWebHostDefaults

* Provides support for HTTP
* Use Kestrel as the web server and configure it using the application configurations providers
* Enables the IIS integration
* Adds the HostFiltering middleware
* Adds the ForwardedHeaders middleware if ASPNETCORE\_FORWARDEDHEADER\_ENABLED=true

### Startup Class

* Startup class have two important methods
  + ConfigureServices
  + Configure
* **ConfigureServices**: This method will take IServiceCollection as an parameter, which is responsible for registering the services(Contract)
* **Configure**: This method will take IApplicationBuilder and IWebHostEnvironment as a paramters, which is responsible for configuring the application requests (pipeline) and provide the information about web hosting environment in which the application in running

# Core Concepts

## Controller

* A controller class in webapi has a “controller” suffix
* The controller class must inherited from **ControllerBase**
  + **ControllerBase** class provider’s basic methods and properties to handle HTTP request.
* Use **ApiController** attribute on the controller, it is responsible for
  + Attribute Routing requirements
  + Handle the client errors like 400 status code etc
  + Multipart/Form data request interface
  + Bind the incoming data with the parameters using some more attributes
* Use Attribute Routing
  + Attribute Routing helps us to define route specific to method

Code sample

|  |
| --- |
| [ApiController]  [Route("test/[action]")]  public class TestController : ControllerBase  {  public string Get()  {  return "Hello From Get";  }  public string Get1()  {  return "Hello From Get1";  }  } |

Urls:

<https://localhost:44369/test/get>

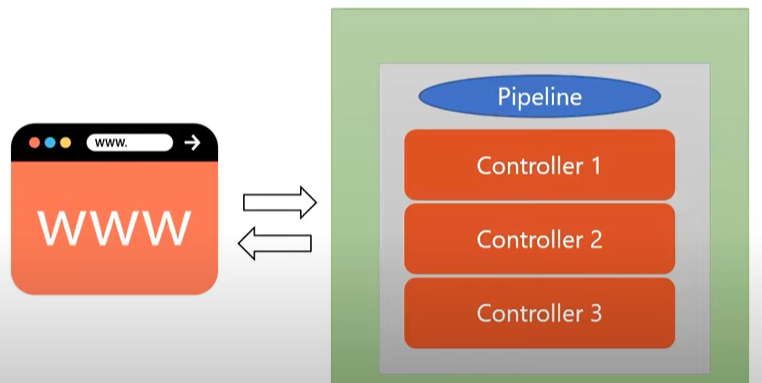
<https://localhost:44369/test/get1>

# Middleware and HTTP Request Pipeline

**Background**: let’s understand the basic flow, when we request a specific url from browser.

In general, we assume that whenever we make a request to the server specific url, a particular controller action method will invoke and processed with output.

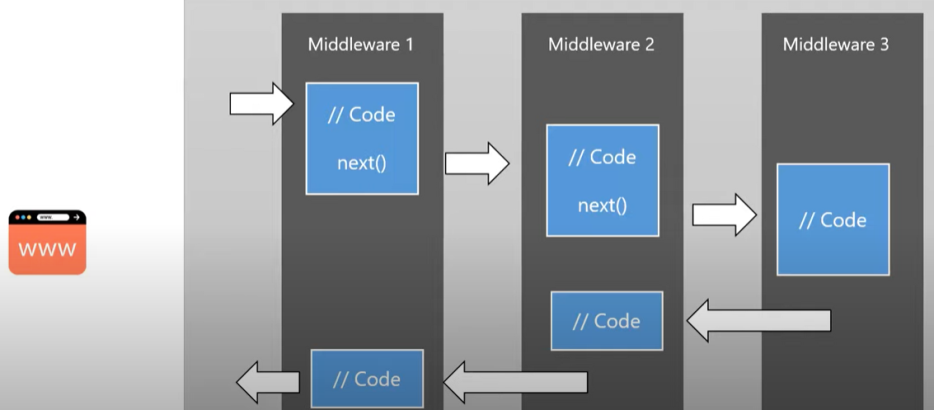
But in reality, we have the Pipeline for Http Requests.



## Http Request Pipeline

We need to configure the middleware in the respective order of their execution, so middleware will internally call the other middleware’s. Upon completing the configuration pipeline the middleware will run.

Note: The order of execution is matters a lot.



## Middleware

* Middleware is the piece of code that is used in the Http request pipeline
* An Asp.net core web application can have “N” number of middleware
* Order of execution matters a lot during execution

Examples of some predefined middleware are Routing, Authentication, Add Error Page etc

### Middleware methods

**Run()**

This method is used to **complete** the middleware execution

**Microsoft - Adds a terminal middleware delegate to the application request pipeline**

Prefer using the Run() as below for better performance

|  |
| --- |
| app.Use((context, next) =>  // {  // return next(context);  // }); |

**Use()**

This method is used to **insert a new** middleware in the pipeline

Microsoft – Adds a middleware delegate define in-line to the application request pipeline

**Next()**

This method is used to **pass** the execution to the next middleware

**Map**(): This method is used to **map** the middleware to the specific URL

Program.cs (in VS2022) Startup.cs (below VS2022)

**Run() Demo**

|  |
| --- |
| app.Run(async option =>  {  await option.Response.WriteAsync("Hello from Run");  });  app.Run(async option =>  {  await option.Response.WriteAsync("Hello from Run 2");  }); |

Output: Hello from Run

Explanation: As said above, Run() method will complete/terminate the application request pipeline. Hence, system will not go for next configured middleware after Run() executed.

**Use() & Next() Demo**

|  |
| --- |
| app.Use(async (context, next) =>  {  await context.Response.WriteAsync("Hello from Use 1 \n");  await next();  });    app.Run(async option =>  {  await option.Response.WriteAsync("Hello from Run \n");  }); |

Output:

Hello from Use 1

Hello from Run

Case Study#1

|  |
| --- |
| app.Use(async (context, next) =>  {  await context.Response.WriteAsync("Hello from Use 1-1 \n");  await next();  await context.Response.WriteAsync("Hello from Use 1-2 \n");  });  app.Run(async option =>  {  await option.Response.WriteAsync("Hello from Run \n");  }); |

Output:

Hello from Use 1-1

Hello from Run

Hello from Use 1-2

Case Study#2

|  |
| --- |
| app.Use(async (context, next) =>  {  await context.Response.WriteAsync("Hello from Use 1-1 \n");  await next();  await context.Response.WriteAsync("Hello from Use 1-2 \n");  });  app.Use(async (context, next) =>  {  await context.Response.WriteAsync("Hello from Use 2-1 \n");  await next();  await context.Response.WriteAsync("Hello from Use 2-2 \n");  });  app.Run(async option =>  {  await option.Response.WriteAsync("Hello from Run \n");  }); |

Output:

Hello from Use 1-1

Hello from Use 2-1

Hello from Run

Hello from Use 2-2

Hello from Use 1-2

Explanation

As discussed in above Http Request Pipeline, Use() method is used to add the middleware, next() is use to pass execution to next middleware and Run() is use to complete middleware execution.

In the above use case #1,#2. System added middleware in order for Use 1-1 & 1-2 (by calling next())

Once system is complete the Run() execution, then control is travelling back from Use 2-2 & 2-1. Please refer above Http Request Pipeline for execution flow.

# Implementing Custom Middleware

Below is the step to create a custom middleware

Step#1: Create a class which Implement **IMiddleware**

Step#2: Provide implementation to the methods of **IMiddleware** interface

Step#3: Dependency Injection (DI) add services to the container (in ConfigureService() method)

Step#4: Configure Middle in Http Request pipeline (in Configure method using .UseMiddleware())

Step#1 & #2

|  |
| --- |
| public class CustomDemoMiddleware : IMiddleware  {  public async Task InvokeAsync(HttpContext context, RequestDelegate next)  {  await context.Response.WriteAsync("Custom: Hello from Use 2-1 \n");  await next(context);  await context.Response.WriteAsync("Custom: Hello from Use 2-2 \n");  }  } |

Step#3

|  |
| --- |
| builder.Services.AddTransient<CustomDemoMiddleware>(); |

Step#4

|  |
| --- |
| app.UseMiddleware<CustomDemoMiddleware>(); |

# Routing

Routing is the process of mapping an incoming http request (URL) to a particular resource (the action method)

* We can access the any resource using the unique URL
* A resource can have multiple unique URL
* Multiple resources cannot have same URL

## How to Enable Routing

In Asp.Net Core web api we can enable routing through middleware.

We need to insert following two middleware into http request pipeline

* UseRouting() – using this method will just enable the routing in web api
* UseEndPoint() – this is use to configure the routing path or the controller

|  |
| --- |
| app.UseRouting();  app.UseEndpoints(endpoints => {  endpoints.MapControllers();  }); |

|  |
| --- |
| [Route("api/get")]  public string Get()  {  return "Hello From Get";  }  [Route("api/getAll")]  public string GetAll()  {  return "Hello from GeAll";  } |

url: <https://localhost:44369/api/get> & <https://localhost:44369/api/get1>

## Working with Variables in Routing

We need to pass the variable value in curly braces.

|  |
| --- |
| [Route("api/getbyid/{id}")]  public string GetById(int id)  {  return "Hello from " + id;  }  [Route("api/getbyid/{id}/author/{name}")]  public string GetById(int id, string name)  {  return "Hello from " + id +" " +name;  } |

URLs -

<https://localhost:44369/api/GetById/10>

<https://localhost:44369/api/GetById/10/author/100>

## Query String in Routing

If we do not want to pass all the parameters in the attribute routing, then we can go for Query string

|  |
| --- |
| [Route("search")]  public string SearchBook(int id, string name = "", string author = "", string publication="", int rating=0)  {  return "Book Search criteria";  } |

url: <https://localhost:44369/test/search?id=4&name=lgr&author=aa&publication=new&rating=4>

## Multiple URLs for Single Resource

Having multiple URLs is possible for the single resource until they are unique.

|  |
| --- |
| [Route("api/getall")]  [Route("get-all")]  [Route("getall")]  public string GetAll()  {  return "Hello from GeAll";  } |

<https://localhost:44369/test/api/getall>

<https://localhost:44369/test/get-all>

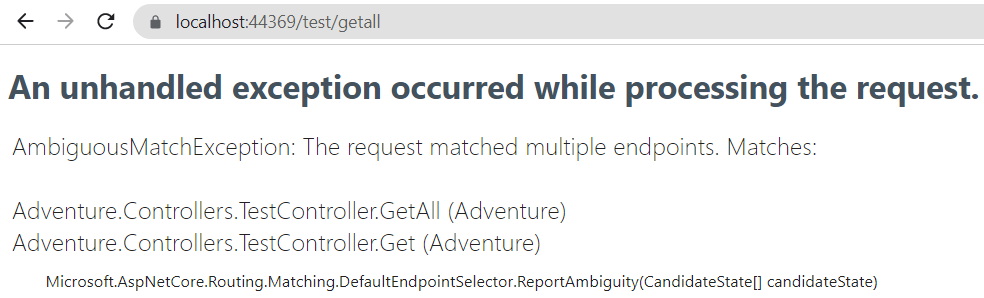
<https://localhost:44369/test/getall>

**Note**:

It is not possible to have same URL for multiple resources.

Having same url for multiple resources will lead in Runtime error while trying to access the particular resource.

|  |
| --- |
| [Route("api/getall")]  [Route("get-all")]  [Route("getall")]  public string GetAll()  {  return "Hello from GeAll";  }  [Route("api/get")]  [Route("getall")] // This is not possible  public string Get()  {  return "Hello From Get";  } |



## Token Replacement in Routing

It is the process of configuring the controller and action (Area if exists) to the action methods. We can specify the Token replacement in any order as below

|  |
| --- |
| [Route("get-all")]  [Route("[controller]/[action]")]  public string GetAll()  {  return "Hello from GeAll";  }  [Route("api/get")]  [Route("[controller]/[action]")]  public string Get()  {  return "Hello From Get";  } |

[Route("[controller]/[action]")] //<https://localhost:44369/test/getall>

[Route("[action]/ [controller]")] //<https://localhost:44369/getall/test>

## Set the Base Route at the Controller level

We can set the base route at controller level instead of duplicating at each method level

The route define at the controller level will be treated as common route for all the method. If we wish to pass any parameter to any method, we can decorate the particular method with the specific parameter route.

If we want to ignore any of the particular method from base routing, just use the “~” symbol and default your own route.

|  |
| --- |
| [ApiController]  [Route("[controller]/[action]")]  public class TestController : ControllerBase  {  public string Get()  {  return "Hello From Get";  }  [Route("~/getall")] // ~ restrict to follow base routing  public string GetAll()  {  return "Hello from GeAll";  }  [Route("{id}")] // only specify the parameter which we want to send  public string GetById(int id)  {  return "Hello from " + id;  }  } |

<https://localhost:44369/test/get>

https://localhost:44369/getall

<https://localhost:44369/test/getByID/10>

## Route Constraint

It is use to define constraint on the attribute level routing. Some of the constraints are

* Type: int, book, datetime, double, float etc
* Min: min number
* Max: max number
* MinLength: minlength(10)
* MaxLength: maxlength(100)
* Length: length(12)
* Range: range(10,15)
* Alpha: to allow only alphabets characters
* Regex: to enhance regular expression capabilites

|  |
| --- |
| [Route("{id:int:range(10,15)}")]  public string GetById(int id)  {  return "from GetById int " + id;  }  [Route("{id:length(3):alpha}")]  public string GetByName(string id)  {  return "from GetById string " + id;  }  [Route("{id:regex(a(b|c))}")]  public string GetByIndex(string id)  {  return "from GetById regex " + id;  } |

# Return Types

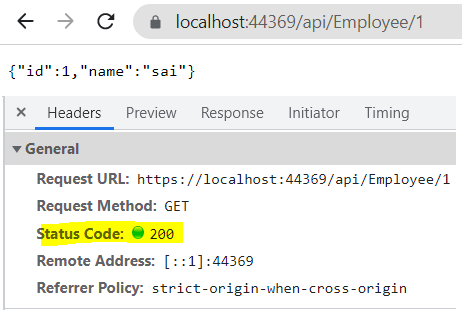
## Specific Return Type

* like int, string, object, bool etc

## IActionResult

* It will allow to return the multiple return types and provide support with many in build methods like NotFound(), Ok() etc
* Return type Ok(): Creates a Microsoft.AspNetCore.Mvc.OkResult object that produces an empty StatusCodes with Status 200.

|  |
| --- |
| [Route("{id}")]  public IActionResult GetEmployeeById(int id)  {  Employee employee = new() { Id = 1, Name = "sai" };  return Ok(employee);  } |



* Similarly we have other action methods like Accepted(), AcceptedRoute() which return 202 status code

# Model Binding

The process of binding HTTP Request data to the parameters of the application controller or properties is known as model binding

There are lot of pre-defined methods for model binding; we can also create our own custom model binder.



## Bind Property

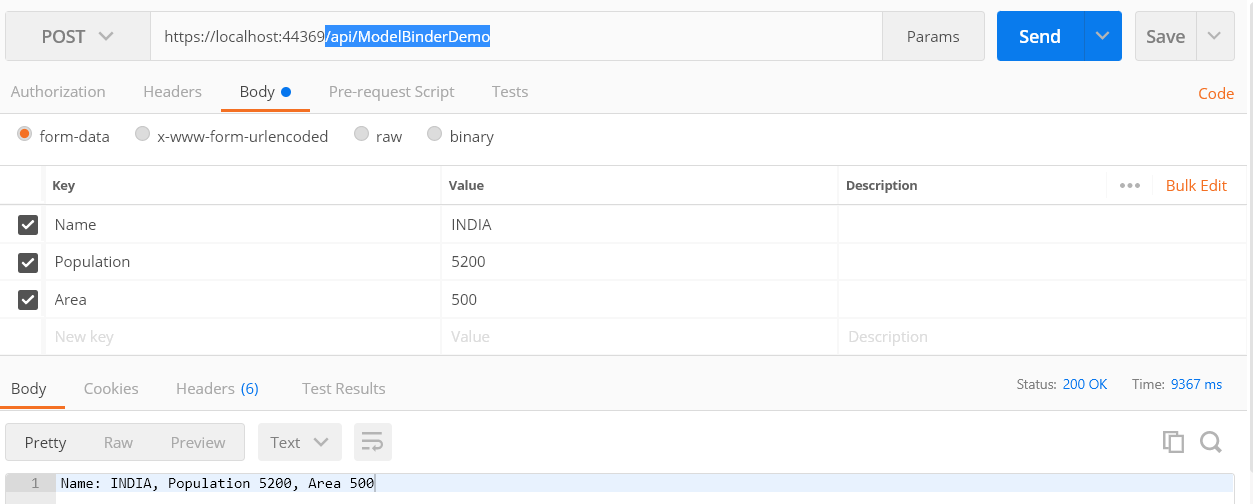
Bind property is an attribute that is used to bind the incoming form-data to the public properties

Bind property is applied on each property individually

[**BindProperty**] – an attribute that is used to specify a model name to be use for property binding.

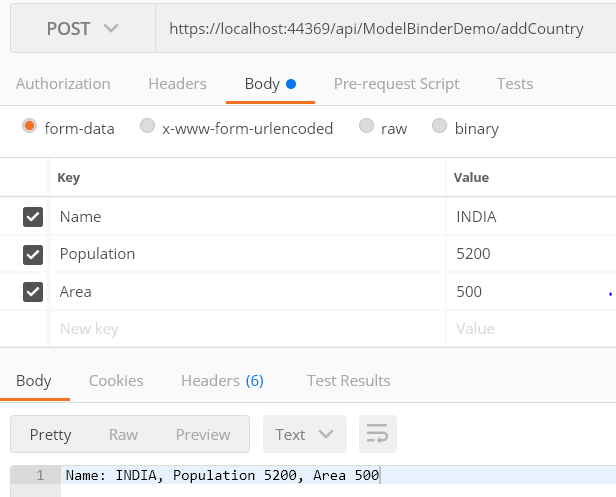
1. Enabling Bind Property at public properties level

|  |
| --- |
| [BindProperty]  public string Name { get; set; }  [BindProperty]  public int Population { get; set; }  [BindProperty]  public int Area { get; set; }  [HttpPost]  public IActionResult Index()  {return Ok("Name: " +this.Name + ", "+"Population " + this.Population + ", "+ "Area " +this.Area);  } |



1. Enabling [BindProperty] at class level

|  |
| --- |
| [BindProperty]  public Country Country { get; set; } |



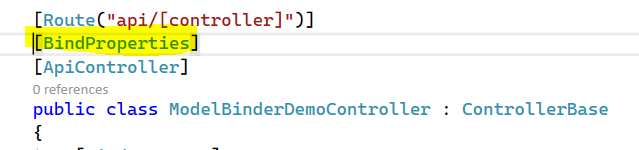
By Default, Bind Property will not work for Get Request.=, to work we need to configure

**Note**: Here, we can see that BindProperty is applied for simple datatypes like (int, float etc) and complex types like Model class.

## BindProperties

As like BindProperty, Bind Properties also can applies to simple data types or complex types like method. And by default, Get requests are not enable.

BindProperties can also applied at Controller level (but not bindProperty) to enable property binding.



# 

# Dependency Injection (DI)

* The main concept behind dependency injection is to implement Inversion of Control (IOC)
* IOC means to have loosely coupling in code
* Unit Testing makes easy with dependency injection
* ASP.Net core provides build in support for dependency injection
* Dependency are registered in containers, and the container in asp.net core is IServiceProvider
* Services are registered in the Startup.ConfigureService method of the application

# Lifetime of services in DI

## Singleton

* Singleton services can be registered using AddSingleton() method
* There will be only one instance of the Singleton service throughout the application

|  |
| --- |
| private readonly IItemRepository \_itemRepository;  private readonly IItemRepository \_itemRepository2; // just to demonstrate Singleton  public ItemController(IItemRepository itemRepository, IItemRepository itemRepository2)  {  this.\_itemRepository = itemRepository;  this.\_itemRepository2 = itemRepository2;  }  [HttpPost("addItem")]  public IActionResult AddItem([FromBody] ItemModel item)  {  \_itemRepository.AddItem(item);  var items = \_itemRepository2.GetItems();  return Ok(items);  } |

|  |
| --- |
| builder.Services.AddSingleton<IItemRepository, ItemRepository>(); |

With AddSingleton, the service scope is throughout the application, even if we created multiple instances

## Scope

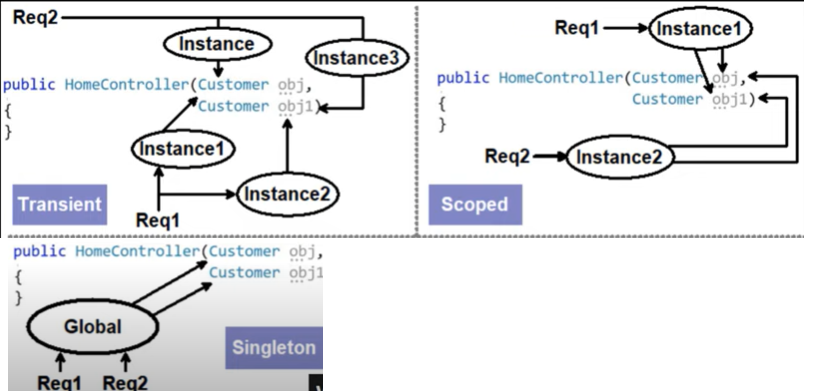
* Scoped services can be registered using AddScoped() method
* A new instance of the service will be created for new Http request

|  |
| --- |
| builder.Services.AddScoped<IItemRepository, ItemRepository>(); |

## Transient

* Transient service can be registered using AddTransient() method
* A new instance of the service will be created every time it is requested.
* Let us say, Controller “C” is using a Service “S” 3 times in the same http request, then there will be 3 separate instance of service “S”. which means, as above (scoped & Singleton) we cannot share the operation with other instance of service.

|  |
| --- |
| builder.Services.AddTransient<IItemRepository, ItemRepository>(); |



# TryAddScoped, TryAddSingleton, TryAddTransient

* If we registered the dependency injection using AddScoped, AddSingleton or AddTransient method they will replace the services in the collections if it has already being registered in the container.
* Now, if we are using “Try” version like TryAddScoped (or TryAddSingleton, TryAddTransient) then it will only register the services if it was not registered earlier. If the services are already registered then the TryAddScoped (or TryAddSingleton, TryAddTransient) will skip the registration for that service.

## Use Case

* With the below registration, ProductRepository replace the ItemRepository implementation.
* When the particular controller action methods are invoke, it will result in wrong output (as in controller will have interface implementation, but not the implementing class details)

|  |
| --- |
| builder.Services.AddTransient<IItemRepository, ItemRepository>();  builder.Services.AddTransient<IItemRepository, ProductRepository>(); |

* Using the below “Try” version will skip the registration ProductRepository, as IItemRepository is already register with ItemRepository.

|  |
| --- |
| builder.Services.TryAddScoped<IItemRepository, ItemRepository>();  builder.Services.TryAddScoped<IItemRepository, ProductRepository>(); |

* Go for “Try” version if we don’t want to override existing registration.

## Resolving Service Dependency in Action Method

* Suppose if we have only one or two methods which requires dependency to inject, we can inject dependency directly through action methods. (as injecting dependencies through constructor result in accessing dependencies throughout class). But injecting dependencies through method will limit to class only.
* We use the attribute [FromServices] to resolve the dependency.

|  |
| --- |
| [HttpGet("name")]  public IActionResult GetName([FromServices] IItemRepository \_itemRepository)  {  var result = \_itemRepository.GetItemName();  return Ok(result);  } |

# Project: Grocery Store

## ASP.NET Core Web APP

* Select the template while creating a new project

## GroceryDB

* Create a Class Library project

### Entity Framework Core

* Entity framework core is an open source framework from Microsoft that works as a bridge between the application and the database.
* Entity framework core is compatible with any dot net application and any RDBMS
* Its Microsoft official technology to interact with RDBMS
* Working approaches
  + Code First
  + Database First
* Requires below libraries to be added to work with Entity Framework core
  + Microsoft.EntityFrameworkCore
  + Microsoft.EntityFrameworkCore.Relational
  + Microsoft.EntityFrameworkCore.SqlServer
  + Microsoft.EntityFrameworkCore.Tools
  + Microsoft.EntityFrameworkCore.Design
* Also, add the Microsoft.EntityFrameworkCore.Design in startup project

# Code First Approach

## Context Class

Create a DB Context class and add the models class which we want to generate as tables.

|  |
| --- |
| public class GroceryContext : DbContext  {  public GroceryContext(DbContextOptions<GroceryContext> options) : base(options)  {  }  public DbSet<Items> Item { get; set; }  public DbSet<Products> Product { get; set; }  } |

## Connection String

* Add the connection string to database which we want to connect

|  |
| --- |
| "ConnectionStrings": {  "GroceryDb": "Server=DESKTOP-PD9SE9U\\SQLEXPRESS01;Database=Grocery;Integrated Security=True"  } |

## Register the Context as service

* Register the DB Context class as a service providing connection string

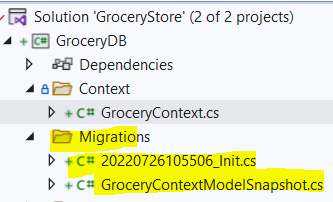
|  |
| --- |
| builder.Services.AddDbContext<GroceryContext>(  options =>  options.UseSqlServer(builder.Configuration.GetConnectionString("GroceryDb"))  ); |

## Migration

For the code first approach, we need to write the migration commands to generate the database and tables.

Execute the below commands in Package Manager Console

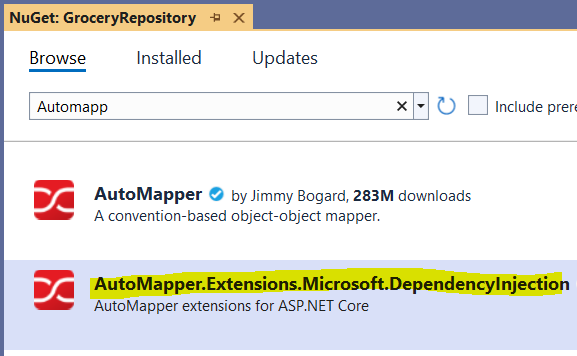
|  |
| --- |
| PM> Add-Migration Init   * This will generate the migration folder with model snapsho |



|  |
| --- |
| PM> Update-Database   * This command is responsible for refreshing the database (create or alter) tables and database |

# Adding Auto Mapper

* Download and install “AutoMapper.Extensions.Microsoft.DependencyInjection” which is specific for ASP.NET core



* Create a class and inherit Profile class of auto mapper

|  |
| --- |
| public class AppMapper : Profile  {  public AppMapper()  {  CreateMap<ItemModel,Items>().ReverseMap();  }  } |

CreateMap method will take source and destination mapping class as inputs.

If we want to map the classes in both the ways, then we need to use .ReverseMap()

* Register auto mapper as dependency

|  |
| --- |
| builder.Services.AddAutoMapper(typeof(AppMapper)); |

* Injecting auto mapper dependency and consuming auto mapper

|  |
| --- |
| private readonly GroceryContext \_context;  private readonly IMapper \_appMapper;  public ItemRepository(GroceryContext context, IMapper appMapper)  {  \_context = context;  \_appMapper = appMapper;    }  public async Task<List<ItemModel>> GetAllItems()  {  var result = await \_context.Item.ToListAsync();  return \_appMapper.Map<List<ItemModel>>(result);  }  public async Task<ItemModel> GetItemById(int id)  {  var result = await \_context.Item.FindAsync(id);  return \_appMapper.Map<ItemModel>(result);  } |