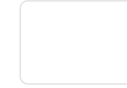Hosting

. . .

Onion Architecture in ASP.NET Core with CQRS : Detailed & Illustrated

Updated on: December 18, 2023

Privacy

Computer Engineer Jeffrey Palermo created Onion Architecture for creating app which are highly maintainable and loosely coupled. This architecture somewhat resembles the layers which can be seen on cutting an onion vegetable. You can separate the layers of onion very easily and eat them in your salads. In the same way the layers of Onion Architecture are separatable as they are loosely coupled, and this gives a highly testable architecture.

In this tutorial I will be Implementing Onion Architecture in ASP.NET Core with CQRS. You can download the complete source code from my GitHub Repository.

Page Contents

What is Onion Architecture

Layers in Onion Architecture for an ASP.NET Core app

      Domain Layer in Onion Architecture

      Application Layer in Onion Architecture

      Infrastructure + Presentation Layer in Onion Architecture

Onion Architecture vs Clean Architecture

Onion Architecture vs N-Tier

Let's Implement Onion Architecture in ASP.NET Core with CQRS

      Creating Domain Layer of Onion Architecture

What is Onion Architecture

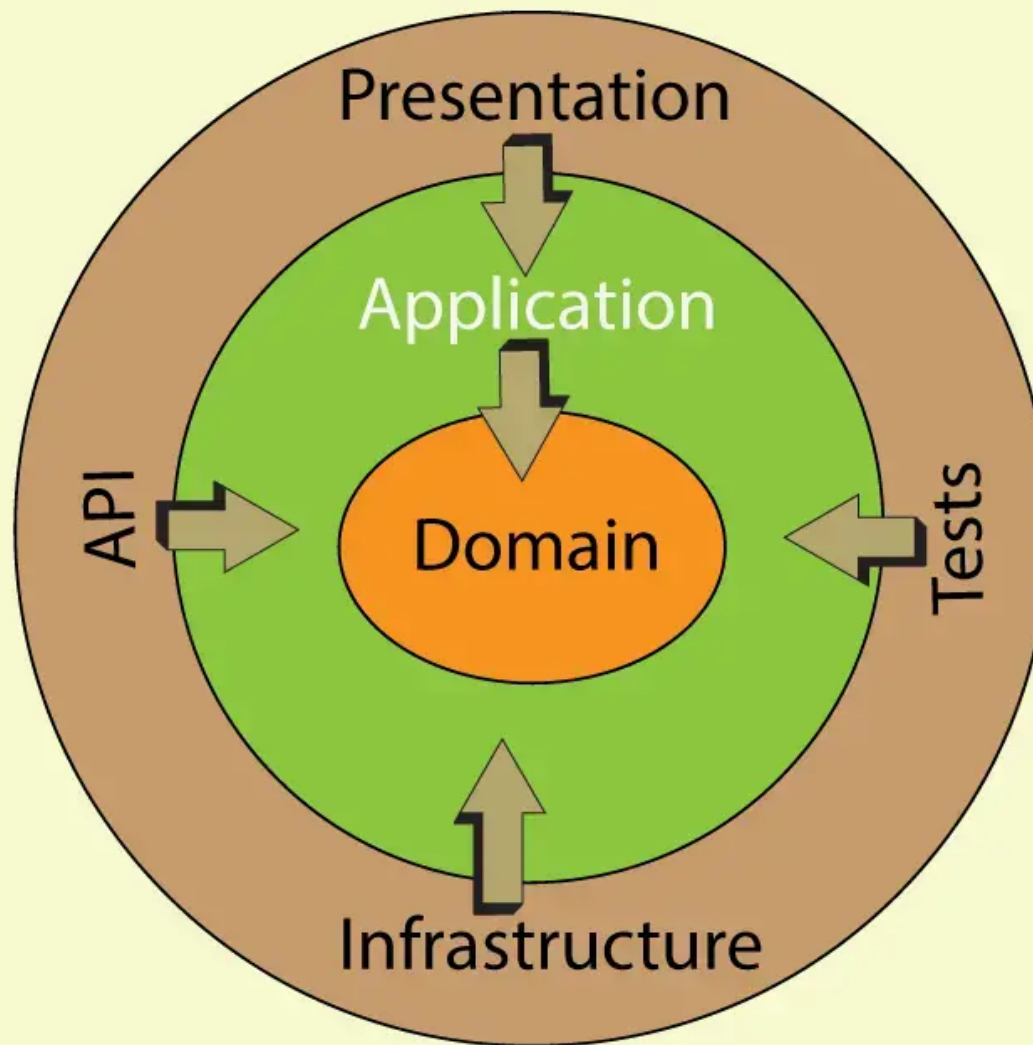In the Onion Architecture there are separatable concentric layers of codes such that the inner most layer is fully independent to other layers. Mostly you have 3 layers in this architecture and these are – Domain, Application and "Infrastructure + Presentation". Domain is the innermost layer while Infrastructure + Presentation is outermost layer.

See the below image illustration of the Onion Architecture.

Privacy

Privacy

Layers in Onion Architecture for an ASP.NET Core app

You must have question like – **What are the main layers of Onion**

In your ASP.NET Core app, Onion Architecture can be created with 3 layers. These are:

Domain

Application

Infrastructure + Presentation

Domain Layer in Onion Architecture

At the center, there is Domain Layer which is not dependent on any other layer. The Domain layer contains Entities which are common throughout the application and hence can be shared across other solutions to. Suppose you are building an app for a School so you will probably add entities like Student, Teacher, Fees classes in the Domain layer. Got the point?

Application Layer in Onion Architecture

The Application Layer in Onion Architecture contains Interfaces and these interfaces will be implemented by the outer layers of Onion Architecture like the Infrastructure + Presentation Layers.

For the School app, we may add an interface dealing with database operations for the entities called Student, Teacher and fees. This interface can be implemented in the outer Infrastructure Layer where the actual database operations are added. This is also creating the Dependency

Privacy

This also helps in building scalable application, how? Example – we can add new Interfaces for dealing with SMS/Email sending codes. Then we can implement these interfaces on the infrastructure layer. If in future the need for changes arise, we can easily change the implementations of the interfaces in the infrastructure Layer without affecting the Domain and Application layers.

What is Dependency Inversion Principle? Dependency Inversion Principle (DIP) states that the high-level modules should not depend on low-level modules. We create interfaces in the Application Layer and these interfaces get implemented in the external Infrastructure Layer.

⭐ The Domain and Application layers are together known as Core layers of Onion Architecture.

Infrastructure + Presentation Layer in Onion Architecture

The Infrastructure and Presentation Layers are outermost layers of Onion Architecture.

In the Infrastructure Layer we add infrastructure level codes like Entity Framework Core for DB operations, JWT Tokens for Authentication and other such works. All the heavy tasks of the app are performed in this layer.

The Presentation Layer will have a project which the user will use. This can be a project of Web API, Blazor, React or MVC type. Note that this project will also contain the User Interfaces.

As the Presentation layer is loosely coupled to other layers so we can easily change it. Like a React Presentation layer can be easily changed to Blazor WebAssembly one.

Onion Architecture vs Clean Architecture

Is Onion Architecture different than Clean Architecture? Clean architecture is just a term which describes creating architecture where the stable layers are not dependent on less stable layers. We already know that domain layer

This means Onion Architecture is a Clean Architecture. I have listed some major points which a clean architecture should have:

Independent of Frameworks – example we should easily replace Entity Framework Core with Dapper.

Testable – we should be able to test the business rules without the UI, Database, Web Server, or any other external dependency.

Database Independent – it should be easy to switch from one database to another like SQL Server to MongoDB.

Onion Architecture vs N-Tier

How Onion Architecture differs from N-Tier architecture? Onion Architecture is a clean architecture while N-Tier is not a clean architecture. N-Tier is neither a scalable architecture. If you see the below given diagram of N-Tier

contains the UI. The Business layer contains the business logic while the Data Access layer interacts with the database.

The data access layer usually contains ORM like Entity Framework core or Dapper. When creating n-tier architecture the layers depend on each other, and we end up building a highly coupled structure. This defeats the purpose of a Clean Architecture.

Privacy

Let's Implement Onion Architecture in ASP.NET Core with CQRS

Start by creating a Blank Solution in Visual Studio.

Privacy

Name it as OnionApp or something else.

In the blank solution add 3 folder which will serves as the layers for the Onion Architecture.

Core – it will hold projects for Domain and Application layers.

Infrastructure – it will hold projects for Infrastructure layer eg Authentication, EF core, Dapper etc.

Presentation – it will hold projects for Presentation layer like web api, blazor, react angular.

Privacy

The 2 most popular tutorials are:

Implementing ASP.NET Core CRUD Operation with CQRS and MediatR Patterns

ASP.NET Core Razor Pages : CRUD Operations with

Creating Domain Layer of Onion Architecture

Right click on the Core folder and select Add ➤ New Project and select

Class Library project.

Name the project as **Domain**.

Also select the framework as .NET Standard 2.1.

Note: We can have any type of library project on the Domain layer as this layer is independent of other layers. This is the reason I chose .NET

Now add a class called Student.cs to the Domain project. You can do this by right clicking on the project name in solution explorer and select Add ➤ New Item. On the item's list, choose "Class" and Name it Student.cs.

The Student.cs class is shown below. It is a fairly simple class containing just 3 properties to describe a student.

```
 1   namespace Domain
 2   {
 3       public class Student
 4       {
 5           public int Id { get; set; }
 6           public string Name { get; set; }
 7           public string Standard { get; set; }
 8           public int Rank { get; set; }
 9       }
10   }
```

★ Start learn Microservices in ASP.NET Core:

1. First ASP.NET Core Microservice with Web API CRUD Operations on a MongoDB database [Clean Architecture]

2. Synchronous Communication between Microservices built in ASP.NET Core

3. Microservices API Gateway to unify Multiple Microservices

Privacy

4. Microservices Asynchronous Communication with RabbitMQ and MassTransit

5. ASP.NET Core Microservices Code Refactoring into Reusable NuGet Package

Creating Application Layer of Onion Architecture

We will be creating another Class Library Project inside the same "Core" folder. This project will be for the **Application Layer**. Go, right click on the

project. Name this project as Application, for this project select framework as .NET 8.0.

As states earlier, the Application Layer contain Interfaces which are

1. Interface for Entity Framework Core

2. CQRS

Interface for Entity Framework Core

We earlier created an entity called Student in the domain layer, this entity should be mapped as a class for Entity Framework Core. This we all know is done by Data Context class. But here we won't be doing this, instead we will add an Interface that will contain Entity Framework Core logic. This interface will be implemented on the Infrastructure layer and so EF codes does not fall under the Application Layer, but goes to the Infrastructure layer which is outside the "Core".

In future if we decide to have a different implementation of this interface then we just have to change the implementation part which lies on the infrastructure layer. We don't need to touch the application layer at all. This way we can build scalable applications.

First add reference of the Domain Project in this project. Right click on Application project and select Add ➤ Project Reference.

A new window will open. On it's left side you will see <u>Projects</u> option, select it, then on the middle section you will see the <u>Domin</u> project. Select the checkbox for the domain project and click the OK button. This will add the Domain project's reference to the Application project.

Privacy

Next, from NuGet Package Manger, install the
Microsoft.EntityFrameworkCore package to the Application project.

Privacy

You can also add the Entity Framework Core package by running the command `Install-Package Microsoft.EntityFrameworkCore` on the Package Manager Console window of Visual Studio. Remember to select the Application project from the "Default project" dropdown. Check the below image.

Now add a new interface on the Application project and call it

```
 2   using Microsoft.EntityFrameworkCore;
 3
 4   namespace Application
 5   {
 6       public interface IAppDbContext
 7       {
 8           DbSet<Student> Students { get; set; }
 9           Task<int> SaveChangesAsync();
10       }
11   }
```

Adding CQRS

Now we will add Command Query Responsibility Segregation (CQRS)
pattern. This pattern specifies that different data models should be used to
for updating / reading the database. I have written a descriptive CQRS
article – Implementing ASP.NET Core CRUD Operation with CQRS and
MediatR Patterns, you will find this article very helpful.

Start by adding the MediatR NuGet package to the Application project.

Then create a new folder and name it CQRS, inside this folder create 2 new folder that will hold Queries and Commands classes. Name these folder as "Queries" and "Commands". Check the below image.

Privacy

Next, inside the Commands folder add CQRS classes for commands, these
are:

CreateStudentCommand.cs

```
1   using Domain;
2   using MediatR;
3
4   namespace Application.CQRS.Commands
5   {
6       public class CreateStudentCommand : IRequest<int>
7       {
8           public string Name { get; set; }
```

```
13          private readonly IAppDbContext context;
14          public CreateStudentCommandHandler(IAppDbCon
15          {
16              this.context = context;
17          }
18          public async Task<int> Handle(CreateStudentC
19          {
20              var student = new Student();
21              student.Name = command.Name;
22              student.Standard = command.Standard;
23              student.Rank = command.Rank;
24
25              context.Students.Add(student);
26              await context.SaveChangesAsync();
27              return student.Id;
28          }
29      }
30    }
31  }
```

UpdateStudentCommand.cs

```
 1  using MediatR;
 2
 3  namespace Application.CQRS.Commands
 4  {
 5      public class UpdateStudentCommand : IRequest<int>
 6      {
 7          public int Id { get; set; }
 8          public string Name { get; set; }
 9          public string Standard { get; set; }
10          public int Rank { get; set; }
11          public class UpdateStudentCommandHandler : IRequ
12          {
13              private readonly IAppDbContext context;
14              public UpdateStudentCommandHandler(IAppDbCon
15              {
16                  this.context = context;
17              }
18              public async Task<int> Handle(UpdateStudentC
19              {
20                  var student = context.Students.Where(a =
21
22                  if (student == null)
23                      return default;
24
25                  student.Name = command.Name;
26                  student.Standard = command.Standard;
27                  student.Rank = command.Rank;
28                  await context.SaveChangesAsync();
29                  return student.Id;
30              }
31          }
32      }
```

DeleteStudentByIdCommand.cs

```csharp
1  using MediatR;
2  using Microsoft.EntityFrameworkCore;
3
4  namespace Application.CQRS.Commands
5  {
6      public class DeleteStudentByIdCommand : IRequest<int
7      {
8          public int Id { get; set; }
9          public class DeleteStudentByIdCommandHandler : I
10         {
11             private readonly IAppDbContext context;
12             public DeleteStudentByIdCommandHandler(IAppD
```

```
17                {
18                        var student = await context.Students.Whe
19                        if (student == null) return default;
20                        context.Students.Remove(student);
21                        await context.SaveChangesAsync();
22                        return student.Id;
23                }
24            }
25        }
26    }
```

Now, inside the Queries folder add CQRS classes for queries, these are:

GetAllStudentQuery.cs

```
1    using Domain;
2    using MediatR;
3    using Microsoft.EntityFrameworkCore;
4
5    namespace Application.CQRS.Queries
6    {
7        public class GetAllStudentQuery : IRequest<IEnumerab
8        {
9
10            public class GetAllStudentQueryHandler : IReques
11            {
12                private readonly IAppDbContext context;
13                public GetAllStudentQueryHandler(IAppDbConte
14                {
15                    this.context = context;
16                }
17                public async Task<IEnumerable<Student>> Hand
18                {
19                    var studentList = await context.Students
```

```
24              return studentList.AsReadOnly();
25          }
26      }
27  }
28 }
```

GetStudentByIdQuery.cs

. . .

```
1  using Domain;
2  using MediatR;
3  using Microsoft.EntityFrameworkCore;
4
5  namespace Application.CQRS.Queries
6  {
```

Privacy

```
11      {
12          private readonly IAppDbContext context;
13          public GetStudentByIdQueryHandler(IAppDbCont
14          {
15              this.context = context;
16          }
17          public async Task<Student> Handle(GetStudent
18          {
19              var student = await context.Students.Whe
20              if (student == null) return null;
21              return student;
22          }
23      }
24  }
25 }
```

These classes will implement the CQRS pattern and will perform database
CRUD operations for the Student entity. As you may have already noticed,
they are using Entity Framework Core for performing database operations.
You can see these classes in the below image.

Privacy

We now will create an extension method to register MediatR library which can be done by adding a class called DependencyInjection.cs. The code of this class is given below:

```
using Microsoft.Extensions.DependencyInjection;
```

```
namespace Application
{
    public static class DependencyInjection
    {
        public static void AddApplication(this IServiceCollection
services)
        {
            services.AddMediatR(a =>
a.RegisterServicesFromAssembly(Assembly.GetExecutingAssembly()));
        }
    }
}
```

We will be calling the method AddApplication from the Program class when we will create the Presentation layer.

We now will move to Infrastructure layer.

Creating Infrastructure Layer of Onion Architecture

Now create a Class Library project inside the Infrastructure folder. Name this project as "Persistence".

Next, perform a couple of things which are:

Privacy

1. Select framework to .NET 8.0 which is the latest version right now.

2. Add project reference for the Application project we build earlier.

3. Install the following NuGet packages.

```
Install-Package Microsoft.EntityFrameworkCore
Install-Package Microsoft.EntityFrameworkCore.SqlServer
Install-Package Microsoft.EntityFrameworkCore.Design
```

In this project we will setup Entity Framework Core which will access the CRUD operations performed by CQRS. Recall, we already created CRUD operations on the Application project.

Note that we will be using this layer to perform Migrations and Generate our database. We will see this in just a moment.

Remember we created an IAppDBContext Interface in the Application Layer? Now it's time to implement it. So add a new class called AppDbContext.cs to the Persistence project and implement "IAppDBContext" interface as shown below.

```
1   using Application;
2   using Domain;
3   using Microsoft.EntityFrameworkCore;
```

```
 7      public class AppDbContext : DbContext, IAppDbContext
 8      {
 9          public AppDbContext(DbContextOptions<AppDbContex
10              : base(options)
11          {
12          }
13
14          public DbSet<Student> Students { get; set; }
15
16          public async Task<int> SaveChangesAsync()
17          {
18              return await base.SaveChangesAsync();
19          }
20      }
21  }
```

With our infrastructure layer complete, we are ready to generate the
database. But there is a catch, EF Core migrations can only be run from a
project which has Program.cs class therefore we can only run the
migrations from the Web API project of the Presentation layer.

Now coming to a very important part which is how the dependency of
IAppDbContext will be resolved and how EF Core will pic the database
during migration? The best way is to create an extension method in our
"Persistence" project and we will call this method from the Program class of
our Web API project. So, create a new class called DependencyInjection.cs
which contains this extension method. The code of this class is given below.

```
1  using Application;
2  using Microsoft.EntityFrameworkCore;
3  using Microsoft.Extensions.Configuration;
```

```
 7    {
 8        public static class DependencyInjection
 9        {
10            public static void AddPersistence(this IServiceC
11            {
12                services.AddDbContext<AppDbContext>(options
13                    options.UseSqlServer(
14                        configuration.GetConnectionString("D
15                        b => b.MigrationsAssembly("WebApi"))
16                services.AddScoped<IAppDbContext>(provider =
17            }
18        }
19    }
```

Clearly see that we are telling EF Core to get the connection string from DefaultConnection node of appsettings.json file.

```
configuration.GetConnectionString("DefaultConnection")
```

And the appsettings.json file should be searched in the assembly called

```
b => b.MigrationsAssembly("WebApi")
```

Now let us move towards the final layer which is the **Presentation layer**.

Creating Presentation Layer of Onion Architecture

Start by adding an ASP.NET Core Web API project in the Presentation folder.

.

Name it <u>WebApi</u> (do you know why? See the extension method we created earlier). On the next screen, select .NET 8.0 for framework and check the checkbox option <u>Enable OpenAPI support</u>.

Privacy

Since this project will be calling the component of the "Application and Persistence" projects so make sure you add the project references to these 2 projects in your WebApi project. You need to right click the <u>Dependencies</u> folder, then click the Add Project Reference option. In the dialog that opens, select <u>Application and Persistence</u> projects and click the OK button.

Now open the appsetting.json file and add the connection string for the database.

```
{
  "ConnectionStrings": {
    "DefaultConnection": "Server=
(localdb)\\mssqllocaldb;Database=OnionAppDb;Trusted_Connection=Tr
ue;MultipleActiveResultSets=true"
  }
```

Here, I have name my database as OnionAppDb.

Next open the Program.cs class and call the 2 extension methods which we created earlier.

These extension methods registers MediatR and adds dependency injection middleware. This is shown by the highlighted code below.

```
1   using Persistence;
2   using Application;
3
4   var builder = WebApplication.CreateBuilder(args);
5
6   // Add services to the container.
7   builder.Services.AddApplication();
8   builder.Services.AddPersistence(builder.Configuration);
9
10  builder.Services.AddControllers();
11
12  // Learn more about configuring Swagger/OpenAPI at https
13  builder.Services.AddEndpointsApiExplorer();
14  builder.Services.AddSwaggerGen();
15
16  var app = builder.Build();
17
18  // Configure the HTTP request pipeline.
19  if (app.Environment.IsDevelopment())
20  {
21      app.UseSwagger();
22      app.UseSwaggerUI();
23  }
24
25  app.UseHttpsRedirection();
```

```
30
31    app.Run();
```

Entity Framework Core Migrations and generate the Database

Before performing migrations, select Build ➤ Build Solution in Visual Studio, this is necessary to create the project references properly. Now open Package Manager Console window and navigate to the WebApi folder. This is done because migration commands must be run from the folder containing the Program.cs class.

Use the "cd" command to navigate to this folder:

```
cd WebApi
```

Now run the below 2 commands which install `dotnet-ef` commannd and install the package Microsoft.EntityFrameworkCore.Tools to the "WebApi" project.

```
dotnet tool install --global dotnet-ef
Install-Package Microsoft.EntityFrameworkCore.Tools
```

Finally run the 2 EF core migration commands one by one.

```
dotnet ef migrations add Migration1
dotnet ef database update
```

⭐ If you get error during migrations saying `System.Globalization.CultureNotFoundException: Only the invariant culture is supported in globalization-invariant mode`, then you have to turn the InvariantGlobalization property on WebApi project to false i.e. `<InvariantGlobalization>false</InvariantGlobalization>`.

The commands will create a "Migrations" folder in the WebApi project and database in the MSSQLLocalDB. Navigate to View ➤ SQL Server Object Explorer in Visual Studio and you can see this newly created database with just one table called "Students".

Privacy

Next we will create Web Api controller for making HTTP Request to perform CRUD operations.

Create Web API Controllers

to the Controllers folder and name is HomeController.cs. Now add the following code to it.

```
1   using Application.CQRS.Commands;
2   using Application.CQRS.Queries;
3   using MediatR;
4   using Microsoft.AspNetCore.Mvc;
5
6   namespace WebApi.Controllers
7   {
8       [ApiController]
9       [Route("api/[controller]")]
10      public class HomeController : ControllerBase
11      {
12          private IMediator mediator;
13          public HomeController(IMediator mediator)
14          {
15              this.mediator = mediator;
16          }
17
18          [HttpPost]
19          public async Task<IActionResult> Create(CreateSt
20          {
21              return Ok(await mediator.Send(command));
22          }
23
24          [HttpGet]
25          public async Task<IActionResult> GetAll()
26          {
27              return Ok(await mediator.Send(new GetAllStud
28          }
29
30          [HttpGet("{id}")]
31          public async Task<IActionResult> GetById(int id)
32          {
```

```
36        [HttpDelete("{id}")]
37        public async Task<IActionResult> Delete(int id)
38        {
39            return Ok(await mediator.Send(new DeleteStud
40        }
41
42        [HttpPut("[action]")]
43        public async Task<IActionResult> Update(int id,
44        {
45            if (id != command.Id)
46            {
47                return BadRequest();
48            }
49            return Ok(await mediator.Send(command));
50        }
51    }
52 }
```

This controller has methods which wire up to the CQRS in the Application Layer. Through these methods we will be performing CRUD operations. So first you should make the WebApi project as the single startup project. To do this right click the Solution in the Solution Explorer and select properties. Then make WebAPI as single startup project.

Testing with Swagger

Swagger comes pre-configured in the Web API template. So run the app in Visual Studio and you will see the **Swagger** screen. It will show all the Web API methods we created.

Privacy

Click the POST button to test the creation of a new student.

In the Request Body add the name, standard and rank of the student as shown below and then clicking the execute button.

```
{
  "name": "Jack Sparrow",
  "standard": "10",
```

You will see success response with the created student id in the response body. This will be 1 since it is the first record which we created.

Privacy

Now click the GET button on Swagger page to test the Read functionality.
You will see all the students records in json.

```
[
  {
    "id": 1,
    "name": "Jack Sparrow",
    "standard": "10",
    "rank": 2
```

Privacy

Next click the PUT button on the swagger page. We will now update the first student record. So on the "Id" field enter 1 which is the id of the student, and on the request body change the name, standard and rank values of the student, see below json.

```
{
  "id": 1,
```

```
        "rank": 1

    }
```

Click the execute button and the record will be updated. Confirm this by the second GET button on swagger page which shows the url /api/Home/{id}. Enter 1 on the id field and click the execute button.

You will see the student's json is updated with the new values.

Privacy

Finally check the DELETE functionality which I leave it to yourself.

I hope this **ASP.NET Core Onion Architecture** guide is easy for you to understand. I can say that it will make you fully ready to develop enterprise type apps on this architecture. Also make sure you download the codes from my GitHub repository whose link is shared in the beginning of this tutorial. Enjoy and happy coding!

SHARE THIS ARTICLE

ABOUT THE AUTHOR

I am Yogi S. I write DOT NET artciles on my sites hosting.work and yogihosting.com. You can connect with me on Twitter. I hope my articles are helping you in some way or the other, if you like my articles consider buying me a coffee - Buy Me A Coffee

Leave a Reply

Your email address will not be published. Required fields are marked *

Comment *

Privacy
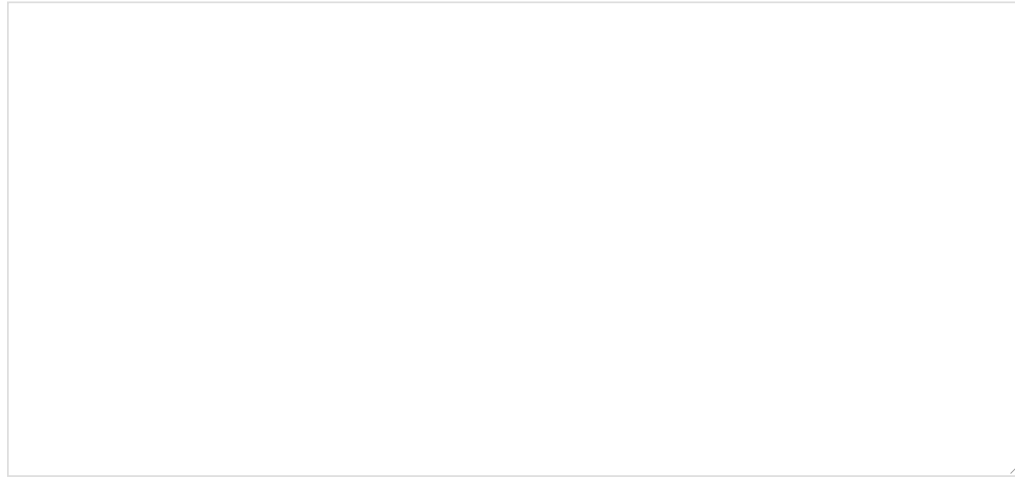
Name *

Email *

Website

Post Comment

Privacy

Related Posts based on your interest

[Necessary .NET Security features for securing your web applications](#)

[Automated UI Testing with Selenium in ASP.NET Core](#)

[How to perform Integration Testing in ASP.NET Core](#)

Hello Everyone,

Welcome to Hosting.Work - A Programming Tutorial Website. It covers ASP.NET Core topics. I hope you enjoy reading it.

SUBSCRIBE TO NEWSLETTER

Enter your email address to subscribe to this blog and receive notifications of new posts by email

Email Address *

Subscribe

Subscribe to our Newsletter and recevie "1 email per week" for the article published on the site. No spamming.

Privacy

Copyright ©2024

Privacy