ePizza Hub: Online Food Delivery Website

Shailendra Chauhan

Microsoft MVP, Technical Consultant & Corporate Trainer

DotNetTricks

Agenda

- Project Architecture (Clean Architecture)
- Database Schema
- Creating Project Layers
- Creating Database Using EF Code First Migrations
- Implementing Repository Design Patterns
- Login/SignUp
- Security - Authentication/Authorization

DotNetTricks

Agenda

- Building UI Using Bootstrap
- Mobile First Design
- Modules - User and Admin
- Product Images Upload
- Shopping Cart
- Payment Gateway
- Deployment

Pre-requisites

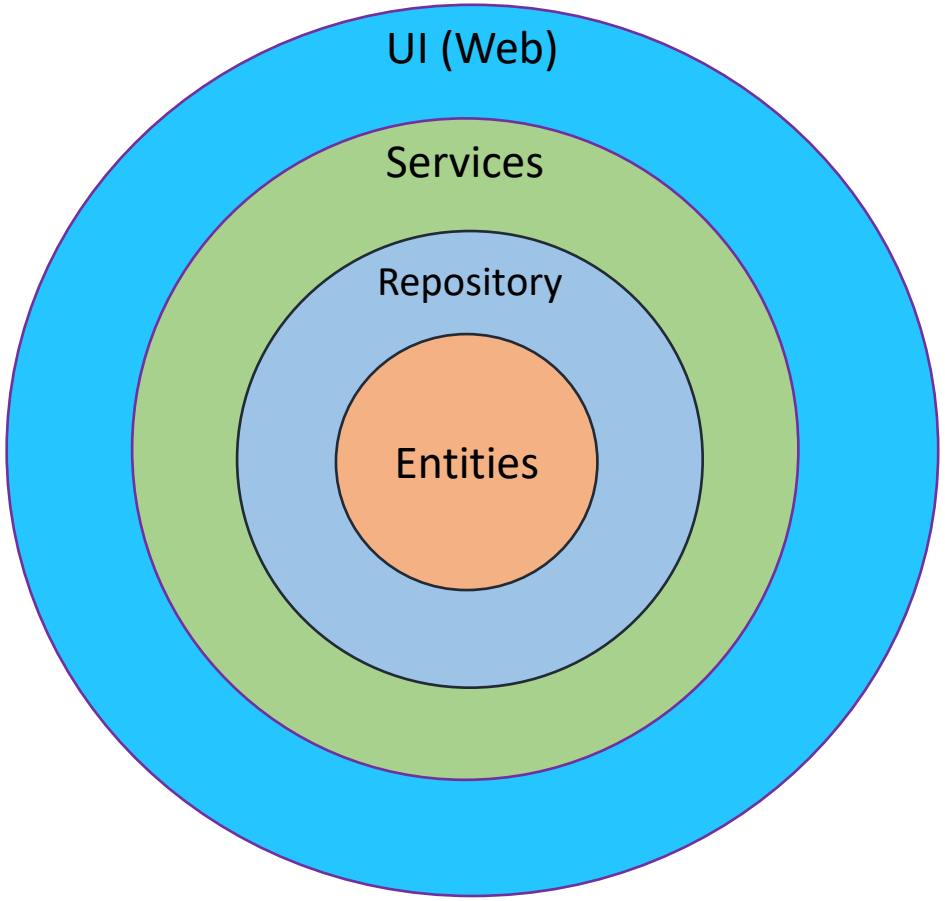
- Familiar with C#, .NET Core, LINQ, Entity Framework Core, ASP.NET Core, Bootstrap, jQuery and SQL Server.
- Tools/IDE : Visual Studio 2019 or Higher Version, SQL Server 2019 or Higher Version.

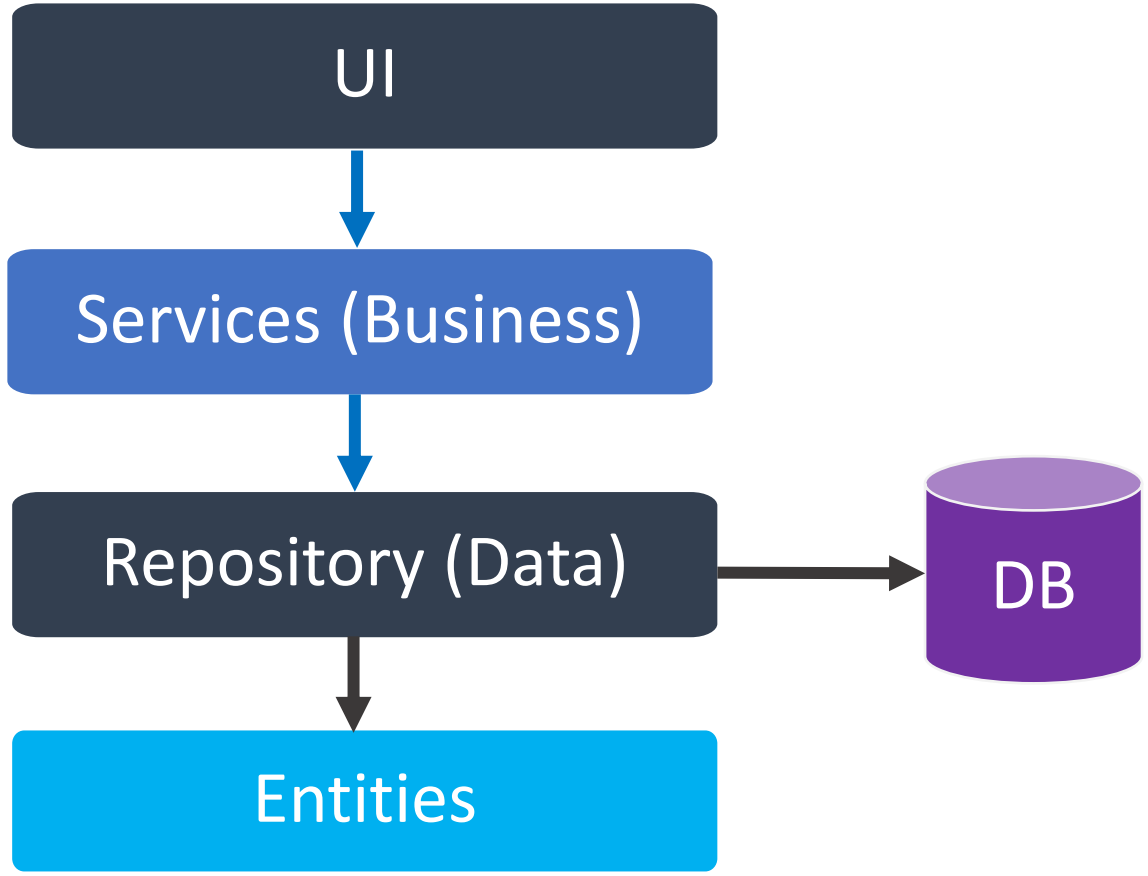
Community Edition

SQL Server Express

DotNetTricks

Project Architecture





DotNetTricks

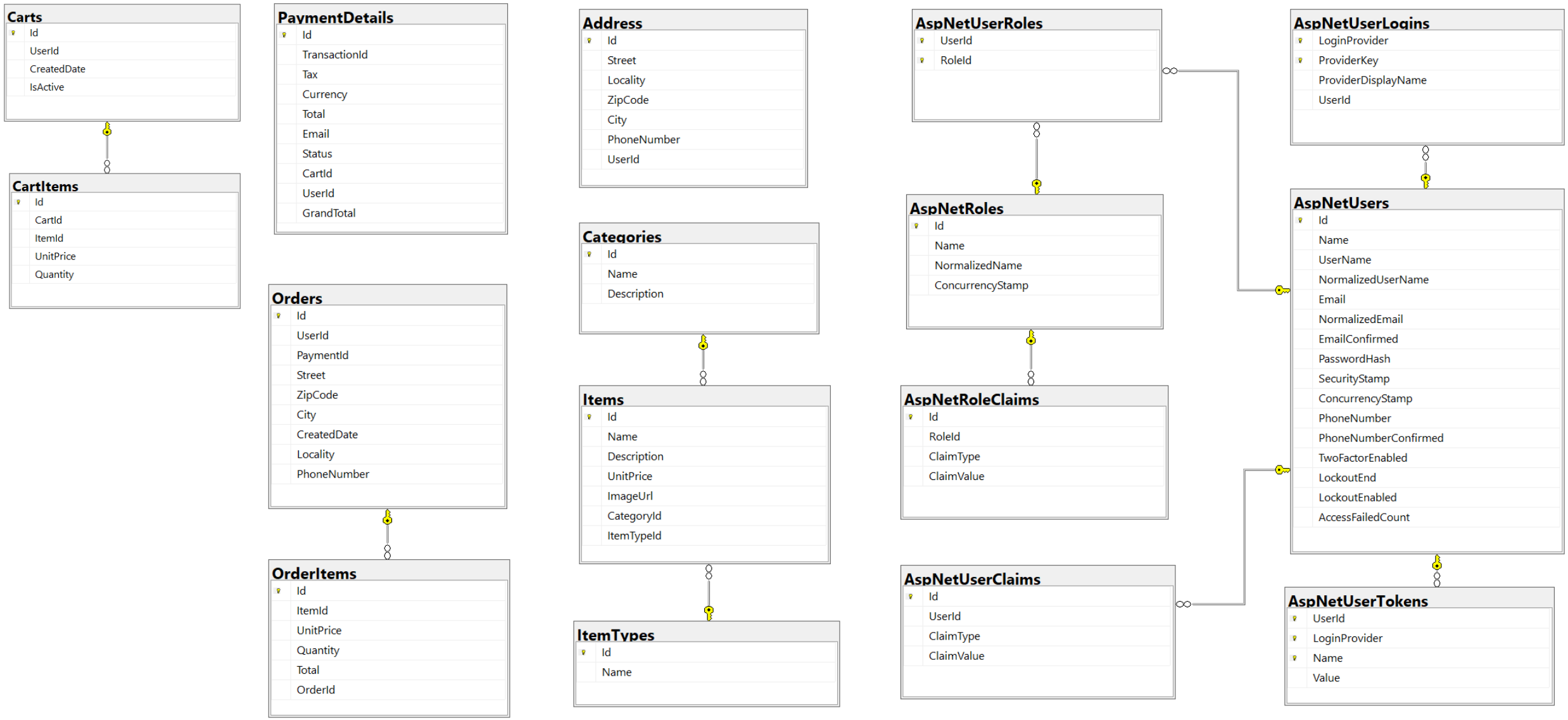
Projects To DO

- EdTech Platform - DotNetTricks
- Ecommerce Website - Flipkart, Amazon
- Food Delivery - McDonald's, PizzaHut, Dominoz
- Online Doctor Consultation - Practo
- Job Portal - Naukri.com
- Online Service - Urban Clap, Uber, OLA
- Video Streaming - Netflix, Amazon prime
- Online Booking - Flight, Hotel

DotNetTricks

Demo :

Creating Project Layers



Database Schema Diagram

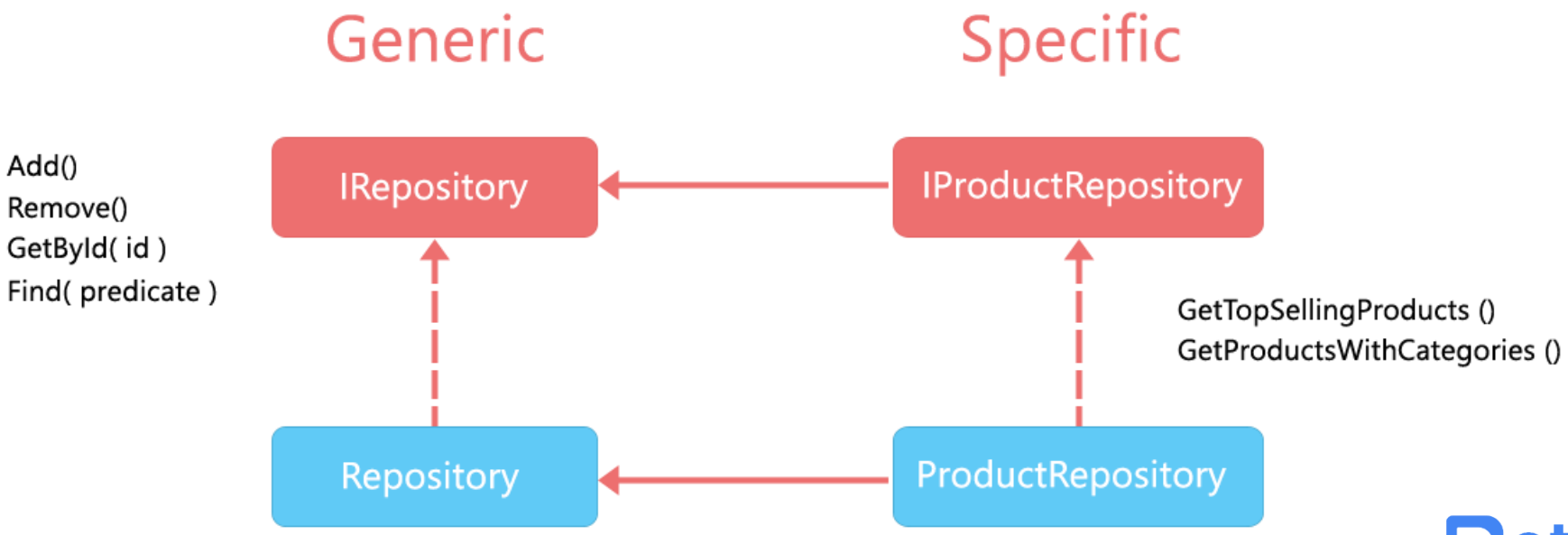
DotNetTricks

Demo :

Creating Entities

Repository Design Pattern

- Mediates between the domain and data mapping layers, acting like an in-memory collection of domain objects.
- Acts as an abstraction between the application and data source



DotNetTricks

Advantages of Repository Design Pattern

- Minimize duplicate query logic
- Supports SOC since application need not to know about data sources and it’s access logic
- Decouples the application from persistence framework like EF, Dapper, Nhibernate etc.
- Allows unit testing since repositories are bound to interfaces which can be injected into classes at run time

Demo :

Creating Repositories

Dependency & Dependency Management

- Suppose Class A needs Class B to do its job, Hence, Class B is a dependency of Class A.

- To make sure that class A will have a class B object, Class A can use one the following option:

1. Create class B object
2. Fetch class B object
3. Receive class B object

Class A is in control of his dependency

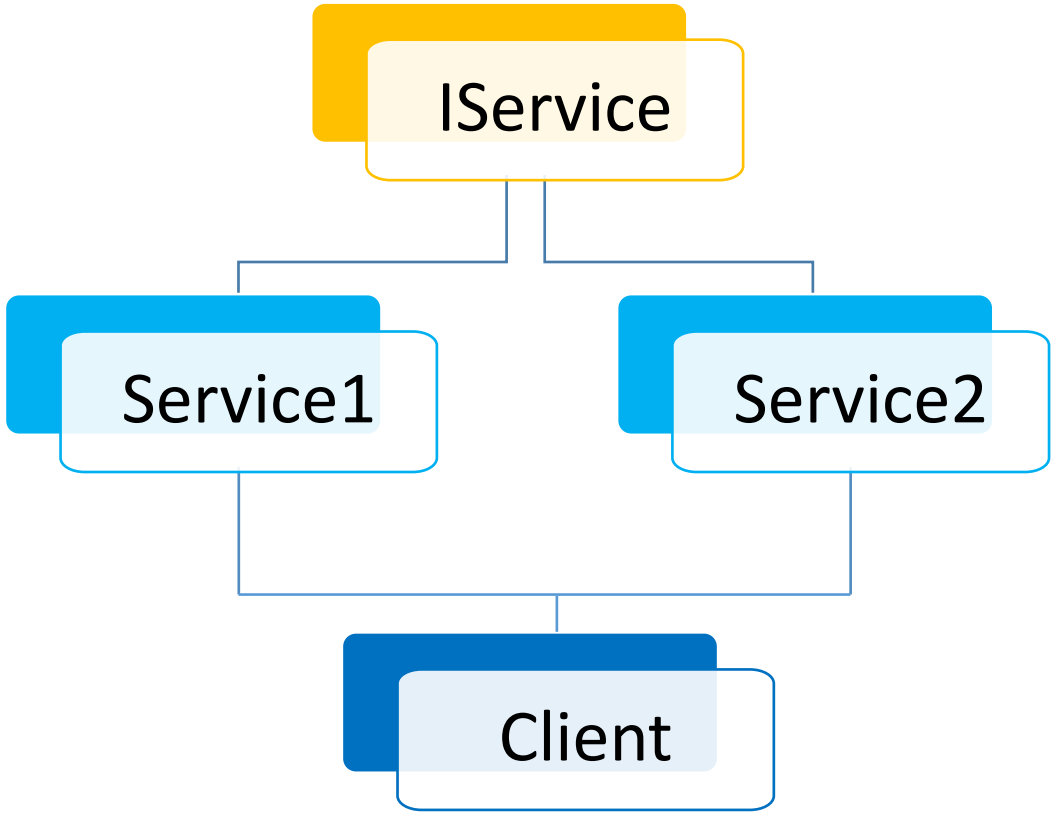
Class A is not in control of his dependency, some one else (IOC)

Dependency Injection

- Software design pattern which implement IOC. IoC is a programming style where the flow of a program has been inverted i.e. changed from the normal way.
- Allow to develop loosely coupled software components.
- In this components consume functionality defined by interface without having any knowledge of classes implementation
- Help to manage future changes and other complexity in a software in a better way.
- The purpose of DI is to make code maintainable.

Dependency Injection Implementation

- Constructor Injection
- Setter/Property Injection
- Method Injection



DotNetTricks

Dependency Injection - Constructor Injection

```
public interface IService    {
   void Serve();
}
public class Service1 : IService    {
   public void Serve()   {  Console.WriteLine("Service1 Called"); }
}
public class Service2 : IService    {
   public void Serve()   {  Console.WriteLine("Service2 Called"); }
}
public class Client    {
   private IService _service;
   public Client(IService service)    {
      this._service = service;
   }
  public ServeMethod()    { this._service.Serve(); }
}
```

```
class Program
{
   static void Main(string[] args)
   {
       //creating object
       Service1 s1 = new Service1();
      //passing dependency
      Client c1 = new Client(s1);
     //TO DO:

      Service2 s2 = new Service2();
      //passing dependency
       c1 = new Client(s2);
     //TO DO:
   }
}
```

DotNetTricks

Dependency Injection - Property Injection

```
public interface IService    {
   void Serve();
}
public class Service1 : IService    {
   public void Serve()   {  Console.WriteLine("Service1 Called"); }
}
public class Service2 : IService    {
   public void Serve()   {  Console.WriteLine("Service2 Called"); }
}
public class Client    {
   private IService _service;
    public IService Service   {
      set {  this._service = value; }
   }
  public ServeMethod()    { this._service.Serve(); }
}
```

```
class Program
{
   static void Main(string[] args)
   {
       //creating object
       Service1 s1 = new Service1();

      Client client = new Client();
       client.Service = s1; //passing dependency
      //TO DO:

      Service2 s2 = new Service2();
     client.Service = s2; //passing dependency
    //TO DO:
   }
}
```

DotNetTricks

Dependency Injection - Method Injection

```
public interface IService    {
   void Serve();
}
public class Service1 : IService    {
   public void Serve()   {  Console.WriteLine("Service1 Called"); }
}
public class Service2 : IService    {
   public void Serve()   {  Console.WriteLine("Service2 Called"); }
}
public class Client    {
    private IService _service;
    public void Start(IService service)   {
      service.Serve();
   }
}
```

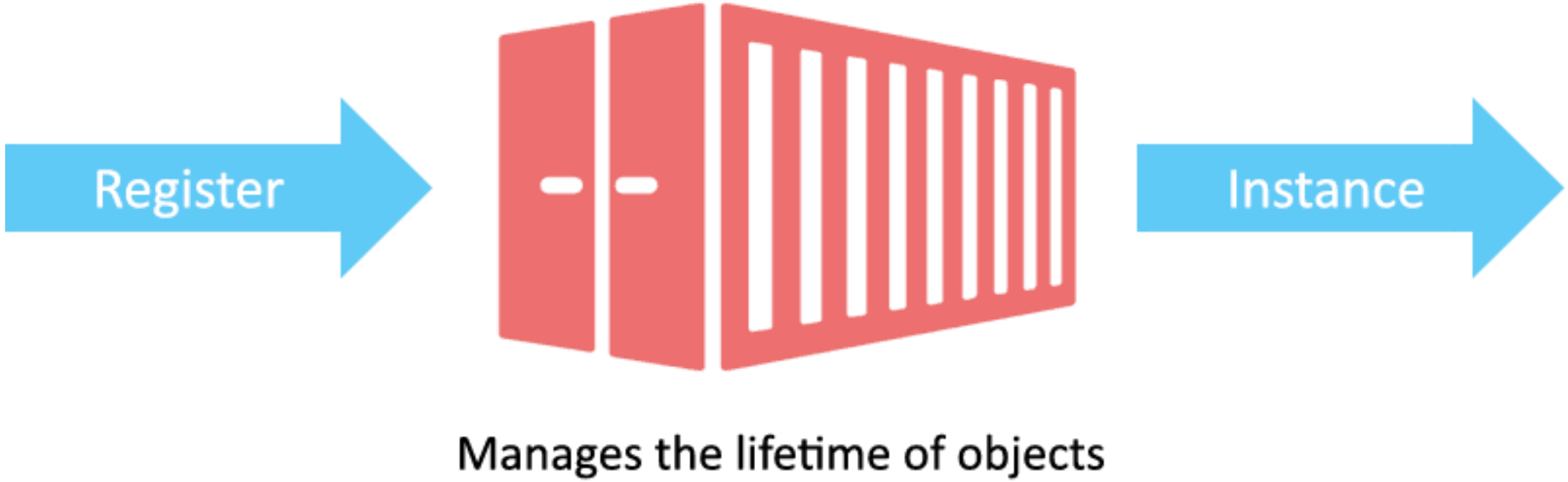
```
class Program
{
   static void Main(string[] args)
   {
       //creating object
       Service1 s1 = new Service1();

      Client client = new Client();
       client.Start(s1); //passing dependency
      //TO DO:

      Service2 s2 = new Service2();
      client.Start(s2); //passing dependency
   }
}
```

DotNetTricks

IOC Container



DotNetTricks

ASP.NET Core Built-In Container Service Methods

- **Singleton** - An object of a service is created only once and supplied to all the requests to that service. So, basically all requests get the same object to work with all calls.
- **Scoped** - An object of a service is created for each request. So, Within the scope, it reuses the existing service object.
- **Transient** - An object of a service is created every time when it is requested. Works best for lightweight and stateless services.

Singleton vs. Scoped vs. Transient

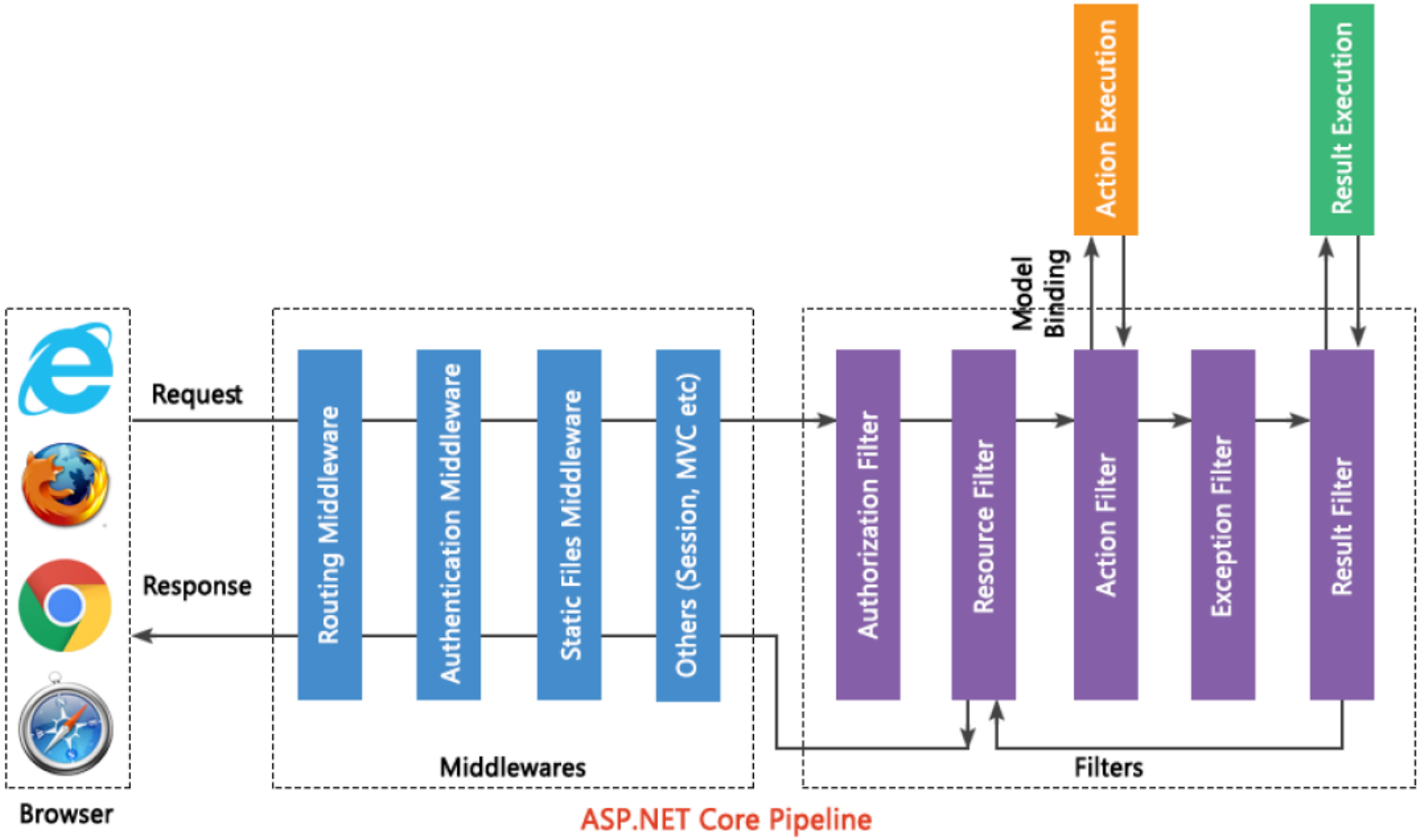
Parameter	Singleton	Scoped	Transient
Instance	One for all Requests	One for each request	Every time new
Disposed	App Shutdown	Request End	Request End
Behavior	Stateful & Singleton for all requests	Stateful for a request	Stateless for a request

ASP.NET Core Identity

- An API to support Login, Signup, Sign out Functionalities
- Manages users, passwords, profile data, roles, claims, tokens, email confirmation, and more.
- Supports external login providers such as Facebook, Google, Microsoft Account, Twitter and more.

DotNetTricks

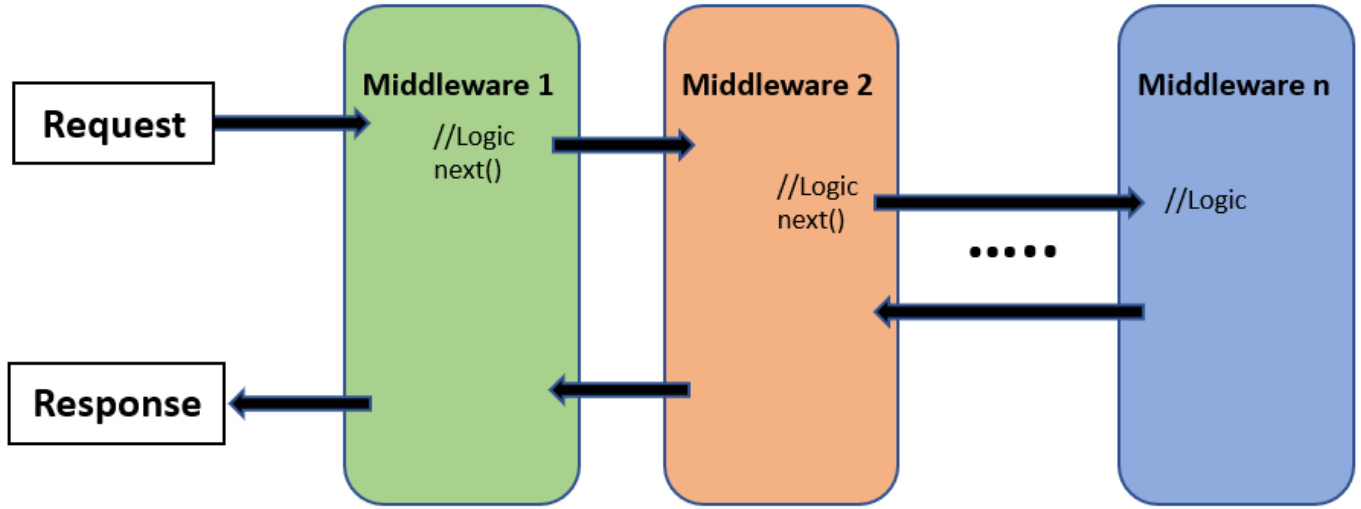
ASP.NET Core Request Pipeline



DotNetTricks

Middleware

- A middleware is a function which is executed before the request and response are being processed.
- A middleware can be used for user authentication, logging etc.



DotNetTricks

Built-in Middleware

- ASP.NET Core comes with built-in middleware e.g. static files, routing, session, authorization, exception handling etc.

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
   app.UseStaticFiles();
   app.UseExceptionHandler("/Home/Error");
   app.UseRouting();
   app.UseAuthorization();
   app.UseEndpoints(endpoints =>
   {
        endpoints.MapControllerRoute(
           name: "default",
           pattern: "{controller=Home}/{action=Index}/{id?}");
   });
}
```

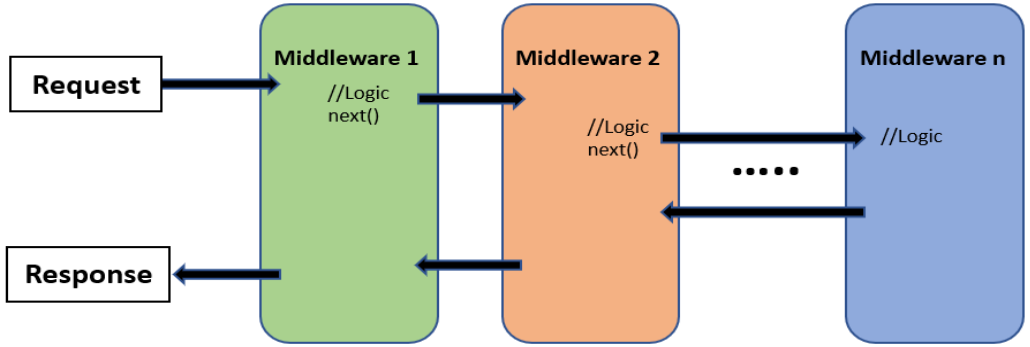
Order of Execution

DotNetTricks

use vs. run methods

- **Use** - Performs action before and after next delegate call.
- **Run** - Terminates the pipeline. No other middleware method will run after this. Should be placed at the end of any pipeline.

```
public void Configure(IApplicationBuilder app)
{
  app.Use(async (context, next) =>
  {
    // Code Logic: Authentication
    await context.Response.WriteAsync("1st delegate before");
    await next.Invoke(); //Call next middleware
    // Code Logic: Change response
    await context.Response.WriteAsync("1st delegate after");
  });
  app.Run(async context =>
  {
     //End pipeline
     await context.Response.WriteAsync("2nd delegate");
  });
}
```



DotNetTricks

Filters

- A Fitter is an attribute class which methods are executed before or after an action is executed.
- Filters are used to perform the following common functionalities:
  - Authentication
  - Authorization (User-based or Role-based)
  - Error handling or logging
  - User Activity Logging
  - Data Caching
  - Data Compression

Different types of Filters

- Authorization filters
- Resource Filters
- Action filters
- Exception filters
- Result filters

Order of Filter Execution

Configuring Filters

- A filter into your code can be configured at following three levels:
  - Global level
  - Controller level
  - Action level

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllersWithViews(options =>
    {
        options.Filters.Add(typeof(CustomFilter));
    });
}
```

```
[Authorize(Roles = "Admin")]
public class AdminController: Controller
{
    //TODO:
}
```

```
public class UserController : Controller
{
    [Authorize(Users = "User1,User2")]
    public ActionResult Login(string provider)
    {
        // TODO:
        return View();
    }
}
```

Demo :

Testing User Signup Workflow

Learn. Build. Empower.

DotNetTricks