**بنام خدا**

پیاده سازی یک زیرساخت دات نتی برای میکروسرویس

امیر حصاری – 9812762383

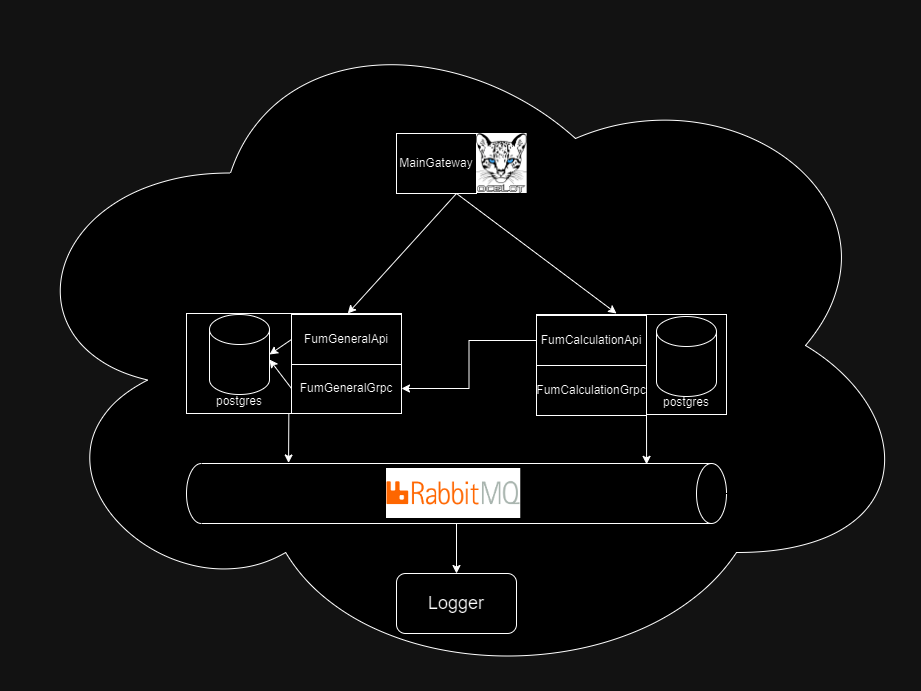
**1 – مقدمه:**

معماری میکروسرویس، یک الگوی طراحی نرم‌افزار است که بر پایه تقسیم سیستم به بخش‌های کوچک‌تر و مستقل تحت عنوان "میکروسرویس‌ها" انجام می‌شود. این بخش‌های مستقل به صورت خودکار و مستقل از یکدیگر عمل می‌کنند و از طریق رابط‌های استاندارد با یکدیگر ارتباط برقرار می‌کنند. این الگو اجازه می‌دهد

که هر میکروسرویس به صورت مستقل و مجزا از دیگری توسعه، مدیریت و ارتقاء شود، که این امر به بهبود انعطاف‌پذیری، قابلیت توسعه و دیگر ویژگی‌های مهم سیستم کمک می‌کند. در این پروژه، ما این الگوی معماری را برای پیاده‌سازی یک سیستم پویا و مقیاس‌پذیر در نظر گرفته‌ایم که از قابلیت‌ها و امکانات برتر دات‌نت بهره‌مند باشد.

معماری میکروسرویس، با تمام مزایایی که ارائه می‌دهد، قابلیت ایجاد یک بهم ریختگی نیز دارد. این بهم ریختگی به وجود می‌آید زیرا هر میکروسرویس به صورت مستقل پیاده‌سازی می‌شود و این امکان را فراهم می‌کند که هر برنامه‌نویس به تفاوت‌ها در پیاده‌سازی‌ها و کدهای خود بپردازد. در این پروژه، هدف ما این است که با ایجاد یک زیرساخت متمرکز و اعمال قواعد یکسان، به تنظیم و نظم بخشی در توسعه و نگهداشت سیستم بپردازیم. این تلاش با هدف ایجاد یک محیط یکپارچه و استاندارد برای توسعه و نگهداشت، در دنیای میکروسرویس‌ها، به ما این امکان را می‌دهد که بهترین عملکرد و کارایی را از این معماری بهره‌مند سازیم و در عین حال به بهم ریختگی‌های احتمالی جلوگیری نماییم.

2 – معماری کلی سیستم:



در شکل قبل اجزای این معماری به شکا کلی آورده شده است.

**پیاده‌سازی میکروسرویس‌ها:**

بخش MainGateway

Ocelot یک کتابخانه میان‌افزار در زمینه مسیریابی و نگه‌داری دروازه‌ها (Gateway) برای سرویس‌های میکروسرویسی در دات‌نت است. این کتابخانه امکاناتی را برای تنظیم، مدیریت، و مسیریابی درخواست‌ها به سرویس‌های داخلی و خارجی فراهم می‌کند. از Ocelot می‌توان برای ایجاد یک دروازه API (API Gateway) استفاده کرد که به عنوان یک نقطه ورودی مرکزی به سرویس‌های مختلف عمل می‌کند و می‌تواند مسیریابی، تنظیمات امنیتی، زمانبندی، و مسائل مربوط به تطبیق پیغام‌ها را مدیریت کند

به عنوان یک مثال ساده، فرض کنید که می‌خواهید یک دروازه API ساده با استفاده از Ocelot در دات‌نت پیاده‌سازی کنید که درخواست‌ها را به سرویس‌های مختلفی هدایت کند. در این مثال، ما یک پروژه دات‌نت Core با استفاده از پکیج Ocelot نصب شده راه اندازی می‌کنیم.

dotnet add package Ocelot

**سپس، می‌توانید یک فایل تنظیمات به نام ocelot.json ایجاد کنید که مسیریابی درخواست‌ها را تعریف می‌کند. به عنوان مثال:**

**{**

**"Routes": [**

**{**

**"DownstreamPathTemplate": "/api/customers/{everything}",**

**"DownstreamScheme": "http",**

**"DownstreamHostAndPorts": [**

**{**

**"Host": "customer-service",**

**"Port": 5000**

**}**

**],**

**"UpstreamPathTemplate": "/customers/{everything}",**

**"UpstreamHttpMethod": [ "Get" ]**

**},**

**{**

**"DownstreamPathTemplate": "/api/products/{everything}",**

**"DownstreamScheme": "http",**

**"DownstreamHostAndPorts": [**

**{**

**"Host": "product-service",**

**"Port": 5001**

**}**

**],**

**"UpstreamPathTemplate": "/products/{everything}",**

**"UpstreamHttpMethod": [ "Get" ]**

**}**

**]**

**}**

**در ادامه لینک Documentation مربوطه آمده است.**

**https://ocelot.readthedocs.io/en/latest/**

**سرویس ها:**

است. به طور کلی هر سرویس از سه جز تشکیل میشود

1 – بخش API که Maingateway با آن تعامل میکند.

2 – بخش GRPC که در حقیقت سرویسی است که در اختیار بقیه سیستم وجود دارد و بقیه سرویس ها از طریق این API با آن تعامل میکنند.

3 – دیتابیس : در اینجا از کانتینر Postgres استفاده میکنیم.

برای جلوگیری از تکرار کد سه DLL زیر ساختی وجود دارد که برای تمام سرویس های دات نتی کاربرد دارد. که عبارتند از:

FumBaseApi

FumBaseCommon

FumBaseBusiness

مستندات برنامه نویسی :

به منظور جلوگیری از کدهای تکراری و وجود عملیات CRUD در سیستم برای هر موجودیت کلاس های زیر تعریف شده اند.

ServiceBase : این سرویس عملیات CRUD را برای هر موجودیت ایجاد میکند.

FumControllerBase : این سرویس برای controller ها اضافه شده است که عملیات CRUD را در محیط controller فراهم میکند.

برای اینکه یک موجودیت جدید به سیستم اضافه کنید صرفا کافی است مراحل زیر را طی بفرمایید.

برای مثال در اینجا قصد داریم یک موجودیت جدید بنام Term ترم تحصیلی اضافه بکنیم.

در ابتدا نیاز هست که یک کلاس معادل دیتابیسی آن به سیستم اضافه بشود.

public class Term: Entity

{

public string Title { get; set; }

public DateTime Start { get; set; }

public DateTime End { get; set; }

[JsonIgnore]

internal virtual List<Course> Courses { get; set; }

}

در قدم بعد نیاز هست که یک ویو مدل برای این موجودیت ساخته شود تا کاربر بتواند از این طریق با سیستم ارتباط بگیرد.

public class TermViewModel: IViewModel

{

public long ID { get; set; }

public string Title { get; set; }

public DateTime Start { get; set; }

public DateTime End { get; set; }

}

برای مباحث IOC یک interface برای این موجودیت میسازیم. چون عملیات CRUD را برای هر موجودیت نیاز داریم interface پایه ای برای این منظور فراهم شده که باید از آن ارث ببریم.

public interface ITermService: IServiceBase<Term, TermViewModel>

{

}

در ادامه پیاده سازی های مربوط به این موجودیت را داریم:

public class TermService : ServiceBase<Term, TermViewModel, GeneralDbContext>, ITermService

{

public TermService(GeneralDbContext ctx) :base(ctx)

{

}

public override TermViewModel FetchByID(long id)

{

return (

from t in FetchAll()

select new TermViewModel

{

ID = t.ID,

End = t.End,

Start = t.Start,

Title = t.Title,

}).Single();

}

}

بعد از این مرحله نیاز هست که سرویس خودمان را به سیستم DI معرفی کنیم.(هم چنین موجودیت باید به context نیز معرفی بشود)

builder.Services.AddScoped<ITermService, TermService>();

در آخر نیز controller متناظر با این موجودیت را به سیستم معرفی میکنیم.

public class TermController : FumControllerBase<Term, TermViewModel, ITermService>

{

public TermController(ITermService service) : base(service) { }

}

در نهایت باید توجه داشته باشیم که routeهای مورد نظر به gateway اضافه شده باشند.

تست و اجرا:

برای اجرا پروه نیاز هست که فایل docker-compose قرار گرفته در root پروژه را اجرا بفرمایید.

برای تست این سرویس هم فایل Postman در root پروژه موجود است.

در نهایت لینک برنامه های استفاده شده در این پروژه آمده است:

<https://hub.docker.com/_/postgres>

<https://hub.docker.com/_/rabbitmq>

<https://github.com/ThreeMammals/Ocelot>

<https://github.com/dotnet/efcore>