**for architecture microservice :**

**ایجاد پروژه:**

**پروژه را به ماژول‌های جداگانه تقسیم کنید (مانند Domain, Application, Infrastructure, Presentation).**

**لایه Domain:**

**تعریف موجودیت‌ها و اینترفیس‌های اصلی.**

**نوشتن قوانین کسب‌وکار.**

**لایه**

**Application:**

**سرویس‌هایی برای مدیریت منطق دامنه و اجرای عملیات‌ها.**

**استفاده از الگوی CQRS یا Mediator.**

**لایه**

**Infrastructure:**

**پیکربندی پایگاه داده (Entity Framework Core).**

**پیاده‌سازی Repositoryها و سایر اینترفیس‌ها.**

**لایه**

**Presentation:**

**ساخت WebApp یا API برای تعامل با کاربران.**

**استفاده از Dependency Injection برای ارتباط بین لایه‌ها.**

**Solution**

**├── OrdersService**

**│ ├── Controllers**

**│ ├── Services**

**│ ├── Data**

**│ ├── DTOs**

**│ ├── Startup.cs**

**│**

**├── UsersService**

**│ ├── Controllers**

**│ ├── Services**

**│ ├── Data**

**│ ├── DTOs**

**│ ├── Startup.cs**

**│**

**├── PaymentsService**

**│ ├── Controllers**

**│ ├── Services**

**│ ├── Data**

**│ ├── DTOs**

**│ ├── Startup.cs**

**alt +195 ---> ├**

**alt + 196 ---> ─**

**alt + 124 ---> |**

**CQRS (Command and Query Responsibility Segregation)**

**CQRS** یک الگوی معماری است که برای جداسازی مسئولیت‌ها در سیستم‌های نرم‌افزاری استفاده می‌شود. این الگو پیشنهاد می‌کند که عملیات مربوط به **خواندن داده‌ها** (Queries) و **نوشتن داده‌ها** (Commands) در سیستم به صورت جداگانه مدیریت شوند.

**مفهوم و مزایا**

1. **جداسازی مسئولیت‌ها:**
   * عملیات خواندن (Query) و نوشتن (Command) از یکدیگر جدا می‌شوند.
   * بهبود خوانایی و ساده‌تر شدن کد.
2. **مقیاس‌پذیری:**
   * عملیات خواندن و نوشتن می‌توانند به صورت جداگانه مقیاس‌پذیر شوند.
   * برای مثال، پایگاه داده‌های مجزا برای خواندن و نوشتن می‌توانند استفاده شوند.
3. **انعطاف‌پذیری بیشتر:**
   * هر قسمت (Command یا Query) می‌تواند با طراحی یا تکنولوژی‌های متفاوتی پیاده‌سازی شود.

**ساختار CQRS**

* **Command:** مسئول نوشتن و تغییر وضعیت سیستم.
* **Query:** مسئول خواندن داده‌ها بدون تغییر وضعیت.

**پیاده‌سازی ساده CQRS در ASP.NET Core**

**Command:**

public class CreateOrderCommand

{

public int ProductId { get; set; }

public int Quantity { get; set; }

}

**Query:**

public class GetOrderByIdQuery

{

public int OrderId { get; set; }

}

**Handler:**

public class CreateOrderCommandHandler

{

public void Handle(CreateOrderCommand command)

{

// نوشتن داده در دیتابیس

}

}

public class GetOrderByIdQueryHandler

{

public Order Handle(GetOrderByIdQuery query)

{

// خواندن داده از دیتابیس

return new Order();

}

}

**Mediator Pattern**

الگوی **Mediator** (میانجی) یک الگوی طراحی رفتاری است که برای مدیریت ارتباط بین اشیاء و کاهش وابستگی مستقیم آن‌ها به یکدیگر استفاده می‌شود. در این الگو، یک شیء مرکزی به نام **Mediator** مسئول هماهنگی ارتباطات بین اشیاء مختلف است.

**مزایای Mediator**

1. **کاهش وابستگی‌ها:**
   * اشیاء مستقیماً با یکدیگر تعامل ندارند؛ بلکه از طریق Mediator ارتباط برقرار می‌کنند.
   * کد ساده‌تر و تغییرپذیری بالاتر.
2. **تمرکزگرایی:**
   * تمام تعاملات در یک نقطه مرکزی مدیریت می‌شود.

**Mediator در ASP.NET Core با استفاده از کتابخانه MediatR**

[**MediatR**](https://github.com/jbogard/MediatR) یک کتابخانه محبوب برای پیاده‌سازی Mediator Pattern است و اغلب با CQRS ترکیب می‌شود.

**ساختار پیاده‌سازی MediatR**

1. **نصب MediatR:**

dotnet add package MediatR.Extensions.Microsoft.DependencyInjection

1. **تعریف Command:**

public class CreateOrderCommand : IRequest<int>

{

public int ProductId { get; set; }

public int Quantity { get; set; }

}

1. **تعریف Handler:**

public class CreateOrderCommandHandler : IRequestHandler<CreateOrderCommand, int>

{

public async Task<int> Handle(CreateOrderCommand request, CancellationToken cancellationToken)

{

// ذخیره داده در دیتابیس

return 1; // شناسه سفارش ایجاد شده

}

}

1. **استفاده از MediatR در کنترلر:**

[ApiController]

[Route("api/orders")]

public class OrdersController : ControllerBase

{

private readonly IMediator \_mediator;

public OrdersController(IMediator mediator)

{

\_mediator = mediator;

}

[HttpPost]

public async Task<IActionResult> CreateOrder(CreateOrderCommand command)

{

var orderId = await \_mediator.Send(command);

return Ok(orderId);

}

}

1. **ثبت در DI Container:** در فایل Program.cs یا Startup.cs:

builder.Services.AddMediatR(typeof(Program));

**مقایسه CQRS و Mediator**

* **CQRS**: یک معماری است که خواندن و نوشتن داده‌ها را از هم جدا می‌کند.
* **Mediator**: یک الگوی طراحی برای مدیریت ارتباط بین اجزاء بدون وابستگی مستقیم.

**ترکیب CQRS و Mediator**

این دو اغلب با هم استفاده می‌شوند. به این صورت که **Command**‌ها و **Query**‌ها با استفاده از **Mediator** مدیریت و هماهنگ می‌شوند. این ترکیب مزایای هر دو روش را فراهم می‌آورد:

* جداسازی خواندن و نوشتن (CQRS)
* کاهش وابستگی‌ها و ارتباطات ساده‌تر بین بخش‌ها (Mediator).

**3. ارتباط بین سرویس‌ها**

ارتباط بین میکروسرویس‌ها معمولاً از طریق پروتکل‌های زیر انجام می‌شود:

1. **HTTP (REST APIs):**
   * استفاده از کنترلرهای ASP.NET Core برای ایجاد APIهای RESTful.
2. **پیام‌رسانی (Messaging):**
   * استفاده از ابزارهایی مانند **RabbitMQ**, **Kafka** یا **Azure Service Bus**.
3. **gRPC:**
   * مناسب برای ارتباطات سریع و بهینه بین سرویس‌ها.

**4. مدیریت Configuration**

* از ابزارهایی مانند **Consul** یا **ETCD** برای مدیریت تنظیمات و Discovery سرویس‌ها استفاده کنید.
* ASP.NET Core قابلیت تنظیم Configuration را از طریق JSON، Environment Variables و Secrets فراهم می‌کند.

**5. دیتابیس مستقل**

* هر سرویس باید پایگاه داده خود را داشته باشد تا استقلال کامل حفظ شود.
* استفاده از الگوی **Database per Service** توصیه می‌شود.

**6. مدیریت API Gateway**

* برای مدیریت درخواست‌های ورودی و مسیریابی آن‌ها به سرویس‌های مناسب از یک API Gateway مانند **Ocelot** یا **YARP** استفاده کنید.

**7. امنیت**

* احراز هویت و مجوزدهی می‌تواند از طریق یک سرویس مرکزی انجام شود.
* استفاده از **IdentityServer** یا **Azure AD** برای OAuth2 و OpenID Connect معمول است.

**8. نظارت و Logging**

* از ابزارهایی مانند **Elastic Stack (ELK)**، **Prometheus** و **Grafana** برای نظارت بر وضعیت سرویس‌ها استفاده کنید.
* ASP.NET Core دارای قابلیت Logging داخلی است که می‌توانید آن را با ابزارهای خارجی ترکیب کنید.

**چالش‌ها و نکات مهم**

1. **پیچیدگی مدیریت:**
   * افزایش تعداد سرویس‌ها نیازمند مدیریت پیچیده‌تر (مانند DevOps و CI/CD) است.
2. **Consistency:**
   * در میکروسرویس‌ها به جای **ACID Transactions**، از الگوهایی مانند **Eventual Consistency** استفاده می‌شود.
3. **استقرار:**
   * نیازمند ابزارهای پیشرفته مانند Docker و Kubernetes برای مدیریت کانتینرها و استقرار سرویس‌ها است.
4. **Dependency Management:**
   * برای کاهش وابستگی‌ها، باید اصول **Loose Coupling** رعایت شود.

**مثال پیاده‌سازی ساده**

**OrdersService**

csharp

Copy code

[ApiController]

[Route("api/orders")]

public class OrdersController : ControllerBase

{

[HttpGet("{id}")]

public IActionResult GetOrderById(int id)

{

return Ok(new { OrderId = id, Product = "Laptop", Price = 1200 });

}

}

**UsersService**

csharp

Copy code

[ApiController]

[Route("api/users")]

public class UsersController : ControllerBase

{

[HttpGet("{id}")]

public IActionResult GetUserById(int id)

{

return Ok(new { UserId = id, Name = "John Doe", Email = "john@example.com" });

}

}

**API Gateway با Ocelot**

ocelot.json:

json

Copy code

{

"Routes": [

{

"DownstreamPathTemplate": "/api/orders/{everything}",

"DownstreamScheme": "http",

"DownstreamHostAndPorts": [{ "Host": "localhost", "Port": 5001 }],

"UpstreamPathTemplate": "/orders/{everything}",

"UpstreamHttpMethod": [ "GET" ]

},

{

"DownstreamPathTemplate": "/api/users/{everything}",

"DownstreamScheme": "http",

"DownstreamHostAndPorts": [{ "Host": "localhost", "Port": 5002 }],

"UpstreamPathTemplate": "/users/{everything}",

"UpstreamHttpMethod": [ "GET" ]

}

],

"GlobalConfiguration": {

"BaseUrl": "http://localhost:5000"

}

}

این ساختار کمک می‌کند تا میکروسرویس‌ها به صورت مستقل عمل کرده و قابلیت مقیاس‌پذیری و انعطاف‌پذیری بیشتری داشته باشند.