DEV ROS Meetup 1° EDICION

GOING MICROSERVICES WITH .NET

David Revoledo

@deirevoledo

Contamos con el apoyo de:

DGENIX endava RSK UAI Microsoft Most Valuable Professional SUBWAY

Contamos con el apoyo de:

About me !

David Revoledo | @deirevoledo

- Working with Microsoft technologies pretty much all my career.
- Speaker
- .Net, Azure Trainer and Consultant.
- Open Source active contributor (Azure event hub, service bus SDK's and a Few Plugins).
- Focused on Asp. NET Core / Azure
- Software Architect at DGENIX

Agenda

1. Microservices Introduction
2. Starting Microservices with .NET
3. Conclusion
4. Q&A

WHAT ARE MICROSERVICES?
SERVICES FOR ANTS???!
makeameme.org

Microservices?

Let’s start saying is a very bad name.

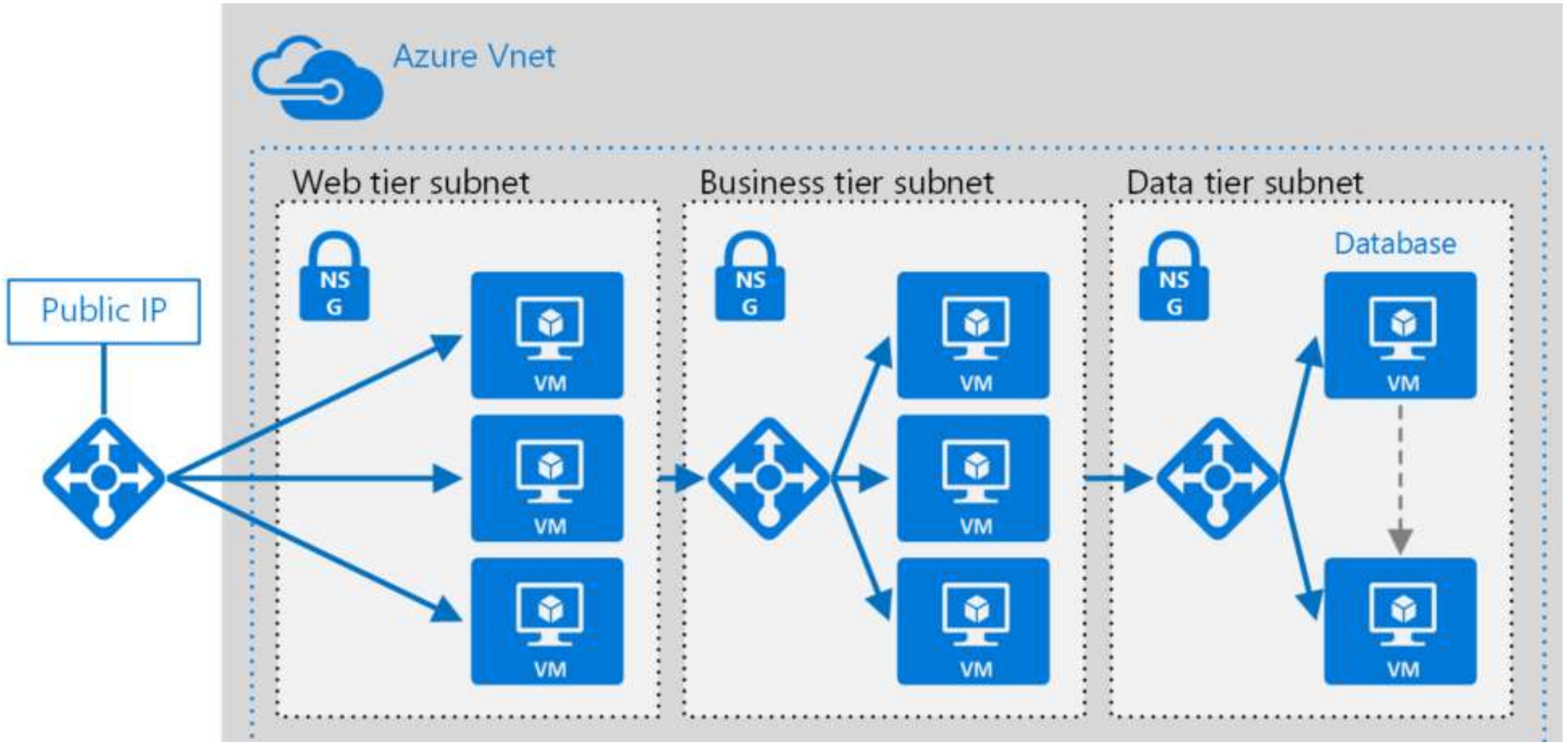
- Serverless ?
- NoSQL ?
- Data Lake ?

What is a Microservices Architecture?

“a microservices architecture as a service-oriented architecture composed of loosely coupled elements that have bounded contexts. “

?

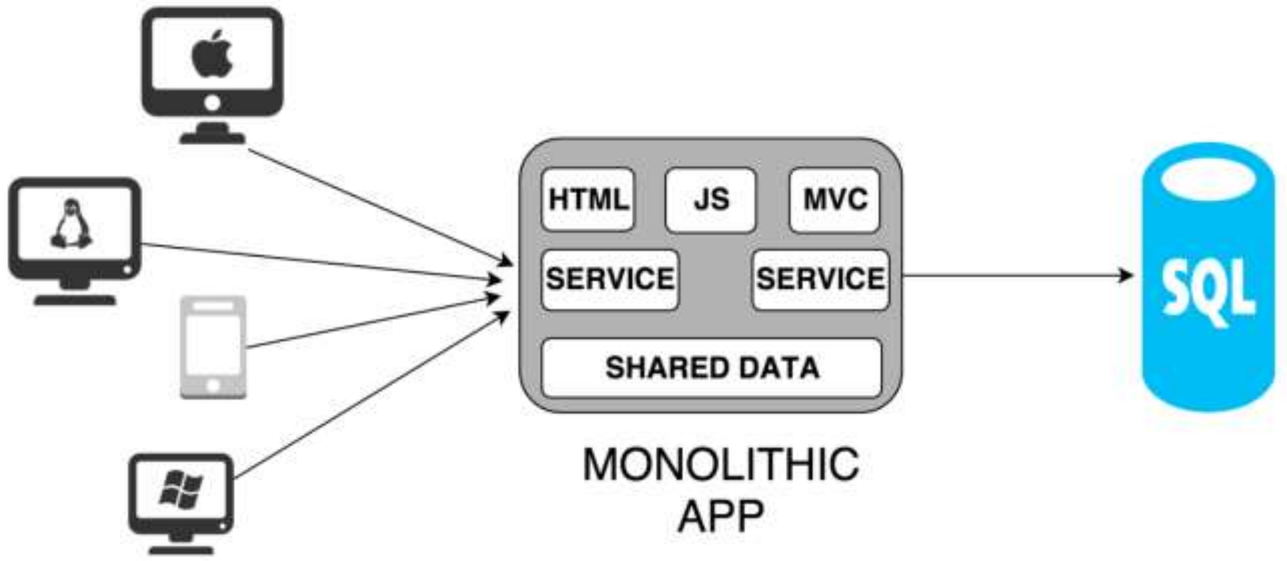
But what is the difference with a Layer/Tier (Coding) Architecture



DEV ROS Meetup 1° EDICION

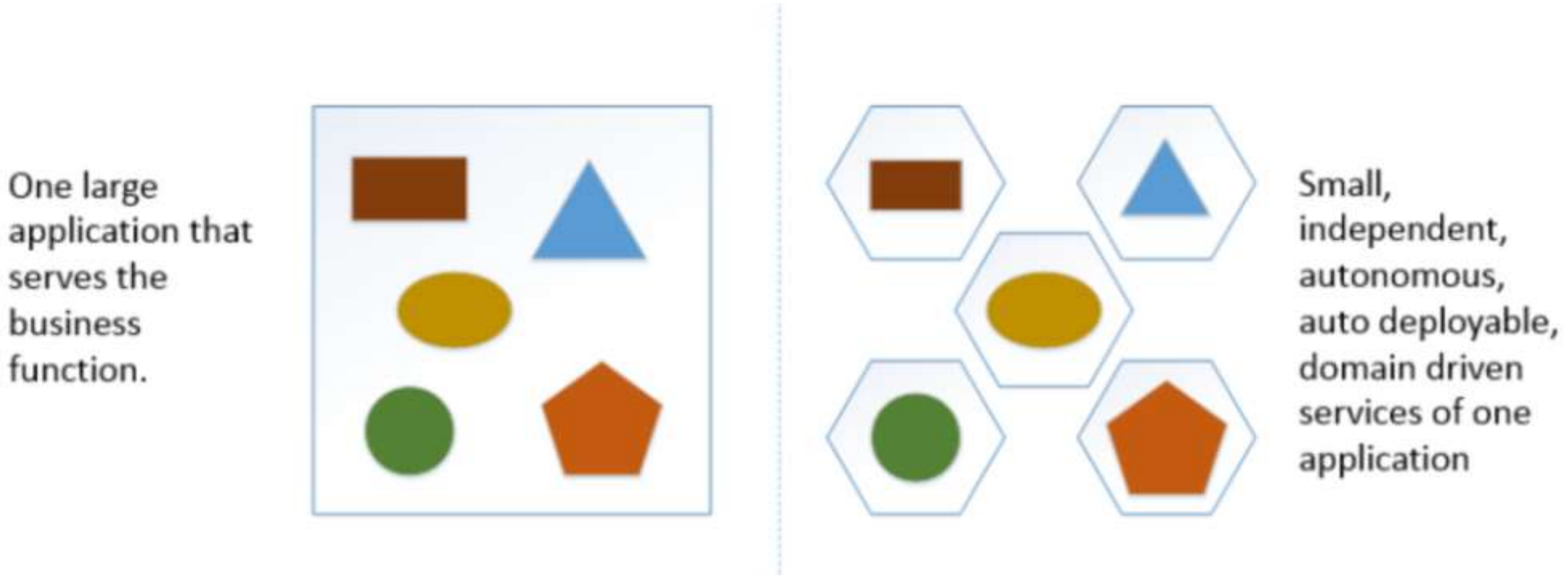
What microservices are not

- When a solutions need to be deployed complete to provide a “new version” of a product.
- When a component failing is causing a complete system failure.



DEV ROS Meetup 1° EDICION

Microservices is “Multi Process Applications”


One large application that serves the business function.
Small, independent, autonomous, auto deployable, domain driven services of one application

Common characteristics

- Componentization via Services by **Domain "bounded context".**
- Organized around business capabilities.
- Decentralized Data Management.
- Infrastructure Automation.
- Design for Failure.
- Aggressive monitoring.

Benefits

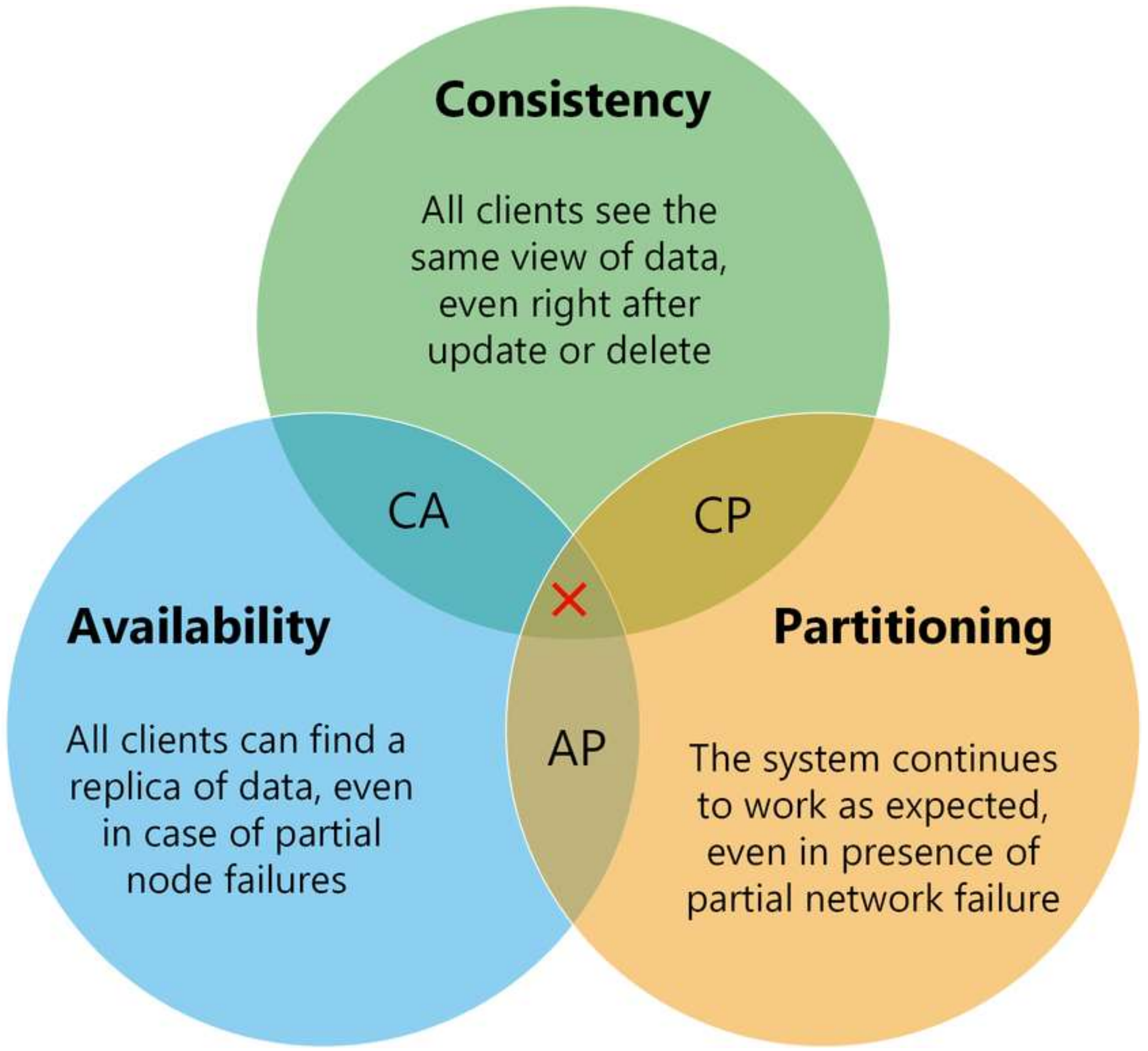
- Allows to create pieces of software in loosely coupled services kind.
- Allow to release features/fixes continuously (better Time To Market)
- Allows to evolve the company technology stack.
- Allows to migrate a services changing nothing if the contract does not change.
- Because MS is built with failures in mind, we are building resilience.
- Because MS is built with automation in mind, we are saving time/money in the future.
- Because MS is built with monitoring in mind, we have a better control how what is happening.
- We don’t have to marry with a technology/framework (devs engagement)
- Let us built software with business first with a service scope instead of a technical layers.

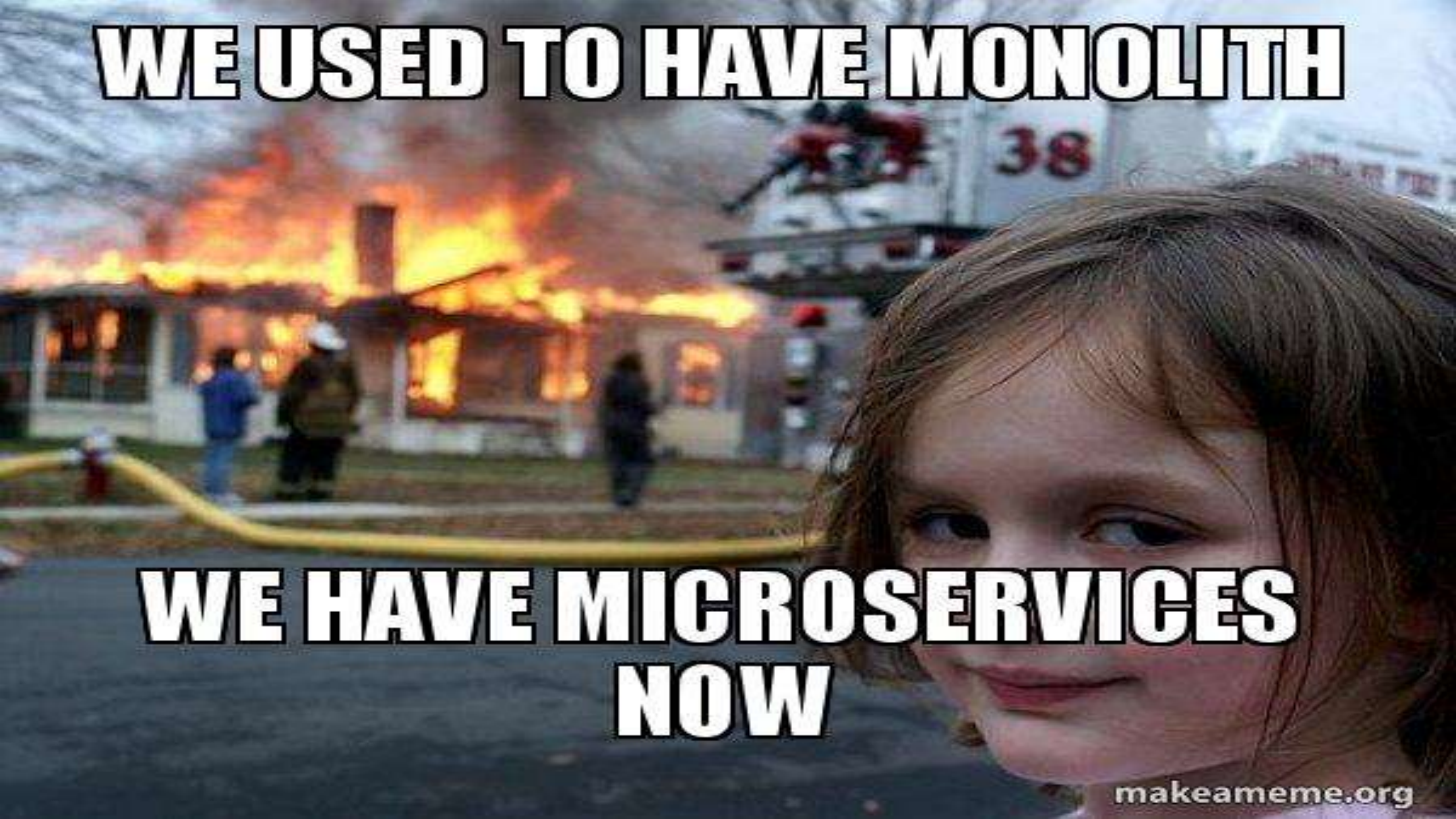
There ain't no such thing as a free lunch

- Inter Process Communication.
- Async Calling by Messaging.
- Eventual Consistency.
- Duplicated data.
- Process failures.

DEV ROS Meetup 1° EDICION

CAP Theorem

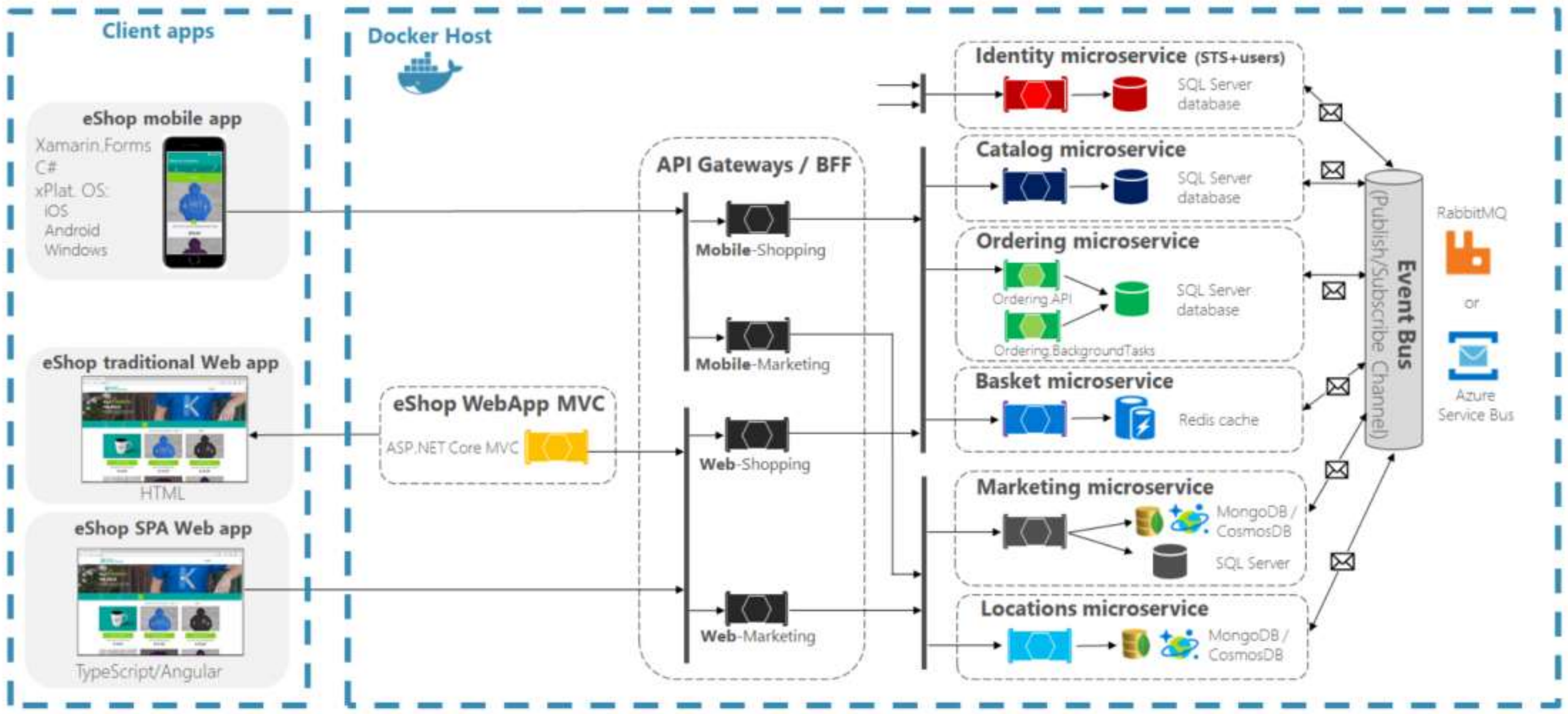

Consistency
All clients see the same view of data, even right after update or delete
CA
CP
×
Availability
Partitioning
AP
All clients can find a replica of data, even in case of partial node failures
The system continues to work as expected, even in presence of partial network failure


WE USED TO HAVE MONOLITH
38
WE HAVE MICROSERVICES NOW
makeameme.org

Microservices Architecture Is Not a Silver Bullet

- As is not just a technical decision, it require a mindset change of the complete team including the client.
- Business boundaries need to be identified.
- Is the solution being too simply, probably MS will bring more complexity than flexibility.
- You need to be mature with Devops. Practices.

Let's start MS with NET

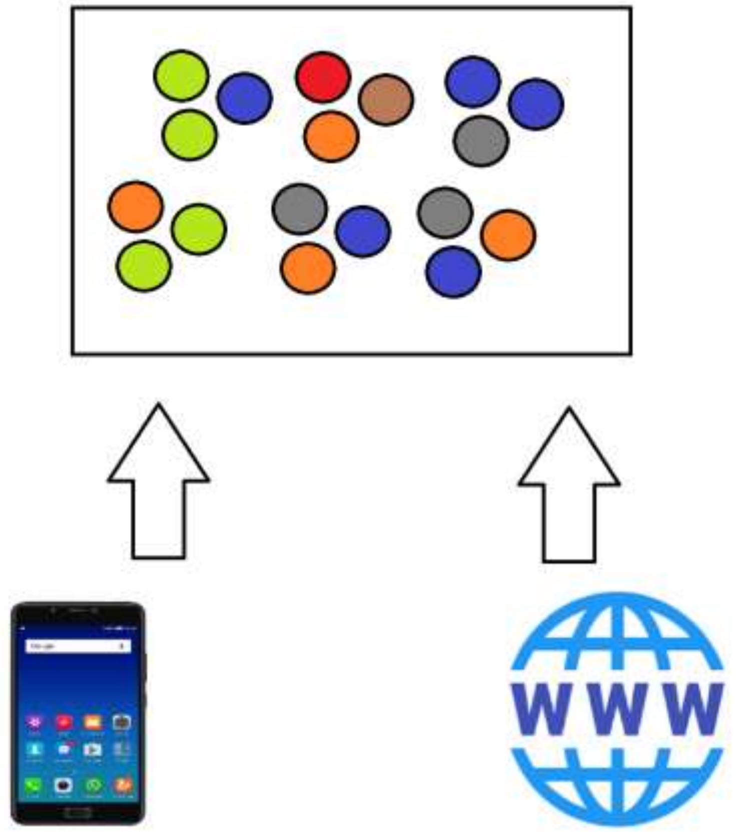


Structure

- DDD / CRQS ❤️ Microservices
- Messaging communication between services
- Api Gateway
- Retries.
- Containers.

Api Gateway

- How do the clients of a Microservices-based application access the individual services?



DEV ROS Meetup 1° EDICION

API Gateways with Ocelot

Api Gateway

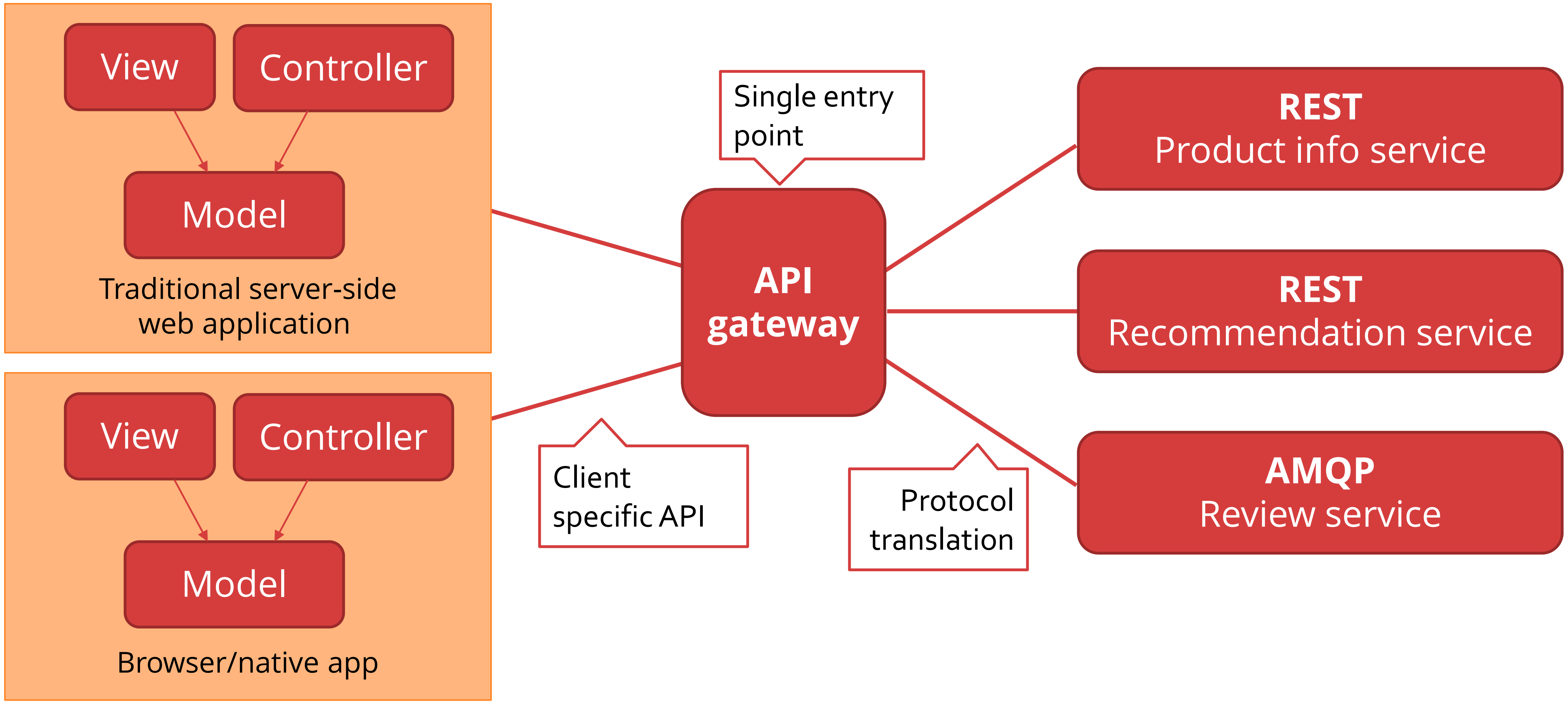
- Coupling: Without the API Gateway pattern the client apps are coupled to the internal microservices.

- Too many round-trips: A single page/screen in the client app might require several calls to multiple services.

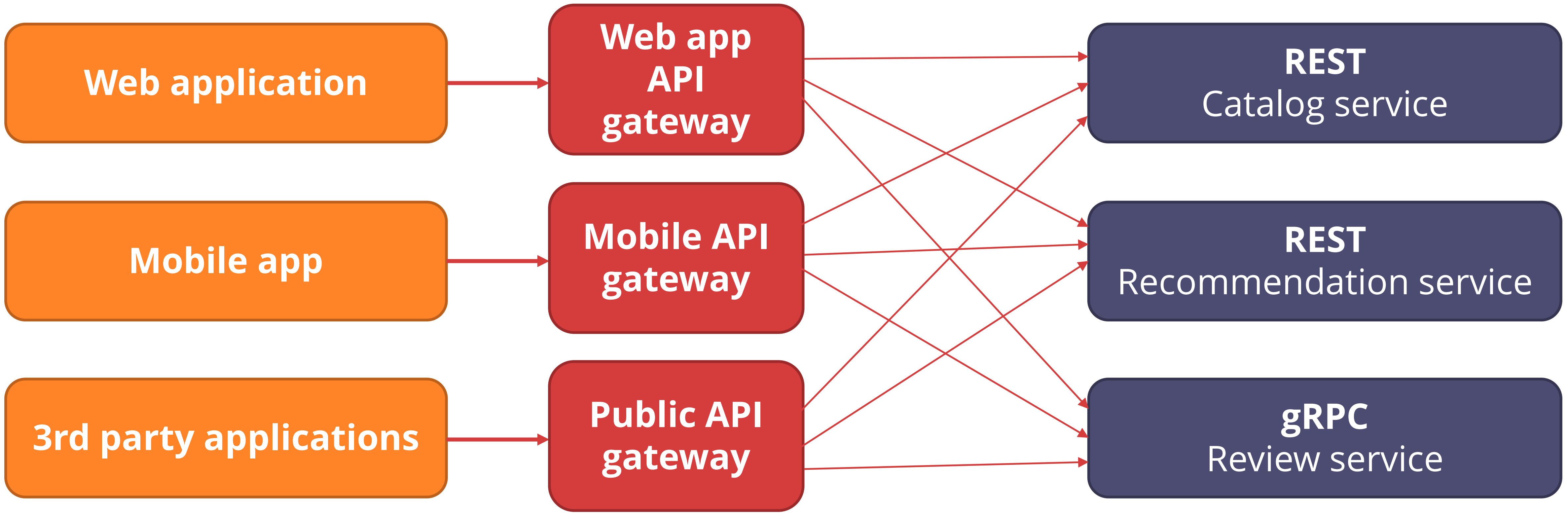
- Security issues: Without a gateway, all the microservices must be exposed to the "external world“. The smaller is the attack surface, the more secure your application can be.

- Cross-cutting concerns: Each publicly published microservice must handle concerns such as authorization, SSL, etc. In many situations those concerns could be handled in a single tier so the internal microservices are simplified.

Use an API Gateway



Variation: Backends for frontends

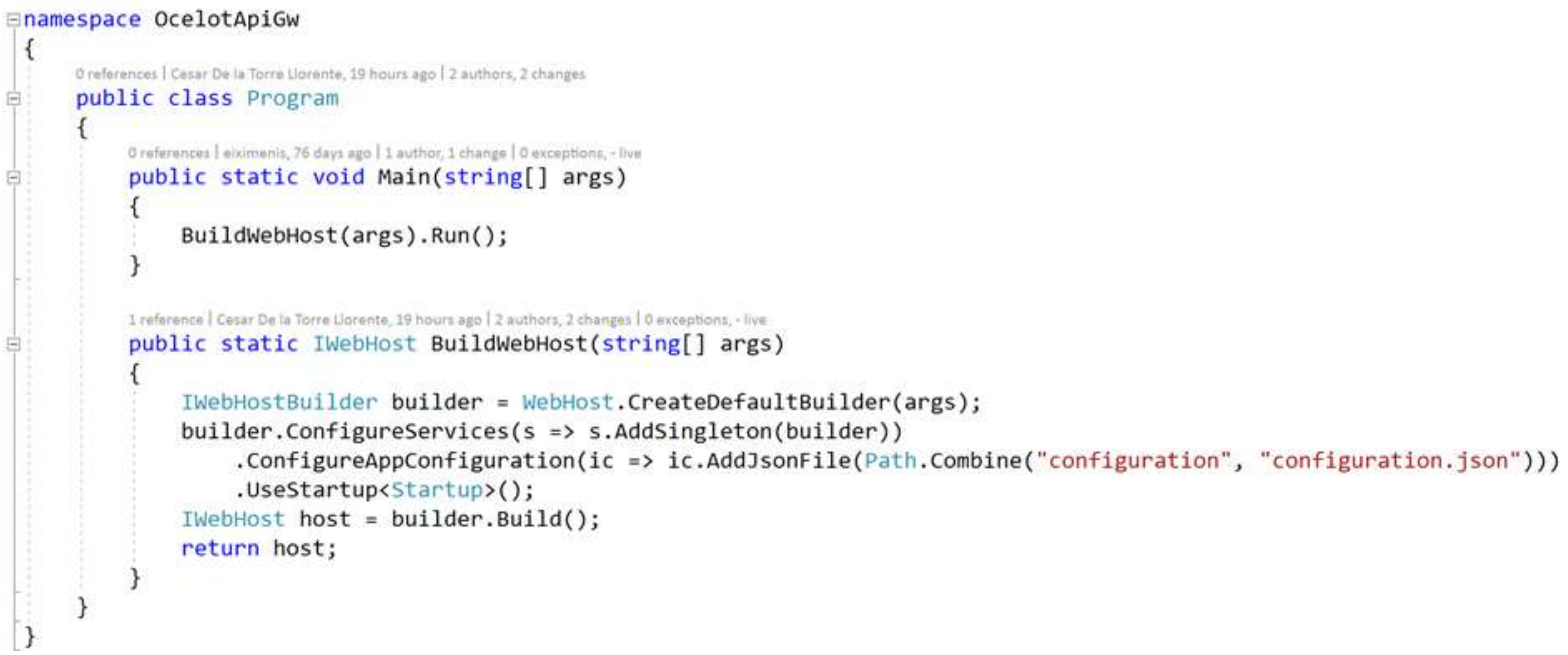


Ocelot Api Gateway

- Routing
- Request Aggregation
- Service Discovery with Consul & Eureka
- Service Fabric
- WebSockets
- Authentication
- Authorization
- Rate Limiting
- Caching
- Retry policies / QoS
- Load Balancing
- Logging / Tracing / Correlation
- Headers / Query String / Claims Transformation
- Custom Middleware / Delegating Handlers
- Configuration / Administration REST API
- Platform / Cloud Agnostic
- Etc.

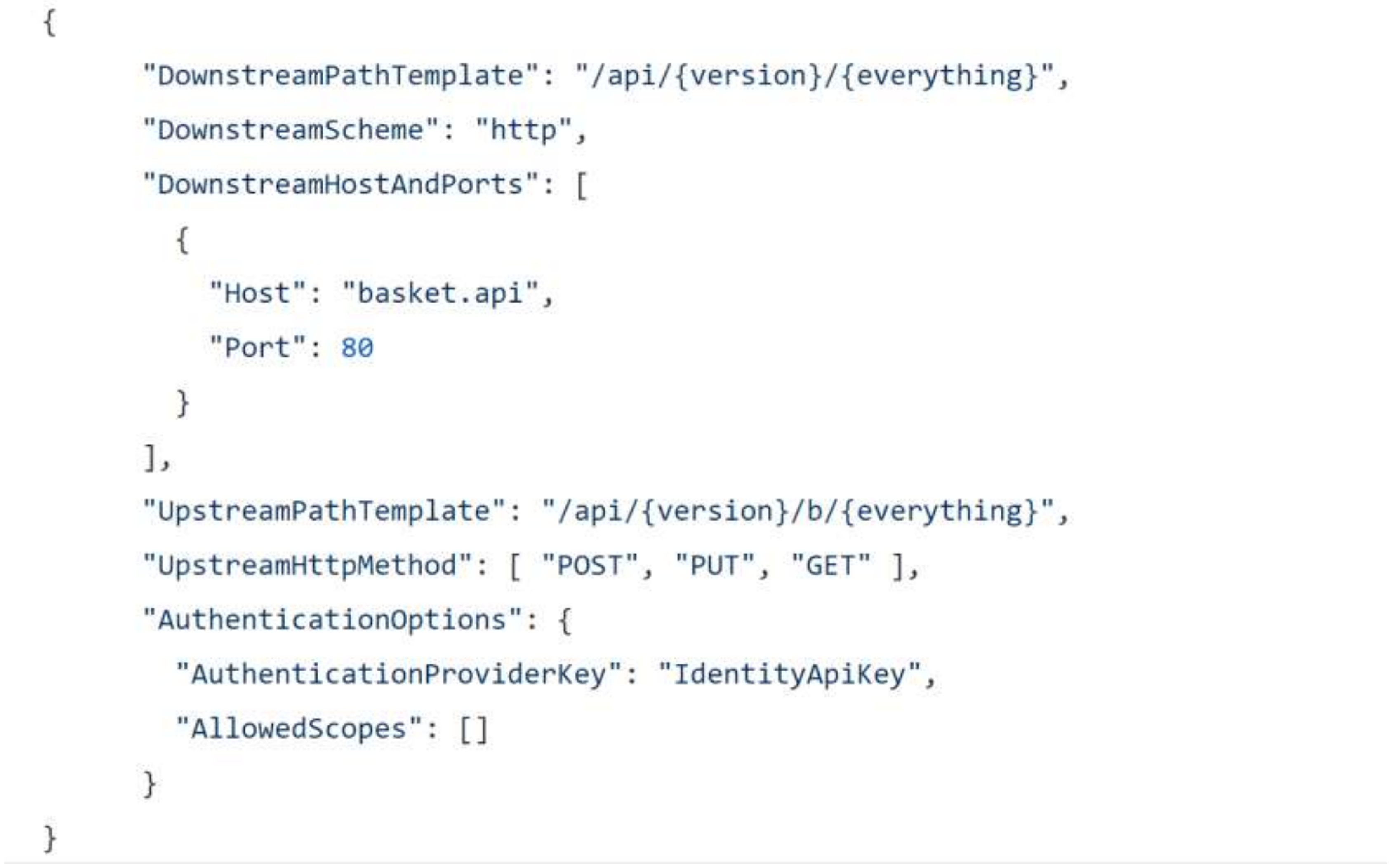
Ocelot ASP.NET Core configuration.

dotnet add package Ocelot



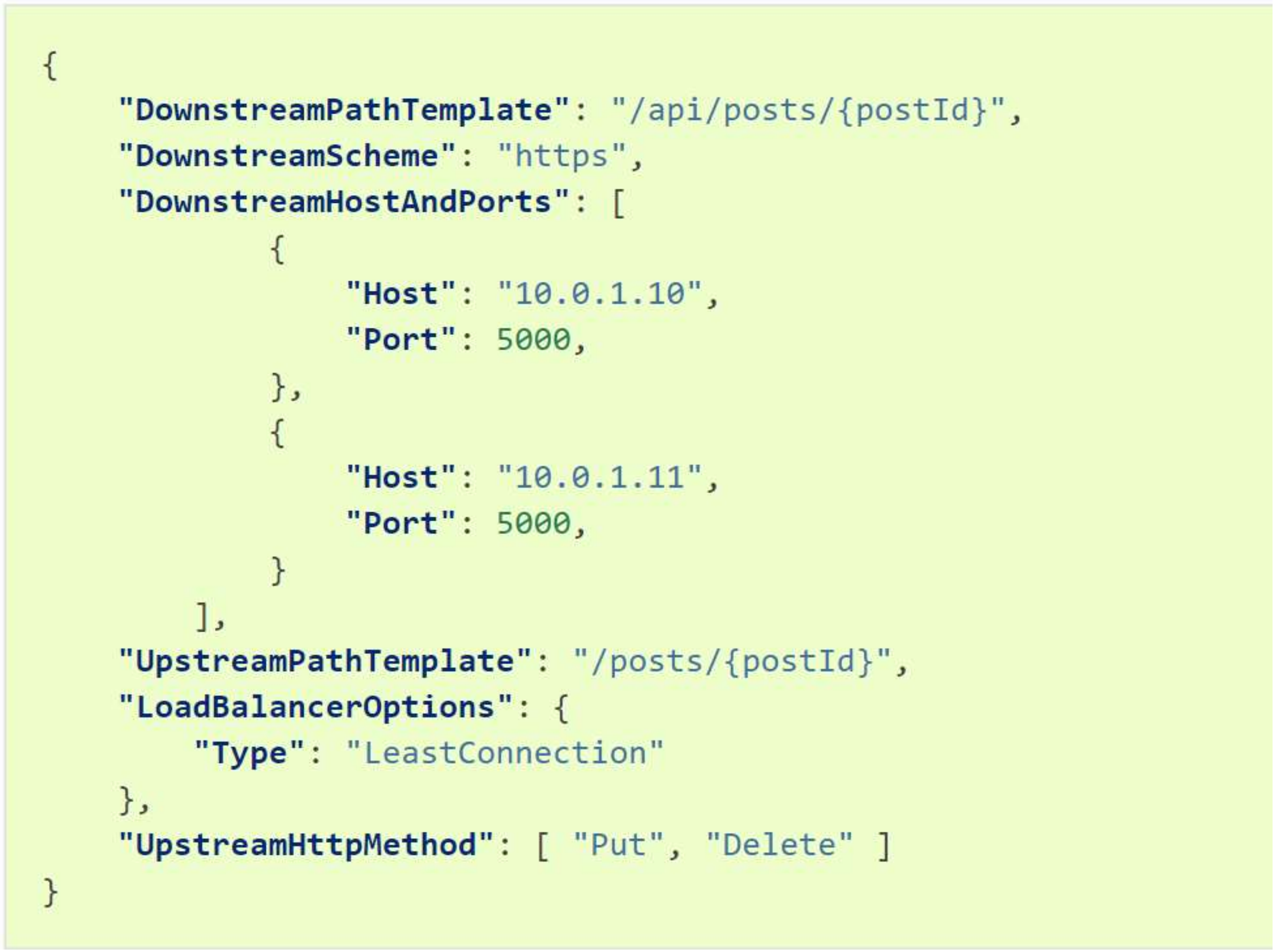
```
namespace OcelotApiGw
{
    0 references | Cesar De la Torre Llorente, 19 hours ago | 2 authors, 2 changes
    public class Program
    {
        0 references | eiximenis, 76 days ago | 1 author, 1 change | 0 exceptions, - live
        public static void Main(string[] args)
        {
            BuildWebHost(args).Run();
        }

        1 reference | Cesar De la Torre Llorente, 19 hours ago | 2 authors, 2 changes | 0 exceptions, - live
        public static IWebHost BuildWebHost(string[] args)
        {
            IWebHostBuilder builder = WebHost.CreateDefaultBuilder(args);
            builder.ConfigureServices(s => s.AddSingleton(builder))
                .ConfigureAppConfiguration(ic => ic.AddJsonFile(Path.Combine("configuration", "configuration.json")))
                .UseStartup<Startup>();
            IWebHost host = builder.Build();
            return host;
        }
    }
}
```

Ocelot ASP.NET Core configuration.



```
{
        "DownstreamPathTemplate": "/api/{version}/{everything}",
        "DownstreamScheme": "http",
        "DownstreamHostAndPorts": [
          {
            "Host": "basket.api",
            "Port": 80
          }
        ],
        "UpstreamPathTemplate": "/api/{version}/b/{everything}",
        "UpstreamHttpMethod": [ "POST", "PUT", "GET" ],
        "AuthenticationOptions": {
          "AuthenticationProviderKey": "IdentityApiKey",
          "AllowedScopes": []
        }
}
```

Ocelot Load balancing.



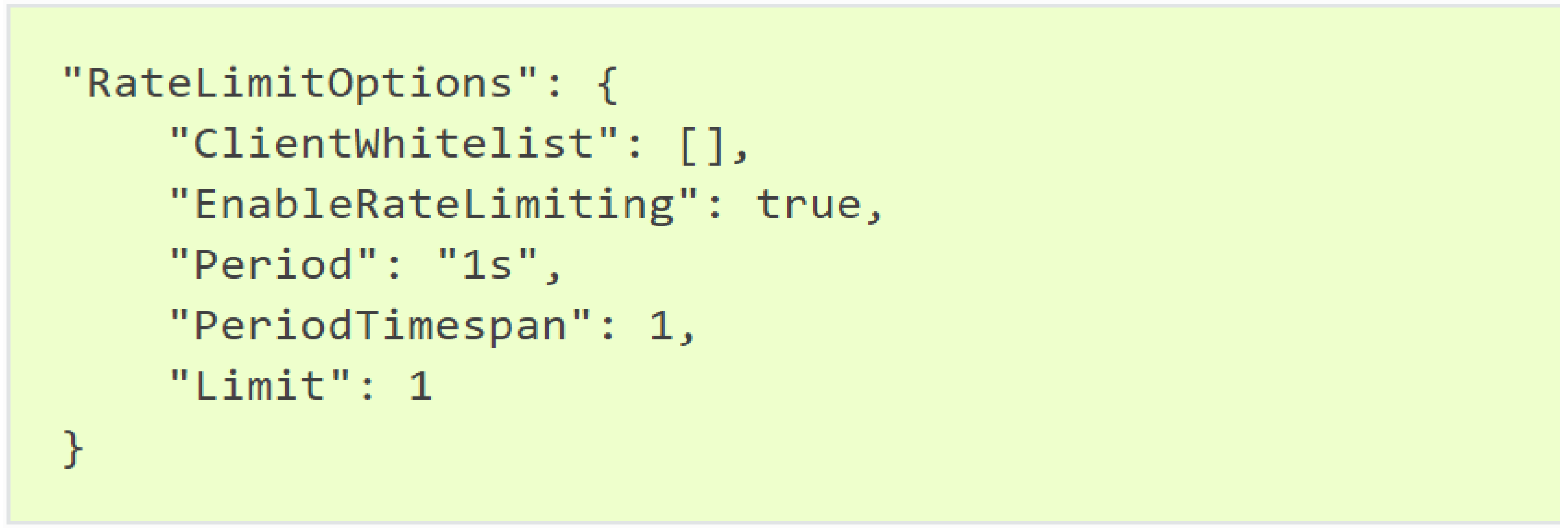
```
{
    "DownstreamPathTemplate": "/api/posts/{postId}",
    "DownstreamScheme": "https",
    "DownstreamHostAndPorts": [
            {
                "Host": "10.0.1.10",
                "Port": 5000,
            },
            {
                "Host": "10.0.1.11",
                "Port": 5000,
            }
        ],
    "UpstreamPathTemplate": "/posts/{postId}",
    "LoadBalancerOptions": {
        "Type": "LeastConnection"
    },
    "UpstreamHttpMethod": [ "Put", "Delete" ]
}
```

Ocelot Auth.

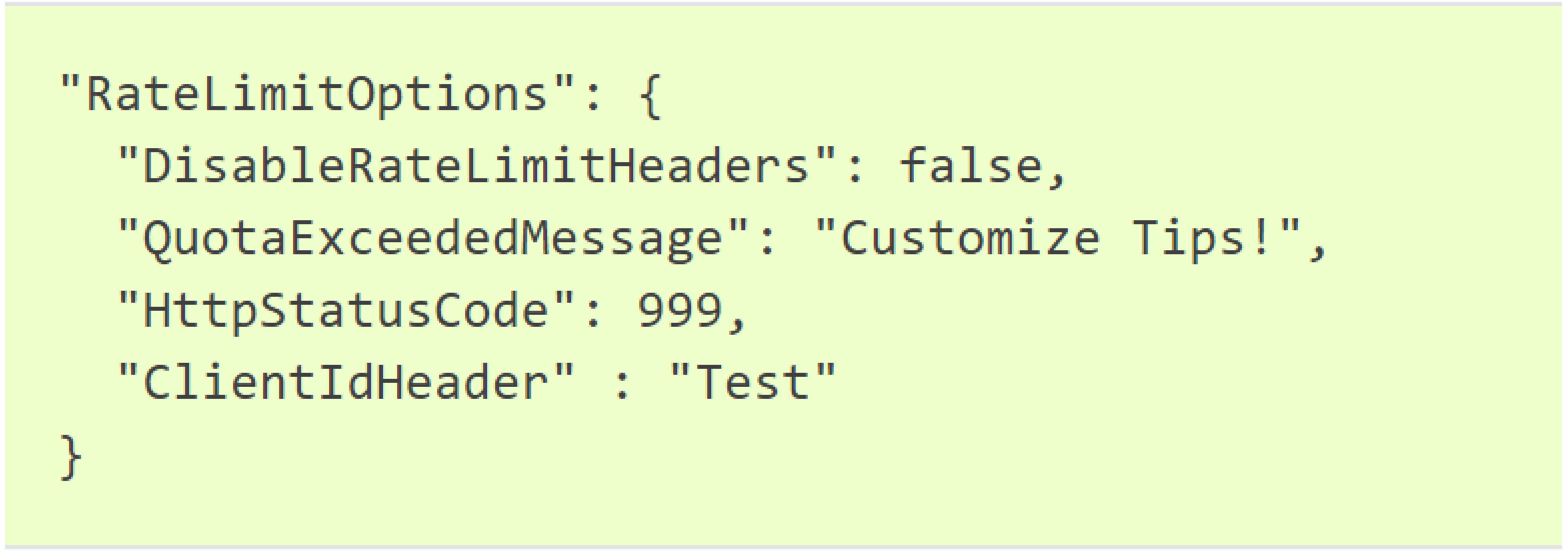
```
public void ConfigureServices(IServiceCollection services)
{
    var authenticationProviderKey = "TestKey";

    services.AddAuthentication()
        .AddJwtBearer(authenticationProviderKey, x =>
        {
            x.Authority = "test";
            x.Audience = "test";
        });

    services.AddOcelot();
}
```

```
"ReRoutes": [{
        "DownstreamHostAndPorts": [
            {
                "Host": "localhost",
                "Port": 51876,
            }
        ],
        "DownstreamPathTemplate": "/",
        "UpstreamPathTemplate": "/",
        "UpstreamHttpMethod": ["Post"],
        "ReRouteIsCaseSensitive": false,
        "DownstreamScheme": "http",
        "AuthenticationOptions": {
            "AuthenticationProviderKey": "TestKey",
            "AllowedScopes": []
        }
    }]
```

Ocelot Rate Limits.



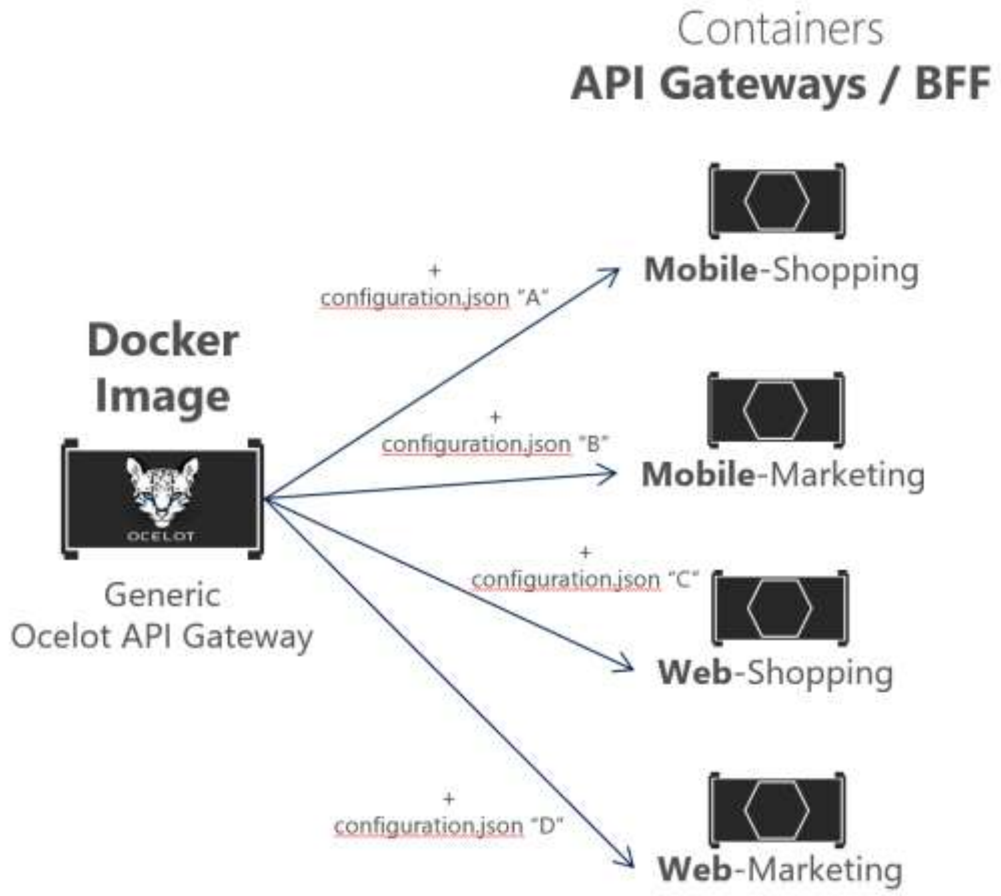
```
"RateLimitOptions": {
    "ClientWhitelist": [],
    "EnableRateLimiting": true,
    "Period": "1s",
    "PeriodTimespan": 1,
    "Limit": 1
}
```



```
"RateLimitOptions": {
  "DisableRateLimitHeaders": false,
  "QuotaExceededMessage": "Customize Tips!",
  "HttpStatusCode": 999,
  "ClientIdHeader" : "Test"
}
```

Ocelot

- Is a code level gateway useful also to run multiple services in a single docker container image.



DEV ROS Meetup 1° EDICION

Options.

For others features not supported by Ocelot or a better infrastructure throughput consider infrastructure solutions like.

- Azure Application Gateway.
- Azure Api Management.
- Others

Messaging between Microservices “Breaking RPC”

- Async communication between microservices
- Event driven design
- Easy to add a sub-action for an event/command without change the communication in the producer

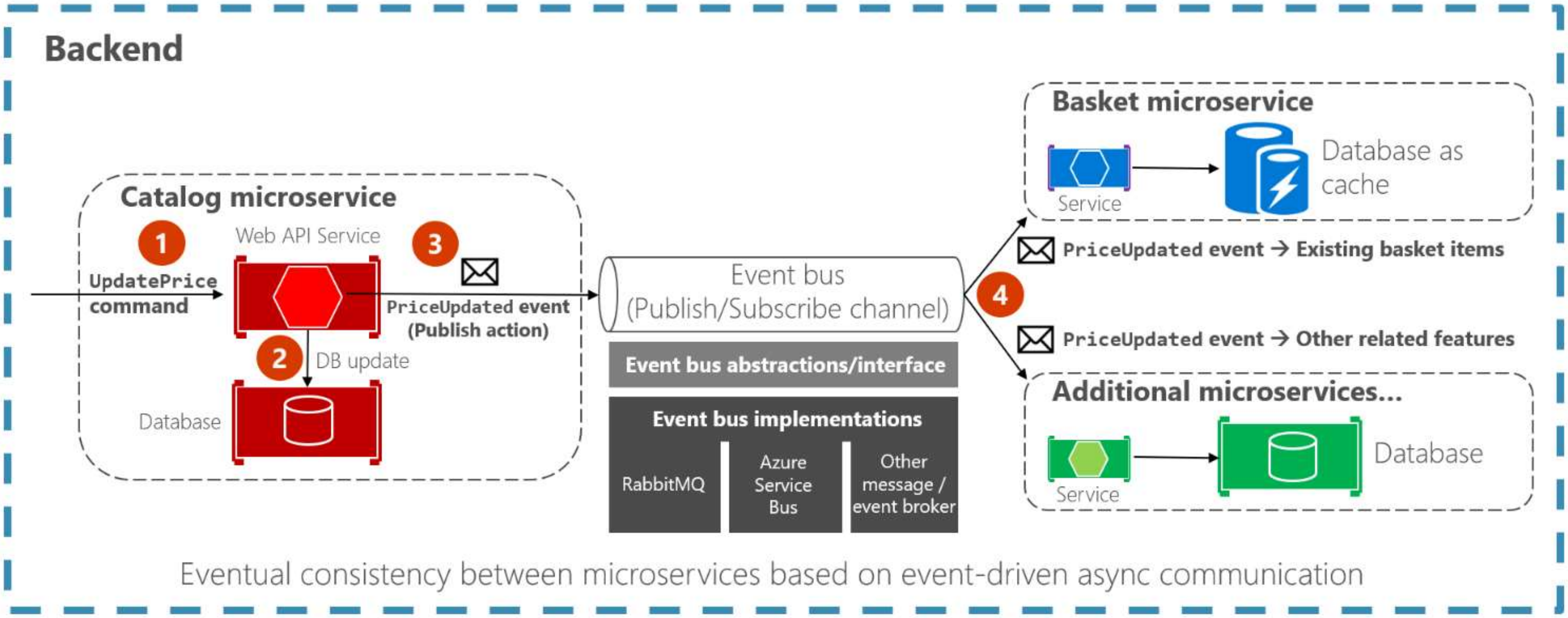
RabbitMQ

Messaging in Microservices

The more nodes there are, more work load Messaging will improve performance, resilience and resources usages.

- If there is a network outage / issue with the code with an RCP call the operations will break down and client need to execute it again.
- With RPC, threads are allocated with load, the more nodes there are more accumulated memory you have. (Gen2 Garbage collector problem).
- Instead of using threads and memory the work going through durable disk saved messages.

Implementing asynchronous event-driven communication with an event bus


Backend
Catalog microservice
1
UpdatePrice command
Web API Service
3
PriceUpdated event (Publish action)
2
DB update
Database
Event bus (Publish/Subscribe channel)
Event bus abstractions/interface
Event bus implementations
RabbitMQ
Azure Service Bus
Other message / event broker
4
Basket microservice
Service
Database as cache
PriceUpdated event → Existing basket items
PriceUpdated event → Other related features
Additional microservices...
Service
Database
Eventual consistency between microservices based on event-driven async communication

Using RabbitMQ

```
ConnectionFactory connectionFactory = new ConnectionFactory();

connectionFactory.Port = 5672;
connectionFactory.HostName = "localhost";
connectionFactory.UserName = "accountant";
connectionFactory.Password = "accountant";
connectionFactory.VirtualHost = "accounting";

IConnection connection = connectionFactory.CreateConnection();
IModel channel = connection.CreateModel();

channel.ExchangeDeclare("mycompany.fanout.exchange", ExchangeType.Fanout, true, false, null);
channel.QueueDeclare("mycompany.queues.management", true, false, false, null);
channel.QueueBind("mycompany.queues.management", "mycompany.fanout.exchange", "");

IBasicProperties properties = channel.CreateBasicProperties();
properties.Persistent = true;
properties.ContentType = "text/plain";
PublicationAddress address = new PublicationAddress(ExchangeType.Fanout, "mycompany.fanout.exchange", "");
channel.BasicPublish(address, properties, Encoding.UTF8.GetBytes("A new huge order has just come in worth $1M!!!!!"));

channel.Close();
connection.Close();
Console.WriteLine(string.Concat("Channel is closed: ", channel.IsClosed));

Console.WriteLine("Main done...");
```

Using RabbitMQ

```csharp
ConnectionFactory connectionFactory = new ConnectionFactory();

connectionFactory.Port = 5672;
connectionFactory.HostName = "localhost";
connectionFactory.UserName = "management";
connectionFactory.Password = "management";
connectionFactory.VirtualHost = "management";

IConnection connection = connectionFactory.CreateConnection();
IModel channel = connection.CreateModel();
channel.BasicQos(0, 1, false);
EventingBasicConsumer eventingBasicConsumer = new EventingBasicConsumer(channel);

eventingBasicConsumer.Received += (sender, basicDeliveryEventArgs) =>
{
    IBasicProperties basicProperties = basicDeliveryEventArgs.BasicProperties;
    string message = Encoding.UTF8.GetString(basicDeliveryEventArgs.Body);
    Console.WriteLine(string.Concat("Message received by the accounting consumer: ", message));
    channel.BasicAck(basicDeliveryEventArgs.DeliveryTag, false);
};

channel.BasicConsume("mycompany.queues.management", false, eventingBasicConsumer);
```

Using Messaging

Using messages we are doing workload by an async call that is not waiting for the response of an external process (RPC).

Using other Messaging providers like Azure Service Bus we have others features

Like:

- Automatic duplicated detection.
- Transactions.
- Pub/Sub scenarios.
- Sessions / Ordered messages.

CQRS + DDD

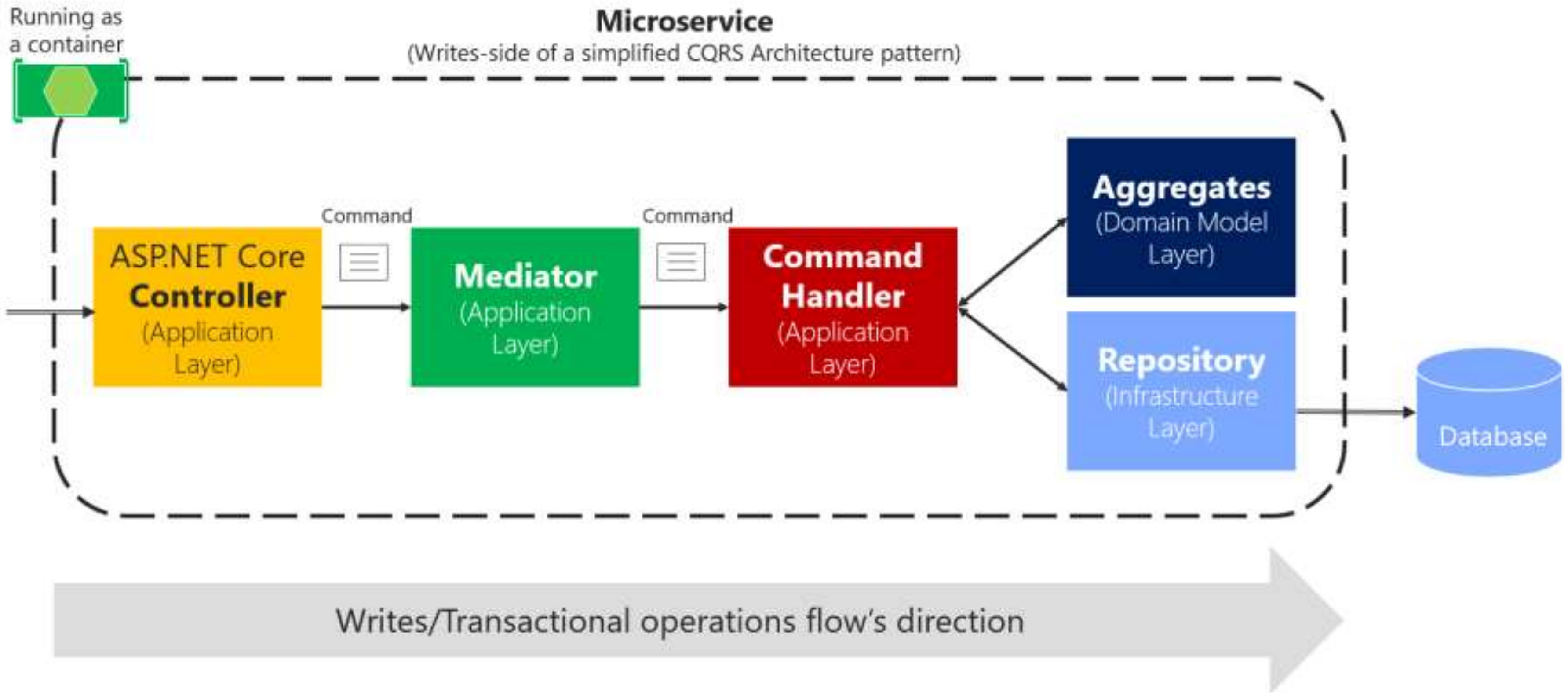
- Not all microservices need same level of complexity but for those who are not trivial, and the business is complex enough, CQRS + DDD is the perfect combination
- DDD let us focus on bounded context, perfect to structure our microservices
- CQRS let us divide Command and Event models, perfect to use messaging, rcp calls.

Drive Domain Design

- Being Aligned with the business' model, strategies, and processes.
- Being isolated from other domains and layers in the business.
- A Common Set of Terms and Definitions Used by the Entire Team
- Keeping Track Is Made Easier
- Better Code
- Agility Is a Standard
- Get a Good Software Architecture
- Stay Focused on the Solution
- Purely Flexible

MediatR

Is the most popular framework to implement domain events implementations under CQRS fashion.

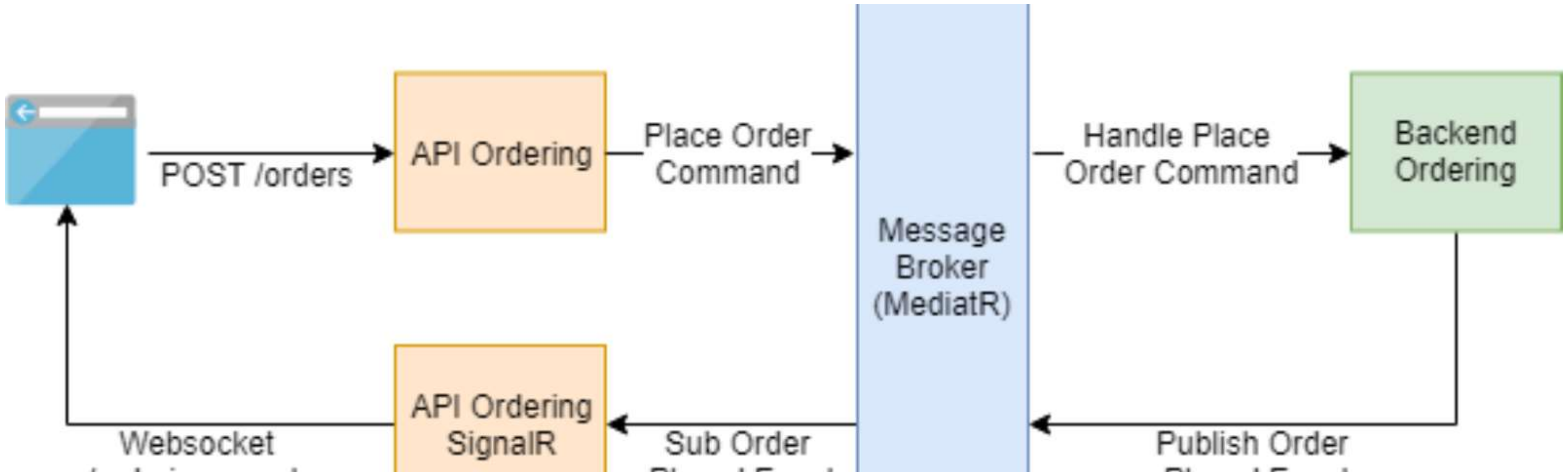

Running as a container
Microservice
(Writes-side of a simplified CQRS Architecture pattern)
Command
Command
ASP.NET Core Controller (Application Layer)
Mediator (Application Layer)
Command Handler (Application Layer)
Aggregates (Domain Model Layer)
Repository (Infrastructure Layer)
Database
Writes/Transactional operations flow's direction

DEV ROS Meetup 1° EDICION

MediatR

MediatR acting as an internal process message broker has two kinds of messages it dispatches:

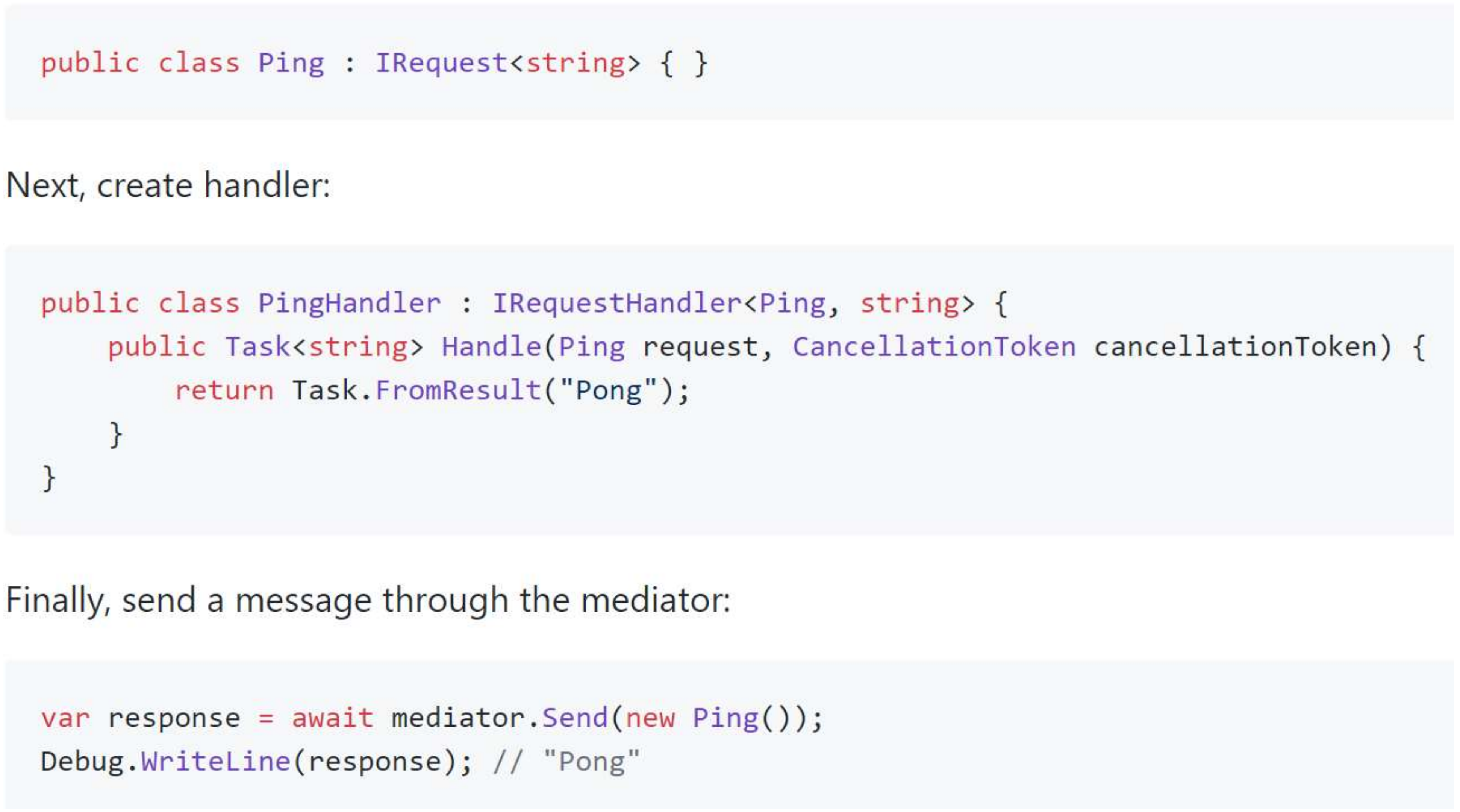
- Request/response messages, dispatched to a single handler
- Notification messages, dispatched to multiple handlers



DEV ROS Meetup 1° EDICION

MediatR

- Request/response messages, dispatched to a single handler



```
public class Ping : IRequest<string> { }
```

Next, create handler:

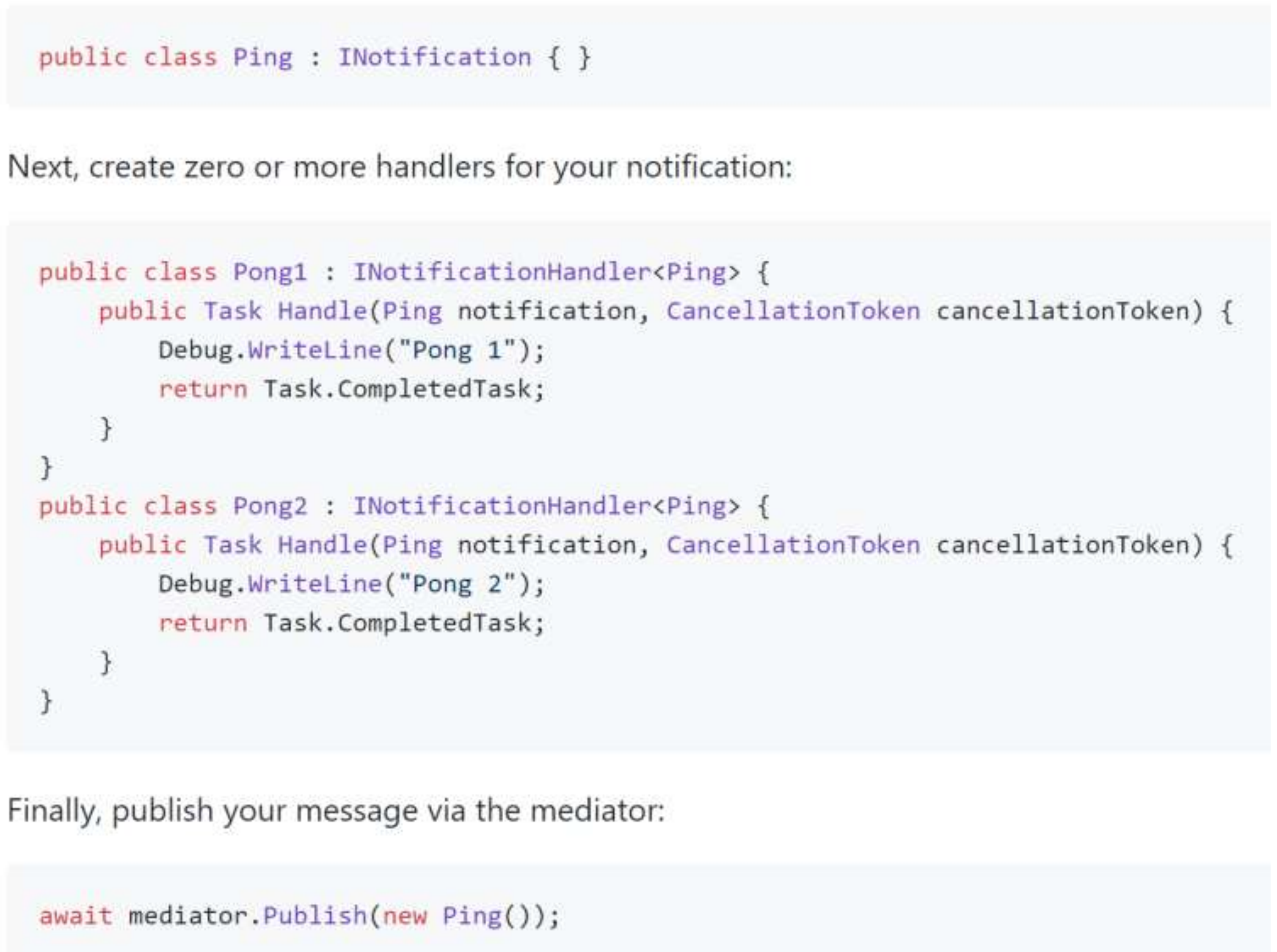
```
public class PingHandler : IRequestHandler<Ping, string> {
    public Task<string> Handle(Ping request, CancellationToken cancellationToken) {
        return Task.FromResult("Pong");
    }
}
```

Finally, send a message through the mediator:

```
var response = await mediator.Send(new Ping());
Debug.WriteLine(response); // "Pong"
```

MediatR

- Notification messages, dispatched to multiple handlers



```
public class Ping : INotification { }
```

Next, create zero or more handlers for your notification:

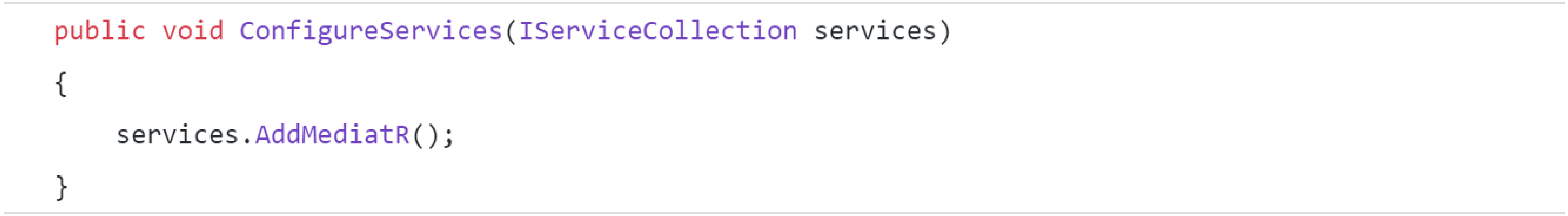
```
public class Pong1 : INotificationHandler<Ping> {
    public Task Handle(Ping notification, CancellationToken cancellationToken) {
        Debug.WriteLine("Pong 1");
        return Task.CompletedTask;
    }
}
public class Pong2 : INotificationHandler<Ping> {
    public Task Handle(Ping notification, CancellationToken cancellationToken) {
        Debug.WriteLine("Pong 2");
        return Task.CompletedTask;
    }
}
```

Finally, publish your message via the mediator:

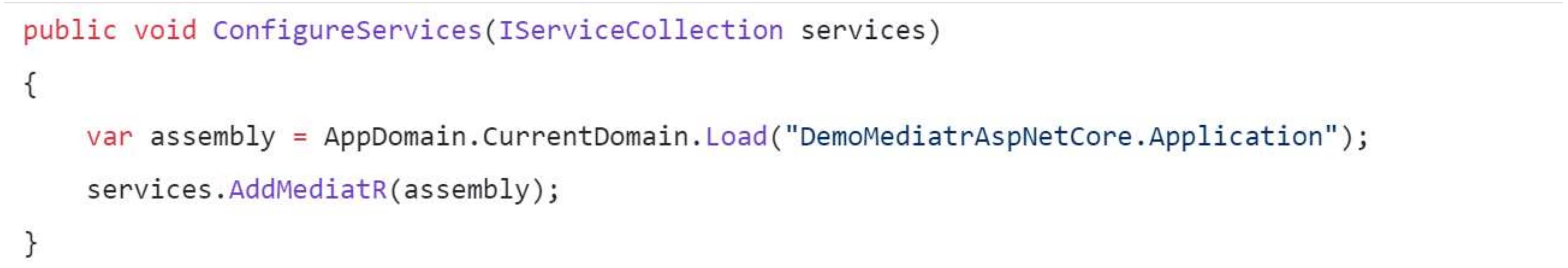
```
await mediator.Publish(new Ping());
```

MediatR Installing in ASP.NET Core

dotnet add package MediatR
dotnet add package MediatR.Extensions.Microsoft.DependencyInjection

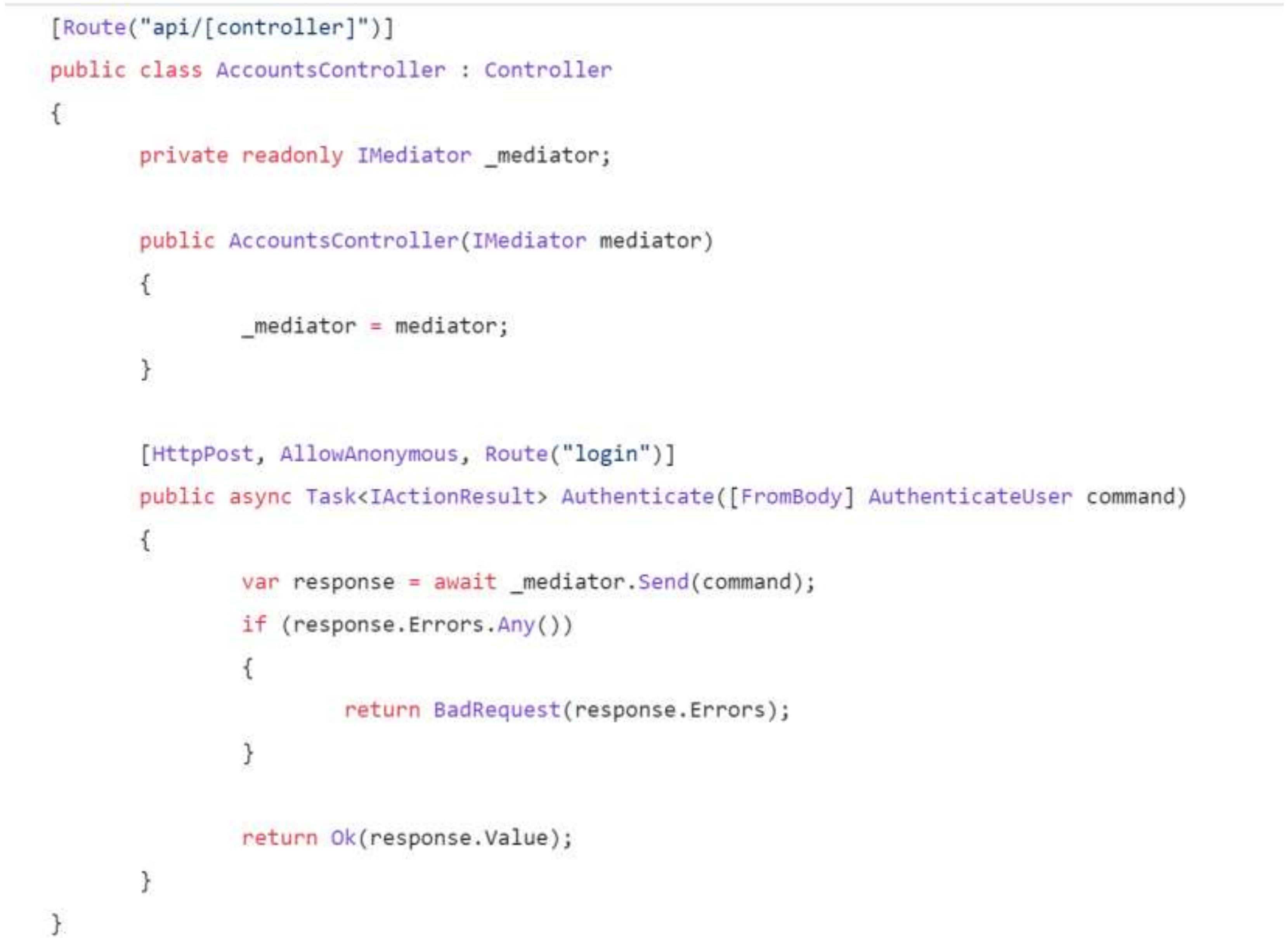


```csharp
public void ConfigureServices(IServiceCollection services)
{
    services.AddMediatR();
}
```



```csharp
public void ConfigureServices(IServiceCollection services)
{
    var assembly = AppDomain.CurrentDomain.Load("DemoMediatrAspNetCore.Application");
    services.AddMediatR(assembly);
}
```

That’s it start using MediatR and IRequests Handler



```
[Route("api/[controller]")]
public class AccountsController : Controller
{
        private readonly IMediator _mediator;

        public AccountsController(IMediator mediator)
        {
                _mediator = mediator;
        }

        [HttpPost, AllowAnonymous, Route("login")]
        public async Task<IActionResult> Authenticate([FromBody] AuthenticateUser command)
        {
                var response = await _mediator.Send(command);
                if (response.Errors.Any())
                {
                        return BadRequest(response.Errors);
                }

                return Ok(response.Value);
        }
}
```

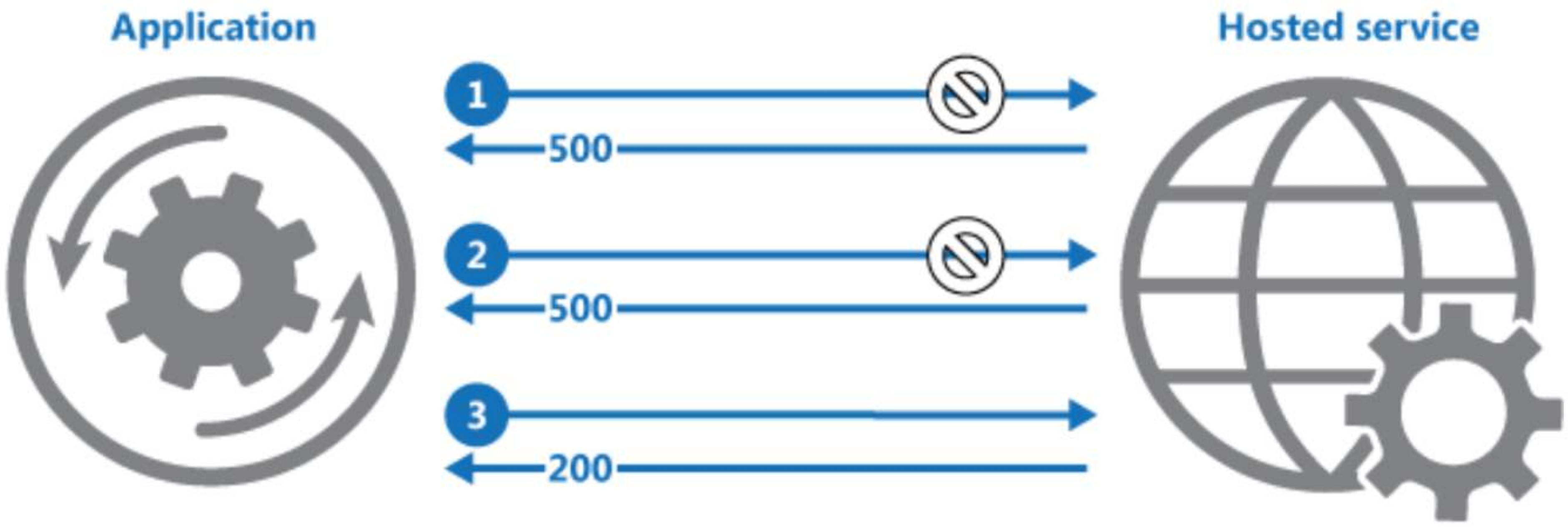
MediatR + Messaging

Combining MediatR + Async Messaging is an excellent choise

For an Event Driven solution that works with high throughput

for high Work Load without data lost in a microservices architecture.

Retry Strategy



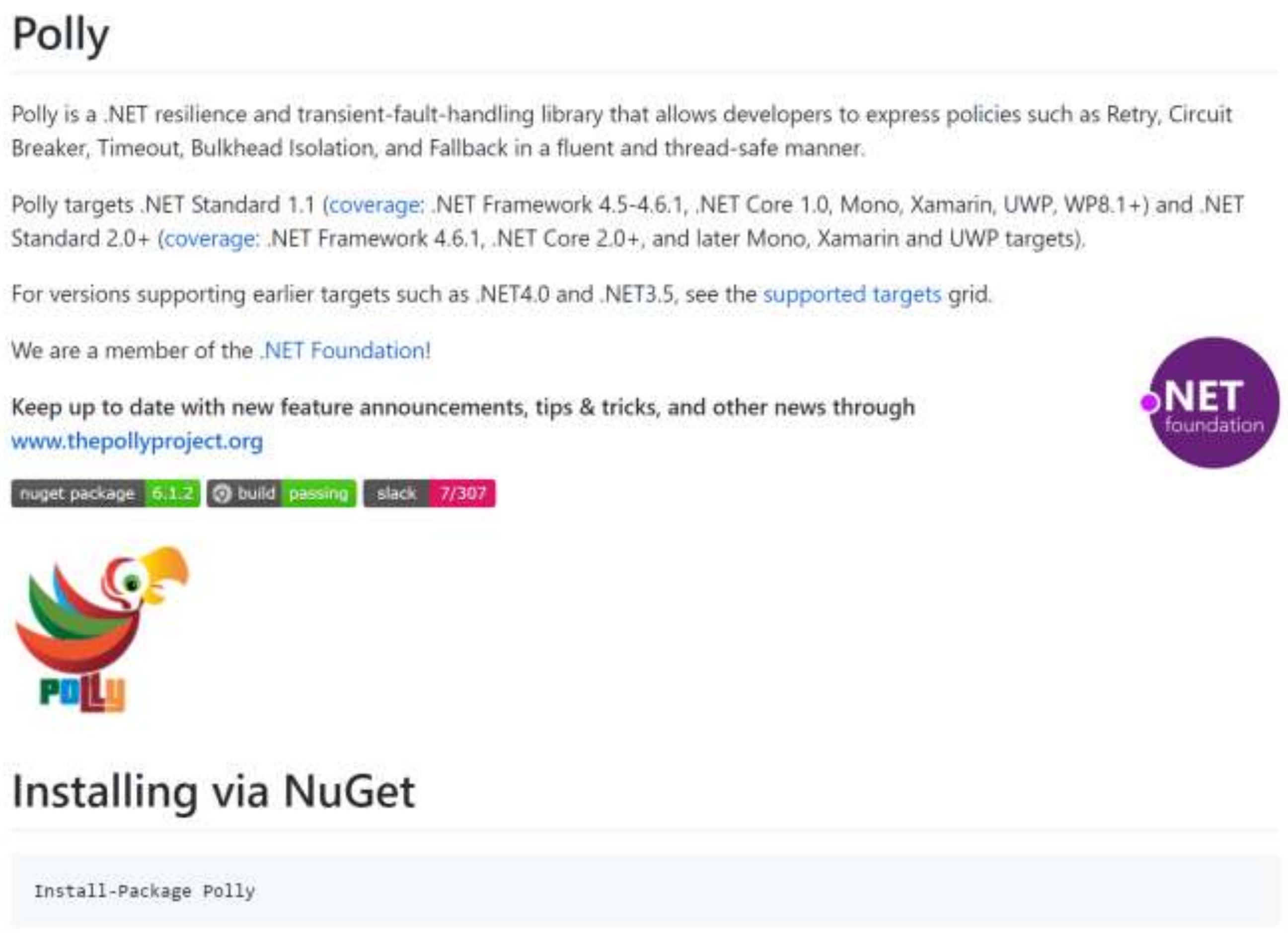
1: Application invokes operation on hosted service. The request fails, and the service host responds with HTTP response code 500 (internal server error).
2: Application waits for a short interval and tries again. The request still fails with HTTP response code 500.
3: Application waits for a longer interval and tries again. The request succeeds with HTTP response code 200 (OK).

Retry Strategy

As we can deal with multiple distributed nodes things can fail and **WILL** fail.

Retry is an important deal to consider in a Microservices architecture.

- Retry Pattern
- Circuit Breaker
- Exponential Retry.



Polly

Polly is a .NET resilience and transient-fault-handling library that allows developers to express policies such as Retry, Circuit Breaker, Timeout, Bulkhead Isolation, and Fallback in a fluent and thread-safe manner.

Polly targets .NET Standard 1.1 (coverage: .NET Framework 4.5-4.6.1, .NET Core 1.0, Mono, Xamarin, UWP, WP8.1+) and .NET Standard 2.0+ (coverage: .NET Framework 4.6.1, .NET Core 2.0+, and later Mono, Xamarin and UWP targets).

For versions supporting earlier targets such as .NET4.0 and .NET3.5, see the supported targets grid.

We are a member of the .NET Foundation!

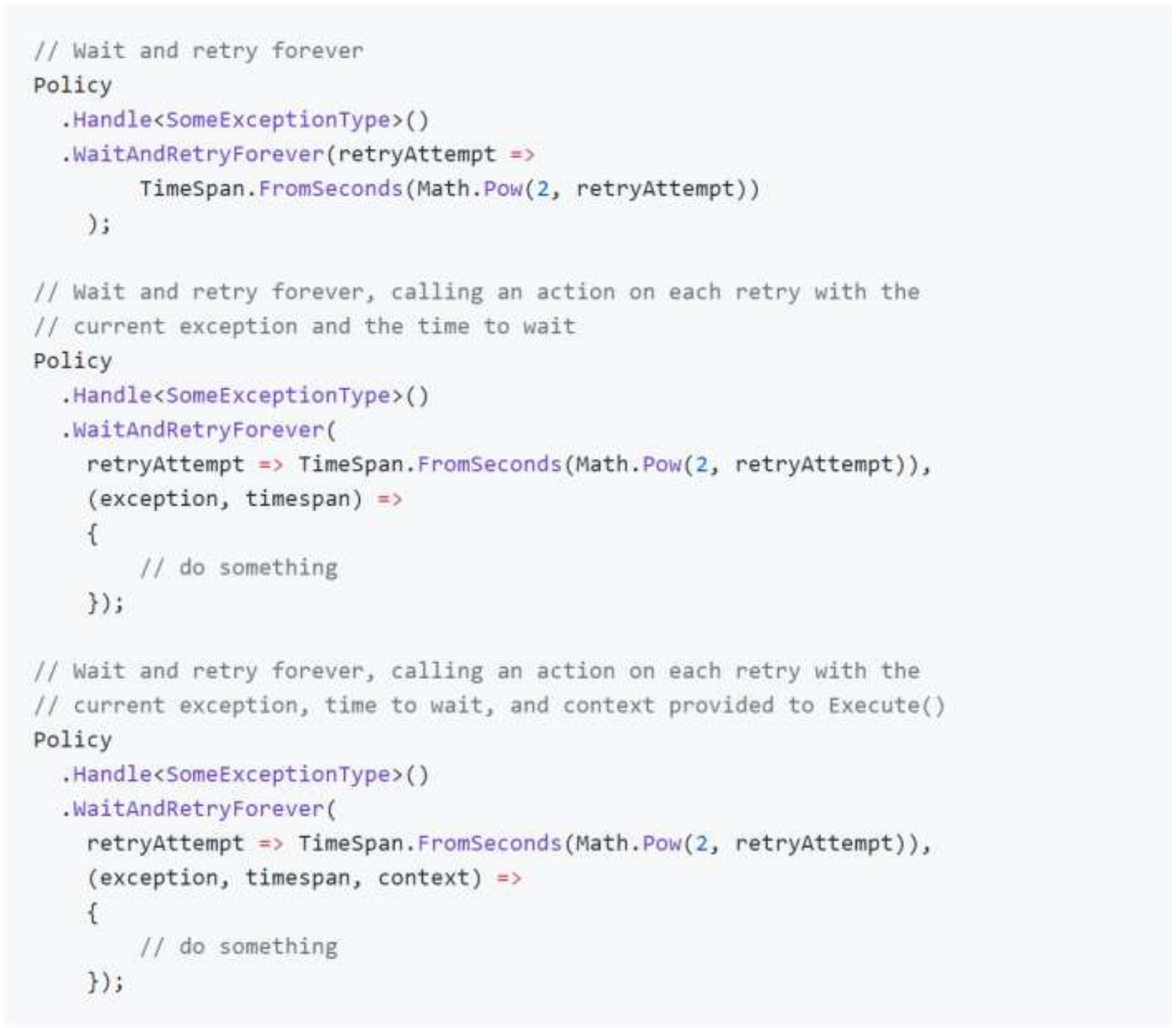
Keep up to date with new feature announcements, tips & tricks, and other news through www.thepollyproject.org

nuget package 6.1.2 | build passing | slack 7/307

Installing via NuGet

```
Install-Package Polly
```

Polly



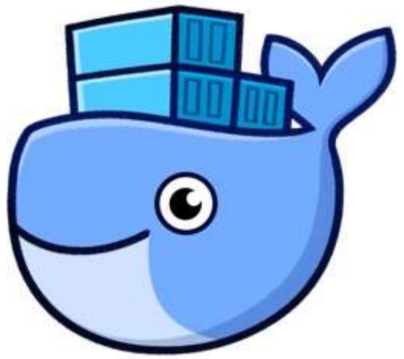
```
// Wait and retry forever
Policy
  .Handle<SomeExceptionType>()
  .WaitAndRetryForever(retryAttempt =>
        TimeSpan.FromSeconds(Math.Pow(2, retryAttempt))
    );

// Wait and retry forever, calling an action on each retry with the
// current exception and the time to wait
Policy
  .Handle<SomeExceptionType>()
  .WaitAndRetryForever(
    retryAttempt => TimeSpan.FromSeconds(Math.Pow(2, retryAttempt)),
    (exception, timespan) =>
    {
        // do something
    });

// Wait and retry forever, calling an action on each retry with the
// current exception, time to wait, and context provided to Execute()
Policy
  .Handle<SomeExceptionType>()
  .WaitAndRetryForever(
    retryAttempt => TimeSpan.FromSeconds(Math.Pow(2, retryAttempt)),
    (exception, timespan, context) =>
    {
        // do something
    });
```

Containers

Contrary to how VMs work, with Docker we don’t need to constantly set up clean environments in the hopes of avoiding conflicts.

With Docker, we know that there will be no conflicts. Docker guarantees that application microservices will run in their own environments that are completely separate from the operating system.

Docker becomes the facto technology for applications containerization.



DEV ROS Meetup 1° EDICION

Containers

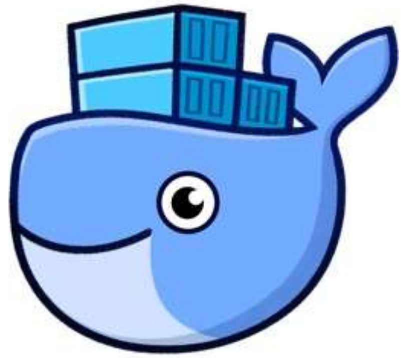
Docker allow us to configure the environment of an app an include all the OS dependencies and configuration insolated of the OS it runs.

With docker we can run containers dependencies for our nodes (Microservices) create the whole environment where they run.

Also configure the required software like Databases, Rabbit, otc.

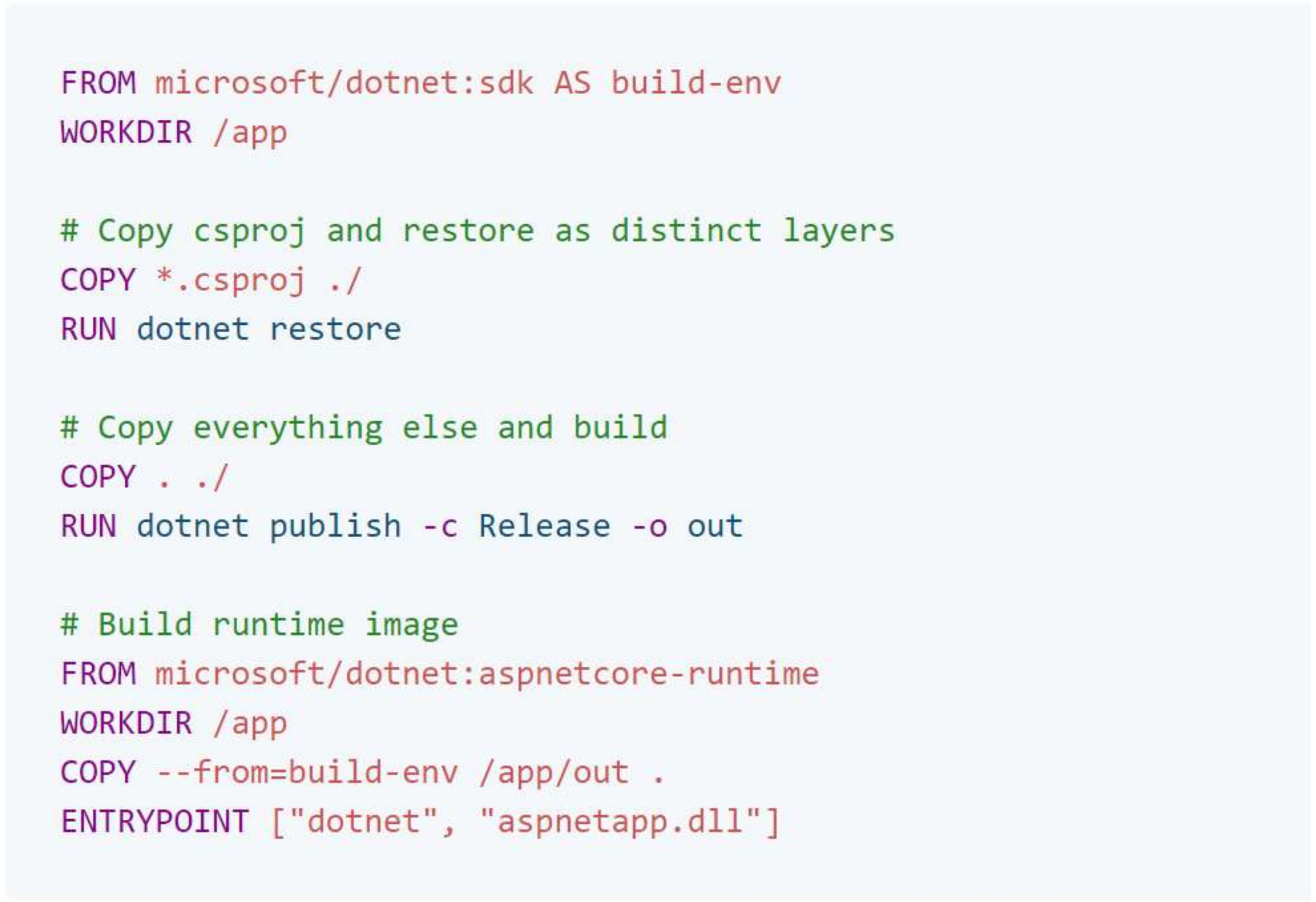
There are tons of images in Docker Hub

- Rabbit
- Redis
- SQL Server



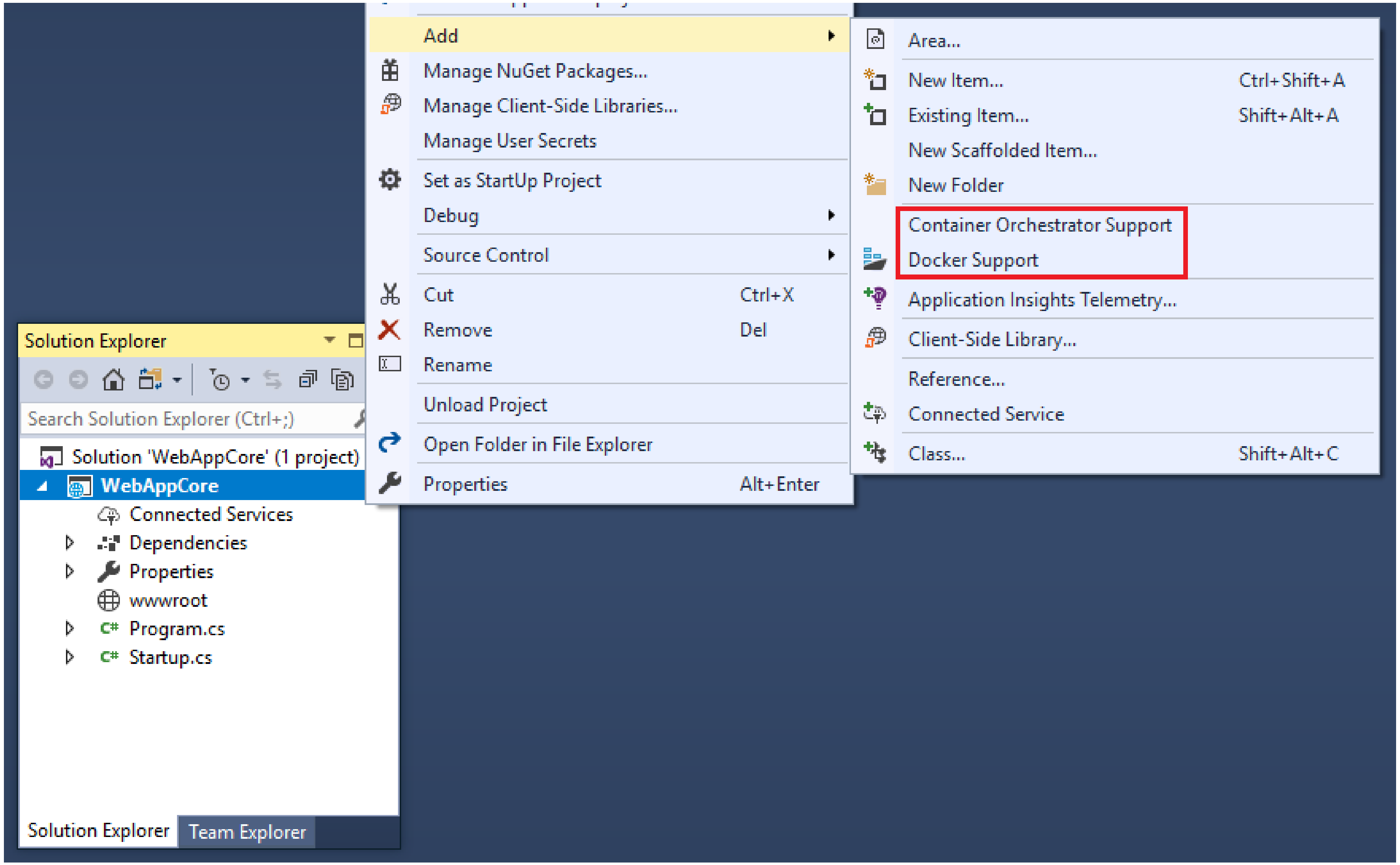
Docker ASP.NET Core Configuration.

Just a file



```
FROM microsoft/dotnet:sdk AS build-env
WORKDIR /app

# Copy csproj and restore as distinct layers
COPY *.csproj ./
RUN dotnet restore

# Copy everything else and build
COPY . ./
RUN dotnet publish -c Release -o out

# Build runtime image
FROM microsoft/dotnet:aspnetcore-runtime
WORKDIR /app
COPY --from=build-env /app/out .
ENTRYPOINT ["dotnet", "aspnetapp.dll"]
```

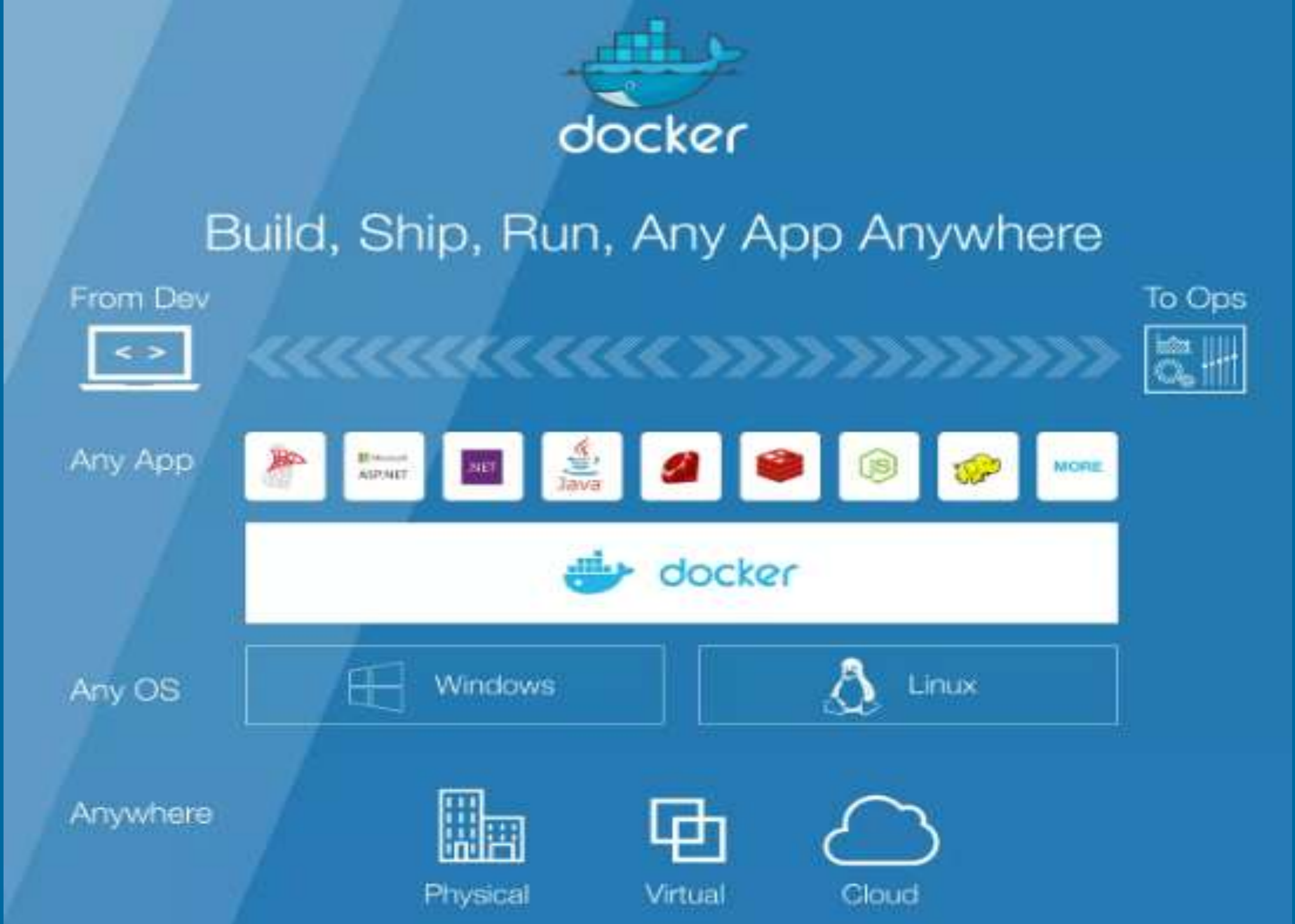
Docker Visual Studio integration.



Add
Manage NuGet Packages...
Manage Client-Side Libraries...
Manage User Secrets
Set as StartUp Project
Debug
Source Control
Cut Ctrl+X
Remove Del
Rename
Unload Project
Open Folder in File Explorer
Properties Alt+Enter

Area...
New Item... Ctrl+Shift+A
Existing Item... Shift+Alt+A
New Scaffolded Item...
New Folder
Container Orchestrator Support
Docker Support
Application Insights Telemetry...
Client-Side Library...
Reference...
Connected Service
Class... Shift+Alt+C

Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'WebAppCore' (1 project)
WebAppCore
Connected Services
Dependencies
Properties
wwwroot
Program.cs
Startup.cs
Solution Explorer
Team Explorer



DEV ROS Meetup 1° EDICION

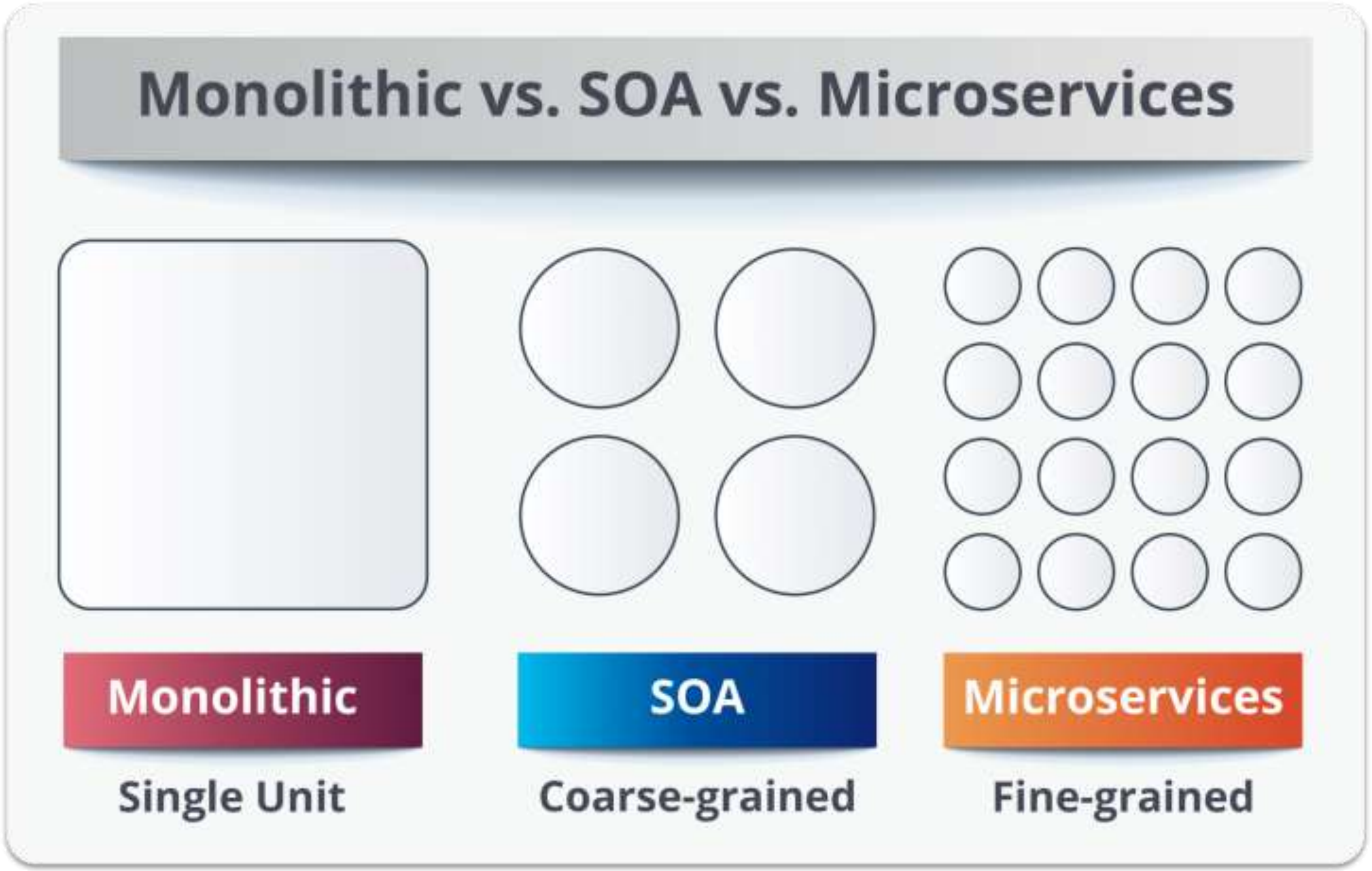
Docker Benefit

- Faster start time. A Docker container starts in a matter of seconds because a container is just an operating system process. A VM with a complete OS can take minutes to load.
- Faster deployment. There's no need to set up a new environment; with Docker, web development team members only need to download a Docker image to run it on a different server.
- Easier management and scaling of containers, as you can destroy and run containers faster than you can destroy and run virtual machines.
- Better usage of computing resources as you can run more containers than virtual machines on a single server.
- Support for various operating systems: you can get Docker for Windows, Mac, Debian, and other OSs.

Conclusion.

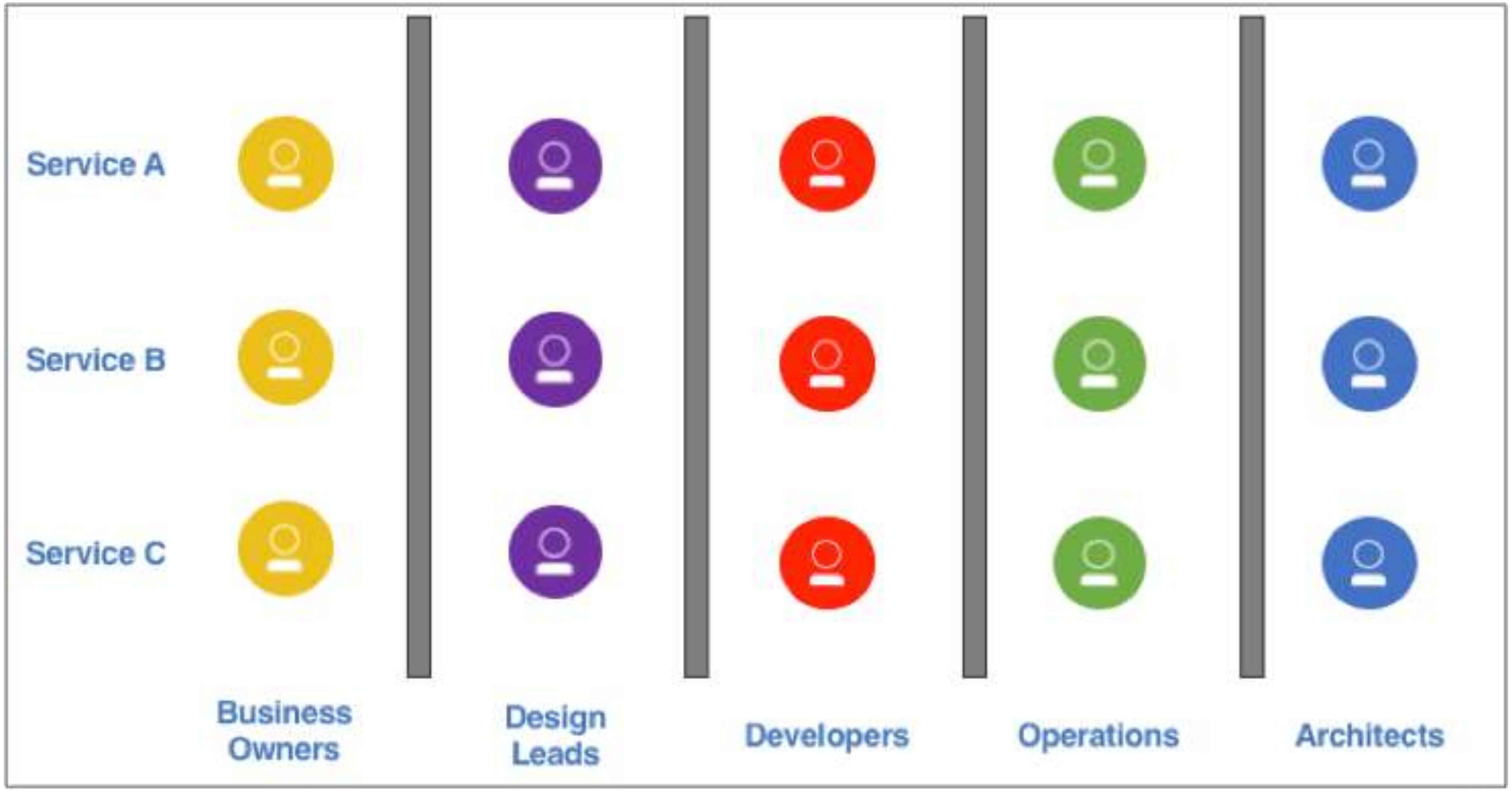
Microservices is an evolution of SOA, actually is not something new.

It’s a more aggressive solution that require the complete team ready to implement it. Implementing MS is not just a technical decision.

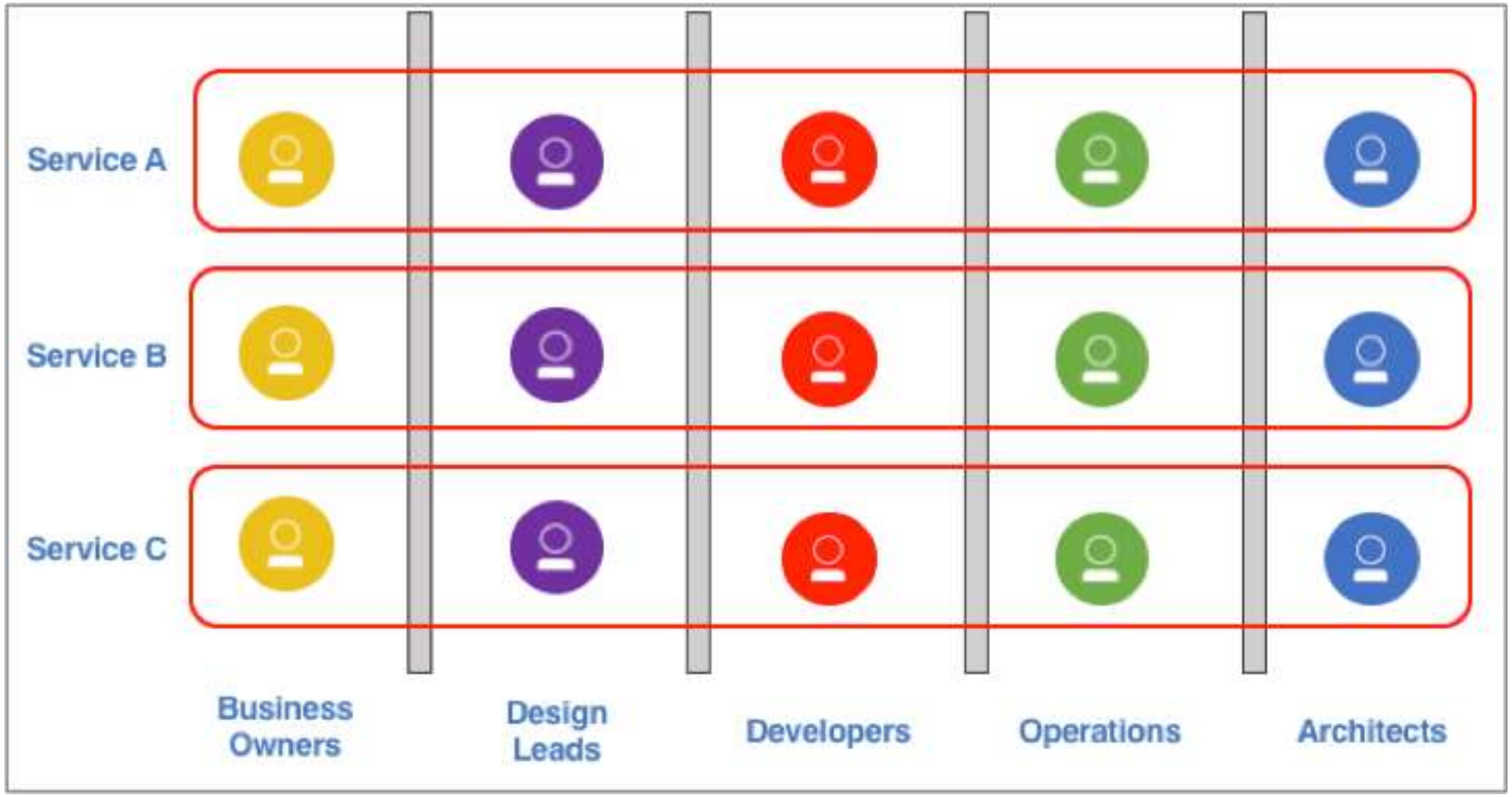


DEV ROS Meetup 1° EDICION

Conclusion.


Service A
Service B
Service C
Business Owners
Design Leads
Developers
Operations
Architects

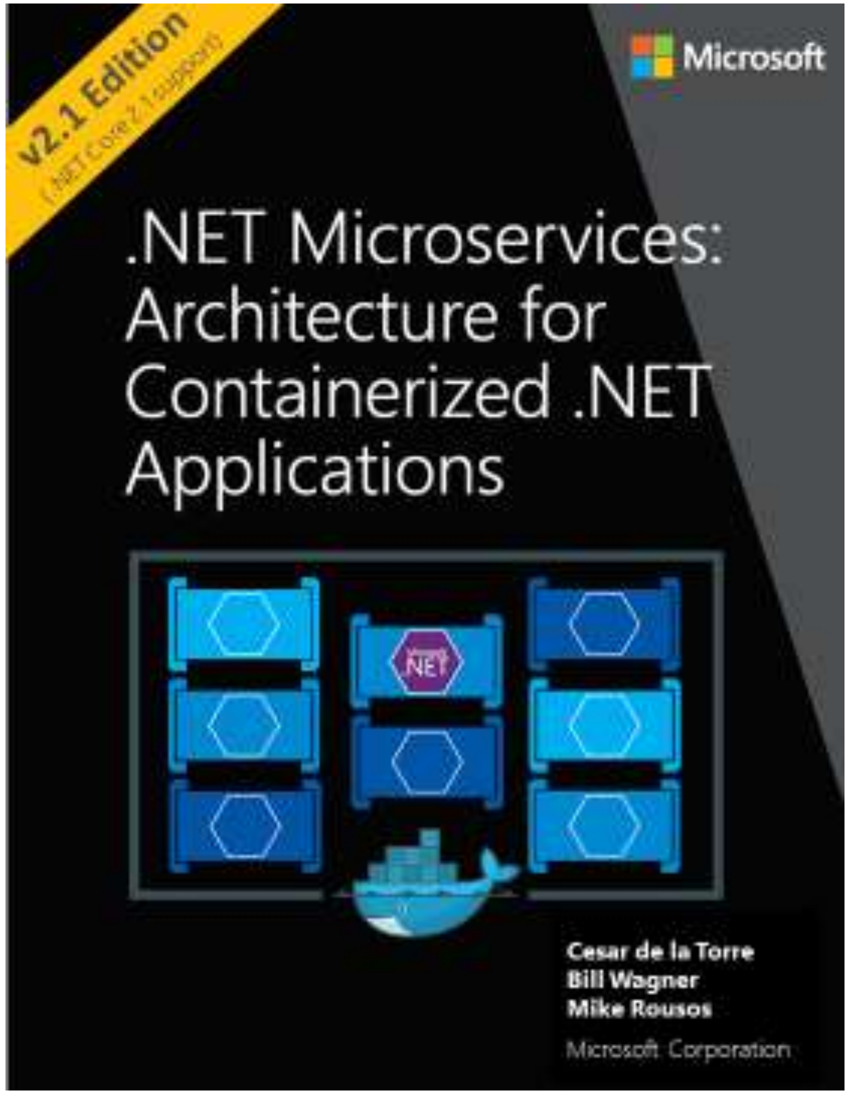
Conclusion.



You Build it, Test it, Deploy it.

More Information.

https://dotnet.microsoft.com/download/thank-you/microservices-architecture-ebook



DEV ROS Meetup 1° EDICION

DEV ROS
Meetup
1° EDICION

¡MUCHAS GRACIAS!

Follow me:

@deirevoledo

davidrevoledo@d-genix.com