

آموزش میکروسرویس

فهرست

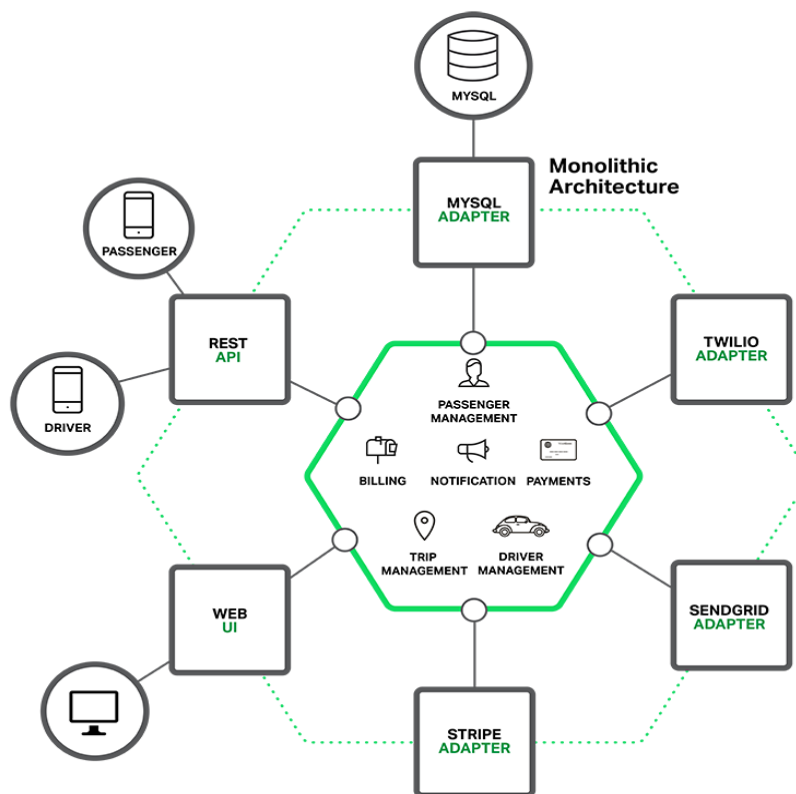
۲.....	<u>قسمت اول</u>
۴.....	<u>قسمت دوم</u>
۵.....	<u>قسمت سوم</u>
۶.....	<u>قسمت چهارم</u>
۸.....	<u>قسمت پنجم</u>
۹.....	<u>قسمت ششم</u>
۱۱.....	<u>قسمت هفتم</u>
۱۲.....	<u>قسمت هشتم</u>
۱۳.....	<u>قسمت نهم</u>
۱۶.....	<u>قسمت دهم</u>
۱۸.....	<u>قسمت یازدهم</u>
۱۹.....	<u>قسمت دوازدهم</u>
۲۰.....	<u>قسمت سیزدهم</u>

قسمت اول:

معماری میکروسرویس:

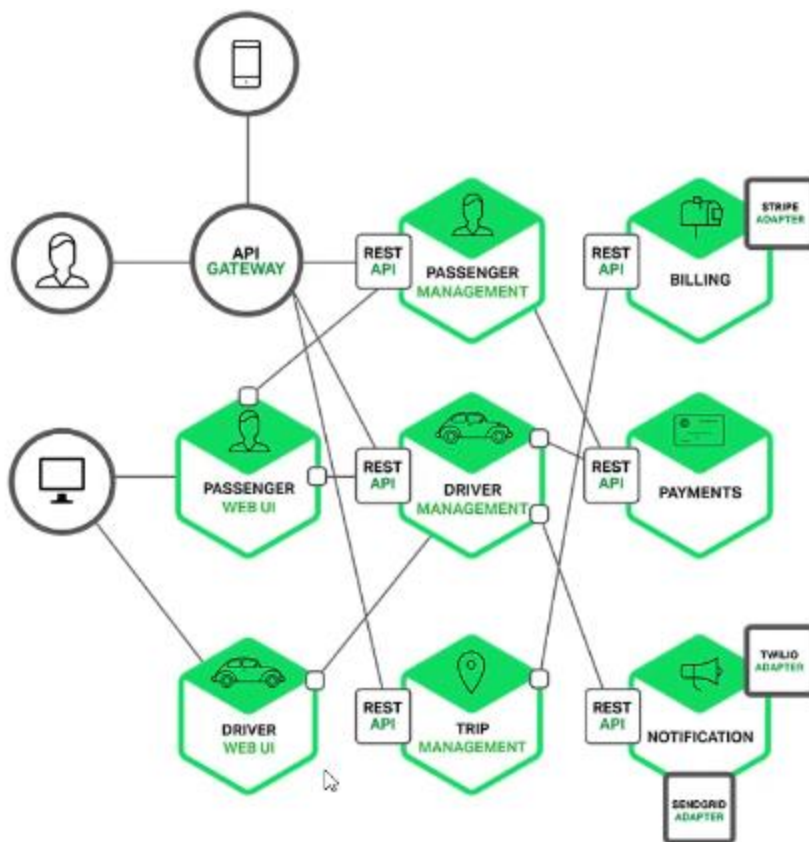
معماری میکروسرویس به شما اجازه می‌دهد اپلیکیشن‌های خود را و یا پروژه خود را به اپلیکیشن‌های کوچک (نه داده‌های کوچکتر) و سرویس‌های کوچکتر تقسیم کنید و بتوانید از آنها استفاده کنید. این کار بسیار مزایایی دارد. این امر باعث می‌شود که اپلیکیشن‌های شما مستقل کار کنند و می‌توانند روی سرورهای مستقل و متفاوت کار بکنند.

الگوی معماری میکروسرویس مزایای قابل توجهی دارد، به ویژه هنگامی که با توانایی توسعه سریع و تحویل برنامه‌های سازمانی پیچیده همراه می‌شود.



شکل ۱-۱ ساختار میکروسرویس

دستگاه‌ها و موبایل‌ها با GATEWAY در ارتباط هستند و سایر جزئیات سرویس‌های میکروسرویس هستند که می‌توانند پیام‌ها، احراز هویت و سایر موارد را بین خود تقسیم کنند.



شکل ۲- ۱ ساختار میکروسرویس

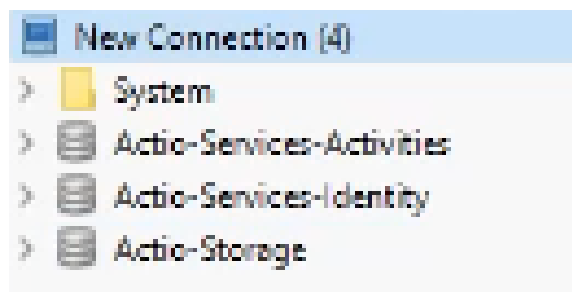
در شکل بالا ، هر کدام از مکعب ها یک سرویس می باشند و زمانی که دستگاه یا کاربر درخواستی را ارسال میکنند از GATEWAY عبور میکنند و API این درخواست را به سرویس های مربوطه به صورت PUSH کردن ارسال می کند و پس از کارکرد سرویس ها، GATEWAY دوباره API را به کاربر ارسال میکند.

پیشنیاز های میکروسرویس:

در این دوره GATEWAY ها با ASP.NET CORE پیاده سازی می شود و پیشنیاز این دوره C# ، ASP.NET CORE و WEB API می باشد.

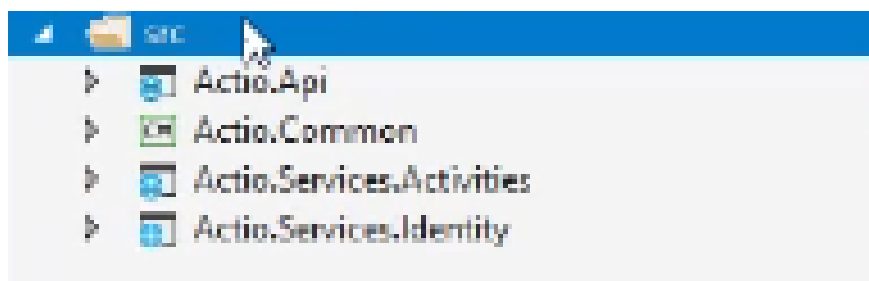
هر سرویس یک بانک اطلاعاتی نیاز دارد و بانک اطلاعاتی مورد استفاده در این دوره Mongo DB و No Sal می باشد. سرویس باس (Service Bus) این دوره که وظیفه آن کنترل کردن GATEWAY ها است، Rabbit MQ می باشد.

در این دوره از داکر (Docker) استفاده می شود و تمام سرویس ها و GATEWAY ها از داکر استفاده شده است.



شکل ۳-۱ بانک اطلاعاتی

در بانک اطلاعاتی سرویس های Activities, Identity, Storage را شامل می شود و در Solution پروژه Common لایه اصلی پروژه است و Gateway در API قرار گرفته، Activity عملیاتی است که در سیستم برنامه نویسی پیاده سازی می شود و سرویس Identity مسئولیت احراز هویت را به عهده دارد.



شکل ۴-۱ solution پروژه

برای سیستم احراز هویت از سیستم JWT (JSON Web TOKEN) استفاده می شود. کاربران از این سیستم احراز هویت می شوند و TOKEN ارسال می شود و تحویل لایه API داده می شود. این سرویس ها به طور مستقل در کنار هم کار میکنند تا بتوانند اپلیکیشن را کنترل بکنند. برای استفاده و تست سرویس ها از Postman استفاده می کنیم.

قسمت دوم:

اگر یک پروژه ساده را بخواهیم در نظر بگیریم که اپلیکیشن موبایل داشته باشد که شامل لایه

Data -> API -> Infrastructure -> Core -> presentation (View)

می باشد که این لایه ها در قالب یک پروژه عمل می کنند . تمامی فایل های DLL پروژه در لایه آخر Presentation قرار می گیرند و شروع به کار می کنند. هر چه اپلیکیشن با ترافیک بیشتر باشد، سرعت پروژه کمتر می شود و این لایه بندی در بهینه سازی و سرعت بهتر در کدنویسی به ما کمک می کند.

معماری میکروسرویس میگوید بجای یک اپلیکیشن، برنامه را به اپلیکیشن های کوچک تبدیل کرد که به صورت مستقل و با منابع مستقل اپلیکیشن ها کار کنند.

بطور مثال لایه های

- Data Service
- API Service -> Gateway
- Identity Service
- Activity Services

زمانی که کاربر درخواست سرویس API ارسال میکند، سرویس API دو پیغام به سرویس به Identity و Activity ها می دهد. این لایه ها در اپلیکیشن های متفاوت قرار دارند و حتی می توانند در سرور های متفاوت قرار بگیرند.

این به ما کمک می کند که سیستم ما یک سیستم توزیع شده باشد و نه یک پارچه. وقتی سیستم توزیع شده باشد ، منابع هم توزیع شده می باشند و می توان درخواست های زیاد را کنترل کرد. مثلا API یک درخواست به لایه Identity می دهد که آیا کاربر احراز هویت شده است و یا امکان استفاده از آن را دارد و Identity به API پاسخ را میدهد و ضمن تایید API، Activity مورد نظر فراخوان می شود.

به طور مثال یک وب سایت فروشگاهی فایل دانلودی، بیشترین بار برای دانلود ها می باشد. برای این پروژه، یک لایه Download Service داریم که وظیفه این لایه دانلود فایل ها می باشد. روند کار به این صورت است که API به Identity پیغام میدهد و تایید می شود و اجازه دانلود را به کاربر می دهد و به لایه Download می رود. اگر بار دانلودی زیاد شد، میتوان در پروژه دوتا لایه Download Service موازی با هم قرار داد که در صورت بار ترافیک زیاد، از لایه اول به لایه دوم فرستاده می شود. این دو لایه می توانند از دو سرور متفاوت نیز باشند. به طور مثال کسی که IP ایرانی داشت از سرویس یک و کسی که آلمان بود از سرویس دوم دانلود کند.

قسمت سوم:

پروژه ما باید یک دروازه ورودی داشته باشد، زمانی که درخواست (Request) به سمت برنامه ما ارسال می شود، یک دروازه وجود دارد که آن سرویس های API پروژه ما است (میتوان از آن API HTTP نیز یاد کرد).

بعد از آنکه در این دروازه وارد شد، به سرویس باس (Service bus) تحویل داده می شود که این سرویس باس ها می توانند Identity باشند و سپس به سراغ Activity Service می رود

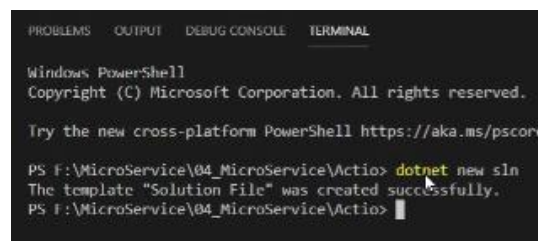
Request -> HTTP API [gateway] -> Service Bus -> Identity Service -> Activity Service
Solution پروژه ما شامل ۶ سرویس می باشد که شامل ۴ سرویس اصلی می باشد:

۱. Actio.API -> پل ورودی یا gateway: در این دوره از MongoDB [CQRS] استفاده می شود.
۲. Actio.Common -> وظیفه آن پیغام رسانی Message ها است: پیام هایی که بین سرویس ها تبادل می شود.
۳. Actio.Services.Identity -> MongoDB کار احراز هویت را انجام می دهد.
۴. Actio.Services.Activities -> Activities که نیاز داریم برای کار، بر روی SQL کار بکنند و یا روی MongoDB باشد که بتواند سرعت اطلاعات را برای بیت دیتا افزایش دهد.
- هر نوع سرویس فوق که ذکر شده می توانند به سرویس های بیشتر تکثیر شوند و معماری ما سرویس گرا می باشد.

۵. Actio.Tests -> تست های پروژه را در آن انجام دهیم.
۶. Actio.Tests.EndToEnd -> پایان سرویس های ما و تست های ما است که بتوان در این لایه تست ها را نوشت تا معماری ما تکمیل باشد.

قسمت چهارم:

یکی از امکاناتی که در این دوره می خواهیم با آن کار کنیم، محیط کدنویسی VsCode می باشد. در این برنامه می خواهیم پروژه خود را بسازیم. از طریق ترمینال دستور *dotnet new sln* را وارد میکنیم.



```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

PS F:\MicroService\04_MicroService\Actio> dotnet new sln
The template "Solution File" was created successfully.
PS F:\MicroService\04_MicroService\Actio> |
```

شکل ۱-۴ دستور ساخت پروژه

با دستور *mkdir src* یک پوشه برای پروژه های ما می سازیم.

```
PS F:\MicroService\04_MicroService\Actio> mkdir src
```

شکل ۲-۴ دستور ساخت پوشه

همچنین پوشه ای برای اسکریپت های پروژه با نام scripts و tests برای تست های پروژه می سازیم. حال میخواهیم در پوشه SRC پروژه های خود را بسازیم و با دستور `cd src` وارد پوشه SRC می شویم. اولین پروژه در آن web api می باشد که با دستور `dotnet new webapi -n Actio.Api` آن را میسازیم. عبارت `-n` برای این است که ما نام پروژه را نامگذاری کنیم و Actio.Api نام پروژه Api ما می باشد.

```
PS F:\MicroService\04_MicroService\Actio\src> dotnet new webapi -n Actio.Api
```

شکل ۳-۴ دستور ساخت پروژه Api

همچنین پروژه دیگر با نام Actio.Services.Identity و Actio.Services.Activities می سازیم. سپس می رویم به سراغ پروژه Common که تفاوتش با قبلی ها این است که دیگر این پروژه WebApi نیست که دستور ساخت آن عبارت است از: `dotnet new classlib -n Actio.Common`

```
PS F:\MicroService\04_MicroService\Actio\src> dotnet new classlib -n Actio.Common
```

شکل ۴-۴ دستور ساخت پروژه Common

اکنون در SRC خود ۴ پروژه داریم. کاری که باید بکنیم این است که پروژه های در پوشه SRC به solution متصل شوند. در آدرس پروژه از با دستور `cd..` از پوشه SRC خارج می شویم و با زدن `/s` دایرکتوری ها و پوشه های در دسترس به ما نمایش داده می شود.

Mode	LastWriteTime	Length	Name
d----	8/15/2019 12:11 PM		scripts
d----	8/15/2019 12:14 PM		src
d----	8/15/2019 12:11 PM		tests
-a----	8/15/2019 12:11 PM	540	Actio.sln

شکل ۵-۴ دایرکتوری های پروژه

عبارت Actio.sln ، Solution ، پروژه ما است. حال باید ۴ پروژه خود را یکی یکی با دستور `dotnet sln add src/...` انجام داد.

```
PS F:\MicroService\04_MicroService\Actio> dotnet sln add src/Actio.Api/Actio.Api.csproj
Project 'src\Actio.Api\Actio.Api.csproj' added to the solution.
```

شکل ۶-۴ افزودن پروژه ها به Solution

- `dotnet sln add src/Actio.Api/Actio.Api.csproj`
- `dotnet sln add src/Actio.Common/Actio.Common.csproj`
- `dotnet sln add src/Actio.Services.Identity/Actio.Services.Identity.csproj`
- `dotnet sln add src/Actio.Services.Activities/Actio.Services.Activities.csproj`

حال می بایست پروژه خود را `restore` کرد تا پکیج های پروژه دانلود شوند و دستور `dotnet restore` را وارد میکنیم.

```
PS F:\MicroService\04_MicroService\Actio> dotnet restore
Restore completed in 17.08 ms for F:\MicroService\04_MicroService\Actio\src\Actio.Api\Actio.Api.csproj.
Restore completed in 17.08 ms for F:\MicroService\04_MicroService\Actio\src\Actio.Services.Activities\Actio.Services.Activities.csproj.
Restore completed in 17.08 ms for F:\MicroService\04_MicroService\Actio\src\Actio.Services.Identity\Actio.Services.Identity.csproj.
Restore completed in 22.46 ms for F:\MicroService\04_MicroService\Actio\src\Actio.Common\Actio.Common.csproj.
```

شکل ۷-۴ `dotnet restore`

سپس باید پروژه خود را `build` کرد تا اگر ارور و مشکلی داشته باشد رفع شود و پروژه آماده شود و دستور `dotnet build` را وارد میکنیم. اگر پیغام `build succeeded` مشاهده شد نشان دهنده درست بودن `build` ما می باشد.

```
Actio.Common -> F:\MicroService\04_MicroService\Actio\src\Actio.Common\bin\Debug\netstandard2.0\Actio.Common.dll
Actio.Services.Activities -> F:\MicroService\04_MicroService\Actio\src\Actio.Services.Activities\bin\Debug\netcoreapp3.0\Actio.Services.Activities.dll
Actio.Api -> F:\MicroService\04_MicroService\Actio\src\Actio.Api\bin\Debug\netcoreapp3.0\Actio.Api.dll
Actio.Services.Identity -> F:\MicroService\04_MicroService\Actio\src\Actio.Services.Identity\bin\Debug\netcoreapp3.0\Actio.Services.Identity.dll

Build succeeded.
    0 Warning(s)
    0 Error(s)

Time Elapsed 00:00:03.52
```

شکل ۸-۴ `build succeeded`

قسمت پنجم:

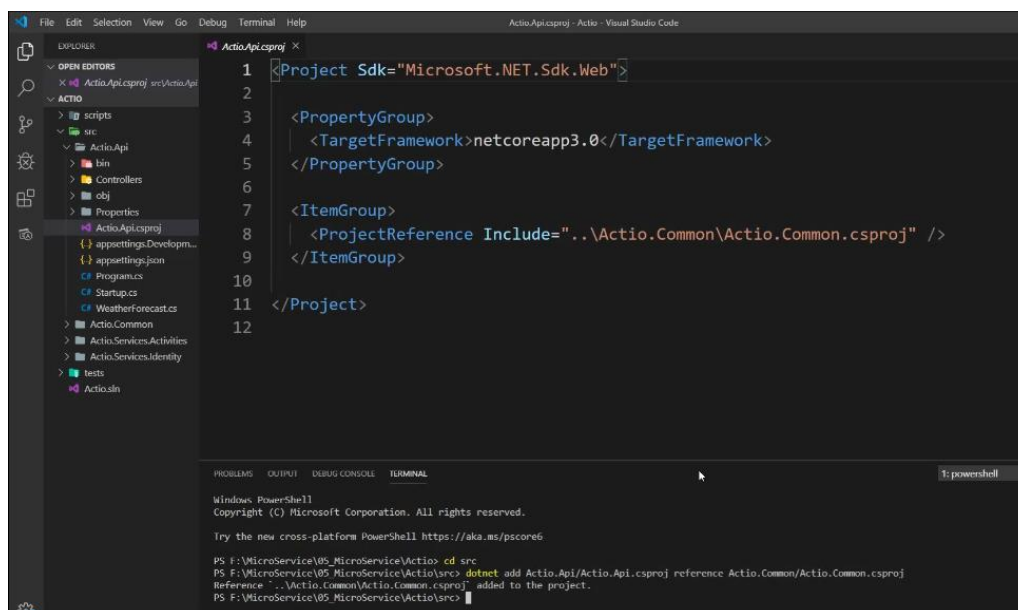
پس از اضافه کردن پروژه ها به `solution` حال باید `reference` دادن پروژه ها به یکدیگر را انجام داد. پروژه ها می بایست `reference` داشته باشند.

در ابتدا ما می بایست پروژه های `api` و `identity` و `activities` را به پروژه `common` رفرنس دهیم تا دسترسی `common` به سایر پروژه ها امکان پذیر باشد. حال باید در ترمینال وارد مسیر `src` شویم. سپس با دستور `dotnet add reference Actio.Common/Actio.Common.csproj` رفرنس می دهیم.

```
PS F:\MicroService\05_MicroService\Actio\src> dotnet add Actio.Api/Actio.Api.csproj reference Actio.Common/Actio.Common.csproj
```

شکل ۵-۱ رفرنس دادن

همچنین پس از رفرنس دادن به پروژه Common، در فایل csproj پروژه Api تگ رفرنس در آن نوشته می شود.



شکل ۵-۲ نتیجه رفرنس

- dotnet add Actio.Api/Actio.Api.csproj reference Actio.Common/Actio.Common.csproj
- dotnet add Actio.Services.Identity/Actio.Services.Identity.csproj reference Actio.Common/Actio.Common.csproj
- dotnet add Actio.Services.Activities/Actio.Services.Activities.csproj reference Actio.Common/Actio.Common.csproj

پس از رفرنس دادن تمام پروژه ها به پروژه Common حال با زدن `cd ..` به مسیر اصلی پروژه رفته و `build` را انجام می دهیم.

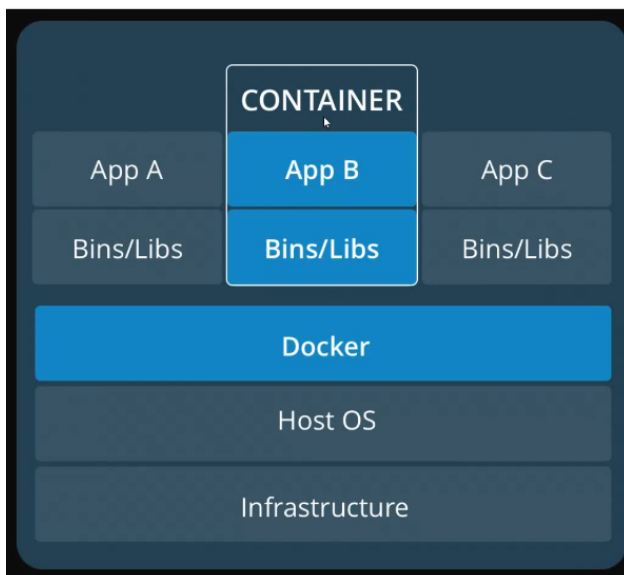
قسمت ششم:

در این دوره ما میخواهیم از داکر (Docker) استفاده بکنیم. داکر یک پلتفرم مبنی بر لینوکس است که مشابه ویرچوال باکس (Virtual Box) یا ماشین مجازی بر روی سیستم می باشد. داکر به ما این قابلیت را میدهد که با Container هایی که به ما ارائه می دهد، آن را بخش بندی کنیم.

بطور مثال سیستم عامل ما ویندوز می باشد و اگر بخواهیم php کدنویسی کنیم ، نیازمند نصب php, Mysql, xamp و سایر ابزار های مورد نیاز میباشد. اکنون ما بجای آنکه ویندوز خود را درگیر نصب این ابزار ها کنیم، به یک محیط ایزوله روی می آوریم که هرگونه رویداد و برنامه را در همان محیط انجام داد و به سیستم عامل ما کاری و لطمه ای نشده باشد. در این موارد پای ماشین های مجازی نیز به وسط کشیده می شود که به

ویرچوال باکس یا ماشین مجازی باید نصب شود، اما ماشین مجازی یا ویرچوال باکس برای سیستم ما سنگین می باشد.

داکر یک پلتفرم متن باز (Open Source) مبتنی بر سیستم عامل لینوکس می باشد که روی سیستم عامل های مطرح مک، لینوکس، ویندوز اجرا می شود. داکر کارکردش به صورت بخش بندی می باشد، بخش بندی به ۳ قسمت از container تشکیل می شود (App A-App B-App C) ، یعنی به سه بخش ایزوله و موازی از هم کار میکنند.



شکل ۶-۱ بخش بندی داکر

برای کار کردن با داکر ، کافیت آنرا نصب کنیم. برای صحت از نصب داکر می بایست WindowsPowershell را اجرا و دستور `docker --version` را وارد کرد.

```
PS C:\Users\Inan> docker --version
Docker version 19.03.1, build 74b1e89
```

شکل ۶-۲ صحت نصب داکر

داکر می تواند image هایی را دریافت کند و نصب کند، مانند image اپلیکیشن پروژه خود میتوانیم بر روی آن استفاده کنیم. بطور مثال برای تست hello world داکر، در پاورشیل خود دستور `docker pull hello-world` را وارد می کنیم.

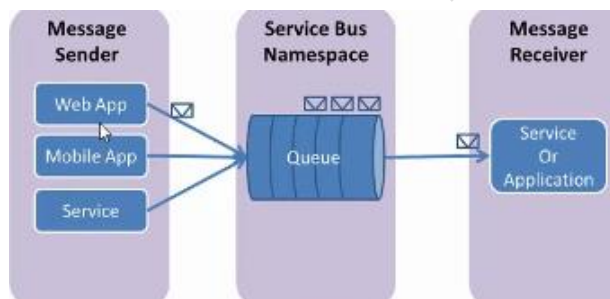
```
PS C:\Users\Iman> docker pull hello-world
Using default tag: latest
latest: Pulling from library/hello-world
1b930d010525: Pull complete
Digest: sha256:451ce787d12369c5df2a32c85e5a83d52cbcef6cb3586dd83075f3034f10adedd
Status: Downloaded newer image for hello-world:latest
docker.io/library/hello-world:latest
```

شکل ۶-۳ image hello world

سپس برای خروجی image باید دستور `docker run hello-world` را وارد می کنیم. همچنین برای دریافت image هایی که روی سیستم داریم، می توانیم دستور `docker image ls` را وارد کنیم. برای دریافت لیست دستورات داکر و جهت کمک گرفتن از آن، دستور `docker --help` را می توان استفاده کرد.

قسمت هفتم:

سرویس باس محلی است پیام های ما در آنجا مدیریت می شود. در واقع سرویس باس یک لایه میانی جهت پیام های پروژه ها باهم است. پروژه های ما به طور مستقیم نمی توانند باهم در ارتباط پیامی باشند و از این رو سرویس باس است که این ارتباط پیامی را کنترل می کند.



شکل ۷-۱ سرویس باس

اکنون برای انتخاب سرویس باس خود، یکی از محبوب ترین سرویس باس ها به نام RabbitMQ انتخاب می کنیم. در انتخاب گزینه های نصب ریت، بر روی گزینه Docker Image کلیک میکنیم.



شکل ۷-۲ نصب ریت

سپس دستوری که در داکر قرار دارد را کپی کرده و در powershell پیست و اجرا می کنیم.

`docker pull rabbitmq`

اکنون می بایست RabbitMQ را بر روی داکر اجرا کنیم. دستور زیر را وارد میکنیم.

```
docker run -p 5672:5672 rabbitmq
```

اکنون ریبیت را با موفقیت نصب کرده ایم.

قسمت هشتم:

در این قسمت به سراغ کدنویسی می رویم و در این راستا چند افزونه برای VSCode معرفی می شوند.

- C#
- C#Extensions
- C#Snippets

حال باید دستورات را شروع به نوشتن کنیم. وارد پوشه src می شویم و لایه Actio.Common را انتخاب می کنیم. این لایه ارتباط ما بقی لایه ها را برقرار می کند. در این لایه یک پوشه جدید به نام Commands ایجاد می کنیم. این پوشه برای دستورات ما است که به مرور زمان تکمیل می شود.

در این پوشه یک فایل interface ایجاد می کنیم و نام آن را ICommand نامگذاری می کنیم. تمامی Command هایی که در پروژه داریم از این Interface باید پیروی کنند.

سپس یک فایل class دیگر در این پوشه می سازیم و نام آنرا CreateUser نام گذاری می کنیم. این کلاس اطلاعات مربوط به یوزرهای خود را ارائه می دهد. این کلاس از ICommand می بایست ارث بری کند.

```
public class CreateUser : ICommand
```

با استفاده از کلمه prop فیلد های مورد نیاز در این کلاس را می نویسیم.

```
public string Email { get; set; }  
public string Password { get; set; }  
public string Name { get; set; }
```

ثبت نام ما از طریق فیلد های فوق انجام می شود.

حال ما نیاز داریم که Activity های خود را ایجاد کنیم. در این پوشه یک فایل class به نام CreateActivity ایجاد می کنیم. سپس یک interface دیگر به نام IAuthenticatedCommand ایجاد می کنیم. این interface برای شناسایی Command ها مورد استفاده قرار می گیرد و از ICommand نیز ارث بری میکند.

```
public interface IAuthenticatedCommand : ICommand
```

کلاس CreateActivity از این interface ارث بری می کند.

```
public class CreateActivity : IAuthenticatedCommand
```

در اینترفیس IAuthenticatedCommand یک Guid نیاز داریم. با این تعریف تمامی کلاس هایی که از IAuthenticatedCommand ارث بری میکنند می بایست UserId داشته باشند.

```
Guid UserId { get; set; }
```

در کلاس CreateActivity باید impeliment را انجام دهیم تا مشکل بر طرف شود.

```
public Guid UserId { get; set; }
```

حال به سراغ فیلد هایی که در این کلاس نیاز داریم می رویم.

```
public Guid Id { get; set; }  
public string Category { get; set; }  
public string Name { get; set; }  
public string Description { get; set; }  
public DateTime CreatedAt { get; set; }
```

حال یک قسمت برای لاگین پروژه نیاز داریم. در این پوشه یک Class به نام AuthenticateUser ایجاد میکنیم و از ICommand ارث بری میکند و فیلد های آنرا قرار میدهیم.

```
public class AuthenticateUser :ICommand  
{  
    public string Email { get; set; }  
    public string Password { get; set; }  
}
```

اما موضوعی که در این پروژه است، ما نمی توانیم به تک تک لایه های دیگر این دستورات را بدهیم. بنابراین به یک مدیریت کننده کامندها نیاز داریم. یک ایتترفیس جدید به نام ICommandHandler میسازیم و نوع آن را generic قرار میدهیم. زمانی که بخواهد نمونه سازی کند ، مشخص می شود که با کدام کامند متصل می شود. چون پروژه ما بزرگ می باشد در این قسمت از async ها نیز استفاده می کنیم.

```
public interface ICommandHandler <in T> where T:ICommand  
{  
    Task HandleAsync(T command);  
}
```

قسمت نهم:

در این قسمت ما می بایست برای کامند های خود event قرار دهیم. در Actio.Comman یک پوشه به نام Events میسازیم. یک ایتترفیس در آن به نام IEvent ایجاد میکنیم. برای تمامی کامند هایی که در گذشته ساختیم ما باید رویداد بنویسیم.

```
public interface IAuthenticatedEvent : IEvent  
{  
    Guid UserId{get;}  
}
```

برای Authenticated، set قرار نمیدهیم و این پارامتر فقط جهت خواندن می باشد. برای UserCreated یک کلاس ایجاد میکنیم. پس از وارد کردن متغیر های نام و ایمیل، چون که getway ما api می باشد،سریالایز

کردن UserCreated را با protected مشخص میکنیم. چون protected وسط در سطح لایه Api.Common در دسترس می باشد. حال یه کانستراکتور به نام UserCreated نیز در آن مینویسیم.

```
public class UserCreated : IEvent
{
    public string Email { get; }
    public string Name { get; }
    protected UserCreated()
    {

    }
    public UserCreated(string email, string name)
    {
        Email = email;
        Name = name;
    }
}
```

حال به سراغ ActivityCreated میرویم.

```
public class ActivityCraeted : IAuthenticatedEvent
{
    public Guid UserId { get; }
    public Guid Id { get; }
    public string Category { get; }
    public string Name { get; }
    public string Description { get; }
    public DateTime CreatedAt { get; }
    protected ActivityCraeted()
    {

    }
    public ActivityCraeted(Guid id,Guid userId,string category,
        string name,string description,DateTime createdAt)
    {
        this.Id=id;
        this.UserId=userId;
        this.Category=category;
        this.CreatedAt=createdAt;
        this.Description=description;
        this.Name=name;
    }
}
```

حالا نوبت ساخت کلاس Authenticated user می باشد.

```
public class UserAuthenticated : IEvent
{
```

```

public string Email { get; }
protected UserAuthenticated()
{

}
public UserAuthenticated(string email)
{
    this.Email=email;
}
}

```

برای اینکه بتوانیم خطاها را به سرویس های دیگر اطلاع دهیم و یا پیغامی را برای آنها بفرستیم، یک interface به نام IRejectedEvent می سازیم.

```

public interface IRejectedEvent : IEvent
{
    string Reason { get; }
    string Code { get; }
}

```

اگر برای user یک خطا رد شد، یک کلاس به نام CreateUserRejected می سازیم. چیزی که ما باید بفهمیم این است که کدام reject شده است.

```

public class UserAuthenticated : IEvent
{
    public string Email { get; }
    protected UserAuthenticated()
    {

    }
    public UserAuthenticated(string email)
    {
        this.Email=email;
    }
}

```

حال به سراغ createdActivity می رویم که اگر در آن مشکلی رخ داد چه باید کرد. کلاسی به نام CreateActivityRejected میسازیم.

```

public class CreateActivityRejected : IRejectedEvent
{
    public Guid Id { get; }
    public string Reason { get; }

    public string Code { get; }
    protected CreateActivityRejected ()
    {

```

```

    }
    public CreateActivityRejected (Guid id,string reason,string code)
    {
        this.Code=code;
        this.Id=id;
        this.Reason=reason;
    }
}

```

و در آخر برای پل ارتباطی بین command ها به یک ایتترفیس به نام IEventHandler نیاز داریم.

```

public interface IEventHandler<in T> where T:IEvent
{
    Task HanldeAsync(T @event);
}

```

قسمت دهم:

در این قسمت به سراغ سرویس ها می رویم. ابتدا باید چند پکیج در لایه Common باید نصب کنیم. وارد Actio.Common.csproj می شویم و پکیج های زیر را پیست میکنیم و گزینه Restore را میزنیم.

```

<ItemGroup>
  <PackageReference Include="Microsoft.AspNetCore" Version="2.0.0" />
  <PackageReference Include="Microsoft.AspNetCore.Hosting" Version="2.0.0" />
  <PackageReference Include="RawRabbit" Version="2.0.0-beta8" />
  <PackageReference Include="RawRabbit.DependencyInjection.ServiceCollection"
Version="2.0.0-beta8" />
  <PackageReference Include="RawRabbit.Operations.Publish" Version="2.0.0-beta8" />
  <PackageReference Include="RawRabbit.Operations.Subscribe" Version="2.0.0-beta8" />
</ItemGroup>

```

حال به سراغ ساخت پوشه سرویس می رویم. در لایه Api.Common یک پوشه به نام Services میسازیم و یک ایتترفیس به نام IServiceHost میسازیم. این ایتترفیس شامل یک تابع اجرا Run میباشد.

```

public interface IServiceHost
{
    void Run();
}

```

اما مهم ترین قسمت کار ما، یک کلاس به نام ServiceHost می باشد که آنرا میسازیم. این کلاس پل ارتباطی ما با مهم ترین ابزارها از جمله ریبیت، تمام کامند ها می باشد. این کلاس از IServiceHost ارث بری میکند. همچنین در این فایل از پکیج هایی که قبلا نصب کرده ایم استفاده میکنیم.

مانند `IWebHost` که وظیفه آن هاست کردن سرویس های ما را بر عهده دارد و آنرا در این کلاس `inject` هم میکنیم و متد `Run` را به وب هاست `return` میکنیم.

```
public class ServiceHost : IServiceHost
{
    private readonly IWebHost _webHost;
    public ServiceHost(IWebHost webHost)
    {
        _webHost = webHost;
    }
    public void Run()=> _webHost.Run();
}
```

حال در این کلاس ما به یک متد سازنده به نام هاست بیلدر `HostBuilder` نیاز داریم. این متد از نوع `public static` می باشد که نیاز به نمونه سازی نیز نداشته باشد، نام آن `Create` می نویسیم و یک نوع جنریک می باشد، ورودی آن `TStartup` می باشد، در صورت فراخوانی فضایی برای کامند های آن نیز اضافه میکنیم، بطور مثال شاید بر روی پورت های دلخواه انجام شود که `args [] string` را مینویسیم و این متد از کلاس نیز ارث بری میکند.

درون آن یک کنسول به نام `Title` مینویسیم که عنوان استارتاپ را به نمایش بگذارد. سپس نوبت کانفیگ است که یک متغیر کانفیگ نیز برای آن مینویسیم که نیازهایی که مورد نیاز باشد در آن اضافه کند و یک `CommandLine` نیاز داریم تا کامند ها در آن نوشته شود و در نهایت بیلد را مینویسیم.

سپس باید وب هاست خود را بنویسیم که در آن کانفیگ، استارتاپ نیز ذکر میکنیم و یک کلاس دیگر به نام `HostBuilder` میسازیم که تنظیمات و ارتباط با ریبیت را در خود جای دهد اما این کلاس را در قسمت بعد میسازیم.

در انتها برگشت این متد را به وب هاست بیلدر میدهیم .

```
public class ServiceHost : IServiceHost
{
    private readonly IWebHost _webHost;
    public ServiceHost(IWebHost webHost)
    {
        _webHost = webHost;
    }
    public void Run() => _webHost.Run();
    public static HostBuilder Create<TStartup>(string[] args) where TStartup : class
    {
        Console.Title = typeof(TStartup).Namespace;
    }
}
```

```

        var config = new
ConfigurationBuilder().AddEnvironmentVariables().AddCommandLine(args).Build();
        var webHostBuilder = WebHost.CreateDefaultBuilder(args)
        .UseConfiguration(config).UseStartup<TStartup>();
        return new HostBuilder(webHostBuilder.Build());
    }
}

```

قسمت یازدهم:

اکنون یک کلاس `abstract` به نام `BuilderBase` می نویسیم که درون آن یک متد `abstract` به نام `سرویس` هاست قرار می دهیم. با این کلاس و متد می توانیم برای متد های دیگر `override` انجام دهیم.

```

public abstract class BuilderBase
{
    public abstract ServiceHost Build();
}

```

اکنون به سراغ کلاس هاست بیلدر می رویم. درون هاست بیلدر به `IWebHost` و یکی از توابع `Rabbit` نیاز داریم.

```

public class HostBuilder : BuilderBase
{
    private readonly IWebHost _webHost;
    private IBusClient _bus;
    public HostBuilder(IWebHost webHost)
    {
        _webHost=webHost;
    }
}

```

همچنین یک کلاس درون `HostBuilder` به نام `BusBuilder` برای ساختن `Client` می نویسیم.

```

public BusBuilder UseRabbitMq(){
    _bus = (IBusClient) _webHost.Services.GetService(typeof(IBusClient));
    return new BusBuilder(_webHost,_bus);
}

```

اکنون به سراغ `ServiceHost` `override` می رویم.

```

public override ServiceHost Build()
{
    return new ServiceHost(_webHost);
}

```

سپس به سراغ نوشتن کلاس BusBuilder می رویم. درون آن به وب هاست، IBusClient و هاست بیلدر نیاز داریم و کدهای قبل را کپی میکنیم. موقع ساختن کلاینت ها باید کامند ها و رویداد ها را درون ریزی کنیم. باس بیلدر باید بتواند فراخوانی کند.

```
public class BusBuilder : BuilderBase {
    private readonly IWebHost _webHost;
    private IBusClient _bus;

    public BusBuilder (IWebHost webHost, IBusClient busClient) {
        _webHost = webHost;
        _bus = busClient;
    }

    public BusBuilder SubscribeToCommand<TCommand> () where TCommand : ICommand {
        var handler = (ICommandHandler<TCommand>) _webHost.Services
            .GetService (typeof (ICommandHandler<TCommand>));
        _bus.WithCommandHandlerAsync (handler);

        return this;
    }

    public BusBuilder SubscribeToEvent<TEvent> () where TEvent : IEvent {
        var handler = (IEventHandler<TEvent>) _webHost.Services
            .GetService (typeof (IEventHandler<TEvent>));
        _bus.WithEventHandlerAsync (handler);

        return this;
    }
    public override ServiceHost Build () {
        return new ServiceHost (_webHost);
    }
}
```

در ادامه باید برای دو async ها کلاس ایجاد کنیم.

قسمت دوازدهم:

در این قسمت باید دو extension باقی مانده را پیاده سازی کنیم. بنابراین باید یک پوشه به نام RabbitMQ در این لایه ایجاد میکنیم. درون آن یک کلاس به نام Extensions ایجاد میکنیم. در صورتی می توان از متدهای extension استفاده کرد که نوع کلاس static باشد و سپس متد ها را درون کلاس مینویسیم.

```

public static class Exetnsions
{
    public static Task WithCommandHandlerAsync<TCommand> (this IBusClient bus,
        ICommandHandler<TCommand> handler) where TCommand : ICommand =>
        bus.SubscribeAsync<TCommand> (msg => handler.HandleAsync (msg),
            ctx => ctx.UseConsumerConfiguration (cfg =>
                cfg.FromDeclaredQueue (q => q.WithName (GetQueueName<TCommand> ()))
            ));

    public static Task WithEventHandlerAsync<TEvent> (this IBusClient bus,
        IEventHandler<TEvent> handler) where TEvent : IEvent =>
        bus.SubscribeAsync<TEvent> (msg => handler.HandleAsync (msg),
            ctx => ctx.UseConsumerConfiguration (cfg =>
                cfg.FromDeclaredQueue (q => q.WithName (GetQueueName<TEvent> ()))
            ));

    private static string GetQueueName<T> () =>
        $"{System.Reflection.Assembly.GetEntryAssembly().GetName()}/{typeof(T).Name}";
}

```

اکنون پروژه ما تکمیل شده است و در کلاس باس بیلدر جهت رفع دو خطا ، فایل ریت در پوشه extensions را import میکنیم.

قسمت سیزدهم:

در این قسمت به سراغ endpoint ها می رویم که بتوانیم در api در از این سرویس باس استفاده بکنیم. در فایل Actio.Api.csproj یک سری بسته می بایست نصب کنیم. سپس یک کلاس در Controller به نام HomeController ایجاد میکنیم و از Controller ارث بری میکند.

برای استفاده از این کنترلر می بایست یک آدرس برای آن تعریف کنیم، بنابراین صفت Route را نیز به آن اضافه میکنیم که آدرس این صفت برای HomeController خالی می باشد. سپس یک سری متد ها برای آن مینویسیم.

ابتدا از متد Get جهت فراخوانی آدرس های برنامه شروع به نوشتن میکنیم. برای آنکه از این متد استفاده کنیم باید صفت HttpGet را به آن اضافه کنیم.

```

[Route("")]
public class HomeController:Controller
{
    [HttpGet("")]
    public IActionResult Get()=>Content("Hello From Actio API!");
}

```

یک کلاس جدید به نام `ActivitiesController` جهت عملیات های مورد نیاز نیز اضافه میکنیم و متد `Route` را به آن میدهیم تا مشخص شود کنترلر با چه آدرسی عمل کند و با کلمه کلیدی `controller` آدرس دهی می شود.

```
[Route("[controller]")]
public class ActivitiesController : Controller
{
    private readonly IBusClient _busClient;
    public ActivitiesController(IBusClient busClient)
    {
        _busClient = busClient;
    }
}
```

در این اکتیویتی به یک باس کلاینت نیاز داریم و آنرا `inject` میکنیم. سپس یک اکتیویتی دیگر مینویسیم جهت ساختن اکتیویتی که از نوع متد `HttpPost` که شامل `async Task` است. اکتیویتی `CreateActivity` نیز در این اکتیویتی استفاده می شود و چون که از `Post` استفاده می شود نیاز به مقدار برگشتی دارد. با استفاده از `await` کامند خود را پابلیش میکنیم و برای `return` یک مسیج `Accepted` نیز مینویسیم و بالاتر از آن برای کامند ها `id` از نوع `Guid` تعریف میکنیم. همچنین زمان ساخت اکتیویتی را نیز تعریف میکنیم.

```
[HttpPost]
public async Task<IActionResult> Post([FromBody] CreateActivity command)
{
    command.Id = Guid.NewGuid();
    command.CreatedAt = DateTime.Now;
    await _busClient.PublishAsync(command);
    return Accepted($"activities/{command.Id}");
}
```

اکنون به سراغ ساخت کنترلر دیگر به نام `UserController` می رویم. در این کنترلر به `IBusClient` نیز نیاز داریم و کد های آنرا مانند کنترلر گذشته تکرار می کنیم. سپس یک متد دیگر از نوع `HttpPost` و نام `register` تعریف میکنیم. متد با `async Task` و نوع `CreateUser` کامند بکار میبریم. با استفاده از این متدهایی که `return` می شوند کار پیش برود.

```
[Route("[controller]")]
public class UsersController : Controller
{
    private readonly IBusClient _busClient;
    public UsersController(IBusClient busClient)
    {

```

```
        _busClient = busClient;
    }
    [HttpPost("register")]
    public async Task<IActionResult> Post([FromBody]CreateUser command){
        await _busClient.PublishAsync(command);
        return Accepted();
    }
}
```