www.takbook.com

CQRS

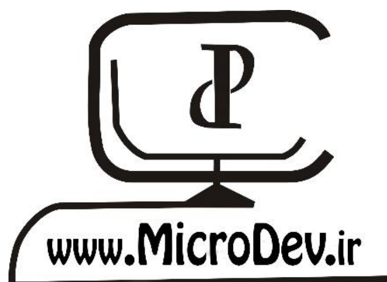
به زبان ساده

مولفین:

زهرابیات قلی لاله

علی بیات قلی لاله

www.takbook.com



CQRS

به زبان ساده

زهرا بیات قلی‌لاله

علی بیات قلی‌لاله

www.takbook.com

CQRS

به زبان ساده

مولفین : زهرا بیات قلی‌لاله – علی بیات قلی‌لاله

طراح جلد : زهرا بیات قلی‌لاله

مشخصات ظاهری : ۱۶۳ص

سال انتشار: خرداد ۹۹

قیمت : رایگان

www.takbook.com

فهرست

تقدیم به

با تشکر

مقدمه

فصل اول : نگاهی به دیزاین پترن CQRS و CQS

CQS چیست؟

CQRS چیست؟

چرا CQRS انتخاب مناسبی است؟

مشکل اپلیکیشن‌های سنتی چیست؟

Task Based Interface چیست؟

انواع پیام در CQRS

فصل دوم : استارت اپلیکیشن Ordering و ایجاد لایه‌ی Domain

ساختار لایه‌های اپلیکیشن

ایجاد لایه Domain

ایجاد SeedWrok اپلیکیشن

افزودن Domain Model

فصل سوم : ایجاد لایه‌ی Infrastructure

افزودن لایه‌ی Infrastructure

Table mapping چیست؟

پیاده‌سازی Repository

www.takbook.com

فصل چهارم : لایه‌ی Application و پیاده‌سازی Commmand

افزودن لایه‌ی Application

نصب و راه‌اندازی MediatR

پیاده‌سازی Command

پیاده‌سازی CommandHandler

ConnectionString چیست؟

ایجاد دیتابیس

ایجاد Migration

Controller

پیاده‌سازی Fluent Validation

Seed چیست؟

تست اپلیکیشن در مرحله Command

فصل پنجم : لایه‌ی Application و پیاده‌سازی Query

معرفی Query

پیاده‌سازی GetOrderByIdQueryHandler

استفاده از Dapper

فصل ششم : Domain event و Behavior در MediatR و گذری بر جداسازی دیتابیس‌ها

Domain Event چیست؟

پیاده‌سازی Domain Event

Behavior چیست؟

www.takbook.com

تست Behavior

جداسازی دیتابیس

استراتژی‌های همگام‌سازی دیتابیس

Consistency بین دیتابیس‌ها

Quiz

Answers

www.takbook.com

تقدیم به

تقدیم به تمام دوستداران برنامه‌نویسی که آماده‌ی استفاده از تمام قابلیت‌های خود برای یادگیری هستند.

www.takbook.com

با تشکر

از شـــرکت مدیریت روشـــمند و مدیر عامل خوبم جناب آقای مهندس عادل فیض‌بلبت دلگرمی‌ها و تشویق‌هایی که در نوشتن کتاب به من داشتند بسیار سپاسگزارم.

www.takbook.com

مقدمه

CQRS پترنی است که در پروژه‌های سازمانی استفاده می‌شود .و مشکل بزرگی را از پروژه‌های نرم‌افزاری حل می‌کند.

متاسفانه استفاده از این پترن در شرکت‌های نرم‌افزاری بسیار محدود است چون افرادی که توانایی کار با این الگوی را در پروژه های واقعی داشته باشند بسیار کم هستند.

در اینترنت اطلاعات زیادی در مورد CQRS وجود دارد اما اکثراً به معرفی الگوی CQRS و چند مثال انتزاعی بسنده می‌کنند. در این کتاب شما یاد می‌گیرید که چرا باید از CQRS استفاده کنید و چگونه آن را در پروژه‌های نرم‌افزاری پیاده‌سازی نمایید.

این کتاب آموزش CQRS را به صورت مختصر، با مثال های‌عملی و بدون مقدمه‌های طولانی به شما یاد می‌دهد و شما را با اصول CQRS و مزایایی که برای اپلیکیشن به ارمغان می‌آورد آشنا می‌کند.

بعد از مطالعه این کتاب یاد می‌گیرید که CQRS دقیقا چیست؟ چه اصولی در پشت آن وجود دارد؟ و چطور می‌توانید در پروژه های واقعی از مزیت‌های آن بهره‌مند شوید.

پروژه این کتاب را می‌توانید از مسیر Github پایین دانلود کنید.

https://github.com/ZahraBayatgh/CQRSFundamentals

www.takbook.com

فصل اول : نگاهی به دیزاین پترن CQRS و CQS

آنچه خواهید آموخت:

- CQS چیست؟
- CQRS چیست؟
- چرا CQRS انتخاب مناسبی است؟
- بررسی مشکلات اپلیکیشن‌های سنتی
- Task Based Interface چیست؟
- انواع پیام در CQRS

www.takbook.com

CQS چیست؟

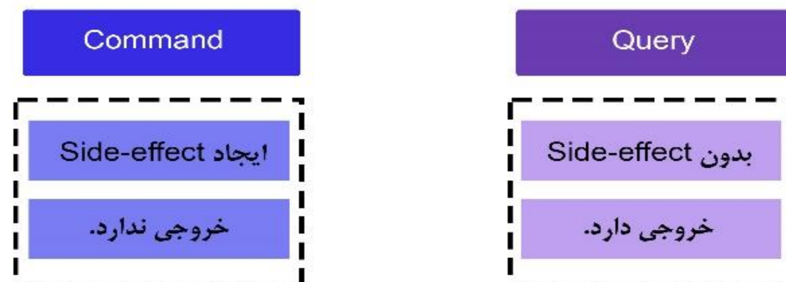
قبل از اینکه بدانیم CQRS چیســـت بیایید گذری بر ایده‌ی CQS بزنیم، چون CQRS براســـاس این ایده بنا شده است.

CQS ایده‌ایســت که توسـط Bertrand Meyer مطرح شـد. در این ایده می‌گوید یک متد یا باید Command باشـــد یا Query، اما نمی‌تواند هر دوی این عملیات را با هم انجام دهد. به طور ســـاده‌تر، یک متد تنها می‌تواند یکی از کارهای پایین را انجام دهد:

- **یا State را تغییر می‌دهد.**
- **یا نتیجه‌ی یک Query را برمی‌گرداند.**

اما هر دوی این عملیات را نباید با هم داشته باشد.

ما با این Pattern می‌توانیم مطمئن شـویم متدی که Queryی را اجرا و مقداری را برمی‌گرداند، هیچ‌گاه State یک آبجکت را تغییر نخواهد داد بنابراین هیچ‌گونه Side Effectی هم ندارد.



برای اسـتفاده از این Pattern باید خروجی متدهایی که State آبجکت را تغییر می‌دهند از نوع void باشـند در غیر این صورت، متد باید یک خروجی داشته باشد.

با این روش نه تنها خوانایی کد بالا می‌رود بلکه با یک نگاه به Signature متد، هدف آن معلوم می‌شود.

نکته!!

توجه داشــته باشــید که دنبال کردن این اصـل همیشــه امکان پذیر نیســت. گاهی موقعیت‌هایی پیش می‌آید که مجبور می‌شویم متدی بنویسیم که هم عمل Read دارد و هم Write. به طور مثال:

www.takbook.com

متد Pop در Stack. این متد هم وظیفه‌ی حذف آخرین عنصر وارد شده به Stack را دارد و هم باید آن عنصر را برگرداند.

بنابراین هم عملیاتی انجام می‌شود و هم نتیجه‌ای بازگشت داده خواهد شد پس این متد اصل CQS را نقض می‌کند.

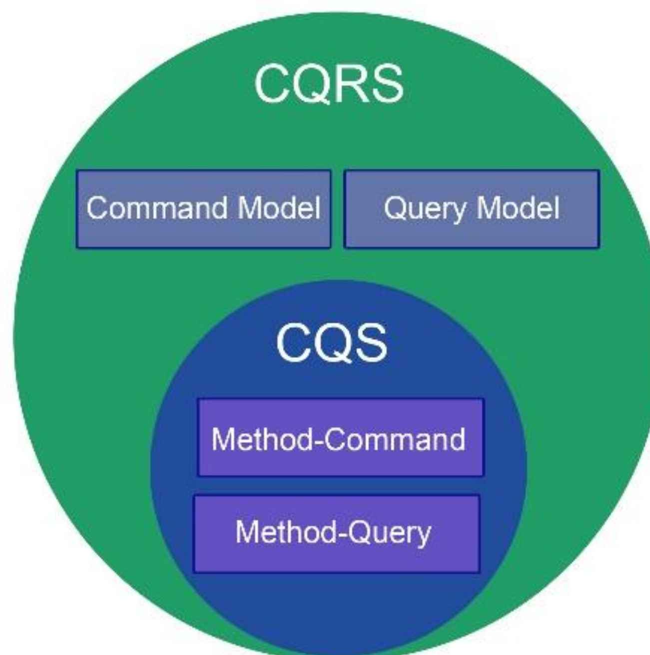
```
var stack = new Stack<string>();
stack.Push("value");          //Command
string value = stack.Pop();   //Both query and command
```

اما...

CQRS چیست؟

CQRS یک Pattern بسیار ساده است که ایده CQS را در سطح بالاتری پیاده‌سازی می‌کند. این Pattern در سال ۲۰۱۰ توسط Greg Young مطرح شد و براساس آن، عملیات Read و Write باید در سطح Domain Model از هم جدا شوند.

بنابراین یک Domain Model را به دو مدل Read و Write تقسیم می‌کنیم که در مدل Write فقط Command اجرا می‌شود و در مدل Read تنها Query خواهیم داشت.



www.takbook.com

چرا CQRS انتخاب مناسبی است؟

خب تا اینجا متوجه شدیم که CQRS یک Domain Model را به دو مدل Read و Write تقسیم می‌کند. اما مطمئنا این موضوع به خودی خود هدف نهایی این Pattern نیست.

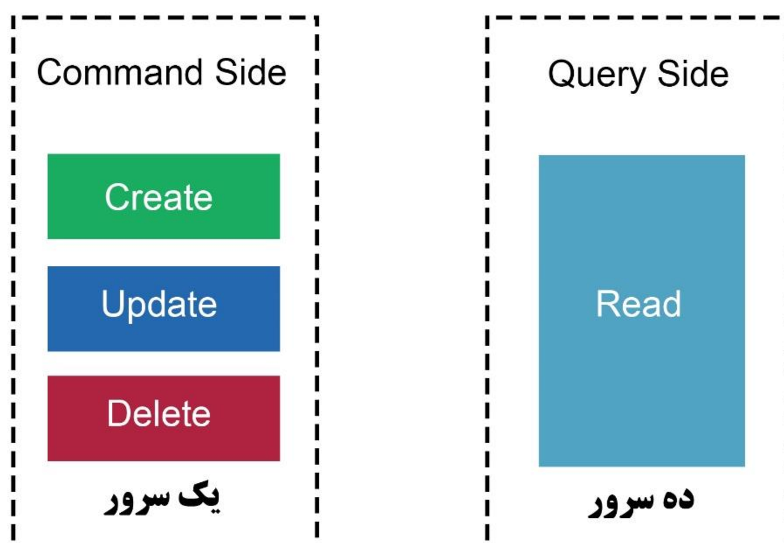
پس هدف این Pattern چیست؟

برای پاسخ به این سوال، بهتر است قبل از هر کاری به مزایای این Pattern بپردازیم:

۱) اولین مزیت Scalability است.

اگر به اپلیکیشن‌های سازمانی نگاه کنید، متوجه خواهید شد که عمل خواندن اطلاعات بیشتر از عملیات ایجاد، حذف و آپدیت استفاده می‌شود بنابراین مهم است که عملیات خواندن اطلاعات نسبت به نوشتن بیشتر Scale شود. **به طور مثال:**

عملیات نوشتن را تنها روی یک سرور Host کنیم و برای عملیات خواندن ده سرور در نظر بگیریم.



۲) دومین مزیت Performance است.

اگر مدل Read و Write را از هم جدا کنیم، حتی اگر هر دو مدل هم در یک سرور Host شوند باز هم می‌توان تکنیک‌های بهینه‌سازی را روی آن‌ها اعمال کرد. **به عنوان مثال:**

می‌توانید برای APIهای Query خود Cache داشته باشید یا اینکه دیتابیس Read و Write را از هم جدا کنید و با این کار از ویژگی‌های خاص دیتابیس استفاده نمایید. البته توجه داشته باشید این مزیت با

www.takbook.com

Scalability ارتباط دارد و با توجه به تخصیص تعداد سرور به عملیات Write و Read می‌توان نتیجه Performance را نیز تغییر داد.

۳) **سومین مزیت Simplicity است.**

این مزیت مهمترین مزیت این Pattern می‌باشد. Command و Query دو نیاز متفاوت هستند که وقتی هر دو در یک مدل قرار می‌گیرند، پیچیدگی مدل بیشتر می‌شود و در این صورت مدل نمی‌تواند به هیچکدام از این نیازها به خوبی پاسخ دهد.

زمانیکه عملیات Read و Write را از هم جدا می‌کنید پیچیدگی کد کم می‌شود و می‌توانیم روی هر کدام به طور مستقل کار کنیم

نتیجه گیری

CQRS اصل Single Responsibility Principle را در سطح اپلیکیشن اعمال می‌کند. در این Pattern شما دو مدل دارید که هر کدام تنها یک کار را انجام می‌دهند.

CQRS تصمیمات ما را برای موقعیت‌های مختلف بهینه خواهد کرد و ما می‌توانیم سطوح مختلفی از Consistency داشته باشیم. علاوه بر این مزایا، ما می‌توانیم دیتابیس Command وQuery را نیز جدا کنیم که این جداسازی باعث می‌شود تا بتوانیم برای عملی مثل خواندن اطلاعات که نیاز به واکشی اطلاعات با سرعت بالا دارد، از دیتابیسی مثل ElasticSearch استفاده کنیم.

نکته!!

Consistency یعنی در سیستم‌های توزیع شده، داده‌هایی که در Nodeهای مختلف هستند باید یکسان باشند تا کلاینت‌هایی که به این Nodeها متصل می‌شوند همگی یک دیتا را ببینند.

مشکل اپلیکیشن‌های سنتی چیست؟

شاید این سوال در ذهن شما پیش آمده باشد که: **مشکل اپلیکیشن‌های سنتی چیست که باید از الگوی CQRS استفاده کنیم؟**

قبل از بررسی مشکل اپلیکیشن‌های سنتی بیایید ببینیم که **CRUD چیست؟**

CRUD تفکری است که می‌گوید:

۱) **عملیات یک شی باید یکی از عملیات Create, Read, Update, Delete باشد.**

www.takbook.com

۲) ما برای تمامی عملیات باید از یک مدل مشترک استفاده کنیم.

۳) و معمولا این تفکر با Anemic Domain Model همراه است.

نکته!!

Anemic Domain Model به مدلی گفته می‌شود که :

۱) فقط Property داشته باشد.
۲) هیچ عملیات یا اعتبارسنجی درون آن انجام نشود.

من مطئنم که شما با اپلیکیشن‌های CRUD Based خیلی کار کرده‌اید اما شاید تا حالا متوجه نشده باشید که این روش مشکلاتی با خود به همراه دارد.

اما مشکل چیست؟

- **اولین مشکل Scalability است.**

چون در اپلیکیشن‌های CRUD Based مدل Read و Write ما یکی است پس نمی‌توانیم قسمتی از اپلیکیشن را به طور جداگانه Scale کنیم.

- **دومین مشکل Performance است.**

در این اپلیکیشن‌ها، ما برای عملیات Read و Write از یک مدل استفاده می‌کنیم که این برای سیستم‌های CRUD Based به خوبی کار می‌کند اما در اپلیکیشن پیچیده این روش Performance خوبی ندارد. **به‌طور مثال:**

معمولا حجم کاری Read و Write اطلاعات یکی نیست و اگر از یک مدل استفاده کنیم باعث افت Performance خواهد شد.

- **سومین مشکل پیچیدگی است.**

در یک اپلیکیشن Queryهای مختلفی وجود دارد که هر کدام خروجی خودشان را دارند که این خروجی با مدل Write یکی نیست. **به عنوان مثال:**

فیلدهای کوئریی که از Join شدن چند جدول بدست می‌آید با مدل Write یکی نیست.

www.takbook.com

از طرف دیگر ممکن است اعتبارسنجی و منطق پیچیده‌ای برای ورود اطلاعات اپلیکیشن داشته باشید اما هنگام خواندن اطلاعات نیازی به این پیچیدگی نداریم.

نکته!!

یکی از نشانه‌های تفکر CRUD این است که از DTOهایی استفاده می‌کنید که پر از فیلدهایی هستند که در ۱۰۰ درصد موارد بلا استفاده می‌باشند.

این فیلدها فقط در سناریوهای خاص استفاده می‌شوند و در سایر موارد کاربردی ندارند.

اما مشکل اپلیکیشن‌های سنتی چیست؟

همانطور که می‌دانید، یکی از مشکلات توسعه‌نرم افزار این است که برنامه‌نویسان داده‌ها را تغییر می‌دهند و همیشه مشکلات در تغییرات بوجود می‌آیند.

کلیه عملیات در یک اپلیکیشن سنتی در یکی از چهار دسته Create, Read, Update, Delete یا اصطلاحا CRUD قرار می‌گیرند که این کار از نظر فنی درست است اما همانطور که بالاتر هم توضیح دادیم استفاده از آن ایده خوبی نیست.

CRUD Based بودن تاثیر بدی رو سیستم می‌گذارد چون:

- نگهداری پروژه را سخت می‌کند.
- کاربران از اپلیکیشن ما تجربه خوبی نخواهند داشت.

به طور خلاصه اپلیکیشن‌های سنتی ۳ مشکل اساسی دارند:

۱) رشد بی رویه پیچیدگی:

این اپلیکیشن‌ها معمولا CURD Based هستند و بیشتر متدهایشان چندین عملیات را با هم انجام می‌دهند به همین دلیل وقتی تغییری در کدهای پروژه ایجاد کنید، اپلیکیشن پر از باگ می‌شود و نمی توانید به موقع پروژه را تحویل دهید. این مسئله خیلی زودتر از آنچه که فکر کنید بر سر شما خواهد آمد حتی اگر برنامه شما خیلی خیلی هم ساده باشد.

۲) مشکل دوم این است که نگاه مشتری با نگاه برنامه‌نویس متفاوت است:

مشتری با اصطلاحات CRUD صحبت نمی‌کند و اگر هم صحبت کند بخاطر این است که شما به او این اصطلاحات را آموزش داده‌اید. **برای مثال:**

www.takbook.com

فرض کنید وارد یک موسسه آموزشی شده‌اید و مدیر موسسه می‌خواهد شما را ثبت‌نام کند. مطمئنا او به شما نمی‌گوید که من باید شما را در سیستم Create کنم چون این کلمه برای شما بی‌معنی است. او به شما می‌گوید که شما را باید ثبت‌نام کنم.

بهترین روش برای رفع این مشکل، وجود یک زبان مشترک و یکپارچه است که هم برنامه‌نویس و هم مشتری بتوانند با آن صحبت کنند. به این زبان که یکی از اصول Domain Driven Design است اصطلاحا Ubiquitous Language می‌گویند.

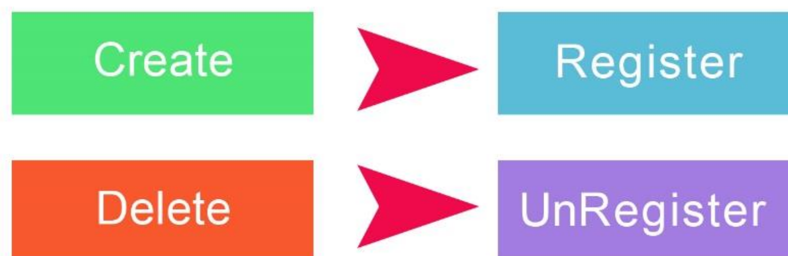
در صورتیکه از این زبان مشترک استفاده نکنید با رشد پیچیدگی و مسائل مربوط به نگهداری کد مواجه خواهید شد زیرا بین کد شما و زبان مشتری سازگاری وجود ندارد و شما مجبورید صحبت‌های مشتری را ترجمه کنید. این باعث می‌شود موضوع را درست درک نکرده و دقت کار پایین بیاید.

با این زبان مشترک نه تنها باید صحبت کنید بلکه باید در کدها و APIهای خود هم از آن استفاده کنید.

اما این زبان مشترک چطور عملیات را توصیف می‌کند؟

به یاد داشته باشید که عنوان هر APIی که با یکی از عملیات CRUD نامگذاری شود یک پرچم قرمز است و هرگاه خواستید APIی خود را با این اصطلاحات نامگذاری کنید، از خود بپرسید **آیا فقط همین نام را می‌توان برای این API گذاشت یا نام دیگری هم وجود دارد**؟ با این کار، شما می‌توانید درک عمیقی در مورد موضوعی که روی آن کار می‌کنید به دست آورید.

تنها راه پیدا کردن این اصطلاحات، پرسیدن زیاد از مشتری است تا بتوانید مثلا به جای گفتن کلمه‌ی Create، کلمه Register را به کار ببرید.

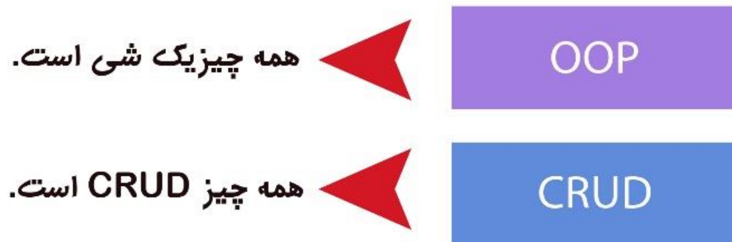


نکته!!

www.takbook.com

جالب اســت بدانید تفکر CRUD از Object Oriented Programming نشــات می‌گیرد. در OOP می‌گوید همه چیز یک شـی اسـت و چون برنامه‌نویسـان دوسـت دارند همه چیز را یکی کنند، این تفکر و این سـیسـتم را تمیز می‌بینند و احساس می‌کنند این سیسـتم جایی است که همه‌ی APIها دارای یک هدف کاملاً تعریف شده هستند. اما یادتان باشد:

تنها باید با مشتری صحبت کنید، اصطلاحات مناسب را کشف و تفکر خود را بر این اساس تنظیم نمایید.



اما مشکل سوم آسیب رساندن به تجربه‌ی کاربری از اپلیکیشن است:

مشکل تفکر CRUD فقط به کدهای سمت Backend ختم نمی‌شود، این موضوع روی UI هم تاثیر می‌گذارد.

به طور مثال:

مطمئنا یک متد Update که چندین کار را با هم آپدیت می‌کند نیاز به طراحی یک UI پیچیده‌تر هم دارد. شما با تفکر CRUD این متد آپدیت را می‌نویسید اما کاربران به سختی این UI را متوجه می‌شوند.

در ســیســـتم‌های پیچیده، کاربران حتی پس از مدت‌ها کار با نرم‌افزار هم نمی‌توانند به همه‌ی قســمت‌های اپلیکیشن مسلط شوند چون UI سیستم پیچیده است و نمی‌تواند کاربران را قدم به قدم راهنمایی کند.

نکته!!

توجه داشته باشید تفکر CRUD همیشه هم بد نیست بعضی مواقع که اپلیکیشن زیاد پیچیده نیست یا اینکه در آینده نمی‌خواهید آن را نگهداری یا توسعه دهید این رویکرد خوب است.

Task Based Interface چیست؟

ما فهمیدیم که رویکرد CRUD پیچیده است و مشکلاتی هم دارد اما **راه‌حل این مشکلات چیست؟**

پاسخ این سوال را رویکرد Task Based Interface می‌دهد.

این رویکر برعکس رویکرد CRUD می‌باشــد. در این رویکرد باید کارهایی که قرار اســت با آبجکت‌های برنامه انجام شود را :

۱) **شناسایی،**

www.takbook.com

۲) **سپس به چند کار کوچک‌تر شکسته،**

۳) **و در نهایت این کارهای کوچک پیاده‌سازی شود.**

چون در این رویکرد متدهای بزرگ و پیچیده شکسته و هر کدام مسئول انجام یک کار می‌باشند بنابراین ما می‌توانیم متدها را ساده‌تر بنویسیم و به اصل Single Responsibility Principle پایبند باشیم.

نتیجه گیری:
برنامه‌نویسی به سبک Task Base Interface باعث می‌شود تا ما متدهای بزرگ ننویسیم پس در نتیجه به راه‌حل‌های دقیق و ساده‌تری برسیم و نرم‌افزار را بهتر پیاده‌سازی نماییم. از همه مهمتر چون هر Task تنها یک کار را انجام می‌دهد بنابرین ما به سادگی می‌توانیم اپلیکیشن خود را توسعه دهیم.

نکته!!
توجه داشته باشید Task Based Interface پیش‌نیاز CQRS نیست اما می‌تواند در کنار CQRS استفاده شود.

انواع پیام در CQRS

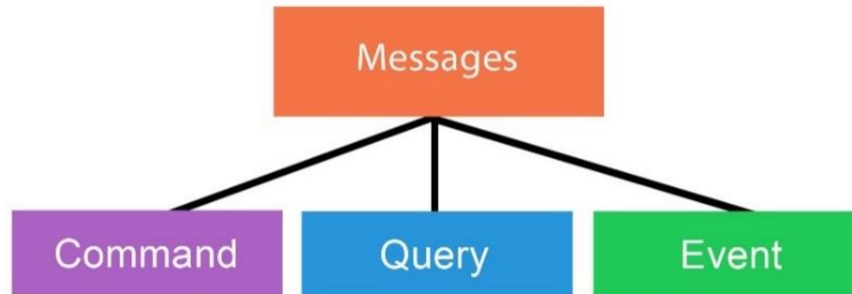
در سیستم CQRS اپلیکیشن از یک سری پیام تشکیل شده که مشخص می‌کند کلاینت چه کاری می‌خواهد انجام دهد. به طور کلی این پیام‌ها که همگی بخشی از Domain Model هستند به سه دسته تقسیم می‌شوند:

۱) **Command** : Command پیامی است که به اپلیکیشن دستور انجام یک کار را می‌دهد.

۲) **Query** : Query پیامی است که از اپلیکیشن می‌خواهد چیزی را نمایش دهد.

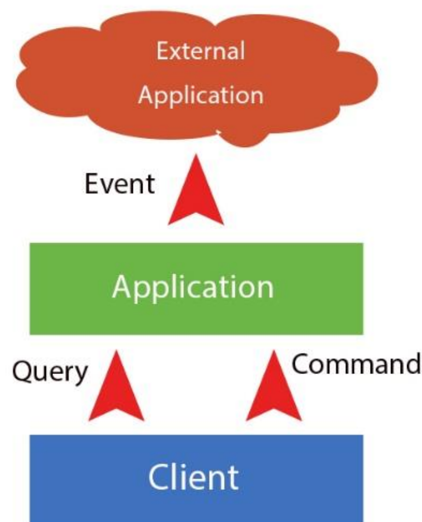
۳) **Event** : Event پیامی است که اپلیکیشن ما برخی تغییرات رخ داده شده را با استفاده از آن به قسمت‌های مختلف اطلاع می‌دهد.

www.takbook.com



نتیجه گیری

به طور کلی کلاینت برای انجام یک عملیات در سیستم، یک Command می‌فرستد و برای پرسید یک سوال یک Query را درخواست می‌کند. از طرف دیگر بعد از اجرای Command اپلیکیشن با یک Event می‌تواند با اپلیکیشن‌های بیرونی ارتباط برقرار کند و تغییرات درون برنامه را اطلاع دهد.



تا اینجا تفاوت این سه پیام را متوجه شدیم **اما چطور باید این پیام‌ها را نامگذاری کنیم؟**

قبل از نامگذاری باید تفاوت Event و Command را بدانیم.

تفاوت بین Command و Event چیست؟

۱) Command کاری است که باعث ایجاد یک تغییر در Domain Model می‌شود و Event نتیجه‌ی همان تغییر است.
۲) Command را کاربر می‌فرستد در حالیکه Event توسط اپلیکیشن فرستاده خواهد شد.
۳) در Command کلاینت به سرور دستور انجام یک کار را می‌دهد و سرور با توجه به شرایط می‌تواند آن را انجام یا رد کند. **به طور مثال:**

www.takbook.com

سیستم می‌تواند درخواست ثبت سفارش را به دلیل نامعتبر بودن محصول رد کند. اما اپلیکیشن نمی‌تواند یک Event را رد کند زیرا Event کاری است که قبلا انجام شده و فقط جنبه اطلاع‌رسانی دارد. پس شما نمی‌توانید اتفاقی که قبلا افتاده را تغییر دهید.

با این سه مقایسه نتیجه می‌گیریم نامگذاری این پیام‌ها بسیار مهم است و باید به درستی انجام شود. پس بیایید با هم دستورالعمل‌های نامگذاری را بررسی کنیم:

۱) نام Command همیشه باید دستوری باشد زیرا این کلاس همیشه دستور انجام یک کار را به اپلیکیشن می‌دهد. **به طور مثال:**

CreateOrderCommand که دستور ایجاد سفارش را می‌دهد.

۲) نام Query معمولا با کلمه Get شروع می‌شود چون Query از اپلیکیشن درخواست نمایش برخی اطلاعات را می‌کند و به نظر من گذاشتن کلمه‌ای غیر از Get مناسب نیست. **به عنوان مثال:**

GetOrderByIdQuery که نمایش یک سفارش را براساس Id آن درخواست می‌کند.

۳) و اما نام Event همیشه باید طوری گذاشته شود که اشاره به کاری باشد که در گذشته انجام شده. **برای مثال:**

OrderStartedDomainEvent که اشاره به این دارد که سفارشی شروع شده است.

۴) و در پایان باید بگویم که در نام‌گذاری این سه پیام بهتر است از پسوندهای Command، Query و Event استفاده کنید زیرا این پسوندها نمایانگر این سه نوع پیام هستند و حتی اگر در نامگذاری درست عمل نکرده و قوانین بالا را رعایت نکنید باز هم اپلیکیشن منظور پیام شما را متوجه خواهد شد.

www.takbook.com

فصل دوم : استارت اپلیکیشن Ordering و ایجاد لایه‌ی Domain

آنچه خواهید آموخت:

- ساختار لایه‌های اپلیکیشن
- ایجاد لایه Domain
- ایجاد SeedWrok اپلیکیشن
- افزودن Domain Modelها

www.takbook.com

ساختار لایه‌های اپلیکیشن

قبل از نوشتن هرگونه کدی ابتدا باید بدانیم **لایه‌بندی چیست؟**

لایه‌بندی راهی اســـت که ما یک اپلیکیشـــن پیچیده را به واحدهای کوچکتر می‌شـــکنیم. در حقیقت لایه‌ها، پارتیشن‌های عمودی برنامه‌ی ما هستند که برای مشخص کردن سطوح مختلف Abstraction و حفظ Single Responsibility Principle استفاده می‌شوند.

هدف لایه‌بندی این است که مهارت برنامه‌نویس با کاری که انجام می‌دهد یکی باشد. **به طور مثال :**

برنامه‌نویس Backend مجبور به انجام کارهای Front نشود.

قبلا یک معماری سه لایه وجود داشت که از سه لایه‌ی زیر تشکیل می‌شد:

۱) **User Interface Layer :** که نمایانگر اینترفیس اپلیکیشن بود.
۲) **Business Layer :** که شامل منطق بیزینسی ما بود.
۳) **Data Access Layer :** که شامل منطق Read و Write دیتابیس بود.

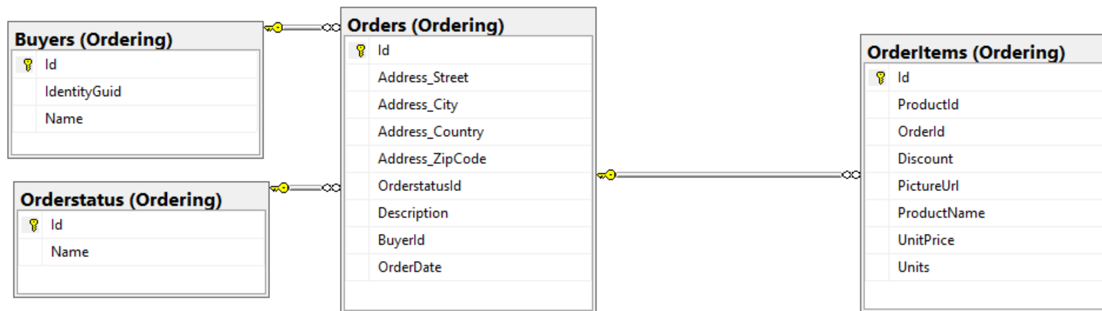
این معماری فقط مناسب یک اپلیکیشن CRUD است و جوابگوی یک Domain Model پیچیده نیست. علاوه بر این، ابهامات زیادی در مورد محل قرارگیری لایه‌ی Domain با لایه‌ی Application وجود دارد. این مســئله باعث به وجود آمدن یک معماری چهار لایه‌ی Domain Driven شد.

معماری Domain Driven چهار لایه دارد :

۱) **Presentation Layer :** که نمایانگر User Interface اپلیکیشن است.
۲) **Application Layer :** که شامل Use Caseها و کدهای اجرایی اپلیکیشن می‌باشد.
۳) **Domain Layer :** که شامل Domain Logic است.
۴) **Infrastructure Layer :** این لایه شامل کدهای زیرساختی و کدهای ارتباط با دیتابیس می‌باشد.

مثال این کتاب یک سیستم Ordering است که می‌خواهیم با استفاده از الگوی CQRS آن را پیاده‌سازی کنیم. معماری این مثال ۴ لایه اســـت اما ما با UI کاری نداریم یعنی Presentation Layer را حذف کردیم و تمرکز اصلی این اپلیکیشن را بر روی Backend گذاشتیم.

www.takbook.com



این اپلیکیشن نیاز به سه لایه‌ی API, Domain و Infrastructure دارد :

- **لایه‌ی Domain :** این لایه شامل Domain Modelهای ماست و مسئول مدیریت مفاهیم بیزینسی می‌باشد. این لایه به هیچ لایه‌ای وابستگی ندارد.
- **لایه‌ی Infrastructure :** داده‌هایی که در Domain Entityهای ما قرار گرفته توسط این لایه در دیتابیس ذخیره می‌شود بنابراین این لایه به لایه Domain Model وابسته است و باید تمام زیر ساخت‌های داده‌ای و تکنولوژی محور نرم‌افزار را در آن پیاده‌سازی کنیم.
- **لایه‌ی API :** این لایه منطق نرم‌افزار را مدیریت و پیاده‌سازی می‌کند و کلاینت از طریق این لایه با اپلیکیشن ارتباط برقرار می‌کند. این لایه به دو لایه‌ی بالا وابسته است.

بیایید شروع کنیم...

ایجاد لایه Domain

یک سیستم نرم‌افزاری از دو بخش اصلی ایجاد شده است :

۱) **Application Logic :** این قسمت بخشی از نرم‌افزار است که به Use-Caseهایی که باید پیاده‌سازی شوند وابسته است.

۲) **Domain Logic :** این بخش شامل مفاهیم و قوانین بیزینسی بوده و کاملا از جزئیات فنی جدا می‌باشد.

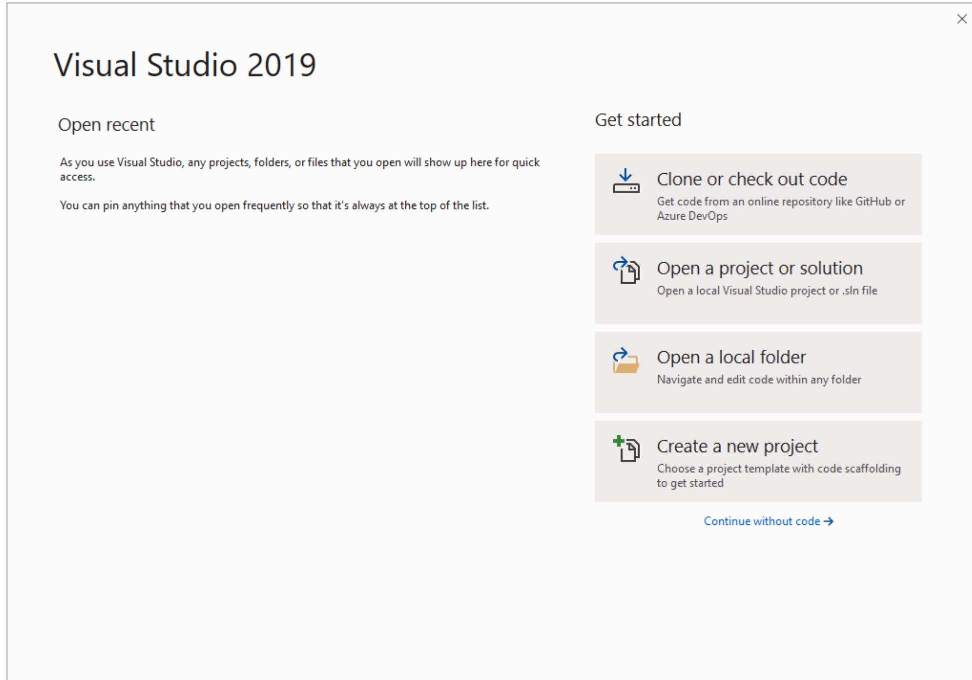
همانطور که بالاتر گفتیم وظیفه‌ی اصلی Domain Layer مفاهیم و قوانین بیزینسی است بنابراین Domain Logic در طراحی یک نرم‌افزار باید در لایه‌ی Domain قرار گیرد.

این لایه به عنوان قلب نرم‌افزارهای تجاری شناخته می‌شود پس باید آن را به بهترین نحوه طراحی کنیم.

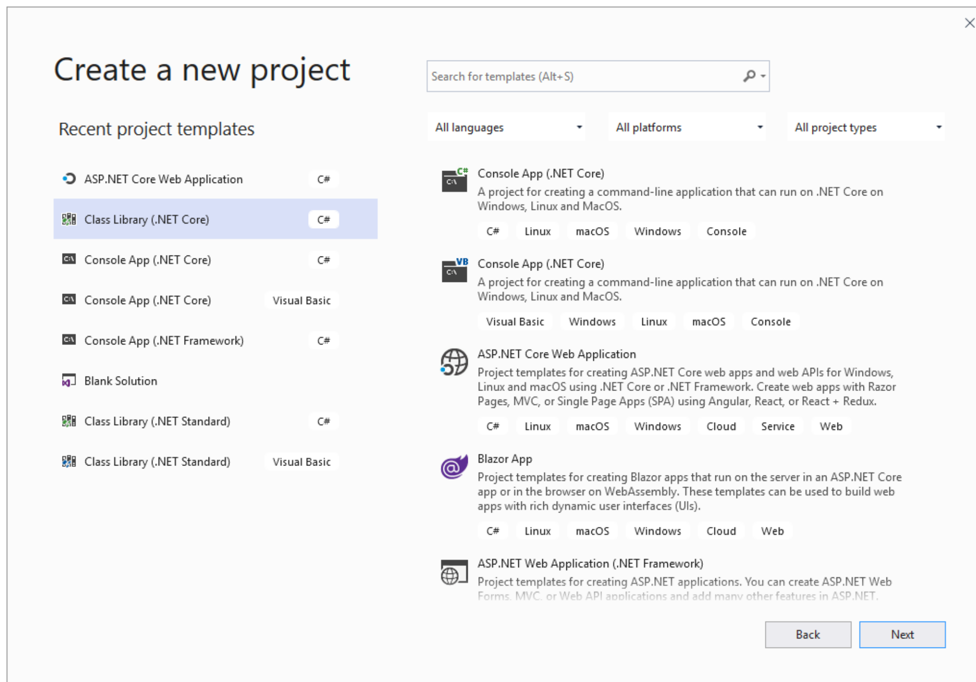
www.takbook.com

بیایید با هم این لایه را طراحی کنیم.

- **Visual Studio 2019** را باز، سپس بر روی **Create New Project** کلیک نمایید.

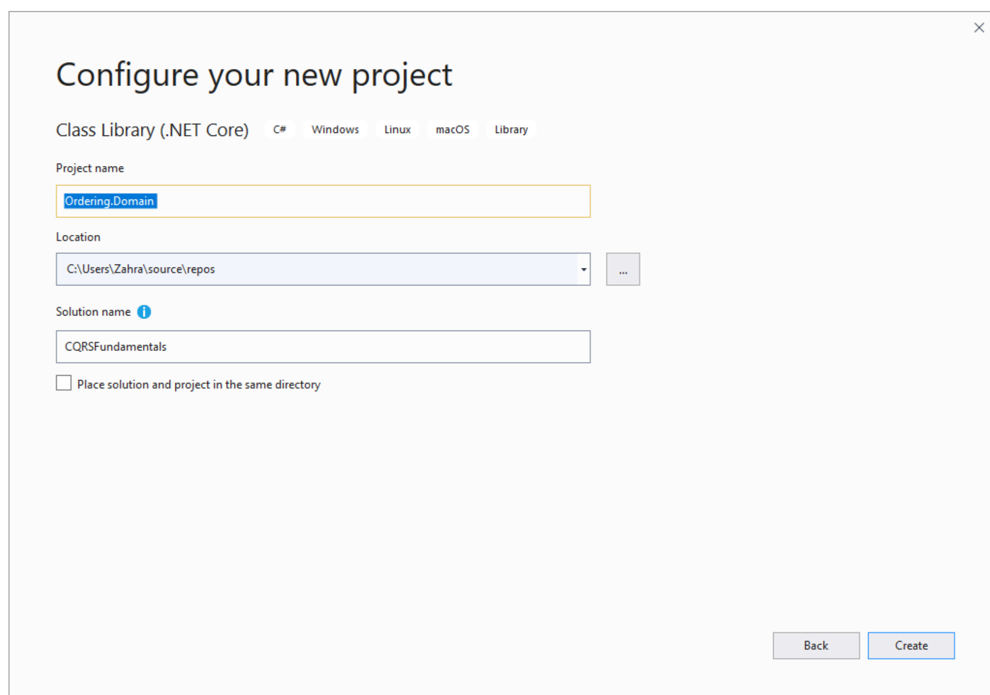


- حالا در کادر باز شده **Class Library (.NET Core)** را انتخاب و بر روی **Next** کنید.



www.takbook.com

- در کادر بعدی‌نام Solution را CQRSFundamentals و‌نام پروژه را Ordering.Domain بگذارید و سپس بر روی Create کلیک کنید.

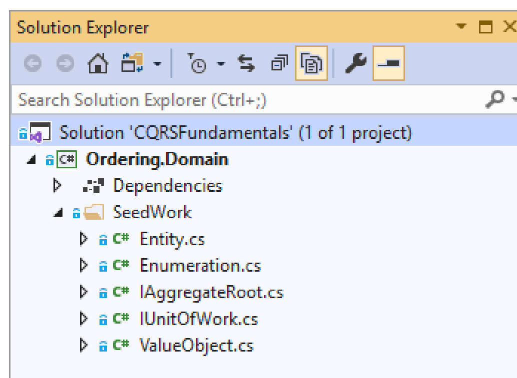


تبریک! پروژه API شما با موفقیت ایجاد شد.

ایجاد SeedWrok اپلیکیشن

اولین گام این پروژه حذف کدهای تکراری در Domain Classهاست بنابراین باید تعدادی کلاس پایه داشته باشیم تا Entityهای ما با ارثبری از آن‌ها، نیاز به Copy/Paste کدهای اضافه نداشته باشند.

پس در اینجا ما فولدری به نام SeedWork ایجاد و سپس کلاس‌های Entity، Enumeration، ValueObject و اینترفیس IUnitOfWork و IAggregateRoot را در آن اضافه می‌کنیم.



www.takbook.com

کدهای کلاس Entity :

یک اپلیکیشن شامل تعدادی Entity است پس باید یک کلاس پایه وجود داشته باشد که بتوانید کدهای مشترکی که در هر Entity تکرار می‌شود را در یک کلاس جمع کنید. کلاس Entity همان کلاسی است که کدهای تکراری مشترک در Domain Entityها را در خود نگه می‌دارد.

```
using System;

namespace Ordering.Domain.SeedWork
{
    public abstract class Entity
    {
        int? _requestedHashCode;

        int _Id;

        public virtual int Id
        {
            get
            {
                return _Id;
            }
            protected set
            {
                _Id = value;
            }
        }

     public bool IsTransient()
        {
            return this.Id == default(Int32);
        }

        public override bool Equals(object obj)
        {
            if (obj == null || !(obj is Entity))
                return false;

            if (Object.ReferenceEquals(this, obj))
                return true;

            if (this.GetType() != obj.GetType())
```

www.takbook.com

```
                return false;

            Entity item = (Entity)obj;

            if (item.IsTransient() || this.IsTransient())
                return false;
            else
                return item.Id == this.Id;
        }

        public override int GetHashCode()
        {
            if (!IsTransient())
            {
                if (!_requestedHashCode.HasValue)
                    _requestedHashCode = this.Id.GetHashCode() ^ 31; // XOR
                return _requestedHashCode.Value;
            }
            else
                return base.GetHashCode();

        }
        public static bool operator ==(Entity left, Entity right)
        {
            if (Object.Equals(left, null))
                return (Object.Equals(right, null)) ? true : false;
            else
                return left.Equals(right);
        }

        public static bool operator !=(Entity left, Entity right)
        {
            return !(left == right);
        }
    }

}
```

کدهای اینترفیس IAggregateRoot :

این اینترفیس جهت مشخص کردن AggregateRootهای Domain ماست.

Aggregate چیست؟

www.takbook.com

Agregregate یک Pattern است که چندین Entity مرتبط به هم را در یک Entity جمع می‌کند و هدف آن حفظ Invariantهای Domain Model می‌باشد. **به عنوان مثال :**

Order، OrderItem و Address به هم مرتبط هستند بنابراین باید در یک Agregregate باشند.

AggregateRoot چیست؟

هر Agregregate باید یک Root داشــته باشــد تا کلاس‌های بیرونی برای اســتفاده از این کلاس‌های داخلی، فقط با آن در ارتباط باشـــند پس ما در اینجا Order را به عنوان AggregateRoot این ســـه کلاس در نظر می‌گیریم.

اما این Pattern چه مشکلی از ما را حل می‌کند؟

۱) هر Aggregate تعدادی Invariant دارد که باعث می‌شــود همیشــه Entity در وضــعیت درســت قرار گیرد.

۲) همچنین ما با این Pattern، دســترســی به Entityها را محدود و تنها با Root آن امکان‌پذیر می‌کنیم. این باعث می‌شــود تا Invariantهای درون Agregregate شــکســته نشــود و State ســیســتم در وضــعیت Invalid قرار نگیرد. **به عنوان مثال:**

اجازه درج OrderItemی که هنوز برای آن Orderی ثبت نشـــده را نمی‌دهیم و ما تنها درون Order می‌توانیم Itemهای آن را ثبت کنیم. این کار باعث می‌شود تا نگهداری اپلیکیشن ساده‌تر شود.

۳) حفظ Consistency : این یعنی، داده‌ها همیشه باید یکپارچه باشند. **به طور مثال:**
برای Orderی که وجود ندارد Item زده نشود.

خب حالا که متوجه مزایای این Pattern شدید باید کلاس‌های Agregregate را با یک اینترفیس علامت بزنیم.

```
namespace Ordering.Domain.SeedWork
{
    public interface IAggregateRoot { }
}
```

کدهای اینترفیس IUnitOfWork :

Unit of Work پترنی اســـت که معمولا با Repository Pattern اســـتفاده می‌شـــود. این دو پترن لایه‌ی Infrastructure را کپسوله می‌کنند و باعث جداسازی این لایه از لایه‌های Domain Model و API می‌شوند.

www.takbook.com

```
using System;
using System.Threading;
using System.Threading.Tasks;

namespace Ordering.Domain.SeedWork
{
    public interface IUnitOfWork : IDisposable
    {
       Task<int> SaveChangesAsync(CancellationToken cancellationToken =
       default(CancellationToken));

       Task<bool> SaveEntitiesAsync(CancellationToken cancellationToken =
       default(CancellationToken));
    }
}
```

کدهای کلاس Enumeration :

Enumeration یک Wrapper بر روی نوع Integer است که با استفاده از آن می‌توانید مقادیر کاربر را محدود کنید. برای مثال کاربر فقط می‌تواند مقادیر پایین را بپذیرد:

- **1= Submitted**
- **2= Awaiting**
- **3= Validation**
- **4= Paid**
- **5= Shipped**
- **6= Cancelled**

این Pattern همانند Enum عمل می‌کند با این تفاوت که :

زمانیکه از Enum استفاده می‌کنید باید بعضی اعتبارسنجی‌ها را به صورت دستی به کد خود اضافه نمایید، که این کار باعث ایجاد Code Smell می‌شود. اما این پترن قابلیت اعتبارسنجی را در کلاس‌های فرزند کپسوله می‌کند.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Reflection;

namespace Ordering.Domain.SeedWork
{
    public abstract class Enumeration : IComparable
    {
```

www.takbook.com

```
public string Name { get; private set; }

public int Id { get; private set; }

protected Enumeration(int id, string name)
{
    Id = id;
    Name = name;
}

public override string ToString() => Name;

public static IEnumerable<T> GetAll<T>() where T : Enumeration
 {
     var fields = typeof(T).GetFields(BindingFlags.Public |
    BindingFlags.Static | BindingFlags.DeclaredOnly);

    return fields.Select(f => f.GetValue(null)).Cast<T>();
}

public override bool Equals(object obj)
{
    var otherValue = obj as Enumeration;

    if (otherValue == null)
        return false;

    var typeMatches = GetType().Equals(obj.GetType());
    var valueMatches = Id.Equals(otherValue.Id);

    return typeMatches && valueMatches;
}

public override int GetHashCode() => Id.GetHashCode();

public static int AbsoluteDifference(Enumeration firstValue,
  Enumeration secondValue)
 {
     var absoluteDifference = Math.Abs(firstValue.Id -
     secondValue.Id);
    return absoluteDifference;
}
```

www.takbook.com

```
        public static T FromValue<T>(int value) where T : Enumeration
        {
          var matchingItem = Parse<T, int>(value, "value", item => item.Id
          == value);
            return matchingItem;
        }

        public static T FromDisplayName<T>(string displayName) where T :
        Enumeration
        {
            var matchingItem = Parse<T, string>(displayName, "display name",
            item => item.Name == displayName);
            return matchingItem;
        }

         private static T Parse<T, K>(K value, string description, Func<T,
         bool> predicate) where T : Enumeration
        {
            var matchingItem = GetAll<T>().FirstOrDefault(predicate);

            if (matchingItem == null)
                throw new InvalidOperationException($"'{value}' is not a
                valid {description} in {typeof(T)}");

            return matchingItem;
        }

        public int CompareTo(object other) =>
        Id.CompareTo(((Enumeration)other).Id);
    }

}
```

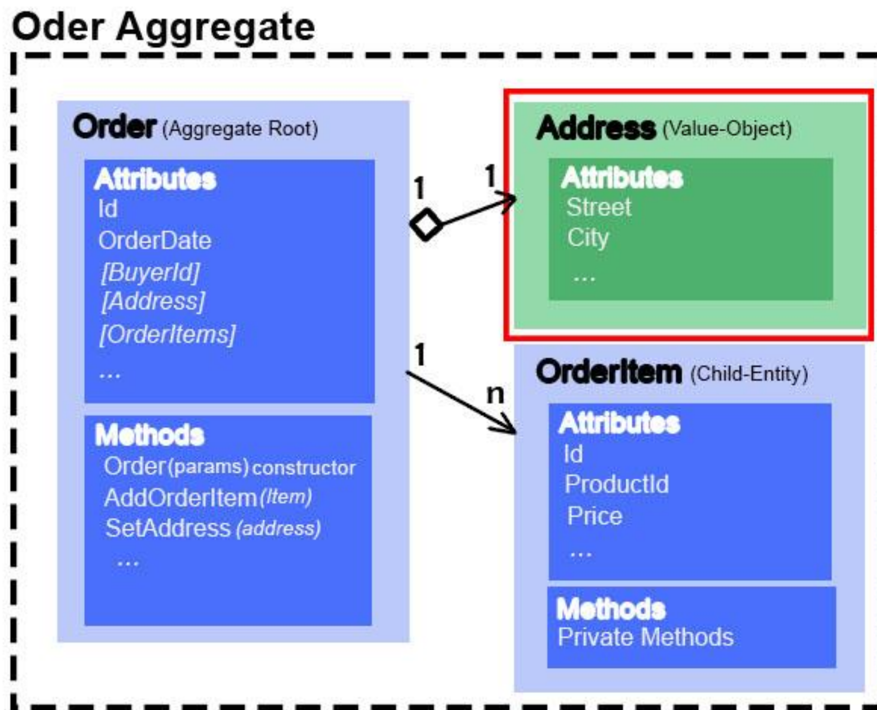
کدهای کلاس ValueObject :

همانطور که بالاتر گفته شد Id برای Entityها الزامیست اما با این حال در سیستم آبجکت‌هایی هم وجود دارند که نیاز به Identity ندارند و Immutable هستند یعنی نباید در طول عمر آبجکت تغییر کنند.

این آبجکت‌ها توسط مقادیر Propertyهایشان شناخته می‌شوند بنابراین اگر دو آبجکت از نوع Entity دارای Id یکسان باشند این دو آبجکت با هم برابر هستند اما در Value Objectها اگر دو آبجکت تمام مقادیر Propertyهایشان با هم یکسان باشند این دو آبجکت با هم برابر هستند. **برای مثال:**

www.takbook.com

Address می‌توانند در قالب یک Value Object طراحی شـــود چون هیچگاه دو آدرس با هم یکی نیست مگر اینکه تمام مشخصه‌های آن‌ها با هم یکی باشند.




```
using System.Collections.Generic;
using System.Linq;

namespace Ordering.Domain.SeedWork
{
    public abstract class ValueObject
    {
       protected static bool EqualOperator(ValueObject left, ValueObject
       right)
        {
            if (ReferenceEquals(left, null) ^ ReferenceEquals(right, null))
            {
                return false;
            }
            return ReferenceEquals(left, null) || left.Equals(right);
        }

        protected static bool NotEqualOperator(ValueObject left, ValueObject
         right)
        {
```

www.takbook.com

```
    return !(EqualOperator(left, right));
}

protected abstract IEnumerable<object> GetAtomicValues();

public override bool Equals(object obj)
{
    if (obj == null || obj.GetType() != GetType())
    {
        return false;
    }

    ValueObject other = (ValueObject)obj;

     IEnumerator<object> thisValues =
     GetAtomicValues().GetEnumerator();

    IEnumerator<object> otherValues =
    other.GetAtomicValues().GetEnumerator();

    while (thisValues.MoveNext() && otherValues.MoveNext())
    {
        if (ReferenceEquals(thisValues.Current, null) ^
        ReferenceEquals(otherValues.Current, null))
        {
            return false;
        }

        if (thisValues.Current != null &&
        !thisValues.Current.Equals(otherValues.Current))
        {
            return false;
        }
    }
    return !thisValues.MoveNext() && !otherValues.MoveNext();
}

public override int GetHashCode()
{
    return GetAtomicValues()
     .Select(x => x != null ? x.GetHashCode() : 0)
     .Aggregate((x, y) => x ^ y);
}
```

www.takbook.com

```
        public ValueObject GetCopy()
        {
            return this.MemberwiseClone() as ValueObject;
        }
    }

}
```

افزودن Domain Model

در طراحی Domain دو اصطلاح رایج وجود دارد :

۱) **Anemic Domain Model** : این Domain Model بر روی State آبجکت‌ها تمرکز کرده و شامل تعدادی Property است. در حقیقت این مدل بدون Behavior طراحی می‌شود پس می‌توان گفت که این مدل طراحی، شیءگرا نیست و مزایایی واقعی یک Domain Model را ارائه نمی‌دهد. این مدل مناسب برای یک اپلیکیشن ساده CRUD است.

۲) **Rich Domain Model** : برعکس مدل بالا این مدل بر روی Behaviorها و Business Logic تمرکز دارد بنابراین می‌تواند هنگام دستیابی به هر مجموعه‌ای، اعتبارسنجی‌ها، متغیرها و قوانینی را اعمال کنید. این مدل یک طراحی شیءگرا است و مزایایی واقعی یک Domain Model را ارائه می‌دهد. اگر قصد ایجاد یک اپلیکیشن پیچیده‌تر را دارید حتما از این مدل استفاده کنید.

بنابراین از آنجایی که ما می‌خواهیم یک سیستم CQRS طراحی کنیم Rich Domain Model انتخاب مناسب‌تری است.

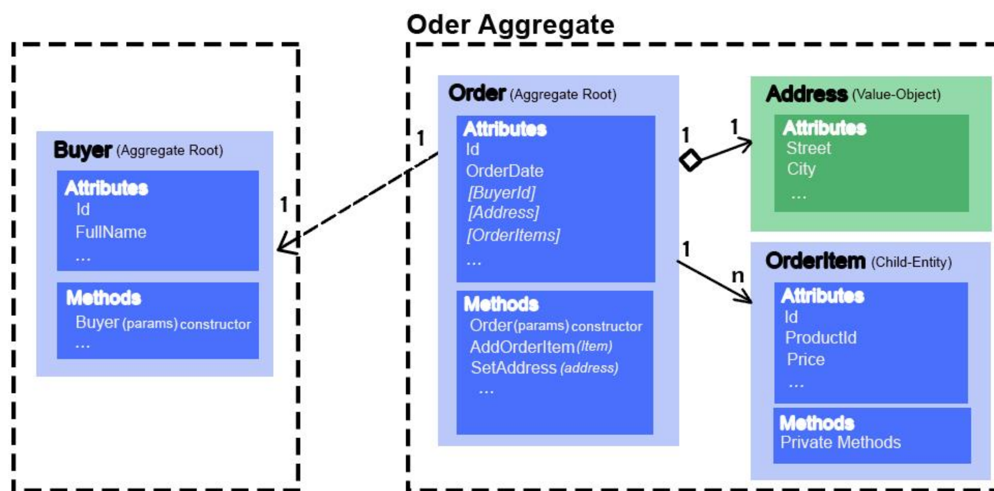
خب قدم بعدی ایجاد Domain Modelهای اپلیکیشن است بنابراین یک فولدر به نام AggregatesModel ایجاد و سپس درون این فولدر دو فولدر دیگر با نام‌های OrderAggregate و BuyerAggregate اضافه نمایید.

قبل از ایجاد کلاس‌های درون این فولدرها، می‌خواهم چند اصطلاح را توضیح دهم :

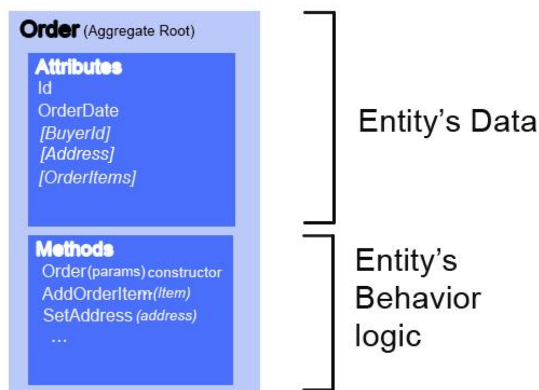
۱) **Aggregate** : همانطور که بالاتر اشاره کردم Aggregate به مجموعه ای از Entityهای مرتبط گفته می‌شود که جهت کنترل تغییرات، درون یک Entity جمع می‌شوند.

www.takbook.com

هر Aggregate یک Aggregate Root دارد که یکی از Entityهای Aggregate می‌باشـد و هدف اصـلی آن ضـمانت Consistency درون Aggregate اسـت. این Entity نقطه‌ی ورودی برای آپدیت Aggregate می‌باشد و تنها شیء است که اشیاء بیرونی می‌توانند به آن دسترسی داشته باشند.



Entity : Entity به اشـیایی گفته می‌شـود که Id داشـته باشـند. Entity یک کلاس است که تعدادی Property و Behavior دارد و از آنجاییکه پایه و اسـاس یک مدل، Entityها می‌باشـند بنابراین باید با دقت شـناسـایی و طراحی شوند.



۲) **Repository Pattern :** Repository واسـطی بین Domain و لایه‌ی Data Access اسـت یا به عبارتی می‌توان گفت: Repository پترنی اسـت که همه‌ی ارتباطات با دیتابیس را کپسـوله می‌کند و باعث می‌شود :

www.takbook.com

- **دسترسی به داده‌ها متمرکز شود.**
- **نگهداری کد ساده‌تر شود.**
- **دسترسی به دیتابیس از Domain Model جدا شود.**

برای پیاده‌سازی این پترن، ما براساس اصل Separated Interface Pattern، اینترفیس و پیاده‌سازی Repository را از هم جدا می‌کنیم تا کلاینت وابسته پیاده‌سازی نشود.

و از آنجاییکه لایه‌ی API تنها باید به نیازمندی‌های تعریف شده در Domain Model وابستگی مستقیم داشته باشد و نباید به طور مستقیم وابسته به لایه‌ی Infrastructure شود بنابراین ما اینترفیس Repository را در Domain و پیاده‌سازی آن را به لایه‌ی Infrastructure محول می‌کنیم. **به عنوان مثال :**

اینترفیس IOrderRepository را در لایه Domain تعریف می‌کنیم و OrderRepository را به لایه Infrastructure می‌سپاریم.

خب حالا بیایید کلاس‌های موردنظر را ایجاد کنیم :

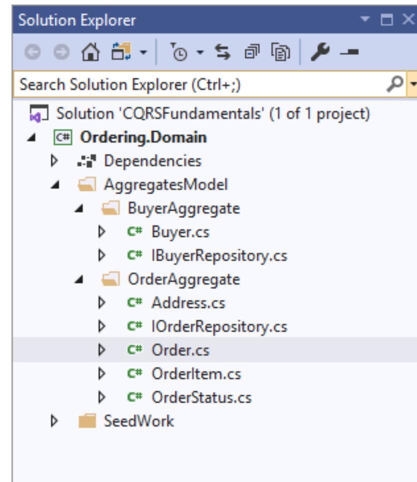
فولدر OrderAggregate شامل کلاس‌های پایین است:

- **یک AggregateRoot به نام Order**
- **یک Entity به نام OrderItem**
- **یک ValueObject به نام Address**
- **یک Enumeration به نام OrderStatus**
- **یک اینترفیس به نام IOrderRepository**

فولدر BuyerAggregate ما شامل کلاس‌های پایین است:

- **یک AggregateRoot به نام Buyer**
- **یک اینترفیس به نام IBuyerRepository**

www.takbook.com



کدهای کلاس Address :

Value Objectها یکی از مهمترین ابزارهایی هســـتند که در ســـاختن یک Rich Domain Model با Encapsulation بالا به شما کمک می‌کنند.

همانطور که بالاتر گفته شـد درون یک Aggregate ممکن اسـت ما تعدادی Value Object داشـته باشـیم که براساس تعریف، این آبجکت‌ها نباید Id داشـته باشـند. در این سـیسـتم Address یک Value Object است که اطلاعات Country/Region, Street, City و... را در خود نگه می دارد و Id هم ندارد.

یکی از ویژگی‌های مهم Value Object این اسـت که طول عمر آن‌ها وابسـته به Entityی اسـت که درون آن قرار گرفته و در اصل آن‌ها نمی‌توانند به تنهایی زندگی کنند و همیشـه باید متعلق به یکی از Entityهای موجود در برنامه باشند.

```
using Ordering.Domain.SeedWork;
using System.Collections.Generic;

namespace Ordering.Domain.AggregatesModel.OrderAggregate
{
    public class Address : ValueObject
    {
        public string Street { get; private set; }
        public string City { get; private set; }
        public string Country { get; private set; }
        public string ZipCode { get; private set; }

        public Address() { }
```

www.takbook.com

```
        public Address(string street, string city, string country, string
         zipcode)
         {
            Street = street;
            City = city;
            Country = country;
            ZipCode = zipcode;
        }

        protected override IEnumerable<object> GetAtomicValues()
        {
            yield return Street;
            yield return City;
            yield return Country;
            yield return ZipCode;
        }
    }

}
```

کدهای کلاس OrderStatus:

این کلاس مانند یک Enum عمل می‌کند و برای وضعیت سفارش اجازه ورود اعداد ۱ تا ۶ را می‌دهد.

```
using Ordering.Domain.SeedWork;

public class OrderStatus : Enumeration
 {
     public static OrderStatus Submitted = new OrderStatus(1,
     nameof(Submitted).ToLowerInvariant());

     public static OrderStatus AwaitingValidation = new OrderStatus(2,
     nameof(AwaitingValidation).ToLowerInvariant());

     public static OrderStatus StockConfirmed = new OrderStatus(3,
     nameof(StockConfirmed).ToLowerInvariant());

     public static OrderStatus Paid = new OrderStatus(4,
     nameof(Paid).ToLowerInvariant());

     public static OrderStatus Shipped = new OrderStatus(5,
     nameof(Shipped).ToLowerInvariant());
```

www.takbook.com

```
        public static OrderStatus Cancelled = new OrderStatus(6,
        nameof(Cancelled).ToLowerInvariant());

         public OrderStatus(int id, string name)
             : base(id, name)
         {
         }

    }
```

کدهای کلاس OrderItem :

این کلاس جهت ذخیره اطلاعات اقلام سفارش تعریف شده است. همانطور که بالاتر گفته شد این کلاس باید از طریق کلاس Order مورد اســـتفاده قرار گیرد اما بیزینس‌های مربوط باید از طریق متدهای درون خودش پیاده‌سازی شود.

```
using Ordering.Domain.SeedWork;
using System;

namespace Ordering.Domain.AggregatesModel.OrderAggregate
{
    public class OrderItem : Entity
    {
        private string _productName;
        private string _pictureUrl;
        private decimal _unitPrice;
        private decimal _discount;
        private int _units;

        public int ProductId { get; private set; }

        protected OrderItem() { }

        public OrderItem(int productId, string productName, decimal unitPrice,
        decimal discount, string PictureUrl, int units = 1)
        {
            if (units <= 0)
            {
                throw new Exception("Invalid number of units");
            }

            if ((unitPrice * units) < discount)
            {
```

www.takbook.com

```
            throw new Exception("The total of order item is lower than
            applied discount");
        }

         ProductId = productId;
        _productName = productName;
        _unitPrice = unitPrice;
        _discount = discount;
        _units = units;
        _pictureUrl = PictureUrl;
    }

    public string GetPictureUri() => _pictureUrl;

    public decimal GetCurrentDiscount()
    {
        return _discount;
    }

    public int GetUnits()
    {
        return _units;
    }

    public decimal GetUnitPrice()
    {
        return _unitPrice;
    }

    public string GetOrderItemProductName() => _productName;

    public void SetNewDiscount(decimal discount)
    {
        if (discount < 0)
        {
            throw new Exception("Discount is not valid");
        }

        _discount = discount;
    }

    public void AddUnits(int units)
    {
```

www.takbook.com

```
            if (units < 0)
            {
                throw new Exception("Invalid units");
            }

            _units += units;
        }
    }

}
```

نکته!!

با تعریف برخی متدها در این کلاس قابلیت کنترل و اعتبارسنجی فیلدها را در Domain اعمال کردیم. به طور مثال:

در متد SetNewDiscount قبل از اعمال تخفیف جدید باید مبلغ تخفیف اعتبارسنجی شود تا عدد زیر صفر نباشد.

کدهای کلاس Order :

این کلاس یک AggregateRoot است که آبجکت‌های Address, OrderStatus, OrderItem تنها از طریق آن قابل دسترسی هستند.

```
using Ordering.Domain.SeedWork;
using System;
using System.Collections.Generic;
using System.Linq;

namespace Ordering.Domain.AggregatesModel.OrderAggregate
{
    public class Order : Entity, IAggregateRoot
    {

        private DateTime _orderDate;

        public Address Address { get; private set; }

        public int? GetBuyerId => _buyerId;
        private int? _buyerId;

        public OrderStatus OrderStatus { get; private set; }
        private int _orderStatusId;
```

www.takbook.com

```
private string _description;

private bool _isDraft;

private readonly List<OrderItem> _orderItems;
public IReadOnlyCollection<OrderItem> OrderItems => _orderItems;

public static Order NewDraft()
{
    var order = new Order();
    order._isDraft = true;
    return order;
}

protected Order()
{
    _orderItems = new List<OrderItem>();
    _isDraft = false;
}

public Order(string userId, string userName, Address address,
int? buyerId = null) : this()
{
    _buyerId = buyerId;
    _orderStatusId = OrderStatus.Submitted.Id;
    _orderDate = DateTime.UtcNow;
    Address = address;

}

public void AddOrderItem(int productId, string productName, decimal
unitPrice,  decimal discount, string pictureUrl, int units = 1)
{
  var existingOrderForProduct = _orderItems.Where(o => o.ProductId ==
  productId).SingleOrDefault();

    if (existingOrderForProduct != null)
    {

        if (discount > existingOrderForProduct.GetCurrentDiscount())
        {
            existingOrderForProduct.SetNewDiscount(discount);
        }
```

```
            existingOrderForProduct.AddUnits(units);
        }
        else
        {

            var orderItem = new OrderItem(productId, productName, unitPrice,
            discount, pictureUrl, units);
            _orderItems.Add(orderItem);
        }
    }

    public void SetBuyerId(int id)
    {
        _buyerId = id;
    }

    public void SetAwaitingValidationStatus()
    {
        if (_orderStatusId == OrderStatus.Submitted.Id)
        {
            _orderStatusId = OrderStatus.AwaitingValidation.Id;
        }
    }

    public void SetStockConfirmedStatus()
    {
        if (_orderStatusId == OrderStatus.AwaitingValidation.Id)
        {
            _orderStatusId = OrderStatus.StockConfirmed.Id;
            _description = "All the items were confirmed with available
             stock.";
        }
    }

    public void SetPaidStatus()
    {
        if (_orderStatusId == OrderStatus.StockConfirmed.Id)
        {
            _orderStatusId = OrderStatus.Paid.Id;
            _description = "The payment was performed at a simulated
          \"American Bank checking bank account ending on XX35071\"";
        }
```

www.takbook.com

```
}

public void SetShippedStatus()
{
    if (_orderStatusId != OrderStatus.Paid.Id)
    {
        StatusChangeException(OrderStatus.Shipped);
    }

    _orderStatusId = OrderStatus.Shipped.Id;
    _description = "The order was shipped.";
}

public void SetCancelledStatus()
{
    if (_orderStatusId == OrderStatus.Paid.Id ||
        _orderStatusId == OrderStatus.Shipped.Id)
    {
         StatusChangeException(OrderStatus.Cancelled);
    }

    _orderStatusId = OrderStatus.Cancelled.Id;
    _description = $"The order was cancelled.";
}

public  void  SetCancelledStatusWhenStockIsRejected(IEnumerable<int>
orderStockRejectedItems)
 {
    if (_orderStatusId == OrderStatus.AwaitingValidation.Id)
    {
        _orderStatusId = OrderStatus.Cancelled.Id;

        var itemsStockRejectedProductNames = OrderItems
             .Where(c =>
             orderStockRejectedItems.Contains(c.ProductId))
            .Select(c => c.GetOrderItemProductName());

      var itemsStockRejectedDescription = string.Join(", ",
      itemsStockRejectedProductNames);
       _description  =  $"The  product  items  don't  have  stock:
      ({itemsStockRejectedDescription}).";
    }
}
```

www.takbook.com

```
        private void StatusChangeException(OrderStatus orderStatusToChange)
        {
          throw new Exception($"Is not possible to change the order status from
          {OrderStatus.Name} to {orderStatusToChange.Name}.");
        }

        public decimal GetTotal()
        {
            return _orderItems.Sum(o => o.GetUnits() * o.GetUnitPrice());
        }
    }

}
```

نکته!!

اگر می‌خواهید پراپرتی در خارج از Entity قابل دسترسی نباشد می‌توانید از فیلد Private استفاده کنید یا اینکه سطح دسترسی Setter را Private نمایید. به طور مثال:

همانطور که در کلاس بالا می‌بینید:

ما یک فیلد orderItems_ داریم که به صورت Private تعریف شده بنابراین بیرون از کلاس قابل دسترس نیست. و همچنین یک پراپرتی OrderItems داریم که شامل مقدار این فیلد است. این پراپرتی بیرون از کلاس قابل دسترس است اما به صورت ReadOnly و ما برای اضافه کردن یک آیتم به این فیلد orderItems_ باید از متد AddOrderItem کمک بگیریم.

کدهای کلاس Buyer :

```
using Ordering.Domain.SeedWork;
using System;

namespace Ordering.Domain.AggregatesModel.BuyerAggregate
{
    public class Buyer : Entity, IAggregateRoot
    {
        public string IdentityGuid { get; private set; }

        public string Name { get; private set; }

        protected Buyer()
        {

        }
```

www.takbook.com

```
        public Buyer(string identity, string name) :this()
        {
            IdentityGuid = !string.IsNullOrWhiteSpace(identity) ? identity :
           throw new ArgumentNullException(nameof(identity));

            Name = !string.IsNullOrWhiteSpace(name) ? name : throw new
          ArgumentNullException(nameof(name));
        }

    }

}
```

کدهای اینترفیس IOrderRepository :

Repository مثل لیستی است که می‌خواهید در آن متدهایی برای درج، حذف و.. داشته باشید.

بنابراین ما این عملیات را در یک اینترفیس IRepositoryX می‌گذاریم و در لایه بعدی این عملیات را پیاده‌سازی می‌کنیم.

```
using Ordering.Domain.SeedWork;
using System.Threading.Tasks;

namespace Ordering.Domain.AggregatesModel.OrderAggregate
{
    public interface IOrderRepository
    {
    IUnitOfWork UnitOfWork { get; }
        Order Add(Order order);
        void Update(Order order);
        Task<Order> GetAsync(int orderId);
    }
}
```

کدهای اینترفیس IBuyerRepository :

```
using Ordering.Domain.SeedWork;
using System.Threading.Tasks;

namespace Ordering.Domain.AggregatesModel.BuyerAggregate
{
    public interface IBuyerRepository
    {
```

www.takbook.com

```
    IUnitOfWork UnitOfWork { get; }
        Buyer Add(Buyer buyer);
        Buyer Update(Buyer buyer);
        Task<Buyer> FindAsync(string BuyerIdentityGuid);
        Task<Buyer> FindByIdAsync(string id);
    }
}
```

www.takbook.com

فصل سوم : ایجاد لایه‌ی Infrastructure

آنچه خواهید آموخت:

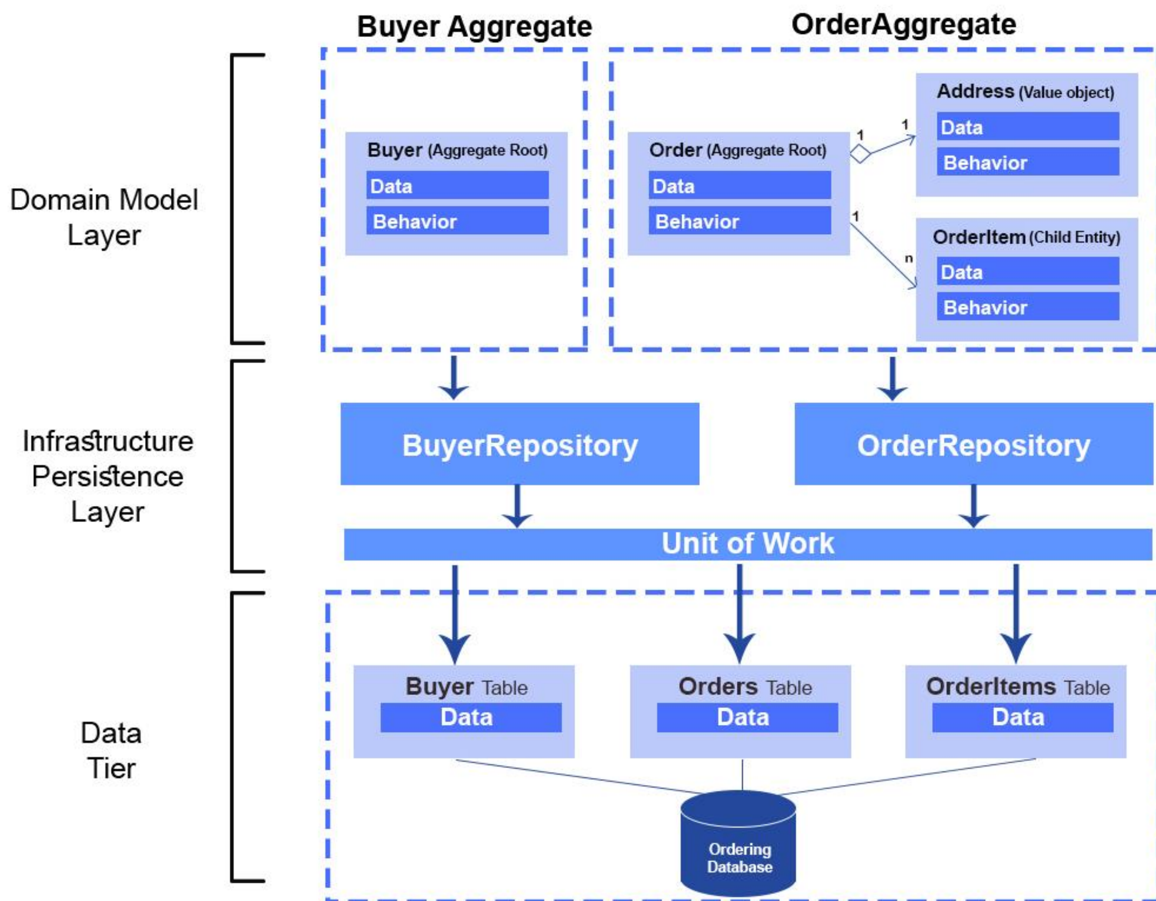
- افزودن لایه‌ی Infrastructure
- Table Mapping چیست؟
- پیاده‌سازی Repository

www.takbook.com

افزودن لایه‌ی Infrastructure

خب مرحله اول این اپلیکیشـن با موفقیت انجام شـد حالا باید وارد لایه‌ی Infrastructure شـویم. ما در لایه‌ی Domain تعدادی Entity ایجاد کردیم و ساختار Domain Model خود را ساختیم اما این Entityها به تنهایی کاری انجام نمی‌دهند و ما باید با دیتابیس ارتباط برقرار کنیم.

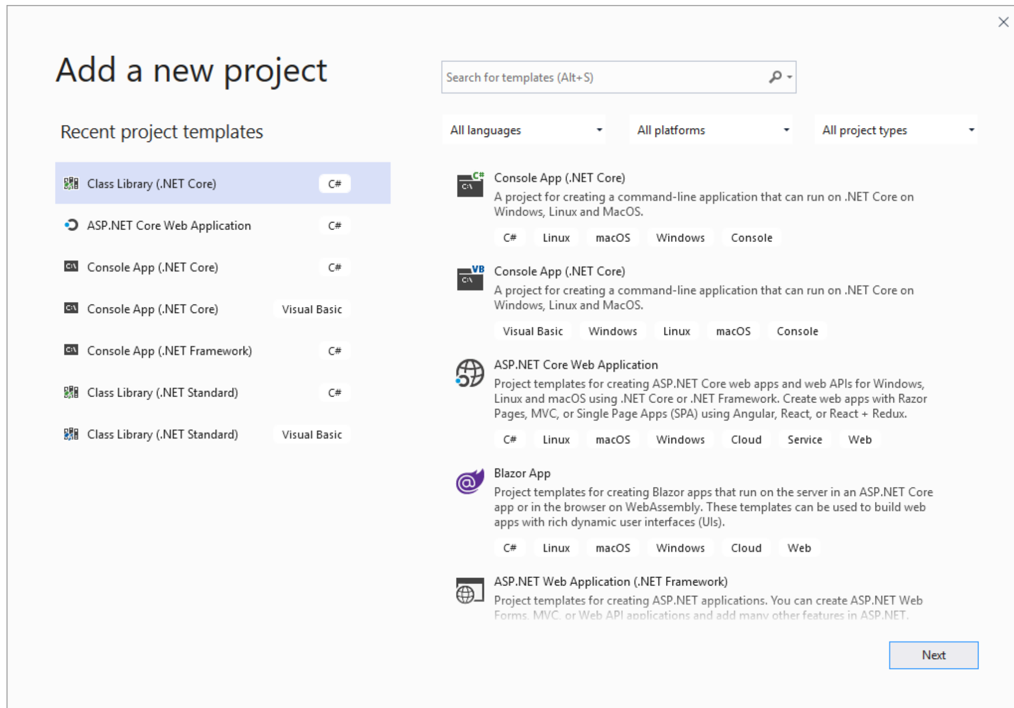
همانطور که بالاتر گفتیم داده‌هایی که در Domain Entityهای ما قرار گرفته باید توسـط لایه‌ای دیگر در دیتابیس ذخیره شـوند بنابراین ما باید یک لایه‌ی Infrastructure ایجاد کنیم تا زیرسـاخت‌های داده‌ای و تکنولوژی محور نرم‌افزار را در آن پیاده‌سازی نماییم.



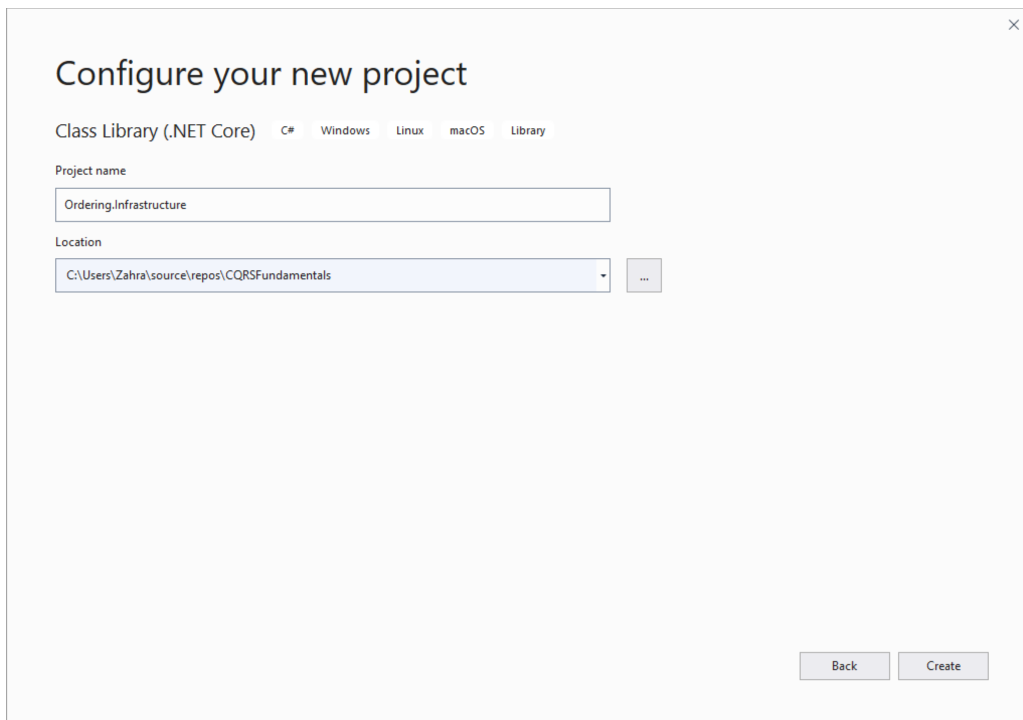
پس بیایید این لایه را ایجاد کنیم.

- **بر روی Solution راست کلیک و سپس Add new Project را انتخاب نمایید.**

www.takbook.com



- حالا Class Library (.NET Core) را انتخاب و بر روی **Next** کلیک کنید.
- در کادر بعدی نام پروژه را **Ordering.Infrastructure** بگذارید و بر روی **Create** کلیک نمایید.



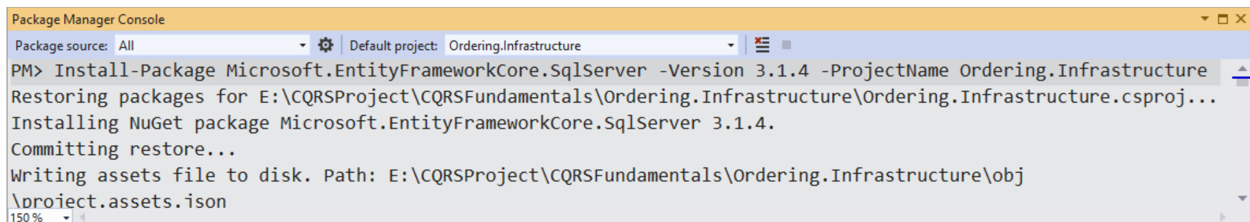
www.takbook.com

هدف این لایه، اتصال به دیتابیس و کار با دیتاست بنابراین برای این اتصال ما نیاز به انتخاب یک ORM داریم. ORM پیشـــنهادی این کتاب اســـتفاده از Entity Framework Core اســـت چون این ORM : ســـبک، قابل توسعه و از همه مهمتر Performance بالایی دارد.

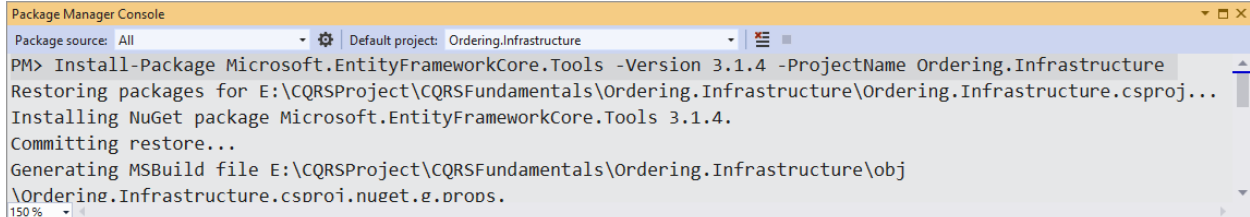
پس برای استفاده از این ORM باید پکیج‌های زیر را به پروژه اضافه نمایید.

وارد مســـیر Tools→NuGet Package Manager→Package Manager Console شـــوید و دستورات پایین را یکی یکی اجرا کنید.

```
Install-Package Microsoft.EntityFrameworkCore.SqlServer -Version 3.1.4 -
ProjectName Ordering.Infrastructure
```

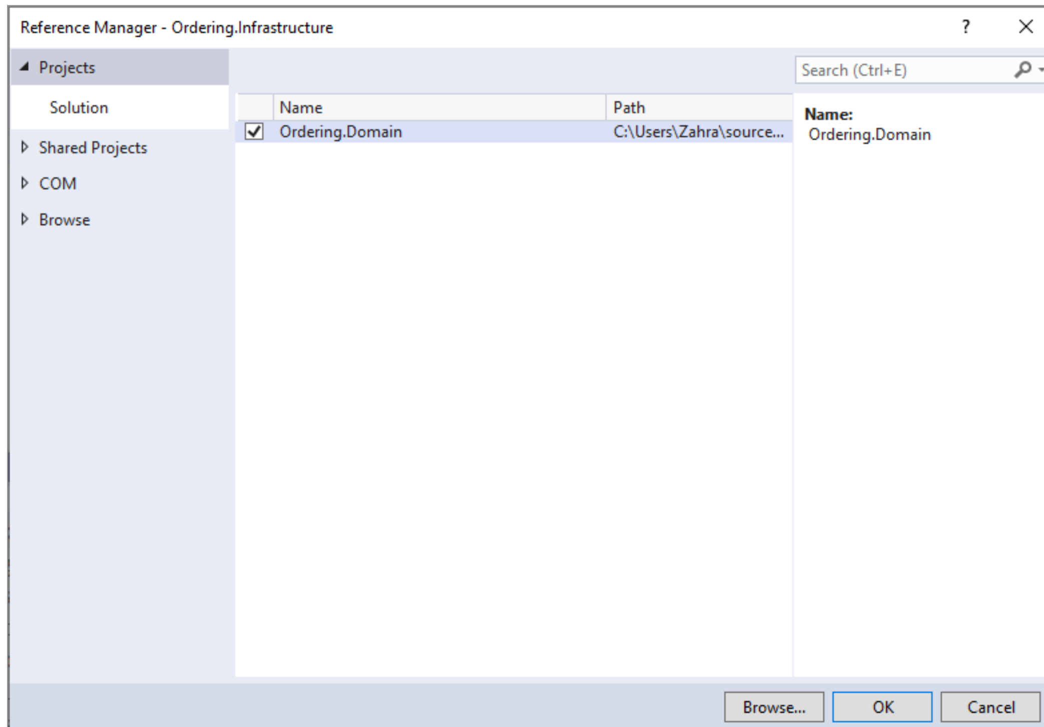


```
Install-Package Microsoft.EntityFrameworkCore.Tools -Version 3.1.4 -
ProjectName Ordering.Infrastructure
```



- **مرحله بعد گرفتن‌یک Reference از پروژه Domain اســـت. پس روی Dependencies راســـت کلیک و سپس Add Reference را انتخاب کنید.**

www.takbook.com



- **حالا پروژه Domain را انتخاب و سپس بر روی OK کلیک کنید.**

پس از Reference به پروژه Domain باید DbContext خود را تعریف کنیم. بنابراین به ریشه پروژه یک کلاس به نام OrderingContext اضافه و سپس کدهای پایین را در آن قرار دهید.

```
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Storage;
using Ordering.Domain.AggregatesModel.BuyerAggregate;
using Ordering.Domain.AggregatesModel.OrderAggregate;
using Ordering.Domain.SeedWork;
using System;
using System.Data;
using System.Threading;
using System.Threading.Tasks;
namespace Ordering.Infrastructure
{
    public class OrderingContext : DbContext , IUnitOfWork
    {
        public const string DEFAULT_SCHEMA = "Ordering";
   private IDbContextTransaction _currentTransaction;
        public DbSet<Order> Orders { get; set; }
        public DbSet<OrderItem> OrderItems { get; set; }
        public DbSet<Buyer> Buyers { get; set; }
        public DbSet<OrderStatus> OrderStatus { get; set; }
```

www.takbook.com

```
public OrderingContext(DbContextOptions<OrderingContext> options) :
base(options) { }

public IDbContextTransaction GetCurrentTransaction() =>
_currentTransaction;

public bool HasActiveTransaction => _currentTransaction != null;
public async Task<IDbContextTransaction> BeginTransactionAsync()
{
    if (_currentTransaction != null) return null;

      _currentTransaction = await
      Database.BeginTransactionAsync(IsolationLevel.ReadCommitted);

    return _currentTransaction;
}

 public async Task CommitTransactionAsync(IDbContextTransaction
 transaction)
{
     if (transaction == null) throw new
     ArgumentNullException(nameof(transaction));

    if (transaction != _currentTransaction) throw new
     InvalidOperationException($"Transaction
     {transaction.TransactionId} is not current");

    try
    {
        await SaveChangesAsync();
        transaction.Commit();
    }
    catch
    {
        RollbackTransaction();
        throw;
    }
    finally
    {
        if (_currentTransaction != null)
        {
```

www.takbook.com

```
                _currentTransaction.Dispose();
                _currentTransaction = null;
            }
        }
    }

    public void RollbackTransaction()
    {
        try
        {
            _currentTransaction?.Rollback();
        }
        finally
        {
            if (_currentTransaction != null)
            {
                _currentTransaction.Dispose();
                _currentTransaction = null;
            }
        }
    }


     public async Task<bool> SaveEntitiesAsync(CancellationToken
     cancellationToken = default(CancellationToken))
    {
        var result = await base.SaveChangesAsync(cancellationToken);

        return true;
    }
  }

}
```

نکته!!

برخی متدهای بالا مربوط به عملیات Transaction است.

Table mapping چیست؟

در کد بالا دیدید که چطور با اســـتفاده از EF Core می‌توان یک دیتابیس را از روی Domain Entityها ایجاد کرد. EF Core تعدادی Convention دارد که برخی عملیات مثل : مشخص شدن کلید اصلی، نام جدول، نوع ستون‌های جداول و... را مشخص می‌کند. **برای مثال :**

www.takbook.com

طبق Convention : در Domain Entity هر پراپرتی که نامش Id باشد به عنوان کلید اصلی درنظر گرفته می‌شود.

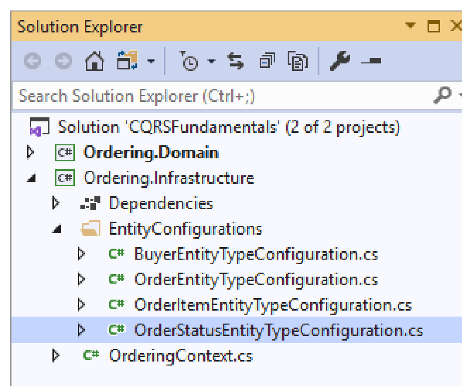
برخی از این Conventionها به صورت معمول به Domain Entityها اضافه می‌شوند(مثل مشخص شدن کلید اصلی) اما برخی از آن‌ها را باید از طریق Data Annotationها یا اضافه کردن Fluent API در متد OnModelCreating کلاس DbContext سفارشی نمود.

Data Annotationها را باید به کلاس‌های Domain Entity اضافه کنیم اما این کار باعث می‌شود زیرساخت‌های دیتابیس وارد Domain Entity شود بنابراین Fluent API روش بهتری است چون این زیرساخت‌ها درون همان کلاس DBContext انجام می‌شود.

تا اینجا متوجه شدیم که بهتر است Convention مربوط به هر Entity را در متد OnModelCreating اضافه کنیم اما از آنجایی که تعداد این Conventionها زیاد است و باعث شلوغی و بهم ریختگی در این متد می‌شود، بهتر است که Convention هر Entity را در یک کلاس جداگلنه قرار داد و سپس این کلاس‌ها را در متد OnModelCreating اضافه نمود.

پس بیایید برای هر Entity یک کلاس Configuration ایجاد کنیم.

در ریشه پروژه یک فولدر به نام EntityConfigurations ایجاد و سپس همانند تصویر پایین کلاس‌های موردنظر را اضافه نمایید.



Configuration برای کلاس Buyer :

```
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Metadata.Builders;
using Ordering.Domain.AggregatesModel.BuyerAggregate;

namespace Ordering.Infrastructure.EntityConfigurations
```

www.takbook.com

```
{
    class BuyerEntityTypeConfiguration : IEntityTypeConfiguration<Buyer>
    {
        public void Configure(EntityTypeBuilder<Buyer> buyerConfiguration)
        {
            buyerConfiguration.ToTable("Buyers",
            OrderingContext.DEFAULT_SCHEMA);

            buyerConfiguration.HasKey(b => b.Id);

             buyerConfiguration.Property(b => b.Id)
               .UseHiLo("buyerseq", OrderingContext.DEFAULT_SCHEMA);

            buyerConfiguration.Property(b => b.Id)
                .UseHiLo("buyerseq", OrderingContext.DEFAULT_SCHEMA);

            buyerConfiguration.Property(b => b.IdentityGuid)
                .HasMaxLength(200)
                .IsRequired();

            buyerConfiguration.HasIndex("IdentityGuid")
              .IsUnique(true);

            buyerConfiguration.Property(b => b.Name);

        }
    }

}
```

Configuration برای کلاس Order :

```
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Metadata.Builders;
using Ordering.Domain.AggregatesModel.BuyerAggregate;
using Ordering.Domain.AggregatesModel.OrderAggregate;
using System;

namespace Ordering.Infrastructure.EntityConfigurations
{
    class OrderEntityTypeConfiguration : IEntityTypeConfiguration<Order>
    {
        public void Configure(EntityTypeBuilder<Order> orderConfiguration)
        {
            orderConfiguration.ToTable("Orders",
            OrderingContext.DEFAULT_SCHEMA);
```

www.takbook.com

```
orderConfiguration.HasKey(o => o.Id);

orderConfiguration.Property(o => o.Id)
    .UseHiLo("orderseq", OrderingContext.DEFAULT_SCHEMA);

orderConfiguration
    .OwnsOne(o => o.Address, a =>
    {
        a.WithOwner();
    });

orderConfiguration
    .Property<int?>("_buyerId")
    .UsePropertyAccessMode(PropertyAccessMode.Field)
    .HasColumnName("BuyerId")
    .IsRequired(false);

orderConfiguration
    .Property<DateTime>("_orderDate")
    .UsePropertyAccessMode(PropertyAccessMode.Field)
    .HasColumnName("OrderDate")
    .IsRequired();

orderConfiguration
    .Property<int>("_orderStatusId")
    .UsePropertyAccessMode(PropertyAccessMode.Field)
    .HasColumnName("OrderStatusId")
    .IsRequired();

orderConfiguration.Property<string>("Description").IsRequired(fal
se);

var  navigation  =
orderConfiguration.Metadata.FindNavigation(nameof(Order.OrderItem
s));


navigation.SetPropertyAccessMode(PropertyAccessMode.Field);

orderConfiguration.HasOne<Buyer>()
    .WithMany()
    .IsRequired(false)
    // .HasForeignKey("BuyerId");
    .HasForeignKey("_buyerId");
```

www.takbook.com

```
            orderConfiguration.HasOne(o => o.OrderStatus)
                .WithMany()
                // .HasForeignKey("OrderStatusId");
                .HasForeignKey("_orderStatusId");
        }
    }
}
```

Configuration برای کلاس OrderItem:

```
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Metadata.Builders;
using Ordering.Domain.AggregatesModel.OrderAggregate;

namespace Ordering.Infrastructure.EntityConfigurations
{
    class OrderItemEntityTypeConfiguration
     : IEntityTypeConfiguration<OrderItem>
    {
          public void Configure(EntityTypeBuilder<OrderItem>
          orderItemConfiguration)
         {
          orderItemConfiguration.ToTable("OrderItems",
          OrderingContext.DEFAULT_SCHEMA);

            orderItemConfiguration.HasKey(o => o.Id);

            orderItemConfiguration.Property(o => o.Id)
                .UseHiLo("orderitemseq");

            orderItemConfiguration.Property(o => o.Id);

            orderItemConfiguration.Property<int>("OrderId")
                .IsRequired();

            orderItemConfiguration
                .Property<decimal>("_discount")
                .UsePropertyAccessMode(PropertyAccessMode.Field)
                .HasColumnName("Discount")
                 .HasColumnType("decimal(18,2)")
                .IsRequired();

            orderItemConfiguration.Property<int>("ProductId")
                .IsRequired();
```

www.takbook.com

```
            orderItemConfiguration
                .Property<string>("_productName")
                .UsePropertyAccessMode(PropertyAccessMode.Field)
                .HasColumnName("ProductName")
                .IsRequired();

            orderItemConfiguration
                .Property<decimal>("_unitPrice")
                .UsePropertyAccessMode(PropertyAccessMode.Field)
                .HasColumnName("UnitPrice")
                .HasColumnType("decimal(18,2)")
                .IsRequired();

            orderItemConfiguration
                .Property<int>("_units")
                .UsePropertyAccessMode(PropertyAccessMode.Field)
                .HasColumnName("Units")
                .IsRequired();

            orderItemConfiguration
                .Property<string>("_pictureUrl")
                .UsePropertyAccessMode(PropertyAccessMode.Field)
                .HasColumnName("PictureUrl")
                .IsRequired(false);
        }
    }

}
```

Configuration برای کلاس OrderStatus:

```
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Metadata.Builders;

namespace Ordering.Infrastructure.EntityConfigurations
{
    class OrderStatusEntityTypeConfiguration
      : IEntityTypeConfiguration<OrderStatus>
    {
         public void Configure(EntityTypeBuilder<OrderStatus>
         orderStatusConfiguration)
        {
            orderStatusConfiguration.ToTable("Orderstatus",
            OrderingContext.DEFAULT_SCHEMA);
```

www.takbook.com

```
            orderStatusConfiguration.HasKey(o => o.Id);

            orderStatusConfiguration.Property(o => o.Id)
                .HasDefaultValueSql("1")
                .ValueGeneratedNever()
                .IsRequired();

            orderStatusConfiguration.Property(o => o.Name)
                .HasMaxLength(200)
                .IsRequired();
        }
    }

}
```

حالا باید این کلاس‌های Configuration را در متد OnModelCreating معرفی کنیم.

```
modelBuilder.ApplyConfiguration(new OrderEntityTypeConfiguration());
modelBuilder.ApplyConfiguration(new  OrderItemEntityTypeConfiguration());
modelBuilder.ApplyConfiguration(new OrderStatusEntityTypeConfiguration());
modelBuilder.ApplyConfiguration(new BuyerEntityTypeConfiguration());
```

کدهای کلاس OrderingContext :

```
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Storage;
using Ordering.Domain.AggregatesModel.BuyerAggregate;
using Ordering.Domain.AggregatesModel.OrderAggregate;
using Ordering.Domain.SeedWork;
using Ordering.Infrastructure.EntityConfigurations;
using System;
using System.Data;
using System.Threading;
using System.Threading.Tasks;
namespace Ordering.Infrastructure
{
    public class OrderingContext : DbContext , IUnitOfWork
    {
        public const string DEFAULT_SCHEMA = "Ordering";
   private IDbContextTransaction _currentTransaction;
        public DbSet<Order> Orders { get; set; }
        public DbSet<OrderItem> OrderItems { get; set; }
        public DbSet<Buyer> Buyers { get; set; }
        public DbSet<OrderStatus> OrderStatus { get; set; }
```

www.takbook.com

```
public  OrderingContext(DbContextOptions<OrderingContext>  options)  :
base(options) { }

protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.ApplyConfiguration(new
    OrderEntityTypeConfiguration());
    modelBuilder.ApplyConfiguration(new
    OrderItemEntityTypeConfiguration());
    modelBuilder.ApplyConfiguration(new
    OrderStatusEntityTypeConfiguration());
    modelBuilder.ApplyConfiguration(new
    BuyerEntityTypeConfiguration());
}

public IDbContextTransaction GetCurrentTransaction() =>
_currentTransaction;

public bool HasActiveTransaction => _currentTransaction != null;
public async Task<IDbContextTransaction> BeginTransactionAsync()
{
    if (_currentTransaction != null) return null;

      _currentTransaction = await
      Database.BeginTransactionAsync(IsolationLevel.ReadCommitted);

    return _currentTransaction;
}

 public async Task CommitTransactionAsync(IDbContextTransaction
 transaction)
{
     if (transaction == null) throw new
     ArgumentNullException(nameof(transaction));

    if (transaction != _currentTransaction) throw new
     InvalidOperationException($"Transaction
     {transaction.TransactionId} is not current");

    try
    {
        await SaveChangesAsync();
```

www.takbook.com

```
            transaction.Commit();
        }
        catch
        {
            RollbackTransaction();
            throw;
        }
        finally
        {
            if (_currentTransaction != null)
            {
                _currentTransaction.Dispose();
                _currentTransaction = null;
            }
        }
    }

    public void RollbackTransaction()
    {
        try
        {
            _currentTransaction?.Rollback();
        }
        finally
        {
            if (_currentTransaction != null)
            {
                _currentTransaction.Dispose();
                _currentTransaction = null;
            }
        }
    }


     public async Task<bool> SaveEntitiesAsync(CancellationToken
     cancellationToken = default(CancellationToken))
    {

        var result = await base.SaveChangesAsync(cancellationToken);

        return true;
    }
}
```

www.takbook.com

```
}
```

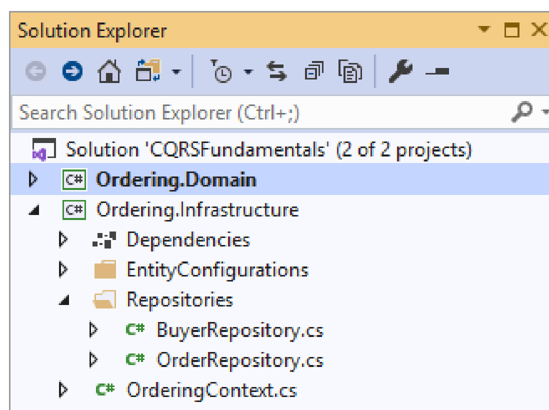
پیاده‌سازی Repository

Repository کلاسی است که منطق موردنیاز برای دسترسی به Data Source را Encapsulate می‌کند. این کلاس باعث می‌شود تکنولوژی مورد استفاده برای دسترسی به دیتابیس را از لایه‌ی Domain جدا و نگهداری کد بهتر شود.

همانطور که در فصل قبل دیدید ما در لایه‌ی Domain برای هر Aggregate Root یک اینترفیس Repository ایجاد کردیم تا لایه API به طور مستقیم به لایه‌ی Infrastructure وابسته نباشد.

ما می‌توانیم با این جداسازی و استفاده از Dependency Injection در کنترلرها، این Repositoryها را Mock کنیم و یک داده Fake به جای واکشی اطلاعات از یک دیتابیس برگردانیم.

خب الان باید در این لایه این اینترفیس‌ها را پیاده‌سازی کنیم پس یک فولدر به نام Repositories ایجاد کنید سپس دو کلاس BuyerRepository و OrderRepository را به آن اضافه نمایید.



کدهای درون کلاس OrderRepository:

```
using Microsoft.EntityFrameworkCore;
using Ordering.Domain.AggregatesModel.OrderAggregate;
using Ordering.Domain.SeedWork;
using System;
using System.Linq;
using System.Threading.Tasks;

namespace Ordering.Infrastructure.Repositories
{
    public class OrderRepository : IOrderRepository
```

www.takbook.com

```
{
    private readonly OrderingContext _context;

      public IUnitOfWork UnitOfWork
        {
            get
            {
                return _context;
            }
        }

    public OrderRepository(OrderingContext context)
    {
       _context = context ?? throw new
       ArgumentNullException(nameof(context));
    }

    public Order Add(Order order)
    {
        return _context.Orders.Add(order).Entity;
    }

    public async Task<Order> GetAsync(int orderId)
    {
        var order = await _context
                              .Orders
                              .Include(x => x.Address)
                              .FirstOrDefaultAsync(o => o.Id == orderId);
        if (order == null)
        {
            order = _context
                          .Orders
                          .Local
                          .FirstOrDefault(o => o.Id == orderId);
        }
        if (order != null)
        {
            await _context.Entry(order)
                .Collection(i => i.OrderItems).LoadAsync();
            await _context.Entry(order)
                .Reference(i => i.OrderStatus).LoadAsync();
        }
```

www.takbook.com

```
            return order;
        }

        public void Update(Order order)
        {
            _context.Entry(order).State = EntityState.Modified;
        }
    }

}
```

کدهای درون کلاس BuyerRepository :

```
using Microsoft.EntityFrameworkCore;
using Ordering.Domain.AggregatesModel.BuyerAggregate;
using Ordering.Domain.SeedWork;
using System;
using System.Linq;
using System.Threading.Tasks;

namespace Ordering.Infrastructure.Repositories
{
    public class BuyerRepository : IBuyerRepository
    {
        private readonly OrderingContext _context;
        public IUnitOfWork UnitOfWork
        {
            get
            {
                return _context;
            }
        }

        public BuyerRepository(OrderingContext context)
        {
            _context = context ?? throw new
           ArgumentNullException(nameof(context));
        }

        public Buyer Add(Buyer buyer)
        {
            if (buyer.IsTransient())
            {
                return _context.Buyers
```

www.takbook.com

```
                .Add(buyer)
                .Entity;
        }
        else
        {
            return buyer;
        }
    }

    public Buyer Update(Buyer buyer)
    {
        return _context.Buyers
                .Update(buyer)
                .Entity;
    }

    public async Task<Buyer> FindAsync(string identity)
    {
        var buyer = await _context.Buyers
            .Where(b => b.IdentityGuid == identity)
            .SingleOrDefaultAsync();

        return buyer;
    }

    public async Task<Buyer> FindByIdAsync(string id)
    {
        var buyer = await _context.Buyers
            .Where(b => b.Id == int.Parse(id))
            .SingleOrDefaultAsync();

        return buyer;
    }
  }

}
```

www.takbook.com

فصل چهارم : لایه‌ی Application و پیاده‌سازی Commmand

آنچه خواهید آموخت:

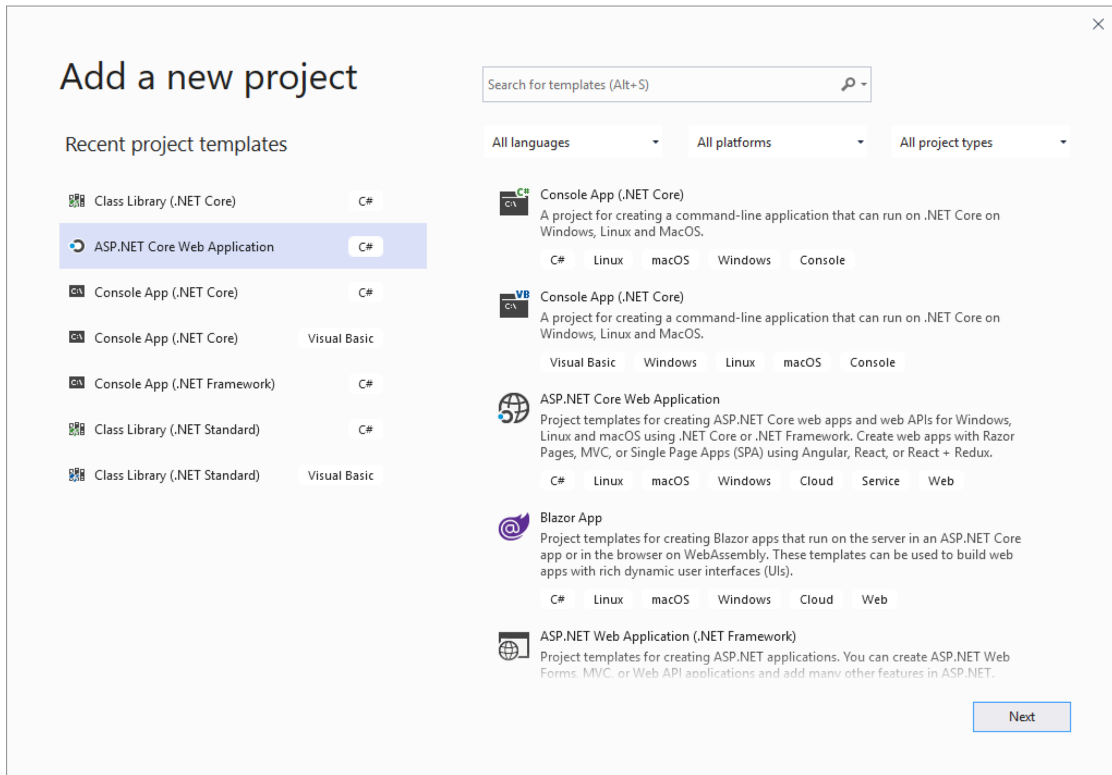
- افزودن لایه‌ی Application
- نصب و راه اندازی MediatR
- پیاده‌سازی Command و CommandHandler
- پیاده‌سازی Fluent Validation
- تست برنامه در مرحله Command

www.takbook.com

افزودن لایه‌ی Application

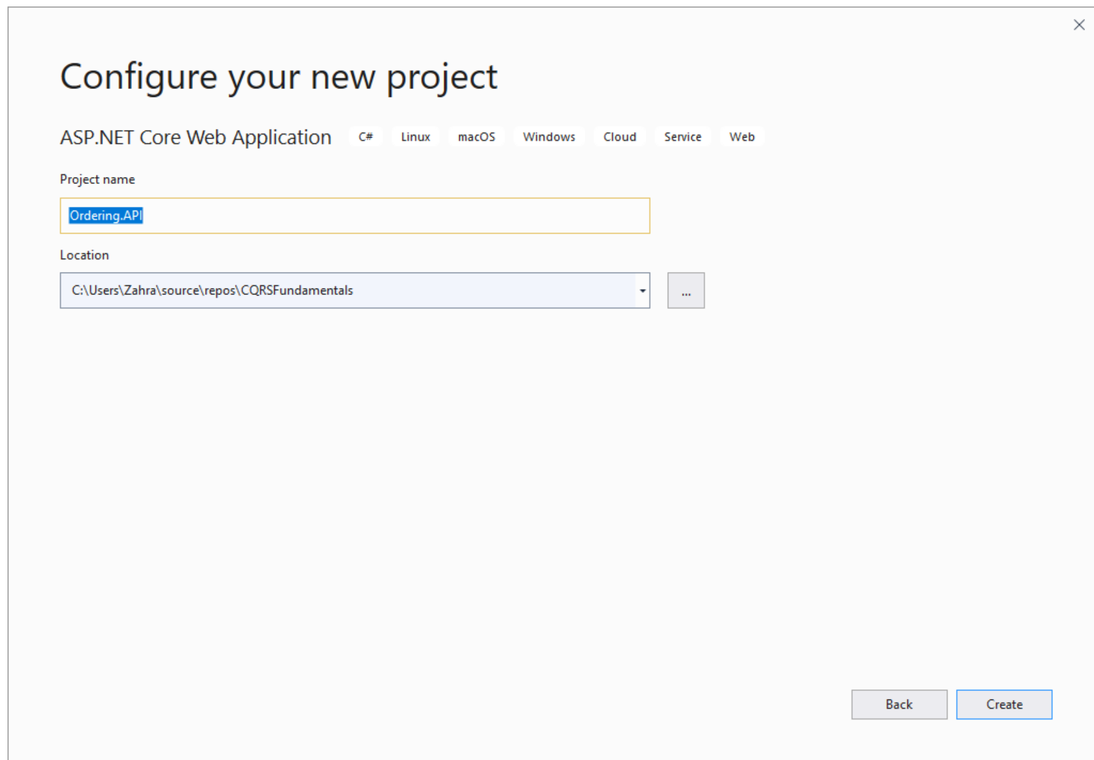
لایه Application جایی اســـت که لایه‌ی Presentation را از لایه‌ی Domain جدا کرده و در آن باید Use Caseهای اپلیکیشـــن را پیاده‌ســـازی کنید. این لایه باید دیتای موردنیاز لایه‌ی Presentation را آماده و در قالب موردنظر برگرداند.

- **برای ایجاد این لایه بر روی Solution راست کلیک و Add new Project را انتخاب نمایید.**

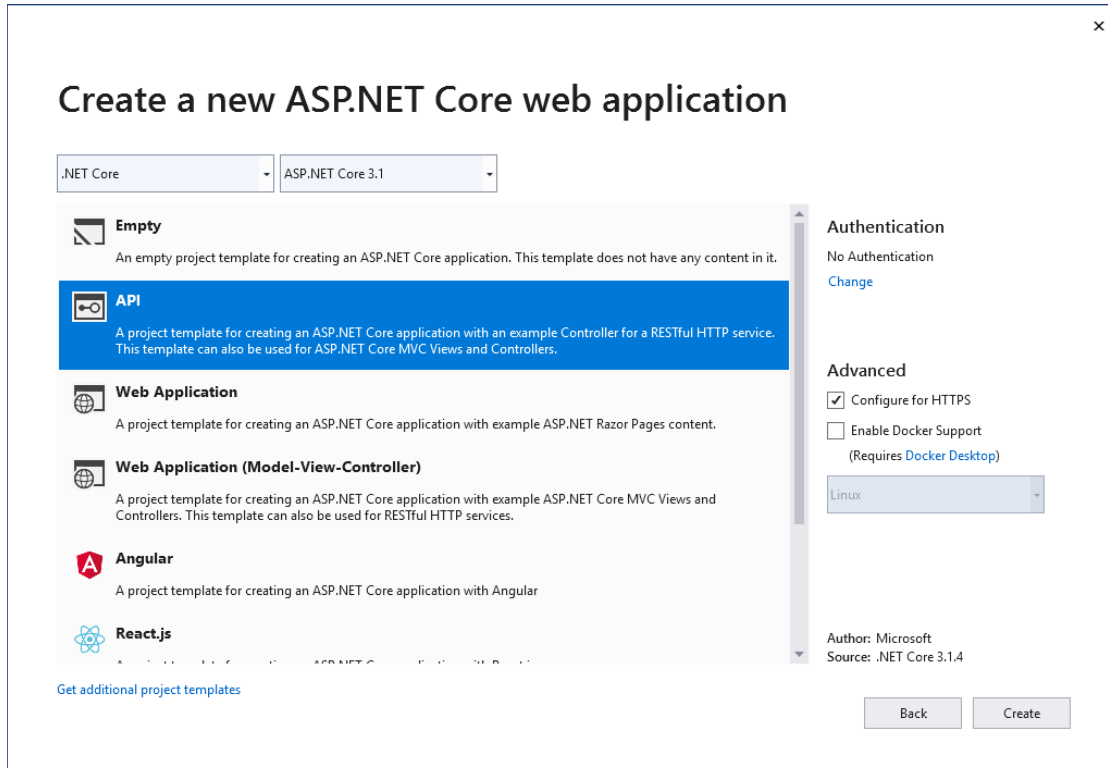


- **حالا همانند تصـــویربالا، ASP.NET Core Web Application را انتخاب و بر روی Next کلیک کنید.**
- **در کادر بعدی نام پروژه را Ordering.API بگذارید و بر روی Create کلیک نمایید.**

www.takbook.com

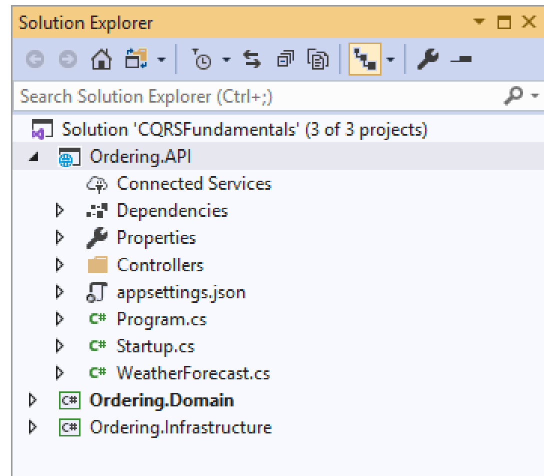


- در کادر بعد گزینه‌ی API و NET Core 3.1. را انتخاب کنید.

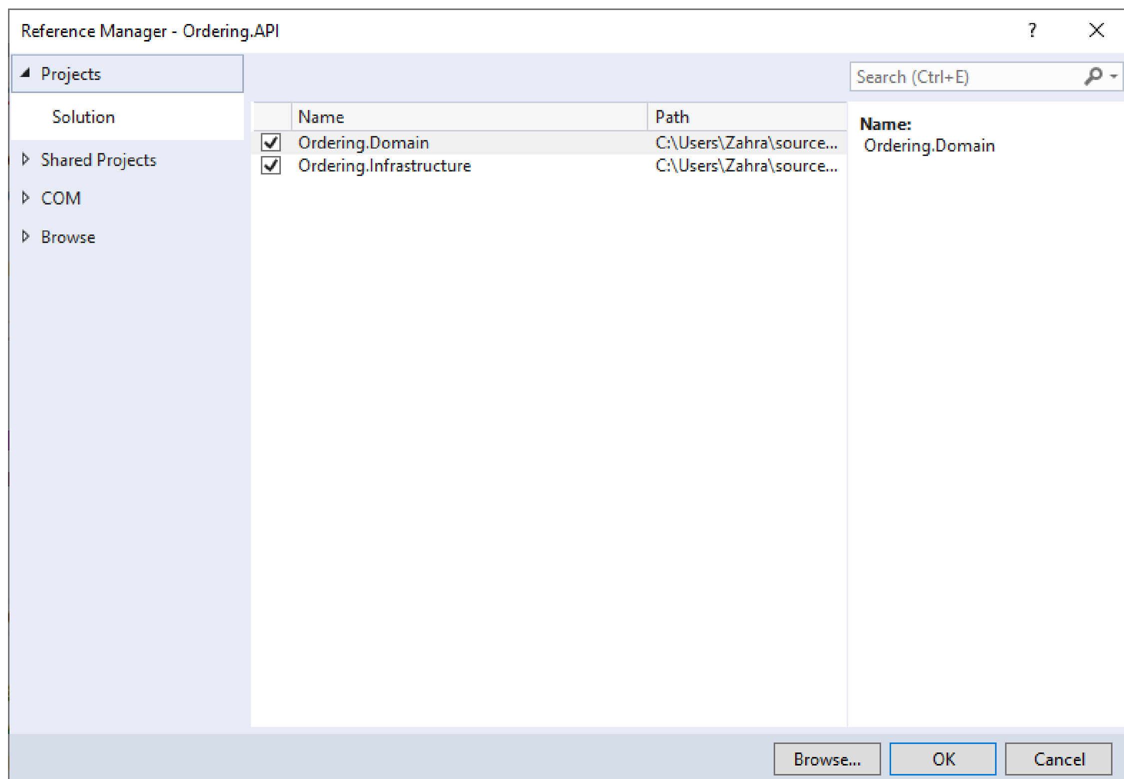


ساختار پروژه :

www.takbook.com



در این پروژه بایـد قبـل از هر کـاری از دو پروژه Ordering.Infrastructure و Ordering.Domain رفرنس بگیریم پس روی Dependencies راسـت کلیک و Add Reference را بزنید و سـپس در کادر باز شـده تیک هر دو پروژه را انتخاب و OK کنید.



www.takbook.com

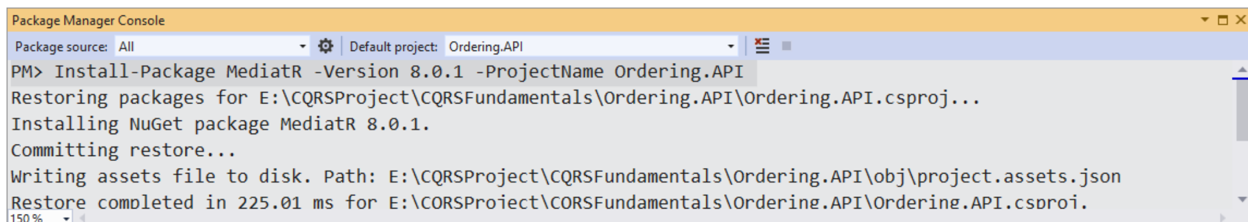
نصب و راه‌اندازی MediatR

همانطور که در فصل اول گفتیم، هر عملیاتی در CQRS یا باید Command یا Query باشد. Query نباید State دیتابیس را تغییر دهد و حتما باید یک مقدار را برگرداند اما Command دقیقا برعکس Query است یعنی هم State دیتابیس را تغییر می‌دهد و هم نباید چیزی را برگرداند.

در این کتاب برای پیاده‌سازی CQRS از MediatR استفاده کرده‌ایم. MediatR یک کتابخانه کوچک و ساده است که به شما امکان ارسال پیام‌های داخل حافظه را می‌دهد.

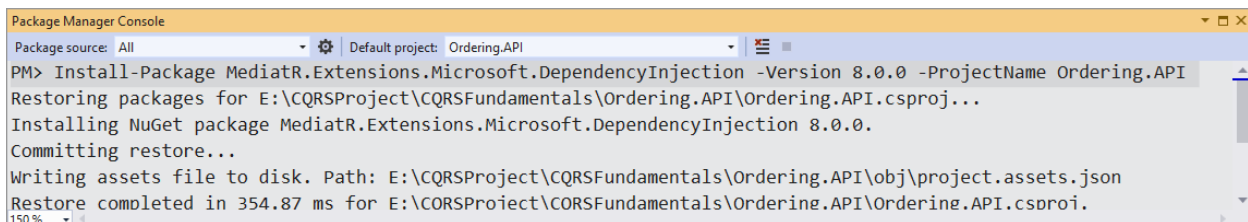
برای افزودن MediatR وارد مسیر Tools→NuGet Package Manager→Package Manager Console شوید و دستورات پایین را اجرا کنید.

```
Install-Package MediatR -Version 8.0.1 -ProjectName Ordering.API
```


Package Manager Console
Package source: All | Default project: Ordering.API

```
PM> Install-Package MediatR -Version 8.0.1 -ProjectName Ordering.API
Restoring packages for E:\CQRSProject\CQRSFundamentals\Ordering.API\Ordering.API.csproj...
Installing NuGet package MediatR 8.0.1.
Committing restore...
Writing assets file to disk. Path: E:\CQRSProject\CQRSFundamentals\Ordering.API\obj\project.assets.json
Restore completed in 225.01 ms for E:\CQRSProject\CQRSFundamentals\Ordering.API\Ordering.API.csproj.
```

```
Install-Package MediatR.Extensions.Microsoft.DependencyInjection -Version 8.0.0 -ProjectName Ordering.API
```


Package Manager Console
Package source: All | Default project: Ordering.API

```
PM> Install-Package MediatR.Extensions.Microsoft.DependencyInjection -Version 8.0.0 -ProjectName Ordering.API
Restoring packages for E:\CQRSProject\CQRSFundamentals\Ordering.API\Ordering.API.csproj...
Installing NuGet package MediatR.Extensions.Microsoft.DependencyInjection 8.0.0.
Committing restore...
Writing assets file to disk. Path: E:\CQRSProject\CQRSFundamentals\Ordering.API\obj\project.assets.json
Restore completed in 354.87 ms for E:\CQRSProject\CQRSFundamentals\Ordering.API\Ordering.API.csproj.
```

بعد از نصب پکیج‌های بالا باید MediatR را در DI Container خود Register کنیم بنابراین به متد ConfigureServices کلاس .Startup رفته و دستور پایین را در آن وارد نمایید.

```
services.AddMediatR(Assembly.GetExecutingAssembly());
```

کدهای کلاس Startup:

```
using MediatR;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
```

www.takbook.com

```
using System.Reflection;

namespace Ordering.API
{
    public class Startup
    {
        public Startup(IConfiguration configuration)
        {
            Configuration = configuration;
        }

        public IConfiguration Configuration { get; }

        public void ConfigureServices(IServiceCollection services)
        {
            services.AddMediatR(Assembly.GetExecutingAssembly());
            services.AddControllers();
        }

       public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
       {
            if (env.IsDevelopment())
            {
                app.UseDeveloperExceptionPage();
            }

            app.UseHttpsRedirection();

            app.UseRouting();

            app.UseAuthorization();

            app.UseEndpoints(endpoints =>
            {
                endpoints.MapControllers();
            });
        }
    }
}
```

www.takbook.com

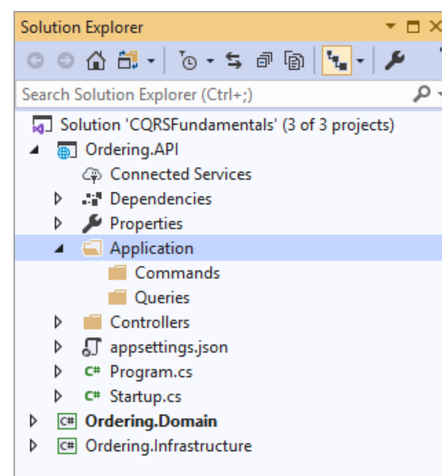
پیاده‌سازی Command

حالا متدهای اپلیکیشن باید مطابق با الگوی CQRS به دو قسمت Command و Query تقسیم شوند سپس برای هر متد Command یا Query یک کلاس ایجاد نماییم و در پایان هر کدام از این کلاس‌ها، اینترفیس IRequest موجود در MediatR را پیاده‌سازی کنند.

توجه!!

از آنجایی که تعداد کلاس‌های Command زیاد است من یک مورد را توضیح می‌دهم و باقی موارد را در کد GitHub قرار خواهم داد.

خب بیایید یک فولدر به نام Application ایجاد و سپس درون این فولدر دو فولدر دیگر با نام‌های Commands و Queries اضافه کنید.

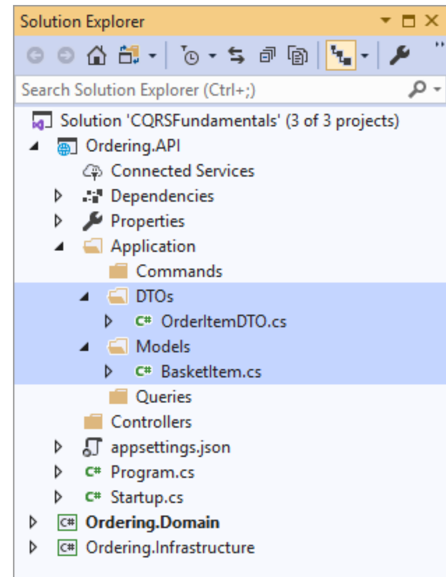


ما با متد CreateOrder شروع می‌کنیم. این متد یک Command است زیرا درخواست تغییر State اپلیکیشن را دارد و به کلاینت هیچ داده‌ای به جز اعلام موفقیت یا عدم موفقیت عملیات برنمی‌گرداند.

برای پیاده‌سازی این Command به یک کلاس نیاز داریم که داده‌های موردنیاز برای انجام این عملیات را در خود نگه دارد. اما قبل از ایجاد این کلاس باید یک کلاس Basket و یک کلاس OrderItemDTO ایجاد نماییم زیرا این دو کلاس پراپرتی‌های موردنیاز این Command هستند.

بنابراین در فولدر Application یک فولدر به نام Models و یکی با نام DTOs ایجاد کنید و سپس در فولدر Models یک کلاس به نام BasketItem و در فولدر DTOs یک کلاس با نام OrderItemDTO اضافه نمایید.

www.takbook.com



کدهای درون کلاس **BasketItem**:

```
namespace Ordering.API.Application.Models
{
    public class BasketItem
    {
        public string Id { get; set; }
        public int ProductId { get; set; }
        public string ProductName { get; set; }
        public decimal UnitPrice { get; set; }
        public decimal OldUnitPrice { get; set; }
        public int Quantity { get; set; }
        public string PictureUrl { get; set; }
    }
}
```

کدهای درون کلاس **OrderItemDTO**:

```
namespace Ordering.API.Application.DTOs
{
    public class OrderItemDTO
    {
        public int ProductId { get; set; }

        public string ProductName { get; set; }

        public decimal UnitPrice { get; set; }

        public decimal Discount { get; set; }
```

www.takbook.com

```
        public int Units { get; set; }

        public string PictureUrl { get; set; }
    }

}
```

قبل از ایجاد کلاس CreateOrderCommand می‌خواهم موضوعی را با هم بررسی کنیم.

بالاتر گفتیم Command باید نتیجه موفقیت یا عدم موفقیت یک عملیات را برگرداند و بهترین نوعی که Command می‌تواند برگرداند یک bool است.

فرض کنید می‌خواهید Commandی برای کنسل کردن Order خود بنویسید :

۱) اولین کاری که باید انجام دهید این است که Id این Order را در دیتابیس جستجو کنید.
۲) سپس در صورتی که مقداری بازگشتی، null باشد باید یک اکسپشن Throw شود.
۳) در غیر این صورت باید سفارش را کنسل نمایید و مقدار True را برگردانید.

در اینجا Throw شدن Exception راه حل خوبی نیست چون Exceptionها می‌توانند از لایه‌های مختلف کد عبور کنند و همین باعث پیچیده شدن کد می‌شود. به عبارت دیگر وقتی برای کنترل جریان برنامه از Exceptionها استفاده می‌کنید مشابه دستور GOTO عمل می‌نمایید.

پیشنهاد من برای حل این مشکل، برگرداندن یک کلاس Result به جای نوع bool است. با این کار می‌توانیم به جای Throw شدن یک Exception، یک مقدار منطقی را، که نمایانگر موفقیت یا عدم موفقیت یک عمل است برگردانیم.

اما برای راحتی کار به جای اینکه یک کلاس Result ایجاد کنید می‌توانید از یک NuGet به نام CSharpFunctionalExtensions استفاده نمایید. این NuGet یک کلاس Result دارد که با استفاده از آن می‌توانید Exceptionها و نتایج عملیات را به به درستی مدیریت کنید.

برای اضافه کردن این NuGet وارد مسیر Tools→NuGet Package Manager→Package Manager Console شوید و سپس دستورات پایین را در آن اجرا کنید.

```
Install-Package CSharpFunctionalExtensions -Version 2.8.0 -ProjectName
Ordering.API
```

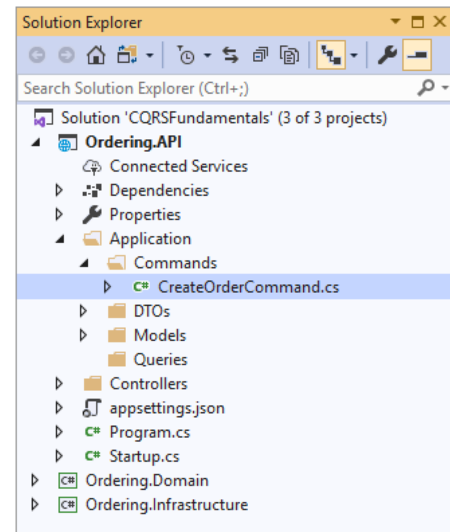
www.takbook.com

```
Package Manager Console
Package source: All    Default project: Ordering.API
PM> Install-Package CSharpFunctionalExtensions -Version 2.8.0 -ProjectName Ordering.API
Restoring packages for E:\CQRSProject\CQRSFundamentals\Ordering.API\Ordering.API.csproj...
  GET http://nuget.geeksms.uat.co/nuget/FindPackagesById()?id='CSharpFunctionalExtensions'&semVerLevel=2.0.0
  GET https://api.nuget.org/v3-flatcontainer/csharpfunctionalextensions/index.json
  OK https://api.nuget.org/v3-flatcontainer/csharpfunctionalextensions/index.json 1107ms
  GET https://api.nuget.org/v3-flatcontainer/csharpfunctionalextensions/2.8.0/
150 %
```

حالا زمان ایجاد کلاس CreateOrderCommand است.

همانطور که می‌دانید یک Command یک کلاس Immutable اســـت که تعدادی فیلد Read Only جهت نگهداری اطلاعات موردنیاز در یک Business Transaction دارد.

بنابراین در فولدر Commands یک کلاس با نام CreateOrderCommand اضافه و سپس کدهای پایین را در آن وارد کنید.



کدهای کلاس CreateOrderCommand:

```
using CSharpFunctionalExtensions;
using MediatR;
using Ordering.API.Application.DTOs;
using Ordering.API.Application.Models;
using System.Collections.Generic;
using System.Linq;

namespace Ordering.API.Application.Commands
{
    public sealed partial class CreateOrderCommand : IRequest<Result>
    {
```

www.takbook.com

```
    private readonly List<OrderItemDTO> _orderItems;
    public string UserId { get; private set; }
    public string UserName { get; private set; }
    public string City { get; private set; }
    public string Street { get; private set; }
    public string Country { get; private set; }
    public string ZipCode { get; private set; }

    public IEnumerable<OrderItemDTO> OrderItems => _orderItems;
    public CreateOrderCommand()
    {
        _orderItems = new List<OrderItemDTO>();
    }

      public CreateOrderCommand(List<BasketItem> basketItems, string
      userId, string userName, string city, string street, string
      country, string zipcode) : this()
      {
        _orderItems = basketItems.Select(item => new OrderItemDTO()
        {
            ProductId = item.ProductId,
            ProductName = item.ProductName,
            PictureUrl = item.PictureUrl,
            UnitPrice = item.UnitPrice,
            Units = item.Quantity
        }).ToList();
        UserId = userId;
        UserName = userName;
        City = city;
        Street = street;
        Country = country;
        ZipCode = zipcode;
    }

  }
}
```

نکته!!

کلاس Command هیچ گونه Behaviorی ندارند.

www.takbook.com

کلاس CreateOrderCommand نیازمندی‌های خود را از طریق Constructor مشخص می‌کند. توجه داشته باشید بعد از ارسال مقادیر موردنیاز به Constructor، این مقادیر قابل تغییر نیستند چون فیلدهای این کلاس فقط خواندنی است.

پیاده‌سازی CommandHandler

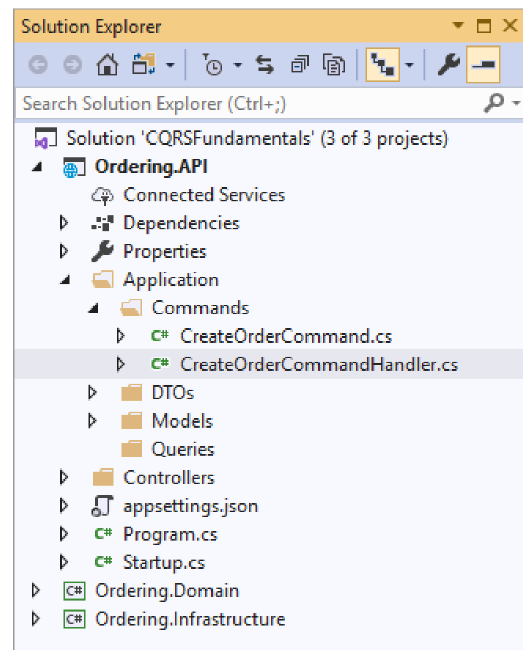
داده‌های Command شامل داده‌هایی است که از DTO می‌آید. Command باید بگوید که چه کاری را می خواهد انجام دهد و اینکه چطور این کار انجام می‌شود در حوزه کاری آن نیست و نباید این مسئولیت را به Command واگذار کرد.

بنابراین از آن‌جا که ما نمی‌خواهیم Command خود را اجرا کنیم باید یک کلاس جداگانه‌ای وجود داشته باشد که این کار را انجام دهد و اینجاست که CreateOrderCommandHandler وارد بازی می‌شود.

پس برای هر Command باید یک کلاس Command Handler مشخص کنید.

در MediatR کلاس‌هایی که وظیفه پردازش یک Command را بر عهده دارند باید از اینترفیس IRequestHandler ارث‌بری و سپس متد Handle آن را پیاده‌سازی کنند.

خب ما باید در فولدر Commands یک کلاس CreateOrderCommandHandler ایجاد و سپس در آن اینترفیس IRequestHandler را پیاده‌سازی کنیم.



www.takbook.com

کدهای درون کلاس CreateOrderCommandHandler :

```
using CSharpFunctionalExtensions;
using MediatR;
using Ordering.Domain.AggregatesModel.OrderAggregate;
using System.Threading;
using System.Threading.Tasks;

namespace Ordering.API.Application.Commands
{
    public class CreateOrderCommandHandler:
    IRequestHandler<CreateOrderCommand, Result>
    {
      private readonly IOrderRepository _orderRepository;

        public CreateOrderCommandHandler(IOrderRepository orderRepository)
        {
            _orderRepository = orderRepository;
        }

         public async Task<Result> Handle(CreateOrderCommand message,
         CancellationToken cancellationToken)
        {
             var address = new Address(message.Street, message.City,
             message.Country, message.ZipCode);
            var order = new Order(message.UserId, message.UserName, address);

            foreach (var item in message.OrderItems)
            {
             order.AddOrderItem(item.ProductId, item.ProductName,
             item.UnitPrice, item.Discount, item.PictureUrl, item.Units);
            }

             _orderRepository.Add(order);
              await
             _orderRepository.UnitOfWork.SaveEntitiesAsync(cancellationToken
             );

             return Result.Success();
        }


    }
```

www.takbook.com

```
}
```

همانطور که در کد بالا می‌بینید:

- **اولین پارامتر ورودی** اینترفیس IRequestHandler کلاسی است که پردازش روی آن انجام می‌شود. این کلاس حتما باید از نوع Command باشد یعنی IRequest را پیاده‌سازی کرده باشد.
- **دومین پارامتر ورودی این اینترفیس،** کلاسی است که نتیجه پردازش عملیات را به‌عنوان Response برمی‌گرداند.
- **به Constructor** کلاس CommandHandler اینترفیس IOrderRepository تزریق شده، تا با آن ثبت اطلاعات Order در دیتابیس را انجام دهیم.
- **در اینجا یک متد Handle** وجود دارد که قرار است یک سفارش را ایجاد و سپس تایید ثبت موفق سفارش را برگرداند. **این متد باید :**
  - **ورودی‌های موردنظر سفارش را بگیرد.**
  - **سپس پراپرتی‌های Command را مقداردهی کند.**
  - **در پایان هم عملیات ذخیره‌سازی دیتابیس را انجام و ()Result.Success را برگرداند.**

نکته!!

توجه داشته باشید که Command تنها باید توسط یک Handler پردازش شود چون Command با Transaction در ارتباط است. این تفاوت مهم بین Command و Event است.

ما در کلاس CreateOrderCommandHandler از IOrderRepository و DBContext استفاده کردیم بنابراین این دو را باید در متد ConfigureServices کلاس Startup رجیستر کنیم تا این متد برای ما از این دو آبجکت Instance بسازد.

اگر این آبجکت‌ها را رجیستر نکنیم، اکسپشنی در زمان اجرا Throw می‌شود و اپلیکیشن با شکست روبرو خواهد شد. بنابراین کدهای پایین را در متد ConfigureServices اضافه کنید.

```
services.AddDbContext<OrderingContext>(options =>
  {
      options.UseSqlServer(Configuration["ConnectionString"]);
  });
  services.AddScoped<IOrderRepository, OrderRepository>();
services.AddScoped<IBuyerRepository, BuyerRepository>();
```

www.takbook.com

کدهای کلاس **Startup** :

```
using MediatR;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Ordering.Domain.AggregatesModel.BuyerAggregate;
using Ordering.Domain.AggregatesModel.OrderAggregate;
using Ordering.Infrastructure;
using Ordering.Infrastructure.Repositories;
using System.Reflection;

namespace Ordering.API
{
    public class Startup
    {
        public Startup(IConfiguration configuration)
        {
            Configuration = configuration;
        }

        public IConfiguration Configuration { get; }

        public void ConfigureServices(IServiceCollection services)
        {
            services.AddMediatR(Assembly.GetExecutingAssembly());
            services.AddDbContext<OrderingContext>(options =>
               {
                   options.UseSqlServer(Configuration["ConnectionString"]);
               });
            services.AddScoped<IOrderRepository, OrderRepository>();
          services.AddScoped<IBuyerRepository, BuyerRepository>();

            services.AddControllers();
        }

       public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
       {
            if (env.IsDevelopment())
            {
                app.UseDeveloperExceptionPage();
```

www.takbook.com

```
            }

            app.UseHttpsRedirection();

            app.UseRouting();

            app.UseAuthorization();

            app.UseEndpoints(endpoints =>
            {
                endpoints.MapControllers();
            });
        }
    }
}
```

ConnectionString چیست؟

زمانیکه می‌خواهید اپلیکیشن‌تان را توسعه دهید و در ماشین‌های مختلف مستقر نمایید، موضوع مشخص کردن مکان دیتابیس مطرح می‌شود.

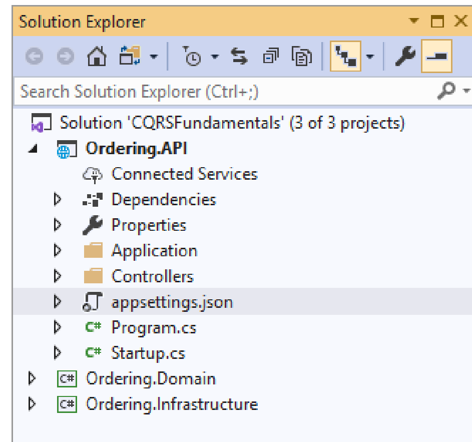
نوع دیتابیس با توجه به بیزنس شما تعریف خواهد شد اما مکان دیتابیس می‌تواند روی سیستم شما یا هر سرور دیگری قرار گیرد.

برای مثال :

معمولا در اپلیکیشن‌های وب، دیتابیس بر روی یک Host قراردارد (جایی که کاربران واقعی به آن دسترسی داشته باشند) و درون سخت افزار شما نیست. بنابراین مکان و تنظیمات مختلف دیتابیس باید در یک ConnectionString ذخیره شود.

از آنجاییکه ConnectionString به فریم‌ورک می‌گوید دیتابیس روی چه سروری قرار دارد، پس بهتر است آن را درون فایل appsettings.json قرار دهیم تا بتوانیم بدون کامپایل مجدد، محل دیتابیس را در کامپیوترهای مختلف مشخص کنیم.

www.takbook.com



کدهای درون appsettings.json:

```
{
"ConnectionString": "Server=.;Database=OrderingDb; Trusted_Connection=True;",
  "Logging": {
    "EnableSqlParameterLogging": true,
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information",
      "Microsoft.EntityFrameworkCore": "Error"
    }
  },
  "AllowedHosts": "*"
}
```

شما می‌توانید ConnectionString را به سلیقه خود سفارشی نمایید. البته باید توجه داشته باشید که نام این ConnectionString بعدا در معرفی دیتابیس موردنیاز است.

ایجاد دیتابیس

بعد از انجام موفقیت آمیز مراحل بالا، نوبت به ایجاد دیتابیس‌می‌رسد. **اما دیتابیس چطور ایجاد می‌شود؟**

یک روش خوب برای ایجاد دیتابیس، وادار کردن EF به ساخت دیتابیس است. ساده‌ترین رویکرد EF برای انجام این کار استفاده از Migration است.

Migration راه‌حلی برای مدیریت جداول دیتابیس می‌باشد. با Migration می‌توانید به راحتی تغییرات را به جداول دیتابیس اعمال نمایید.

www.takbook.com

اما قبل از Migration نیاز به پکیج پایین دارید پس Package Manager Console را باز کرده و دستور زیر را اجرا کنید :

```
Install-Package Microsoft.EntityFrameworkCore.Design -Version 3.1.4 -
ProjectName Ordering.API
```

```
Package Manager Console
Package source: All   Default project: Ordering.API
PM> Install-Package Microsoft.EntityFrameworkCore.Design -Version 3.1.4 -ProjectName Ordering.API
Restoring packages for C:\Users\Zahra\source\repos\CQRSFundamentals\Ordering.API\Ordering.API.csproj...
  GET http://nuget.geeksms.uat.co/nuget/FindPackagesById()?id='Microsoft.EntityFrameworkCore.Design'&semVerLevel=2.0.0
  GET https://api.nuget.org/v3-flatcontainer/microsoft.entityframeworkcore.design/index.json
  OK http://nuget.geeksms.uat.co/nuget/FindPackagesById()?id='Microsoft.EntityFrameworkCore.Design'&semVerLevel=2.0.0 570ms
  OK https://api.nuget.org/v3-flatcontainer/microsoft.entityframeworkcore.design/index.json 1094ms
  GET https://api.nuget.org/v3-flatcontainer/microsoft.entityframeworkcore.design/3.1.4/microsoft.entityframeworkcore.design.3.1.4.nupkg
  OK https://api.nuget.org/v3-flatcontainer/microsoft.entityframeworkcore.design/3.1.4/microsoft.entityframeworkcore.design.3.1.4.nupkg 66ms
  GET https://api.nuget.org/v3-flatcontainer/microsoft.entityframeworkcore.relational/index.json
  GET http://nuget.geeksms.uat.co/nuget/FindPackagesById()?id='Microsoft.EntityFrameworkCore.Relational'&semVerLevel=2.0.0
  OK http://nuget.geeksms.uat.co/nuget/FindPackagesById()?id='Microsoft.EntityFrameworkCore.Relational'&semVerLevel=2.0.0 217ms
150 %
```

ایجاد Migration

برای ایجاد Migration و تولید ساختار دیتابیس باید دستور Add-Migration را در Package Manager Console اجرا کنید. این دستور، ایجاد ساختار دیتابیس را برعهده دارد.

```
Add-Migration InitCQRS -Project  Ordering.Infrastructure
```

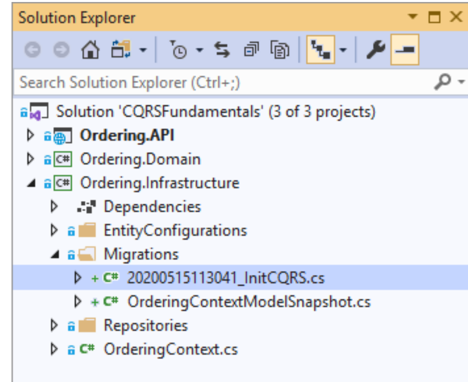
```
Package Manager Console
Package source: All   Default project: Ordering.Infrastructure
PM> Add-Migration InitCQRS -Project Ordering.Infrastructure
Build started...
Build succeeded.
To undo this action, use Remove-Migration.
PM> |
150 %
```

نکته!!

توجه داشته باشید که قبلا از اجرای این دستور باید پروژه پیش‌فرض Ordering.API باشد. اگر این پروژه پیش‌فرض نیست بر روی آن راست کلیک کنید و گزینه‌ی Project Set as Startup را انتخاب نمایید.

حالا یک فولدر به نام Migrations به Solution اضافه شده است. در این فولدر کلاسی وجود دارد که کد ایجاد دیتابیس و جداول، درون آن قرار گرفته است.

www.takbook.com



حالا نوبت به اعمال کدهای Migration به دیتابیس است. سه روش برای اعمال این کدها وجود دارد:

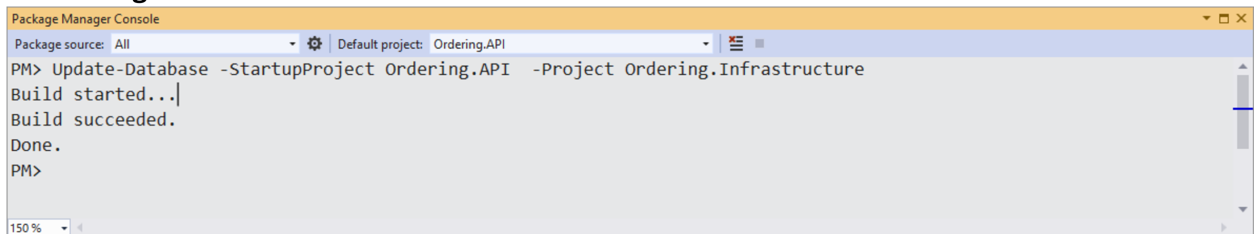
۱) اپلیکیشن شما می‌تواند در طول اجرا شدن Startup دیتابیس را چک و Migrate کند.

۲) می‌توانید یک اپلیکیشن مستقل برای Migrate دیتابیس داشته باشید.

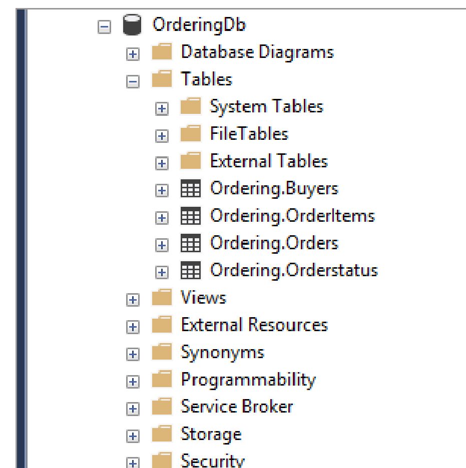
۳) می‌توانید از دستورات SQL برای Update دیتابیس استفاده کنید.

ساده‌ترین روش گزینه سوم است. شما می‌توانید تنها با نوشتن دستور Update-Database در Package Manager Console این کدها را به دیتابیس اعمال نمایید.

```
Update-Database -StartupProject Ordering.API  -Project
Ordering.Infrastructure
```



به مسیر View→SQL Server Object Explorer مراجعه کنید تا دیتابیس ایجاد شده را ببینید.

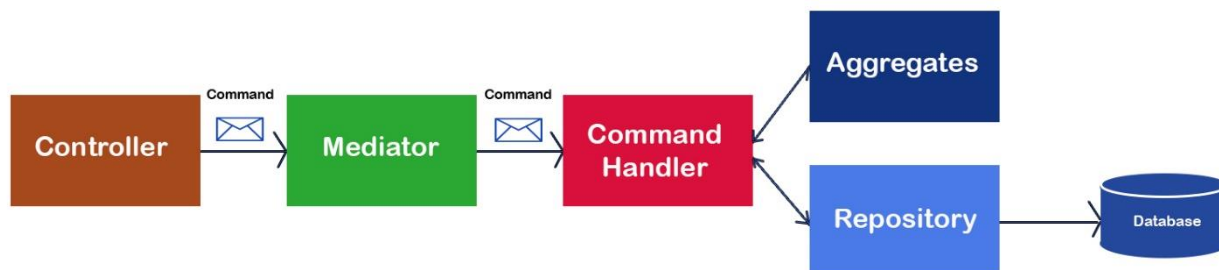


www.takbook.com

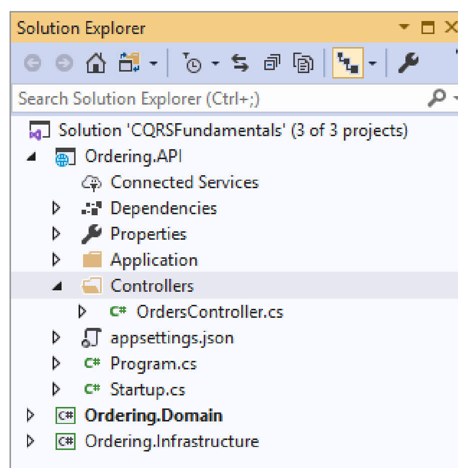
Controller

تا اینجا همه چیز عالی پیش رفت اما یک سوال؟ Handlerها چطور صدا زده می‌شوند؟

جواب این ســـوال را Controllerها می‌دهند. کدی که مســـئولیت رســـیدگی به این ســـناریو را دارد باید در Controller باشد. به طور دقیق‌تر، Controller جایی است که Command از آنجا صدا زده می‌شود و سپس این کد توسط Mediator به Handler می‌رسد و کد Handler اجرا خواهد شد.

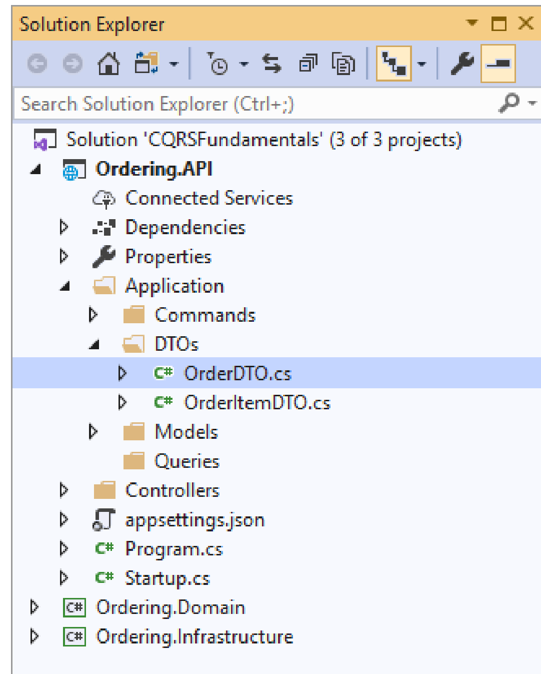


بیایید با هم این ســـناریو را کامل کنیم بنابراین در فولدر Controller یک کلاس به نام OrdersController ایجاد نمایید.



قبل از نوشـــتن کدهای OrdersController باید در فولدر DTOs یک کلاس OrderDTO ایجاد کنیم. این کلاس به عنوان ورودی اکشن متد CreateOrderFromBasketDataAsync موردنیاز است.

www.takbook.com



کدهای درون کلاس OrderDTO :

```
using Ordering.API.Application.Models;
using System;
using System.Collections.Generic;

namespace Ordering.API.Application.DTOs
{
    public class OrderDTO
    {
        public string UserId { get; set;}
        public string UserName { get; set;}
        public string City { get; set; }
        public string Street { get; set; }
        public string Country { get; set; }
        public string ZipCode { get; set; }
        public string Buyer { get; set; }
        public Guid RequestId { get; set; }
        public string BuyerId { get; set; }
        public List<BasketItem> Items { get; set; }
        public IEnumerable<OrderItemDTO> OrderItems { get; set; }
    }

}
```

کدهای درون OrdersController :

www.takbook.com

```
using CSharpFunctionalExtensions;
using MediatR;
using Microsoft.AspNetCore.Mvc;
using Ordering.API.Application.Commands;
using Ordering.API.Application.DTOs;
using System;
using System.Threading.Tasks;

namespace Ordering.API.Controllers
{
    [Route("api/v1/[controller]")]
    [ApiController]
    public class OrdersController : ControllerBase
    {
        private readonly IMediator _mediator;

        public OrdersController(IMediator mediator)
        {
          _mediator = mediator ?? throw new
          ArgumentNullException(nameof(mediator));
        }

        [Route("Order")]
        [HttpPost]
        public async Task<ActionResult<Result>>
        CreateOrderFromBasketDataAsync([FromBody] OrderDTO orderDTO)
        {
           var createOrderCommand = new CreateOrderCommand(orderDTO.Items,
           orderDTO.UserId, orderDTO.UserName, orderDTO.City,
           orderDTO.Street, orderDTO.Country, orderDTO.ZipCode);

            var result= await _mediator.Send(createOrderCommand);
            if (result.IsFailure)
            {
                return BadRequest(result.Error);
            }
            return Ok();
      }

    }
}
```

در اینجا OrdersController کاربرد یک اپلیکیشن واقعی را نشان می‌دهد و کد درون آن شامل موارد زیر است:

www.takbook.com

- **این کنترلر باید از کلاس ControllerBase ارثبری کند.**
- **ما درون Constuctor این کنترلر یک IMediator تزریق کردیم تا مسئولیت ارسال پیام‌های Command به Handler را برعهده بگیرد.**
- **متد CreateOrdeFromBasketDataAsync برای درج یک سفارش است. این متد یک DTO را از ورودی می‌گیرد و سپس درون متد، آن رابه Cammand تبدیل می‌نماید. این یعنی کلاینت برای ما یک DTO می‌فرستد و ما در سمت سرور آن را تبدیل به یک Cammand می‌کنیم.**

یک سوال؟؟

استفاده از DTO باعث ایجاد تکرار کد می‌شود پس چرا ورودی این متد را Cammand ندادیم؟

اگر ما از ورودی این متد به جای DTO، کلاس Command می‌گرفتیم تعداد خط کدهای ما کمتر می‌شد و دیگر نیازی به Map کردن هم نبود اما باید بدانید که Command و DTO ابزارهای مختلفی هستند که هر کدام برای حل یک مشکل طراحی شده‌اند.

Command به شما این امکان را می‌دهد تا به صراحت بیان کنید که اپلیکیشن چه کاری می‌تواند انجام دهد درحالیکه DTO (مخفف Data Transfer Object) یک کلاس است که داده‌ها را بین لایه‌های مختلف انتقال و به ما امکان می‌دهد تا ساختار داده‌ای Backward Compatible و مطابق با نیاز کلاینت ایجاد کنیم.

Backward Compatibility یعنی اینکه اگر Domain Model تغییرکند API ما خراب نشود چون همیشه دیتاهایی که کلاینت برای ما می‌فرستد باید با مدلی که ما از ورودی می‌گیریم یکی باشد.
استفاده از Command به جای DTO مانند این است که در ورودی متد، مستقیما از Domain Entityها استفاده کنیم پس هیچ Backward Compatibility نداریم.
استفاده از DTO و Map کردن اطلاعات به Command تضمین می‌کند که اپلیکیشن ما Backward Compatible است و به راحتی قابل Refactor می‌باشد. بنابرین این دو مسئولیت را با هم ترکیب نکنید مگر اینکه به دلایلی به Backward Compatibility نیازی نداشته باشید.

حالا قبل از اینکه اپلیکیشن را اجرا کنیم می‌خواهم کمی در مورد اعتبارسنجی صحبت کنم.

www.takbook.com

پیاده‌سازی Fluent Validation

CreateOrderCommand نقش ایجاد یک سفارش را ایفا می‌کند اما در آن هیچ ولیدیشنی برای اعتبارسنجی مقادیر ورودی نیســـت و کاربر می‌تواند با هر مقادیری این Command را فراخوانی و در نهایت امکان ورود دیتاهای نامعتبر وجود دارد. در این قســـمت می‌خواهیم با اســـتفاده از کتابخانه Fluent Validation امکان اعتبارسنجی را به Commandهای خود اضافه کنیم.

قدم اول اضـــافه کردن یک NuGet به نام FluentValidation.AspNetCore به پروژه اســـت بنابراین وارد مسـیر Tools→NuGet Package Manager→Package Manager Console شـوید و دسـتور پایین را اجرا کنید.

```
Install-Package FluentValidation.AspNetCore -Version 8.6.2 -ProjectName
Ordering.API
```

```
Package Manager Console
Package source: All    Default project: Ordering.API
PM> Install-Package FluentValidation.AspNetCore -Version 8.6.2 -ProjectName Ordering.API
Restoring packages for E:\CQRSProject\CQRSFundamentals\Ordering.API\Ordering.API.csproj...
Installing NuGet package FluentValidation.AspNetCore 8.6.2.
Committing restore...
Writing assets file to disk. Path: E:\CQRSProject\CQRSFundamentals\Ordering.API\obj\project.assets.json
Restore completed in 604.09 ms for E:\CQRSProject\CQRSFundamentals\Ordering.API\Ordering.API.csproj.
Successfully installed 'FluentValidation 8.6.2' to Ordering.API
150 %
```

بعد از افزودن این کتابخانه باید آن را داخل DI Container خود رجیســـتر کنیم بنابراین کد پایین را در متد ConfigureServices کلاس Startup اضافه نمایید :

```
services.AddControllers().AddFluentValidation(cfg =>
cfg.RegisterValidatorsFromAssemblyContaining<Startup>());
```

کدهای کلاس Startup :

```
using FluentValidation.AspNetCore;
using MediatR;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Ordering.Domain.AggregatesModel.BuyerAggregate;
using Ordering.Domain.AggregatesModel.OrderAggregate;
using Ordering.Infrastructure;
using Ordering.Infrastructure.Repositories;
using System.Reflection;
```

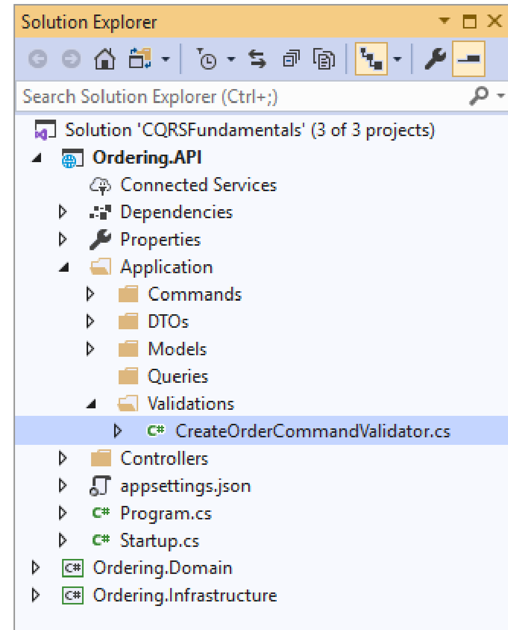
www.takbook.com

```
namespace Ordering.API
{
    public class Startup
    {
        public Startup(IConfiguration configuration)
        {
            Configuration = configuration;
        }

        public IConfiguration Configuration { get; }

        public void ConfigureServices(IServiceCollection services)
        {
            services.AddMediatR(Assembly.GetExecutingAssembly());
            services.AddDbContext<OrderingContext>(options =>
             {
                 options.UseSqlServer(Configuration["ConnectionString"]);
             });
            services.AddScoped<IOrderRepository, OrderRepository>();
          services.AddScoped<IBuyerRepository, BuyerRepository>();

            services.AddControllers().AddFluentValidation(cfg =>
            cfg.RegisterValidatorsFromAssemblyContaining<Startup>());

        }

       public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
       {
            if (env.IsDevelopment())
            {
                app.UseDeveloperExceptionPage();
            }

            app.UseHttpsRedirection();

            app.UseRouting();

            app.UseAuthorization();

            app.UseEndpoints(endpoints =>
            {
                endpoints.MapControllers();
            });
```

www.takbook.com

```
        }
    }
}
```

حالا درون فولدر Application یک فولدر با نام Validations ایجاد و ســـپس درون این فولدر یک کلاس با نام CreateOrderCommandValidator اضافه نمایید.



این کلاس باید از کلاس AbstractValidator (مربوط به Fluent Validation) ارث‌بری کند تا بتوانیم منطق اعتبارسنجی را برای کلاس CreateOrderCommand تعریف کنیم.

کدهای کلاس CreateOrderCommandValidator :

```
using FluentValidation;
using Ordering.API.Application.Models;
using Ordering.API.Application.DTOs;
using System.Collections.Generic;
using System.Linq;

namespace Ordering.API.Application.Validations
{
    public class CreateOrderCommandValidator : AbstractValidator<OrderDTO>
    {
        public CreateOrderCommandValidator()
        {
            RuleFor(command => command.City).NotEmpty();
            RuleFor(command => command.Street).NotEmpty();
```

www.takbook.com

```
            RuleFor(command => command.Country).NotEmpty();
            RuleFor(command => command.ZipCode).NotEmpty();
            RuleFor(command =>
            command.Items).Must(ContainOrderItems).WithMessage("No order
            items found");

        }

        private bool ContainOrderItems(IEnumerable<BasketItem> orderItems)
        {
            return orderItems.Any();
        }
    }

}
```

همانطور که در کد بالا می‌بینید :

- ما کلاس OrderDTO را به پارامتر AbstractValidator پاس دادیم. کلاس OrderDTO به ورودی اکشـــن متد CreateOrderFromBasketDataAsync داده شـــده پس باید قبل از Map شـــدن به Command اعتبارسنجی شود.
- همچنین با استفاده از تعدادی RuleFor پراپرتی ورودی Command را چک کردیم. **به طور مثال:** خالی نبودن City و Street .

نکته!!

تمامی اعتبارســـنجی‌ها باید قبل از ورود به کلاس Command صـــورت گیرد و در صـــورت ناموفق بودن اعتبارسنجی نباید وارد متد Handle شویم.

Seed چیست؟

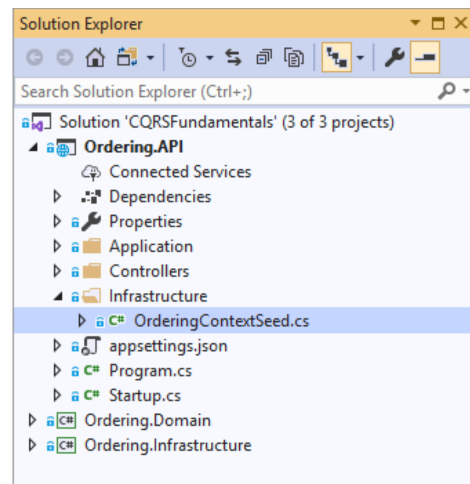
در بســـیاری از مواقع نیاز اســـت قبل از اجرای اپلیکیشـــن برخی از جدوال دیتابیس با اطلاعات پیش‌فرضـــی مقداردهی اولیه شوند که به این کار Seeding دیتابیس می‌گویند.

برای اعمال قابلیت Seed به اپلیکیشن:

- **مرحله اول :** باید کلاســی به نام OrderingContextSeed داشـته باشـیم. این کلاس وظیفه‌ی مقداردهی اولیه به جداول را برعهده دارد.

www.takbook.com

در ریشه پروژه یک فولدر با نام Infrastructure ایجاد و سپس یک کلاس به نام OrderingContextSeed به آن اضافه نمایید.



کدهای کلاس OrderingContextSeed :

```
using Ordering.Infrastructure;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace Ordering.API.Infrastructure
{
    public class OrderingContextSeed
    {

        public OrderingContextSeed(OrderingContext context)
        {
            _context = context;
        }

        public readonly OrderingContext _context;

        public async Task SeedAsync()
        {
            using (_context)
            {
               try
                {   _context.Database.EnsureCreated();

                if (!_context.OrderStatus.Any())
```

www.takbook.com

```
                {
                    var orderStatus = new List<OrderStatus>()
                      {
                            OrderStatus.Submitted,
                            OrderStatus.AwaitingValidation,
                            OrderStatus.StockConfirmed,
                            OrderStatus.Paid,
                            OrderStatus.Shipped,
                            OrderStatus.Cancelled
                      };
                    _context.OrderStatus.AddRange(orderStatus);
                }

                await _context.SaveChangesAsync();
            }
             catch (System.Exception)
             {

             }
        }
      }

    }
}
```

همانطور که در کد بالا می‌بینید :

- در Constructor این کلاس OrderingContext را Inject کردیم.
- در مرحله بعد مقدار ورودی این Constructor را درون یک متغیر Readonly قرار دادیم تا بتوانیم در متد SeedAsync با دیتابیس ارتباط برقرار کنیم.
- در متد SeedAsync ابتدا با دستور ;()_context.Database.EnsureCreated مطمئن می‌شویم که دیتابیس ایجاد شده است.
- بعد از این دستور یک عبارت شرطی داریم که چک می‌کند آیا OrderStatus رکوردی دارد یا خیر؟ در صورت منفی بودن، لیستی از OrderStatusها را ایجاد و در این جدول درج می‌نماید.
- **مرحله دوم** : در این مرحله باید کلاس OrderingContextSeed را در متد ConfigureServices رجیستر نماییم بنابراین دستور پایین را در متد ConfigureServices کلاس Startup اضافه نمایید.

```
services.AddTransient<OrderingContextSeed>();
```

www.takbook.com

- **مرحله سـوم:** تعریف یک Scope در متد Configure کلاس Startup و پاس دادن سـرویس بالا به آن است.

در این مرحله :

- **ما یک Scope ایجاد می‌کنیم.**
- **سپس کلاس Seeder را به متد GetService پاس می‌دهیم.**
- **در پایان متد SeedAsync را صـدا می‌زنیم. عبارت Wait به این خاطر اسـت که متد Seed کلاس OrderingContextSeed به صورت Task تعریف شده است.**

```
using (var scope = app.ApplicationServices.CreateScope())
{
  scope.ServiceProvider.GetService<OrderingContextSeed>().SeedAsync().Wait();
}
```

کدهای کلاس Startup :

```
using FluentValidation.AspNetCore;
using MediatR;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Ordering.API.Infrastructure;
using Ordering.Domain.AggregatesModel.BuyerAggregate;
using Ordering.Domain.AggregatesModel.OrderAggregate;
using Ordering.Infrastructure;
using Ordering.Infrastructure.Repositories;
using System.Reflection;

namespace Ordering.API
{
    public class Startup
    {
        public Startup(IConfiguration configuration)
        {
            Configuration = configuration;
        }
```

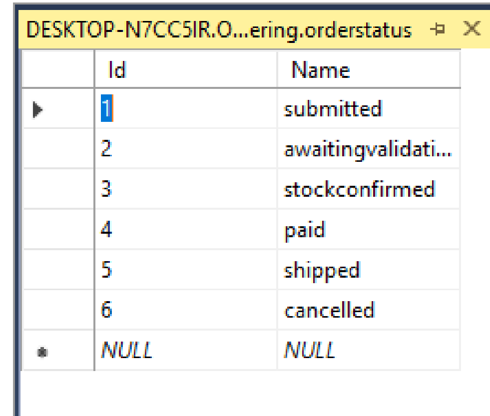
www.takbook.com

```
public IConfiguration Configuration { get; }

public void ConfigureServices(IServiceCollection services)
{
    services.AddMediatR(Assembly.GetExecutingAssembly());
    services.AddDbContext<OrderingContext>(options =>
     {
        options.UseSqlServer(Configuration["ConnectionString"]);
     });
    services.AddScoped<IOrderRepository, OrderRepository>();
  services.AddScoped<IBuyerRepository, BuyerRepository>();
  services.AddTransient<OrderingContextSeed>();

    services.AddControllers().AddFluentValidation(cfg =>
    cfg.RegisterValidatorsFromAssemblyContaining<Startup>());

}

public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }

    app.UseHttpsRedirection();
    using (var scope = app.ApplicationServices.CreateScope())
    {

       scope.ServiceProvider.GetService<OrderingContextSeed>().SeedAs
       ync().Wait();
    }


    app.UseRouting();

    app.UseAuthorization();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllers();
    });
}
```

www.takbook.com

```
        }
    }
```

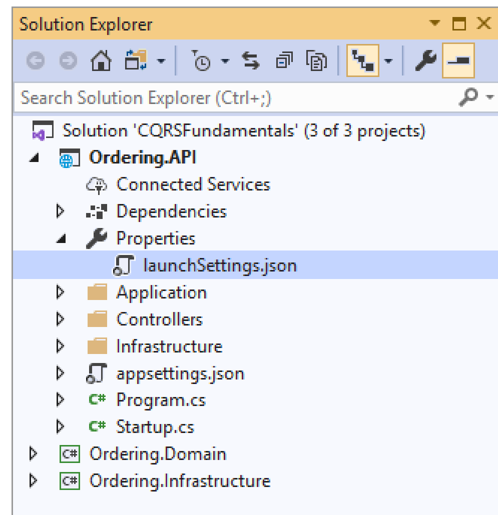
حالا اپلیکیشن را اجرا کنید تا برای اولین بار داده‌های موردنظر شما در دیتابیس ذخیره شود.



Id	Name
1	submitted
2	awaitingvalidati...
3	stockconfirmed
4	paid
5	shipped
6	cancelled
NULL	NULL

تست اپلیکیشن در مرحله Command

قبل از هر کاری بر روی launchSettings.json از زیر مجموعه Properties دابل کلیک کنید.

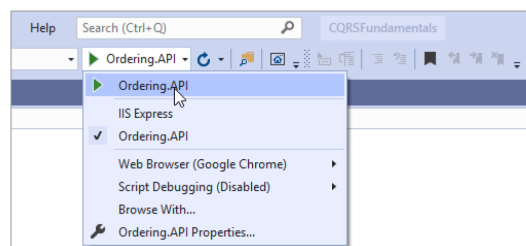


حالا آدرس نوشته شده در applicationUrl را جهت تست API کپی کنید.

www.takbook.com

```json
{
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iisExpress": {
      "applicationUrl": "http://localhost:59321",
      "sslPort": 44326
    }
  },
  "$schema": "http://json.schemastore.org/launchsettings.json",
  "profiles": {
    "IIS Express": {
      "commandName": "IISExpress",
      "launchBrowser": true,
      "launchUrl": "weatherforecast",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    },
    "Ordering.API": {
      "commandName": "Project",
      "launchBrowser": true,
      "launchUrl": "weatherforecast",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      },
      "applicationUrl": "https://localhost:5001;http://localhost:5000"
    }
  }
}
```

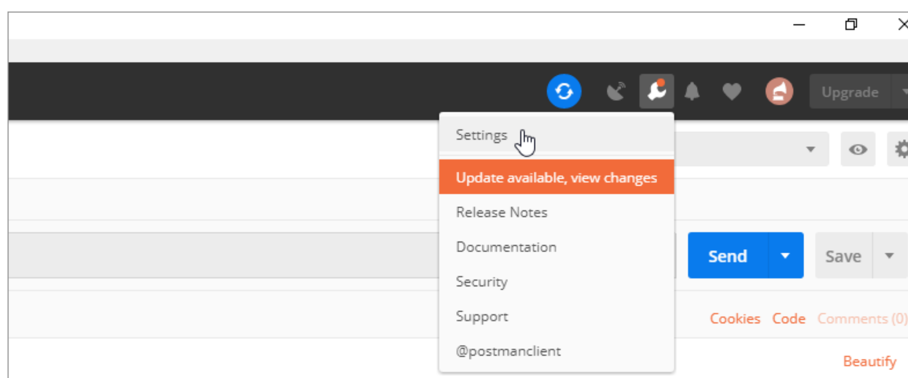
خب در نوار بالایی بر روی Ordering.API کلیک کنید تا پروژه اجرا شود.



الان برنامه Postman را اجرا نمایید. اگر این برنامه را نصـــب نکرده‌اید می‌توانید از آدرس پایین آن را دانلود کنید.

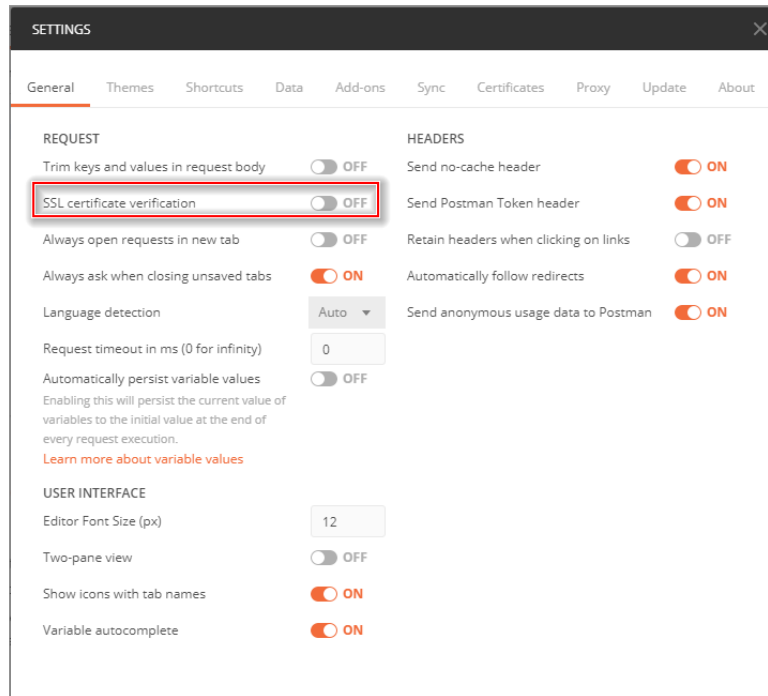
https://www.postman.com/downloads/

قبل از صدا زدن Endpoint باید SSL را غیرفعال کنیم پس همانند تصویر وارد Setting این برنامه شوید.



www.takbook.com

سپس گزینه SSL certificate verification را OFF کنید.



خب API پایین را همانند تصویر آماده و سپس بر روی Send کلیک کنید.

Api: https://localhost:5001/api/v1/orders/order

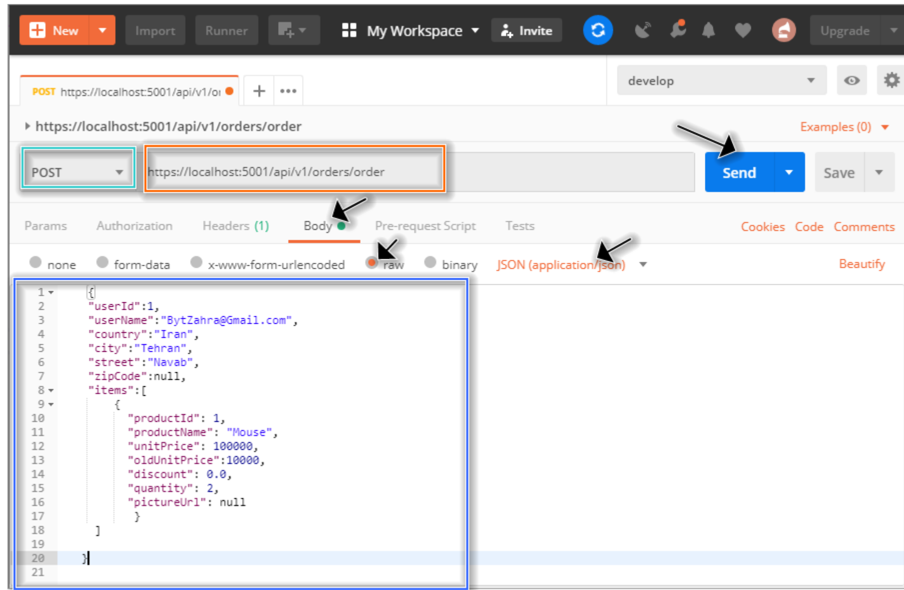
Verb: POST

Body:

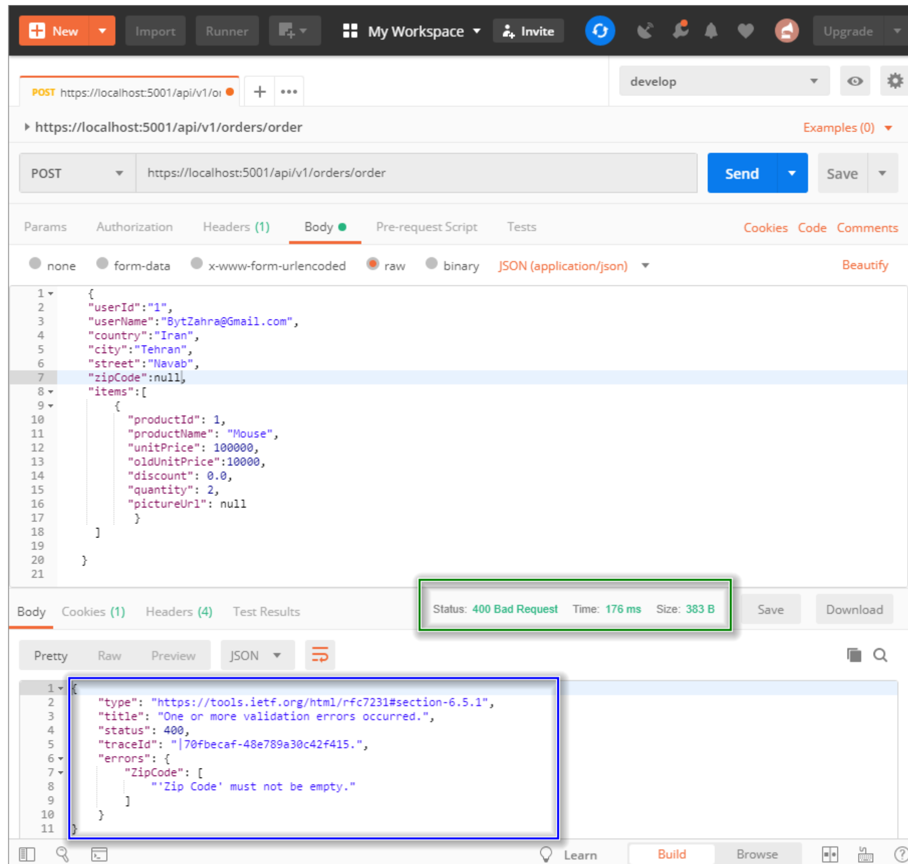
```
{
  "userId":"1",
  "userName":"BytZahra@Gmail.com",
  "country":"Iran",
  "city":"Tehran",
  "street":"Navab",
  "zipCode":null,
  "items":[
        {
          "productId": 1,
          "productName": "Mouse",
          "unitPrice": 100000,
         "oldUnitPrice":10000,
          "discount": 0.0,
          "quantity": 2,
          "pictureUrl": null
        }
    ]
```

www.takbook.com

}



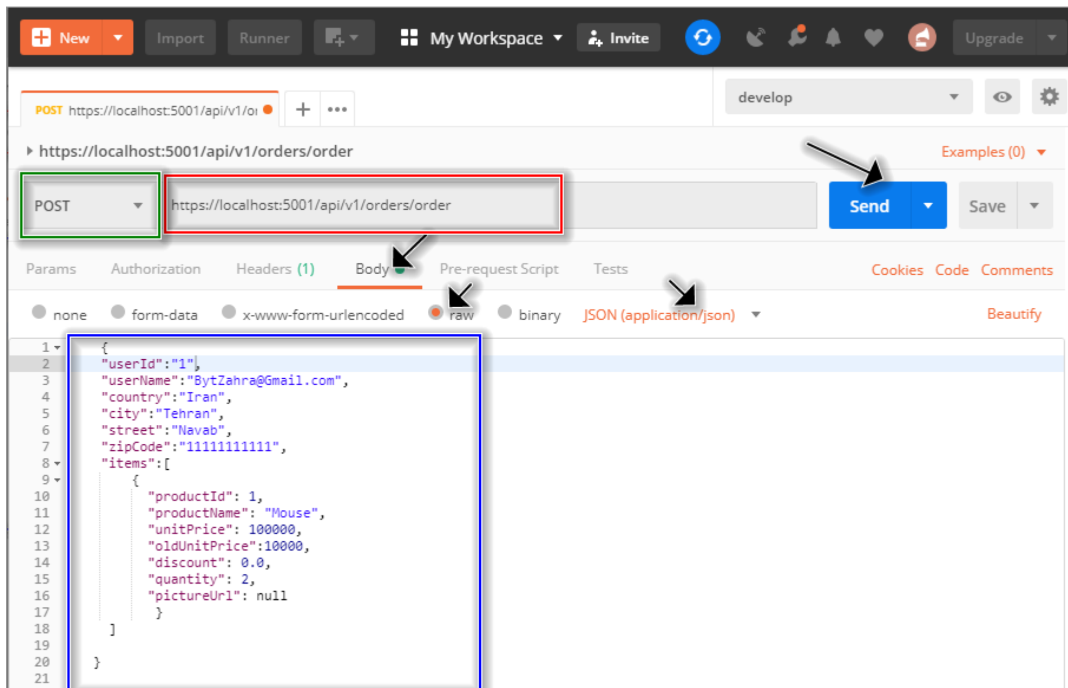
همانطور که در تصویر پایین می‌بینید خروجی این API به دلیل Null بودن فیلد ZipCode با خطا مواجه شده است، که این خطا نشان می‌دهد Fluent Validation بدرستی وظیفه اعتبارسنجی ورودی‌ها را انجام داده است.



www.takbook.com

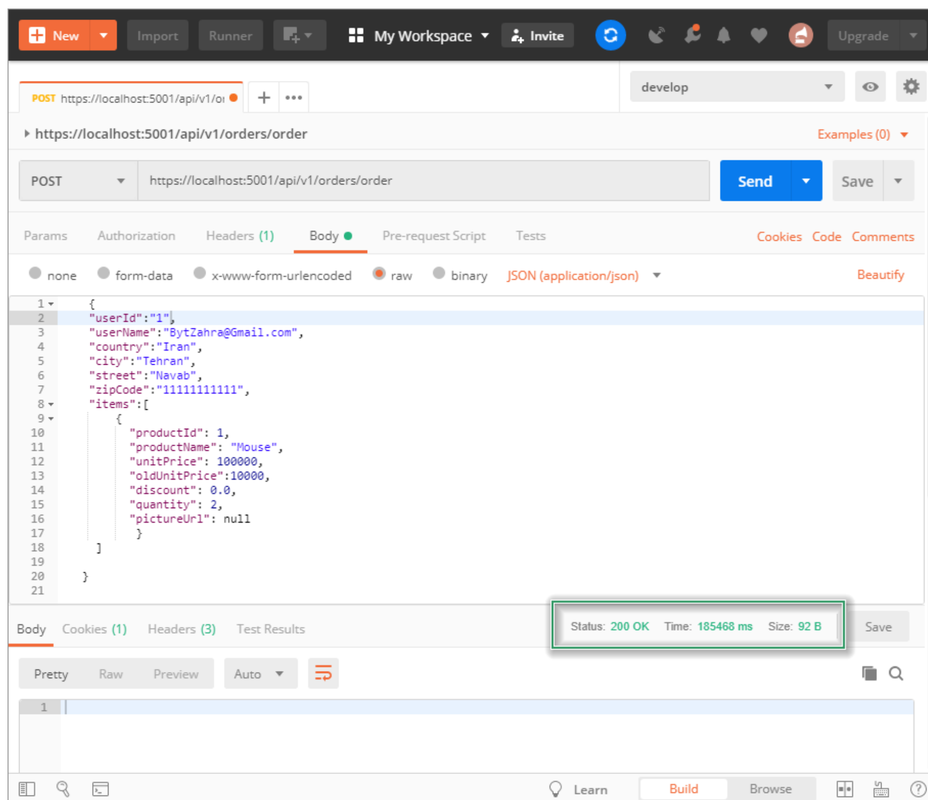
حالا یکبار دیگر این API را با JSON پایین تست کنید.

```
{
  "userId":"1",
  "userName":"BytZahra@Gmail.com",
  "country":"Iran",
  "city":"Tehran",
  "street":"Navab",
  "ZipCode":"1111111111",
  "items":[
        {
         "productId": 1,
         "productName": "Mouse",
         "unitPrice": 100000,
        "oldUnitPrice":10000,
         "discount": 0.0,
         "quantity": 2,
         "pictureUrl": null
        }
    ]

}
```



بعد از اجرا با خروجی زیر روبرو خواهید شد.

www.takbook.com



همانطور که می‌بینید این API به درستی اجرا و یک رکورد در دیتابیس ثبت خواهد شد.

نتیجه این API را در جدول Orders ببینید.

DESKTOP-N7CC5IR.O...- Ordering.Orders

	Id	Address_Street	Address_City	Address_Coun...	Address_ZipC...	OrderStatusId	Description	BuyerId	OrderDate
▶	1	Navab	Tehran	Iran	11111111111	1	NULL	NULL	2020-05-21 02:3...
*	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

DESKTOP-N7CC5IR....dering.OrderItems

	Id	ProductId	OrderId	Discount	PictureUrl	ProductName	UnitPrice	Units
▶	1	1	1	0.00	NULL	Mouse	100000.00	2
*	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

www.takbook.com

فصل پنجم : لایه‌ی Application و پیاده‌سازی Query

آنچه خواهید آموخت:

- معرفی و پیاده‌سازی Query
- استفاده از Dapper
- تست برنامه در مرحله Query

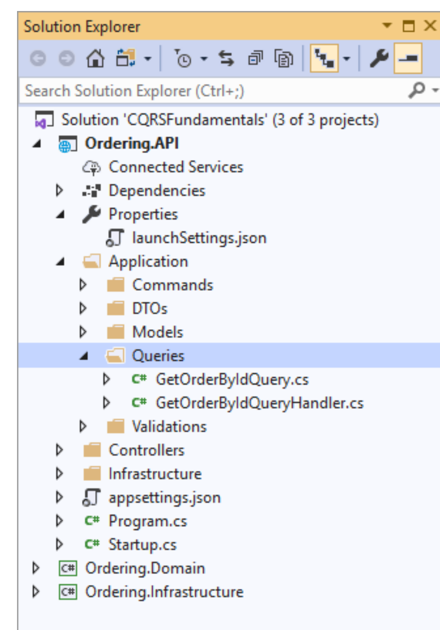
www.takbook.com

معرفی Query

تا اینجا پیاده‌سازی Command را دیدیم حالا بیایید یک پیاده‌سازی از Query انجام دهیم.

اپلیکیشـن ما یک متد GetOrderAsync دارد که سـفارش را براسـاس Id برمی‌گرداند. بنابراین ما نیاز به ایجاد یک کلاس Query به نام GetOrderByIdQuery و یک Handler به نام GetOrderByIdQueryHandler داریم. این کلاس‌ها باید هماننـد پیاده‌ســـازی Command اینترفیس IRequest و IRequestHandler را Implement کنند.

پس بیایید این دو کلاس را در فولدر Queries اضافه کنیم.



ما در Command یک خروجی Result داشـــتیم چون Command چیزی جز تایید موفق یا عدم موفقیت عملیات را برنمی‌گردلند. اما Query باید برخی اطلاعات را نیز برگردلند که این بســـته به هر Query، متفاوت است.

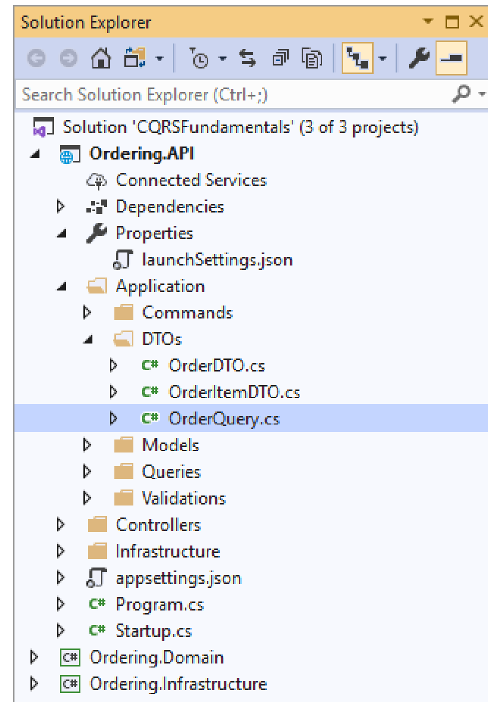
ما باید همراه با نتیجه Query یک نوعی را هم بفرســـتیم. پس کلاس GetOrderByIdQuery علاوه بر پیاده‌سازی IRequest، خروجی Result<OrderQuery> را هم به این اینترفیس پاس می‌دهد.

نکته!!

OrderQuery کلاسی است که نتیجه Query را نگه می‌دارد.

قبل از نوشـــتن کدهای کلاس GetOrderByIdQuery باید در فولدر DTOs یک کلاس OrderQuery ایجاد کنید.

www.takbook.com



کدهای کلاس OrderQuery :

```
using System;
using System.Collections.Generic;

namespace Ordering.API.Application.DTOs
{

    public class OrderQuery
    {
        public int Ordernumber { get; set; }
        public DateTime Date { get; set; }
        public string Status { get; set; }
        public string Description { get; set; }
        public string Street { get; set; }
        public string City { get; set; }
        public string Zipcode { get; set; }
        public string Country { get; set; }
        public List<Orderitem> Orderitems { get; set; }
        public decimal Total { get; set; }
    }

    public class Orderitem
    {
        public string Productname { get; set; }
```

www.takbook.com

```
        public int Units { get; set; }
        public double Unitprice { get; set; }
        public string Pictureurl { get; set; }
    }


}
```

حالا کدهای کلاس GetOrderByIdQuery را اضافه کنید :

```
using CSharpFunctionalExtensions;
using MediatR;
using Ordering.API.Application.DTOs;

namespace Ordering.API.Queries
{
    public class GetOrderByIdQuery : IRequest<Result<OrderQuery>>
    {
        public GetOrderByIdQuery(int id)
        {
            Id = id;
        }

        public int Id { get; }
    }
}
```

چون قرار نیسـت فیلدی از این کلاس را مقداردهی نماییم پس باید این Query را Immutable کنیم. همانطور که می‌بینید من Setter پراپرتی Id را حذف و سپس مقداردهی آن را در Constructor انجام دادم.

حالا نوبت به پیاده‌سازی GetOrderByIdQueryHandler رسید.

همانطور که می‌دانید با LINQ نمی‌توانیم مثل SQL خام کوئری‌هایی با Performance بالا بنویسـیم و به طور کامل از قابلیت‌های دیتابیس اســـتفاده کنیم پس در نتیجه کوئری LINQ نتیجه‌ی مطلوبی نمی‌دهد. **اینجا ما متوجه دو موضوع می‌شویم:**

۱) اســـتفاده از Domain Model مشـــترک برای Read و Write توانایی نوشـــتن Query SQLهایی با Performance بالا را محدود می‌کند.

۲) داشـــتن یک Domain Model برای هر دوی این عملیات علاوه بر پیچیده شـــدن Domain Model باعث می‌شود هیچ کدام از مسئولیت‌های Read و Write به درستی انجام نشود.

www.takbook.com

یک سوال؟ آیا با این صحبت‌ها می‌توان نتیجه گرفت که باید دو Domain Model داشته باشیم؟

پاسخ خیر است، زیرا یکی از اهداف Domain Model داشتن Encapsulation بالاست که این باعث می‌شود زمانیکه چیزی را در سیستم تغییر می‌دهید تمام داده‌ها یکپارچه بمانند و هیچ منطق بیزینسی خراب نشود. اما از آنجا که ما در قسمت Read هیچ چیز را تغییر نمی‌دهیم بنابراین دیگر نیازی هم بهEncapsulation نداریم پس نیازی به Domain Model اضافه هم نیست.

تنها موردی که Read باید نگران آن باشد ارائه خوب داده‌ها به کلاینت است. ما Domain Modelی برای Read طراحی نمی‌کنیم چون نیاز به درگیر شدن با پیچیدگی‌های ORM را نداریم. بنابراین می‌توانیم تمام کدهای دسترسی به دیتابیس را دستی بنویسید و با این کار از قابلیت‌ها و ویژگی‌های دیتابیس به طور کامل استفاده و Queryهایی با Performance بالا تولید کنیم.

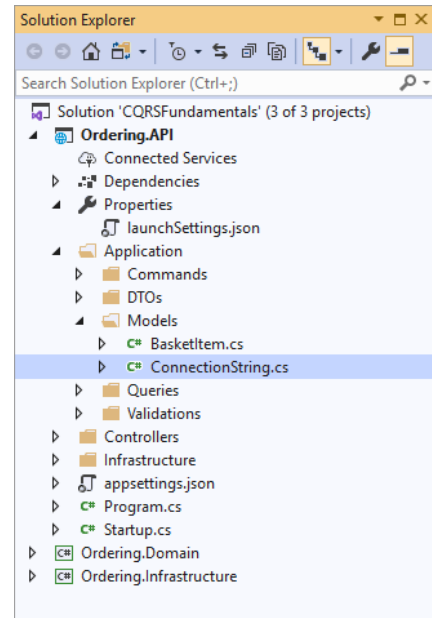
نتیجه گیری
این جدایی باعث می‌شود سمت Command ساده‌تر شود زیرا کدهایی که بخش Query به آن نیاز دارد از Domain Model حذف شده و از آنجاییکه می‌توان مستقیما با دیتابیس ارتباط برقرار کرد، پس می‌توانیم از ویژگی‌های خاص دیتابیس بهرمند شویم و سمت Query اپلیکیشن را بهینه‌تر کنیم.

پیاده‌سازی GetOrderByIdQueryHandler

جهت ارتباط با دیتابیس در کلاس GetOrderByIdQueryHandler به یک Connection String نیاز داریم اما از آنجایی که ما نمی‌توانیم از ASP. NET Dependency Injection Container برای Inject کردن یک String به این کلاس استفاده کنیم پس باید از یک کلاس دیگر به نام ConnectionString کمک بگیریم. این کلاس تنها کاری که باید انجام دهد پاس دادن یک String به کلاس GetOrderByIdQueryHandler است.

در فولدر Models یک کلاس به نام ConnectionString ایجاد کنید.

www.takbook.com



کدهای کلاس ConnectionString :

```
namespace Ordering.API.Application.Models
{
    public sealed class ConnectionString
    {
        public ConnectionString(string value)
        {
            Value = value;
        }

        public string Value { get; }
    }
}
```

خب حالا باید به کلاس Startup برویم و Connection String درون Configuration را بگیریم و به این کلاس پاس دهیم و در پایان هم کلاس ConnectionString را رجیستر کنیم.

```
var connectionString = new
ConnectionString(Configuration["ConnectionString"]);

services.AddSingleton(connectionString);
```

کدهای کلاس Startup :

```
using FluentValidation.AspNetCore;
using MediatR;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.EntityFrameworkCore;
```

www.takbook.com

```
 using Microsoft.Extensions.Configuration;
 using Microsoft.Extensions.DependencyInjection;
 using Microsoft.Extensions.Hosting;
 using Ordering.Domain.AggregatesModel.BuyerAggregate;
 using Ordering.Domain.AggregatesModel.OrderAggregate;
 using Ordering.Infrastructure;
 using Ordering.API.Application.Models;
 using Ordering.Infrastructure.Repositories;
 using System.Reflection;
 using Ordering.API.Infrastructure;

namespace Ordering.API
{
    public class Startup
    {
        public Startup(IConfiguration configuration)
        {
            Configuration = configuration;
        }

        public IConfiguration Configuration { get; }

        public void ConfigureServices(IServiceCollection services)
        {
            services.AddMediatR(Assembly.GetExecutingAssembly());
            services.AddDbContext<OrderingContext>(options =>
             {
           options.UseSqlServer(Configuration["ConnectionString"]);
             });
            services.AddScoped<IOrderRepository, OrderRepository>();
          services.AddScoped<IBuyerRepository, BuyerRepository>();

            services.AddTransient<OrderingContextSeed>();
            var connectionString = new
            ConnectionString(Configuration["ConnectionString"]);
        services.AddSingleton(connectionString);

            services.AddControllers().AddFluentValidation(cfg =>
            cfg.RegisterValidatorsFromAssemblyContaining<Startup>());

        }

      public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
      {
```

www.takbook.com

```
        if (env.IsDevelopment())
         {
                app.UseDeveloperExceptionPage();
            }

            app.UseHttpsRedirection();
            using (var scope = app.ApplicationServices.CreateScope())
            {
              scope.ServiceProvider.GetService<OrderingContextSeed>().SeedAsyn
              c().Wait();
            }

            app.UseRouting();

            app.UseAuthorization();

            app.UseEndpoints(endpoints =>
            {
                endpoints.MapControllers();
            });
        }
    }
}
```

استفاده از Dapper

در عملیات Read نیاز به نوشتن SQL خام داریم پس می‌توانیم از یک Micro ORM به نام Dapper استفاده کنیم. این ORM بسیار سبک و مناسب این سناریو است و می‌تواند کوئری‌ها را با Performance بالا اجرا کند.

وقتی در کد خود از Dapper استفاده می‌کنید، مستقیماً به کلاس SqlConnection موجود در فضای نام System.Data.SqlClient دسترسی دارید بنابراین می‌توانید از طریق متد QueryAsync و سایر اکستنشن متدهایی که کلاس SqlConnection را Extend کردند، Queryهایی با Performance بالا بنویسید.

این Library بسیار کاربردی است چون نتایج Queryها را به نوع دلخواهتان Map می‌کند و این امکان را به شما می‌دهد تا بخش خواندن اطلاعات را بطور چشمگیری Scale کنید.

برای نصب Dapper دستور پایین را در Package Manager Console وارد کنید :

```
Install-Package Dapper -Version 2.0.35 -ProjectName Ordering.API
```

www.takbook.com

```
Package Manager Console
Package source: All   Default project: Ordering.API
PM> Install-Package Dapper -Version 2.0.35 -ProjectName Ordering.API
Restoring packages for E:\CQRSProject\CQRSFundamentals\Ordering.API\Ordering.API.csproj...
Installing NuGet package Dapper 2.0.35.
Committing restore...
Writing assets file to disk. Path: E:\CQRSProject\CQRSFundamentals\Ordering.API\obj\project.assets.json
Restore completed in 5.36 sec for E:\CQRSProject\CQRSFundamentals\Ordering.API\Ordering.API.csproj.
Successfully uninstalled 'System.Reflection.Emit.Lightweight 4.3.0' from Ordering.API
150 %
```

حالا باید کلاس GetOrderByIdQueryHandler را پیاده‌سازی و نوع Resultی از OrderQuery را برگردانیم.

خب ما در قسمت Read می‌خواهیم از دست Domain Modelها خلاص شویم پس دیگر نیازی به Repository نداریم. تنها چیزی که ما نیاز داریم ارتباط مستقیم با دیتابیس جهت اجرای دستورات SQL است پس باید کلاس ConnectionString به این کلاس Inject کنیم.

حالا بیایید به متد Handle این کلاس بپردازیم.

همانطور که می‌دانید این متد هیچ Stateی را تغییر نمی‌دهد بنابراین تنها کاری که باید انجام دهیم این است که: یک سفارش را با توجه به پارامترهای ورودی از دیتابیس فیلتر کرده و سپس در یک DTO بریزیم و برگردانیم.

کدهای کلاس GetOrderByIdQueryHandler :

```
using CSharpFunctionalExtensions;
using Dapper;
using MediatR;
using Microsoft.Data.SqlClient;
using Ordering.API.Application.DTOs;
using Ordering.API.Application.Models;
using System.Collections.Generic;
using System.Threading;
using System.Threading.Tasks;

namespace Ordering.API.Queries
{
    public class GetOrderByIdQueryHandler : IRequestHandler<GetOrderByIdQuery,
    Result<OrderQuery>>
    {
        private readonly ConnectionString _connectionString;

        public GetOrderByIdQueryHandler(ConnectionString connectionString)
        {
```

www.takbook.com

```
        _connectionString = connectionString;
    }

    public async Task<Result<OrderQuery>> Handle(GetOrderByIdQuery
    request, CancellationToken cancellationToken)
    {

        using (var connection = new
        SqlConnection(_connectionString.Value))
        {
            connection.Open();

            var result = await connection.QueryAsync<dynamic>(
                       @"select o.[Id] as ordernumber,o.OrderDate as date,
                       o.Description as description,
                       o.Address_City   as   city,   o.Address_Country   as
                       country,      o.Address_Street      as      street,
                       o.Address_ZipCode  as  zipcode,os.Name  as  status,
                       oi.ProductName as productname, oi.Units as units,
                       oi.UnitPrice   as   unitprice,   oi.PictureUrl   as
                       pictureurl
                     FROM ordering.Orders o
                     LEFT JOIN ordering.Orderitems oi ON o.Id = oi.orderid
                   LEFT JOIN ordering.orderstatus os on o.OrderStatusId
                      = os.Id
                         WHERE o.Id=@id"
                    , new { request.Id }
                );

            if (result.AsList().Count == 0)
                throw new KeyNotFoundException();

            return MapOrderItems(result);
        }
    }

    private Result<OrderQuery> MapOrderItems(dynamic result)
    {
        var order = new OrderQuery
        {
            Ordernumber = result[0].ordernumber,
            Date = result[0].date,
```

www.takbook.com

```
                Status = result[0].status,
                Description = result[0].description,
                Street = result[0].street,
                City = result[0].city,
                Zipcode = result[0].zipcode,
                Country = result[0].country,
                Orderitems = new List<Orderitem>(),
                Total = 0
            };

            foreach (dynamic item in result)
            {
                var orderitem = new Orderitem
                {
                    Productname = item.productname,
                    Units = item.units,
                    Unitprice = (double)item.unitprice,
                    Pictureurl = item.pictureurl
                };

                order.Total += item.units * item.unitprice;
                order.Orderitems.Add(orderitem);
            }

            return Result.Success(order);
        }

    }
}
```

همانطور که در کد بالا می‌بینید:

- کوئری SQL خود را در این متد نوشتیم.
- سپس پارمترهای موردنیاز Query را از پارامتر ورودی متد دریافت کردیم.
- در مرحله بعد نتیجه را به DTO موردنظر Map نمودیم.
- در پایان هم، این DTO را برگردانیم.

توجه داشــته باشــید که این کوئری فقط یکبار به دیتابیس ســر می‌زند و همین باعث افزایش Performance خواهد شد.

نکته!!

www.takbook.com

ما خروجی متد بالا را از نوع Result گذاشــتیم چون احتمال شـکسـت Query هم وجود دارد. به عنوان مثال : مشکل ارتباط با سرور یا وجود پارامتر ورودی نامعتبر و...

حالا نوبت به صدا زدن Query است.

وارد OrdersController شوید و متد پایین را در آن بنویسید.

```
[HttpGet("{id}")]
 public async Task<ActionResult<Result>> GetOrderByIdAsync([FromRoute] int
 id)
  {
      var order = await _mediator.Send(new GetOrderByIdQuery(id));

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

کدهای کلاس OrdersController :

```
using MediatR;
using CSharpFunctionalExtensions;
using Microsoft.AspNetCore.Mvc;
using Ordering.API.Application.Commands;
using Ordering.API.Application.DTOs;
using Ordering.API.Queries;
using System;
using System.Threading.Tasks;

namespace Ordering.API.Controllers
{
    [Route("api/v1/[controller]")]
    [ApiController]
    public class OrdersController:ControllerBase
    {
        private readonly IMediator _mediator;

        public OrdersController(IMediator mediator)
        {
            _mediator = mediator ?? throw new
            ArgumentNullException(nameof(mediator));
        }
```

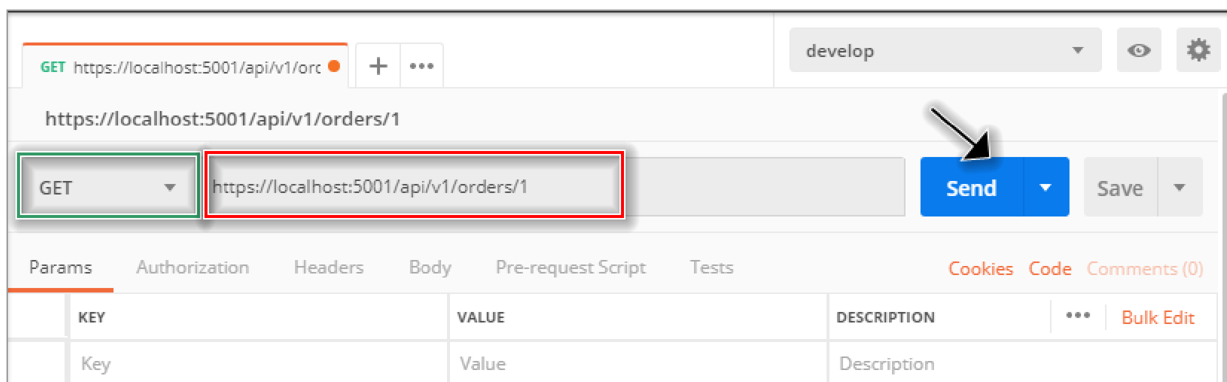
www.takbook.com

```
[Route("Order")]
[HttpPost]
 public async Task<ActionResult<Result>>
 CreateOrderFromBasketDataAsync([FromBody] OrderDTO orderDTO)
 {
      var createOrderCommand = new CreateOrderCommand(orderDTO.Items,
      orderDTO.UserId, orderDTO.UserName, orderDTO.City,
      orderDTO.Street, orderDTO.Country, orderDTO.ZipCode);

     var result= await _mediator.Send(createOrderCommand);
       if (result.IsFailure)
       {
           return BadRequest(result.Error);
       }
            return Ok();
 }

    [HttpGet("{id}")]
        public async Task<ActionResult<Result>>
        GetOrderByIdAsync([FromRoute] int id)
       {
           var order = await _mediator.Send(new GetOrderByIdQuery(id));

           return Ok(order.Value);
       }

    }
}
```

خب حالا برنامه را اجرا کنید و سپس API پایین را همانند تصویر در Postman صدا بزنید.

Api: https://localhost:5001/api/v1/orders/1

Verb: GET



www.takbook.com

بعد از اجرا باید نتیجه پایین را ببینیم.


```
{
    "ordernumber": 1,
    "date": "2020-05-18T07:57:43.4098959",
    "status": "submitted",
    "description": null,
    "street": "Navab",
    "city": "Tehran",
    "zipcode": "1111111111",
    "country": "Iran",
    "orderitems": [
        {
            "productname": "Mouse",
            "units": 2,
            "unitprice": 100000,
            "pictureurl": null
        }
    ],
    "total": 200000
}
```

www.takbook.com

فصل ششم : Domain event و Behavior در MediatR و گذری بر جداسازی دیتابیس‌ها

آنچه خواهید آموخت:

- Domain Event چیست؟
- پیاده‌سازی Domain Event
- Behaviorها در MediatR
- تست Domain Event و Behaviorها
- گذری بر جداسازی دیتابیس Read و Write
- استراتژی همگام‌سازی دیتابیس‌ها
- Consistency بین دیتابیس‌ها

www.takbook.com

Domain Event چیست؟

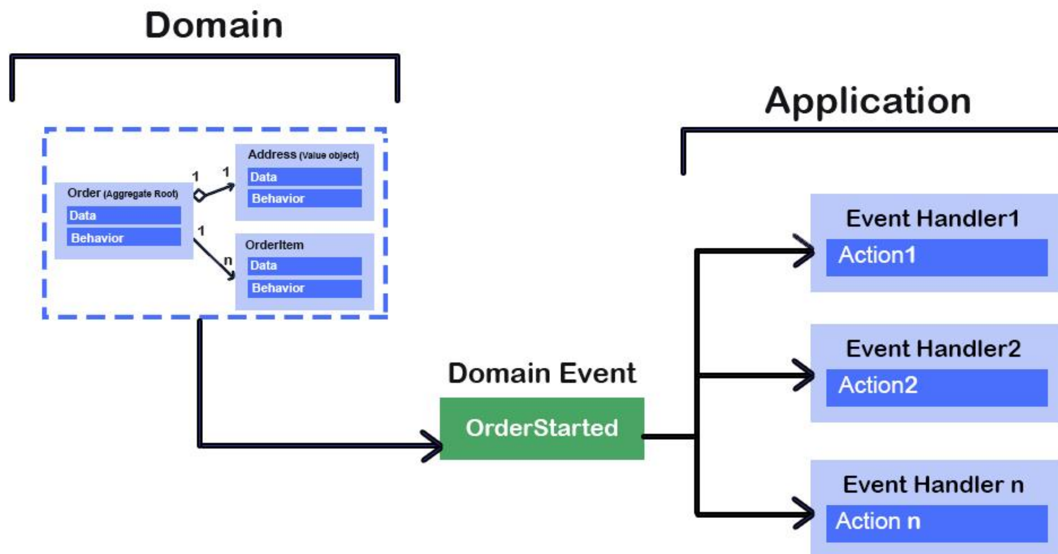
در فصل اول گفتیم Event رویدادی است که در گذشته اتفاق افتاده است اما در مورد اینکه چه مشکلی از اپلیکیشن را حل می‌کند چیزی نگفتیم.

Domain Event اتفاقی است که در Domain افتاده و شما می‌خواهید سایر قسمت‌های همان Domain از آن آگاه شوند و به آن اتفاق واکنش نشان دهند.

فرض کنید می‌خواهید زمانیکه ثبت یک سفارش زده شد، کاربر وارد شده به سیستم را به عنوان یک خریدار اضافه نمایید و سپس یک ایمیل هم به او بفرستید. افزودن خریدار و ارسال ایمیل وظیفه CreateOrderCommand نیست و در صورت افزودن خریدار و ارسال ایمیل در این Command اصل SRP را نقض کرده‌ایم.

برای حل این مشکل می‌توانیم از Domain Eventها به صورت زیر کمک بگیریم :

- هنگام ایجاد یک سفارش، یک پیغام به کلاس Buyer داده شود تا سیستم یک خریدار را ثبت کند.
- سپس یک ایمیل به خریدار ارسال شود.



پیاده‌سازی Domain Event

پیاده‌سازی Domain Event شبیه Command است با این تفاوت که Command فقط یک Handler دارد اما Domain Event می‌تواند چندین Handler داشته باشد.

www.takbook.com

مزیت داشتن چند Handler برای Eventها این است که شما می‌توانید همزمان چندین کار را انجام دهید. **به طور مثال:**

بعد از ایجاد سفارش، یک خریدار را ایجاد و سپس به او یک ایمیل ارسال کنید.

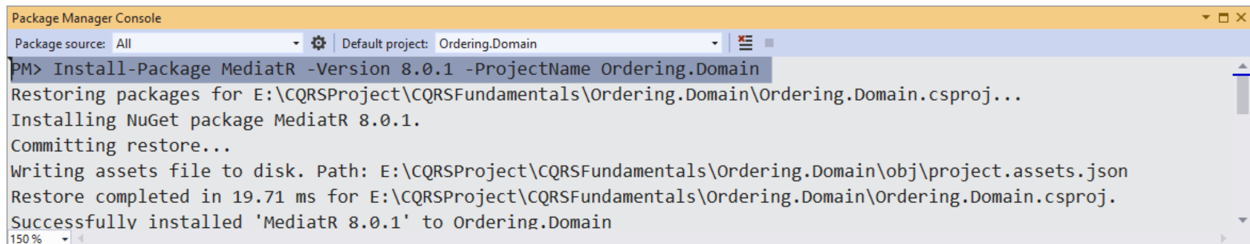
نکته!!

از آنجایی که Event اتفاقی است که در گذشته رخ داده، پس نام آن را با فعل گذشته می‌گذاریم.

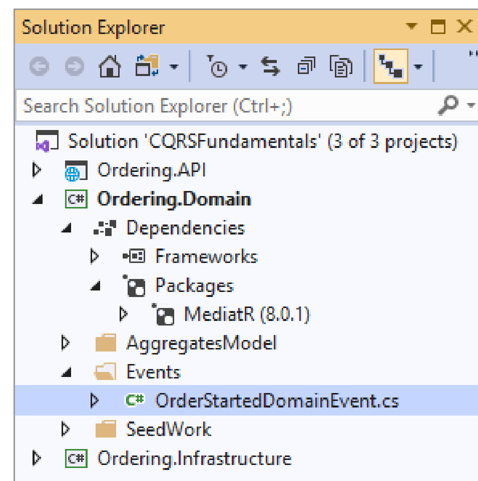
خب حالا برای پیاده‌سازی Domain Event ابتدا وارد پروژه Domain شوید و پکیج MediatR را نصب کنید.

Package Manager Console را باز و دستور پایین را اجرا کنید.

```
Install-Package MediatR -Version 8.0.1 -ProjectName Ordering.Domain
```



حالا در لایه Domain یک فولدر با نام Events ایجاد و سپس یک کلاس با نام OrderStartedDomainEvent به آن اضافه کنید.



در MediatR ارسال و Handle کردن Eventها با دو اینترفیس INotification و INotificationHandler صورت می‌گیرد بنابراین این کلاس باید اینترفیس INotification و Handlerهای آن، باید INotificationHandler راپیاده‌سازی کنند.

www.takbook.com

کدهای کلاس OrderStartedDomainEvent :

```
using MediatR;
using Ordering.Domain.AggregatesModel.OrderAggregate;

namespace Ordering.Domain.Events
{
    public class OrderStartedDomainEvent : INotification
    {
        public string UserId { get; }
        public string UserName { get; }
        public Order Order { get; }

          public OrderStartedDomainEvent(Order order, string userId, string
          userName)
           {
               Order = order;
               UserId = userId;
               UserName = userName;
           }
    }

}
```

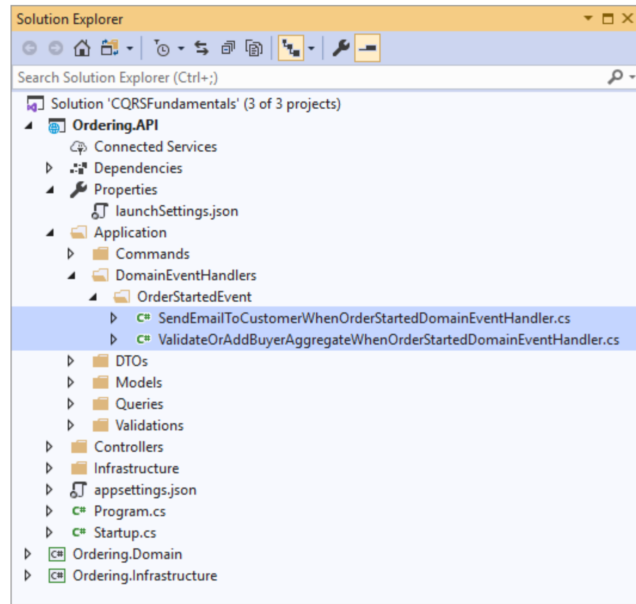
قدم بعدی نوشتن دو Handler برای این Event است. یکی از این Handlerها وظیفه ارسال ایمیل و دیگری افزودن Buyer را بر عهده خواهد داشت.

بنابراین وارد لایه‌ی API شـــوید و در فولدر Application یک فولدر دیگر با نام DomainEventHandlers ایجاد کنید. ســـپس درون فولدر DomainEventHandlers یک فولدر دیگر با نام OrderStartedEvent اضافه نمایید.

حالا درون این فولدر دو کلاس با نام‌های پایین اضافه کنید:

SendEmailToCustomerWhenOrderStartedDomainEventHandler
ValidateOrAddBuyerAggregateWhenOrderStartedDomainEventHandler

www.takbook.com



در اینجا فقط Handler مربوط به افزودن Buyer را توضیح می‌دهیم و Handler بعدی را در Github ببینید.

کدهای کلاس ValidateOrAddBuyerAggregateWhenOrderStartedDomainEventHandler:

```
using MediatR;
using Ordering.Domain.AggregatesModel.BuyerAggregate;
using Ordering.Domain.Events;
using System;
using System.Threading;
using System.Threading.Tasks;

namespace Ordering.API.Application.DomainEventHandlers.OrderStartedEvent
{
    public class
    ValidateOrAddBuyerAggregateWhenOrderStartedDomainEventHandler
                            : INotificationHandler<OrderStartedDomainEvent>
    {
        private readonly IBuyerRepository _buyerRepository;

        public
        ValidateOrAddBuyerAggregateWhenOrderStartedDomainEventHandler(
            IBuyerRepository buyerRepository)
        {
          _buyerRepository = buyerRepository ?? throw new
          ArgumentNullException(nameof(buyerRepository));
        }
```

www.takbook.com

```
        public async Task Handle(OrderStartedDomainEvent orderStartedEvent,
        CancellationToken cancellationToken)
         {
            var buyer = await
            _buyerRepository.FindAsync(orderStartedEvent.UserId);
            bool buyerOriginallyExisted = (buyer == null) ? false : true;

            if (!buyerOriginallyExisted)
            {
             buyer = new Buyer(orderStartedEvent.UserId,
             orderStartedEvent.UserName);
            }

            var buyerUpdated = buyerOriginallyExisted ?
                _buyerRepository.Update(buyer) :
                _buyerRepository.Add(buyer);

            await _buyerRepository.UnitOfWork
                .SaveEntitiesAsync(cancellationToken);
         }
     }

 }
```

قدم سوم این است که چطور این Domain Eventها را Raise کنیم‌تا Handlerهای آن متوجه شوند؟

قبل از پاسخ دادن به این سوال می‌خواهم در مورد مسئله‌ای که ممکن است در کد بوجود بیاید با هم صحبت کنیم.

وقتی یک Domain Event اتفاق می‌افتد باید Handlerهای آن متوجه شود اما چه زمانی این اتفاق باید بیفتد؟؟ **به طور مثال :**

هنگام شروع سفارش، Handler صدا زده شود یا بعد از ذخیره در دیتابیس؟

اگر قبل از Transaction این Handlerها صدا زده شود پس Transaction ما باید منتظر بماند تا تمام این Domain Eventها Raise شوند و اگر تعداد این Domain Eventها زیاد باشند زمان انتظار Transaction افزایش می‌یابد. مزیت این روش این است که اگر یک Domain Event به مسئله برخورد کند Transaction ما هم RollBack خواهد شد.

www.takbook.com

اگر این Handlerها بعد از Transaction صـدا زده شـوند، Transaction ما منتظر نمی‌ماند اما مشـکل این اسـت که اگر هر کدام از Handlerها به مسـئله‌ای برخورد کنند باید به صـورت دسـتی این Transaction را RollBack کنیم.

دومین موضوع Domain Eventها این است که ما در یک Transaction باید Domain Eventها را یکی یکی Raise کنیم یا اینکه همه را یکجا و در زمان SaveChange به Handlerهایشان بفرستیم؟

Raise کردن Domain Eventها به صورت تکی، می‌تواند Side Effectهایی به دنبال داشته باشد و تست Domain Model را سخت کند. اما ارسال گروهی این Eventها در زمان SaveChange، این Side Effectها را از تست Domain Eventها جدا می‌کند. بنابراین یک روش خوب برای انجام این کار این است که Domain Eventها را به یک لیست اضافه کنیم و سپس قبل از Commit کردن Transaction، این مجموعه Domain Event را به Handlerهایشان ارسال نماییم.

نکته!!

تصـمیم گیری درمورد اینکه آیا Domain Event را قبل یا پس از انجام Transaction ارسـال کنید موضـوع بسـیار مهمی اسـت زیرا این موضـوع می‌تواند Side Effectی را به همان Transaction اضـافه کند و یا اینکه Transactionهای دیگر را هم درگیر نماید.
بنابراین با توجه به تصـمیم‌گیری بالا بهتر است درون هر Domain Model لیسـتی از Domain Event داشـته باشیم تا بتوانیم این Event ها را در یکجا متمرکز کنیم.

پس ما این لیسـت Domain Event را درون کلاس پایه Entity قرار می‌دهیم تا در تمام Domain Modelها اضافه شود. سـپس در اینجا برای اضافه شدن، کم شدن و پاک شدن لیست Domain Eventها، متدهایی را هم می‌نویسیم.

کدهای کلاس Entity :

```
using MediatR;
using System;
using System.Collections.Generic;

namespace Ordering.Domain.SeedWork
{
    public abstract class Entity
    {
```

www.takbook.com

```
        int _Id;
        public virtual int Id
        {
            get
            {
                return _Id;
            }
            protected set
            {
                _Id = value;
            }
        }

        public bool IsTransient()
        {
            return this.Id == default(Int32);
        }

        private List<INotification> _domainEvents;
        public IReadOnlyCollection<INotification> DomainEvents =>
        _domainEvents?.AsReadOnly();

        public void AddDomainEvent(INotification eventItem)
        {
            _domainEvents = _domainEvents ?? new List<INotification>();
            _domainEvents.Add(eventItem);
        }

        public void RemoveDomainEvent(INotification eventItem)
        {
            _domainEvents?.Remove(eventItem);
        }

        public void ClearDomainEvents()
        {
            _domainEvents?.Clear();
        }

    }

}
```

www.takbook.com

خب حالا باید از متدهای AddDomainEvent، RemoveDomainEvent و ClearDomainEvents در Aggregate Rootهایتان اســـتفاده کنید(Aggregate Root تنهایی جایی اســـت که ما به Domain Modelهایمان دسترسی داریم):

تغییر کدهای کلاس Order :

```
using Ordering.Domain.Events;
using Ordering.Domain.SeedWork;
using System;
using System.Collections.Generic;
using System.Linq;

namespace Ordering.Domain.AggregatesModel.OrderAggregate
 {
     public class Order : Entity, IAggregateRoot
     {

         private DateTime _orderDate;

         public Address Address { get; private set; }

         public int? GetBuyerId => _buyerId;
         private int? _buyerId;

         public OrderStatus OrderStatus { get; private set; }
         private int _orderStatusId;

         private string _description;

         private bool _isDraft;

         private readonly List<OrderItem> _orderItems;
         public IReadOnlyCollection<OrderItem> OrderItems => _orderItems;

        public static Order NewDraft()
        {
            var order = new Order();
            order._isDraft = true;
            return order;
        }

         protected Order()
```

www.takbook.com

```
    {
        _orderItems = new List<OrderItem>();
        _isDraft = false;
    }

   public Order(string userId, string userName, Address address,
   int? buyerId = null) : this()
     {
          _buyerId = buyerId;
          _orderStatusId = OrderStatus.Submitted.Id;
          _orderDate = DateTime.UtcNow;
          Address = address;
   AddOrderStartedDomainEvent(userId, userName);
      }

       private void AddOrderStartedDomainEvent(string userId, string
        userName)
   {
         var orderStartedDomainEvent = new OrderStartedDomainEvent(this,
         userId,  userName);

       this.AddDomainEvent(orderStartedDomainEvent);
}

     public void AddOrderItem(int productId, string productName,
     decimal unitPrice,  decimal discount, string pictureUrl, int units
     = 1)
       {
          var existingOrderForProduct = _orderItems.Where(o =>
           o.ProductId == productId).SingleOrDefault();

        if (existingOrderForProduct != null)
        {

            if (discount > existingOrderForProduct.GetCurrentDiscount())
            {
                existingOrderForProduct.SetNewDiscount(discount);
            }

            existingOrderForProduct.AddUnits(units);
        }
        else
        {
```

www.takbook.com

```
        var orderItem = new OrderItem(productId, productName,
        unitPrice, discount, pictureUrl, units);
        _orderItems.Add(orderItem);
    }
}

public void SetBuyerId(int id)
{
    _buyerId = id;
}

public void SetAwaitingValidationStatus()
{
    if (_orderStatusId == OrderStatus.Submitted.Id)
    {
        _orderStatusId = OrderStatus.AwaitingValidation.Id;
    }
}

public void SetStockConfirmedStatus()
{
    if (_orderStatusId == OrderStatus.AwaitingValidation.Id)
    {
        _orderStatusId = OrderStatus.StockConfirmed.Id;
         _description = "All the items were confirmed with available
        stock.";
    }
}

public void SetPaidStatus()
{
    if (_orderStatusId == OrderStatus.StockConfirmed.Id)
    {
        _orderStatusId = OrderStatus.Paid.Id;
        _description = "The payment was performed at a simulated
      \"American Bank checking bank account ending on XX35071\"";
    }
}

public void SetShippedStatus()
{
    if (_orderStatusId != OrderStatus.Paid.Id)
```

www.takbook.com

```
    {
        StatusChangeException(OrderStatus.Shipped);
    }

    _orderStatusId = OrderStatus.Shipped.Id;
    _description = "The order was shipped.";
}

public void SetCancelledStatus()
{
    if (_orderStatusId == OrderStatus.Paid.Id ||
        _orderStatusId == OrderStatus.Shipped.Id)
    {
        StatusChangeException(OrderStatus.Cancelled);
    }

    _orderStatusId = OrderStatus.Cancelled.Id;
    _description = $"The order was cancelled.";
}

 public void SetCancelledStatusWhenStockIsRejected(IEnumerable<int>
  orderStockRejectedItems)
 {
    if (_orderStatusId == OrderStatus.AwaitingValidation.Id)
    {
        _orderStatusId = OrderStatus.Cancelled.Id;

        var itemsStockRejectedProductNames = OrderItems
         .Where(c => orderStockRejectedItems.Contains(c.ProductId))
.Select(c => c.GetOrderItemProductName());
         var itemsStockRejectedDescription = string.Join(", ",
         itemsStockRejectedProductNames);
         _description = $"The product items don't have stock:
        ({itemsStockRejectedDescription}).";
    }
}

private void StatusChangeException(OrderStatus orderStatusToChange)
{
  throw new Exception($"Is not possible to change the order status
  from {OrderStatus.Name} to {orderStatusToChange.Name}.");
}
```

www.takbook.com

```
        public decimal GetTotal()
        {
            return _orderItems.Sum(o => o.GetUnits() * o.GetUnitPrice());
        }
    }

}
```

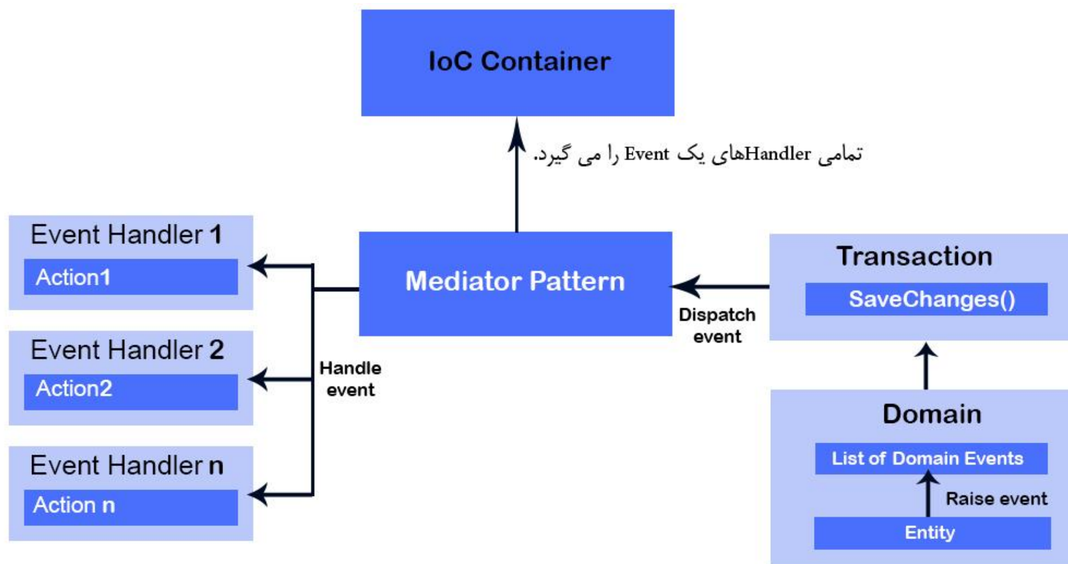
همانطور که در کد بالا می‌بینید :

- ما یک متد OrderStarterDomainEvent داریم که وظیفه‌ی آن اضافه کردن یک Event به لیست Domain Eventهاست.
- این متد را در Constructor صدا می‌زنیم تا زمانیکه یک Order ایجاد شد این لیست برای ما ایجاد و یک Event به آن اضافه شود.

خب تا اینجا لیست Domain Eventها ایجاد شد اما هنوز هیچ Eventی Publish و هیچ Handlerی هم صدا زده نشده است. بنابراین ما باید درون متد SaveChanges کلاس DbContext عملیات Publish را انجام دهیم. با این کار اگر SaveChanges با شکست روبرو شود Transaction تمام تغییرات را RollBack می‌کند.

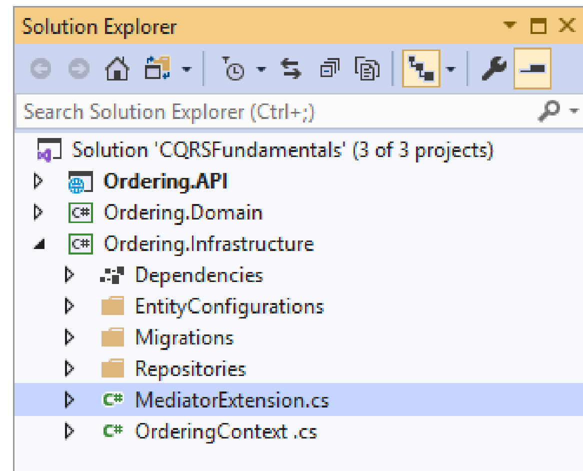
اما چطور عملیات Publish این Eventها را انجام دهیم؟

این کار وظیفه‌ی آبجکت Mediator است. وقتی شما یک یا چند Event را Publish می‌کنید، Mediator وظیفه‌ی Map کردن Handlerهای مرتبط با آن Event را انجام می‌دهد.



www.takbook.com

خب حالا Mediator وارد بازی می‌شـود. پس وارد لایه‌ی Ordering.Infrastructure شـوید و سـپس کلاس MediatorExtension را به ریشه پروژه اضافه کنید.



کدهای کلاس MediatorExtension:

```
using MediatR;
using Ordering.Domain.SeedWork;
using System.Linq;
using System.Threading.Tasks;

namespace Ordering.Infrastructure
{
    static class MediatorExtension
    {
     public static async Task DispatchDomainEventsAsync(this IMediator
     mediator, OrderingContext ctx)
       {
            var domainEntities = ctx.ChangeTracker
                .Entries<Entity>()
                .Where(x => x.Entity.DomainEvents != null &&
                x.Entity.DomainEvents.Any());

            var domainEvents = domainEntities
                .SelectMany(x => x.Entity.DomainEvents)
                .ToList();

            domainEntities.ToList()
                .ForEach(entity => entity.Entity.ClearDomainEvents());

            foreach (var domainEvent in domainEvents)
```

www.takbook.com

```
                await mediator.Publish(domainEvent);
            }
        }

}
```

این کلاس یک Extension Method بر روی IMediatR است که ارسال Event به Handlerها را برعهده دارد.

- ابتدا این Extension Method از طریق ChangeTracker تمام Entityهایی که پراپرتی DomainEvents آن‌ها Null نیست و آیتمی دارد را پیدا می‌کند.
- سپس پراپرتی DomainEvents آن‌ها را در یک متغیر domainEvents نگهداری و بعد Domaineventsهای تمامی Entityها را پاک می‌کند.
- در پایان هم مقادیری که در متغیر domainEvents گذاشته بود را با یک foreach شروع به Publish می‌کند.

خب حالا باید این Extension Method را قبل از عمل SaveChanges صدا بزنید پس وارد DbContext شوید و تغییرات پایین را اعمال کنید.

```
private IMediator _mediator;
public OrderingContext(DbContextOptions<OrderingContext> options, IMediator
mediator) : base(options)
 {
    _mediator = mediator ?? throw new ArgumentNullException(nameof(mediator));

 }

  public async Task<bool> SaveEntitiesAsync(CancellationToken
    cancellationToken = default(CancellationToken))
  {
      await _mediator.DispatchDomainEventsAsync(this);

      var result = await base.SaveChangesAsync(cancellationToken);

      return true;
  }
```

کدهای کلاس OrderingContext :

```
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Storage;
```

www.takbook.com

```
using MediatR;
using Ordering.Domain.AggregatesModel.BuyerAggregate;
using Ordering.Domain.AggregatesModel.OrderAggregate;
using Ordering.Domain.SeedWork;
using Ordering.Infrastructure.EntityConfigurations;
using System;
using System.Data;
using System.Threading;
using System.Threading.Tasks;
namespace Ordering.Infrastructure
{
    public class OrderingContext : DbContext , IUnitOfWork
    {
        public const string DEFAULT_SCHEMA = "Ordering";
    private IMediator _mediator;
    private IDbContextTransaction _currentTransaction;
        public DbSet<Order> Orders { get; set; }
        public DbSet<OrderItem> OrderItems { get; set; }
        public DbSet<Buyer> Buyers { get; set; }
        public DbSet<OrderStatus> OrderStatus { get; set; }


        public  OrderingContext(DbContextOptions<OrderingContext>  options)  :
        base(options) { }

        public OrderingContext(DbContextOptions<OrderingContext> options,
        IMediator mediator) : base(options)
        {
        _mediator = mediator ?? throw new
        ArgumentNullException(nameof(mediator));

         }

        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
          modelBuilder.ApplyConfiguration(new
          OrderEntityTypeConfiguration());
          modelBuilder.ApplyConfiguration(new
          OrderItemEntityTypeConfiguration());
          modelBuilder.ApplyConfiguration(new
          OrderStatusEntityTypeConfiguration());
          modelBuilder.ApplyConfiguration(new
          BuyerEntityTypeConfiguration());
```

www.takbook.com

```
 }

 public IDbContextTransaction GetCurrentTransaction() =>
_currentTransaction;

public bool HasActiveTransaction => _currentTransaction != null;
public async Task<IDbContextTransaction> BeginTransactionAsync()
 {
     if (_currentTransaction != null) return null;

      _currentTransaction = await
      Database.BeginTransactionAsync(IsolationLevel.ReadCommitted);

     return _currentTransaction;
 }

 public async Task CommitTransactionAsync(IDbContextTransaction
 transaction)
 {
     if (transaction == null) throw new
     ArgumentNullException(nameof(transaction));

    if (transaction != _currentTransaction) throw new
     InvalidOperationException($"Transaction
     {transaction.TransactionId} is not current");

    try
     {
         await SaveChangesAsync();
         transaction.Commit();
     }
     catch
      {
          RollbackTransaction();
          throw;
      }
      finally
       {
           if (_currentTransaction != null)
           {
               _currentTransaction.Dispose();
               _currentTransaction = null;
           }
```

www.takbook.com

```
            }
        }

        public void RollbackTransaction()
        {
           try
            {
                _currentTransaction?.Rollback();
            }
            finally
            {
                if (_currentTransaction != null)
                {
                    _currentTransaction.Dispose();
                    _currentTransaction = null;
                }
            }
        }


        public async Task<bool> SaveEntitiesAsync(CancellationToken
        cancellationToken = default(CancellationToken))
        {
            await _mediator.DispatchDomainEventsAsync(this);

                var result = await
                base.SaveChangesAsync(cancellationToken);

            return true;
        }
    }

}
```

نکته!!

توجه داشـــته باشـــید که پراپرتی DomainEvents را در کلاس‌های BuyerEntityTypeConfiguration و OrderEntityTypeConfiguration و OrderItemEntityTypeConfiguration باید Ignore کنید تا این پراپرتی در هنگام Migrate کردن DbContext به ستونی در جداول دیتابیس تبدیل نشود. پس کدهای پایین را به این سه کلاس اضافه نمایید.

کلاس BuyerEntityTypeConfiguration:

```
buyerConfiguration.Ignore(b => b.DomainEvents);
```

www.takbook.com

کلاس OrderEntityTypeConfiguration :

```
orderConfiguration.Ignore(b => b.DomainEvents);
```

کلاس OrderItemEntityTypeConfiguration:

```
orderItemConfiguration.Ignore(b => b.DomainEvents);
```

خب برنامه را اجرا و یکبار دیگر اپلیکیشن را با API پایین تست کنید.

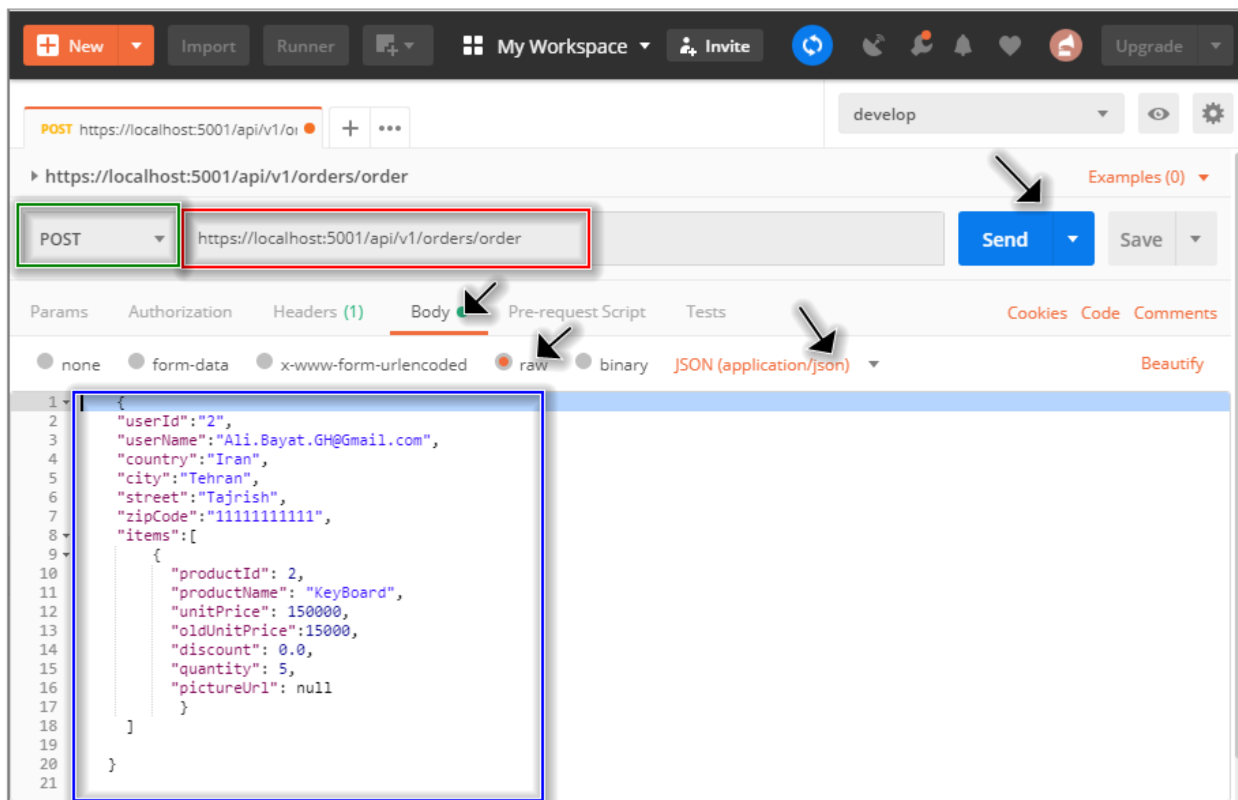
Api: https://localhost:5001/api/v1/orders/order

Verb: POST

Body:

```
{
      "userId":"۲",
      "userName":"Ali.Bayat.GH@Gmail.com",
      "country":"Iran",
      "city":"Tehran",
      "street":"Tajrish",
      "zipCode":"1111111111",
      "items":[
            {
                    "productId": 1,
                    "productName": "KeyBoard",
                    "unitPrice": 150000,
                   "oldUnitPrice":15000,
                    "discount": 0.0,
                    "quantity": 5,
                    "pictureUrl": null
            }
        ]

}
```

www.takbook.com



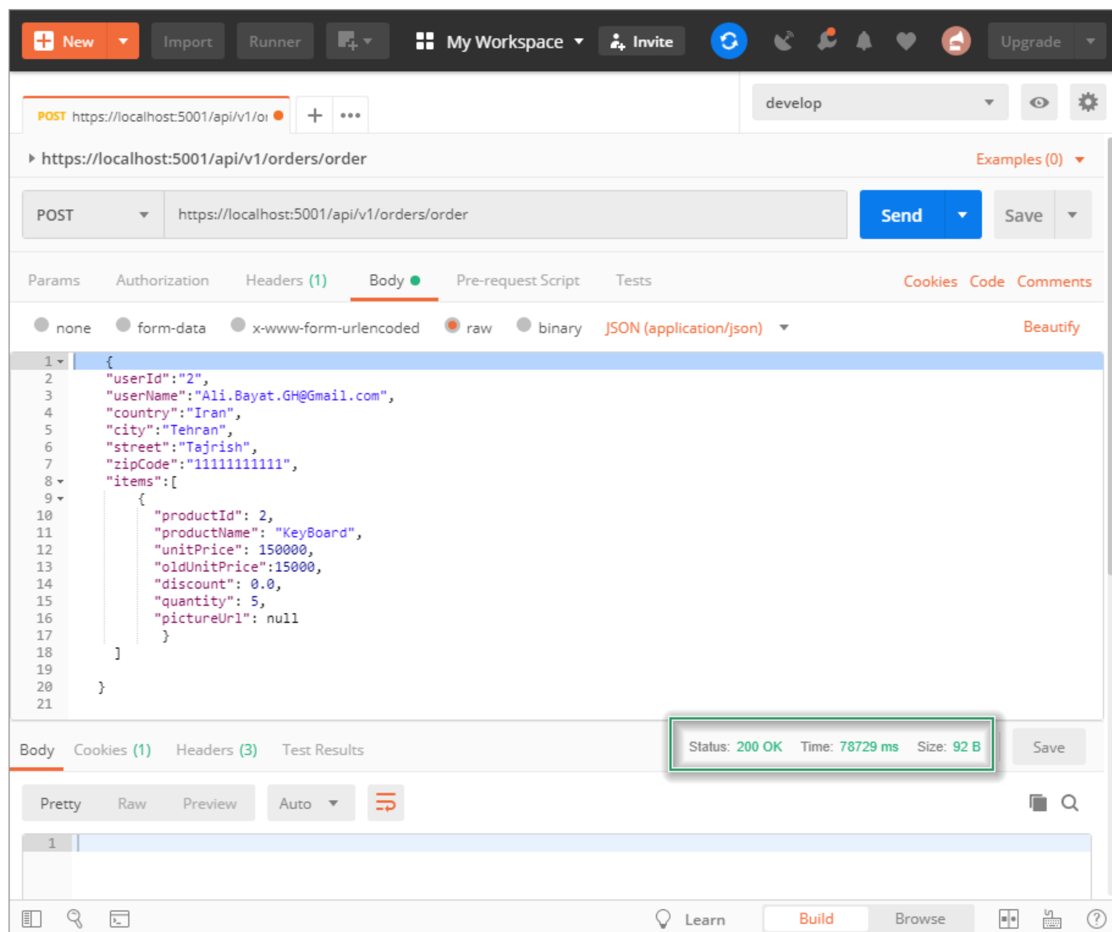
پس از زدن دکمه Send اولین جایی که بعد از Controller وارد می‌شود، متد Handle کلاس CreateOrderCommandHandler است.

زمانیکه به خط SaveEntitiesAsync میرسد وارد متد Handle کلاس ValidateOrAddBuyerAggregateWhenOrderStartedDomainEventHandler می‌شود.

www.takbook.com

```csharp
        public async Task Handle(OrderStartedDomainEvent orderStartedEvent, CancellationToken cancellationToken)
        {
            var buyer = await _buyerRepository.FindAsync(orderStartedEvent.UserId);
            bool buyerOriginallyExisted = (buyer == null) ? false : true;

            if (!buyerOriginallyExisted)
            {
                buyer = new Buyer(orderStartedEvent.UserId, orderStartedEvent.UserName);
            }

            var buyerUpdated = buyerOriginallyExisted ?
                _buyerRepository.Update(buyer) :
                _buyerRepository.Add(buyer);

            await _buyerRepository.UnitOfWork
                .SaveEntitiesAsync(cancellationToken);
        }
    }
}
```

و بعد از اجرا این متد و پایان اجرای برنامه باید خروجی پایین را در Postman ببینید.



حالا نگاهی به دیتابیس می‌اندازیم تا دیتای جداول پایین را ببینیم.

جدول Buyer:

www.takbook.com

Solution1

DESKTOP-N7CC5IR.O...- Ordering.Buyers

	Id	IdentityGuid	Name
▶	1	2	Ali.Bayat.GH@...
*	NULL	NULL	NULL

جدول **Orders**:

DESKTOP-N7CC5IR.O...- Ordering.Orders

	Id	Address_Street	Address_City	Address_Coun...	Address_ZipC...	OrderStatusId	Description	BuyerId	OrderDate
	1	Navab	Tehran	Iran	11111111111	1	NULL	NULL	2020-05-21 02:3...
▶	2	Tajrish	Tehran	Iran	11111111111	1	NULL	NULL	2020-05-21 02:3...
*	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

جدول **OrderItems**:

DESKTOP-N7CC5IR....dering.OrderItems

	Id	ProductId	OrderId	Discount	PictureUrl	ProductName	UnitPrice	Units
	1	1	1	0.00	NULL	Mouse	100000.00	2
▶	2	2	2	0.00	NULL	KeyBoard	150000.00	5
*	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

Behavior چیست؟

قبل از اینکه ببینیم Behavior چیست بیایید مسئله‌ای را با هم بررسی کنیم.

همانطور که می‌دانید از آنجایی که ارتباط بین اپلیکیشن و دیتابیس Stable نیست، ممکن است دیتابیس هر از گاهی آفلاین شود و این باعث شکست اپلیکیشن خواهد شد.

برای رفع این مشکل نیازمند پیاده‌سازی مکانیزمی به نام Retry هستیم. از طرفی هم برای اجرای دستورات درون Command باید عملیات Transaction و Rollback را داشته باشیم تا زمانی که به مشکل ارتباطی با دیتابیس برخوردیم تمامی عملیات Rollback شوند.

بنابراین ما باید در بدنه تمامی متدهایی که با دیتابیس در ارتباط هستند یک try-catch بگذاریم و سپس عملیات Transaction را پیاده‌سازی کنیم.

اما برای پیاده‌سازی این قابلیت ۲ مشکل وجود دارد:

۱) باید کدهای زیادی را در هر متد تکرار کنیم که این باعث افزونگی کد خواهد شد.

www.takbook.com

۲) براساس اصل O، متد باید برای تغییر بسته و برای توسعه باز باشد و اگر ما متد را تغییر دهیم این اصل را زیرپا گذاشته‌ایم.

پس چاره چیست؟

برای حل این مشکل می‌توانیم از Behaviorهای MediatR کمک بگیریم.

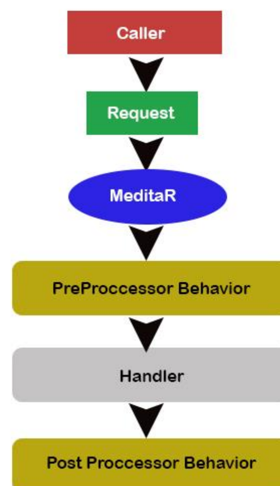
در فریم ورک MediatR یک ویژگی جذاب به نام Behavior وجود دارد که شـــبیه Filterهای ASP.NET MVC اســـت. با اســـتفاده از این ویژگی، شـــما می‌توانید عملیاتی را قبل و بعد از Command و یا Query اجرا کنید.

مزیت اصلی استفاده از این قابلیت، این است که دیگر نیاز به تکرار کد در تک تک Commandها نیست پس به راحتی می‌توان Cross Cutting Concernها را اضافه کرد.

این ویژگی باعث می‌شود:

۱) تا به اصـل Single Responsibility Principle پایبند باشـید چون منطق Retry به یک کلاس مجزا منتقل می‌شود.

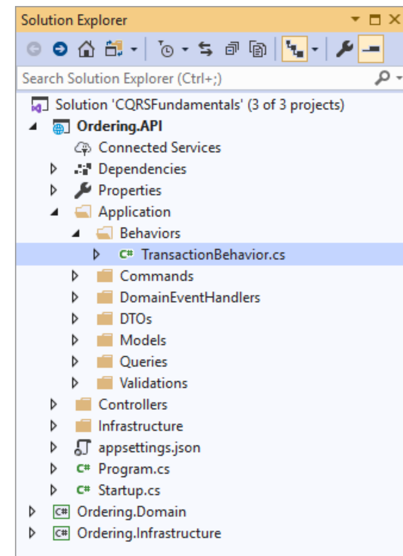
۲) باعث ســـاده نگه داشـــتن Handlerها می‌شـــوید زیرا در هر Handler به جای متمرکز شـــدن بر روی جزییات فنی(مثل اتصال دیتابیس) می‌توانید بر روی بیزینس فوکوس کنید.



در اینجا می‌خواهم با اســـتفاده از Behaviorها به شـــما نشـــان دهم که چطور می‌توان Handlerها را با روشـــی بسیار ساده بهینه‌تر کنیم.

www.takbook.com

بنابراین در فولدر Application یک فولدر به نام Behaviors ایجاد کنید و سپس کلاسی به نام TransactionBehavior را در آن اضافه نمایید.



این کلاس باید اینترفیس IPipelineBehavior را پیاده‌سازی کند.

کدهای کلاس TransactionBehavior :

```
using MediatR;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Logging;
using Ordering.Infrastructure;
using System;
using System.Threading;
using System.Threading.Tasks;

namespace Ordering.API.Application.Behaviors
{
public class TransactionBehavior<TRequest, TResponse> :
IPipelineBehavior<TRequest, TResponse>
 {
     private readonly ILogger<TransactionBehavior<TRequest, TResponse>>
     _logger;
      private readonly OrderingContext _dbContext;

      public TransactionBehavior(OrderingContext dbContext,
          ILogger<TransactionBehavior<TRequest, TResponse>> logger)
      {
```

www.takbook.com

```
        _dbContext = dbContext ?? throw new
        ArgumentException(nameof(OrderingContext));
         _logger = logger ?? throw new ArgumentException(nameof(ILogger));
    }

  public async Task<TResponse> Handle(TRequest request, CancellationToken
   cancellationToken, RequestHandlerDelegate<TResponse> next)
    {
        var response = default(TResponse);
        var typeName = request.GetType();
         for (int i = 0; i < 3; i++)
          {
           try
           {
               if (_dbContext.HasActiveTransaction)
               {
                   return await next();
               }

              var strategy = _dbContext.Database.CreateExecutionStrategy();

               await strategy.ExecuteAsync(async () =>
               {
                Guid transactionId;

                using (var transaction = await
                _dbContext.BeginTransactionAsync())
                {
                    response = await next();
                   _logger.LogInformation("----- Commit transaction
                   {TransactionId} for {CommandName}",
                   transaction.TransactionId, typeName);

                   await _dbContext.CommitTransactionAsync(transaction);
                     transactionId = transaction.TransactionId;
                }
           });

           return response;
         }
        catch (Exception ex)
        {
```

www.takbook.com

```
            _logger.LogError(ex, "ERROR Handling transaction for {CommandName}
            ({@Command})", typeName, request);

             if (i >= 3)
        throw;

            }
        }
        return response;
    }
  }
}
```

همانطور که می‌بینید:

- این کلاس از IPipelineBehavior ارث‌بری و متد Handle آن را پیاده‌سازی کرده است.
- سپس به Constructor آن یک OrderingContext تزریق کردیم تا با استفاده از آن بتوانیم عملیات Transaction را به Commandهایمان اضافه کنیم.
- در ورودی متد Handle نیز همانند Middlewareهای ASP.NET Core یک RequestHandlerDelegate به نام next داریم که با اجرای آن، روند اجرای بقیه Command/Queryها ادامه پیدا خواهد کرد.
- سپس درون متد Handle یک حلقه for گذاشتیم که سه بار عمل اتصال به دیتابیس را تکرار کند.
- درون این for نیاز به یک try-catch داریم که در قسمت try هندلرهای ما را صدا بزند.
- درون catch باید عمل چک کردن تعداد دفعات تلاش برای اتصال دیتابیس انجام شود. اگر تعداد دفعات بیش از ۳ بار باشد باید اکسپشن Throw شود.

بعد از نوشتن کدهای بالا حالا باید Behavior خود را از طریق DI به MediatR معرفی کنیم. پس وارد کلاس Startup شوید و کد پایین را به آن اضافه کنید.

```
services.AddScoped(typeof(IPipelineBehavior<,>),
typeof(TransactionBehavior<,>));
```

کدهای کلاس Startup :

```
using FluentValidation.AspNetCore;
using MediatR;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Configuration;
```

www.takbook.com

```
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Ordering.API.Application.Behaviors;
using Ordering.API.Infrastructure;
using Ordering.API.Application.Models;
using Ordering.Domain.AggregatesModel.BuyerAggregate;
using Ordering.Domain.AggregatesModel.OrderAggregate;
using Ordering.Infrastructure;
using Ordering.Infrastructure.Repositories;
using System.Reflection;
namespace Ordering.API
{
    public class Startup
    {
        public Startup(IConfiguration configuration)
        {
            Configuration = configuration;
        }

        public IConfiguration Configuration { get; }

        public void ConfigureServices(IServiceCollection services)
        {
          services.AddMediatR(Assembly.GetExecutingAssembly());
          services.AddDbContext<OrderingContext>(options =>
          {
          options.UseSqlServer(Configuration["ConnectionString"]);
          });
          services.AddScoped<IOrderRepository, OrderRepository>();
        services.AddScoped<IBuyerRepository, BuyerRepository>();

          services.AddTransient<OrderingContextSeed>();
          var connectionString = new
          ConnectionString(Configuration["ConnectionString"]);
         services.AddSingleton(connectionString);

         services.AddScoped(typeof(IPipelineBehavior<,>),
         typeof(TransactionBehavior<,>));
         services.AddControllers().AddFluentValidation(cfg =>
         cfg.RegisterValidatorsFromAssemblyContaining<Startup>());

        }

    public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
```

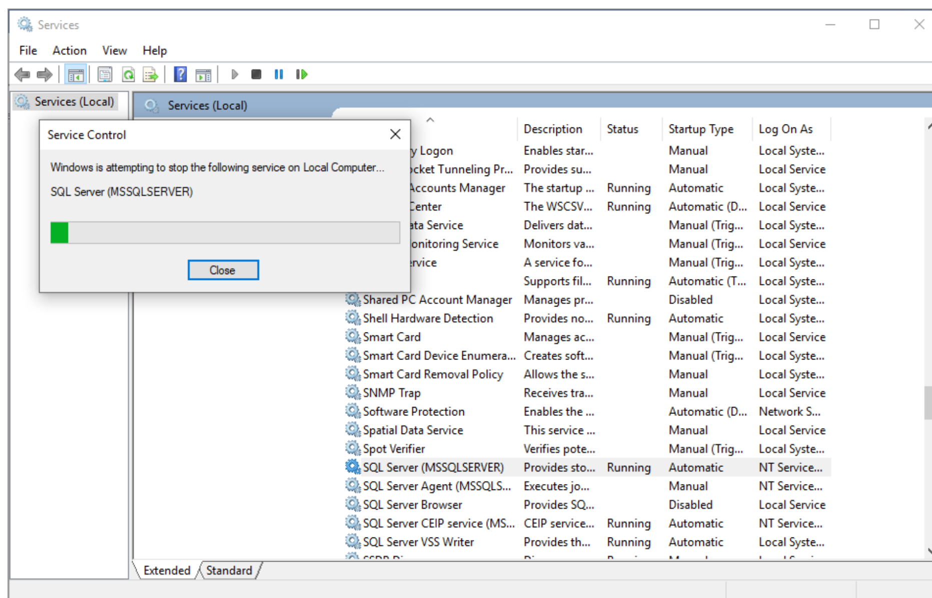
www.takbook.com

```
        {
            if (env.IsDevelopment())
            {
                app.UseDeveloperExceptionPage();
            }

            app.UseHttpsRedirection();
           using (var scope = app.ApplicationServices.CreateScope())
            {
             scope.S--------
             erviceProvider.GetService<OrderingContextSeed>().SeedAsync().Wait
             ();
             }

              app.UseRouting();

              app.UseAuthorization();

              app.UseEndpoints(endpoints =>
              {
                  endpoints.MapControllers();
              });
        }
    }
}
```

تست Behavior

برای تســـت این TransactionBehavior بـایـد Connection دیتـابیس را قطع کنیم. پس Windows Services را باز کنید و SQL Server را Stop نمایید.

www.takbook.com



سپس در متد Handle کلاس TransactionBehavior یک Break Point بگذارید و برنامه را اجرا کنید.

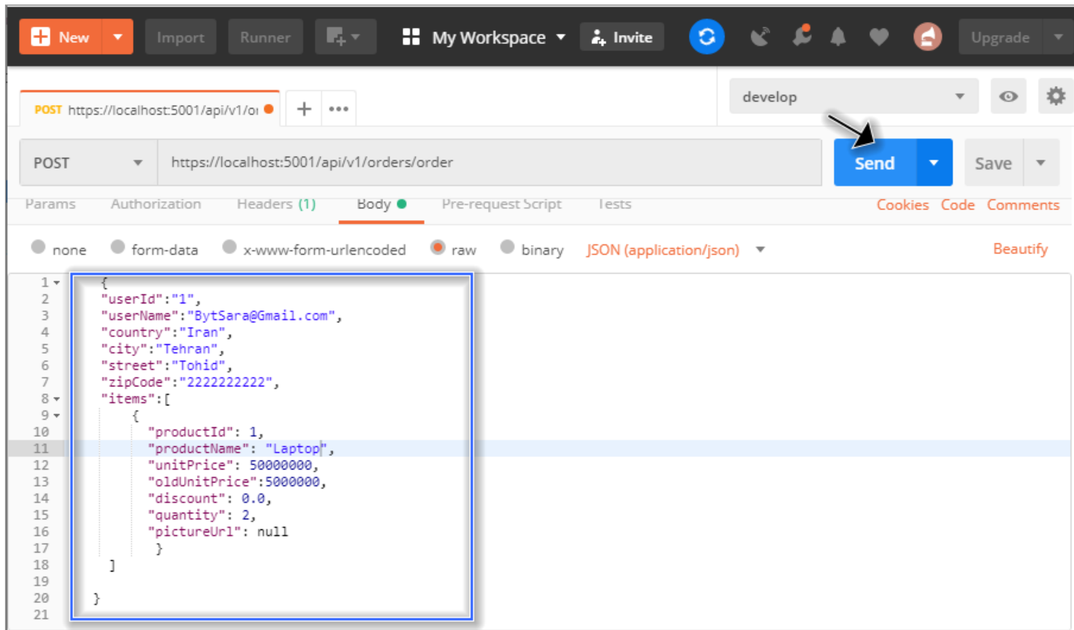
حالا API پایین را در Postman اجرا نمایید.

Api: https://localhost:5001/api/v1/orders/order

Verb: POST

Body:

```
{
  "userId":"1",
  "userName":"BytSara@Gmail.com",
  "country":"Iran",
  "city":"Tehran",
  "street":"Tohid",
  "zipCode":"2222222222",
  "items":[
      {
    "productId": 1,
    "productName": "Laptop",
    "unitPrice": 50000000,
  "oldUnitPrice":5000000,
    "discount": 0.0,
    "quantity": 2,
    "pictureUrl": null
     }
  ]

}
```



همانطور که می‌بینید :

- بعد از زدن دکمه‌ی Send وارد متد Handle کلاس TransactionBehavior می‌شـــوید و بعد از تلاش برای اتصال به دیتابیس وارد catch خواهد شد.
- سپس در catch بررسی می‌کند که عدد متغیر i بزرگتر از ۳ شده یا خیر؟ چون عدد i هنوز ۱ است پس دوباره از ابتدا این متد اجرا می‌شود.

```
0 references
public async Task<TResponse> Handle(TRequest request, CancellationToken cancellationToken, RequestHandlerDelegate<TResponse> next)
{
    var response = default(TResponse);
    var typeName = request.GetType();
    for (int i = 0; i < 3; i++)
    {
        try
        {
            if (_dbContext.HasActiveTransaction)
            {
                return await next();
            }

            var strategy = _dbContext.Database.CreateExecutionStrategy();

            await strategy.ExecuteAsync(async () =>
            {
                Guid transactionId;

                using (var transaction = await _dbContext.BeginTransactionAsync())
                {
                    response = await next();

                    _logger.LogInformation("----- Commit transaction {TransactionId} for {CommandName}", transaction.TransactionId, typeName);

                    await _dbContext.CommitTransactionAsync(transaction);

                    transactionId = transaction.TransactionId;
                }
            });

            return response;
        }
        catch (Exception ex)
        { ≤17,787ms elapsed
            if (i >= 3 )
                throw;
        }
    }
```

حالا دوباره باید از اول حلقه For اجرا شود.

www.takbook.com

```
0 references
public async Task<TResponse> Handle(TRequest request, CancellationToken cancellationToken, RequestHandlerDelegate<TResponse> next)
{
    var response = default(TResponse);
    var typeName = request.GetType();
    for (int i = 0; i < 3; i++)
    {
        try
        {
            if (_dbContext.HasActiveTransaction)
            {
                return await next();
            }

            var strategy = _dbContext.Database.CreateExecutionStrategy();

            await strategy.ExecuteAsync(async () =>
            {
                Guid transactionId;
```

جداسازی دیتابیس

در قسمت‌های قبل در مورد جداسازی API و Domain Model و بهینه شدن اپلیکیشن صحبت کردیم. **به طور مثال**:

به جای اینکه ما APIهای بسیار سنگین CRUD را تجربه کنیم بهتر است اپلیکیشن را به سمت Task Based ببریم و کنترلرها و اینترفیس‌ها را ساده‌تر کنیم. این باعث می‌شود تا API ما از منظر کاربر با مفهوم‌تر شود.

همچنین با حذف Domain Model از Queryها می‌توانیم Queryهای بهینه‌تری داشته باشیم و Domain Model را فقط برای پردازش Commandها متمرکز کنیم.

این دو مورد مزایای اول و دوم CQRS (یعنی Simplicity وPerformance) بودند که ما در اپلیکیشن پیاده‌سازی کردیم. اما باید یک گام دیگر حرکت کنیم تا به سومین مزیت CQRS هم (یعنی Scalability) دست یابیم. بنابراین باید دیتابیس Read را از Write جدا نماییم.

یک سوال؟

شاید بپرسید فراهم کردن Scalability چه مشکلی از ما را حل می‌کند و چرا باید دیتابیس‌ها را از هم جدا کرد؟

پاسخ سوال اول:

در حال حاضر اپلیکیشن ما درون یک سرور قرار دارد پس تنها به منابع یک دستگاه محدود شده و با رشد اپلیکیشن باید این سرور را قوی‌تر کنیم و این ماجرا هزینه بر است.

از طرفی در یک برنامه سازمانی تعداد عملیات خواندن و نوشتن با هم یکی نیست **به طور مثال** :

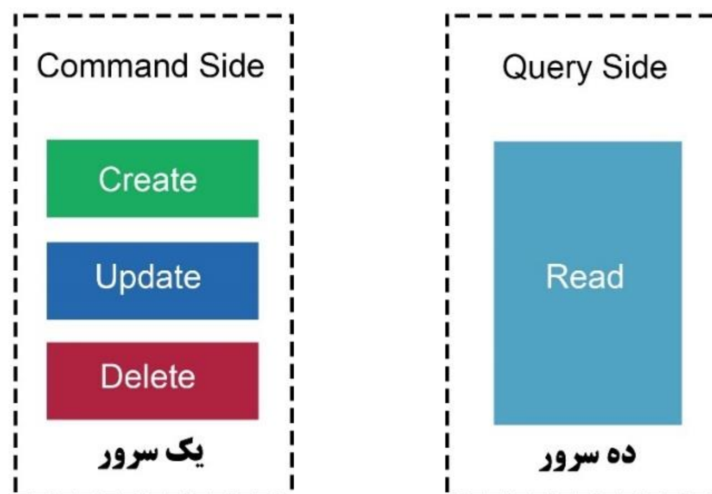
www.takbook.com

همیشه عملیات خواندن اطلاعات از نوشتن، آپدیت و حذف اطلاعات بیشتر درخواست می‌شود پس منطقی است که این عملیات را مستقل از هم Scale کنیم.

در اینجا می‌توان به جای یک سـرور خیلی قوی چند سـرور داشـته باشـیم تا هم هزینه سـرور پایین بیاید و هم عملیات متفاوت را به شیوه‌های مختلف Scale کنیم. **به طور مثال:**

برای عمل Read سرور را قوی‌تر و برای باقی عملیات از سرورهای ضعیف‌تر استفاده کنیم و این چیزی است که Scalability برای ما فراهم می‌کند.

بنابراین Scalability یعنی از منابع چندین سرور به طور بالقوه استفاده نمود.

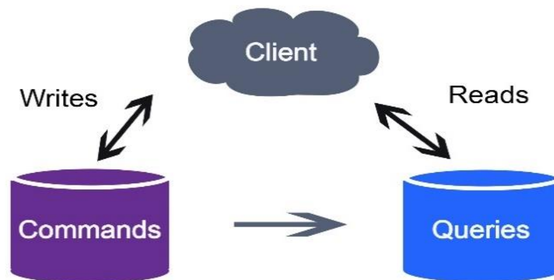


پاسخ سوال دوم:

اما چرا باید دیتابیس‌ها را جدا کنیم؟

همانطور که در پاسخ سوال قبلی گفتم پردازش عملیات Write , Read نامتقارن است بنابراین بهتر است جدا از هم Scale شـوند. اما مشـکل اینجاسـت که اگر دیتابیس هر دو عملیات یکی باشـد ما هر بار مجبوریم کل دیتابیس را Scale کنیم و این یک مشکل برای اپلیکیشن است.

www.takbook.com



خب تا اینجا دلیل جداسازی دیتابیس را متوجه شدیم. **اما برای جداسازی این دو دیتابیس سوالاتی وجود دارد :**

- این جداسازی باعث می‌شود تا دیتابیس Read را بیشتر Scale کنید و بتوانید آن را ساده‌تر طراحی نمایید اما به نظر شما دیتابیس Read باید چه ساختاری داشته باشد؟
- اطلاعات این دیتابیس باید با اطلاعات دیتابیس Command همگام باشد پس در حقیقت باید دیتابیس Query همان ساختار دیتابیس Command را داشته باشد اما همانطور که می‌دانید Read و Write نیازهای متفاوتی هستند و همان ساختار دیتابیس برای جنبه Read ما مناسب نیست. پس **چاره چیست؟**

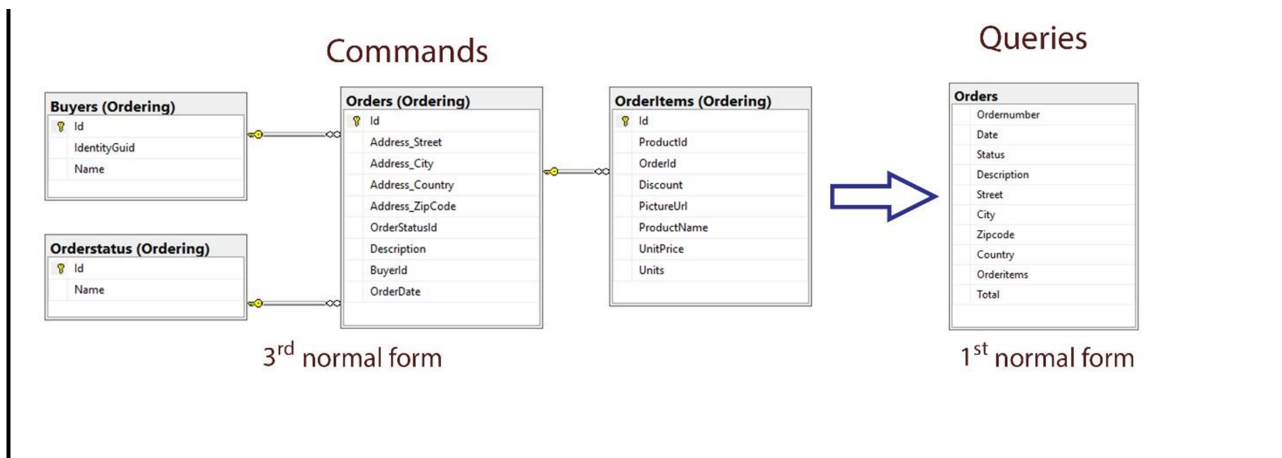
بهترین راه حل برای طراحی دیتابیس Read :

- بهتر است این دیتابیس یک جدول Denormal و Flat داشته باشد که تمام فیلدهای موردنیاز درون آن قرار گیرد.
- سپس با هر بار درج اطلاعات در دیتابیس Command این جدول نیز آپدیت شود. با داشتن این جدول دیگر نیاز به Joinهای عجیب و غریب SQL نداریم.

نتیجه گیری

بنابراین دیتابیس Command با فرم نرمال سطح ۳ مطابقت دارد و برای Transaction مناسب است. در حالی که دیتابیس Read باید Denormalized باشد تا گرفتن داده‌ها با کمترین Join انجام شود.

www.takbook.com



توجه داشته باشید که هر Design Pattern در مقابل مزایایی که ارائه می‌دهد هزینه‌هایی هم دارد که شما باید قبل از پیاده‌سازی، با دقت آن‌ها را در نظر بگیرید و محتاط عمل کنید. **به طور مثال:**

در اکثر اپلیکیشن‌های معمولی، با جداسازی دو مورد قبلی CQRS مزایای زیادی به اپلیکیشن اضافه می‌شود و چندان هزینه‌بر نیست اما جداسازی دیتابیس، پیچیدگی زیادی به دنبال دارد **از جمله :** همگام‌سازی دیتابیس‌Write , Read و حفظ Consistency بین آن‌ها کار ساده ای نیست و در مواردی هم باعث سردگمی کاربر می‌شود. **به طور مثال:**

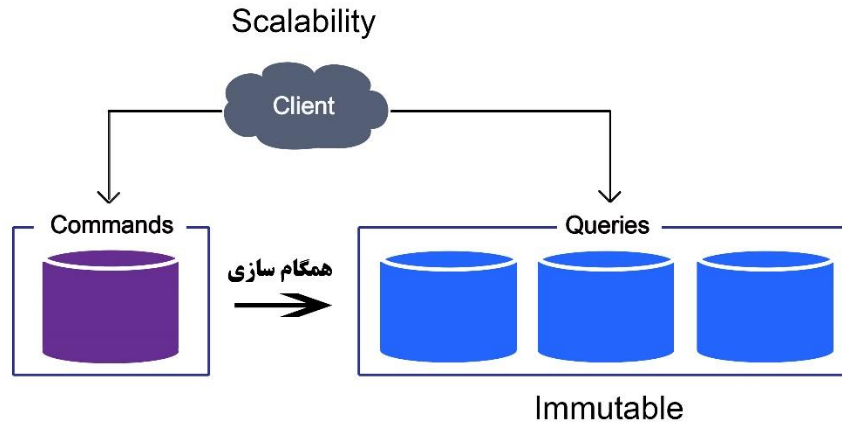
کاربر یک سفارش را اضافه می‌کند و انتظار دارد بعد از درج سفارش، فورا اطلاعات را ببیند اما از آنجایی که اطلاعات در دیتابیس Command ذخیره شده و مدتی زمان نیاز است تا دیتابیس Query آپدیت شود بنابراین کاربر فکر می‌کند که اطلاعات اضافه نشده و دوباره اقدام به درج می‌نماید و همین باعث افزونگی اطلاعات می‌شود.

البته برای این گونه موارد راه کارهایی هم هست **به طور مثال** :

می‌توانید بعد از درج سفارش، پیامی مبنی بر این که سفارش شما با موفقیت ثبت شد و به زودی سیستم آپدیت می‌شود را به کاربر نمایش دهید. این باعث می‌شود افزونگی به حداقل برسد.

بنابراین تا اینجا متوجه شدید که داشتن دیتابیس جداگانه هزینه‌های نگهداری و سازگاری اطلاعات را پی دارد و در بیشتر موارد بهتر است جداسازی انجام نشود مگر اینکه شما قصد Scale سیستم خود را داشته باشید.

www.takbook.com



استراتژی‌های همگام سازی دیتابیس

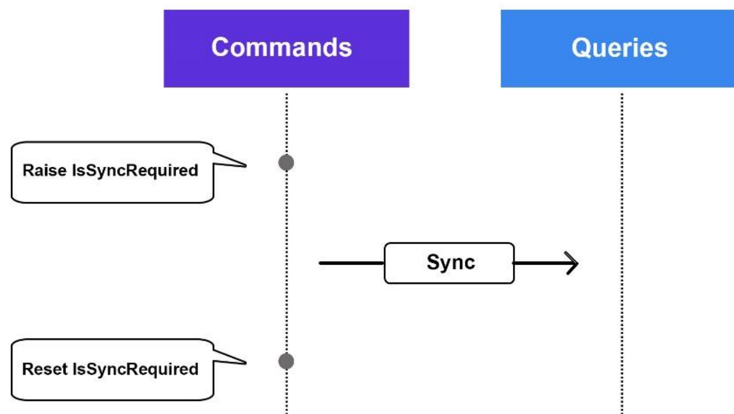
تا اینجا دیدیم که اگر قصد جداسازی دیتابیس را دارید باید عملیات همگام‌سازی آن‌ها را هم در نظر بگیرید. ما در اینجا می‌خواهیم دو استراتژی برای پیاده‌سازی این همگام‌سازی را بررسی کنیم :

۱) **همگام سازی با State** : این استراتژی بسیار ساده است. در این همگام‌سازی باید در هر جدول یک Flag بگذارید که وظیفه‌اش این است که اعلام کند آیا این Entity از آخرین همگام‌سازی تاکنون آپدیت شده یا خیر؟ **به طور مثال**:

در هر Entity یک پراپرتی به نام IsSyncRequired اضافه می‌کنید تا هر زمان در جدول Write تغییری صورت گرفت مقدار این فیلد به True تغییر کند و سپس دیتابیس Read با توجه به این تغییر، آپدیت شود. بعد از آپدیت دیتابیس هم دوباره این فیلد باید False شود.

با این روش کاربردی همیشه دیتابیس Read بعد از تغییرات دیتابیس Write سریعا آپدیت خواهد شد.

همگام سازی با State



www.takbook.com

در این روش یک چالش وجود دارد:

اگر یک ردیف جدول حذف شود دیتابیس Read آپدیت نمی‌شود چون فیلد IsSyncRequired آن ردیف، دیگر در جدول Write وجود ندارد در حالیکه در جدول Read این ردیف همچنان باقی است. بنابراین چاره چیست؟

راه حل این مسئله استفاده از SoftDelete به جای HardDelete است. یعنی به جای اینکه ردیف جدول را به طور کامل از حافظه حذف کنیم، یک فیلد IsDeleted در هر ردیف می‌گذاریم که اگر مقدار آن True شد یعنی این ردیف حذف منطقی شده است پس فیلد IsSyncRequired هم True می‌شود و دیتابیس Read مطلع این حذف خواهد شد.

> **نکته!!**
>
> **از آنجاییکه مدل‌های Read وWrite بلافاصله با هم یکی نمی‌شوند، معمولا کاربران دچار اشتباه خواهند شد و دوباره درخواست Write را می فرستند. بنابراین حتما پیغام مناسب را به کاربر نمایش دهید. یا اگر کاربران شما مخالف چنین رفتاری هستند پس دیتابیس‌ها را جدا نکنید.**

۱) **همگام‌سازی با Eventها :** استراتژی بعدی برای همگام شدن دیتابیس‌ها، استفاده از Eventهاست.
در این همگام‌سازی هر زمان که دیتابیس Write تغییر کند یک Event فرستاده می‌شود و سپس دیتابیس Read با استفاده از اطلاعات آن Event، خود را آپدیت می‌کند.

مزایا و معایب این روش:

- این روش نسبت به قبلی، Scale خیلی خوبی دارد چون اپلیکیشن Domain Eventها را یکجا Publish می‌کند و شما می‌توانید هر تعداد Handler که می‌خواهید به آن Subscribe کنید.
- اما این رویکرد یک ایراد مهم دارد و آن این است که اگر مشکلی در دیتابیس Read بوجود بیایید، ReBulid دیتابیس کار ساده‌ای نیست.
- و دومین مشکل اینکه اگر این Eventهارا جایی ذخیره نکنید و به هر دلیلی یک Domain Event را از دست دهید، دیتابیس‌ها از حالت Sync بیرون می‌آیند.

راه حل این مسئله، ذخیره Domain Eventها در دیتابیس است بنابراین باید یک معماری Event Sourcing داشته باشید.

> **نکته!!**

www.takbook.com

گاهی افراد فکر می‌کنند که CQRS باید با Event Sourcing پیاده شود در حالیکه CQRS به تنهایی قابل استفاده است و گاهی استفاده از Event Sourcing همراه با آن باعث افزودن پیچیدگی هم می‌شود.

معمولا سیستم‌هایی که از Event Sourcing استفاده می‌کنند سیستم‌های خاصی هستند که معمولا مواردی در آن‌ها پیش میاید که باید Eventها را پیگیری کنند. برای مثال یک سیستم مالی.

و نکته دوم اینکه برای اجرای CQRS نیازی به Event Sourcing نیست اما برای اجرای Event Sourcing باید CQRS باشد چون Event Sourcing بدون جدا کردن Read و Write کمترین scalable را به همراه دارد.

Consistency بین دیتابیس‌ها

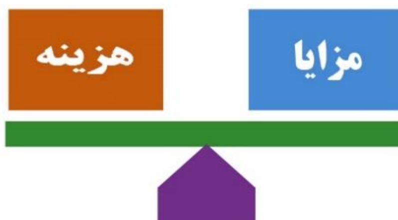
شاید برای شما هم این سوال پیش آمده باشد: اگر در یک Command به کوئری نیاز داشتیم باید از دیتابیس Read اطلاعات را بگیریم یا Write ؟ برای مثال:

قبل از درج مشتری نیاز است بررسی کنیم که شخصی با این مشخصات وجود دارد خیر؟ اگر وجود نداشت درج کند در غیر این صورت کاری انجام ندهد.

برای پاسخ به این سوال باید بگویم که شما همیشه باید از دیتابیس Command اطلاعات را بخوانید چون دیتابیس Read از روی دیتابیس Write اطلاعات خود را آپدیت می‌کند و ممکن است زمانی که نیاز به خواندن اطلاعات از دیتابیس Read داشته باشیم، این دیتابیس هنوز آپدیت نشده باشد.

نکته!!

لازم نیست که تمام تکنیک‌هایی که در این کتاب گفته شده را در پروژه خود پیاده سازی کنید زیرا در کنار مزایای هر دیزاین پترنی یک سری هزینه‌هایی وجود دارد که شما باید مطمئن شوید که استفاده از این مزایا در مقابل هزینه‌های آن به صرفه هست یا خیر؟



www.takbook.com

Quiz

1. What is a command?

- **A.** A message that informs external applications about some change.
- **B.** A message that asks the application about something.
- **C.** A message that tells the application to do something.

2. What is a query?

- **A.** A message that tells the application to do something.
- **B.** A message that asks the application about something.
- **C.** A message that informs external applications about some change.

3. How should you name commands?

- **A.** They should always be in the past tense.
- **B.** They should always have a "Command" postfix.
- **C.** They should always be in the imperative tense.
- **D.** They should aways start with "Get".

4. How should you name queries?

- **A.** They should always be in the imperative tense.
- **B.** They should always be in the past tense.
- **C.** They should always have a "Query" postfix.
- **D.** They should aways start with "Get".

5. How can you reuse code in command handlers?

- **A.** Dispatch a command inside the other command's hander.
- **B.** Extract it into a domain service.
- **C.** Nohow, you have to duplicate the code.

6. How should you model the database for reads as opposed to the database for writes?

www.takbook.com

A. It should have a low normal form (usually, the 1st).

B. It should always be a document database, not relational.

C. It should have a high normal form. (usually, the 3rd).

7. What is the end goal of CQRS?

A. To allow for making different choices in different situations

B. To reduce complexity, improve performance, and enable scalability

C. To introduce two models for reads and writes where, previously, there was only one

8. What is the problem with using a unified model for reads and writes?

A. Such a model takes too much development time to implement.

B. Such a model is unable to handle any of them well.

C. Such a model is impossible to come up with.

9. What does the word "interface" in the notion CRUD-based interface refer to?

A. .NET interfaces

B. API and UI

C. UI

D. API

10. What is the difference between a commands in CQS and a commands in CQRS?

A. A CQS command is a method; a CQRS command is a class.

B. A CQS command is a class; a CQRS command is a method.

C. A CQS command can be either a class or a method; a CQRS command can only be a class.

D. A CQS command can only be a method; a CQRS command can be either a class or a method.

www.takbook.com

11. What is an example of an asynchronous state-driven projection in the real world?

A. Stored procedures.

B. Indexed views.

C. Database replication.

12. Which of the SOLID principles the task-based interface allows you to adhere to?

A. Single responsibility principle.

B. Dependency inversion principle.

C. Liskov substitution principle.

D. Open/closed principle.

13. How many layers of the Onion Architecture should a read model implement?

A. At lest the core domain and the non-core domain layers.

B. At least the core domain layer.

C. All: the core domain, the non-core domain, and the application services layers.

D. None, the read model should be flat and provide no abstraction over the database.

14. What is an example of a synchronous state-driven projection in the real world?

A. Database replication.

B. Database triggers.

C. Stored procedures.

D. Indexed views.

15. When should you apply CQRS versus the Specification pattern?

www.takbook.com

A. In large projects where the loose coupling is more important than DRY.

B. You should always prefer CQRS over the Specification pattern.

C. You should always prefer the Specification pattern over CQRS.

D. In small projects where DRY is more important than the loose coupling.

16. Among the following choices, what is NOT an example of applying the CQRS pattern at the data level?

A. Database replication

B. A task-based interface

C. A full-text search engine

D. Indexed views

17. What happens if a decorator modifies the code it decorates?

A. Public interface.

B. Nothing.

C. Behavior.

D. Both behavior and public interface.

18. When should you use ASP.NET middleware over custom-written decorators?

A. For ASP.NET-related concerns that are cross-cutting.

B. For any ASP.NET-related concerns.

C. For ASP.NET-related concerns that are ubiquitous and cross-cutting.

19. When should you introduce DTOs in addition to commands?

A. Only if you need to keep the application backward-compatible.

B. Always.

C. Only if you want to adhere to the Onion Architecture.

www.takbook.com

20. How should you implement the separation at the domain level between commands and queries?

A. Get rid of the domain model in the queries.

B. Get rid of the domain model in the commands.

C. Introduce its own Onion with multiple layers in the queries.

21. What is the difference between scalability and performance?

A. Scalability allows to harness resources of multiple servers.

B. Scalability allows to make the best use of the resources of a single server.

C. Scalability and performance are synonyms.

22. What is the relationship between CRUD-based interface and CRUD-based thinking?

A. CRUD-based interface leads to CRUD-based thinking.

B. CRUD-based interface and CRUD-based thinking are unrelated.

C. CRUD-based interface is required to introduce CRUD API operations, while CRUD-based thinking is not.

D. CRUD-based thinking leads to CRUD-based interface.

23. What is the biggest problem with CRUD-based interface?

A. An uncontrolled growth of complexity

B. A disconnect between how the domain experts view the system and how it is implemented

C. A lack of ubiquitous language

D. Potential performance issues

24. What is an event?

A. A message that informs external applications about some change.

B. A message that asks the application about something.

www.takbook.com

C. A message that tells the application to do something.

25. Where do commands and queries belong in the Onion Architecture?

A. In the core domain layer.

B. In the non-core domain layer.

C. In the application services layer.

26. How should you organize command handlers and ASP.NET controllers?

A. Either of them can handle any type of logic.

B. Command handlers should handle ASP.NET-related concerns, controllers - application-related.

C. Controllers should handle ASP.NET-related concerns, command handlers - application-related.

D. You should get rid of ASP.NET controllers and keep all their logic in command handlers.

27. Why is CRUD-based interface widely spread?

A. Because it helps simplify the code base

B. Because it is a commonly accepted best practice

C. Because of the programmers' desire to unify everything

www.takbook.com

Answers

Answer 1: C

Answer 2: B

Answer 3: C

Answer 4: D

Answer 5: B

Answer 6: A

Answer 7: B

Answer 8: B

Answer 9: B

Answer 10: A

Answer 11: C

Answer 12: A

Answer 13: D

Answer 14: D

Answer 15: A

Answer 16: B

Answer 17: C

Answer 18: C

Answer 19: A

Answer 20: A

Answer 21: A

Answer 22: D

Answer 23: A

Answer 24: A

Answer 25: A

Answer 26: C

Answer 27: C

www.takbook.com