عنوان: **EF Code First #11**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۱/۰۲/۲۴ ۲۲:۵۱:۰۰
آدرس: www.dotnettips.info
گروه‌ها: Entity framework

استفاده از الگوی Repository اضافی در EF Code first؛ آری یا خیر؟!

اگر در ویژوال استودیو، اشاره‌گر ماوس را بر روی تعریف DbContext قرار دهیم، راهنمای زیر ظاهر می‌شود:

```
A DbContext instance represents a combination of the Unit Of Work and Repository patterns such that
it can be used to query from a database and group together changes that will then be written back to
the store as a unit.  DbContext is conceptually similar to ObjectContext.
```

در اینجا تیم EF صراحتا عنوان می‌کند که DbContext در EF Code first همان الگوی Unit Of Work را پیاده سازی کرده و در داخل کلاس مشتق شده از آن، DbSetها همان Repositories هستند (فقط نام‌ها تغییر کرده‌اند؛ اصول یکی است).
به عبارت دیگر با نام بردن صریح از این الگوها، مقصود زیر را دنبال می‌کنند:
لطفا بر روی این لایه Abstraction ایی که ما تهیه دیده‌ایم، یک لایه Abstraction دیگر را ایجاد نکنید!
«لایه Abstraction دیگر» یعنی پیاده سازی الگوهای Unit Of Work و Repository جدید، برفراز الگوهای Unit Of Work و Repository توکار موجود!
کار اضافه‌ای که در بسیاری از سایت‌ها مشاهده می‌شود و ... متاسفانه اکثر آن‌ها هم اشتباه هستند! در ذیل روش‌های تشخیص پیاده سازی‌های نادرست الگوی Repository را بر خواهیم شمرد:

1) قرار دادن متد Save تغییرات نهایی انجام شده، در داخل کلاس Repository

متد Save باید داخل کلاس Unit of work تعریف شود نه داخل کلاس Repository. دقیقا همان کاری که در EF Code first به درستی انجام شده. متد SaveChanges توسط DbContext ارائه می‌شود. علت هم این است که در زمان Save ممکن است با چندین Entity و چندین جدول مشغول به کار باشیم. حاصل یک تراکنش، باید نهایتا ذخیره شود نه اینکه هر کدام از این‌ها، تراکنش خاص خودشان را داشته باشند.

2) نداشتن درکی از الگوی Unit of work

به Unit of work به شکل یک تراکنش نگاه کنید. در داخل آن با انواع و اقسام موجودیت‌ها از کلاس‌ها و جداول مختلف کار شده و حاصل عملیات، به بانک اطلاعاتی اعمال می‌گردد. پیاده سازی‌های اشتباه الگوی Repository، تمام امکانات را در داخل همان کلاس Repository قرار می‌دهند؛ که اشتباه است. این نوع کلاس‌ها فقط برای کار با یک Entity بهینه شده‌اند؛ در حالیکه در دنیای واقعی، اطلاعات ممکن است از دو Entity مختلف دریافت و نتیجه محاسبات مفروضی به Entity سوم اعمال شود. تمام این عملیات یک تراکنش را تشکیل می‌دهد، نه اینکه هر کدام، تراکنش مجزای خود را داشته باشند.

3) وهله سازی از DbContext به صورت مستقیم داخل کلاس Repository

4) Dispose اشیاء DbContext داخل کلاس Repository

هر بار وهله سازی DbContext مساوی است با باز شدن یک اتصال به بانک اطلاعاتی و همچنین از آنجائیکه راهنمای ذکر شده فوق را در مورد DbContext مطالعه نکرده‌اند، زمانیکه در یک متد با سه وهله از سه Repository موجودیت‌های مختلف کار می‌کنید، سه تراکنش و سه اتصال مختلف به بانک اطلاعاتی گشوده شده است. این مورد ذاتا اشتباه است و سربار بالایی را نیز به همراه دارد.
ضمن اینکه بستن DbContext در یک Repository، امکان اعمال کوئری‌های بعدی LINQ را غیرممکن می‌کند. به ظاهر یک شیء IQueryable در اختیار داریم که می‌توان بر روی آن انواع و اقسام کوئری‌های LINQ را تعریف کرد اما ... در اینجا با LINQ to Objects که بر روی اطلاعات موجود در حافظه کار می‌کند سر و کار نداریم. اتصال به بانک اطلاعاتی با بستن DbContext قطع شده، بنابراین کوئری LINQ بعدی شما کار نخواهد کرد.
همچنین در EF نمی‌توان یک Entity را از یک Context به Context دیگری ارسال کرد. در پیاده سازی صحیح الگوی Repository (دقیقا همان چیزی که در EF Code first به صورت توکار وجود دارد)، Context باید بین Repositories که در اینجا فقط نامش

DbSet تعریف شده، به اشتراک گذاشته شود. علت هم این است که EF از Context برای ردیابی تغییرات انجام شده بر روی موجودیت‌ها استفاده می‌کند (همان سطح اول کش که در قسمت‌های قبل به آن اشاره شد). اگر به ازای هر Repository یکبار وهله سازی DbContext انجام شود، هر کدام کش جداگانه خاص خود را خواهند داشت.
5) عدم امکان استفاده از تنها یک DbConetext به ازای یک Http Request
هنگامیکه وهله سازی DbContext به داخل یک Repository منتقل می‌شود و الگوی واحد کار رعایت نمی‌گردد، امکان به اشتراک گذاری آن بین Repositoryهای تعریف شده وجود نخواهد داشت. این مساله در برنامه‌های وب سبب کاهش کارآیی می‌گردد (باز و بسته شدن بیش از حد اتصال به بانک اطلاعاتی در حالیکه می‌شد تمام این عملیات را با یک DbContext انجام داد).

نمونه‌ای از این پیاده سازی اشتباه را در اینجا می‌توانید پیدا کنید. متاسفانه شبیه به همین پیاده سازی، در پروژه MVC Scaffolding نیز بکارگرفته شده است.

چرا تعریف لایه دیگری بر روی لایه Abstraction موجود در EF Code first اشتباه است؟

یکی از دلایلی که حین تعریف الگوی Repository دوم بر روی لایه موجود عنوان می‌شود، این است:
« **به این ترتیب به سادگی می‌توان ORM مورد استفاده را تغییر داد** » چون پیاده سازی استفاده از ORM، در پشت این لایه مخفی شده و ما هر زمان که بخواهیم به ORM دیگری کوچ کنیم، فقط کافی است این لایه را تغییر دهیم و نه کل برنامه را.
ولی سؤال این است که هرچند این مساله از هزار فرسنگ بالاتر درست است، اما واقعا تابحال دیده‌اید که پروژه‌ای را با یک ORM شروع کنند و بعد سوئیچ کنند به ORM دیگری؟!
ضمنا برای اینکه واقعا لایه اضافی پیاده سازی شده انتقال پذیر باشد، شما باید کاملا دست و پای ORM موجود را بریده و توانایی‌های در دسترس آن را به سطح نازلی کاهش دهید تا پیاده سازی شما قابل انتقال باشد. برای مثال یک سری از قابلیت‌های پیشرفته و بسیار جالب در NH هست که در EF نیست و برعکس. آیا واقعا می‌توان به همین سادگی ORM مورد استفاده را تغییر داد؟ فقط در یک حالت این امر میسر است: از قابلیت‌های پیشرفته ابزار موجود استفاده نکنیم و از آن در سطحی بسیار ساده و ابتدایی کمک بگیریم تا از قابلیت‌های مشترک بین ORMهای موجود استفاده شود.
ضمن اینکه مباحث نگاشت کلاس‌ها به جداول را چکار خواهید کرد؟ EF راه و روش خاص خودش را دارد، NH چندین و چند روش خاص خودش را دارد! این‌ها به این سادگی قابل انتقال نیستند که شخصی عنوان کند: «هر زمان که علاقمند بودیم، ORM مورد استفاده را می‌شود عوض کرد!»

دلیل دومی که برای تهیه لایه اضافه‌تری بر روی DbContext عنوان می‌کنند این است:
« **با استفاده از الگوی Repository نوشتن آزمون‌های واحد ساده‌تر می‌شود** ». زمانیکه برنامه بر اساس Interfaceها کار می‌کند می‌توان آن‌ها را بجای اشاره به بانک اطلاعاتی، به نمونه‌ای موجود در حافظه، در زمان آزمون تغییر داد.
این مورد در حالت کلی درست است اما .... نه در مورد بانک‌های اطلاعاتی!
زمانیکه در یک آزمون واحد، پیاده سازی جدیدی از الگوی Interface مخزن ما تهیه می‌شود و اینبار بجای بانک اطلاعاتی با یک سری شیء قرارگرفته در حافظه سروکار داریم، آیا موارد زیر را هم می‌توان به سادگی آزمایش کرد؟
ارتباطات بین جداول‌را، cascade delete، فیلدهای identity، فیلدهای unique، کلیدهای ترکیبی، نوع‌های خاص تعریف شده در بانک اطلاعاتی و مسایلی از این دست.
پاسخ: خیر! تغییر انجام شده، سبب کار برنامه با اطلاعات موجود در حافظه خواهد شد، یعنی LINQ to Objects.
شما در حالت استفاده از LINQ to Objects آزادی عمل فوق العاده‌ای دارید. می‌توانید از انواع و اقسام متدها حین تهیه کوئری‌های LINQ استفاده کنید که هیچکدام معادلی در بانک اطلاعاتی نداشته و ... به ظاهر آزمون واحد شما پاس می‌شود؛ اما در عمل بر روی یک بانک اطلاعاتی واقعی کار نخواهد کرد.
البته شاید شخصی عنوان که بله می‌شود تمام این‌ها نیازمندی‌ها را در حالت کار با اشیاء درون حافظه هم پیاده سازی کرد ولی ... در نهایت پیاده سازی آن بسیار پیچیده و در حد پیاده سازی یک بانک اطلاعاتی واقعی خواهد شد که واقعا ضرورتی ندارد.

و پاسخ صحیح در اینجا و این مساله خاص این است:
لطفا در حین کار با بانک‌های اطلاعاتی مباحث mocking را فراموش کنید. بجای SQL Server، رشته اتصالی و تنظیمات برنامه را به SQL Server CE تغییر داده و آزمایشات خود را انجام دهید. پس از پایان کار هم بانک اطلاعاتی را delete کنید. به این نوع آزمون‌ها اصطلاحا integration tests گفته می‌شود. لازم است برنامه با یک بانک اطلاعاتی واقعی تست شود و نه یک سری شیء ساده

قرار گرفته در حافظه که هیچ قیدی همانند شرایط کار با یک بانک اطلاعاتی واقعی، بر روی آن‌ها اعمال نمی‌شود.
ضمنا باید درنظر داشت بانک‌های اطلاعاتی که تنها در حافظه کار کنند نیز وجود دارند. برای مثال SQLite حالت کار کردن صرفا در حافظه را پشتیبانی می‌کند. زمانیکه آزمون واحد شروع می‌شود، یک بانک اطلاعاتی واقعی را در حافظه تشکیل داده و پس از پایان کار هم ... اثری از این بانک اطلاعاتی باقی نخواهد ماند و برای این نوع کارها بسیار سریع است.

نتیجه گیری:
حین استفاده از EF code first، الگوی واحد کار، همان DbContext است و الگوی مخزن، همان DbSetها. ضرورتی به ایجاد یک لایه محافظ اضافی بر روی این‌ها وجود ندارد.
در اینجا بهتر است یک لایه اضافی را به نام مثلا Service ایجاد کرد و تمام اعمال کار با EF را به آن منتقل نمود. سپس در قسمت‌های مختلف برنامه می‌توان از متدهای این لایه استفاده کرد. به عبارتی در فایل‌های Code behind برنامه شما نباید کدهای EF مشاهده شوند. یا در کنترلرهای MVC نیز به همین ترتیب. این‌ها مصرف کننده نهایی لایه سرویس ایجاد شده خواهند بود.
همچنین بجای نوشتن آزمون‌های واحد، به Integration tests سوئیچ کنید تا بتوان برنامه را در شرایط کار با یک بانک اطلاعاتی واقعی تست کرد.

برای مطالعه بیشتر:
Abstracting your ORM is a futile exercise
Repository is the new Singleton
Night of the living Repositories
Architecting in the pit of doom: The evils of the repository abstraction layer
?Is Repository old skool
?EF Code First: Where’s My Repository
Generic Repository With EF 4.1 what is the point

نظرات خوانندگان

نویسنده: NTC
تاریخ: ۱۳۹۱/۰۲/۲۴ ۲۳:۱۹:۵۵

سلام و ممنون
میشود در مورد خط زیر بیشتر توضیح دهید؟
"در اینجا بهتر است یک لایه اضافی را به نام مثلا Service ایجاد کرد و تمام اعمال کار با EF را به آن منتقل نمود. سپس در قسمت‌های مختلف برنامه می‌توان از متدهای این لایه استفاده کرد. "
یعنی چی تمام اعمال کار با EF را به آن منتقل کنیم؟
چطور؟

نویسنده: ناشناس
تاریخ: ۱۳۹۱/۰۲/۲۴ ۲۳:۵۱:۵۶

http://www.dotnettips.info/2009/10/nhibernate_17.htmlقبلا که این روش رو توصیه می کردید "کلا استفاده‌ی از هر کدام از ORMs موجود بدون پیاده سازی الگوی Repository اشتباه است. به چند دلیل:" و همین دلایل بالا رو نوشته بودید؟

نویسنده: Javad Darvish Amiry
تاریخ: ۱۳۹۱/۰۲/۲۵ ۰۰:۰۰:۳۹

توی این سری پست ها، از این پست واقعا واقعا واقعا لذت بردم. دست گلتون درد نکنه. خدا خیرتون بده. زنده باشین.

نویسنده: Javad Darvish Amiry
تاریخ: ۱۳۹۱/۰۲/۲۵ ۰۰:۰۲:۲۶

جانا سخن از زبان ما می گویی...
توی این سری پست ها، از این پست واقعا واقعا واقعا لذت بردم. دست گلتون درد نکنه. خدا خیرتون بده. زنده باشین.

نویسنده: mohammad
تاریخ: ۱۳۹۱/۰۲/۲۵ ۰۰:۲۸:۰۳

سلام آقای نصیری. آیا این روش که در خود سایت asp.net انجام شده هم اشتباه هستش؟

http://www.asp.net/mvc/tutorials/getting-started-with-ef-using-mvc/implementing-the-repository-and-unit-of-work-patterns-in-an-asp-net-mvc-application

نویسنده: شاهین کیاست
تاریخ: ۱۳۹۱/۰۲/۲۵ ۰۱:۴۵:۴۵

سلام ،

ذکر کردید "بهتر است یک لایه ی اضافی به نام سرویس ایجاد شود و اعمال مربوط به EF را به آن منتقل نمود" یعنی به ازای هر موجودیت یک سرویس هم داشته باشیم ؟ این (http://pastebin.com/iUCn1303) نمونه کدی که اینجا قرار دادم صحیح است ؟ - Service همان Business logic ما خواهد بود ؟ یعنی علاوه بر اعمال CRUD ، بررسی منطق تجاری ، Validate کردن Entity و یا اعمال شرط های مختلف در Query ها در همین لایه ی سرویس انجام می شود ؟
-اگر بخواهیم اطلاعات User را با اطلاعات پروفایل در قالب یک لیست برای لایه ی پایین تر بفرستیم ، نوع خروجی چه باید باشد ؟ مثلا UserProfileDataTransferObject ؟

در سری هشتم Refactoring در مورد God Classes توضیح دادید. برای منطق های پیچیده روی یک Enity مثل User مثلا چندین Query با حالت های مختلف کلاس UserService به God Class تبدیل می شود. راه حل چیست ؟ اگر ممکن است پروژه ی کد باز مورد تایید خودتون برای مرور کد ها معرفی کنید.
خیلی ممنون

نویسنده: مهمان
تاریخ: ۰۸:۴۳:۰۷ ۱۳۹۱/۰۲/۲۵

سلام. با توجه به مواردی که تا این شماره در مورد EF Code First فرمودید، بنده یک معماری مناسب برنامه را با استفاده از EF اینگونه درک کردم:

1- Domain Layer یا Model Layer: با استفاده از Code First فقط Entityها تعریف می شود (بدون Mapping و Data Annotation)

2- DAL: با استفاده از DbContext و Fluent API موارد مرتبط با Mapping و Data Annotation و ساخت DbSet ها تعریف می شود.

3- Facade DAL: با استفاده از WCF یا هر تکنیک سرویس گرایی دیگر یک لایه سطح بالا بر روی DAL کشیده می شود تا در BLL تنها از آن استفاده شود و عملا لایه های BLL و UI از EF هیچ اطلاعی نخواهند داشت.

4- BLL: کلاس های مربوط به برنامه، Rule ها و Infrastructureها
5- UI: شامل پروژه WPF، ASP.NET MVC و یا برنامه های موبایلی

آیا این معماری درک درستی از مطالب ارائه شده توسط شما است؟
و سوال دوم اینکه در مورد برنامه هایی با منطق MVC و یا MVVM لایه مدل تقریبا با این معماری چیزی نخواهد بود و تنها ویو مدل ها به چشم خواهند خورد. آیا این مورد هم درست است؟

نویسنده: وحید نصیری
تاریخ: ۰۹:۲۴:۲۶ ۱۳۹۱/۰۲/۲۵

اون پیاده سازی هم اشتباه است! چون متد Save داخل خود Repository است.
ضمنا اون مطالب رو بر اساس اطلاعات آن روز نوشتم. الان هم پیاده سازی UOW و Repository کاملی رو از NH دارم ولی ... هیچ وقت NH رو در پروژه‌ای که از آن استفاده کردم، تغییر ندادم یا از آن repository برای Unit testing استفاده نکردم.
وابستگی به ORM در عمل زیاد خواهد بود. بنابراین تعویض آن غیرممکن می‌شود.
استفاده از بانک اطلاعاتی واقعی بهتر است و به همین جهت دو کاربردی که برای آن در اکثر سایت‌ها ذکر شده، در عمل استفاده نمی‌شود.
بنابراین استفاده از Repository صرفا پیچیدگی بی‌موردی را به سیستم تحمیل می‌کند.

نویسنده: وحید نصیری
تاریخ: ۰۹:۳۳:۴۶ ۱۳۹۱/۰۲/۲۵

اولین StudentRepository را که نوشته، بله. در اینجا Unit of work را نقض کرده.
ولی درادامه ... خیر (زمانیکه UnitOfWork را ارائه داده). ولی کار اضافی انجام داده.
بینید در کلاس UnitOfWork کار وهله سازی Context انجام شده. بنابراین درست است. در همین کلاس هم Save قرار دارد بنابراین درست عمل شده و می‌شود با این سیستم یک تراکنش را انجام داد. ضمنا در این کلاس Uow کار معرفی Repository ها رو انجام داده.
ولی ... خود Context اصلی هم این موارد را دارد (خودش Uow است. کلاس مشتق شده از آن دارای DbSet است). شما به چه نتیجه‌ای از این Abstraction بی‌مورد خواهید رسید؟ احتمالا دو مورد عنوان شده در بالا ... که توضیح دادم در عمل هیچ وقت رخ نخواهد داد.

نویسنده: وحید نصیری
تاریخ: ۰۹:۴۴:۴۶ ۱۳۹۱/۰۲/۲۵

-1 بله.
-2 بله.
-3 این هم خوبه ولی اگر بانک اطلاعاتی و برنامه وب شما مثلا در یک سرور قرار دارند ضرورتی به استفاده از WCF نیست و به کارآیی بیشتری حین استفاده مستقیم از بانک اطلاعاتی خواهید رسید. WCF برای معماری چند tier توصیه می‌شود (هر tier رو یک سرور در اینجا فرض کنید. یک سرور جدای وب، یک سرور جدای اس کیوال و الی آخر ....)
-4 BLL همان لایه سرویسی‌است که عنوان کردم. جایی که از EF استفاده می‌شود.
-5 بله.

این M در MVC مرتبط با ASP.NET MVC جای بحث زیاد دارد. بیشتر ViewModel است تا Model به معنای Domain Classes .

نویسنده: وحید نصیری
تاریخ: ۰۹:۵۳:۱۳ ۱۳۹۱/۰۲/۲۵

- بله. چیزی شبیه به همین، البته وجود Interface هم در اینجا غیرضروری است. چون در مورد mocking صحبت کردم که در بانک‌های اطلاعاتی روش مناسبی نیست و بیشتر «توهم» صحیح کارکردن سیستم را به همراه خواهد داشت. توهم پاس شدن آزمون‌های واحدی که در عمل ممکن است تعداد زیادی از آن‌ها روی بانک اطلاعاتی واقعی اجرا نشوند. فرق است بین کار با اشیاء درون حافظه و بانک اطلاعاتی واقعی که بسیاری از قیود را اعمال می‌کند.
- بله. اسامی مهم نیست. یکی عنوان می‌کند BLL یکی Infrastructure یکی Service یکی...
- خروجی لایه سرویس فقط باید از نوع اشیاء معمولی، لیست یا IEnumerable باشد. اگر از IQueryable به عنوان خروجی متد استفاده کردید، به یک Abstraction دارای «نشتی» خواهید رسید. چون به سادگی خارج از منطقی که مدنظر بوده کاملا قابل تغییر است.
- God Class کلاسی است که اطلاعات زیادی را در اختیار سایرین قرار می‌دهد، نه کلاسی که جهت برآورده سازی منطق تجاری برنامه، از اطلاعات زیادی استفاده می‌کند. کلاسی که 1000 تا متد را در اختیار سایر کلاس‌ها قرار می‌دهد God class نام دارد. نه متدی که برای عملکرد صحیح خود نیاز به اطلاعات زیادی است.

نویسنده: وحید نصیری
تاریخ: ۰۹:۵۷:۲۶ ۱۳۹۱/۰۲/۲۵

یک مثال ساده: بجای اینکه در Code behind برنامه شروع کنید به وهله سازی از DbContext و بعد کوئری بنویسید و حاصل را مثلا در اختیار یک Grid قرار دهید، یک کلاسی ... جایی در یک پروژه Class library مجزا درست کنید که حاوی متد مشخصی است که فقط یک IList را برمی‌گرداند. این متد، محل کار با EF است نه Code behind یک فرم. در آنجا فقط باید از لیست نهایی تهیه شده استفاده شود.

نویسنده: مهمان
تاریخ: ۱۰:۰۶:۵۷ ۱۳۹۱/۰۲/۲۵

-1 پس به نظر شما نیازی به ایجاد یک لایه facade که بین DAL و BLL قرار گیرد و توابع EF و LINQ را برای استفاده در لایه بیزینس Wrap کند نیست و می توان از EF و LINQ مستقیما در BLL استفاده کرد و نهایتا به یک معماری چهار لایه رسید؟

-2 سوال دیگر اینکه جدا سازی Domain ها از لایه DAL چه مزایایی دارد؟ آیا در مهاجرت به یک ORM دیگر مفید است یا ملاحضات دیگری در میان است؟

نویسنده: وحید نصیری
تاریخ: ۱۰:۱۴:۴۳ ۱۳۹۱/۰۲/۲۵

- به همین دلیل در مورد عدم استفاده از Repository توضیح دادم. کار Repository همین warping عملکرد یک ORM است که نه

تنها ضرورتی ندارد بلکه در یک پروژه واقعی به شدت جلوی آزادی عمل شما را خواهد گرفت و همچنین پیاده سازی شما هم قابل انتقال نخواهد بود. استفاده از ORMها وابستگی‌های زیادی را به همراه دارند که شاید تنها قسمتی از آن‌ها را بتوانید مخفی کنید. همچنین برای اینکار باید از قابلیت‌های پیشرفته آن‌ها که ممکن است در سایر ORMs موجود نباشد، صرف نظر کرد. آزمون‌های واحد مرتبط با بانک‌های اطلاعاتی نیاز به بانک اطلاعاتی واقعی دارند تا بتواند قیود را اعمال کند. کار با اشیاء درون حافظه در اینجا اصلا توصیه نمی‌شود.
- بهتره. ضرورتی نداره. در حد یک مدیریت پروژه بهتر است که با یک نگاه بتوان تشخیص داد ... حداقل یک پوشه Models در برنامه هست. تا این حد کفایت می‌کند.

نویسنده: A. Karimi
تاریخ: ۱۳۹۱/۰۲/۲۵ ۱۰:۲۱:۵۴

یک دلیل مهم برای ایجاد یک Abstraction بر روی EF CodeFirst وجود دارد و آن هم استقلال از EF و IQueryable است. مثلاً ممکن است شما بخواهید بعداً به جای EF از NHibernate استفاده کنید. و یا ممکن است (حتماً) تکنولوژی جدیدتری بعد از EF وارد کار شود. از همه مهمتر ممکن است بخواهید از الگوی Adapter یا Decorator استفاده کنید. مثلاً یک Repository که اطلاعات یک جدول را به صورت Encrypt شده در بانک اطلاعات ذخیره می‌کند. و مثلاً به صفحه بندی سمت سرور هم نیاز دارید و حتی امکان استفاده از IQueryable هم وجود ندارند.

نویسنده: وحید نصیری
تاریخ: ۱۳۹۱/۰۲/۲۵ ۱۰:۳۲:۰۵

این همان مورد «به این ترتیب به سادگی می‌توان ORM مورد استفاده را تغییر داد» است که در بالا توضیح دادم. نمی‌تونید به این سادگی وابستگی‌های ذاتی به ORM مورد استفاده را حذف کنید. تمام کار با ORM به Update و Delete ختم نمی‌شود. برای نوشتن یک لایه قابل انتقال نیاز خواهید داشت دست و پای ORM مورد استفاده را قطع کنید که مثال زدید ... استقلال از IQueryable . و سؤال اینجا است که تمام لطف EF همین IQueryable است؛ چرا باید از این مستقل شد. من از کار کردن مستقیم با آن لذت می‌برم! به همین دلیل EF را انتخاب کرده‌ام. فقط لایه سرویس مورد استفاده نباید IQueryable را بازگرداند.

نویسنده: A. Karimi
تاریخ: ۱۳۹۱/۰۲/۲۵ ۱۱:۱۵:۴۸

ما با ایجاد لایه های متفاوت از Abstraction این قضیه را حل می‌کنیم. مثلاً در چهارچوبی که تهیه کرده‌ام، یک اینتفریس IRepository دارم که از 7 دولت آزاد است. و یک IQueryableRepository دارم که مخصوص ORMهای با پشتیبانی از IQueryable است. شاید شما بفرمایید وقتی از IQueryableRepository در برنامه استفاده کنی به IQueryable وابسته می‌شوی. بله، درست است، اما مثلا برای عملیات CRUD کلاسهایی برای WPF یا ASP.NET MVC در چهارچوب دارم که کاملاً از ORM مستقل هستند. به این ترتیب وقتی بخواهیم از یک ORM به دیگری Switch کنیم حداقل نیازی نیست کلاس‌های Base (همان کلاسهایی که در چهارچوب و طی زمان بیشتری تهیه شده) کوچکترین تغییری بکنند. و با این مکانیزم در صورتی که پروژه بلند مدت باشد باز هم می‌توان این استقلال را در لایه های[1] دیگر حفظ کرد به این ترتیب:

1. لایه Data Access دارای یک Infrastructure است که فقط اینترفیس‌های UnitOfWork و Repository در آن تعریف شده و هیچ خبری از IQueryable نیست مثلا برای دریافت داده صفحه بندی شده چیزی شبیه startRowIndex و maximumRows و حتی sortExpression پاس داده می‌شود.2. برای هر ORM یک پروژه جدا که به Data.Infrastructre رفرنس دارد ایجاد می‌شود. حالا در این پروژه UnitOfWork و Repository پیاده‌سازی می‌شود که از نظر داخلی به IQueryable وابسته است و مثلاً در پیاده‌سازی همان متد صفحه بندی شده به راحتی از LINQ استفاده می‌کند.3. هیچ کس حق ندارد در لایه های دیگر از IQueryable استفاه کند و باید ابتدا متد مورد نظرش را به Data.Infrastructure اضافه و بعد در لایه مرحله دو پیاده‌سازی کند.4. با استفاده از DI مسئله جایگذین کردن لایه مرحله 2 به راحتی اتفاق می‌افتد و هیچ لایه‌ای مستقیم به پیاده‌سازی سمت Data وابسته نیست بلکه به Data.Infrastructure وابسته است.

من قبول دارم که این کار زمان بیشتری را می‌گیرد اما شما هم خوب می‌دانید که اگر می‌خواهیم وقت صرف کنیم و یک چهارچوب ماندگار ایجاد کنیم باید طراحی مستقل از تکنولوژی داشته باشیم (برای مثال حتی WPF و MVC که عرض کردم هم در چهارچوب جز

Concrete class به حساب می‌آیند؛ هر چند از دید پروژه نهایی و مصرف کننده چهارچوب اینطور نیست).

مثال کاربردی این قضیه این است که در یک پروژه مجبور بودیم اطلاعات را رمزنگاری شده در DB ذخیره و بازیابی کنیم و البته فرم مورد نظر از هما CRUD های داخل چهارچوب بود و اگر این طراحی استفاه نمی‌شد مجبور بودیم برای آن یک فرم از یک ابتدا پیاده‌سازی کنیم و چون CRUD موجود در چهارچوب دارای امکانات قابل توجهی بود در زمان و هزینه تاثیر زیادی داشت.

[1] منظور از لایه، لایه‌های فیزیکی و سنتی نیست.

نویسنده: وحید نصیری
تاریخ: ۱۳۹۱/۰۲/۲۵ ۱۱:۵۳:۱۱

- ORMهای با پشتیبانی از IQueryable ، پشتیبانی یکسانی از متدهای الحاقی تعریف شده ندارد. کد شما قابل انتقال نیست. برای مثال NH یک سری متد الحاقی خاص خودش را دارد. بنابراین اگر تصور می‌کنید که می‌توان این پیاده سازی را به ORM دیگری تغییر داد، در عمل ممکن نیست؛ مگر اینکه از توانایی‌های محدود و مشترکی استفاده کنید.
- یکی دیگر از اشتباهات طراحی الگوی Repository متدهای عمومی هستند که دارای خروجی IQueryable است. به این نوع طراحی، طراحی نشتی دار گفته می‌شود چون ابتدای کار مشخص است اما انتهای کار بازگذاشته شده است: (^)
- یک مثال دیگر: NH دارای متدی است به نام tofeautre که توانایی اجرای چندین و چند مثلا sum را بر روی بانک اطلاعاتی در یک رفت و برگشت دارد. لایه Repository شما نمی‌تواند این را مخفی کند و در صورت استفاده از آن قابل انتقال نخواهد بود. استفاده از آن هم ایرادی ندارد چون دلیل استفاده از یک ORM، استفاده از توانایی‌های پیشرفته آن‌ها است. از این نمونه مثال زیاد است.
حالا اینجا اگر شخصی از الگوی Repository استفاده کند، هم بی‌جهت خودش را محدود کرده و هم نهایتا به یک leaky abstraction رسیده.
- «چهارچوبی دارم که کاملاً از ORM مستقل هستند» این همان لایه سرویس است که از آن صحبت شد. نباید کدهای یک ORM (هر ORM ایی) داخل Code behind یا کنترلرهای MVC مشاهده شوند. این روش توصیه شده است.
- رمزنگاری را می‌شود با یک سری متدالحاقی در یک پروژه دیگری به نام Common پیاده سازی کرد. در لایه Service از آن استفاده کرد. بحث ما در اینجا در مورد Repository و عدم ضرورت استفاده از آن است. یا مباحث صفحه بندی هم به همین ترتیب. این‌ها یک سری متدالحاقی عمومی هستند؛ خارج از تعریف الگوی Repository قرار می‌گیرند.

نویسنده: Javad Darvish Amiry
تاریخ: ۱۳۹۱/۰۲/۲۵ ۱۲:۰۱:۳۴

و فکر می کنم یک دلیل مهم برای ایجاد انتزاعی (Abstraction) به نام Service پیش بینی همچین مهاجرتی هست. من Service رو لایه BLL نمیدونم (شخصا و طبق تحلیل و تجربه خودم؛ درست و غلطش رو نمیدونم) بلکه یه لایه واسط بین BLL و DAL میدونم. به این ترتیب همیشه یه DAL کاملا مستقل و منتزع از بقیه برنامه دارم. هر وقت خواستم از EF به NH برم -یا تجربه واقعی تر این که اگه خواستم به یه تکنولوژی جدیدتر کوچ کنم- کافیه لایه DAL جدیدم رو بنویسم و فقط نحوه دسترسی در سرویس تغییر میکنه. پس باز هم انتزاع مورد نیاز رو دارم.
اما در مورد UoW خوب باز هم طبق تجربه حتی با وجود DbContext (و Session در NH) فکر میکنم وجودش خیلی مفیده. مثلا تو برنامه های وب UoWم رو برپایه HttpModule قرار میدم و موقع Dispose شدن ماژول، تغییرات رو ذخیره میکنم.
در مورد M در MVC کاملا با آقای نصیری موافقم. این بحثو مدتی پیش با یه دوستی هم داشتیم که زیر بار نرفت. مثالی که اونجا زدم اینجا میزارم، فکر کنم مفیده.
فکر کنید شیئی به نام Member دارید با مثلا 20 پراپرتی. و این 20 خاصیت، در 5 ویوی مختلف نمایش داده میشن. یعنی یه ویو 6 تا، یه ویو 3 تا و همینطور تا آخر. خوب عاقلانه نیست که وقتی که به هر 20 خاصیت نیاز نداریم، همشو واکشی کنیم. پس دو تا راه داریم. یا باید اون 3 4 تا خاصیتی که نیاز داریم رو جداگانه (مثلا تو یه Tuple یا Anon یا حتی متغیرهای منحصر) واکشی کنیم و در اختیار ویو بذاریم؛ یا بیایم و یه ساختمون (class یا struct) تعریف کنیم مخصوص اون 3 4 تا پراپرتی. خوب من روش دوم رو ترجیح میدم که همون view-model های منو میسازه. یا نمایی رو در نظر بگیرید که لازمه از ترکیبی از دو یا چند مدل داده استفاده کنه. تو همون مثال Member، فرض کنید نمایی هست که نام کاربری و نام و نام خانوادگی رو از Member و تعداد ارسال ها رو از Comments و آخرین فعالیت رو از MemberActivitis و آدرس ایمیل عمومی رو از AuthTokens میخونه! چه میکنید؟

ضمنا به فلسفه وجودی مثلا AutoMapper هم فکر کنیم.

نویسنده: Javad Darvish Amiry
تاریخ: ۱۳۹۱/۰۲/۲۵ ۱۲:۰۸:۵۲

{ ORMهای با پشتیبانی از IQueryable ، پشتیبانی یکسانی از متدهای الحاقی تعریف شده ندارد.}
مثال واقعی و زنده ای که همین الان میشه زد تفاوت پیاده سازی L2E در NH و EF هست. عملا مهاجرت غیر ممکنه مگه از روشی که آقای نصیری گفتن. (دیدم که میگم ;))

نویسنده: وحید نصیری
تاریخ: ۱۳۹۱/۰۲/۲۵ ۱۲:۱۲:۲۱

همه این‌ها نیکو! ولی زمانیکه از یک ORM استفاده می‌کنید، DAL همین ORM است. تعویض آن هم به سادگی میسر نیست. من با شناختی که مثلا از NH دارم عرض می‌کنم که کسی که از NH استفاده می‌کنه نمی‌تونه توانایی‌های توکار آن‌را با هیچ ORM دیگری عوض کنه. سطح دوم کش، غیرفعال کردن سطح اول کش برای مباحث گزارشگیری. متدهای ویژه‌ای که در QueryOver آن پیش‌بینی شده. استفاده از HQL آن برای اعمالی که نه با LINQ میسر است و نه با QueryOver.
خلاصه از سطحی خیلی بالا اینطور به نظر میرسه که می‌شود یک لایه انتزاعی بر روی ORM درست کرد ولی در عمل اینطور نیست. این لایه قابل انتقال نیست مثلا به EF یا نمونه‌های مشابه.

نویسنده: Javad Darvish Amiry
تاریخ: ۱۳۹۱/۰۲/۲۵ ۱۲:۲۳:۵۶

خوب منم همینو عرض میکنم. میگم انتقال ممکن نیست مگر به روش شما (محدود کردن ORM و صرفنظر از کلی از قابلیت هاشون). اگه سرویس رو بین DAL و بقیه اجزا برنامه داشته باشیم چی؟ من از EF استفاده میکنم. سرویس من نه انتیتی ها بلکه viewmodel های مشخصی که هر بخش برنامه نیاز داره میگیره یا بر میگردونه. مثلا متدی که تو همون بخش MVC مثالشو گفتم در نظر بگیرید. اسمشو میذاریم GetMemberInfo. سرویس موظفه viewmodel مرتبط رو پر کنه و برگردونه. امروز از EF استفاده میکنه؛ پس عاقلانه ترین راه استفاده از پروژکشن هست. فردا روزی به هر دلیلی مجبورم به NH کوچ کنم. خوب باقی اجزا برنامه از این تغییر بی خبر میمونن. حالا NH هم پروژکشن داره هم tofeautre که بعنوان یه نمونه، فکر میکنم استفاده از امکان tofeautre اینجا بهتره. خوب کی از این تغییر باخبره؟ فقط سرویس. سرویسه که داره پیاده سازی های مختلفی میشه. بقیه اجزا کاملا از این تغییر مصون میمونن.

نویسنده: وحید نصیری
تاریخ: ۱۳۹۱/۰۲/۲۵ ۱۲:۳۰:۰۹

بله. این روش طراحی خیلی خوبه؛ برنامه استفاده کننده از نحوه پیاده سازی GetMemberInfo بی‌خبر است. حالا می‌خواهد NH باشد یا EF یا هر مورد دیگری.
ضمن اینکه این لایه میانی رو که عنوان کردید هم در برنامه‌های وب می‌تونه استفاده شود و هم ویندوزی.

نویسنده: Javad Darvish Amiry
تاریخ: ۱۳۹۱/۰۲/۲۵ ۱۲:۳۴:۱۴

اگه باز بخوام توضیح بدم زیاده گویی میشه. ولی راه حل خودم اینه:
برای هر بخش منطقی برنامه که نیاز به داده پایگاه داره یه سرویس در نظر میگیرم. یه interface برای اون سرویس میگیرم. مثلا IMemberService. حالا لایه دسترسی به داده تو یه پروژه جدا قرار میگیره. اگه قراره از EF استفاده کنم، EfMemberService رو میسازم. به همه قابلیت های EF هم دسترسی دارم. متود GetMemberInfo هم به هر روشی که خودش میدونه موظفه اطلاعات مورد نیاز رو واکشی و برگردونه. چون روشش در اختیار خودشه پس میتونه از همه قابلیت های EF استفاده کنه. حالا مثلا اگه بیام و StructureMap استفاده کنم (که میکنم) میتونم فایل رجیستری برای IMemberService رو هم تو همون پروژه بذارم. با استارت برنامه، DependencyResolver میفهمه هر جا به IMemberService نیاز داشت، باید از EfMemberService کنه.
فردا یه تکنولوژی جدیدتر میاد و EF رو به نابودی (یا حداقل حاشیه) میره. مثلا اسم اون ORM رو میذاریم NH. یه پروژه جدید تعریف میکنم برای NH. سرویس ها رو توش پیاده سازی میکنم. مثلا حالا NhMemberService دارم. فایل رجیستری تو همون پروژه قرار

داره. DLL نهایی رو با DLL قبلی عوض میکنم و برنامه رو دوباره استارت میکنم. حالا هر جا به IMemberService نیاز داشتم، NhMemberService استفاده میشه.
یعنی دقیقا همونکاری که شما فرمودید. EfMemberService و NhMemberService کاملا مستقل هستن و هر کدوم میتونن از تمام قابلیت های ORM مورد استفاده شون استفاده کنن.
کل منظورم همین بود.

نویسنده: Javad Darvish Amiry
تاریخ: ۱۳۹۱/۰۲/۲۵ ۱۳:۰۰:۳۵

{هم در برنامه‌های وب می‌تونه استفاده شود و هم ویندوزی}
خوب این هم باز یه نکته است. اون UoW که گفتم بالا، دقیقا مربوط میشه به این بخش. مثال:
تو کامنت قبلی از استقلال لایه سرویس عرض کردم. حالا اضافه میکنم، سرویس من داره از EF استفاده میکنه. اما مستقیما نمیاد DbContext رو صدا بزنه. بلکه اون رو از یه UoW میگیره. خاصیتش اینه که UoW من از طریق یه HttpModule وهله سازی و نابود میشه. تو نابود شدنش dirty بودنش رو بررسی و در صورت نیاز commit میشه.
حالا چطور برم رو ویندوز؟ HttpContext اگه null باشه پس رو وب نیستم! راهکارم شبیه سازی HttpContext مثلا با یه کلاس به اسم HatContext هست. حالا میتونم لایه سرویس رو هم در وب و هم در ویندوز استفاده کنم.
UoWم هم سه مرحله توسعه داره. بالاترین سطح، فقط IsDirty رو در اختیار میذاره. سطح دوم فقط Commit و Rollback و سطح سوم TContext. سطح دوم و سوم فقط در اختیار سرویس قرار داره (internal). سطح اول در اختیار کل برنامه است (public).
پس IsDirty، مثلا تو WPF خیلی میتونه مفید واقع بشه. حالا Forward کردن انواع (مثلا با StructureMap) کمکم میکنه که دسترسی به هر کدوم از interface های سه گانه بالا، فقط یه نمونه رو برگردونه.

{این یه مثال انتزاعی نیست؛ دقیقا یه پروژه ی واقعی بود که درگیرش بودم.}

نویسنده: وحید نصیری
تاریخ: ۱۳۹۱/۰۲/۲۵ ۱۳:۰۸:۰۵

بله. این مورد به الگوی «Session Per Request» معروف است که کار آن به اشتراک گذاری یک DbContext در طول مدت عمر یک Http Request است. در این مورد یک مطلب خواهم نوشت.

نویسنده: A. Karimi
تاریخ: ۱۳۹۱/۰۲/۲۵ ۱۳:۲۲:۲۰

- همانطور که عرض کردم ما به هیچ عنوان وابستگی Contract ی به IQueryable نداریم. دقت بفرمایید که برای چیزی مثل NH ما اصلاً از IQueryable استفاده نخواهیم کرد. صحبت بنده Abstract تر بود. "لایه Data Access دارای یک Infrastructure است که فقط اینترفیس‌های UnitOfWork و Repository در آن تعریف شده و هیچ خبری از IQueryable نیست".

- طراحی Repository اصلی خروجی و ورودی IQueryable ندارد با این صحبت کاملاً موافق هستم و غیر از این هم نیست.

- همانطور که عرض کردم این موضوع کاملاً قابل اجراست. شما به ازاء هر ORM یک DataAccess خواهید داشت و امکانات قوی هر ORM را در داخل DataAccess خودش به راحتی استفاده می‌کنید به لایه بندی و ارجاعاتی که عرض کردم دقت بفرمایید. مصرف کننده Data به جای استفاده از DataAccess های Concrete از سطوح بالاتری استفاده می‌کنند و DI این موضوع را حل کرده است. شما می‌توانید از هر امکانی که دوست دارید و مخصوص ORM خودتان است در آن استفاده کنید.

- لایه سرویس با لایه Data متفاوت است. با چیزی که شما فرمودید اگر ORM تغییر کند باید کدهای لایه سرویس را دستی تغییر دهید و با Breaking Change روبرو می‌شویم. اما همانطور که قبلاً هم گفتم، با وجود این Abstraction که گفتم، فقط با یک Config ساده در DI این موضوع به طور کامل حل می‌شود. کلاً فلسفه وجودی Repository همین ورود و خروج داده (ذخیره و بازیابی) و Abstract بودن آنهاست. کدهای ORM نه تنها نباید در Code Behinde ها دیده شود بلکه نباید در Service یا Business سیستم هم باشند. به همان دلیلی که عرض کردم. و البته مثال رمزنگاری مثال خوبی است (Service چرا باید درگیر رمزنکاری شود؟ این

وظیفه لایه ذخیره و بازیابی است که همان Repository خواهد بود).

- Repository http://martinfowler.com/eaaCatalog/repository.html همانطور که از نامش پیداست یک مخزن است که ما نمی‌دانید اطلاعات را در کجا نگهداری خواهد کرد (حافظه، بانک و ...) حال در یک پروژه این محل نگهداری باید امن باشد همین! به جای حافظه و بانک و ... تصور کنید یک جای امن را. آیا برای این نمی‌توان (نباید) یک Repository ساخت؟ مورد بعدی اینکه رمزنگاری به لایه دیگری منتقل شده و Repository که گفتم از همان استفاده می‌کند. نکته بعدی اینکه با چیزی که شما فرمودید، باید مراقبت از اینکه همه چیز رمزنگاری شود را به برنامه نویس محول کرد و حتماً حواسش باشد که فراموش نکند قبل از ذخیره یا بازیابی یک Entity متدهای لازم برای رمزنگاری را فراخوانی کند. در صورتی که لایه سرویس باید متمرکز بر روی Business کار باشد نه فراخوانی متدهای رمزنگاری برای ذخیره سازی و ...

نویسنده: A. Karimi
تاریخ: ۱۳۹۱/۰۲/۲۵ ۱۳:۳۰:۳۵

"لایه Data Access دارای یک Infrastructure است که فقط اینترفیس‌های UnitOfWork و Repository در آن تعریف شده و هیچ خبری از IQueryable نیست"

صحبت بنده این بوده؛ ما به این فکر می‌کنیم که از IQueryable هم Abstract باشیم. چرا فکر می‌کنید قرار است ORM ها پشتیبانی یکسانی از متدهای الحاقی داشته باشند؟ مشخص است که یکسان نیست! در صورتی که بخواهیم از NH استفاده کنیم انتخاب اول QueryOver است نه LINQ to NH. غیر از مورد نشتی، یکی دیگر از دلایل اینکه گفته می‌شود از IQueryable در Contract ها استفاده نشود همین عدم سازگاری و پشتیبانی است.

به طوری کلی، به عبارت ساده لایه DataAccess ما پیمانه‌ای تر است (دارای Modularity بالاتری است) و بحث اصلا بر سر IQueryable نیست شما باید بتوانید یک Repository بسازید که حتی به عنوان محل ذخیره سازی از یک فایل Text استفاده کند.

نویسنده: وحید نصیری
تاریخ: ۱۳۹۱/۰۲/۲۵ ۱۳:۳۳:۲۵

چندتا بحث هست. مدیریت پروژه. استفاده از Repository.
اینکه پوشه درست کنید، class library درست کنید. اسم‌های متفاوت استفاده کنید. همه این‌ها خوب است.
باز هم در طراحی خودتون اومدید ORM رو مخفی کردید. کل بحث جاری این است که اینکار اتلاف وقت است. «فقط با یک Config ساده در DI این موضوع به طور کامل حل می‌شود» : .... نمی‌شود. خیر! قصد ندارم مواردی رو که عنوان کردم تکرار کنم. مدتی با یک ORM با قابلیت‌های بالا کار کنید، این مطلب رو دیگر عنوان نخواهید کرد.
به نظر در مورد اسامی کمی تداخل اینجا هست. مفهومی رو که از Repository دنبال می‌کنید، همان چیزی است که در لایه سرویس من به آن اشاره کردم.
اگر علاقمند بودید، به پیاده سازی generic repository که لینک دادم مراجعه کنید.
واژه‌های مورد استفاده رو اگر یکسان کنیم شاید خیلی از سؤ تفاهم‌ها برطرف شود.

نویسنده: A. Karimi
تاریخ: ۱۳۹۱/۰۲/۲۵ ۱۴:۰۴:۰۰

موافقم، بحث مدیریت پروژه و دیگر مسائل خیلی مطرح و تاثیر گذارند.

در خصوص Config ساده در DI و حلی شدن موضوع، با نظر شما موافق نیستم و این کار انجام شدنیست (بحث مخفی کردن یا پیمانه‌ای کردن). این بحث‌ها بسیار جالب و جذاب است و ای کاش در محیط مناسب تر و راحتتری به بحث می‌پرداختیم.

نویسنده: Javad Darvish Amiry
تاریخ: ۱۳۹۱/۰۲/۲۵ ۱۴:۰۹:۰۶

{شما باید بتوانید یک Repository بسازید که حتی به عنوان محل ذخیره سازی از یک فایل Text استفاده کند} کاملا موافقم. و این همون چیزیه که ازش بعنوان لایه سرویس یاد کردم. توضیحات کامل رو تو کامنت های پایین تر دادم.
فکر کنم اختلاف رو نامگذاری ها و برداشت متفاوتی که از تعاریف داریم، باشه.

نویسنده: وحید نصیری
تاریخ: ۱۳۹۱/۰۲/۲۵ ۱۴:۲۵:۳۵

من فکر می‌کنم طرحی که از Repository در ذهن شما است، تعریف یک اینترفیس که دارای متدهای مثلا Add و Get و امثال آن است. سپس پیاده سازی کامل آن با EF. چون مبتنی بر Interface است می‌شود یک پیاده سازی مبتنی با NH را هم برای آن تدارک دید. بعد این‌ها رو میشه با DI مدیریت کرد. بله. این شدنی است. فقط واژه‌های استفاده شده در اینجا بین من و شما یکی نیست. من Repository رو به عنوان یک لایه سبک محصور کننده خود EF مد نظر دارم. یعنی پیاده سازی که در هزاران سایت اینترنتی داره تبلیغ میشه و به نظر من لزومی ندارد. انتقال پذیر نیست و لذت استفاده از یک ORM واقعی رو از بین می‌بره.
در کل من از واژه سرویس استفاده کردم شما از واژه مخزن. ولی به نظر برداشت هر دو یکی است.

نویسنده: مجید شمخانی
تاریخ: ۱۳۹۱/۰۲/۲۵ ۱۷:۵۳:۰۸

به نظر من در این بحث «تعویض ORM» اصل YAGNI را نباید فراموش کرد، ولی با این حال آیا شما استفاده از الگوی Repository و UOW را برای NH پیشنهاد می‌کنید یا خیر؟

نویسنده: وحید نصیری
تاریخ: ۱۳۹۱/۰۲/۲۵ ۱۹:۳۰:۱۱

شما فقط به uow برای پیاده سازی session per request نیاز خواهید داشت (به اشتراک گذاری فقط یک سشن در طول عمر یک http request در لایه‌های مختلف برنامه).
الگوی Repository خصوصا در مورد NH قابل انتقال نیست. چون تا حد بسیار زیادی به جزئیات QueryOver، LINQ و حتی HQL وابسته می‌شود که در سایر ORMهای دیگر معادل ندارد. به همین جهت تحمیل این لایه به سیستم غیرضروری است.
یعنی در ابتدا مقالات را که مطالعه کنید، همه عنوان می‌کنند که با استفاده از Repository به سیستمی خواهید رسید که می‌توانید ORM آن‌را به سادگی تعویض کنید. اما در مورد NH ابدا اینطور نیست. هنوز حتی خیلی از حالات mapping رو که این سیستم پشتیبانی می‌کنه در سایر ORMها نمی‌تونید پیدا کنید. بنابراین این Repository قابل تعویض نیست. بهتره بگم این ORM اصلا قابل تعویض نیست؛ چون اصطلاحا feature rich است. مگر اینکه از توانایی‌های پیشرفته آن استفاده نشود که آن وقت ضرورت استفاده از آن‌را زیر سؤال می‌برد.

نویسنده: مجید شمخانی
تاریخ: ۱۳۹۱/۰۲/۲۵ ۲۲:۲۶:۴۰

اگر امکان دارد نمونه‌ای از این نوع پیاده سازی را برای NH معرفی کنید.

نویسنده: وحید نصیری
تاریخ: ۱۳۹۱/۰۲/۲۵ ۲۲:۳۸:۳۹

این مورد برای مثال: (^)

نویسنده: Mohsen Esmailpour
تاریخ: ۱۳۹۱/۰۲/۲۶ ۰۱:۵۳:۰۶

از وقتی ک شروع به یاد گرفتن ASP.NET MVC کردم از کتاب Pro ASP.NET MVC 3 Framework گرفت تا آموزشهای خود سایت asp.net و در بسیاری از وبلاگها همه استفاد از repository رو به عنوان best pratice توصیه کردن. استفاده از Interface + EF + DI و mock در unit test به نظر خوب میاد. حالا این سئوال برام پیش اومده آیا از اشکال از تیم EF هست که در پیاده سازی DbContext از الگوی Unit of Work استفاده کرده و یک لایه Abstraction روی اون کشیده و یا ایراد از بی اطلاعی کسانی هست

الگوی Repository و EF Code First رو با هم به کار میبرن. این موضوع برای من مثل این میمونه ک یک اینترفیس رو با پیادسازی اجزاش به آدم بدن و بعد خودت دوباره بیای اجزاش رو پیاده سازی کنی و خبر نداشته که قبلا پیاده سازی شدن. من که گیج شدم.

نویسنده: وحید نصیری
تاریخ: ۱۳۹۱/۰۲/۲۶ ۰۸:۲۹:۰۶

- استفاده از uow صحیح است. نیاز است یک uow بین لایه‌های مختلف به اشتراک گذاشته شود.
- استفاده از Repository که به صورت یک لایه سبک روی EF عمل کند اتلاف وقت است. به دلایلی که عرض کردم.
- استفاده از لایه سرویس توصیه شده است.
این موارد رو به صورت یک مثال عملی در قسمت بعدی توضیح خواهم داد.

نویسنده: بازرگان
تاریخ: ۱۳۹۱/۱۱/۲۰ ۲۰:۳۳

با سلام؛

خواهشمند یه پروژه نمونه برای asp.net mvc4 که در آن از 5 entity framework code first و unit of work استفاده شده و بحث فوق در آن رعایت شده باشد معرفی نمایید.

آیا پروژه efmvc آنچه شما در اینجا گفته اید رعایت کرده و اگر نه موارد را بیان نمایید.

با سپاس فراوان

نویسنده: سعید
تاریخ: ۱۳۹۱/۱۱/۲۱ ۱۹:۵۲

پروژه قسمت 12 رو که من دیدم با ef5 و mvc4 سازگار است و اصولش فرقی نکرده.

نویسنده: ناظم
تاریخ: ۱۳۹۲/۱۰/۱۷ ۲۲:۲۵

با سلام
جناب نصیری من پس از خواندن مطالب واقعا مفید شما و چند تا مطلب دیگه تو سایتهای CodeProject , Stackoverflow ,Asp.NET , ... تقریبا گیج شدم بنابر این چند تا سوال دارم
-1 وقتی از یک orm مثلا EF استفاده میکنم داشتن یک class library به نام DAL و انتقال edmx یا کلاس‌های code first به اون اشکالی که نداره؟
-2 لایه سرویس همان BLL هست؟ میتوان اونجا مستقیم به DbContext و توابع اون مثل Delete ، add و غیره دسترسی داشت؟
-3 یه جا مثل اینکه فرموده بودید UoW خوبه استفاده کنیم و مخزن نه درسته ؟ اگه آره چرا و چطور ؟
-4 من یه Classlibrary تشکیا میدم به اسم Entities و POCOهای EF رو منتقل میکنم اونجا در آینده مشکل ساز که نیست؟
-5 جای صحیح استفاده از الگوهای مخزن و UoW کجاست؟
-6 میشه یه مثال که همه اینهارو که فرمودید رو رعایت کرده و مورد تایید شماست رو معرفی بفرمایید؟

نویسنده: وحید نصیری
تاریخ: ۱۳۹۲/۱۰/۱۷ ۲۲:۳۴

لطفا قسمت بعدی یعنی 12 را به همراه تمام پرسش و پاسخ‌های آن، مطالعه کنید. پیاده سازی UOW، مثال لایه بندی، تزریق وابستگی‌ها، بحث و غیره، تمامی در آن موجود است. در پرسش و پاسخ‌های آن، دوره‌های پیشنیاز مرتبط و پروژه‌های تکمیلی مرتبط هم معرفی شده‌اند.