عنوان: **اهمیت Controller های ساده در ASP.NET MVC**
نویسنده: شاهین کیاست
تاریخ: ۱۳۹۱/۰۵/۳۱ ۱۴:۳۳
آدرس: www.dotnettips.info
گروه‌ها: Refactoring, ASP.Net MVC

Controllerها به نوعی رابط بین View و Model هستند. *ساده ترین* محل برای قرار دادن کدهای تصمیم گیری (decision-making code) ، قرار دادن منطق تجاری و یا فراهم ساختن داده برای View مثل ایجاد یک لیست از Select List برای یک DropDownList می‌باشند. اما انجام این کارها به نرم افزار ما پیچیدگی تحمیل می‌کند. Controllerها باید در طول زمان توسعه‌ی یک نرم افزار کم حجم و سبک باقی بمانند. در این مطلب بحث شد که یکی از اهداف استفاده از ASP.NET MVC نوشتن نرم افزار هایی تست پذیر می‌باشد. نوشتن آزمون واحد و نگهداری Controller هایی که مسئولیت زیادی بر عهده دارند سخت می‌باشد. Controllerها باید به نحوی توسعه پیدا کنند که نگهداری آن‌ها ساده باشد ، تست پذیر باشند و اصل SRP را رعایت کرده باشند.

نگهداری آسان

کدی که درک آن سخت باشد ، تغییر دادن آن نیز سخت است ، هنگامی که درک کدی سخت باشد زمینه برای به وجود آمدن خطاها ، دوباره کاری و سردرد فراهم می‌شود. هنگامی که مسئولیت یک Action به صورت شفاف مشخص نباشد و انواع و اقسام کارها به آن سپرده شده باشد تغییر در آن سخت می‌شود ، ممکن است این تغییر باید چند جای دیگر هم داده شود در نتیجه فاز نگهداری هزینه و زمان اضافی به نرم افزار تحمیل می‌کند.

تست پذیری

بهترین راه برای اطمینان از این که درک و تست پذیری سورس کد ما ساده هست انجام و تمرین توسعه‌ی تست محور (TDD) می‌باشد. هنگامی که از روش TDD استفاده می‌کنیم با سورسی کدی کار می‌کنیم که هنوز وجود ندارد. در این مرحله از به وجود آمدن کلاس هایی که تست آن‌ها دشوار یا غیر ممکن است (به دلیل داشتن مسئولیت‌های اضافی) از جمله Controllerها جلوگیری می‌شود. نوشتن آزمون‌های واحد برای Controllerهای کم حجم ساده می‌باشد.

تمرکز بر روی یک مسئولیت

بهترین راه برای ساده سازی Controllerها گرفتن مسئولیت‌های اضافی از آن می‌باشد به Action زیر توجه کنید :

```
public RedirectToRouteResult Ship(int orderId)
{
   User user = _userSession.GetCurrentUser();
   Order order = _repository.GetById(orderId);
   if (order.IsAuthorized)
   {
      ShippingStatus status = _shippingService.Ship(order);
      if (!string.IsNullOrEmpty(user.EmailAddress))
      {
         Message message = _messageBuilder
            .BuildShippedMessage(order, user);
         _emailSender.Send(message);
      }
      if (status.Successful)
      {
         return RedirectToAction("Shipped", "Order", new {orderId});
      }
   }
   return RedirectToAction("NotShipped", "Order", new {orderId});
}
```

این Action کار زیادی انجام می‌دهد ، تقریبا می‌توان تعداد مسئولیت‌های این Action را با شمارش تعداد Ifها پیدا کرد . مسئولیت تصمیم گیری درباره این که آیا Order مورد نظر آماده برای تحویل است یا خیر به این Action سپرده شده ، همچنین تصمیم گیری درباره اینکه آیا کاربر دارای آدرس ایمیل جهت ارسال ایمیل می‌باشد یا خیر به این Action سپرده شده است. منطق‌های domain logic و business logic نباید در کلاس‌های presentation همانند Controllerها قرار داده شود. تست و نگهداری کدی مثل کد بالا دشوار خواهد بود. Refactoring که باید در این Code اعمال شود Refactor Architecture by Tiers نام دارد. این Refactoring به توسعه دهنده می‌گوید که منطقی که در لایه‌ی نمایش (Presentation) پیاده کرده را به لایه‌ی Business انتقال دهد. پس از انتقال منطق کد بالا به OrderShippingService ، کد Action ما ساده‌تر می‌شود :

```
public RedirectToRouteResult Ship(int orderId)
{
   var status = _orderShippingService.Ship(orderId);
   if (status.Successful)
```

```
   {
      return RedirectToAction("Shipped", "Order", new {orderId});
   }
   return RedirectToAction("NotShipped", "Order", new {orderId});
}
```

پس از انتقال منطق تجاری به محل مناسب خودش تنها مسئولیتی که برای برای Controller باقی مانده این است که کاربر را به کجا Redirect کند. پس از این Refactoring علاوه برا اینکه مسئولیت‌ها در جای مناسب خود قرار گرفتند ، اکنون می‌توان به سادگی منطق کار را بدون تحت تاثیر قرار گرفتن کدهای لایه‌ی نمایش تغییر داد.
در آینده به تکنیک‌های ساده سازی Controllerها خواهیم پرداخت.

نظرات خوانندگان

نویسنده: محمد صاحب
تاریخ: ۱۳۹۱/۰۵/۳۱ ۱۸:۹

ممنون موضوع جالبیه...
فصل 7وم کتاب Programming Microsoft ASP.NET MVC هم به همین موضوع اختصاص داده شده که برای سبک کردن کنترلر دوتا الگوی Responsibility-Driven Design و iPODD رو معرفی میکنه و...

نویسنده: شاهین کیاست
تاریخ: ۱۳۹۱/۰۵/۳۱ ۱۸:۱۳

ممنون می‌شم اگر درباره الگوی iPODD لینک بدید.

به این موضوع در سری کتاب‌های MVC In Action هم یک فصل کامل اختصاص داده شده .

نویسنده: حسین مرادی نیا
تاریخ: ۱۳۹۱/۰۵/۳۱ ۲۰:۳۵

ممنون
واقعا لذت بردیم
امیدوارم این مباحث رو همچنان ادامه بدید ...

نویسنده: رضا بزرگی
تاریخ: ۱۳۹۱/۰۵/۳۱ ۲۰:۳۶

آیا در معماری چندلایه (N-Tier arch) مرزبندی شفافی وجود دارد که از نظر شی‌گرایی روش یا روش‌های مرجح وجود داشته باشند؟
مثلا علت اینکه شما کنترل‌ها در MVC رو لایه نمایش به‌حساب میارین متوجه نشدم و مگر در واقع BL ما در کنترلرها اتفاق نمیوفته؟ یا مثلا ViewModel ما اگر با View در دولایه متفاوت هستند_(که هستند) جزو همان لایه نمایش به‌حساب میان؟
اگر برایتان مقدور است در مورد مرزهای شفاف تفکیک منطقی و فیزیکی (Layer & Tier) در MVC توضیح بدین،ممنون میشم.

نویسنده: حسین مرادی نیا
تاریخ: ۱۳۹۱/۰۵/۳۱ ۲۰:۴۹

طبق چیزی که من میدونم Controller محتوای لازم جهت نمایش در View رو آماده میکنه اما با لایه View متفاوت است. هدف ما از این پیاده سازی همین است (آماده سازی محتوای لازم جهت نمایش در View). با این حال Controller این محتوا را از منبع یا منابع مختلف دریافت میکند. به عبارتی اطلاعات خود را از لایه Service یا همان BL دریافت میکند و کدهای لایه Business درون Controller قرار نمیگیرند.

نویسنده: میثم
تاریخ: ۱۳۹۱/۰۵/۳۱ ۲۰:۵۵

مرسی ، جالب بود

نویسنده: شاهین کیاست
تاریخ: ۱۳۹۱/۰۶/۰۱ ۰:۳۱

Controller فقط مصرف کننده‌ی منطق نهایی است ، بدنه‌ی کنترلرها جای مناسبی برای پیاده سازی منطق تجاری نیست. BL که شما از آن یاد می‌کنید در لایه‌ی دیگری رخ می‌دهد ، یکی اسم آم را Task می‌گذارد ، یکی Service و دیگری BlaBla ..
MVC جایگزینی برای N-Tier نیست ، بلکه روشی برای سازماندهی لایه‌ی نمایش می‌باشد.
ViewModelها (ViewModel === Model Of View) اشیایی هستند که از طریق لایه‌ی سرویس تولید می‌شوند و در واقع داده ای که باید در View به کاربر نمایش داده شود را نگهداری می‌کنند.
نگرانی و مسئولیت Controller فراهم کردن داده (از طریق اجرای Business logic) برای UI می‌باشد.

نویسنده: رضا بزرگی
تاریخ: ۱۳۹۱/۰۶/۰۱ ۱:۱۴

- تصور من اینست که معماری چندلایه یک stack است از لایه‌های مختلف با وظایف متفاوت. که پایین‌ترین سطح(دیتا سورس) تا بالاترین سطح(رابط کاربری) هست. در این بین بقیه اجزا مثل BL و غیره قرار میگیره. سوالم اینه که اگر بخواهیم یه الگوی نسبتا کلی ارائه بدیم این لایه‌ها چطور روی هم چیده میشه.
- تجسم این لایه‌ها شاید از این لحاظ مهم باشه که یکی از مهمترین مفاهیم شی‌گرایی یعنی کپسوله‌سازی و Information hidding باعث تولید ایده‌ی چند لایه‌ای هست. و دانستن و اجرای درست اون خیلی کار توسعه را آسانتر میکنه.
- N-Tier یک معماری برای طراحی هست. ولی MVC یه الگوی طراحی. و این‌ها جایگزین که نه، بلکه به نوعی با هم‌پوشانی کار را اصولی‌تر میکند. همانطور که جناب نصیری در سری MVC اشاره‌ای موکد داشتند که Model در MVC در واقع همان ViewModel است . که این ViewModel از الگوی MVVM آمده.

نویسنده: شاهین کیاست
تاریخ: ۱۳۹۱/۰۶/۰۱ ۱:۴۸

-معماری n-tier یک مدل برای توسعه ی نرم افزارهای انعطاف پذیر و با قابلیت استفاده‌ی مجدد می‌باشد. MVC یک Architectural_pattern است.
-MVC در یک معماری چند لایه وظیفه‌ی لایه‌ی نمایش (Presentation) را بر عهده دارد. همانند MVP در Windows Forms یا Web Forms. http://allthingscs.blogspot.de/2011/03/mvc-vs-3-tier-pattern.html

-ViewModel الگوی MVVM با ViewModel که ما در MVC داریم تفاوت زیادی دارند و ارتباطی با هم ندارند.

نویسنده: اصغر ترابی
تاریخ: ۱۳۹۱/۰۶/۰۱ ۴:۷

با عذر خواهی
ولی متاسفانه این ایده که منطق تجاری را در کنترلر قرار دهیم کاملا غلط است.واضح‌ترین دلیل هم این است که منطق تجاری ممکن است بین چندین استفاده کننده مشترک باشد، خیلی ساده فرض کنید قرار است نسخه وب و ویندوز یک سیستم ساخته شود.

نویسنده: رضا
تاریخ: ۱۳۹۱/۰۶/۰۱ ۹:۶

با سلام، معماری لایه ای طراحی‌های متفاوتی داره
و هر طراحی تو سناریوی خاصی ممکنه پیگیری بشه
ولی دو نکته همیشه پابرجا هستش
اول این که چون لایه UI پیچیده‌ترین لایه هستش، با یکی از **الگوهای معماری** زیر سر و سامانی پیدا می‌کنه :
Model - View - Controller
Model View - View Model

Model View Presenter
نکته دوم این هستش که سعی می‌کنن معماری لایه ای رو به صورتی پیاده سازی کنند که بر **N-Tier Development** و Tierهای فیزیکی برنامه منطبق بشه
مثلا DA و BL و Service Layer در سمت سرور، و View و View Model در سمت کلاینت در یک برنامه Desktop
امیدوارم مطلبم شفافیت رو افزایش داده باشه

نویسنده: محمد صاحب
تاریخ: ۱۳۹۱/۰۶/۰۱ ۹:۲۴

iPODD سرنام Idiomatic Presentation, Orchestration, Domain and Data هست من لینک خاصی تو نت ازش ندیدم تو همون کتاب که معرفی کردم باهاش آشنا شدم...
کتاب رو اگه پیدا نکردی بگو برات میل کنم.

نویسنده: شاهین کیاست
تاریخ: ۱۳۹۱/۰۶/۰۱ ۱۱:۵

در مطلب فعلی هم سعی شده به همین موضوع پرداخته شه که منطق تجاری را در Controller قرار ندهید.
گفته شده Controller باید محل استفاده از خروجی منطق تجاری باشد.

نویسنده: امیرحسین جلوداری
تاریخ: ۱۳۹۱/۰۶/۰۲ ۰:۶

الان برای من یه سوالی پیش اومد :
طبق حرفایه شاهین و بقیه‌ی بچه‌ها الان MVC از دید معماری سه لایه داره رو لایه‌ی نمایش مانور میده ... خوب برا چی نیومدن بگن VMVC ؟! (ViewModel-View-Controller) ...چون الان Model در واقع همون ViewModel هست! (اون چیزی که تو View استفاده میشه نه Domain Model!)

نویسنده: شاهین کیاست
تاریخ: ۱۳۹۱/۰۶/۰۲ ۰:۲۵

در مطالب قبلی هم گفته شده بود که M در ASP.NET MVC همان ViewModel هست یا Model Of View .
M در الگوی MVC (الگوی MVC به صورت عمومی در همه‌ی چارچوب ها) هم تنها وظیفه‌ی تامین داده‌ی View را دارد. در صفحه‌ی Wikipedia الگوی MVC نوشته شده :

```
A view requests from the model the information that it needs to generate an output representation.
```

تعریف Domain Entity چیز دیگری هست و برای استفاده از الگوی MVC لزوما نیازی نیست که برنامه‌ی ما دیتابیس داشته باشد.
این لینک
هم مفید است.

نویسنده: حسین مرادی نیا
تاریخ: ۱۳۹۱/۰۸/۱۹ ۲۲:۳۱

کدهای بالا رو در نظر بگیرید؛
ممکن است در حین عملیات‌های بررسی آماده بودن یک سفارش ، بررسی معتبر بودن ایمیل کاربر و ... در صورتی که نتیجه عملیات false باشد بخواهیم پیامی به کاربر نمایش داده شود(نمایش دلیل خطا به کاربر خطا(سفارش آماده نیست - ایمیل وارد نشده و ...)). در این حالت چگونه می‌توان این عملیات را به درستی در کد پیاده کرد؟

نویسنده: شاهین کیاست

تاریخ: ۱۳۹۱/۰۸/۱۹ ۲۳:۲۷

برای نمایش پیام به کاربر در ASP.NET MVC روش‌های زیادی وجود دارد ، مثلا می‌توان در ViewModel خود یک پراپرتی جهت نمایش پیغام به کاربر تعبیه کرد و در صورت نیاز به کمک یک Helper در View پیام مورد نظر را نشان داد یا از شیء TempData استفاده کرد.

نویسنده: محسن درپرستی
تاریخ: ۱۳۹۲/۱۲/۰۶ ۲۰:۳۱

اگر طبق راهنمایی‌های همین پست پیش بریم ، من می‌تونم این روش رو پیشنهاد بدم که مقدار متغیر status اگر برابر با successful نبود . به یک متد دیگه ای ارسال بشه که مسئولیت ساخت و برگرداندن پیام مناسب رو برعهده داره . اون پیام‌ها هم بهتره که به جای اینکه در قالب یک سری if یا switch محاسبه بشن در قالب یک dictionary نگهداری بشن .

```
public RedirectToRouteResult Ship(int orderId)
{
   var status = _orderShippingService.Ship(orderId);
   if (status.Successful)
   {
      return RedirectToAction("Shipped", "Order", new {orderId});
   }
   return RedirectToAction("NotShipped", "Order", new {id = orderId, desc = status});
}
```

ولی محل فراخوانی این تابع دوم کجا باید باشه ؟ نظر من اینه که نباید توی این اکشن باشه چون وظیفه‌ی این اکشن چیز دیگری است . بهتره که مقدار status به اکشن NotShipped ارسال بشه و در اونجا پیام استخراج و به view ارسال بشه . چون در اون اکشن احتمالا دلیل ship نشدن سفارش باید به کاربر نمایش داده بشه.