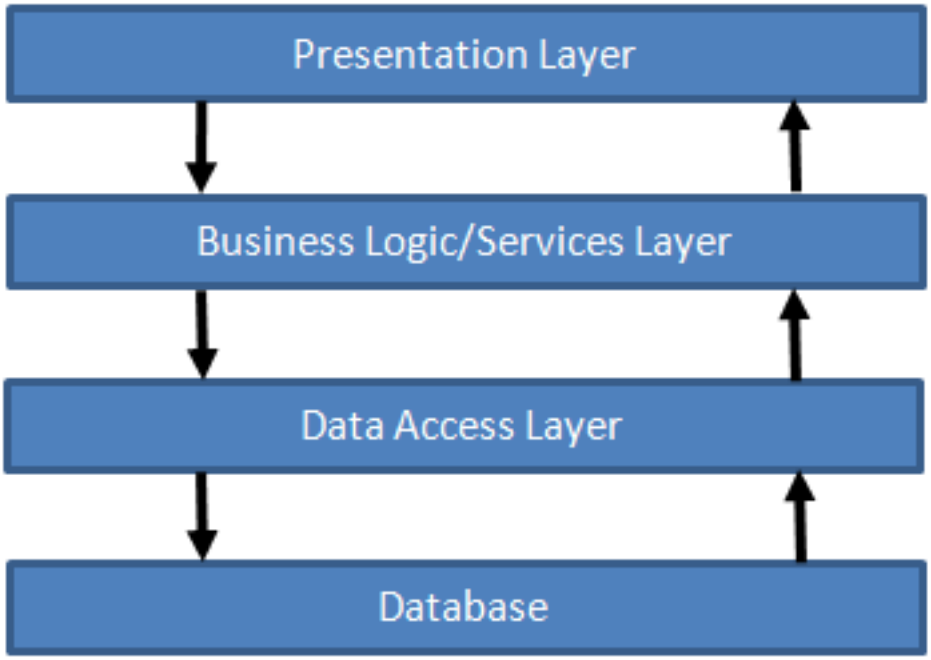
عنوان: **چگونه کد قابل تست بنویسیم - قسمت اول**
نویسنده: محمد بنازاده
تاریخ: ۱۳۹۳/۱۰/۰۱ ۱۳:۲۵
آدرس: www.dotnettips.info
گروه‌ها: Unit testing, Dependency Injection, Mock

مقدمه

نوشتن تست برای کدها بسیار عالی است، در صورتیکه بدانید چگونه این کار را بدرستی انجام دهید. متأسفانه بسیاری از منابع آموزشی موجود، این مطلب که چگونه کد قابل تست بنویسیم را رها می‌کنند؛ بدلیل اینکه آنها مراقبند در بین لایه هایی که در کدهای واقعی وجود دارند گیر نکنند، جایی که شما لایه‌های خدمات (Service Layer)، لایه‌های داده، و غیره را دارید. به ضرورت، وقتی میخواهید کدی را تست کنید که این وابستگی‌ها را دارد، تست‌ها بسیار کند و برای نوشتن دشوار هستند و اغلب بدلیل وابستگی‌ها شکست میخورند و نتیجه غیر قابل انتظاری خواهند داشت.

پیش زمینه

کدی که به خوبی نوشته شده باشد از لایه‌های جداگانه ای تشکیل شده است که هر لایه مسئول یک قسمت متفاوت از وظایف برنامه خواهد بود. لایه‌های واقعی بر اساس نیاز و نظر توسعه دهندگان متفاوت است، ولی یک ساختار رایج به شکل زیر خواهد بود.



لایه نمایش / رابط کاربری : این قسمت کد منطق نمایش و تعامل رابط کاربری می‌باشد. **منطق تجاری / لایه خدمات :** این قسمت منطق تجاری کد شما میباشد. برای نمونه در کد مربوط به یک کارت خرید، این کارت خرید میداند چگونه جمع کارت را محاسبه نماید و یا چگونه اقلام موجود در سفارش را شمارش کند. **لایه دستیابی به داده / لایه ماندگاری :** این کد میداند چگونه به منبع داده متصل شود و یک کارت خرید را بازگرداند و یا چگونه یک کارت را در منبع داده ذخیره نماید.
منبع داده : اینجا جایی است که محتویات کارت در آن ذخیره میشود.
مزیت مدیریت وابستگی‌ها

بدون مدیریت وابستگی‌ها، وقتی شما برای لایه نمایش تست می‌نویسید، کد شما در خدمات واقعی که به کد واقعی دستیابی به منبع داده وابسته است، گرفتار می‌شود و سپس به منبع داده اصلی متصل می‌شود. در واقع، وقتی شما در حال تست نویسی برای

گزینه "اضافه به کارت" و یا "دریافت تعداد اقلام" هستید، می‌خواهید کد را به صورت مجزا تست کنید و قادر باشید نتایج قابل پیش بینی را تضمین نمایید. بدون مدیریت وابستگی‌ها، تست‌های نمایش شما برای گزینه "اضافه به کارت" کند هستند و وابستگی‌ها نتایج غیر قابل پیش بینی بازمیگردانند که میتوانند باعث پاس نشدن تست شما شوند.

راه حل تزریق وابستگی است

راه حل این مسأله تزریق وابستگی است. تزریق وابستگی برای کسانی که تا بحال از آن استفاده نکرده اند، اغلب گیج کننده و پیچیده به نظر میرسد، اما در واقع، مفهومی بسیار ساده و فرآیندی با چند اصل ساده است. آنچه میخواهیم انجام دهیم مرکزیت دادن به وابستگی هاست. در این مورد، استفاده از شیء کارت خرید است و سپس رابطه بین کدها را کم می‌کنیم تا جاییکه وقتی شما برنامه را اجرا می‌کنید، از خدمات و منابع واقعی استفاده کند و وقتی آنرا تست می‌کنید، می‌توانید از خدمات جعلی (mocking) استفاده نمایید که سریع و قابل پیش بینی هستند. توجه داشته باشید که رویکردهای متفاوت بسیاری وجود دارند که میتوانید استفاده کنید، ولی من برای ساده نگهداشتن این مطلب، فقط رویکرد Constructor Injection را شرح میدهم.

گام 1 - وابستگی‌ها را شناسایی کنید

وابستگی‌ها وقتی اتفاق می‌افتند که کد شما از لایه‌های دیگر استفاده می‌نماید. برای نمونه، وقتی لایه نمایش از لایه خدمات استفاده می‌نماید. کد نمایش شما به لایه خدمات وابسته است، ولی ما میخواهیم کد لایه نمایش را به صورت مجزا تست کنیم.

```
public class ShoppingCartController : Controller
{
    public ActionResult GetCart()
    {
        //shopping cart service as a concrete dependency
        ShoppingCartService shoppingCartService = new ShoppingCartService();
        ShoppingCart cart = shoppingCartService.GetContents();
        return View("Cart", cart);
    }
    public ActionResult AddItemToCart(int itemId, int quantity)
    {
        //shopping cart service as a concrete dependency
        ShoppingCartService shoppingCartService = new ShoppingCartService();
        ShoppingCart cart = shoppingCartService.AddItemToCart(itemId, quantity);
        return View("Cart", cart);
    }
}
```

گام 2 – وابستگی‌ها را مرکزیت دهید

این کار با چندین روش قابل انجام است؛ در این مثال من میخواهم یک متغیر عضو از نوع ShoppingCartService ایجاد کنم و سپس آنرا به وهله ای که در Constructor ایجاد خواهم کرد، منتسب کنم. حال هرجا ShoppingCartService نیاز باشد بجای آنکه یک وهله جدید ایجاد کنم، از این وهله استفاده می‌نمایم.

```
public class ShoppingCartController : Controller
{
    private ShoppingCartService _shoppingCartService;
    public ShoppingCartController()
    {
        _shoppingCartService = new ShoppingCartService();
    }
    public ActionResult GetCart()
    {
        //now using the shared instance of the shoppingCartService dependency
        ShoppingCart cart = _shoppingCartService.GetContents();
        return View("Cart", cart);
    }
    public ActionResult AddItemToCart(int itemId, int quantity)
    {
        //now using the shared instance of the shoppingCartService dependency
        ShoppingCart cart = _shoppingCartService.AddItemToCart(itemId, quantity);
        return View("Cart", cart);
    }
}
```

در قسمت بعد درباره مواردی مانند از بین بردن ارتباط لایه ها، تزریق وابستگی و نوشتن تست بحث خواهم کرد.

نظرات خوانندگان

نویسنده: ح مرادف
تاریخ: ۱۳۹۳/۱۱/۱۸ ۰:۲۵

با سلام،

ابتدا از مقاله جذابتون تشکر می‌کنم.

سوالی ذهن بنده رو درگیر کرده :
طبق مقالات آموزش ام وی سی همین سایت بنده لایه سرویسی توی پروژه هام می‌سازم که کارش مشابه بیان شماست :
" **لایه دستیابی به داده / لایه ماندگاری :** این کد میداند چگونه به منبع داده متصل شود و یک کارت خرید را بازگرداند و یا چگونه یک کارت را در منبع داده ذخیره نماید. "

احساس می‌کنم که جای لایه بیزینس توی پروژهام خالیه ، لایه ای که کار محاسبات ریاضی و سایر محاسبات عددی رو به عهده داشته باشه.
از یکی از اساتیدم هم شنیدم که پروژها رو بصورت زیر می‌سازند
لایه دیتا - لایه بیزینس - لایه سرویس - لایه UI

که به نظرم لایه دیتا عملیات CRUD رو به عهده داشته باشه و لایه بیزینس هم محسابات و کارای پیچیده رو انجام بده و لایه سرویس هم لایه ای است که متدهای لازم جهت دسترسی UI به متدهای مورد نیاز در لایه‌های دیتا و بیزینس رو فراهم می‌کنه.
مثلا ثبت یک خرید جدید که موجب اجرای متد Add در کلاس ProductService میشه که در این متد ، متد CalcCommission جهت محاسبه پورسانت‌ها اجرا میشه و سپس نتیجه دریافتیبه کمک متدهای مربوطه در لایه دیتا در دیتابیس ثبت میشه.

به نظر میاد این لایه بندی قشنگ‌تر باشه.
(کل لایه‌های DomainClassess و DatabaseContext و Services پروژه‌های من در لایه دیتا قرار می‌گیرن)

می‌خواستم نظر شما رو درباره لایه بندی بدونم ؟
(به دنبال بهترین روش لایه بندی می‌گردم ، یک استاندارد مطمئن)

نویسنده: محسن خان
تاریخ: ۱۳۹۳/۱۱/۱۸ ۰:۴۶

شما به نظر پرسش و پاسخ‌های EF 12 رو نخوندید یکبار. خلاصه‌اش اینه: زمانیکه از یک ORM استفاده می‌کنید، اون ORM هست که لایه Data شما رو تشکیل می‌ده و لازم نیست که بازنویسی‌اش کنید. لایه بیزنس هم همون لایه سرویس هست (با اون ادغام شده). اگر در مثالی که زده شده، لایه سرویس داخلش فقط Add یا Get هست (که نتیجه‌ی کار با لایه‌ی دیتا رو در اختیار لایه UI قرار می‌ده)، مابقی رو به خلاقیت خواننده واگذار کرده تا خودش جاهای خالی رو پر کنه. مثلا محاسبات هم انجام بده یا کارهای دیگر. هدفش بیشتر این بوده نمایش بده چطور می‌تونید از لایه دیتا در لایه بیزنس استفاده کنید و به اون دسترسی پیدا کنید.
ضمنا سعی کنید دچار over engineering (مدام لایه جدید اختراع کردن) و طراحی باقلوایی نشید.

نویسنده: محمد بنازاده
تاریخ: ۱۳۹۴/۰۴/۰۱ ۱۱:۷

سلام
و عرض پوزش از وقفه در پاسخ

در این مقاله سعی بر این است که نحوه نوشتن یک کد خوب و قابل تست ارائه شود و در مورد معماری نرم افزار و انواع لایه

بندی سطوح نرم افزار بحث نمی‌کند. در واقع آنچه بیان شده برای درک بهتر چرایی نوشتن تست و نحوه آن است.

اما در خصوص لایه بندی نرم افزار تئوری‌های مختلفی وجود دارد و انواع معماری‌های 3 لایه ، 4 لایه (که شما هم اینجا اشاره داشتین) و Domain Drriven Design معرفی شده اند. اما به نظر من DDD بدلیل بروزتر بودن و اینکه قابلیت ترکیب با ابزارها و معماریهای دیگر مانند CQRS , Enterprise Service Bus, ... را دارد معماری بهتری به حساب می‌آید.