عنوان: **پیاده‌سازی الگوی Transaction Per Request در EF**
نویسنده: سیروان عفیفی
تاریخ: ۱۳۹۴/۰۵/۲۰ ۱۷:۱۰
آدرس: www.dotnettips.info
گروه‌ها: Entity framework, MVC, Dependency Injection, StructureMap, DesignPatterns

قبلاً در سایت جاری در رابطه با پیاده‌سازی الگوی Context Per Request مطالبی منتشر شده است. در ادامه می‌خواهیم تمامی درخواست‌های خود را اتمیک کنیم. همانطور که قبلاً در این مطلب مطالعه کردید یکی از مزایای الگوی Context Per Request، استفاده‌ی صحیح از تراکنش‌ها می‌باشد. به عنوان مثال اگر در حین فراخوانی متد SaveChanges، خطایی رخ دهد، کلیه‌ی عملیات RollBack خواهد شد. اما حالت زیر را در نظر بگیرید:

```
_categoryService.AddNewCategory(category);
_uow.SaveAllChanges();

throw new InvalidOperationException();

return RedirectToAction("Index");
```

همانطور که در کدهای فوق مشاهده می‌کنید، قبل از ریدایرکت شدن صفحه، یک استثناء را صادر کرده‌ایم. در این حالت، تغییرات درون دیتابیس ذخیره می‌شوند! یعنی حتی اگر یک استثناء نیز در طول درخواست رخ دهد، قسمتی از درخواست که در اینجا ذخیره‌سازی گروه محصولات است، درون دیتابیس ذخیره خواهد شد؛ در نتیجه درخواست ما اتمیک نیست.
برای رفع این مشکل می‌توانیم یکسری وظایف (Tasks) را تعریف کنیم که در نقاط مختلف چرخه‌ی حیات برنامه اجرا شوند. هر کدام از این وظایف تنها کاری که انجام می‌دهند فراخوانی متد Execute خودشان است. در ادامه می‌خواهیم از این وظایف جهت پیاده‌سازی الگوی **Transaction Per Request** استفاده کنیم. در نتیجه اینترفیس‌های زیر را ایجاد خواهیم کرد:

```
public interface IRunAtInit
{
	void Execute();
}
public interface IRunAfterEachRequest
{
	void Execute();
}
public interface IRunAtStartUp
{
	void Execute();
}
public interface IRunOnEachRequest
{
	void Execute();
}
public interface IRunOnError
{
	void Execute();
}
```

خوب، این اینترفیس‌ها همانطور که از نامشان پیداست، همان اعمال را پیاده سازی خواهند کرد: **IRunAtInit** : اجرای وظایف در زمان بارگذاری اولیه‌ی برنامه. **IRunAfterEachRequest** : اجرای وظایف بعد از اینکه درخواستی فراخوانی (ارسال) شد. **IRunAtStartUp** : اجرای وظایف در زمان StartUp برنامه. **IRunOnEachRequest** : اجرای وظایف در ابتدای هر درخواست. **IRunOnError** : اجرای وظایف در زمان بروز خطا یا استثناءهای مدیریت نشده‌ی برنامه.
خوب، یک کلاس می‌تواند با پیاده‌سازی هر کدام از اینترفیس‌های فوق تبدیل به یک task شود. همچنین از این جهت که اینترفیس‌های ما ساده هستند و هر اینترفیس یک متد Execute دارد، عملکرد آن‌ها تنها اجرای یکسری دستورات در حالات مختلف می‌باشد.
قدم بعدی افزودن قابلیت پشتیبانی از این وظایف در برنامه‌مان است. اینکار را با پیاده‌سازی ریجستری زیر انجام خواهیم داد:

```
public class TaskRegistry : StructureMap.Configuration.DSL.Registry
{
        public TaskRegistry()
        {
            Scan(scan =>
            {
```

```
                scan.Assembliy("yourAssemblyName");
                scan.AddAllTypesOf<IRunAtInit>();
                scan.AddAllTypesOf<IRunAtStartUp>();
                scan.AddAllTypesOf<IRunOnEachRequest>();
                scan.AddAllTypesOf<IRunOnError>();
                scan.AddAllTypesOf<IRunAfterEachRequest>();
            });
        }
}
```

با این کار استراکچرمپ اسمبلی معرفی شده را بررسی کرده و هر کلاسی که اینترفیس‌های ذکر شده را پیاده‌سازی کرده باشد، رجیستر می‌کند. قدم بعدی افزودن رجیستری فوق و بارگذاری آن درون کانتینرمان است:

```
ioc.AddRegistry(new TaskRegistry());
```

اکنون وظایف درون کانتینرمان بارگذاری شده‌اند. سپس نوبت به استفاده‌ی از این وظایف است.
خوب، باید درون فایل Global.asax کدهای زیر را قرار دهیم. چون همانطور که عنوان شد وظایف ایجاد شده می‌بایستی در نقاط مختلف برنامه اجرا شوند:

```
protected void Application_Start()
{
   // other code
   foreach (var task in SmObjectFactory.Container.GetAllInstances<IRunAtInit>())
   {
                task.Execute();
    }
}
protected void Application_BeginRequest()
 {
           foreach (var task in SmObjectFactory.Container.GetAllInstances<IRunOnEachRequest>())
           {
                task.Execute();
           }
}
protected void Application_EndRequest(object sender, EventArgs e)
{
            try
            {
                foreach (var task in SmObjectFactory.Container.GetAllInstances<IRunAfterEachRequest>())
                {
                    task.Execute();
                }
            }
            finally
            {
                HttpContextLifecycle.DisposeAndClearAll();
                MiniProfiler.Stop();
            }
}
protected void Application_Error()
{
            foreach (var task in SmObjectFactory.Container.GetAllInstances<IRunOnError>())
            {
                task.Execute();
            }
}
```

همانطور که مشاهده می‌کنید، هر task در قسمت خاص خود فراخوانی خواهد شد. مثلاً IRunOnError درون رویداد Application_Error و دیگر وظایف نیز به همین ترتیب.
اکنون برنامه به صورت کامل از وظایف پشتیبانی می‌کند. در ادامه، کلاس زیر را ایجاد خواهیم کرد. این کلاس چندین اینترفیس را از اینترفیس‌های ذکر شده، پیاده‌سازی می‌کند:

```
public class TransactionPerRequest : IRunOnEachRequest, IRunOnError, IRunAfterEachRequest
{
        private readonly IUnitOfWork _uow;
        private readonly HttpContextBase _httpContext;
        public TransactionPerRequest(IUnitOfWork uow, HttpContextBase httpContext)
        {
```

```
            _uow = uow;
            _httpContext = httpContext;
        }


        void IRunOnEachRequest.Execute()
        {
            _httpContext.Items["_Transaction"] =
                _uow.Database.BeginTransaction(System.Data.IsolationLevel.ReadCommitted);
        }

        void IRunOnError.Execute()
        {
            _httpContext.Items["_Error"] = true;
        }

        void IRunAfterEachRequest.Execute()
        {
            var transaction = (DbContextTransaction) _httpContext.Items["_Transaction"];
            if (_httpContext.Items["_Error"] != null)
            {
                transaction.Rollback();
            }
            else
            {
                transaction.Commit();
            }
        }
}
```

توضیحات کلاس فوق:

در کلاس TransactionPerRequest به دو وابستگی نیاز خواهیم داشت: IUnitOfWork برای کار با تراکنش‌ها و HttpContextBase برای دریافت درخواست جاری. همانطور که مشاهده می‌کنید در متد IRunOnEachRequest.Execute یک تراکنش را آغاز کرده‌ایم و در IRunAfterEachRequest.Execute یعنی در پایان یک درخواست، تراکنش را commit کرده‌ایم. این مورد را با چک کردن یک فلگ در صورت عدم بروز خطا انجام داده‌ایم. اگر خطایی نیز وجود داشته باشد، کل عملیات roll back خواهد شد. لازم به ذکر است که فلگ خطا نیز درون متد IRunOnError.Execute به true مقداردهی شده است.

خوب، پیاده‌سازی الگوی Transaction Per Request به صورت کامل انجام گرفته است. اکنون اگر برنامه را در حالت زیر اجرا کنید:

```
_categoryService.AddNewCategory(category);
_uow.SaveAllChanges();

throw new InvalidOperationException();

return RedirectToAction("Index");
```

خواهید دید که عملیات roll back شده و تغییرات در دیتابیس (در اینجا ذخیره سازی گروه محصولات) اعمال نخواهد شد.