ALLAN THRAEN | 2 years ago | PDF |

Life as a Coder Optimizely (Episerver) CMS .NET Development

WHEN BEST PRACTICE ISN'T THE BEST - DEPENDENCY INJECTION AND OPTIMIZELY CMS

Some people live and breath 'best practice' development. I am not one of them. Risk is, in-experienced developers (or sometimes experienced) might use them just cause they are 'best practice' and not think more about it. Even when it turns out they are not. Here is little example...

Dependency Injection is great, right? Easy to configure services and then you can use them anywhere you need them. And if you later want to replace a service or mock it for testing it's easy peasy.

I use it many places - and the ability to intercept and replace existing services can sometimes help do almost magical stuff when you build on a big framework such as Optimizely CMS.

When using services from DI, the best practice that has always been preached has always been:

1. "Use Constructor injection if at all possible."
2. "If it's not possible, then at least use the Injected<> property"
3. "By all means avoid ServiceLocator.Current.GetInstance<> as it's an antipattern."

And - when you think about it, often this advice makes sense. For example Constructor injection makes your requirements very obvious from the start as the class cannot be instantiated unless they are provided. And it's easy to see when you start having so many dependencies that perhaps your class is trying to do too much and not adhering to the Single Responsibility Principle (if you follow that religion).

If you can't do constructor injection then an Injected<> property makes sense as you'd assume it does some sort of lazy load and only does the service lookup work once - not to mention that if you structure your code nicely the Injected property are fairly easy to spot and keep track of.

And naturally the approach to avoid is to fill your code body with individual lookups - both for performance, readability and maintainability.

But - sometimes that is actually the best method to use. Here's a little story from the real life:

Self service solution with user specific data-contexts

I have several clients that have some sort of self-service portal - which is very normal on Optimizely CMS sites. A place where clients or customers can log in and see their data and edit it. This means that once a user is logged in, it's very convenient to have one (or multiple) data context services available everywhere in your code where you can access data around the currently logged in user you are serving. Naturally, these services are registrered through dependency injection - but usually as Scoped services - meaning they will be created once for every request. Once created they either load the data right away or maybe they have some sort of lazy loading mechanism.

Here is a very simplified example of one:

```
public class UserContextService
{
    public string Name { get; set; }

    public DateTime LoginTime { get; set; }

    public UserContextService() {
        //Load and cache details about the current user
        LoginTime= DateTime.Now;

        Name = (LoginTime.Second % 2 == 0) ? "Allan" : "Olga";
    }
}
```

To simulate different users, I simply change the name depending on the load time (odd/even second). So far so good. It's a service with a user context and it works.

But - let's then pretend that one day we are building another service for dependency injection. Maybe it's a utility service that can be used many places so you register it as a Singleton, so it won't be instantiated every time you need it.

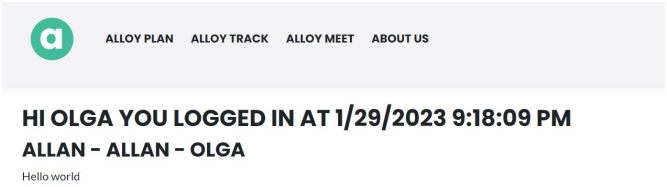
```
services
    .AddCmsAspNetIdentity<ApplicationUser>()
    .AddCms()
    .AddAlloy()
```

```
            .AddScoped<UserContextService>()

            .AddSingleton<MyOtherService>()

            .AddAdminUserRegistration()
            .AddEmbeddedLocalization<Startup>();
```

And then one day - a junior developer needs it to use some properties from the current users context - and of course follows best practice and uses constructor injection. It could look like this:

```
public class MyOtherService
{
    private readonly UserContextService _userContextService;

    private Injected<UserContextService> _injector;
    public MyOtherService(UserContextService userContextService)
                                                                {
        _userContextService = userContextService;
    }

    public string Name1 => _userContextService.Name;

    public string Name2 => _injector.Service.Name;

    public string Name3 => ServiceLocator.Current.GetInstance<UserContextService>().Name;

}
```

Now - think about it! If we were to see which values "MyOtherService" would return as Name1, Name2 or Name3 - what would they be?



The first line uses correct data take from the UserContextService.

Below I output Name1 - Name2 - Name3 from "MyOtherService". Since MyOtherService is a singleton, it will keep and use the same instance of the constructor injected service as it was given when it startet - and whatever user happened to be there at that time. As for Name2, the Injected<> is lazy loaded and placed in a singleton - so it will keep the value of the current user that requests it the first time. *The only approach here that works as intended is the ServiceLocator.*

- "But, you could just have registered the other service as a Scoped service as well", I hear you cry. And you are correct. But it's not always obvious. And sometimes the lifetimes of the class you use are not as easy to control as we'll see in this next example...

The custom visitor group criterion

When you are building websites in Optimizely CMS where users can log in, it's very normal to include some custom visitor group criteria to personalize the experience throughout the website. And if you already have a User context service in your dependency injection they are fast and easy to write. Like this one:

```
[VisitorGroupCriterion(DisplayName = "Current User", Category = "User criteria")]
public class UserCriterion : CriterionBase<UserCriterionModel>
{

    private UserContextService _contextService;
    public UserCriterion(UserContextService ucs) {
        _contextService= ucs;
    }

    public override bool IsMatch(IPrincipal principal, HttpContext httpContext)
                                                                            {
        return _contextService.Name == Model.Name;
    }
}

public class UserCriterionModel : CriterionModelBase
{
    #region Editable Properties

    public string Name { get; set; }

    #endregion

    public override ICriterionModel Copy()
                                        {
        return base.ShallowCopy();
    }
}
```

Notice how we again follow best practice and use constructor injection? That's best practice so we'll blindly follow it. And the criterion is not registered as a singleton so we should be just fine, right?

Nope. Not. At. All. Behind the scenes Optimizely is using the individual criteria (including new ones we make) as something very 'singletonish'. You would have no way of knowing this unless you were in the product management or product development team around 2010/2011 when the Visitor Groups feature was built - or unless you really studied visitor groups very well. They are built this way so you can initialize them and subscribe to events in the system (like SessionStart and similar). And probably a few other reasons I've forgotton over the years.

So - once again your personalization might not work quite as you had hoped if you followed 'best practices' all the way.

Life as a Coder | Optimizely (Episerver) | CMS | .NET Development

CodeArt ApS
Teknikerbyen 5, 2830 Virum, Denmark
Email: info@codeart.dk
Phone: +45 26 13 66 96
CVR: 39680688