← Blog posts

Anti-Pattern: Don't modify Optimizely CMS (Episerver) content objects in the Controller

C#, TIPS AND TRICKS, CMS, .NET DEVELOPMENT, OPTIMIZELY (EPISERVER)


Allan Thraen | 3 Months Ago | PDF |

Using your content object (CurrentPage / CurrentBlock) as a makeshift viewmodel where you change settings or extend it with user data in the controller before passing it to the view, is unfortunately (and to my surprise) a pretty wide-spread practice among developers implementing Optimizely (Episerver) web sites. But it really needs to stop.

In my line of work I get to visit a lot of different clients and review or enhance their Optimizely(Episerver) codebase. So I regularly get to see code written by many different developers from many different agencies - through the last 10 years.

I see a lot of great code - but also sometimes something really bad. This blog post is about an anti-pattern (bad programming practice) that I unfortunately have seen quite a few times - in different codebases with different clients.

A bit of history

Back in the day, when the latest Episerver was 5 or 6, and strongly-typed content was barely imagined in Joel Abrahamsson's mind, a developer would get a PageData object from the DataFactory, which was a 1-size-fits-all class for content. It had a property named "Property" (yes, the property-property as we affectionately called it) and it was a dictionary containing all the properties of the specified content object. All the built-in properties had names starting with "Page" like "PageName", "PageTypeName", "PageLink" and so on.

Fun fact - a lot of this still exists behind the scenes - try to look for the Property-property in your IContentData interface.

If you wanted to write to any property, you would use the SetData(string key, object value) on the PageData object. But usually you would know better - because doing so would always throw an exception, unless you first created a writeable clone of the pagedata object by calling the .CreateWriteableClone() - a method that would create a new PageData object with a copy of the properties, but with a flag set, making it writeable.

You would then change the properties you needed to, and then call the DataFactory.Save method to save it back to the database. Cause that was the only reason for changing the object. Ever.

Back then - I used to do Episerver developer courses from time to time - and pretty much every time this thing with .CreateWriteableClone came up - why was it needed? Why can't I modify the object? It's just a an object, right?

The problem

A lot has changed to where we are with Optimizely CMS today (version 11/12). But under the hood, there's also quite a few things that remain the same (albeit "optimized" of course - pun intended). We've learned dependency injection, and we are now on MVC instead of web forms (some have probably even moved on to Blazor or Razor Pages or decouple frontend or ...). It's no longer just pages - it's IContent. But when we get a content object it's still the exact same situation as it was back then. But since the content objects are now strongly typed, we no longer get an exception if we write to a read-only object. And this is the problem:

Content objects you get from the CMS is a reference to *the* object that represents that page in the cache. Changing that object will change it for all other requests on that server.

This was why the content objects are read-only. It's shared with everybody else - on that server.

Now - you might be shaking your head saying "Allan, I know this already, why are you wasting my time with this post" and that's fine. If that's you, then stop reading and go get a milkshake :-) If not, let me show why this is so bad:

```
public class MyAwfulBlockController : BlockController<MyAwfulBlock>
{
    public override ActionResult Index(MyAwfulBlock currentBlock)
                                                        {
        var userDetails = FetchCurrentUserDetails(); //Imaginary helper method that fetches information specific to this request
        currentBlock.UserDetails = userDetails;
        return PartialView(currentBlock);
    }
}
```

The above example is not uncommon. Why bother creating a seperate viewmodel to pass between the Controller and the view, when we already have this fine Content object, right? We can just add a property (perhaps hide it from the editors) and then use it as we would on any other object, right? Oh shared object you say? Well, that's ok, cause I set it every time the controller is called, so everybody is going to see their own details, right?

Well - most of the time, that would be true. Except when you have 2 users hitting the same page at almost the same time. The first user would be through the controller logic and into rendering when the second starts the controller logic - let's say 10 ms apart. And then the values in the shared object would start to change for User 1, while rendering.

Yes, that's a dreaded **race-condition**. And it can be really hard to spot unless you see it in code-reviews. And user error reports will be impossible to recreate.

Obviously the above example is a worst-case. Showing one user data that belongs to another is data-breach and can have big consequences. But the same problem can show itself in different ways.

What if you do a controller filtering of items in a content area based on personalization? Then some marketing campaigns might be wrong, for the user seeing them. What if you have a commerce setup and look up personalized prices in the controller? If you also let them piggybag on your content model, you'll potentially be showing the wrong prices to the wrong people. And so on.

The right solution

The solution to this is easy. Just create a viewmodel and use it to pass between the controller and the view. Sure, your viewmodel can contain the current content object as a property - that's no problem. As long as you stick to modifying the view model and not the content object. Like this:

```
public class MyGreatBlockController : BlockController<MyGreatBlock>
{
    public override ActionResult Index(MyGreatBlock currentBlock)
                                                        {
        var userDetails = FetchCurrentUserDetails(); //Imaginary helper method that fetches information specific to this request
        MyGreatBlockViewModel viewModel = new MyGreatBlockViewModel(currentBlock); //Alternative use a view model fac
        viewModel.UserDetails = userDetails;
        return PartialView(currentBlock);
    }
}
```

C# TIPS AND TRICKS CMS .NET DEVELOPMENT OPTIMIZELY (EPISERVER)

Post Comments 0