ALLAN THRAEN | 3 years ago | PDF |

Tips and Tricks CMS Optimizely (Episerver)

SELF SERVICE WITH FLUID TEMPLATING FOR EDITORS IN OPTIMIZELY CMS

A classic need in many websites - especially self-service sites - is a placeholder mechanism, so editors can use placeholders in their text to be replaced with user specific data. Recently, working with a client, we ran into this problem and tried out a new approach to empower the content creators to solve this themselves.

The problem

It's quite a common problem on many websites that you need to embed dynamic textual content mixed in with your editorial content.

For example:

- **Commerce websites** that has content articles to promote a certain product, but want to make sure that product naming and pricing that is part of the article is always up to date. Case: Airline website has an article featuring all the great stuff you can see and explore in Venice, and also includes "as low as [price (+ currency)] return from [current users city] this [month]". But the price and month, city and currency changes all the time and has dependencies on the current visitor.
- **Selfservice websites** that shows texts with personal information like the current users name ("Welcome back, Allan"), messages that vary based on profile/account info ("Your account is almost empty, would you like to make a deposit") and so on.
- **Corporate websites** that includes factual information various places on the site, that can change. Like "Copyright [year]" or "Reach out to [name and email of north eastern sales rep] to inquire about pricing."

I've seen a multitude of solutions to this. Some will have many very customized block types, that contain many different string properties like "Text before name" or "Text if account is empty" and so on. Some will have .NET string formatting placeholders in the strings "You have {0} $ on your account" - and then an explanation in the property description as to what the placeholder does. And some will inject blocks in Xhtml properties or use many different visitor group rules to differentiate the messaging.

A few have implemented better templating support - often based on a string replace for certain keys or if it's really advanced regular expression.

All these solutions have issues on their own. And many of them require code changes and new deploys for even smaller logical changes.

Fluid to the rescue

Shopify has developed a templating language that solves this in what I think is a pretty neat way, called Liquid. It's open source, and Sebastien Ros has made a great .NET implementation called Fluid. It's pretty great, for a number of reasons:

1. It's pretty powerful and still fairly easy to use.
2. It's an easy nuget package to implement and running a parser is just a few lines.
3. You can easily expose your own datamodels and even methods in it
4. Even though it's powerful, it's still limited and sandboxed - so you are not opening up a big security risk in letting editors use this.

On the Fluid websites there are some examples of how to use it - in it's simple form it can be placeholders wrapped in double-curly brackets {{user.name}}, but it can go a lot further - like loops, conditional statements, transformations and so on.

Legendary Deane Barker also wrote a nice guide to Fluid, here: https://deanebarker.net/tech/fluid/.

All in all, this seemed pretty perfect as an editorial templating language and we started implementing it.

Implementation

See in the bottom of the post for the GIST with the actual code samples. This sample is implemented on Optimizely CMS (Episerver) version 12 and running .NET 5, but I suspect it should also work on earlier versions without much trouble.

There are multiple approaches to how this could be implemented.

First of all, we need some logic to setup the parser, configure the data models and parse a given string.

For this purpose I set up a FluidHelper service and registered it in the Startup class, as

```
            services
                .AddCmsAspNetIdentity<ApplicationUser>()
                .AddHttpContextAccessor()
```

```
                .AddCms()

                //Add Fluid related services
                .AddScoped<UserContext>()
                .AddScoped<FluidHelper>()
                //---

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

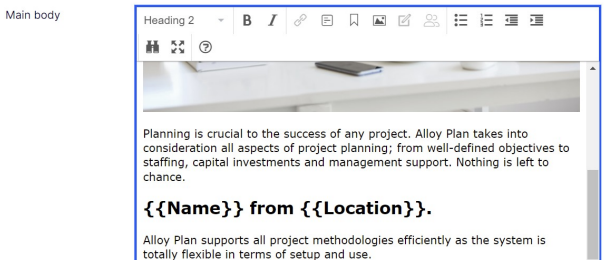
The key thing here is to register it as Scoped, so it's persisted throughout the request - but not shared between requests. This way we can use the same parser, loaded with the current user context multiple places.

For my example the user context is mocked - but naturally this could be where you would have stored profile info about the current user.

As you can see in the example code below, I've opted for both an extension method that can be used during rendering, but also a DisplayTemplate for FluidString that will work on any string property with the attribute [UIHint("FluidString")]. This is done by registrering an EditorDescriptor for it.

I also wanted it supported in all XhtmlStrings, so I made a general DisplayTemplate for those which incorporates it (however, be aware that the current version might not be ideal - as it hasn't been tested to work with personalization yet).

Now, we can start using fluid mixed with our content. Here is a pretty basic example, I'll leave you to explore the before mentioned resources for a full description of the possibilities with Fluid.



And - when rendered in Preview or View mode, this is obviously what you get:

Is it too complicated?

A very valid concern is of course: Is it too complicated a syntax for content creators to learn?

And yes - it might not be for all content creators to enjoy the full functionality available with the more complex syntax of Liquid/Fluid. But let's keep in mind that this is a way to solve a problem that's already very complicated for content creators to deal with in the above outline scenarios. And this does empower them further. And sure, sometimes they might need to ask a developer for some help in a very specific case, but without Fluid that case would almost certainly have meant that a new feature had to be developed and a new site would have to be deployed to production. Now, a developer can help put together a single line of content, standing next to the editor, trying it out in preview mode.

Future

I think it could be really useful to enhance the editorial support for Fluid. This could either be with better fluid-enabled properties (for example with syntax highlighting - the ACE editor actually already supports that, so it wouldn't be hard to add to the editordescriptor in my sample code below.

Or even better - a sidebar widget with a 'cheat-sheet'. A list of placeholder expressions that can easily be dragged into your content where needed. And perhaps a "Fluid Tester Page" where web managers or developers can construct and test new fluid statements for the cheat-sheet.

Look out for future blog posts on this topic - and let me know your thoughts and ideas in the comments below!

```
using EPiServer.ServiceLocation;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace LocalAlloy.Fluid
{
    public static class FluidExtensions
    {
        public static string ParseFluid(this string text)
        {
            return ServiceLocator.Current.GetInstance<FluidHelper>().Parse(text);
        }
    }
}
```

FluidExtensions.cs hosted with ❤ by GitHub view raw

```
using EPiServer;
using EPiServer.Core;
using Fluid;
using LocalAlloy.Models.Pages;
using Microsoft.AspNetCore.Http;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace LocalAlloy.Fluid
{
    public class FluidHelper
    {
        private FluidParser _parser;
        private TemplateContext _context;

        public FluidHelper(UserContext userContext, IHttpContextAccessor httpContext, IContentRepository repo)
        {
            _parser = new FluidParser();
            //Since this is registered as a Scoped service, we can create one context to use.
            _context = new TemplateContext(userContext);
            //Register other models to be available in Fluid, for example the HttpRequest:
            _context.Options.MemberAccessStrategy.Register<HttpRequest>();
            _context.SetValue("request", httpContext.HttpContext.Request);

            //Or the start page with shared properties
            _context.Options.MemberAccessStrategy.Register<StartPage>();
            _context.SetValue("start", repo.Get<StartPage>(ContentReference.StartPage));


            //TODO: Set other context settings or add other services

        }

        public string Parse(string input)
        {
            if (_parser.TryParse(input, out IFluidTemplate result))
            {
                return result.Render(_context);
            } else return input;
        }
    }
}
```

FluidHelper.cs **hosted with ❤ by** GitHub — view raw

```
//Located in Shared/DisplayTemplates
@model String
@inject LocalAlloy.Fluid.FluidHelper helper
@Html.Raw(helper.Parse(Model))
```

FluidString.cshtml **hosted with ❤ by** GitHub — view raw

```
using EPiServer.Shell.ObjectEditing;
using EPiServer.Shell.ObjectEditing.EditorDescriptors;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace LocalAlloy.Fluid
{
    [EditorDescriptorRegistration(TargetType = typeof(string), UIHint = "FluidString")]
    public class FluidStringEditorDescriptor : StringEditorDescriptor
    {

        public FluidStringEditorDescriptor() {
            this.ClientEditingClass = "dijit/form/Textarea";
        }

    }
}
```

FluidStringEditorDescriptor.cs **hosted with ❤ by** GitHub — view raw

```
//Only ConfigureServices included here:
 public void ConfigureServices(IServiceCollection services)
        {
            if (_webHostingEnvironment.IsDevelopment())
            {
                AppDomain.CurrentDomain.SetData("DataDirectory", Path.Combine(_webHostingEnvironment.ContentRootPath, "App_Data"));

                services.Configure<SchedulerOptions>(options => options.Enabled = false);
            }

            services
                .AddCmsAspNetIdentity<ApplicationUser>()
                .AddHttpContextAccessor()
                .AddCms()
                //Add Fluid related services
                .AddScoped<UserContext>()
                .AddScoped<FluidHelper>()
                //---
                .AddAlloy()
                .AddAdminUserRegistration()
                .AddEmbeddedLocalization<Startup>();
        }
```

Startup.cs **hosted with ❤ by** GitHub — view raw

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace LocalAlloy.Fluid
{

    //Sample, hardcoded user context
    public class UserContext
    {
        public string Name => "Allan";

        public int Age => 42;

        public string Location => "Denmark";

        public string Company => "CodeArt";

        public bool IsAdmin => true;
    }
```

```
22  }
```

UserContext.cs hosted with ❤ by GitHub view raw

```
1  //In Shared/DisplayTemplates
2  @model XhtmlString
3  @inject LocalAlloy.Fluid.FluidHelper helper
4  @Html.Raw(helper.Parse(Model.ToHtmlString()))
```

XhtmlString.cshtml hosted with ❤ by GitHub view raw

Tips and Tricks CMS Optimizely (Episerver)

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