uJet Documentation

Release 4.2.0

anton(at)logikfabrik.se

May 05, 2018

Contents

Umbraco Jet (uJet) is a Code First approach to building MVC applications in Umbraco 7. Declare your document, media, member, and data types in code, and have uJet move them into Umbraco for you - simplifying development, deployment and versioning.

uJet is capable of serving you with instances of your document types. With uJet you're no longer bound by the *RenderModel*, or by the constraints set by the built-in *ControllerActionInvoker*. uJet brings you strongly typed views without you having to roll your own view models.

uJet is a developer tool for Umbraco 7, released as Open Source (MIT). It supports document types and media types (including inheritance and composition), member types, data types, and template synchronization.

CHAPTER 1

Contents

1.1 Requirements

uJet is compatible with Umbraco 7, target framework .NET 4.5 and requires Full Trust.

1.1.1 NuGet Requirements

The latest uJet NuGet targets Umbraco CMS 7.6.0 and above.

1.1.2 Development Requirements

The uJet source code is written in C# 6.0 (.NET 4.5) using Visual Studio 2015.

1.2 Installation

uJet can be installed by downloading, and compiling the uJet source code, and then referencing the compiled uJet assembly. Or by using a precompiled assembly through NuGet.

1.2.1 NuGet

uJet is available on NuGet. Simply create a new project in Visual Studio and add the uJet NuGet. References to the Umbraco 7 core binaries and uJet binaries will be set up for you.

```
PM> Install-Package uJet
```

Find all available versions on NuGet.

1.2.2 Source Code

You'll find the uJet source code on GitHub. Open Visual Studio, clone the GitHub repository and compile the solution. Create a new project, reference Umbraco 7 and the compiled uJet assembly.

1.3 Configuration

uJet can be configured in *web.config*. The section *logikfabrik.umbraco.jet* allows for configuration.

```
<configuration>
  <configSections>
    <section name="logikfabrik.umbraco.jet" type="Logikfabrik.Umbraco.Jet.
↪Configuration.JetSection, Logikfabrik.Umbraco.Jet" />
  </configSections>
  <logikfabrik.umbraco.jet synchronize="...">
    <assemblies>
      <add name="..." />
    </assemblies>
  </logikfabrik.umbraco.jet>
</configuration>
```

1.3.1 Types of Type Classes to Scan

uJet scans assemblies, looking for all types of type classes (document, media, member, and data types), by default. To limit the scan it's possible to combine the constants of the *SynchronizationMode* enumeration in *web.config*, e.g. *DocumentTypes, DataTypes* to scan and synchronize document, and data types only.

The following constants are declared in the *SynchronizationMode* enumeration.

Constants	
None	Do not scan or synchronize type classes
DocumentTypes	Scan and synchronize document type classes
MediaTypes	Scan and synchronize media type classes
MemberTypes	Scan and synchronize member type classes
DataTypes	Scan and synchronize data type classes

Note: Template synchronization and use of the built-in preview template, by using *PreviewTemplateAttribute*, requires document type synchronization to be enabled. Once uJet has synchronized all document types, document type synchronization can be disabled; it will still be possible to preview documents through the Umbraco back office. Template synchronization will, on the other hand, be disabled.

```
<configuration>
  <configSections>
    <section name="logikfabrik.umbraco.jet" type="Logikfabrik.Umbraco.Jet.
↪Configuration.JetSection, Logikfabrik.Umbraco.Jet" />
  </configSections>
  <logikfabrik.umbraco.jet synchronize="..." />
</configuration>
```

1.3.2 Assemblies to Scan

uJet scans all assemblies in the app domain, looking for all types of type classes, by default. To limit the scan, it's possible to declare assemblies to scan in *web.config*. Assemblies are added by full name, case sensitive. No other assemblies in the app domain will be scanned.

```
<configuration>
  <configSections>
    <section name="logikfabrik.umbraco.jet" type="Logikfabrik.Umbraco.Jet.
↪Configuration.JetSection, Logikfabrik.Umbraco.Jet" />
  </configSections>
  <logikfabrik.umbraco.jet>
    <assemblies>
      <add name="..." />
    </assemblies>
  </logikfabrik.umbraco.jet>
</configuration>
```

1.4 Working with Types

uJet supports document types and media types (including inheritance and composition), member types and data types.

1.4.1 Document Types

A document type is created by decorating a public non-abstract class, with a default constructor that takes no parameters, using the *DocumentTypeAttribute* attribute.

```
using Logikfabrik.Umbraco.Jet;

namespace Example.Models.DocumentTypes
{
    [DocumentType("My Page")]
    public class MyPage
    {
    }
}
```

Tip: Your document type classes can be concidered models. Following ASP.NET MVC conventions, models are placed in the *Models* folder. It's recommended to place all document type classes in *Models\DocumentTypes*.

When your Umbraco application is started, uJet will scan all assemblies in the app domain, looking for document type classes. Found classes will be used as blueprints to synchronize your database.

Note: Assemblies to scan can be configured. Having uJet scan all app domain assemblies will have an impact on performance. Configuring assemblies is recommended if synchronization is enabled in your production environment.

DocumentTypeAttribute Properties

The following document type properties can be set using the *DocumentTypeAttribute* attribute.

Id

The document type identifier. Specifying a document type identifier will enable document type tracking. Tracked document types can be renamed; uJet will keep your Umbraco database synchronized.

```
using Logikfabrik.Umbraco.Jet;

namespace Example.Models.DocumentTypes
{
    [DocumentType("F3D4B9F1-711D-40A8-9091-FF5104CE0ACE", "My Page")]
    public class MyPage
    {
    }
}
```

Name

Required The name of the document type. The document type name is displayed in the Umbraco back office.

Description

A description of the document type. The document type description is displayed in the Umbraco back office.

Icon

The icon for the document type. The document type icon is displayed in the Umbraco back office.

```
using Logikfabrik.Umbraco.Jet;

namespace Example.Models.DocumentTypes
{
    [DocumentType("My Settings Page", Icon = "icon-settings")]
    public class MySettingsPage
    {
    }
}
```

Thumbnail

The thumbnail for the document type. The document type thumbnail is displayed in the Umbraco back office.

IsContainer

Whether or not documents of this type are containers (labeled *Enable list view* in the Umbraco back office).

AllowedAsRoot

Whether or not documents of this type can be created at the root of the content tree.

AllowedChildNodeTypes

Which other types are allowed as child nodes to documents of this type in the content tree.

```
using Logikfabrik.Umbraco.Jet;

namespace Example.Models.DocumentTypes
{
    [DocumentType("My Page", AllowedChildNodeTypes = new[] {typeof(OurPage),␣
↪typeof(TheirPage)})]
    public class MyPage
    {
    }
}
```

CompositionNodeTypes

The composition document types of the document type.

```
using Logikfabrik.Umbraco.Jet;

namespace Example.Models.DocumentTypes
{
    [DocumentType("My Page", CompositionNodeTypes = new[] {typeof(OurPage),␣
↪typeof(TheirPage)})]
    public class MyPage
    {
    }
}
```

Templates

The available templates (aliases) of the document type.

```
using Logikfabrik.Umbraco.Jet;

namespace Example.Models.DocumentTypes
{
    [DocumentType("My Page", Templates = new []{"ourTemplate", "theirTemplate"})]
    public class MyPage
    {
    }
}
```

See also:

For more information on the topic of templates see *Working with Templates*.

DefaultTemplate

The default template (alias) of the document type.

```
using Logikfabrik.Umbraco.Jet;

namespace Example.Models.DocumentTypes
{
    [DocumentType("My Page", DefaultTemplate = "myTemplate")]
    public class MyPage
    {
    }
}
```

See also:

For more information on the topic of templates see *Working with Templates*.

1.4.2 Media Types

A media type is created by decorating a public non-abstract class, with a default constructor that takes no parameters, using the *MediaTypeAttribute* attribute.

```
using Logikfabrik.Umbraco.Jet;

namespace Example.Models.MediaTypes
{
    [MediaType("My Media")]
    public class MyMedia
    {
    }
}
```

Tip: Your media type classes can be concidered models. Following ASP.NET MVC conventions, models are placed in the *Models* folder. It's recommended to place all media type classes in *Models\MediaTypes*.

When your Umbraco application is started, uJet will scan all assemblies in the app domain, looking for media type classes. Found classes will be used as blueprints to synchronize your database.

Note: Assemblies to scan can be configured. Having uJet scan all app domain assemblies will have an impact on performance. Configuring assemblies is recommended if synchronization is enabled in your production environment.

MediaTypeAttribute Properties

The following media type properties can be set using the *MediaTypeAttribute* attribute.

Id

The media type identifier. Specifying a media type identifier will enable media type tracking. Tracked media types can be renamed; uJet will keep your Umbraco database synchronized.

```
using Logikfabrik.Umbraco.Jet;

namespace Example.Models.MediaTypes
```

(continues on next page)

(continued from previous page)

```
{
    [MediaType("6E1F2ED5-CBC2-4B46-AE70-79C5C6A9FACC", "My Media")]
    public class MyMedia
    {
    }
}
```

Name

Required The name of the media type. The media type name is displayed in the Umbraco back office.

Description

A description of the media type. The media type description is displayed in the Umbraco back office.

Icon

The icon for the media type. The media type icon is displayed in the Umbraco back office.

Thumbnail

The thumbnail for the media type. The media type thumbnail is displayed in the Umbraco back office.

IsContainer

Whether or not media of this type are containers (labeled *Enable list view* in the Umbraco back office).

AllowedAsRoot

Whether or not media of this type can be created at the root of the content tree.

AllowedChildNodeTypes

Which other types are allowed as child nodes to media of this type in the content tree.

```
using Logikfabrik.Umbraco.Jet;

namespace Example.Models.MediaTypes
{
    [MediaType("My Media", AllowedChildNodeTypes = new[] {typeof(OurMedia),␣
↪typeof(TheirMedia)})]
    public class MyMedia
    {
    }
}
```

CompositionNodeTypes

The composition media types of the media type.

```
using Logikfabrik.Umbraco.Jet;

namespace Example.Models.MediaTypes
{
    [MediaType("My Media", CompositionNodeTypes = new[] {typeof(OurMedia),␣
↪typeof(TheirMedia)})]
    public class MyMedia
    {
    }
}
```

1.4.3 Member Types

A member type is created by decorating a public non-abstract class, with a default constructor that takes no parameters, using the *MemberTypeAttribute* attribute.

```
using Logikfabrik.Umbraco.Jet;

namespace Example.Models.MemberTypes
{
    [MemberType("My Member")]
    public class MyMember
    {
    }
}
```

Tip: Your member type classes can be concidered models. Following ASP.NET MVC conventions, models are placed in the *Models* folder. It's recommended to place all member type classes in *Models\MemberTypes*.

When your Umbraco application is started, uJet will scan all assemblies in the app domain, looking for member type classes. Found classes will be used as blueprints to synchronize your database.

Note: Assemblies to scan can be configured. Having uJet scan all app domain assemblies will have an impact on performance. Configuring assemblies is recommended if synchronization is enabled in your production environment.

MemberTypeAttribute Properties

The following member type properties can be set using the *MemberTypeAttribute* attribute.

Id

The member type identifier. Specifying a member type identifier will enable member type tracking. Tracked member types can be renamed; uJet will keep your Umbraco database synchronized.

```
using Logikfabrik.Umbraco.Jet;

namespace Example.Models.MemberTypes
{
    [MemberType("DAE131E7-1159-4841-A669-3A39A4190903", "My Member")]
    public class MyMember
    {
    }
}
```

Name

Required The name of the member type. The member type name is displayed in the Umbraco back office.

Description

A description of the member type. The member type description is displayed in the Umbraco back office.

Icon

The icon for the member type. The member type icon is displayed in the Umbraco back office.

1.4.4 Data Types

A data type is created by decorating a public non-abstract class, with a default constructor that takes no parameters, using the *DataTypeAttribute* attribute.

```
using Logikfabrik.Umbraco.Jet;

namespace Example.Models.DataTypes
{
    [DataType(typeof(int), "Umbraco.MediaPicker")]
    public class MyData
    {
    }
}
```

Tip: Your data type classes can be concidered models. Following ASP.NET MVC conventions, models are placed in the *Models* folder. It's recommended to place all data type classes in *Models\DataTypes*.

When your Umbraco application is started, uJet will scan all assemblies in the app domain, looking for data type classes. Found classes will be used as blueprints to synchronize your database.

Note: Assemblies to scan can be configured. Having uJet scan all app domain assemblies will have an impact on performance. Configuring assemblies is recommended if synchronization is enabled in your production environment.

DataTypeAttribute Properties

The following data type properties can be set using the *DataTypeAttribute* attribute.

Type

Required The type of the data type. The type property will determine how Umbraco stores property values of this data type in the Umbraco database (*DataTypeDatabaseType.Ntext*, *DataTypeDatabaseType.Integer*, or *DataTypeDatabaseType.Date*).

Editor

Required The editor of the data type. The editor property will determine which property editor will be used for editing property values of this data type in the Umbraco back office.

PreValues

uJet supports pre-values defined in code. Simply add a public property (getter required) with the name *PreValues* of a type implementing interface *IDictionary<string, string>*. uJet will find the property, get the return value and save it as pre-values for the data type.

Implementing interface *IDataType* is optional.

```
using Logikfabrik.Umbraco.Jet;

namespace Example.Models.DataTypes
{
    [DataType(typeof(int), "Umbraco.MediaPicker")]
    public class MyData : IDataType
    {
        public Dictionary<string, string> PreValues => new Dictionary<string, string>
        {
            { "PreValue0", "Value0" },
            { "PreValue1", "Value1" },
            { "PreValue2", "Value2" }
        };
    }
}
```

1.4.5 Type Tracking

When a document, media, or member type is synchronized, uJet tries to match the type declared in code with a type definition. uJet creates an Umbraco alias for the type, based on the type name (namespace excluded), and uses that alias to look for a matching type definition in the database. If a match is found the definition is updated; if not, a new type definition is created. Renaming a type that has been synchronized, in code or using the Umbraco back office, will cause duplicate definitions to be created, with different aliases.

Type tracking refers to the use of the *id* parameter when declaring document types, media types, and member types in code. Using the *id* parameter uJet can keep track of types and their corresponding type definitions without relying on the type names. With type tracking, types can be renamed; uJet will keep your Umbraco database synchronized.

1.5 Working with Properties

Document, media, and member type properties are created by adding properties with public getters and setters to classes decorated using the *DocumentTypeAttribute*, *MediaTypeAttribute*, or *MemberTypeAttribute* attributes.

When your Umbraco application is started, uJet will scan your document, media, and member type classes, looking for properties. Found properties will be used as blueprints to synchronize your database.

1.5.1 Data Annotations

Data annotations can be used to customize the user experience in the Umbraco back office, e.g. to set property name and description.

Supported Data Annotations

uJet supports the following data annotations.

- *RequiredAttribute*
- *DefaultValueAttribute*
- *RegularExpressionAttribute*
- *UIHintAttribute*
- *DisplayAttribute*
- *ScaffoldColumnAttribute*
- *AliasAttribute* (uJet specific)
- *IdAttribute* (uJet specific)

RequiredAttribute

Properties decorated using the *RequiredAttribute* attribute will be mandatory in the Umbraco back office.

```
using Logikfabrik.Umbraco.Jet;
using System.ComponentModel.DataAnnotations;

namespace Example.Models.DocumentTypes
{
    [DocumentType("My Page")]
    public class MyPage
    {
        [Required]
        public string MyProperty { get; set; }
    }
}
```

DefaultValueAttribute

Properties decorated using the *DefaultValueAttribute* attribute will have a default value of whatever value was set when setting the attribute.

```
using Logikfabrik.Umbraco.Jet;
using System.ComponentModel;

namespace Example.Models.DocumentTypes
{
    [DocumentType("My Page")]
    public class MyPage
    {
        [DefaultValue("My Default Value")]
        public string MyProperty { get; set; }
    }
}
```

RegularExpressionAttribute

Properties decorated using the *RegularExpressionAttribute* attribute will be validated in the Umbraco back office using the regular expression specified.

```
using Logikfabrik.Umbraco.Jet;
using System.ComponentModel.DataAnnotations;

namespace Example.Models.DocumentTypes
{
    [DocumentType("My Page")]
    public class MyPage
    {
        [RegularExpression("\\w+([-+.']\\w+)*@\\w+([-.]\\w+)*\\.\\w+([-.]\\w+)*")]
        public string MyEmailProperty { get; set; }
    }
}
```

DisplayAttribute

Use the *DisplayAttribute* attribute to customize the property name and description in the Umbraco back office. The *DisplayAttribute* attribute makes it possible to set sort order (*Order*), and property group (*GroupName*) also.

```
using Logikfabrik.Umbraco.Jet;
using System.ComponentModel.DataAnnotations;

namespace Example.Models.DocumentTypes
{
    [DocumentType("My Page")]
    public class MyPage
    {
        [Display(Name = "My Property", Description = "Description of My Property",␣
↪GroupName = "My Tab", Order = 1)]
        public string MyProperty { get; set; }
    }
}
```

UIHintAttribute

Use the *UIHintAttribute* attribute to specify the Umbraco data type used. The Umbraco data type is inferred by the .NET property type by default, but can be overridden using this attribute.

```
using Logikfabrik.Umbraco.Jet;
using System.ComponentModel.DataAnnotations;

namespace Example.Models.DocumentTypes
{
    [DocumentType("My Page")]
    public class MyPage
    {
        [UIHint("ContentPicker")]
        public int MyContentProperty { get; set; }
    }
}
```

ScaffoldColumnAttribute

Properties decorated using the *ScaffoldColumnAttribute* attribute (set to false) will not be available for editors through the Umbraco back office.

```
using Logikfabrik.Umbraco.Jet;
using System.ComponentModel.DataAnnotations;

namespace Example.Models.DocumentTypes
{
    [DocumentType("My Page")]
    public class MyPage
    {
        [ScaffoldColumn(false)]
        public string MyHiddenProperty { get; set; }
    }
}
```

AliasAttribute

Use the *AliasAttribute* attribute to specify a custom property alias, overriding the default value.

IdAttribute

Use the *IdAttribute* attribute to specify a property ID, enabling property tracking.

1.5.2 Data Types

.NET data types are mapped to Umbraco data types using data type definition mappings.

The Umbraco data type mapped will determine how Umbraco stores the property value in the database, and what property editor to use for editing the property value in the Umbraco back office.

Supported .NET Data Types

uJet supports the following .NET data types out-of-the-box.

- *Int16* and *Int16?*
- *Int32* and *Int32?*
- *UInt16* and *UInt16?*
- *UInt32* and *UInt32?*
- *string*
- *decimal* and *decimal?*
- *float* and *float?*
- *double* and *double?*
- *DateTime* and *DateTime?*

See also:

The uJet .NET data type support can be extended by writing custom data type definition mappings and property value converters. For more information on the topic of custom data type definitions and property value converters see *Working with Data Types*.

1.5.3 Property Tracking

When a document, media, or member type is synchronized, uJet tries to match the type declared in code with a type definition. uJet does so for properties too. uJet creates an Umbraco alias for the property, based on the property name, and uses that alias to look for a matching property definition in the database. If a match is found the definition is updated; if not, a new property definition is created. Renaming a property that has been synchronized, in code or using the Umbraco back office, will cause duplicate definitions to be created, with different aliases.

Property tracking refers to the use of the *id* parameter when declaring document type, media type, and member type properties in code. Using the *id* parameter, through *IdAttribute*, uJet can keep track of properties and their corresponding property definitions without relying on the property names. With property tracking, properties can be renamed; uJet will keep your Umbraco database synchronized.

1.6 Working with Data Types

1.6.1 Data Type Definition Mappings

When creating a document type, media type, or member type with properties, the Umbraco data types used are inferred by the .NET data types of the properties declared. A property of type *bool* will be mapped to the data type *True/False* by default, a property of type *string* to the data type *Textstring* and so on.

.NET data types are mapped to Umbraco data types using data type definition mappings (DTDM). The Umbraco data type mapped will determine how Umbraco stores the property value in the database, and what property editor to use for editing the property value in the Umbraco back office.

Built-in Data Type Definition Mappings

The following data type definition mappings are built into uJet. Class names have been shortened.

- *BooleanDTDM*
- *DateTimeDTDM*
- *FloatingBinaryPointDTDM*
- *FloatingDecimalPointDTDM*
- *IntegerDTDM*
- *StringDTDM*

BooleanDTDM

Will map .NET types *bool*, and *bool?* to the Umbraco data type *TrueFalse*.

DateTimeDTDM

Will map .NET types *DateTime*, and *DateTime?* to the Umbraco data type *DatePicker*.

FloatingBinaryPointDTDM

Will map .NET types *float*, *float?*, *double*, and *double?* to the Umbraco data type *Textstring*. Converted using property value converter *FloatingBinaryPointPropertyValueConverter*.

FloatingDecimalPointDTDM

Will map .NET types *decimal*, and *decimal?* to the Umbraco data type *Textstring*. Converted using property value converter *FloatingDecimalPointPropertyValueConverter*.

IntegerDTDM

Will map .NET types *Int16*, *Int16?*, *Int32*, *Int32?*, *UInt16*, *UInt16?*, *UInt32*, and *UInt32?* to the Umbraco data type *Numeric*.

StringDTDM

Will map .NET type *string* to the Umbraco data type *Textstring*.

Custom Data Type Definition Mappings

uJet can easily be extended to support additional .NET types and Umbraco data types. Implement the *IDataTypeDefinitionMapping* interface and add the implementation to the list of data type definition mappings by calling *DataTypeDefinitionMappings.Mappings.Add()*.

1.6.2 Property Value Converters

The Umbraco database schema has it's limitations; e.g. the Umbraco database schema for Microsoft SQL Server supports property values of types *int*, *datetime*, *nvarchar*, and *ntext*. .NET types without a supported SQL Server counterpart must therefore be stored as *nvarchar* or *ntext*. Property value converters (PVC) are used to convert property values stored as *nvarchar* or *ntext* to .NET types on model binding, e.g. *decimal* and *float*.

Property value converters can also be used to add support for custom complex types.

Built-in Property Value Converters

The following property value converters are built into uJet. Class names have been shortened.

- *FloatingBinaryPointPVC*
- *FloatingDecimalPointPVC*
- *HtmlStringPVC*

Custom Property Value Converters

uJet can easily be extended to support additional .NET types. Implement the *IPropertyValueConverter* interface and add the implementation to the list of property value converters by calling *PropertyValueConverters.Converters.Add()*.

1.7 Working with Templates

1.7.1 Template Synchronization

When creating a template through the Umbraco back office, the template markup is saved to a *.cshtml* file in the *Views* folder, and the database.

Copying *.cshtml* files from one Umbraco setup to another is not enough to make the copied templates available in the back office. The database must reflect the contents of the *Views* folder.

uJet supports template synchronization. When your Umbraco application is started, uJet will scan the *Views* folder, looking for *.cshtml* files. Found files will be used as blueprints to synchronize your database. When your database has been updated, the copied templates will be available in the back office.

Note: uJet will not synchronize files with file names that starts with an underscore. E.g. files such as *_layout.cshtml* and *_viewstart.cshtml* will be excluded when synchronizing templates. uJet does not scan subfolders in the *Views* folder.

1.7.2 Preview Template

The preview button in the Umbraco back office does not support documents of types without templates. This is by design in Umbraco.

When developing applications in uJet, using ASP.NET MVC conventions, Umbraco templates are not used. And, as a result, the preview button will be unavailable.

See also:

For more information on the topic of ASP.NET MVC conventions see *Working with ASP.NET MVC Conventions*.

1.8 Working with ASP.NET MVC Conventions

In addition to enabling Code First development, uJet closes the gap between Umbraco and conventional ASP.NET MVC. uJet makes it possible for you to develop conventional ASP.NET MVC sites within Umbraco, while still levering the power of the CMS.

1.8.1 Controllers and Model Binding

With uJet you're no longer bound by the *RenderModel*. Model binding allows uJet to serve your controller action methods with typed document instances. The models for Code First development are used to enable easy, typed access to the documents in Umbraco. These models can also be passed on to your views. Strongly typed views, without the need for view models.

To take advantage of this functionality, have your controllers inherit from *Logikfabrik.Umbraco.Jet.Web.Mvc.JetController* and redeclare your *Index* action methods.

1.8.2 Views

In Umbraco the concept of templates and the concept of views are interchangeable; templates are Razor views saved as *.cshtml* files in the root of the *Views* folder.

When developing with uJet, the ASP.NET MVC naming conventions are supported. Views can be saved in controller subfolders e.g. *Views\Home*, and/or as shared views in the subfolder *Views\Shared*. uJet does so by providing its own implementation of the *IViewEngine* interface.

Views following the ASP.NET MVC naming conventions will not be treated as templates; they will not be available for manual selection in the Umbraco back office. This is by design; views following the ASP.NET MVC naming conventions are not Umbraco templates.

1.9 Archetype

Archetype is a popular property editor for Umbraco 7. With Archetype it possible to create complex properties using existing editors.

The following is a guide to how Archetype can be used with uJet.

1. First, create a new solution in VS.

```
PM> Install-Package uJet
PM> Install-Package UmbracoCms
PM> Install-Package Archetype
```

1. Create your Archetype data type. Pre-values define the Archetype configuration.

```
namespace Example.Models.DataTypes
{
    using System.Collections.Generic;
    using Logikfabrik.Umbraco.Jet;

    [DataType(typeof(string), "Imulus.Archetype")]
    public class MyDataType : IDataType
    {
        public Dictionary<string, string> PreValues => new Dictionary<string, string>
↪{ { "archetypeConfig", @"{
```

(continues on next page)

(continued from previous page)

```
            'showAdvancedOptions':false,
            'startWithAddButton':false,
            'hideFieldsetToolbar':false,
            'enableMultipleFieldsets':false,
            'hideFieldsetControls':false,
            'hidePropertyLabel':false,
            'maxFieldsets':null,
            'enableCollapsing':true,
            'enableCloning':false,
            'enableDisabling':true,
            'enableDeepDatatypeRequests':false,
            'fieldsets':[
            {
                'alias':'myProperty',
                'remove':false,
                'collapse':false,
                'labelTemplate':'',
                'icon':'',
                'label':'My Property',
                'properties':[
                {
                    'alias':'property1',
                    'remove':false,
                    'collapse':true,
                    'label':'Property 1',
                    'helpText':'',
                    'dataTypeGuid':'0cc0eba1-9960-42c9-bf9b-60e150b429ae',
                    'value':'',
                    'aliasIsDirty':true,
                    '$$hashKey':'0RR'
                },
                {
                    'alias':'property2',
                    'remove':false,
                    'collapse':true,
                    'label':'Property 2',
                    'helpText':'',
                    'dataTypeGuid':'0cc0eba1-9960-42c9-bf9b-60e150b429ae',
                    'value':'',
                    'aliasIsDirty':true,
                    '$$hashKey':'0RS'
                }],
                'group':null,
                'aliasIsDirty':true,
                '$$hashKey':'0RI'
                }],
            'fieldsetGroups':[],
            'selection':[]
        }" } };

        public string Property1 { get; set; }

        public string Property2 { get; set; }
    }
}
```

2. Add an Archetype property to your document type; a public property of your data type.

```
namespace Example.Models.DocumentTypes
{
    using DataTypes;
    using Logikfabrik.Umbraco.Jet;

    [DocumentType("My Page")]
    public class MyPage
    {
        public MyDataType MyProperty { get; set; }
    }
}
```

3. Create and register a data type definition mapping for your data type. The data type definition mapping will be used by uJet to map the property MyProperty to your data type.

```
namespace Example
{
    using System;
    using System.Linq;
    using Logikfabrik.Umbraco.Jet.Mappings;
    using Models.DataTypes;
    using Umbraco.Core;
    using Umbraco.Core.Models;

    public class MyDataTypeDataTypeDefinitionMapping : DataTypeDefinitionMapping
    {
        protected override Type[] SupportedTypes => new[] { typeof(MyDataType) };

        public override IDataTypeDefinition GetMappedDefinition(Type fromType)
        {
            return !CanMapToDefinition(fromType) ? null : GetDefinition();
        }

        private static IDataTypeDefinition GetDefinition()
        {
            var definitions = ApplicationContext.Current.Services.DataTypeService.
↪GetDataTypeDefinitionByPropertyEditorAlias("Imulus.Archetype");

            return definitions.First(definition => definition.Name.
↪Equals(typeof(MyDataType).Name));
        }
    }
}
```

```
namespace Example
{
    using Logikfabrik.Umbraco.Jet;
    using Logikfabrik.Umbraco.Jet.Mappings;
    using Models.DataTypes;
    using Umbraco.Core;

    public class MyApplicationHandler : ApplicationHandler
    {
        public override void OnApplicationStarting(UmbracoApplicationBase␣
↪umbracoApplication, ApplicationContext applicationContext)
        {
            DataTypeDefinitionMappingRegistrar.Register<MyDataType>(new␣
↪MyDataTypeDataTypeDefinitionMapping());
```

(continues on next page)

(continued from previous page)

```
        }
    }
}
```

4. Make sure uJet is configured to synchronize data types and document types. Fire up your Umbraco application, create a new document of type MyPage, and edit the value for MyProperty. That's it!

5. This step is optional. If you're using the uJet model binder, create and register a property value converter. The property value converter will be used by uJet to convert the property value into and instance of the data type created in step 1.

```
namespace Example
{
    using System;
    using System.Linq;
    using Archetype.Models;
    using Logikfabrik.Umbraco.Jet.Web.Data.Converters;
    using Models.DataTypes;

    public class MyDataTypePropertyValueConverter : IPropertyValueConverter
    {
        public bool CanConvertValue(string uiHint, Type from, Type to)
        {
            return to == typeof(MyDataType);
        }

        public object Convert(object value, Type to)
        {
            var model = value as ArchetypeModel;

            if (model == null)
            {
                return null;
            }

            var fieldset = model.Fieldsets.First();

            return new MyDataType
            {
                Property1 = fieldset.Properties.First(p => p.Alias.Equals("property1
↪")).Value as string,
                Property2 = fieldset.Properties.First(p => p.Alias.Equals("property2
↪")).Value as string
            };
        }
    }
}
```

```
namespace Example
{
    using Logikfabrik.Umbraco.Jet;
    using Logikfabrik.Umbraco.Jet.Mappings;
    using Logikfabrik.Umbraco.Jet.Web.Data.Converters;
    using Models.DataTypes;
    using Umbraco.Core;
```

(continues on next page)

(continued from previous page)

```
    public class MyApplicationHandler : ApplicationHandler
    {
        public override void OnApplicationStarting(UmbracoApplicationBase␣
↪umbracoApplication, ApplicationContext applicationContext)
        {
            DataTypeDefinitionMappingRegistrar.Register<MyDataType>(new␣
↪MyDataTypeDataTypeDefinitionMapping());
            PropertyValueConverterRegistrar.Register<MyDataType>(new␣
↪MyDataTypePropertyValueConverter());
        }
    }
}
```

```
namespace Example.Controllers
{
    using System.Web.Mvc;
    using Logikfabrik.Umbraco.Jet.Web.Mvc;
    using Models.DocumentTypes;

    public class MyPageController : JetController
    {
        public ActionResult Index(MyPage model)
        {
            return View(model);
        }
    }
}
```

1.10 Logging

uJet logs using the Umbraco wrapper for Apache log4net. uJet logging can easily be configured by editing the logging configuration in *Config\log4net.config*. For more information on how to configure log4net, please refer to the online manual.

The following configuration will write uJet debug logs to a separate file.

```
<log4net>

  <root>
    <priority value="Info"/>
    <appender-ref ref="AsynchronousLog4NetAppender" />
  </root>

  <appender name="rollingFile" type="log4net.Appender.RollingFileAppender">
    <file type="log4net.Util.PatternString" value="App_Data\Logs\UmbracoTraceLog.
↪%property{log4net:HostName}.txt" />
    <lockingModel type="log4net.Appender.FileAppender+MinimalLock" />
    <appendToFile value="true" />
    <rollingStyle value="Date" />
    <maximumFileSize value="5MB" />
    <layout type="log4net.Layout.PatternLayout">
      <conversionPattern value=" %date [P%property{processId}/D%property{appDomainId}/
↪T%thread] %-5level %logger - %message%newline" />
    </layout>
```

(continues on next page)

(continued from previous page)

```
    <encoding value="utf-8" />
  </appender>

  <appender name="AsynchronousLog4NetAppender" type="Umbraco.Core.Logging.
↪ParallelForwardingAppender,Umbraco.Core">
    <appender-ref ref="rollingFile" />
    <filter type="log4net.Filter.LoggerMatchFilter">
      <loggerToMatch value="Logikfabrik.Umbraco.Jet" />
      <acceptOnMatch value="false" />
    </filter>
  </appender>

  <logger name="Logikfabrik.Umbraco.Jet">
    <level value="DEBUG" />
    <appender-ref ref="uJetAsynchronousLog4NetAppender" />
  </logger>

  <appender name="uJetAsynchronousLog4NetAppender" type="Umbraco.Core.Logging.
↪ParallelForwardingAppender,Umbraco.Core">
    <appender-ref ref="uJetRollingFile" />
  </appender>

  <appender name="uJetRollingFile" type="log4net.Appender.RollingFileAppender">
    <file type="log4net.Util.PatternString" value="App_Data\Logs\uJetTraceLog.
↪%property{log4net:HostName}.txt" />
    <lockingModel type="log4net.Appender.FileAppender+MinimalLock" />
    <appendToFile value="true" />
    <rollingStyle value="Date" />
    <maximumFileSize value="5MB" />
    <layout type="log4net.Layout.PatternLayout">
      <conversionPattern value=" %date [P%property{processId}/D%property{appDomainId}/
↪T%thread] %-5level %logger - %message%newline" />
    </layout>
    <encoding value="utf-8" />
  </appender>

</log4net>
```

1.11 License

The MIT License (MIT)

Copyright (c) 2016 anton(at)logikfabrik.se

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION

OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

1.12 Contributions

uJet is Open Source (MIT), and you're welcome to contribute! You'll find the uJet source code on GitHub.

If you have a bug report, feature request, or suggestion, please open a new issue on GitHub. To submit a patch, please send a pull request on GitHub.