Jabberwocky Documentation

Release 1

Chris Smith

October 28, 2016

Contents

CHAPTER 1

Getting Started

1.1 Jabberwocky Overview

The following packages are included as part of the Jabberwocky library suite.

The **core libraries** represent the common stack for developing on all new Sitecore solutions. The **extra libraries** are recommended (but entirely optional), and include specific features that can be included in solutions piecemeal.

The **code analysis** libraries are complements of their respective Core libraries, and provide diagnostics and code fixes for common patterns when using the respective Core libraries. They are highly recommended if you are developing in Visual Studio 2015.

1.1.1 Core Libraries

- Jabberwocky.Core
- Jabberwocky.Autofac
- Jabberwocky.Glass
- Jabberwocky.Glass.Autofac
- Jabberwocky.Glass.Autofac.Mvc
- Jabberwocky.WebApi

1.1.2 Extra Libraries

- Jabberwocky.Autofac.Extras.MiniProfiler

1.1.3 Code Analysis Libraries

- Jabberwocky.Core.CodeAnalysis (for *Jabberwocky.Core*)
- Jabberwocky.Glass.CodeAnalysis (for *Jabberwocky.Glass*)

1.1.4 Dependency Graph

The naming convention for each project attempts to make clear its dependencies.

For instance, the **Jabberwocky.Core** library contains no dependencies, and can be included in any standard .NET project.

Similar to the Core project, the **Jabberwocky.Autofac** project has no external dependencies other than **Jabberwocky.Core** and Autofac, and by extension, its dependencies (ie. Castle.Core). This means that the **Jabberwocky.Autofac** project may be used outside of Sitecore projects, wherever IoC may be desirable.

The **Jabberwocky.WebApi** project also has no dependencies other than on **Jabberwocky.Core**, and WebApi.

The remaining **Jabberwocky.Glass.*** projects all rely on Glass Mapper, with each subsequent package relying on more and more dependencies (Autofac, MVC). These packages are meant to bootstrap and codify Sitecore development with a common set of patterns and practices.

1.2 Quickstart

If you are starting a new project, you will want to download the Jabberwocky packages from the Velir Nuget feed.

If you haven't yet set that up, you'll want to go to Visual Studio -> Tools -> Options -> NuGet Package Manager -> Package Sources

Then you'll want to add 'http://nuget.velir.com/nuget' as a feed (you can also name the feed 'Velir').

If you're starting a new MVC project, you can start by just including the **Jabberwocky.Glass.Autofac.Mvc** package into your website project. If you're not using MVC, you can include the **Jabberwocky.Glass.Autofac** project instead.

You will also want to pull in the **Glass.Mapper.Sc** package. This will allow you to setup your Glass Models and configure the attribute loader in the GlassMapperScCustom.cs file.

Note: For instructions on setting up Glass.Mapper, refer to the guide here.

Notice that in either case, we're going to assume you'll be using the common stack of **Glass.Mapper** and **Autofac**.

The first thing that you'll want to do is create a new folder in the root of your Website project, called 'App_Start'. In this folder, you'll want to create a new class, called **AutofacConfig**.

You can use the following as a template:

```
public class AutofacConfig
{
        public static void Start()
        {
                var builder = new ContainerBuilder();

                builder.RegisterGlassServices();
                builder.RegisterCacheServices();
                builder.RegisterProcessors(Assembly.Load("YOURPROJECT.Library"));
                builder.RegisterGlassMvcServices("YOURPROJECT.Web");
                builder.RegisterGlassFactory("YOURPROJECT.Library");
                builder.RegisterApiControllers(Assembly.GetExecutingAssembly());
                builder.RegisterControllers(Assembly.GetExecutingAssembly());

                // Module registrations
                builder.RegisterModule(new MiniProfilerModule("YOURPROJECT.Library", "YOURPROJECT.Web
                builder.RegisterModule(new LogInjectionModule<ILog>(LogManager.GetLogger);
```

```
                // Custom Registrations

                // This is necessary to 'seal' the container, and make it resolvable from the Autofac
                IContainer container = builder.Build();
                container.RegisterContainer();

                // Create the dependency resolver.
                var resolver = new AutofacWebApiDependencyResolver(container);

                // Configure Web API with the dependency resolver.
                GlobalConfiguration.Configuration.DependencyResolver = resolver;

                // Configure the MVC dependency resolver
                DependencyResolver.SetResolver(new AutofacDependencyResolver(container));
        }
}
```

Note: If you are going to use the `MiniProfilerModule` like in the sample above, you will also need to include the `Jabberwocky.Autofac.Extras.MiniProfiler` nuget package.

CHAPTER 2

Caching

2.1 Overview

Jabberwocky has a built-in caching framework based on the underlying .NET MemoryCache. This makes it extremely portable and flexible - it can be used across any .NET project, and it can be extended to provide additional functionality.

The base implementation provided by the Core library is the `BaseCacheProvider`. This implements the `ICacheProvider` interface, which itself is a composition of two sub-interfaces:

- `ISyncCacheProvider`
- `IAsyncCacheProvider`

These interfaces expose **synchronous** and **asynchronous** caching functions, respectively.

Finally, there are multiple cache classes defined within the Jabberwocky libraries:

- `GeneralCache`
- `SiteCache`

Note: If you plan on using IoC (via Autofac, which is highly recommended for all Sitecore projects), then you should avoid direct references to either of these concrete implementations, and instead rely on the `ICacheProvider` interface instead.

2.2 Usage

You should consider using the `ICacheProvider` service whenever the result of an expensive operation needs to be cached for later use.

Prime candidates include:

- Web service calls
- Sitecore descendants traversal
- *and so much more...*

Attention: It is also recommended to go through the *cache callback documentation* to understand how the callback mechanism works.

2.2.1 Simple Usage

Generally speaking, when working in a Sitecore solution, you should consider using the following overload that accepts a `string` key parameter, and a `Func<T>` callback parameter:

```
var returnVal = _cacheProvider.GetFromCache<object>("key", () => obj);
```

This is the simplest way to cache an item, and will keep the cached item in memory until either:

1. A publish operation occurs, or
2. The .NET runtime is under memory pressure, and the cache item gets evicted automatically

You can also use the **asynchronous** functions to perform the same function, but instead of blocking on the call, you can `await` it instead.

```
var returnVal = await _cacheProvider.GetFromCacheAsync<object>("key", () => obj);
```

or

```
var returnVal = await _cacheProvider.GetFromCacheAsync<object>("key", async ct => await GetResultAsyn
```

2.2.2 Generating a Cache key

When choosing a cache key, you should ensure that it is unique to the object being cached, and that it appropriately varies by any **contextual** values that may affect the result of the callback operation.

For instance, if the object being cached relies on the current **user** (perhaps via `HttpContext.Current.User`), then the cache key should include a variable that uniquely identifies that user.

Note: By convention, the `key` parameter should be scoped to its particular function area, and be formatted with varying parameters, like so: `var key = string.Format("MyClass.MyMethod:{0}:{1}", myParam1, myParam2);`

2.2.3 Advanced Scenarios

You can also use the provided overloads with an absolute expiration.

For the asynchronous overloads, there is also an optional `CancellationToken` parameter that you can pass to the `GetFromCacheAsync` function. If the token is cancelled while the function is being awaited, you can either handle it yourself in the callback, or in the case where the current call is awaiting on the underlying asynchronous lock, the operation will be cancelled.

Be sure to check out the section on Asynchronous Caching for more information on the asynchronous functions, including cancellation.

2.3 Cache Provider

The primary caching provider is exposed through the `ICacheProvider` interface. This implements both the `ISyncCacheProvider` and `IAsyncCacheProvider` interfaces.

The base implementation of the `ICacheProvider` is the `BaseCacheProvider`, which provides an abstract implementation for the interface. This allows developers to extend the base caching facility with their own functionality.

Important: If you intend to extend the base caching implementation, be aware the caching classes are expected to be **thread-safe**, and any descendants must also be **thread-safe**.

For general caching purposes, a `GeneralCache` class has been provided, which inherits from the `BaseCacheProvider` abstract class.

2.3.1 General Cache

The `GeneralCache` uses .NET's MemoryCache under the hood, and is thus suitable for a variety of situations. Notably, it has no dependencies on the HttpContext, or System.Web DLL, so it can be used in any type of application.

2.3.2 Site Cache

The `SiteCache` is a **singleton** implementation of the `BaseCacheProvider`. It is intended to be used for ASP.NET websites, and provides a handler to clear the entire cache.

This is ideal for Sitecore solutions, where it is desirable to clear the site's cache on publish.

Note: If you include the `Jabberwocky.Glass` package in your project, a **Jabberwocky.Glass.config** file will automatically be added to your `App_Config\Include` directory, which will wire up publish event handlers to clear the cache.

In the next section, we will take a look at how the `SiteCache` is extended to work seamlessly with Sitecore out-of-the-box, requiring virtually no setup.

2.3.3 Understanding The Cache Callback

It is important to understand how the caching operation works. The `BaseCacheProvider` implements the reusable caching logic, and so all inheritors also inherit this logic.

When caching an object via one of the caching functions, the cache will attempt to locate the object in the underlying `MemoryCache` using the provided cache `key` parameter. If it is found, the cached object is immediately returned, and no further processing is required.

On the other hand, if the object is not found, then the cache will be required to execute the cache callback to calculate the value to be cached. Before this happens, the cache will enter a **critical region** (by locking on an a temporary cache object, discriminated by the unique cache `key` parameter).

Important: This is why it is important to create **correct** cache keys. These keys act as mutually exclusive locks, so that only a single cache `callback` can be executed at the same time, per unique `key`.

By wrapping all invocations of the cache `callback` in a critical region, we ensure that only a single thread can execute the callback at a time, thus causing all other concurrent requests for the same cache `key` to block.

In most cases, this ends up saving time and resources, as this prevents the expensive `callback` operation from being called more than once. Intead, all blocking threads will receive the cached result once the initial `callback` operation completes.

2.4 Sitecore Integration

When the `Jabberwocky.Glass` package is imported, you'll have access to the `ContainerBuilder` extension method: `builder.RegisterCacheServices()`.

Note: If you followed the Quickstart, then you should already have this registered.

When cache services have been registered with Autofac by calling the extension method above, then the `ICacheProvider` (and `ISyncCacheProvider`/`IAsyncCacheProvider`) will have been registered for you.

Included as part of this registration is a decorator that will wrap the underlying cache provider, and provide Sitecore specific behavior. By default, the underlying cache provider is the `SiteCache`.

This behavior is provided by the `SitecoreCacheDecorator` class, and it ensures that all cache calls works seamlessly with Sitecore, by providing the following services:

1. **Cache keys will automatically vary by Sitecore Context parameters:**
    - Language
    - Database
    - Site Name
2. **Caching will only occur in the `web` database context**
    - The cache acts as a 'no-op' when in the `master` context.
3. The Cache will automatically clear on publish

2.5 Asynchronous Caching

2.5.1 Overview

The `IAsyncCacheProvider` interface exposes the asynchronous caching functions.

Unlike their synchronous counterparts found in the `ISyncCacheProvider`, these functions all follow the Task-based Asynchronous Pattern (TAP), are are thus `Task<T>` returning.

Effectively, this means that these functions are all **non-blocking**, and will all execute asynchronously. Furthermore, this means that each of the asynchronous functions can be **awaited** with the `await` keyword.

The benefits to using the asynchronous functions become clear in situations where you may have long-running IO operations (like network calls), or are already executing code in an asynchronous context, and want to prevent blocking on synchronous locks.

Thus, by using the asynchronous caching functions, you gain the benefit of:

1. Non-blocking asynchronous locking
2. Asynchronous cache callback execution

The first point is important to note, because as outlined in the *callback documentation*, when resorting to executing the `callback`, the cache will enter a **critical region**. In an asynchronous execution context, it is important to eliminate blocking calls, as these can (potentially) lead to deadlocks.

Thus, the asynchronous caching functions use an asynchronous locking primitive that does **not** block the thread.

2.5.2 GetFromCacheAsync

There are a few variations on the asynchronous caching functions (ignoring the overloads with expiration):

```
Task<T> GetFromCacheAsync<T>(string key, Func<T> callback,
        CancellationToken token = default(CancellationToken))

Task<T> GetFromCacheAsync<T>(string key, Func<CancellationToken, Task<T>> callback,
        CancellationToken token = default(CancellationToken))
```

Regardless of which function is used, they are all asynchronous, and will return a `Task<T>`. This allows you to `await` the result of the call asynchronously.

There are two variations that allow you to specify either a synchronous cache `callback`, or an asynchronous one. While it may seem counter-intuitive to include an asynchronous overload that allows for executing a synchronous callback, the reason for this is to allow you to execute expensive synchronous callbacks, while still allowing you to asynchronously await the caching function.

The primary benefit of doing this is that the **critical region** of the asynchronous function still uses an asynchronous lock, and thus it is desirable to use the asynchronous overload when caching a synchronous callback, so that multiple threads attempting to retrieve the same cached value don't **block**.

2.5.3 Cancellation

All of the provided overloads accept an optional `CancellationToken`. By passing in a vaild token, you can cancel an ongoing cache operation. There are two points where a caching operation can be cancelled:

1. When the function attempts to execute the cache `callback`, and enters the **critical region**, or
2. When the function has already entered the **critical region**, and is currently executing the `callback`.

In the first case, if the `CancellationToken` is cancelled while the current thread is awaiting on the asynchronous lock, the function will throw.

In the second case, it is up to the implementor of the `callback` fuction to appropriately handle cancellation, as the token is passed into the `callback` as the sole parameter.

CHAPTER 3

Glass Interface Factory

3.1 Overview

3.1.1 Introduction

The Glass Interface Factory is a re-imagining of the (Custom) Item Interface Factory.

It is built to work with **Glass Mapper** items, and is intended to be used with auto-generated glass templates (ie. with **TDS**).

In essence, it provides services that can be thought of as codifying both the **Adapter** and (more loosely) **Dynamic/Multi-Dispatch** patterns into a single easy to use framework.

What this means in practice is that you can now easily encapsulate business logic that should apply to specific Sitecore templates within common **interfaces**. With those defined, you can then create implementations of those interfaces for a given Sitecore template. This defines the **adapter pattern** part.

Note: The **interfaces** and **implementations** of those interfaces referred to above, are called the **Glass Factory Interface** and **Glass Factory Type** respectively.

The real magic occurs when you consider the Sitecore template hierarchy. Using a form of **dynamic dispatch**, we can create faux object-inheritance hierarchies between implementations of these **interfaces**.

Because each **Glass Factory Type** is an adapter for a particular Sitecore template, when a particular function is not implemented for a particular **Glass Factory Type**, we can search the base template hierarchy for that particular template to see if there exists another **Glass Factory Type** that both implements the specified **Glass Factory Interface**, and acts an adapter for a base Sitecore template of the original template.

This allows us to write code once and apply common behavior to base Sitecore templates, while letting us also override said behavior as needed on more derived Sitecore templates.

You can find more information on how this works in the Implementation Guide section.

3.1.2 Implementations

There are two primary implementations of the Glass Interface Factory to allow for use both with, and without Autofac.

In both cases, a factory **builder** is provided for ease of setup:

- Without Autofac: `DefaultGlassFactoryBuilder`
- With Autofac: `AutofacGlassFactoryBuilder`

Note: If you followed the quickstart guide, then you don't need to use any builders directly. Instead, the `builder.RegisterGlassFactory()` extension will automatically register the `IGlassInterfaceFactory` service for you, and will handle construction of the factory.

The next section will cover setting up the factory.

3.2 Initial Setup

Setting up the Glass Interface Factory is both easy and straightforward. Depending on your needs, you may use it with, or without Autofac.

3.2.1 With Autofac

If you're using Autofac, then you don't need to do anything other than register the factory via the `builder.RegisterGlassFactory()` extension method.

Then in your code, you should be able to use constructor injection to request the `IGlassInterfaceFactory` interface as a dependency, like so:

```
private readonly IGlassInterfaceFactory _factory;
public MyService(IGlassInterfaceFactory factory) {
        _factory = factory;
}
```

3.2.2 Without Autofac

If for any reason you can't use Autofac, you can manually construct the factory using the provided `DefaultGlassFactoryBuilder`:

```
var options = new ConfigurationOptions(debugEnabled: false, assemblies: "YOURPROJECT.Library");
var sitecoreServiceFunc = () => new SitecoreContext() ?? new SitecoreService("web");
_builder = new DefaultGlassFactoryBuilder(options, sitecoreServiceFunc);

var factory = _builder.BuildFactory();
```

Important: It is important that you also store the result of the `_builder.BuildFactory()` as a singleton, as the `GlassInterfaceFactory` is **thread-safe**, and the process of creating the factory is expensive.

Attention: The above code is provided as a sample only. The `sitecoreServiceFunc` will return the current context's glass version of the Sitecore.Context. However, this will only work within the context of an HTTP request, so be aware that this may not work when in the context of a Sitecore pipeline, or other non-HTTP context. In those cases, the above func will return use the **web** database.

3.2.3 Configuration Options

Regardless of whether or not you use Autofac, you can configure the setup of the `GlassInterfaceFactory` to specify:

- Enable/Disable Debug mode
- Which assemblies to scan for `GlassFactoryInterface` and `GlassFactoryType` declarations.

With Autofac, that might look something like:

```
var options = new ConfigurationOptions(debugEnabled: true, assemblies: "YOURLIBRARY");
builder.RegisterGlassFactory(options);
```

The **debug** flag is useful for catching errors during development. It enables behavior that will throw a `TypeMismatchException` when the Glass Interface Factory attempts to convert a Glass Mapper item to its corresponding Glass Factory Type implementation, and it detects that the runtime type of the Glass Model is incompatible with the Glass Factory Type's expected Glass Model Type.

This is usually the result of out-of-sync TDS models, and/or incorrectly passing in Glass Mapper models to the `GetItem<T>` function without using the `SitecoreService.GetItem<IGlassBase>(inferType: true)` overload.

3.3 Implementation Guide

There are two pieces to implement when using the **Glass Interface Factory**:

- Glass Factory **Interface**
- Glass Factory **Type**

The **Glass Factory Interface** defines the adapter contract (like any other interface), and the **Glass Factory Type** implements the interface, and binds it to a specific Glass Mapper model type.

This is accomplished by using two attributes:

- `GlassFactoryInterfaceAttribute`
- `GlassFactoryTypeAttribute`

You can apply these attributes to an **interface** and a **class** for each of the `GlassFactoryInterfaceAttribute` and `GlassFactoryTypeAttribute` respectively.

Check out the respective pages below for instructions on how to implement and use each of these types:

3.3.1 Glass Factory Interface

The **Glass Factory Interface** should encapsulate common business functionality that should have potential applications across multilple Sitecore templates.

For instance, the canonical example would be an interface to adapt Sitecore items as listable items, usable within search listings across the site.

We might name this interface `IListable`, and we could define it like so:

```
[GlassFactoryInterface]
public interface IListable
{
        string ListTitle { get; }
        string Url { get; }
        string Topic { get; }
        string DisplayDate { get; }
        string Author { get; }
        string ListImage { get; }
}
```

The goal here is to encapsulate all of the information that we might need in order to display an arbitrary Sitecore item within a listing component on the site.

Important: Notice that the interface is decorated with the `[GlassFactoryInterface]` attribute. This is required in order to use this interface with the Glass Interface Factory.

We can include arbitrary functions or properties in the interface declaration, however convention dictates that we make all properties **get-only**. It is also a best-practice to ensure that all exposed functions and properties have no side-effects - ie. they are pure functions.

In the next section, we'll examine how to implement this functionality, and bind it to specific Sitecore templates.

3.3.2 Glass Factory Type

The **Glass Factory Type** should implement the specific functionality defined in one or more **Glass Factory Interfaces**.

Requirements

In order to create a **Glass Factory Type**, the implementing class must satisfy the following criteria:

1. Be marked `abstract`
2. Be decorated with the `GlassFactoryTypeAttriute`
3. Specify a valid Glass Mapper type in the `GlassFactoryTypeAttribute`'s parameter

Note: You can also optionally inherit from the `BaseInterface<T>` abstract class. Doing so allows you to access the underlying Glass Model via a `InnerItem` property.

Example

Continuing on from the `IListable` example in the previous section, let's assume we have the following template hierarchy in Sitecore (indented lines represent nested base templates):

1. **Article Page Template**
    - Global Page Template
2. **Blog Post Template**
    - Global Page Template

With such a template hierarchy, we can start to implement our `IListable` interface for any of the given templates.

Here's an example of what our implementation of `IListable` might look like as it pertains to the **Article Page Template**.

```
[GlassFactoryType(typeof(IArticlePage))]
public abstract class ArticlePageModel : BaseInterface<IArticlePage>, IListable
{
        public ArticlePageModel(IArticlePage model) : base(model)
        {
        }

        public abstract string ListTitle { get; }
```

```
        public abstract string Url { get; }
        public abstract string Topic { get; }
        public abstract string DisplayDate { get; }

        public string Author
        {
                get { return InnerItem.Authors; }
        }

        public abstract string ListImage { get; }
}
```

Notice that the `GlassFactoryTypeAttribute` has the `typeof(IArticlePage)` parameter defined. This binds the implementation to that particular Sitecore template.

Important: If you elect to inherit from the (recommended) `BaseInterface<T>`, then you must ensure that the generic type param `T` matches the parameter in the `GlassFactoryTypeAttribute`.

What's important to note here is that the class is marked `abstract`, and you only need implement the specific functionality that may differ from the base functionality. In this case, only the `Author` property has been implemented (by returning the underlying `IArticlePage`'s `Authors` field value.)

All of the other properties have been marked as `abstract`, and by doing this it is assumed that one of the underlying base-templates of the **Article Page Template** will implement this functionality.

Fall-back Behavior

For any given property or function that is not implemented for a given **Glass Factory Type**, it is assumed that the implementation must lie in another **Glass Factory Type** that is bound to a base template. When this is the case, the Glass Interface Factory will dynamically dispatch calls to these 'unimplemented' functions to base-template implementations.

This behavior is referred to as **fall-back** behavior, and is a powerful feature of the **Glass Interface Factory**.

With this feature, we can write common units of business logic, and apply those to common base templates in Sitecore. Whenever a specific template that inherits from these base templates needs different logic, we can simply change the logic in the corresponding **Glass Factory Type** implementation, without affecting any of the base template logic. Any functions that don't require special logic need not change, since by marking them as `abstract`, we automatically gain the ability to inherit the functionality from existing base template implementations.

Given the `ArticlePageModel` definition above, if we were to define an implementation of `IListable` for the **Global Page Template**, then all of the functions marked as `abstract` in the `ArticlePageModel` would **fall-back** to the implementation in the `GlobalPageModel`:

```
[GlassFactoryType(typeof(IGlobalPage))]
public abstract class GlobalPageModel : BaseInterface<IGlobalPage>, IListable
{
        public GlobalPageModel(IArticlePage model) : base(model)
        {
        }

        public string ListTitle => InnerItem.Title;
        public string string Url => InnerItem.Url;
        public string Topic => InnerItem.Topic;
        public string DisplayDate => InnerItem.PublishDate.ToString();
```

```
        public abstract string Author { get; }

        public string ListImage => InnerItem.ThumnailImage.Url;
}
```

Note: If a property or function has no base implementation (because all implementors have marked it as `abstract`), then the return value will simply be `null`.

3.4 Usage

Once you have your **Glass Factory Interfaces** and **Glass Factory Types** defined and implemented, then you are ready to use the `IGlassInterfaceFactory` in your code:

```
public IListable GetIListable(IGlassInterfaceFactory factory,
        ISitecoreContext context) {

        var contextItem = context.GetItem<IGlassBase>(inferType: true);
        IListable listable = factory.GetItem<IListable>(contextItem);

        return listable;
}
```

Important: Note that we use `inferType: true` when getting the current context item via Glass Mapper's `ISitecoreContext` service. This ensures that the runtime type of the returned `contextItem` variable is set to the actual Glass Mapper model that matches the current item's Sitecore template.

In the example above, we assume that an `IListable` **Glass Factory Interface** has been defined, and that the current context item in Sitecore has a corresponding **Glass Factory Type** implementation. If not, the `factory.GetItem<IListable>` will return null.

CHAPTER 4

Mini Profiler

4.1 Overview

A MiniProfiler integration has been included in the `Jabberwocky.Autofac.Extras.MiniProfiler` nuget package.

By installing this package, you will be provided with deep profiling telemetry of all of your code. It provides timing information for each of your classes' methods, allowing you to examine where the most time is spent in your code.

Depending on how you set this up, you can choose to have this enabled only on certain environments, or even configure it to switch on and off on Production environments to troubleshoot performance issues.

4.2 Module Setup

This integration takes the form of an Autofac Module, and can be installed with a single line:

```
builder.RegisterModule(new MiniProfilerModule("YOURPROJECT.Library", "YOURPROJECT.Web"));
```

Note: Be sure that you have included the `Jabberwocky.Autofac.Extras.MiniProfiler` nuget package in your Website project, otherwise you won't have access to the module.

As an example, you can consider conditionally enabling/disabling the module based on the DEBUG compiler flag:

```
#if DEBUG
                builder.RegisterModule(new MiniProfilerModule("YOURPROJECT.Web"));
#endif
```

4.3 Configuration

There are multiple configuration options for the MiniProfiler integration. First and foremost, you must specify which assemblies to instrument. The MiniProfiler module will only provide profiling information for types found in those assemblies.

Important: If you are also using the Glass Interface Factory, and wish to profile the Glass Interface Types, then you should use the MiniProfileModule's constructor that accepts an array of `IProxyStrategy` types, and pass in a new

instance of the following type: `GlassInterfaceFactoryStrategy`.

The available configuration options (in reverse order of precendence):

1. **assemblies**: Specifies which assemblies to instrument
2. **includeNamespaces**: Specifies root namespaces to instrument types from (filtered by assemblies)
3. **excludeNamespaces**: Excludes specific root namespaces from instrumentation
4. **excludeTypes**: Excludes specific types from instrumentation
5. **excludeAssemblies**: Excludes specific assemblies from instrumentation
6. **strategies**: Allows for specifying new strategies for generating proxies for instrumentation

The `strategies` parameter is provided in case you ever need to extend the proxy implementation. As with the `GlassInterfaceFactoryStrategy`, if there is a specific use case where you need to change the proxy behavior, you can do so by creating your own strategy.

Note: The order of the strategies matters, as the selection algorithm will use the first strategy that returns true when `CanHandle` is called.

4.4 MiniProfiler Setup

To configure MiniProfiler itself, you will need to follow the instructions here

In short, on your primary base layout, add the following directive (**RenderIncludes**):

```
@using StackExchange.Profiling;
<head>
 ..
</head>
<body>
  ...
  @MiniProfiler.RenderIncludes()
</body>
```

Then add the following to your Global.asax(.cs):

```
#if DEBUG
        protected void Application_BeginRequest()
        {
                if (Sitecore.Context.Item != null && !Sitecore.Context.PageMode.IsPageEditorEditing
                        && !Sitecore.Context.PageMode.IsPreview)
                {
                        MiniProfiler.Start();
                }
        }

        protected void Application_EndRequest()
        {
                MiniProfiler.Stop();
        }
#endif
```

Like in the example, you can optionally choose to include the conditional preprocessor directives to only enable profiling when the `DEBUG` compiler flag is set to `true`.

CHAPTER 5

Inversion of Control

5.1 Overview

Inversion of Control (IoC) is a software design pattern that inverts the resposibility of the ownership and creation of dependencies. It advocates for removing such direct dependencies from user code, and pushing the flow of control out into a separate library or process.

Jabberwocky has a tight integration with **Autofac**, a library that enables Dependency Injection (DI, a specific form of IoC).

In order to improve upon the experience when working within these frameworks, Jabberwocky provides a few extensions and services to greatly ease the burden of development.

5.2 Autowired Services

One of the tedious parts of using DI frameworks is in registering your dependencies.

Consider the following code, which is standard for most DI Containers (like Autofac):

```
var builder = new ContainerBuilder();
builder.RegisterType<NavigationBuilder>().As<INavigationBuilder>().InstancePerLifetimeScope();
builder.RegisterType<EmailService>().As<IEmailService>().InstancePerLifetimeScope();
builder.RegisterType<AccountNotificationService>().As<INotificationService>().InstancePerLifetimeScop
```

We can reduce the amount of boilerplate code required (like above) by using the provided `AutowireServiceAttribute`.

You can decorate your concrete classes with this attribute, and you can replace all of the `builder.RegisterType<>` calls with a single line:

```
builder.AutowireServices("YOUR.Library");
```

What this does is automatically scan the provided assemblies, and for each type that is decorated with `AutowireServiceAttribute`, Jabberwocky with automatically register that type as each of its implemented interfaces.

So given a service with two interfaces, decorated with the `AutowireServiceAttribute`:

```
[AutowireService]
public class MyService : IMyService, IAnotherService
{
```

```
        // Implementation elided
}
```

This would be equivalent to registering the following manually:

```
builder.RegisterType<MyService>().As<IMyService>().As<IAnotherService>();
```

5.3 Configuration

The `AutowireServiceAttribute` class has a few optional parameters that can be supplied to it:

- **Scope**: This represents the Lifetime Scope of the registration
- **IsAggregateService**: Whether or not the registration is for an *Aggregate Service*

Scope can have one of the following assignments:

- **Default**: This has transient behavior; ie. one instance per dependency
- **PerScope**: An instance per parent resolution scope
- **PerRequest**: For web-based scenarios; instance per web request
- **NoTracking**: Externally owned; will not be tracked by the container
- **SingleInstance**: Singleton semantics

For more information on lifetime scopes in Autofac, take a look at the documentation here.

IsAggregateService is a special case, and should be used on **interface** definitions. It registers that a given interface as an **aggregate service**, which you can read more about here.

CHAPTER 6

MVC Integration

6.1 Overview

Sitecore MVC provides many implementation options, and initially it can difficult to choose a path among all of them:

- Controller Renderings
- View Renderings
- Item Renderings
- etc.

To make matters a little more confusing, Sitecore MVC is a different beast than vanilla ASP.NET MVC. In the former, routes aren't handled by the controller naming convention like in ASP.NET MVC, but instead are handled by Sitecore's regular routing methodology - namely, based on an item path in Sitecore.

In order to help accelerate MVC development in Sitecore, we've developed a standard set of practices that you can incorporate into your own projects.

6.2 Best Practices

For simplicity's sake, and as a general rule of thumb, we recommend developing primarily against View Renderings when using Sitecore MVC.

The reason for this is that we can rely on a few utilities in the Jabberwocky libraries to greatly reduce the amount of boilerplate we need to write in order to create a rendering. With a View Rendering, all we need to do is create a **View** (razor cshtml file), and a **ViewModel** (a class that inherits from `GlassViewModel<T>`). The actual binding of the model to the view (which normally occurs in the Controller code) happens automatically in the case of Jabberwocky's `GlassViewModel<T>`.

Contrast this to **Controller Renderings**, which are much more aligned with standard MVC, but require the addition of a Controller class. The downside to this approach is that you have to manually construct a model, populate it's values, and pass it off to the view itself, which becomes tedious and time consuming with the more components you have. Therefore, the primary reason to use a Controller Rendering over a View Rendering would be in situations requiring handling of post-backs (or GET-requests with complicated query string parsing). Otherwise, for simple components, View Renderings are the way to go.

On the topic of **ViewModels**, the approach we are recommending is more in line with Model-View-Presenter (MVP), or more loosely, Model-View-ViewModel (MVVM).

When using the **View Rendering** approach to developing components for Sitecore MVC, using a **ViewModel** over a straight up Glass Mapper model offers many benefits, the biggest of which is probably that you can choose to create a

ViewModel that composes *multiple* Glass Mapper models. This makes it easier to develop a component that has *more* than one data source. Furthermore, using a ViewModel as opposed to the direct Glass Mapper model allows you to implement View-specific logic within your ViewModel – logic which might not make sense to put within the Glass Mapper model itself.

6.3 View Renderings

6.3.1 Setup

Getting started with **View Renderings** is easy. All you need to do is register the `RegisterGlassMvcServices` extension method with Autofac, and create a few Views and ViewModels.

Registering the services with Autofac is as simple as:

```
builder.RegisterGlassMvcServices("YOURPROJECT.Web");
```

Note: If you followed the instructions in the Quickstart, then you should already have this extension method registered, and the appropriate config file in your App_Config/Include directory.

6.3.2 Creating a View

Creating a View is as easy as adding a new Razor View to your web project. You should then use the `@inherits` directive to specify the base type for the View. We recommend using the `CustomGlassView<T>` like so:

```
@inherits Jabberwocky.Glass.Autofac.Mvc.Views.CustomGlassView<MyViewModel>
```

The generic type parameter is the type of your ViewModel.

6.3.3 Creating a ViewModel

A ViewModel is simply a class that inherits from the `GlassViewModel<T>` base class. The generic type parameter for the `GlassViewModel<T>` is the type of Glass Mapper model to use as the `GlassModel` property. This allows you to, from your View, access your ViewModel from the `Model` property, or your Glass Mapper model from your `Model.GlassModel` property.

If you don't have a need for a particular Glass Mapper model in your View, but you still want to use a ViewModel, you can do so by specifying the generic type parameter as `IGlassBase`, which all Sitecore items should be assignment compatible with.

Because your View is now decoupled from your Glass Mapper model, and instead is using your custom ViewModel, you can also elect to use constructor injection within your ViewModel. This allows you to pull in arbitrary dependencies in your ViewModel to write logic that dictates specific behavior for your View, without having to resort to polluting your Glass Mapper models (or creating one-off extension methods) with view-specific logic.

In the next section, we'll look at some of the advanced options available to you when using the Jabberwocky View Model pattern.

6.4 The View Model

There are three configurable components to the ViewModel:

1. The **DataSource**
2. The **Rendering Parameters**
3. The **nested datasource strategy**

6.4.1 DataSource

The `GlassViewModel<TDatasource>` type is used for creating a view-model that expects a strongly typed datasource. This datasource should map to any datasource template type defined on the View Rendering in Sitecore itself.

In the case where no datasource is specified, the type `IGlassBase` can be used, as all glass templates should inherit from this base type.

You can access the underlying Glass datasource model from the `GlassModel` property exposed on the `GlassViewModel<TDatasource>` type.

Important: Attempting to use the `GlassModel` property from within the constructor will result in a `NullReferenceException`. If you need access to the underlying datasource from within the constructor, use **constructor injection** instead, by including a constructor parameter of type `TDatasource`.

6.4.2 Rendering Parameters

Rendering Parameters provide a nice way to define another set of properties that can be used to configure a rendering. These are defined separately in Sitecore via **Rendering Parameter Templates**, and are then assigned to View Renderings, independently of Datasource Templates.

You can use Rendering Parameters in a strongly-typed fashion by using the generic type of `GlassViewModel<TDatasource, TRenderingParameter>`, which defines **two generic parameters**: the first, for the datasource template, and the second for the type of rendering parameters.

In this manner, you can access the rendering parameters of the rendering via the `RenderingParameters` property.

Important: As called out above in the Datasource section, if you need to access the **rendering parameters** from within the constructor, you **cannot** use the `RenderingParameters` property. Instead, you can include a constructor argument of type `TRenderingParameters`, which will be injected for you automatically.

6.4.3 Nested Datasource Strategy

Sitecore 8 introduced the concept of **nested datasources**. Previously, if a rendering did not have a datasource specified, the `Datasource` property on that rendering would resolve to the current `Context.Item`. However, with Sitecore 8+, there is a new setting (enabled by default) that resolves a rendering's datasource to any parent rendering's datasource:

```
<setting name="Mvc.AllowDataSourceNesting" value="true"/>
```

The resolution logic now looks like the following:

1. Explicit DataSource specified (in the Datasource field)
2. Nested DataSource (specified by the Rendering.Item property, and inherited from any parents)

3. Sitecore Context Item (page item)

Step 2 may be unexpected for many people used to pre-8.0 resolution logic (steps 1 & 3), as this was the new step added in Sitecore 8+. To solve this, we have introduced the following attributes, each of which may be applied directly to a user-defined GlassViewModel.

- DisableNestedDatasource
- AllowNestedDatasource
- ConfigureDatasource

The third attribute (ConfigureDatasource) exposes a constructor param that dictates whether or not to use nested datasource resolution. The first two attributes are simply extensions of the ConfigureDatasource attribute, and are provided as shortcuts for their particular behavior.

Note: The absence of any of these attributes indicates that the default Sitecore datasource resolution strategy should be used - this will be dictated by the value in the `Mvc.AllowDataSourceNesting` setting.

You can use these attributes on ViewModels like so:

```
[DisableNestedDatasource]
public class NeverFallbackViewModel : GlassViewModel<IGlassBase>
{
    // PRE-Sitecore 8 behavior; ignores Mvc.AllowDataSourceNesting value

    // The datasource may come from the direct datasource, or Context Item
    // It will never be from the nested parent datasource
}
```

or

```
[AllowNestedDatasource]
public class AlwaysFallbackViewModel: GlassViewModel<IGlassBase>
{
    // POST-Sitecore 8+ behavior; ignores Mvc.AllowDataSourceNesting value

    // The datasource may come from the direct datasource, or nested parent datasource
    // Or finally the Context Item
}
```

CHAPTER 7

Packages

7.1 Jabberwocky.Core

This library is the root of all Jabberwocky packages. It has no third-party dependencies, and so can be included in any project type, Sitecore or otherwise.

It mostly contains useful utilities and services that end up being reused a lot across projects.