# Introduction to the Sitecore ItemBinding Framework v. 2.4

Bo Breiting: 28-12-2015

This document covers the various elements of the Sitecore ItemBinding framework and their intended usage.

## Model classes

The item binding framework revolves around model classes. Basically any class can be used as the basis for a model class the only requirement is that the class has to have a constructor that accepts a single Sitecore Item parameter. This is however not a technical limitation in the framework and the framework can easily be extended to work without this requirement. The behaviour can be changed centrally in the framework by modifying the model repository or locally by introducing a local model factory. However since the framework is called Item Binding it makes good sense to require that the model classes should accept an item that is bound to the model.

### ItemBoundModel base class

The framework contains an abstract ItemBoundModel class that can be inherited to the concrete model classes for ease of use. The ItemBoundModel contains a constructor that accepts a single Sitecore Item parameter that it exposes through the public property InnerItem. This makes it easy to access the bound item in a concrete model class when inheriting from the ItemBoundModel class.

ItemModel definition:

using Sitecore.Data.Items;

namespace ItemBinding.Model

{

  public class ItemBoundModel

  {

    public ItemBoundModel(Item item)

    {

      \_innerItem = item;

    }

    public Item InnerItem

    {

      get { return \_innerItem; }

    }

    private readonly Item \_innerItem;

  }

}

### Concrete model class

As previously mentioned practically any class can form the basis of a model class. However it is recommended to inherit from the ItemBoundModel abstract class to expose the bound item in a standardized way.

Concrete model class example:

using System;

using Sitecore.Data;

using Sitecore.Data.Items;

namespace ItemBinding.Model

{

  public class Article : ItemBoundModel

  {

    public Article(Item item) : base(item)

    {

    }

    public readonly ID TitleFieldId = new ID("{93CA1A43-6D82-4A3E-BEA7-902AE99A3618}");

    public String Title

    {

// Usage of the InnerItem property exposed by the ItemBoundModel base class

      get { return InnerItem[TitleFieldId]; }

    }

  }

}

The example class above inherits from the ItemBoundModel which means that the InnerItem property of the ItemBoundModel is available. Apart from that a model class can expose any set of properties and methods both computed as is the case with the Title property or for instance as raw values that expose the Sitecore field IDs of the bound item as is the case with the TitleFieldId property.

The model class is not required to expose the Sitecore item or any other Sitecore specific details and if direct access to the item, the field IDs or field values are not required or desired these can be omitted. In fact a model class could simply be a class object with auto assigned properties and a custom factory could be used to create the model object and populate the properties (see the following section about custom factories). The model class could even be based on a foreign model class from another framework like Glass or the custom item generator from Velir.

### Modelling Sitecore template inheritance

In some cases it may be necessary to model a Sitecore item that inherits from multiple base templates. There are different ways to achieve this but by default it is not recommended to construct model classes that combine multiple Sitecore templates into a single model class. Previous experience indicates that using a single model class to represent multiple Sitecore templates can lead to unintentional mixed responsibility and to Sitecore base template information being duplicated across multiple model classes simply because the template is inherited by other templates that are then represented as model classes in the solution. Because of these considerations the framework recommends a strict one template per model class restriction. Instead multiple model classes should be linked together either through inheritance or composition to represent Sitecore template inheritance.

#### Modelling through inheritance

A good case for modelling through inheritance is presented with an article template that is derived into multiple specialized article templates such as a video-article template and an image-article template. In this case it is recommended to create an Article model class that contains all the information pertaining to the article template and create additional specialized VideoArticle and ImageArticle model classes that contains the information related to the Video and Image templates and then link the Article model classes to the VideoArticle and ImageArticle model classes through class inheritance.

Class inheritance example:

using System;

using Sitecore.Data;

using Sitecore.Data.Items;

namespace ItemBinding.Model

{

// Class modelling the Sitecore Article template

  public class Article : ItemBoundModel

  {

    public Article(Item item) : base(item)

    {

    }

    public readonly ID TitleFieldId = new ID("{93CA1A43-6D82-4A3E-BEA7-902AE99A3618}");

  }

// Class modelling the Sitecore Video template inheriting from the Article template

// The inherited Article template is represented by inheriting from the Article class

  public class VideoArticle : Article

  {

    public VideoArticle(Item item) : base(item)

    {

    }

    public readonly ID VideoFieldId = new ID("{0B19E512-49EC-4D7F-8926-FF090F4D2CDF}");

  }

// Class modelling the Sitecore Image template inheriting from the Article template

// The inherited Article template is represented by inheriting from the Article class

  public class ImageArticle : Article

  {

    public ImageArticle(Item item) : base(item)

    {

    }

    public readonly ID ImageFieldId = new ID("{86601334-7906-47F1-8135-2C01825AB07A}");

  }

}

#### Modelling through composition

In some cases an item will inherit from multiple templates without a clear sub/super class chain of inheritance. This could be the case with items inheriting from general base templates that are used throughout the solution or from multiple unrelated base templates. In this case template inheritance can be modelled through the use of composition where one model class contains and exposes other model classes as properties.

Composition inheritance example:

using System;

using System.Collections;

using System.Collections.Generic;

using Sitecore.Data;

using Sitecore.Data.Items;

namespace ItemBinding.Model

{

// Class modelling a base keywords Sitecore template

 public class Keywords : ItemBoundModel

  {

    public Keywords(Item item) : base(item)

    {

    }

    public IEnumerable<String> SelectedKeywords { get; set; }

  }

// Class modelling an article Sitecore template

  public class Article : ItemBoundModel

  {

    public Article(Item item) : base(item)

    {

    }

    public readonly ID TitleFieldId = new ID("{93CA1A43-6D82-4A3E-BEA7-902AE99A3618}");

  }

// The VideoArticleWithKeywords template inherits both from the Article template and

// the Keywords template.

// The Article template inheritance is expressed through a normal class inheritance

// and the Keywords template inheritance is expressed as a composition where a

// Keywords class object is bound to the Keywords property of the

// VideoArticleWithKeywords class

 public class VideoArticleWithKeywords : Article // class inheritance

  {

    public VideoArticleWithKeywords(Item item) : base(item)

    {

      Keywords = new Keywords(item);

    }

    public readonly ID VideoFieldId = new ID("{0B19E512-49EC-4D7F-8926-FF090F4D2CDF}");

    public Keywords Keywords { get; private set; } // composition

  }

}

In the example above the VideoArticleWithKeywords model inherits from the Article model and is linked to the Keywords model through composition which means that it exposes an instance of the Keywords model that is instantiated when the VideoArticleWithKeywords model is instantiated. Of course there are many variations on this pattern. For instance the Kewords model instance could be private and the VideoArticleWithKeywords model could expose the properties of the contained Keywords model through public properties as is shown in the next example.

Example of private composition:

public class VideoArticleWithKeywords : Article // class inheritance

{

  public VideoArticleWithKeywords(Item item) : base(item)

  {

    \_keywords = new Keywords(item);

  }

  public readonly ID VideoFieldId = new ID("{0B19E512-49EC-4D7F-8926-FF090F4D2CDF}");

  public IEnumerable<String> SelectedKeywords // public exposure of the composition

  {

    get { return \_keywords.SelectedKeywords; }

  }

  private readonly Keywords \_keywords; // private composition

}

When creating model classes that utilize compositions to include other model classes it is recommended to use the CompositeTypeAttribute to annotate the model class. When this is used in conjunction with the default AttributeBasedBindingContract this will ensure that all binding contracts set forth by the composite types will also be included when the item to model class binding is asserted.

## Binding contracts

Binding contracts are used to assert or verify that a Sitecore item can be bound to a specific model class. The framework contains a default binding contract called the AttributeBasedBindingContract that uses class attributes to specify the individual clauses of the binding contract. These class attributes can be added to the model classes to express binding contract requirements. The default AttributeBasedBindingContract can be extended with new class attributes to handle specialized requirements or the AttributeBasedBindingContract as a whole can be substituted with a custom binding contract if needed. The binding contract is used by the default model factory and the item extensions described later in this document.

### AttributeBasedBindingContract

The AttributeBasedBindingContract uses class attributes to specify its binding contact. The attributes are applied to the model classes to specify the different requirements that an item must meet in order to comply with the binding contract. The default attributes RequiredBaseTemplate and RequiredField are available in the framework and additional custom attributes can be added by creating new attributes that inherits from the ClauseAttribute base attribute. In addition the AttributeBasedBindingContract includes a CompositeTypeAttribute used for annotating model classes that include other model classes that are not inherited directly through the inheritance chain.

#### RequiredBaseTemplate attribute

The RequiredBaseTemplate attribute is used by the AtttributeBasedBindingContract. It can be applied to a model class to specify a template that the Sitecore item has to inherit from in order for the item to comply with the binding contract. Multiple RequiredBaseTemplate attributes can be set on a model class to specify that the item must inherit from multiple templates. However it is not recommended to base model classes on multiple templates as described earlier in this document.

RequiredBaseTemplate attribute example:

[RequiredBaseTemplate("{7F97671B-E1CA-4537-823C-230A7EAC4DED}")]

public class News : ItemBoundModel

{

  public News(Item item) : base(item)

  {

  }

}

#### RequiredField attribute

The RequiredField attribute is used by the AtttributeBasedBindingContract. It can be applied to a model class to specify that a specific field on the Sitecore item has to contain a value in order for the item to comply with the binding contract. Optionally the attribute can also specify whether this clause should be ignored when Sitecore is running in PageEdit mode. This is to facilitate editing empty fields on items that should not be deemed valid in normal page mode but however should be allowed in page edit mode so that empty fields can be edited. The default behaviour is to ignore this claim when running in PageEdit mode to allow for editing empty fields. Multiple RequiredField attributes can be set on a model class to specify multiple required fields.

RequiredField attribute example:

[RequiredField("{82B7DEB1-293A-49F4-8434-57B9B62F420F}", false)]

public class News : ItemBoundModel

{

  public News(Item item) : base(item)

  {

  }

}

#### Extending the AttributeBasedBindingContract

The AttributeBasedBindingContract can be extended with additional clause attributes to specify other types of binding contract clauses in addition to the existing RequiredBaseTemplate and RequiredField attributes. Additional attributes are created by creating a new class that inherits from the ClauseAttribute class. The new attribute has to implement the methods IsComplied(Item item) and Assert(Item item, Type type). The IsComplied method returns a boolean result to indicate whether the specified item complies with the binding contract clause or not. The Assert method should throw an error if the specified item does not comply with the binding contract clause.

ClauseAttribute definition:

using System;

using Sitecore.Data.Items;

namespace ItemBinding.Model.BindingContracts

{

  [AttributeUsage(AttributeTargets.Class, AllowMultiple = true)]

  public abstract class ClauseAttribute : Attribute

  {

    public abstract Boolean IsComplied(Item item);

    public abstract void Assert(Item item, Type type);

  }

}

#### CompositeTypeAttribute

The CompositeTypeAttribute is used to mark model classes that encompass other model class objects that are not directly inherited by the model class. In order to use the attribute simply add it to the class declaration and point it to the type of the encompassed model class. When used in conjunction with the AttributeBasedBindingContract the CompositeTypeAttribute ensures that all binding contract attributes set on the encompassed model class are asserted as well when a Sitecore item is bound to the model class or when the validity of the item binding is checked.

CompositeTypeAttribute usage example:

[RequiredBaseTemplate("{7F97671B-E1CA-4537-823C-230A7EAC4DED}")]

[CompositeType(typeof(Keywords))]

public class News : ItemModel

{

  public News(Item item) : base(item)

  {

  }

  public Keywords Keywords { get; set; }

## Model factory

The ModelFactory is used to create model instances and bind Sitecore items to the instances. The default ModelFactory can be supplemented with additional specialized model factories to create model class instances that require specialized logic. This is done either by creating a new factory class that inherits from the ModelFactory class to reuse parts of the existing functionality or by creating a new factory class that implements the IModelFactory interface. The custom ModelFactory can then be injected in the ModelBoundUserControl and the Item extensions to facilitate localized custom behaviour or registered globally with the ModelFactoryService to ensure that the model factory is always used when creating model classes of this type.

The model factory contains a Create method that accepts a Sitecore item as a parameter and returns a model class of the type specified by the generic type T and a Create method that also accepts a Sitecore item and an additional Type parameter that specifies the Type of the model class to create. The Create methods of the standard model factory will automatically enforce any constraints set by the binding contract that is bound to the model factory through the BindingContract property and will throw an exception if the constraints are not met.

IModelFactory interface definition:

using System;

using ItemBinding.Model.BindingContracts;

using Sitecore.Data.Items;

namespace ItemBinding.Model

{

  public interface IModelFactory : ICloneable

  {

    T Create<T>(Item item) where T : class;

Object Create(Item item, Type type;

    IBindingContract BindingContract { get; set; }

  }

}

The model factory can be instantiated manually either by calling the method GetPrototypeClone()in the ModelFactoryService as described below or by creating a new instance directly when needed. Additionally a custom binding contract can be injected to replace the default AttributeBasedBindingContract if needed. The binding contract is used in the Create methods of the factory to assert that the provided item complies with the contract and can safely be bound to the specified model class.

Custom model factory instantiation example:

public IEnumerable<Keyword> SelectedKeywords

{

  get

  {

    IModelFactory modelFactory = new KeywordFactory(); // Custom factory

    ChildList children = InnerItem.GetChildren();

    foreach (Item child in children)

    {

      yield return modelFactory.Create<Keyword>(child);

    }

  }

}

Please note that the method for returning child items bound to model classes shown in the example above is not the recommended way of handling child item bindings in the framework. Instead use the BindChildrenAs item extension method described below.

### BindingContract property

The BindingContract property on the ModelFactory exposes the BindingContract that is used internally by the factory to assert that the Sitecore items can be bound to the model classes. A different BindingContract can be set to change the BindingContract behaviour. The default ModelFactory uses the AttributedBasedBindingContract.

## ModelFactoryService

Custom model factories can be created, instantiated and used locally in all operations in the framework. This means that a custom model factory can be created and passed as an argument to the various Item extensions or can be set on a ModelBoundUserControl through overriding the default model factory as described below. However if the framework should always use a custom model factory instead of the default ModelFactory that is provided by the framework the ModelFactoryService can be used to change the default model factory and to register custom model factories globally. To change the global default model factory first create an instance of the custom model factory that should replace the global implementation and configure it as required, this can involve assigning a specialized binding contract to the model factory. Once the model factory is configured it should be passed to the static SetPrototype method exposed by the ModelFactoryService. The various parts of the ItemBinding framework will now use the model factory that was passed to the ModelFactoryService as the default model factory. The GetPrototypeClone method will create a new clone of the model factory instance that was passed to the SetPrototype method which will ensure that the integrity of the model factory prototype is maintained even if the returned clone is changed. Please note that the GetPrototypeClone is marked as obsolete and that the ResolveModelFactory methods should preferably be used to ensure that globally registered custom model factories are taken into account. To register a global custom model factory the method RegisterModelFactory can be used to register an instance of the model factory and specify the model class type that the factory should handle.

ModelFactoryService definition:

using ItemBinding.Model;

namespace ItemBinding.Application

{

  public static class ModelFactoryService

  {

    static ModelFactoryService()

    {

      \_prototypeModelFactory = new ModelFactory();

ModelFactories = new Dictionary<Type, IModelFactory>();

    }

public static void RegisterModelFactory<T>(IModelFactory modelFactory) where T : class

{

Type type = typeof (T);

if (ModelFactories.ContainsKey(type))

ModelFactories[type] = modelFactory;

ModelFactories.Add(type, modelFactory.Clone() as IModelFactory);

}

public static IModelFactory ResolveModelFactory<T>() where T : class

{

return ResolveModelFactory(typeof(T));

}

public static IModelFactory ResolveModelFactory(Type type)

{

if (ModelFactories.ContainsKey(type))

return ModelFactories[type].Clone() as IModelFactory;

return GetPrototypeClone();

}

    public static void SetPrototype(IModelFactory modelFactory)

    {

      \_prototypeModelFactory = modelFactory.Clone() as IModelFactory;

    }

[Obsolete]

    public static IModelFactory GetPrototypeClone()

    {

      return \_prototypeModelFactory.Clone() as IModelFactory;

    }

    private static IModelFactory \_prototypeModelFactory;

private readonly static Dictionary<Type, IModelFactory> ModelFactories;

  }

}

Note that the value of the \_prototypeModelFactory will automatically default to the ModelFactory supplied by the framework.

## Item extensions

The framework provides a series of Sitecore Item extensions to provide easy access to the item binding functionality. The following extensions are available from the ItemBinding.Model namespace.

Boolean IsBindable<T>(this Item item)

Boolean IsBindable<T>(this Item item, IBindingContract bindingContract)

* When called on an item with a model class specified as T this method will return true if the item complies with the binding contract set on the model factory prototype exposed by the ModelFactoryService. The optional overload can be used to specify a custom binding contract to use for verifying contract compliance instead of the default binding contract.

Boolean IsSelectedItemBindable<T>(this Item item, ID fieldId)

**Boolean IsSelectedItemBindable<T>(this Item item, ID fieldId, IBindingContract bindingContract)**

* When called on an item with a model class specified as T this method will return true if the selected item in the field specified by the fieldId complies with the binding contract set on the model factory prototype exposed by the ModelFactoryService. The optional overload can be used to specify a custom binding contract to use for verifying contract compliance instead of the default binding contract.

T BindAs<T>(this Item item)

T BindAs<T>(this Item item, IModelFactory modelFactory)

* When called on an item with a model class specified as T this method will return a new model class instance with the item bound to it. The optional overload can be used to specify a custom model factory to use for the binding. When the standard model factory is used this method will call the Assert method on the BindingContract currently attached to the model factory which will throw an exception if the item doesn’t meet the binding contract constraints.

IEnumerable<T> BindChildrenAs<T>(this Item item)

IEnumerable<T> BindChildrenAs<T>(this Item item, IModelFactory modelFactory)

* When called on an item with a model class specified as T this method will return a collection of model class instances bound to the child items of the item. The optional overload can be used to specify a custom model factory to use for the binding. When the standard model factory is used any children that are not valid will be omitted from the collection and no binding exceptions will be thrown by the binding contract.

IEnumerable<T> BindSelectedItemsAs<T>(this Item item, ID fieldId)

**IEnumerable<T> BindSelectedItemsAs<T>(this Item item, ID fieldId, IModelFactory modelFactory)**

* When called on an item with a model class specified as T and a field id of a valid Sitecore item multi select field present on the item this method will return a collection of model class instances bound to the target items selected in the field. The optional overload can be used to specify a custom model factory to use for the binding. When the standard model factory is used any target items that are not valid will be omitted from the collection and no binding exceptions will be thrown by the binding contract.

T BindSelectedItemAs<T>(this Item item, ID fieldId)

T BindSelectedItemAs<T>(this Item item, ID fieldId, IModelFactory modelFactory)

* When called on an item with a model class specified as T and a field id of a valid Sitecore item single select field present on the item this method will return a new model class instance with the selected target item in the field bound to it. The optional overload can be used to specify a custom model factory to use for the binding. When the standard model factory is used this method will call the Assert method on the BindingContract currently attached to the model factory which will throw an exception if the target item doesn’t meet the binding contract constraints.

Boolean IsAncestorOrSelfBindable<T>(this Item item)

Boolean IsAncestorOrSelfBindable<T>(this Item item, IBindingContract bindingContract)

* When called on an item with a model class specified as T this method will return true if the item or one of its ancestors complies with the binding contract set on the model factory prototype exposed by the ModelFactoryService. The optional overload can be used to specify a custom binding contract to use for verifying contract compliance instead of the default binding contract.

T BindAncestorOrSelfAs<T>(this Item item)

T BindAncestorOrSelfAs<T>(this Item item, IModelFactory modelFactory)

* When called on an item with a model class specified as T this method will return a new model class instance with the item or its nearest ancestor bound to it. The optional overload can be used to specify a custom model factory to use for the binding. When the standard model factory is used this method will call the Assert method on the BindingContract currently attached to the model factory which will throw an exception if the item or one of its ancestors doesn’t meet the binding contract constraints.

IEnumerable<T> BindItemsAs<T>(this IEnumerable<Item> items)

IEnumerable<T> BindItemsAs<T>(this IEnumerable<Item> items, IModelFactory modelFactory)

* When called on an Ienumerable collection of items with a model class specified as T this method will return a collection of model class instances bound to the items of the provided collection. The optional overload can be used to specify a custom model factory to use for the binding. When the standard model factory is used any items in the provided collection that are not valid will be omitted from the resulting collection and no binding exceptions will be thrown by the binding contract.

## ModelBoundUserControl

The ModelBoundUserControl is an abstract base user control that can be inherited by a standard ASP.NET UserControl. When inherited the ModelBoundUserControl provides an easy way to bind a Sitecore item to a model class instance that is exposed through the Model property of the ModelBoundUserControl.

The Model property and the members exposed by the specific model class can then be accessed either from the markup of the user control or from the codebehind file by using the Model property.

Example of model access in markup:

<sc:Text Item="<%# Model.InnerItem %>" Field="<%# Model.NameFieldId %>" runat="server"/>

Example of model access in codebehind:

String Name = Model.InnerItem[Model.NameFieldId];

ModelBoundUserControl definition:

using System;

using System.Web.UI;

using ItemBinding.Model;

using SitecoreExtensions;

using Sitecore.Data.Items;

using Sitecore.Diagnostics;

namespace ItemBinding.Presentation

{

  public abstract class ModelBoundUserControl<T> : UserControl where T : class

  {

    public virtual T Model

    {

      get

      {

        if (\_model != null)

          return \_model;

        if (SourceItem == null)

          return \_model = null;

        try

        {

          return \_model = ModelFactory.Create<T>(SourceItem);

        }

        catch (Exception exception)

        {

          Log.Error(

String.Format(

"An error occured in {0}", GetType().FullName), exception, this);

          return null;

        }

      }

    }

    protected virtual IModelFactory ModelFactory

{

get {

return \_modelFactory ??

(\_modelFactory = ModelFactoryService.ResolveModelFactory<T>());

}

}

    protected virtual Item SourceItem

    {

      get { return \_sourceItem ?? (\_sourceItem = this.GetDataSourceOrContextItem()); }

    }

    protected override void OnInit(EventArgs e)

    {

      base.OnInit(e);

      if (Model != null)

        DataBind();

    }

    private Item \_sourceItem;

    private T \_model;

    private IModelFactory \_modelFactory;

  }

}

### Virtual properties of the ModelBoundUserControl

The ModelBoundUserControl has a series of virtual properties that can be overridden to alter the standard behaviour.

#### SourceItem

The default SourceItem property exposes the data source item attached to the Sitecore rendering or the Sitecore.Context.Item if the data source is not set.

This property can be overridden to expose any other item that should be bound to the model class instance stored in the Model property. A typical scenario where this property would be overridden is when a fixed item such as the website home item or a specific settings item should be used for the model binding. For instance this could be used in a Logo rendering to permanently bind the rendering to the website home item where the information is stored.

Override example:

protected override Item SourceItem

{

  get { return Sitecore.Context.Database.GetItem("/sitecore/content/Website/Home"); }

}

#### ModelFactory

The ModelFactory property exposes the default model factory instance that is used to bind the item returned from the SourceItem property to the model class instance stored in the Model property.

This property can be overridden either to instantiate a standard model factory with a custom binding contract or to instantiate an entirely different custom model factory.

Default model factory override with custom binding contract example:

protected override IModelFactory ModelFactory

{

  get

  {

    IModelFactory modelFactory = new ModelFactory();

modelFactory.BindingContract = new CustomBindingContract();

    return modelFactory;

  }

}

Custom factory override example:

protected override IModelFactory ModelFactory

{

  get

  {

    return new CustomFactory();

  }

}

#### Model

The default Model property exposes the model class instance that is created by binding the item provided by the SourceItem property to the model class using the model factory exposed by the ModelFactory property.

Usually this property should not be overridden but in rare cases it may be required to override this property to expose a Model instance that is created entirely without the use of the SourceItem or ModelFactory properties.

Model override example:

public override Contact Model

{

  get

  {

    return new Contact();

  }

}

### DataSourceUnavailable and DataSourceUnpublishable

The ModelBoundUserControl has a built in functionality to handle situations where the datasource is unavailable or unpublished. In these scenarios the content of the ModelBoundUserControl will not be rendered if the page is running in normal mode. If the page is running in page edit mode the content will be replaced with a DataSourceInfo web control that will display an error message to the editor. The error messages displayed in the DataSourceInfo control can be altered in several different ways if required to support translation. To alter the texts globally either add two new appSettings to the appSettings section in the web.config file with the keys ItemBinding.DataSourceUnavailableInfoText and ItemBinding.DataSourceUnpublishableInfoText and the texts to display as the values or set the texts on the two properties DataSourceUnavailableInfoText and DataSourceUnpublishableInfoText exposed by the static TextService located at ItemBinding.Application.TextService . It is recommended to set the properties of the TextService at application start which can be achieved by means of the Global.asax, the Sitecore initialization pipeline or by using WebActivatorEx which is available through NuGet. To alter the texts locally on a single user control that inherits from the ModelBoundUserControl please override the virtual DataSourceUnpublishableInfoText and DataSourceUnavailableInfoText members that are available on the ModelBoundUserControl and return the text to be displayed.

## Sitecore MVC support

The ItemBinding framework also supports item to model binding in Sitecore’s MVC implementation when working with MVC controllers and view renderings where the model is automatically assigned to the view. Large parts of the ItemBinding framework functionality such as model factories, item extensions and binding contracts also applies when working with the MVC framework but in addition the framework offers an additional subset of functionality which is intended for use with the MVC framework.

### Controller Renderings

When creating a controller rendering in Sitecore the controller class that should handle the request is specified. This controller would normally either be derived from the Standard ASP.NET MVC controller or from the SitecoreController which is a subclass of the standard controller class that offers additional Sitecore specific functionality. The ItemBinding framework includes a ModelBoundController that is essentially a subclass of the SitecoreController class with additional logic to instantiate a model using the ItemBinding features that can then be passed to the view. There are two major benefits when using the ModelBoundController. Firstly it means that the model is automatically created and bound to the source item of the rendering and secondly it ensures that the standard BindingContract scheme in the ItemBinding framework is invoke and that the source item is therefore validated towards this contract before being bound to the model. As with the rest of the ItemBinding framework the only requirement is that the model class specified by the ModelBoundController must contain a constructor that accepts a single item parameter which means that the model class is free to inherit from any base class like for instance the standard Sitecore renderingmodel class which will be treated as a standard renderingmodel instance by the rest of the Sitecore MVC framework.

#### ModelBoundController

ModelBoundController definition:

using System;

using System.Web.Mvc;

using ItemBinding.Application;

using ItemBinding.Model;

using Sitecore.Data.Items;

using Sitecore.Diagnostics;

using Sitecore.Mvc.Presentation;

namespace ItemBinding.Mvc.Presentation

{

public class ModelBoundController<T> : Controller where T : class

{

public virtual T Model

{

get

{

if (\_model != null)

return \_model;

if (SourceItem == null)

return \_model = null;

try

{

\_model = ModelFactory.Create<T>(SourceItem);

if (\_model is IRenderingModel)

InitializeRenderingModel(\_model);

return \_model;

}

catch (Exception exception)

{

Log.Error(String.Format("Unable to bind the source item '{0}' to the model class '{1}'", SourceItem.Paths.FullPath, typeof(T).FullName), exception, this);

return null;

}

}

}

private void InitializeRenderingModel(T model)

{

IRenderingModel renderingModel = model as IRenderingModel;

if (renderingModel == null)

return;

renderingModel.Initialize(Rendering);

}

protected virtual IModelFactory ModelFactory

{

get { return \_modelFactory ?? (\_modelFactory = ModelFactoryService.ResolveModelFactory<T>()); }

}

protected virtual Item SourceItem

{

get { return \_sourceItem ?? (\_sourceItem = Rendering.Item ?? PageContext.Current.Item); }

}

protected virtual Rendering Rendering

{

get { return RenderingContext.Current.Rendering; }

}

private T \_model;

private Item \_sourceItem;

private IModelFactory \_modelFactory;

}

}

### View Renderings

When using Sitecore view renderings a model class is specified in the view rendering definition item and when the view is instantiated Sitecore will automatically create an instance of the model class in the Mvc.GetModel pipeline. The ItemBinding framework extends this behaviour by inserting an additional GetModelProcessor in this pipeline that will instantiate the model class in place of the standard processor if the specified model class contains a constructor that accepts a single item parameter and will ensure that the model class is either bound to the rendering source item if present or the context item and at the same time ensure that any binding contract restrictions are met by the item. The same model class requirements that are required by the ModelBoundController also apply to the view rendering implementation.

The view rendering processor hookup is added by including the ItemBinding.Mvc.config file in the App\_Config/Include folder.

GetModelProcessor Definition:

using System;

using Sitecore.Data.Items;

using Sitecore.Mvc.Extensions;

using Sitecore.Mvc.Pipelines.Response.GetModel;

using Sitecore.Mvc.Presentation;

namespace ItemBinding.Mvc.Infrastructure.Pipelines.Response.GetModel

{

public class GetFromRenderingItem : GetModelProcessor

{

public override void Process(GetModelArgs args)

{

if (args.Result != null)

return;

Rendering = args.Rendering;

args.Result = GetModelFromField();

}

protected virtual Object GetModelFromField()

{

Item item = Rendering.RenderingItem.ValueOrDefault(i => i.InnerItem);

return item != null ? ModelLocator.GetModel(item["Model"], SourceItem, true) : null;

}

protected Rendering Rendering { get; private set; }

protected Item SourceItem

{

get { return \_sourceItem ?? (\_sourceItem = Rendering.Item ?? PageContext.Current.Item); }

}

protected ModelLocator ModelLocator

{

get { return \_modelLocator ?? (\_modelLocator = new ModelLocator()); }

}

private Item \_sourceItem;

private ModelLocator \_modelLocator;

}

}

### ItemBoundRenderingModel

The ItemBinding framework contains an ItemBoundRenderingModel base class that can be used as the basis for creating model classes. The base class offers a standardized handling of the item being bound to the model class. Additionally the base class inherits from Sitecore’s standard RenderingModel class which means that the rest of the MVC framework will treat the model class in the same way as the RenderingModel class.

ItemBoundRenderingModel Definition:

using Sitecore.Data.Items;

using Sitecore.Mvc.Presentation;

namespace ItemBinding.Mvc.Model

{

/// <summary>

/// Represents a base rendering model class that can be used by model classes to standardize access to the bound item.

/// </summary>

public class ItemBoundRenderingModel : RenderingModel

{

/// <summary>

/// Initializes a new instance of the <see cref="ItemBoundRenderingModel"/> class.

/// </summary>

/// <param name="item">The item.</param>

public ItemBoundRenderingModel(Item item)

{

\_innerItem = item;

}

/// <summary>

/// Gets the inner item bound to the ItemBoundRenderingModel.

/// </summary>

/// <value>The inner item bound to the ItemBoundRenderingModel.</value>

public Item InnerItem

{

get { return \_innerItem; }

}

private readonly Item \_innerItem;

}

}