Synthesis Reference

What is Synthesis?
Synthesis is a development framework for Sitecore CMS that enables developing more reliable and maintainable sites in less time than traditional Sitecore development. It combines a strongly typed template object generator with a powerful presentation framework that’s easily understandable for developers with either a Sitecore or traditional .NET background.

Some key features:
- Synthesis is designed with the developer’s user experience in mind. Strong typing, IntelliSense support, and a simple but powerful configuration model allow you to start out easily but customize as deep as you need without resorting to hacks.
- Synthesis thinks like Sitecore. It uses the same conventions and terms you’re already used to, and implements all standard Sitecore functionality such as Page Edit/Design mode, OMS, and rendering data sources.
- The architecture behind Synthesis has been proved over three years of active development on more than ten major sites that are now in production.
- Using a site architecture based on template composition? You’ll love Synthesis. It generates an interface hierarchy to represent template inheritance hierarchies, allowing you to easily interrogate an item to see if it has a certain set of fields or hide a rendering if the data source doesn’t contain the template it expects.
- Powerful presentation framework allows you to create very DRY, single-responsibility renderings. Focus on presentation code and not data access by coding renderings against interface contracts or ViewModels. Your renderings can be ignorant that Sitecore exists, opening up easily testable renderings. It’s also extensible to handle even complex renderings with multiple data sources without increasing coupling.
- Simple, extensible field API makes it easy to read or write all template field types in an efficient, strongly-typed manner. Never have to deal with boilerplate LinkManager or media item code again.

Comment [KF1]: Photoshop Elements lol. Could probably use a better intro though.
Lightning fast. Synthesis can turn 1000 Sitecore items into their strongly typed equivalents in less than 1 millisecond.

**How Synthesis Generates Objects from Templates**

Before diving in to how to use Synthesis, it’s important to have a basic theoretical foundation about how strongly typed template objects get created. Consider the following simple template hierarchy in Sitecore:

<table>
<thead>
<tr>
<th>Page Information</th>
<th>Navigation Information</th>
<th>Simple Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Title : Single-Line Text</td>
<td>Show in Navigation : Checkbox</td>
<td>inherits Page Information</td>
</tr>
<tr>
<td></td>
<td></td>
<td>inherits Navigation Metadata</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Content : Rich Text</td>
</tr>
</tbody>
</table>

Once Synthesis had generated objects for these templates, the following classes and interfaces are created to represent them:
Each template has two generated representations: a concrete class and an interface. The class implements the interface. The template’s interface implements interfaces representing any base templates. As a result, one is able to refer to the SimplePage class as an IPaginationItem. This powerful feature provides support for information hiding and generalization. It is possible to design a rendering that expects an object implementing IPaginationItem that will work not only for a Simple Page, but also any template that descends from the Page Information template. The rendering can also have polymorphic behavior based on the template type it receives, or simply hide itself if its data source isn’t a relevant type. The ability to refer to an item by any of its base template interfaces is one of the most powerful abilities of Synthesis. It opens up a whole new class of polymorphic architectures that would be very difficult or perform badly without it.
Demonstration: Implementing a Rendering using Synthesis

Using Synthesis’ presentation framework on a Web Control or Sublayout rendering is very simple. For the purposes of demonstration it’s assumed that you’ve already created a WebControl rendering item and attached it to an item’s layout details.

Your code for the basic Sitecore Web Control rendering probably looks something like this:

```csharp
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;

namespace SynthesisDemo.Web.layouts
{
    public class SampleWebControl : Sitecore.Web.UI.WebControl
    {
        protected override void DoRender(System.Web.UI.HtmlTextWriter output)
        {
            output.Write("Hello, world!");
        }
    }
}
```

Let’s turn this simple control into a control that renders the Simple Page template from the previous template example. First, we need to change the class the control derives from to Synthesis.UI.WebControls.WebControlView<ISimplePageItem>. This base class defines our rendering to be a Synthesis View. Views have a Model property that contains their content data source transformed into a strongly-typed ViewModel. In this case our ViewModel is an ISimplePageItem interface – you can also define your own custom ViewModels (see the section on presentation framework for details).

After changing to the View base class, it’s time to delete the DoRender() method and instead implement the view’s abstract RenderModel() and GetCachingID() methods. The RenderModel method behaves the same as Sitecore’s DoRender. GetCachingID() returns a unique caching identifier (see the section on presentation framework for more detail). Now we can render the sample template by accessing the properties of our model. The rendering’s code now looks something like this:
When this rendering is rendered to the page, it emits the Title field value from Sitecore in a `<h1>` tag. This example is the simplest use of Synthesis; it is capable of a lot more. For example, in the above code it’s actually taking advantage of an implicit cast to string present in the Model.Title property – the property is actually an object with metadata and multiple rendering options. Ready to give Synthesis a shot? Next we’ll go over how to install it.

**Getting Started: Installation**

Synthesis is designed to be as self-contained as possible, and is therefore quite easy to install. Copy one configuration file, one assembly, and (optionally) register a HttpHandler and HttpModule and you’re done. It doesn’t scatter files all over your Sitecore installation, and it needs not a single item created in Sitecore to do its job. To install Synthesis:

- Copy Synthesis.config to your site’s App_Config/Include folder.
- Copy Synthesis.dll to your site’s /bin folder or add a reference to it on your web project in Visual Studio.
- Register the Synthesis HttpModule in your Web.config file. *If you don’t want to take advantage of automatic synchronization checks on startup, you do not need to register this module.*

Register the module last among the existing modules by adding the following line:

```xml
<add type="Synthesis.HttpModule, Synthesis" name="Synthesis"/>
```

You will want to add this line in two places:

- Under configuration/system.webServer/modules (IIS7+, integrated pipeline)
- Under configuration/system.web/httpModules (IIS6/IIS7 classic pipeline)
- Register the Synthesis HttpHandler in your Web.config file. **If you don’t want to use the Synthesis control panel, you do not need to register this handler.** Register the handler first among the existing handlers. You will want to register the handler in two places:
  - Under configuration/system.webServer/handlers (IIS7+, integrated pipeline):
    ```xml
    <add name="Synthesis" path="synthesis.axd" verb="GET" type="Synthesis.HttpHandler, Synthesis"/>
    ```
  - Under configuration/system.web/httpHandlers (IIS6/IIS7 classic pipeline)
    ```xml
    <add verb="GET" path="synthesis.axd" type="Synthesis.HttpHandler, Synthesis"/>
    ```
- That’s it – you’re done. You’ll probably want to read the next section about configuration though, as the default configuration will work but may not be how you want it.

**Getting Started: Configuration**

Synthesis has one place for configuration: the Synthesis.config file located in your web site’s App_Config/Include folder. This file’s primary purpose is to configure the implementations used for Synthesis’ Provider Model, and the parameters for each provider. The Provider Model is how Synthesis allows customization of its behaviors, such as which templates get objects generated for them or how to translate a Sitecore field name into a Synthesis field type.

Out of the box Synthesis comes with a set of very flexible providers. These providers are configurable by using the very powerful configuration grammar Sitecore provides. The stock Synthesis.config file includes all possible parameters for each standard provider and is heavily commented. This document won’t be covering each parameter in detail, so make sure to look over the configuration file. If you choose to implement your own provider, you may implement a different set of parameters.

Synthesis has five different types of provider. Let’s take a closer look at each one.

**The Generator Parameters Provider**

In Synthesis lexicon the Generator is the component that generates the concrete and interface implementations of a Sitecore template. The Generator Parameters Provider then is a provider that configures where and how the generator emits code, such as namespaces, output file paths, and the base class of all concrete model objects. This provider must implement the Synthesis.Generation.IGeneratorParametersProvider interface. The default implementation just maps the provided parameters onto the properties of the Synthesis.Generation.GeneratorParameters class.

**The Template Input Provider**

Synthesis needs to be given a set of templates from which it should generate an object model or compare against when checking if the model is synchronized. It also needs to know which fields on those templates are valid model properties and which should be ignored – for example, you probably don’t need a property for Sitecore’s Icon field. This is the purpose of the Template Input Provider. This provider must implement the Synthesis.Templates.ITemplateInputProvider interface. The default implementation uses a ridiculously flexible system where you can include or exclude templates and fields in almost any way imaginable – by path, name, ID, or even wildcards. The default Template Input
Provider has over 70 integration tests to check if it’s behaving correctly – it’s that flexible. The configuration grammar for the default provider is documented in the stock configuration file.

Note: Because of their additive nature, you may find that interfaces are generated for templates you have excluded using the Template Input Provider. This will occur if a template that is included for generation derives from a template that is not included. You can use this to your advantage by not including “abstract” templates that you don’t want a concrete implementation of and still having their interface available via a derived template’s implementation.

The Type List Provider
When a template synchronization check runs, Synthesis needs to know which .NET types could possibly contain Synthesis generated objects. Similarly, when the Presentation Framework looks for custom Presenters, it needs to have a list of candidate .NET types where presenters could live. This is the purpose of the Type List Provider. The provider returns a superset of classes that could include generated objects or presenters, and should cache the set of objects for later quick retrieval.

Implementations must implement the Synthesis.Configuration.ITypeListProvider interface. There are two default implementations of this provider: Synthesis.Configuration.AppDomainTypeListProvider returns the complete set of types loaded into the entire AppDomain and is guaranteed to find all types in any loaded assembly. It requires no configuration, but adds a marginal amount of time (200ms-1s) to the web application’s startup time. Synthesis.Configuration.ConfigurationTypeListProvider takes a more precise approach, allowing you to specify exactly which assemblies you want to search for types in using a configuration grammar. It is several times faster than the AppDomainTypeListProvider, but may require reconfiguration during refactoring.

The Field Mapping Provider
When the Generator processes a field on a Sitecore template, it has to be able to figure out what .NET type it should use to represent that field. This is where the Field Mapping Provider comes in. The Field Mapping Provider implements a conversion mapping between a Sitecore field and a .NET type. This provider is one of the most powerful customization points Synthesis offers – you can change the mapping to your own class and implement custom behaviors for a field type across your entire model. All of the default field classes use largely virtual properties and methods to allow you to easily patch their behaviors. This provider must implement the Synthesis.FieldTypes.IFieldMappingProvider interface. The default implementation uses an XML grammar similar to Sitecore’s FieldTypes.config file to map a template field name such as “Single-Line Text” to a .NET type such as “Synthesis.FieldTypes.TextField,” and is documented in the stock configuration file.

The Template Signature Provider
Synthesis has the ability to compare its generated object model to the current state of templates in Sitecore and look for differences. It accomplishes this by storing a signature value for each template in an attribute on the generated interfaces. When a comparison occurs, the signature is calculated for the template in Sitecore and then compared with the stored signature on the interface; if they match, the template is considered to be synchronized. This is where the Template Signature Provider is used; this provider creates a unique signature for a template’s current state. Provider implementations must
implement the Synthesis.Templates.ITemplateSignatureProvider interface. The default provider implementation creates a SHA-1 hash of the template's name, path, ID, and the names and IDs of each field on the template.

**Working with Template Objects**

Template objects are Synthesis' single most important feature. They are the concrete classes and interfaces that result from the Generator, and represent Sitecore templates in code. Template objects are capable of both reading and writing values, and abstract out many of the boilerplate code situations that are encountered when using Sitecore's built in field types. It's very easy to convert a Sitecore Item class into a strongly typed template object (and back again, if need be).

All Template Objects have common features that a Sitecore Item would have, such as ID, TemplateID, Database, Language, Name, Paths, Axes, and Statistics. These work exactly like their Item equivalents, except for Axes. The Synthesis Axes object returns Template Objects instead of Items. Synthesis also adds a Url property that gets the correct link for the object. There is an InnerItem property that lets you get at the underlying Item, but use it only when absolutely necessary - the Synthesis API is always preferable as it separates concerns.

**Transforming Items into Template Objects**

At its heart Synthesis' Template Objects are just a strongly typed wrapper around a Sitecore Item instance. There are two primary methods you can use to obtain Template Objects:

1. You can implement a Presentation Framework View rendering, which is automatically supplied the rendering's data source as a Template Object. This method is appropriate to use when implementing renderings. See the section on the Presentation Framework for additional details.
2. There is a set of extension methods defined in the Synthesis.Extensions namespace that allow you to transform Sitecore API objects (Item) and collections (IEnumerable<Item>) into Template Objects.
   a. Item.AsStronglyTyped() converts an Item into its Template Object equivalent. An IStandardTemplateItem is returned, but is castable to the actual template type.
   b. Item.As<T>() converts an Item into its Template Object equivalent and expects the result to be of a particular type. If the item does not resolve to that type, null will be returned instead.
   c. IEnumerable<Item>.AsStronglyTypedCollection() converts a collection of items into an IEnumerable<IStandardTemplateItem>(). Each item is castable to its actual template type. Note: this uses lazy enumeration, so copy it to a collection if you plan to iterate over it several times.
   d. IEnumerable<Item>.AsStronglyTypedCollectionOf<T>() converts a collection of items into an IEnumerable<T> (a specific template object type). Any items in the original collection that are not representable using the requested template object type will be excluded from the resulting collection. Note: this uses lazy enumeration, so copy it to a collection if you plan to iterate over it several times.

Comment [KF2]: These are ugly. Maybe use a code style for the final version to make them a tad less...proseless.
Navigating the Item Tree with Template Objects

Once you have a Template Object, you may need to work with items in the Sitecore tree related to the object – parents, children, relative Sitecore queries, and other axis queries. Synthesis provides all template objects with an Axes property that gives you a façade to the Sitecore Item’s Axes property – only all the results are returned as Template Objects. This gives you the ability to manipulate the Sitecore tree without leaving Synthesis.

Note: Unlike a Sitecore Item object, the GetChildren method and Parent property are on the Axes property, not the actual Template Object.

Working with Fields

When you transform a Sitecore Item into a Synthesis Template Object, you gain strongly typed properties that represent each field on the item. These properties are not simply copies of the standard field classes Sitecore offers (i.e. LinkField, ImageField, LookupField), they are extensions that are designed to make it even easier to deal with fields and their values. There is a wide array of field types, from the simple TextField or IntegerField to an ItemReferenceListField (aka multilist).

Each of these types are mapped to one or more Sitecore field types by the Field Mapping Provider. See the section on configuration for more about how the Field Mapping Provider works.

This document does not dive into the details of each field type, as they should be easily discoverable using IntelliSense. That said there are some common features across Synthesis’ field types that are useful to be aware of.

• The fields that support FieldRenderer all have a RenderedValue property that allows you to get the value as it would be if a FieldRenderer was used. The default value of the field is never the rendered value; you must explicitly choose which fields you want to make editable in Page Editor Mode.
• Where it makes sense, the field may be implicitly cast to a primitive .NET data type that represents the field’s value (i.e. DateTimeField can be implicitly used as a DateTime, TextField as a string, etc.)
• They are all implicitly castable to their Sitecore field class equivalent (i.e. a DateTimeField can be cast directly to Sitecore’s DateField class).
• Field types that would be commonly rendered onto ASP.NET web controls (images, links) have Attach methods that allow you to quickly attach the field’s value to the relevant properties of a control (image alt text, link titles and targets, etc.) or render them to HtmlTextWriter for use in WebControl renderings.

Saving Template Objects to Sitecore

Synthesis fully supports editing Template Objects and persisting the results back to Sitecore. The editing API is largely identical to the Sitecore API. Synthesis extends the Sitecore editing API to allow you to set individual property values without explicitly putting the item in edit mode – Synthesis will transparently place the item in edit mode if it needs to when a field’s value is set, and restore editing mode to the way
it was when it’s done. You should still place the item in edit mode manually if editing more than one field for performance reasons.

`ISamplePageItem` typedTemplateObject;

// changes to a single field can use the automatic edit context features
typedTemplateObject.Title.RawValue = "Hello, world";

// you can use `SynthesisEditContext` just like `EditContext` in the Sitecore API
using (new SynthesisEditContext(typedTemplateObject))
{
    typedTemplateObject.Title.RawValue = "Hello, world";
}

// you can use `templateObject.Editing` just like the Sitecore API
try
{
    testTemplateObject.Editing.BeginEdit();
    typedTemplateObject.Title.RawValue = "Hello, world";
}
finally
{
    testTemplateObject.Editing.EndEdit();
}

Synthesis does not provide any direct support for adding new instances of Synthesis Template Objects. To accomplish this, you will need to add the new item using the Sitecore API and then change it into a Template Object to set its properties. Each concrete template object has a static `ItemTemplateID` property that is useful to get the template ID to add for a given template object result, for example:

`Item` myParentItem;
`Item` newItem = myParentItem.Add("New Simple Page", new TemplateID(SamplePage.ItemTemplateID));
`ISamplePageItem` synthesisNewItem = newItem.As<ISamplePageItem>();

using (new SynthesisEditContext(synthesisNewItem))
{
    synthesisNewItem.Title.RawValue = "I’m a brand new Sitecore item!";
}

**Synthesis Presentation Framework**

The Synthesis Presentation Framework was born out of a desire to separate the data logic of a rendering from its presentation logic. A rendering should only need to worry about how it renders its data – not where or how it got data. This lets the application have simpler renderings and data access, and reuse data code across multiple renderings that need the same data.
The presentation framework uses the Model-View-Presenter\(^1\) design pattern. In this pattern, the View (a Sitecore Rendering) is invoked by the Sitecore layout engine just like any other rendering. The View is declared as requiring data of a specific ViewModel\(^2\) type, and it asks the PresenterFactory to find a Presenter (a data access class) that can supply it with a ViewModel. The Presenter then performs some logic to create the requested ViewModel object and returns it to the View for use. The View never knows what Presenter was used to get it the rendering data, or how the data was retrieved – all it has to focus on is rendering.

What about simple renderings that don’t really have any data access logic – like a content page with a title and some content? Simple renderings are handled as well: you can simply use the default presenter when you just need a Template Object as your data source. The default presenter resolves the correct data source item – either the data source attribute set on the rendering in the item’s layout details, or the Sitecore context item if no data source is set – and converts it to the requested Template Object and sends it to the View. Using the default presenter is easy. Just set the ViewModel type on your View to a Template Object type or interface.

TODO: a diagram or the MVP flow might be good here?

**Views**

A View is just a fancy term for a Sitecore rendering that knows how to use the presentation framework. Views may either be WebControl renderings or UserControl renderings (Sublayouts). The View hooks into the presentation framework by deriving from the UserControlView<T> or WebControlView<T> class. The base types, by virtue of their generic parameter, allow the View to specify the type of ViewModel object it expects to get its data from. The expected ViewModel type is then looked up by the Presenter Factory to find a compatible Presenter to provide the data to the View.

Views also have some convenience functionality compared to standard Sitecore renderings.

- By default if the Model comes back as null, the rendering will be hidden and its rendering methods will not be called\(^3\).
- The rendering parameters assigned in Sitecore are exposed as the RenderingParameters collection
- UserControlView exposes the Data Source property set in the Sitecore layout details, as well as the output caching properties configured on the Sublayout
- WebControlView exposes the ability to wrap the control inside an EditFrame by defining a constructor in your View and setting the EnableEditFrame property to true. You may set the Buttons property on the View as you would with an EditFrame.

\(^1\) [http://en.wikipedia.org/wiki/Model-view-presenter](http://en.wikipedia.org/wiki/Model-view-presenter)

\(^2\) A ViewModel is a class designed to transfer the data a View needs in order to render from the Presenter to the View. It may be any type of class or interface; it’s just a data container.

\(^3\) For WebControlView this means the RenderModel() method; for UserControlView the OnInit/Page_Init, OnLoad/Page_Load, OnPreRender/Page_PreRender, and Render methods. This behavior can be disabled by defining a constructor in your View and setting the RenderIfDataIsNull property to true in its body.
- WebControlView requires you to implement the GetCachingID() method, without which Sitecore will not ever output cache your WebControl regardless of cacheable settings.

Presenters and ViewModels
Creating a Presenter allows you to control the contents of a particular type of ViewModel object. As each View declares the type of ViewModel it expects to receive, creating a Presenter that provides that data type causes the View’s data to be provided by the Presenter’s code. It’s important to note that there is no coupling between the View and Presenter however, as they both declare only the ViewModel type. This means that you can reuse the same Presenter with multiple Views, but it also means that a View may only have one Presenter (which is good, because data-switching logic belongs in the Presenter).

Using the Default Presenter
The simplest Presenter is the Default Presenter. To use this Presenter your View just needs to declare its ViewModel to be of a Synthesis Template Item type; the Default Presenter will resolve the rendering’s data source item and return it as the Synthesis Item type requested, if it is compatible with the type.

Using the SynthesisItemPresenter to Extend the Default Presenter
Many times renderings will need mostly the data from a Synthesis Template Object, but need a bit extra logic or to pull data from additional queries. To do this you’ll need to define your own Presenter. To do this,

- Create a new class in a location that is included by the Synthesis Type List Provider
- Derive the class from Synthesis.Presentation.SynthesisItemPresenter<Tmodel, Titem>. Tmodel is the ViewModel type the Presenter will return, and Titem is a Synthesis Template Object (interface) type that the rendering’s data source is expected to be.
- Implement the GetModel method. This method is provided the View and the pre-processed item data source, and returns the ViewModel.
- Any View that defines its ViewModel type as the Tmodel in your Presenter will now invoke your Presenter to get its ViewModel.

Implementing Custom Presenters
It’s also possible to implement a presenter that has no Synthesis logic or data source resolution by directly implementing the Synthesis.Presentation.IPresenter<Tmodel> interface on your Presenter class.

Keeping your Template Objects Synchronized with Sitecore
One of Synthesis’ most powerful capabilities is the ability to validate and self-update the generated Template Objects when changes occur in Sitecore. Synchronization checks may occur at application startup or on-demand. It’s also possible to have a project file get rebuilt automatically when a resynchronization has occurred, keeping your compiled code automatically up to date as well.

Synchronizing on Application Start
Synthesis uses a .NET HTTP Module (see the installation section) to provide on-startup synchronization checks. These checks, and what to do with a desynchronization is found, are configured in the...
App_Config/Synthesis.config file. The comments in the config file explain the various options and their working. If you do not want to use on-startup synchronization checks, you may remove the HTTP Module safely from your Web.config.

**On-Demand Synchronization and Rebuild with the Control Panel**
Synthesis uses a .NET HTTP Handler to provide a simple “control panel.” By default this control panel may be accessed at /synthesis.axd but you can customize the handler definition to change the URL. This control panel allows you to view a graphical version of the synchronization status and trigger a forced rebuild of the generated code whenever you want.

**How Synchronization Works**
Synthesis knows if your templates are in sync with Sitecore’s templates by means of a signature value that changes with any significant modifications to the template. This signature value is added as an attribute to all generated interfaces (the interfaces are used because they represent the total set of templates including inherited templates). Then when a sync check is run the freshly generated signature of all Sitecore templates is compared to the signatures of the currently loaded interfaces.

**Best Practices**
- Designing a template hierarchy, in which your pages are largely or entirely composed of a set of inherited “page section” templates will both make your template maintenance easier and set you up to utilize Synthesis’ interface generation feature to its full potential.
- Your template architecture should be similar to your presentation design. Try to aim for a 1:1 correspondence between a template component and a presentation component. This allows you to easily define a View against the template component’s interface without needing to define a presenter and adds coherence to your architecture. This isn’t always possible however, so don’t bend to accomplish this when it feels like a hack.
- Wherever possible, always refer to template objects as interface types instead of concrete types. Further, use the least derived interface necessary (don’t use an IPageltem that includes navigation data if all you need is an INavigationSettingsItem). This decreases the coupling of your code and helps maintainability when adding new templates.
- Try to use Views for as many presentation components as you can. Using a View allows you to decouple the rendering from its data by injecting the appropriate data object based on the Sitecore context and rendering Data Source.
- Only include the minimum set of templates you need to generate. If you have any templates in Sitecore that are “abstract” and would never have an item directly based on them, they should be excluded from generation. If other templates inherit from your abstract templates, they will still get an interface generated, which is probably what you wanted to do.

**FAQ**
There should be an interface for my template but there doesn’t seem to be?
Check to make sure that the template is either included or that it is inherited by a template that is included.

Why is there a property that ends with a number (“Foo1”) on my template or interface?

In .NET, members of a type may not match the name of the parent object. Chances are your template has a field with the same name as the template – Synthesis made the name in code unique to prevent a compilation error. It’s also possible to have two fields with the same name on a template, which would also cause this.

Why does my template object’s type name end with a number (“Foo1”)?

Synthesis uses some logic to make sure names are unique in generated code so as not to create code that doesn’t compile. If you find a template or interface that ends with a number it indicates that there were more than one template of the same name in the same namespace in the generated code or that a namespace existed with the same name as the object type.

Can you use Synthesis with XSLT renderings?

No. XSLT renderings don’t compile and aren’t strongly typed so there’s not much point to Synthesis supporting them. .NET-based renderings are faster and much easier to implement logic in, and with the right controls are as simple to make as XSLT.

Credits
Synthesis was designed and written by Kam Figy
The Presentation Framework is based on an idea by Rick Cabral
Special thanks to Chris Anderson and Rick Cabral for all the architecture discussions
Thanks to the whole ISITE Design tech team for invaluable commentary, patches, and real-world testing
Robin Hermanussen’s Compiled Domain Model for the idea of checking if the model is in sync with Sitecore