Classy Sitecore v2

Contents

[Overview 1](#_Toc299455044)

[Installation 2](#_Toc299455045)

[Using the ClassySC Manager 2](#_Toc299455046)

[UI 2](#_Toc299455047)

[Edit Folder 4](#_Toc299455048)

[Edit Template 5](#_Toc299455049)

[Edit Field 6](#_Toc299455050)

[Preview Code 7](#_Toc299455051)

[Generate 8](#_Toc299455052)

[Using Generated Classes 8](#_Toc299455053)

[Setting up your .sln 8](#_Toc299455054)

[Generated Class Architecture 8](#_Toc299455055)

[Constants 9](#_Toc299455056)

[Attributes 10](#_Toc299455057)

[Property Mappings 10](#_Toc299455058)

[Let’s use the classes! 12](#_Toc299455059)

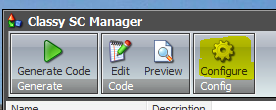
[Editing an Item 14](#_Toc299455060)

[Extending the Generated Classes 14](#_Toc299455061)

# Overview

Classy Sitecore is a c# code generator that creates class wrappers around Sitecore Items based on the template of those items. These concrete classes force us to write cleaner code that will help spot data model bugs at compile time rather than runtime and help us think of our application programming more abstractly. The generator creates clean hierarchical class files and ID-based field accessors so that we don’t rely on “magic strings” to access our fields or query by template names. Part of the overall quality drive this code generator is part of is to remove all “magic strings” from our code (Item Paths, field names, and to a degree queries) in favor of tightly controlled groups of constants. This will also help us separate data/business/presentation in our solutions.

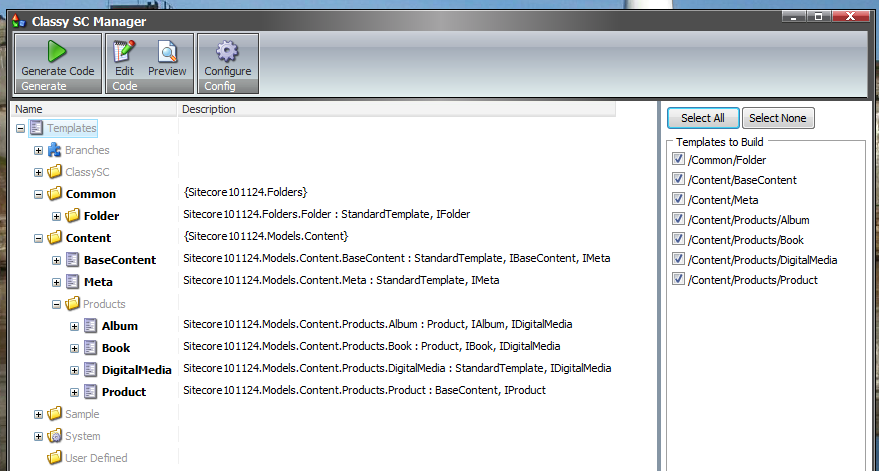
# Installation

1. Install ClassySC-xx.zip in the Sitecore Package Installer
2. If this is updating a previous installation you should backup
   1. The item /sitecore/system/Modules/ClassySC/ClassySC Config
   2. The file /Website/App\_Config/Include/ ClassySC.config
3. In /Website/App\_Config/Include/ ClassySC.config, set the *SolutionPath* setting to the Root path of your solution. For Example if your website is C:\inetpub\wwwroot\MySuperSite\Website, then your solution path is C:\inetpub\wwwroot\MySuperSite. This value can vary depending on the individual development environment so it needs to be in the .config file rather than Sitecore.
4. From the Start Menu ->All Programs -> ClassySC Manager
5. Click the Configure button   
   
6. In the Config Dialog enter the following values
   1. Using Namespaces – These namespaces are referenced at the top of every generated class file. The included values should have you covered, edit only if necessary.
   2. Base Template Class Name – This class is the base class that all generated classes eventually inherits. The default value is a class named “StandardTemplate” and should be the value of this field 99% of the time unless you REALLY need a different base class. **[note the ability to replace this class isn’t fully enabled, don’t touch]**
   3. Default Namespace – This is the default namespace for any generated classes. This value can/should be overridden on your top template folders to enable automatic hierarchical namespacing of classes.
   4. Default FilePath – This is the file location of all the generated files relative to [slnpath].   
      For example “[slnpath]\TestNamespace.Models\” will replace [slnpath] with the value from your ClassySC.config before appending the class file.  
      C:\inetpub\wwwroot\MySuperSite\TestNamespace.Models\MyClass.cs  
      Like the namespacing, you should override this value on the top template folders of your templates so that you can get hierarchical filepaths to match your template hierarchy.
   5. Click OK to save the configuration.

# Using the ClassySC Manager

## UI

The central mode of managing your generated classes will be through the ClassySC Manager Application. It manages the class generation settings of Templates, Template Folders, and Fields, storing their values as an XML element inside /sitecore/system/modules/ClassySC/ClassySC Config.

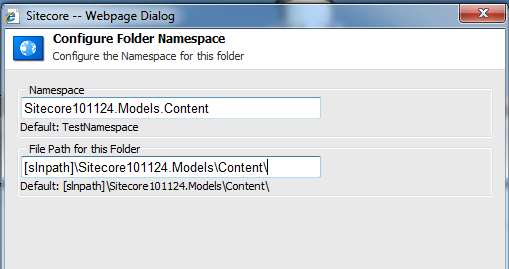


The left hand side is the Template navigator. It lets you browse the template tree for templates to generate. A bolded Template or Field indicate that they are being generated. A bold Template Folder indicates that it has Namespace/Filepath settings to cascade to its child templates. Templates that are selected to generate show up bold with class signatures to the right. The class descriptions show you what the class will look like once generated (only if that template has ‘Generated’ checked). Fields will have the property type and property name to the right.

The right portion of the App is the build list. Any templates with ‘Generate’ checked will show in this list. You may select one or more templates from this list to be generated when the “Generate Code” button is clicked.

On the top menu bar the Generate Button will generate the class files for any templates listed on the right. The Edit button will pull up the Edit Template dialog when a template is selected in the navigator, or the Edit Field dialog when a field is selected in the navigator, or the Edit Folder dialog when a template folder is selected in the navigator. The preview button will show you the Preview Code dialog when a template is selected in the navigator. The configure button will show the Configuration dialog when clicked.

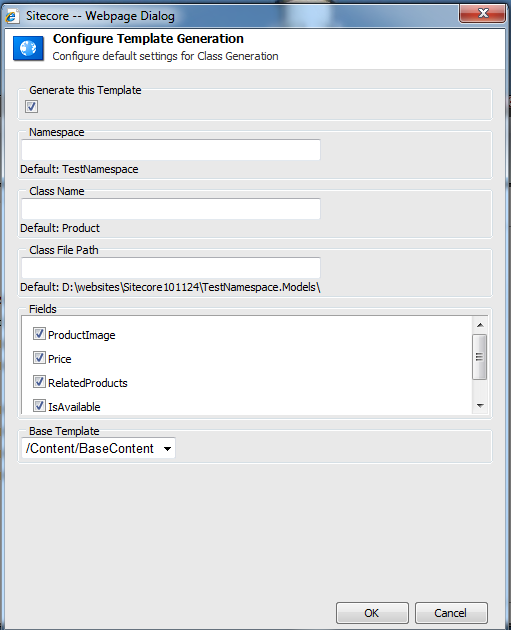
## Edit Folder

Click on any Template Folder then click the Edit button in the menu bar.   


1. Namespace – This field lets you set a namespace corresponding to this folder. Ideally you would set this only at your top most template folders (right under /sitecore/templates). Descendent templates and template folders would automatically figure out their namespacing from this folder. Ex. If /templates/Content/ has its namespace set to “Sitecore101124.Models.Content”, then the template /templates/Content/Products/Book would determine its namespace to be “Sitecore101124.Models.Content.Products” without any further configuration.  
   If a template can’t find any Parent template folders with a Namespace set, it will use the default Namespace from the Configuration Dialog.
2. File Path for this Folder – This field lets you map a template folder to a particular folder on the filesystem (using the [slnpath] replacement). Ideally you would set this only at your top most template folders (right under /sitecore/templats). Descendent templates and template folders would automatically figure out their filesystem mappings from this template folder. Ex. If /templates/Content/ has its filepath set to “[slnpath]\Sitecore101124.Models\Content\”, then the template /templates/Content/Products/Book would determine its filepath to be “[slnpath]\Sitecore101124.Models\Content\Products\Book.cs” without any further configuration.  
   If a template can’t find any Parent template folders with a Filepath set, it will use the default Filepath from the Configuration Dialog.

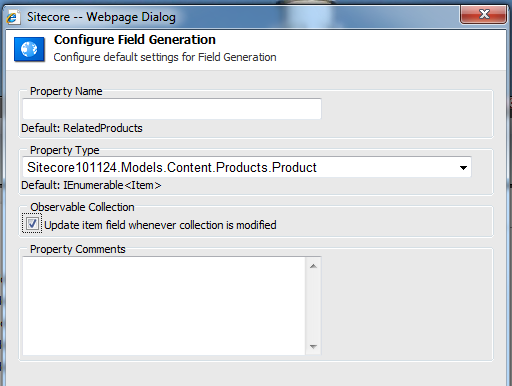
Long story short, make sure all your generated classes are under at least one Template Folder with these Namespace and Filesystem mappings set.

## Edit Template

Click on any template you wish to generate in the navigator, then click the Edit button in the menu bar.   


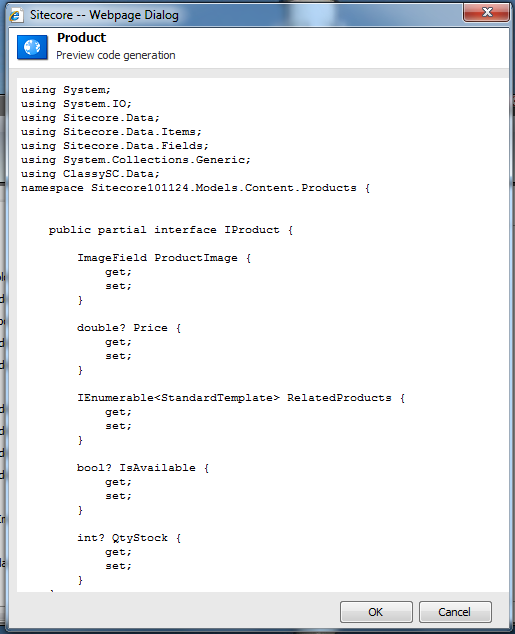
1. **Generate this Template – Check this to indicate this template should be generated. This is the first thing to do when editing a class.**
2. Namespace is where you can override the default namespace setting for this template only. Otherwise leave blank.
3. Class Name is where you can override the class name of this template. Leave it blank to default to using the name of the template.
4. Class File Path is where you can override the file path for this template only. Leave blank to use the default path indicated.
5. The fields list is where you can check off each field you want to become a property of your class. Typically you will check all fields.
6. BaseTemplate is where you can choose between this template’s existing base templates to indicate which to use as this class’s base class. Generated classes will inherit StandardTemplate (by default if blank), or the class of the selected template. If your template has several base templates, choose the one that is conceptually the most appropriate base class. The other inherited templates will be implemented as interfaces, which will be discussed later in this document.

## Edit Field

Select a Field in the navigator and click Edit from the tool bar.  


1. Property Name – This is where you can override the property name of this field. Leaving it blank will default to the Field name.
2. Property Type – This is where you can override what class type this property is cast as. **This option is only available for field types that link to other items (link, treelist, etc)**. If the items referenced by this field also have their own class type, you can select it from this dropdown of generated class types. If the referenced items are of mixed templates/types, always choose the nearest common base class. (StandardTemplate being the lowest common class to all generated classes) Leaving this blank will default the output to Item or IEnumerable<Item> depending on the field type.
3. Observable Collection – **This option is only available for List type fields (multilist, treelist).** Normally list fields will become IEnumerable<Item> or IEnumerable<MyClass> if you’ve set the Property Type. Checking this box will change the Property to ClassyCollection<Item> or ClassyCollection<MyClass>. The ClassyCollection<T> is an observable IEnumerable, IList collection with event hooks to automatically update the ID list for its corresponding Field whenever the collection is modified by adding, removing, clearing, etc. The basic IEnumerable<T> property does not automatically update its internal field upon modifications. Check this if you’re using ClassySC items for writing (such as an import script) rather than just for reading properties.
4. Property Comments – You can specify code comments to be inserted in the generated property code.

## Preview Code

Select a generated Template in the navigator and click Preview.  


This will preview the code output for this class so you can make fine adjustments before writing to the filesystem.

## Generate

Clicking the generate button will generate the code of any templates checked in the list on the right.

# Using Generated Classes

## Setting up your .sln

Ideally the models would be generated into a project that is mostly or completely separated from presentation and business. For this demonstration we’ll call this project **Sitecore110124.Models**.

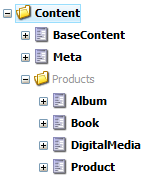
**IMPORTANT**  
Any project that wants to use Generated classes has to reference **ClassySC.Data.dll** in the csproj and also add **using ClassySC.Data;** in the code files. This **using** lets the file both recognize the StandardTemplate class and use the extension methods associated to Item and IEnumerable<Item>.

So pretend we have our Models project which references ClassySC.Data.dll and contains the generated classes. We have the business project, which references ClassySC.Data.dll & Models project. Finally we have the Website project, which references ClassySC.Data.dll, the Business project and the Models Project.

## Generated Class Architecture

The base class for generated classes is ClassySC.Data.StandardTemplate. This class derives from Sitecore’s commonly used CustomItemBase, which is basically just a wrapper for the Item class and always exposes the Item it wraps with the InnerItem property. The constructor for StandardTemplate or any other generated class is:   
public StandardTemplate(Item innerItem)

StandardTemplate has dozens of Get and Set methods for various property types that the generated Properties will use for accessing the fields. The Properties generated for our templates use these Get and Set methods along with the field’s ID.

For the generated example, we’ll use this template setup  
  
BaseContent has Meta as a section template. Product has BaseContent as a base template. Book and Album each have Product as a base template along with DigitalVersion as a section template.

Generating the these classes give us the following classes:

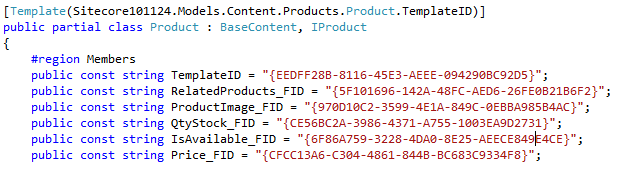
public partial class BaseContent : StandardTemplate, IBaseContent, IMeta  
public partial class Product : BaseContent, IProduct  
public partial class Album : Product, IAlbum, IDigitalVersion  
public partial class Book : Product, IBook, IDigitalVersion  
public partial class DigitalVersion : StandardTemplate, IDigitalVersion  
public partial class Meta : StandardTemplate, IMeta

Classes always inherit an interface of themselves, plus either StandardTemplate or the BaseTemplate you selected. The most important base template of any generated template is used as the base class. Inherited templates that are tacked on just to add a section or a useful field are more like ”section templates”. Section templates will still get their own class. No other classes will inherit them as a baseclass, but they will implement them as an interface. This concept is what reconciles the multiple inheritance capabilities of Sitecore templates with the single inheritance, multiple interface capabilities of C#. **Long story short: If your template has multiple inherited templates, choose the most important one as the Base Template for class generation.** The Album template has both Product and DigitalVersion as base templates, but conceptually Album is a Product so Product is the base class for Album.

As for the Fields, each generated field becomes a Property of the class (and of the class’s interface) with a get and set that handles returning or setting objects from the sitecore fields.

## Constants

As a class is generated, the Template ID and Field IDs are represented as constant strings within the class. This allows Attributes to be set on the class and properties to help the framework with mapping, and also allows you to use these IDs easily in the rest of your application such as creating a query.

 To access the ID of a template from any other file, you can simply use [ClassName].TemplateID

To access the ID of a field from any other file, you can simply use [ClassName].[FieldName]\_FID

An example of using them throughout your project would be in constructing queries:  
public const string GetProductsQuery = "fast://\*[@@templateid='" + Album.TemplateID + "' or @@templateid='" + Book.TemplateID + "']";

## Attributes

Generated Classes use two custom attributes to help the framework.

[Template(Sitecore101124.Models.Content.Products.Product.TemplateID)]

public partial class Product : BaseContent, IProduct {

The Template attribute associates the TemplateID with the class. This is used by the ClassManager to resolve the correct type for an unknown Item based on its TemplateID. During startup ClassManager will go through all generated classes caching the Template ID to Class Type relations AND confirming that each Template attribute corresponds to a template in the database. **This will throw errors if your classes don’t match your templates! Don’t forget to publish your template changes to web.**

[Field(Sitecore101124.Models.Content.BaseContent.Title\_FID)]

public virtual string Title {

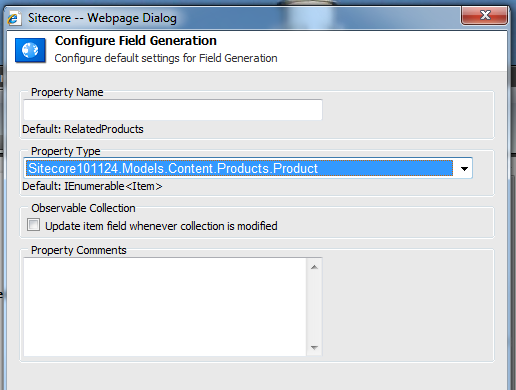
The Field Attribute associates a Field ID to each property that gets generated. Like the Templates ClassManager will check these at application startup to make sure each field actually exists and the code matches the database. **This will throw errors if your classes don’t match your templates! Don’t forget to publish your template changes to web.**

## Property Mappings

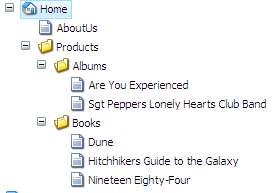
The generator has an intuitive mapping of Field types to c# classes for each property, but also allows you to override these.

The default mappings look like this.

|  |  |
| --- | --- |
| From | To |
| Single-Line Text Multi-Line Text Rich Text Password | string |
| Date Datetime | DateTime? |
| DropLink DropTree  Grouped DropLink | Item  Or YourClass |
| CheckList  Droplist  Grouped Droplist  Multilist  Treelist | IEnumerable<Item> Or IEnumerable<YourClass> |
| Number | double? |
| Integer | int? |
| Checkbox | bool? |
| All other Sitecore field types | Sitecore.Data.Fields.[FieldType] |

For Link or List fields they can return basic Item and IEnumerable<Item>, **or** you can have them map to other Generated classes. For this example we’ll set the Property Class Type in the Edit Field dialog for the RelatedProduct’s field of the Product template to Product.  
  
Don’t worry that the Property could be returning lists of either Book or Album. Casting to Product is exactly what we want and will discuss later in this document.

## Let’s use the classes!

Here’s the test tree we’ll use for this example.  
  
**/About Us** is BaseContent  
**/Products/Albums/\*** are Albums  
**/Products/Books/\*** are Books

In your business logic or presentation control, add   
using ClassySC.Data;   
Then add the Namespace you chose for your classes  
using Sitecore110124.Models.Content;  
using Sitecore110124.Models.Content.Products;

ClassySC.Data adds a few extension methods to Item and IEnumerable<Item>.

Item testItem1 = db.GetItem("/sitecore/content/home/AboutUs");

BaseContent aboutUsPage = testItem1.**ToClass<BaseContent>()**;

Response.Write(aboutUsPage.Title); // "About Us"

Database methods are extended to also cast directly like:

BaseContent aboutUsPage = db.GetItem**<BaseContent>**("/sitecore/content/home/AboutUs");

Response.Write(aboutUsPage.Title); // "About Us"

The **ToClass<T>** Extension will take the Item or IEnumerable<Item> that you feed it and for each item, find a generated type that matches that Template ID, construct the object in the appropriate class for that Template, then it casts that result as <T>.

IEnumerable<Book> books = db.SelectItems<Book>("/sitecore/content/home/Products/Books/\*");

IEnumerable<Album> albums = db.SelectItems**<Album>**("/sitecore/content/home/Products/Albums/\*");

Here we see class casting collections into classes where we know the results are a single template. What happens if we want to query a mixed result?

IEnumerable<Product> products = db.SelectItems**<Product>**("/sitecore/content/home/Products//\*");

When you have an IEnumerable<Item> with mixed templates, cast them into the highest base class that all the results will share. In

foreach (Product product in products)

{

if (product is IBook)

{

Response.Write(product.GetType().FullName + " - " + product.Title + " by " + (product as IBook).Author + "<br/>");

}

else if (product is IAlbum)

{

Response.Write(product.GetType().FullName + " - " + product.Title + " by " + (product as IAlbum).Artist + " " + "<br/>");

}

}

Sitecore101124.Models.Content.Products.Album - Are You Experienced by Jimi Hendrix   
Sitecore101124.Models.Content.Products.Album - Sgt. Pepper's Lonely Hearts Club Band by The Beatles   
Sitecore101124.Models.Content.Products.Book - Dune by Frank Herbert  
Sitecore101124.Models.Content.Products.Book - The Hitchhiker's Guide to the Galaxy by Douglas Adams  
Sitecore101124.Models.Content.Products.Book - Nineteen Eighty-Four by George Orwell

While cast our result to Product, the actual types are still Book or Album because they were constructed with their specific class before being cast. This lets us test an object for the interface of the template we’re looking for in a mixed result, and then cast to that interface in order to get to those fields. For this to work correctly you must cast to a base class shared by all the results. If a BaseContent page were mixed in those results, we’d have to cast to BaseContent since BaseContent, Product, Book, and Album all share BaseContent as their highest common base class. The worst case for a mixed result is StandardTemplate, since **all** items share this.

## Editing an Item

Generated classes are essentially wrapping an Item. The item is always available via the .InnerItem property.

using (new EditContext(product.InnerItem))

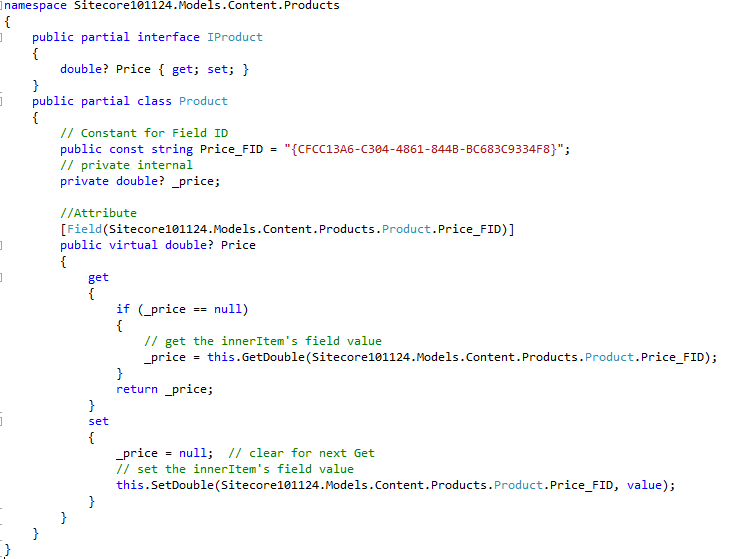
{

product.QtyStock -= 1;

}

## Extending the Generated Classes

Occasionally the generator won’t be able cover every possible case. If you need to add your own Properties or methods follow these directions.

1. If you’re making your own Property for a field in a template, make sure the field is unchecked in the template’s generation settings from the Edit Template dialog in the ClassySC Manager so that it isn’t generated.
2. Create a separate partial file for the class (ex. Product.Partial.cs) with the same namespace, add the partial class and, if appropriate, the partial interface.  
   public partial interface IProduct  
   { }  
   public partial class Product  
   { }
3. Add the property back similar to how the generator does, but with your own custom code  
   
4. You can also add Methods to perform calculations (although that belongs more in Business than Models). You can add Properties that aren’t based on field values. You can add Properties that return specific sub-items of the current item (use for template structures that require multiple sub items) ex:   
   return InnerItem.Axes.SelectItems<T>("./\*[@@templateid='{Another ID}'");

Email [mhovany@reveregroup.com](mailto:mhovany@reveregroup.com) for any questions.