Cuyahoga Module Development Guide

Table of Contents

Introduction........1
Requirements........1
Setting up the project........2
The simplest module possible........3
    The module controller........3
    The display user control........4
    The cuyahoga_moduletype table........5
    Running the first module........6
Display dynamic content with the module........7
    The domain........7
    The database........9
    Mapping the class to the database........9
    Module configuration........10
    Displaying objects........10
Module administration........11
Make the module searchable........11
Create an RSS feed for the module........11
Language resources........11
Deployment........11

Introduction

Cuyahoga is a web site framework that has functionality to manage the site structure, templates and authorization. All content management is done by separate modules. This allows for easily adding new functionality without having to change the framework itself.

This document explains all the aspects of building your own modules and should contain enough information to get things started. For specific questions or remarks, visit the modules forum at http://www.cuyahoga-project.org/home/forum.aspx?g=topics&f=4.

The important thing: building custom modules for Cuyahoga is fun! It is not difficult to get results fast and it's a great opportunity to learn about web frameworks and tools like NHibernate or dotLucene.

Requirements

We're assuming that Visual Studio .NET 2003 is used to create the modules. Of course, NAnt users can also build modules, but the majority of .NET developers use VS.NET and it makes development a little bit easier. Module developers need to have some experience developing ASP.NET pages and user controls.

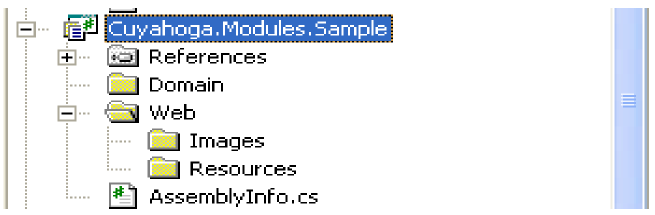
To get things started, you'll need a working version of Cuyahoga. This guide is based on the 0.9 release. It doesn't matter which database platform you choose. Modules that are developed with these guidelines should be database independent.

The language used in all examples here is C#, but you're not restricted to use it. In fact, any .NET language can be used to build modules.

Setting up the project

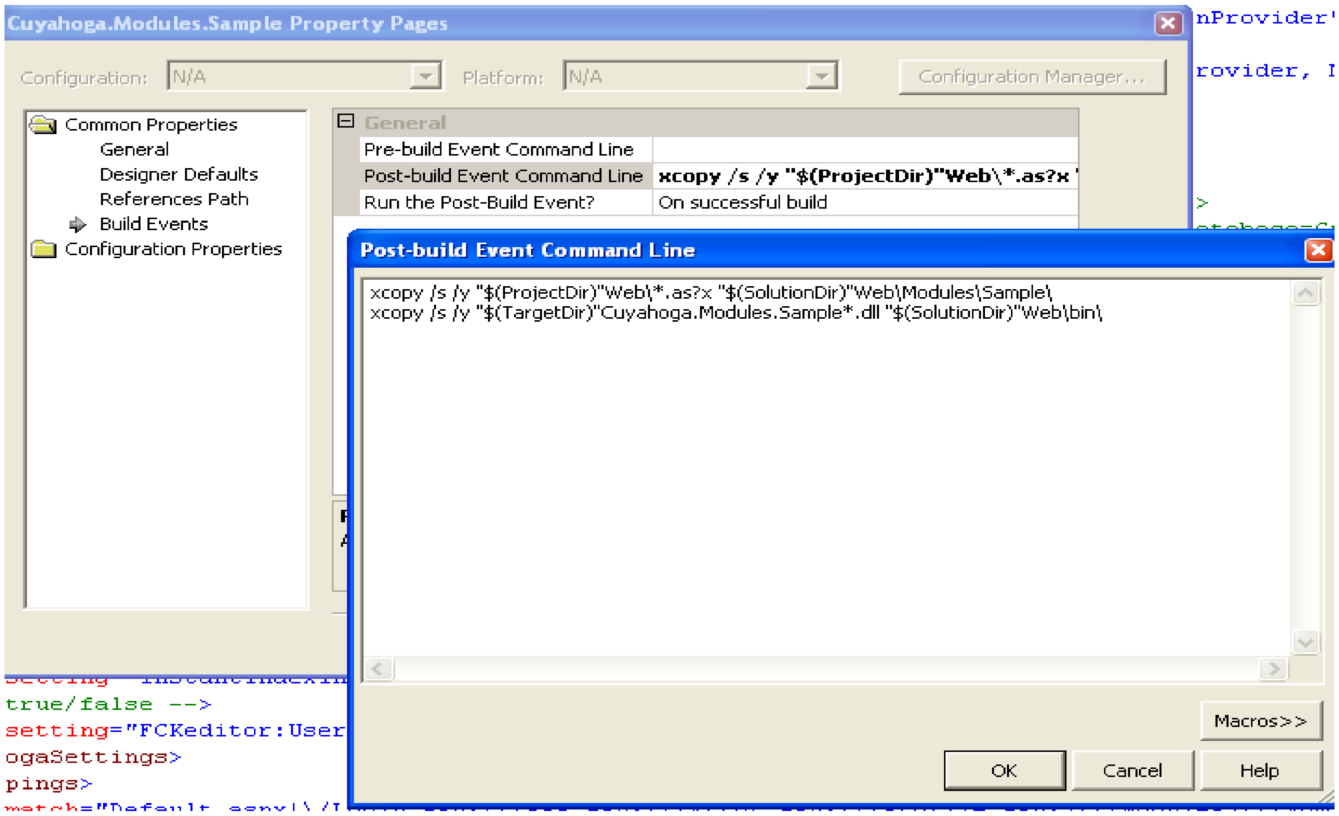
It's recommended to create one VS.NET project (which creates one assembly) for every module. In frameworks like DotNetNuke or Rainbow these are called 'Private Assemblies'. Don't add any new modules to the core modules project (StaticHtml, Articles etc) because this will get you in trouble with upgrading to a new Cuyahoga version.

Choose 'Class Library' as the project type. The initial directory structure for a module should look something like this:



There are no restrictions for a specific directory structure. If the module is going to be very small, you may as well put everything in one directory.

To enable debugging, it is required that the module files are copied into the Cuyahoga/Web/bin and Cuyahoga/Web/Modules directory before debugging. This can be accomplished with a post-build event:



General	
Pre-build Event Command Line	
Post-build Event Command Line	xcopy /s /y "$(ProjectDir)"Web*.as?x
Run the Post-Build Event?	On successful build

```
xcopy /s /y "$(ProjectDir)"Web\*.as?x "$(SolutionDir)"Web\Modules\Sample\
xcopy /s /y "$(TargetDir)"Cuyahoga.Modules.Sample*.dll "$(SolutionDir)"Web\bin\
```

The above event copies the resulting assembly and the .aspx and .ascx files to the right directories.

The simplest module possible

It takes only three steps to create a first module:

- Create a class that serves as a module controller, usually called *ModuleName*Module.cs.
- Create an .ascx user control that displays module content.
- Add a record to the cuyahoga_moduletype table so that Cuyahoga knows how to create the module instance.

The module controller

This is the central class of the module. It's not a controller like in a pure MVC sense but merely a class that contains or delegates functionality that doesn't belong in the .ascx controls or .aspx pages that are in the module. The module controller is a subclass of Cuyahoga.Core.Domain.ModuleBase and requires a constructor with one argument of the 'Section' type.

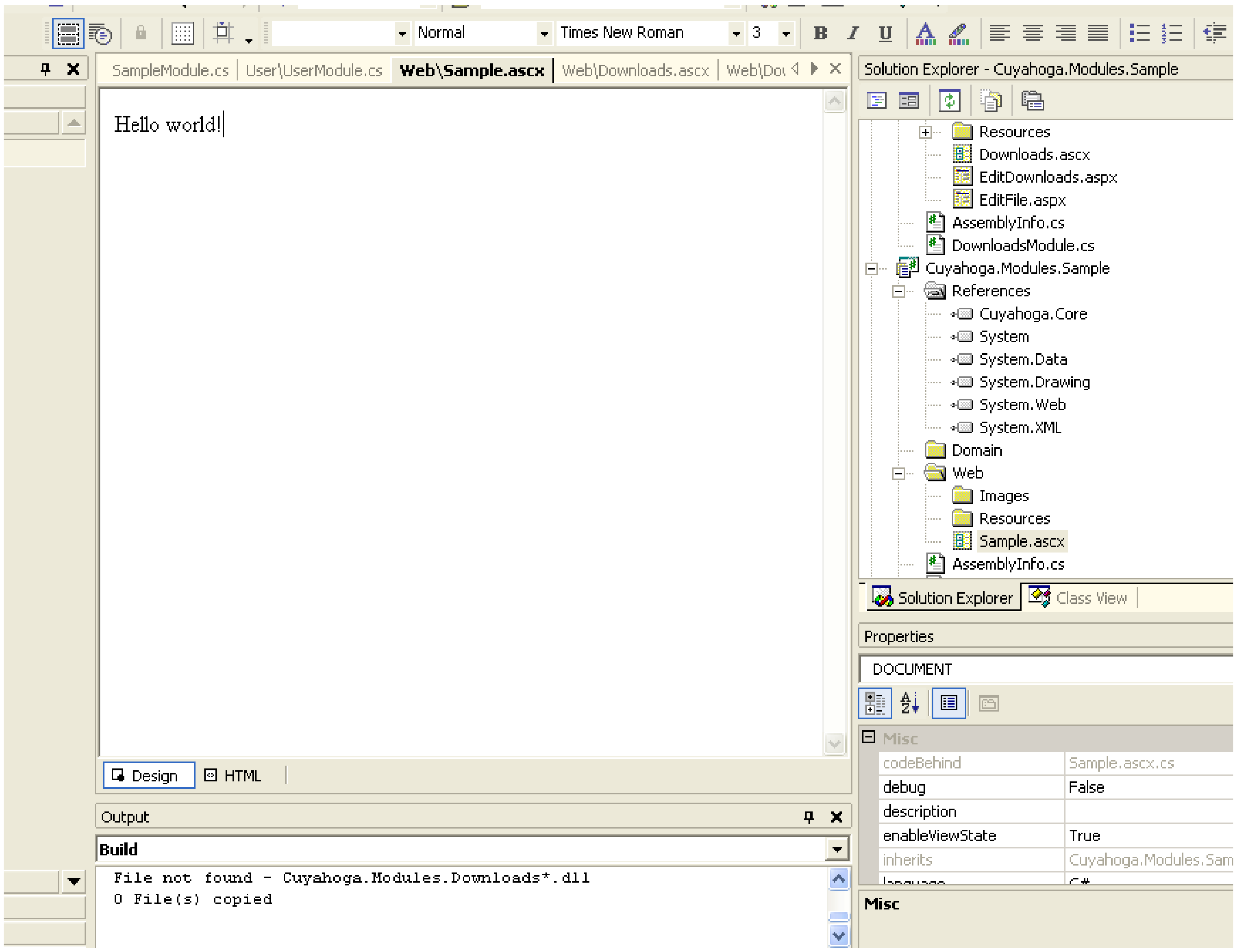
```
using System;

using Cuyahoga.Core.Domain;

namespace Cuyahoga.Modules.Sample
{
	/// <summary>
	/// The module controller class.
	/// </summary>
	public class SampleModule : ModuleBase
	{
		public SampleModule(Section section) : base(section)
```

```
			{
				//
				// TODO: Add constructor logic here
				//
			}
		}
}
```

In this very first sample there is nothing else to do for this class, so we leave it like above.
Note: to use the Cuyahoga.Core.Domain classes you have to add a reference to the Cuyahoga.Core project.

The display user control

Cuyahoga 'injects' templates with module user controls to display content. The content in this sample is the infamous sentence 'Hello world'. Just create a new User Control in the Web directory of the module and add the line.



codeBehind	Sample.ascx.cs
debug	False
description	
enableViewState	True
inherits	Cuyahoga.Modules.Sam

```
File not found - Cuyahoga.Modules.Downloads*.dll
0 File(s) copied
```

The code behind class of the display control has to inherit from Cuyahoga.Web.UI.BaseModuleControl:

```
namespace Cuyahoga.Modules.Sample.Web
{
	using System;
	using System.Data;
	using System.Drawing;
	using System.Web;
	using System.Web.UI.WebControls;
	using System.Web.UI.HtmlControls;

	using Cuyahoga.Web.UI;

	/// <summary>
	///		The sample display control.
	/// </summary>
	public class Sample : BaseModuleControl
	{

		private void Page_Load(object sender, System.EventArgs e)
		{
			// Put user code to initialize the page here
		}

		#region Web Form Designer generated code
		override protected void OnInit(EventArgs e)
		{
			//
			// CODEGEN: This call is required by the ASP.NET Web Form Designer.
			//
			InitializeComponent();
			base.OnInit(e);
		}

		/// <summary>
		///		Required method for Designer support - do not modify
		///		the contents of this method with the code editor.
		/// </summary>
		private void InitializeComponent()
		{
			this.Load += new System.EventHandler(this.Page_Load);
		}
		#endregion
	}
}
```

Note: to use the Cuyahoga.Web.UI.BaseModuleControl class you have to add a reference to the Cuyahoga.Web project.

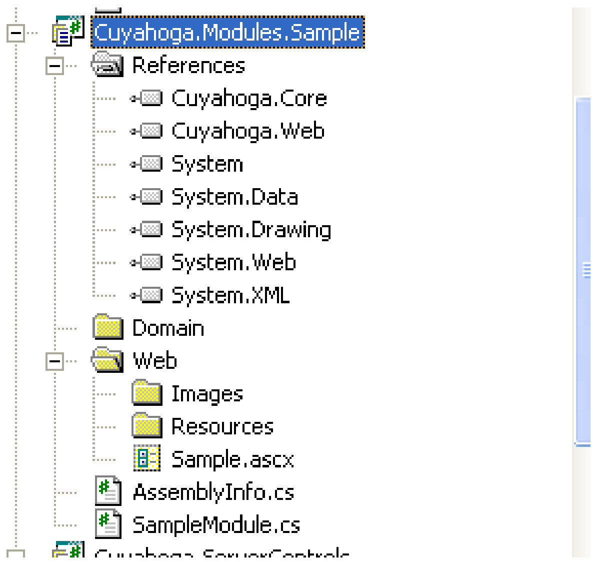
The cuyahoga_moduletype table

All installed modules are registered in the cuyahoga_moduletype table. Enter the following values:

moduletypeid	*auto-generated*
name	Sample
assemblyname	Cuyahoga.Modules.Sample
classname	Cuyahoga.Modules.Sample.SampleModule
path	Modules/Sample/Sample.ascx
editpath	<null>
inserttimestamp	*auto-generated*
updatetimestamp	*the current date*

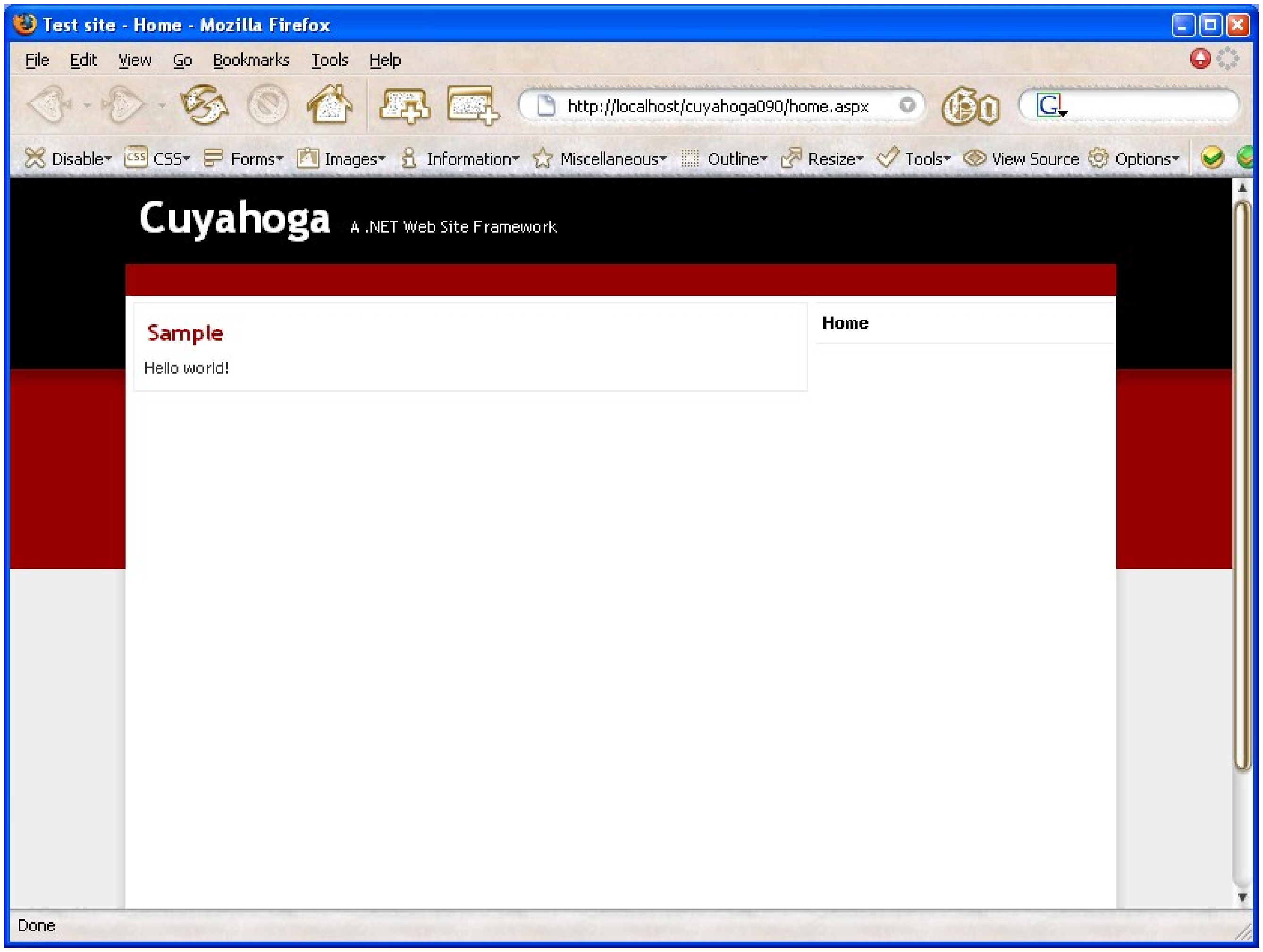
Running the first module

The module project in VS.NET should contain the following files and references:



```
Cuyahoga.Modules.Sample
    References
        Cuyahoga.Core
        Cuyahoga.Web
        System
        System.Data
        System.Drawing
        System.Web
        System.XML
    Domain
    Web
        Images
        Resources
        Sample.ascx
    AssemblyInfo.cs
    SampleModule.cs
```

Now run Cuyahoga and try to create a Section in the site administration with the newly created module. If everything is alright, you should see the first module in action:


Test site - Home - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
http://localhost/cuyahoga090/home.aspx
Disable CSS Forms Images Information Miscellaneous Outline Resize Tools View Source Options
Cuyahoga
A .NET Web Site Framework
Sample
Hello world!
Home
Done

Display dynamic content with the module

When developing custom modules for Cuyahoga you can choose any data-access strategy you want. However, Cuyahoga uses NHibernate for persistence and it's very convenient to also use it for the modules. It takes away the boring work and gives you database independency.

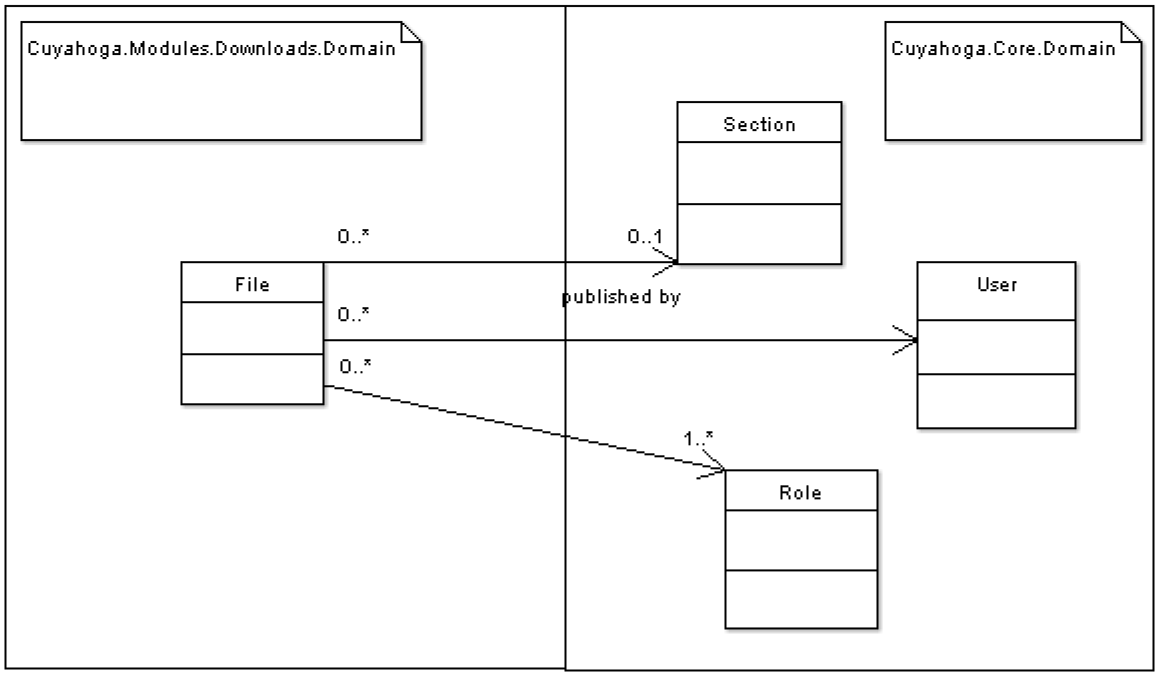
The samples in this chapter are from the Cuyahoga.Modules.Downloads module that is included with the Cuyahoga sources. This module is very basic. Check the Articles module for a little more advanced example.

The domain

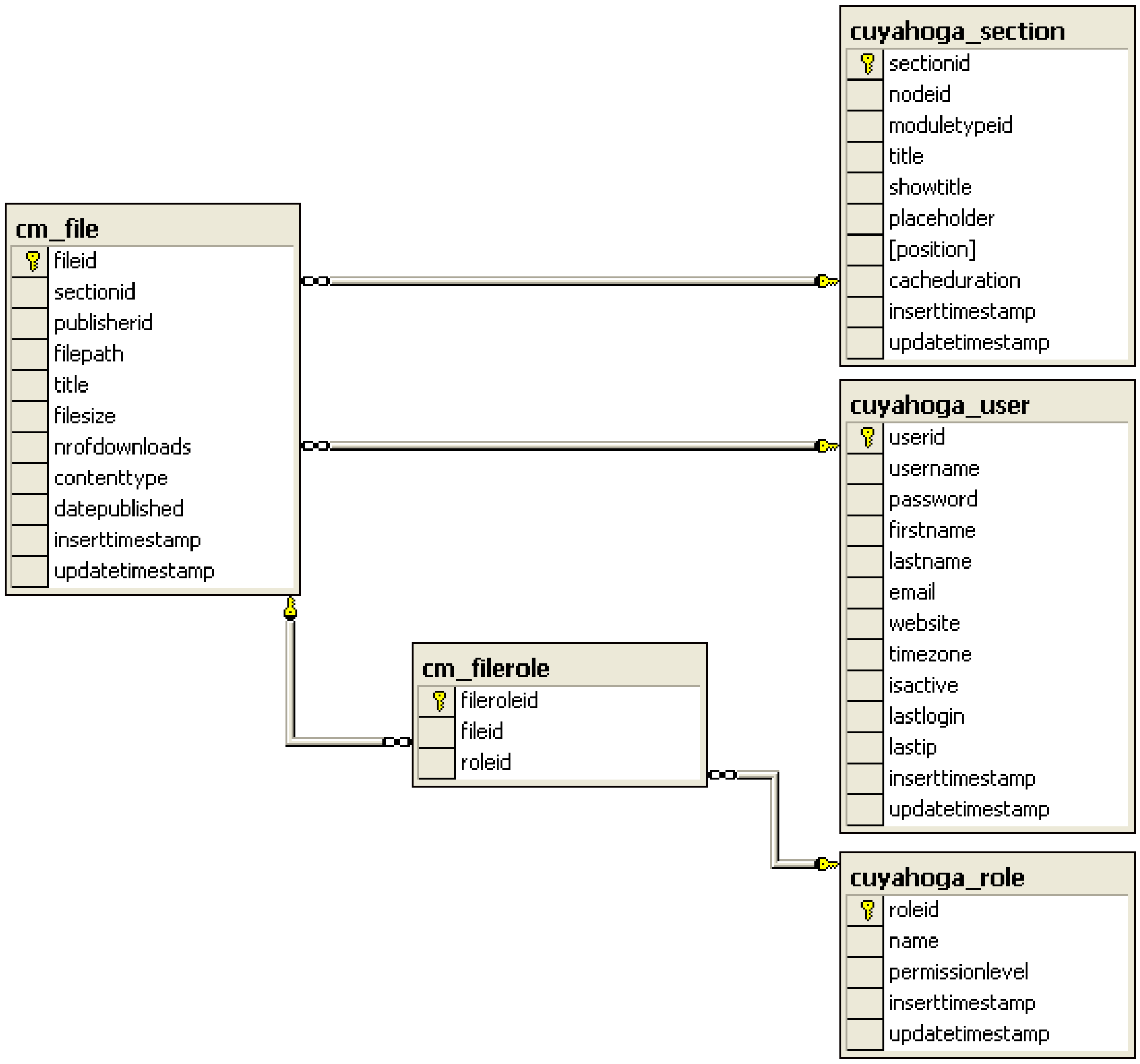
We'll start with the domain (business logic) first. This example has almost no business logic, so people might wonder why we need such a class at all. Well, the answer is simple: for consistency reasons (when having larger modules, the business logic can get more complicated) and because we have a nice infrastructure that works well with this kind of solution.

The requirements for the Downloads module are simple: users can download files and access to specific files can sometimes be restricted to one or more specific roles.

This results in only one domain class: File. This class contains the meta data of the physical files and some methods to perform authorization checks. Note that it's fine to reference Cuyahoga core classes in the module domain (see the references to Section, User and Role).



The database



Mapping the class to the database

NHibernate needs to know how classes are mapped to database tables and uses a mapping file for this. Most of the times, there is one mapping file for one class. In this case we have the mapping file File.hbm.xml:

```
<?xml version="1.0" encoding="utf-8" ?>
<hibernate-mapping xmlns="urn:nhibernate-mapping-2.0">
        <class name="Cuyahoga.Modules.Downloads.Domain.File, Cuyahoga.Modules.Downloads"
                table="cm_file">

                <id name="Id" column="fileid" type="Int32" unsaved-value="-1">
                        <generator class="native">
                                <param name="sequence">cm_file_fileid_seq</param>
                        </generator>
                </id>

                <timestamp name="UpdateTimestamp" column="updatetimestamp" unsaved-value="1/1/0001" />
```

```
               <property name="FilePath" column="filepath" type="String" length="255" />
               <property name="Title" column="title" type="String" length="100" not-null="false" />
               <property name="Size" column="filesize" type="Int32" />
               <property name="NrOfDownloads" column="nrofdownloads" type="Int32" />
               <property name="ContentType" column="contenttype" type="String" length="50" />
               <property name="DatePublished" column="datepublished" type="DateTime" />

               <many-to-one name="Section" class="Cuyahoga.Core.Domain.Section, Cuyahoga.Core"
                      column="sectionid" cascade="none" />
               <many-to-one name="Publisher" class="Cuyahoga.Core.Domain.User, Cuyahoga.Core"
                      column="publisherid" cascade="none" />

               <bag name="AllowedRoles" cascade="none" lazy="true" table="cm_filerole">
                      <key column="fileid" />
                      <many-to-many class="Cuyahoga.Core.Domain.Role, Cuyahoga.Core" column="roleid" />
               </bag>

       </class>
</hibernate-mapping>
```

This mapping file resides in the same directory as the File.cs file has to marked as Embedded Resource in VS.NET.

Module configuration

Before we can go to the user control that displays a list of files there is one important step to be taken: make sure that Cuyahoga knows about the File class and how to persist it. This is done by registering the class in the constructor of the module controller (DownloadsModule.cs).

```
public DownloadsModule(Section section) : base(section)
{
       SessionFactory sf = SessionFactory.GetInstance();
       // Register classes that are used by the DownloadsModule
       sf.RegisterPersistentClass(typeof(Cuyahoga.Modules.Downloads.Domain.File));

       base.SessionFactoryRebuilt = sf.Rebuild();

       [...other constructor stuff...]
}
```

SessionFactory in the code above is a singleton wrapper around the NHibernate configuration. There is one unique instance of this class during the lifetime of the application, so a class only has to be registered at the first request (*SessionFactory* handles this internally).
The *SessionFactory.RegisterPersistentClass()* method integrates the mapping of the module class with the rest of Cuyahoga's mappings. After adding a class, calling the *SessionFactory.Rebuild()* method is required, because Cuyahoga needs to be notified when the configuration has changed during a request. Disclaimer: this is not the most elegant piece of code. Feel free to investigate the possibilities of improving this part.

Displaying objects

The Web/Downloads.ascx user control is responsible for displaying the content. It displays a list of files in a Repeater control. Just like the sample user control before, the code-behind class has to inherit from *Cuyahoga.Web.UI.BaseModuleControl.* By inheriting from this class, the code-behind page knows its module controller (DownloadsModule.cs) and can call methods to retrieve or store the objects. You can choose to implement the persistence logic in the module controller or, in large modules, delegate it to some kind of repository object.

An example: to show all files that belong to a specific section, the *DownloadsModule.GetAllFiles()* method is called and the resulting list of objects is bound to the Repeater:

```
/// <summary>
/// Retrieve the meta-information of all files that belong to this module.
/// </summary>
/// <returns></returns>
public IList GetAllFiles()
{
	string hql = "from File f where f.Section.Id = :sectionId order by f.DatePublished desc";
	IQuery q = base.NHSession.CreateQuery(hql);
	q.SetInt32("sectionId", base.Section.Id);
	try
	{
		return q.List();
	}
	catch (Exception ex)
	{
		throw new Exception("Unable to get Files for section: " + base.Section.Title, ex);
	}
}
```

The GetAllFiles() method uses an NHibernate IQuery object to retrieve the File objects from the database. See the NHibernate docs for more details about the query language.

Module administration

Make the module searchable

Create an RSS feed for the module

Language resources

Deployment