

Cuyahoga Module Development Guide

Table of Contents

Version information.....	1
Introduction.....	1
Requirements.....	2
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.....	5
Display dynamic content with the module.....	6
The domain.....	6
The database.....	8
Mapping the class to the database.....	8
Module configuration.....	9
Displaying objects.....	9
Leveraging services via dependency injection.....	10
Using the PathInfo parameters to pass variables.....	10
Custom module settings.....	11
Module administration.....	11
Make the module searchable.....	12
Create an RSS feed for the module.....	12
Language resources.....	12
Inter-module communication.....	12
Deployment.....	13
Copy the files to the server.....	14
Database scripts for the module installer.....	14

Version information

The information in this document is based on the Cuyahoga 1.5.0 release.

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 Castle, NHibernate or

dotLucene.

Requirements

We're assuming that Visual Studio .NET 2005 is used to create the modules. You also need VS.NET 2005 SP1 or the separate download of the Web Application Project model (<http://webproject.scottgu.com/>). As an alternative, NAnt can also be used to build modules. All Cuyahoga modules have their own NAnt build scripts that can be used as an example.

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 1.5.0 release. It doesn't matter which database platform you choose. Modules that are developed with these guidelines should be database independent.

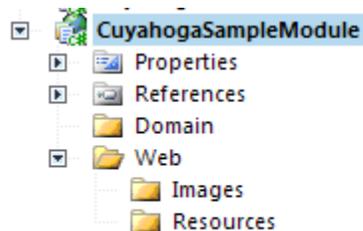
The language used in all examples here is C#, but you're not restricted to use it. In fact, any .NET language can be used to build modules.

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.

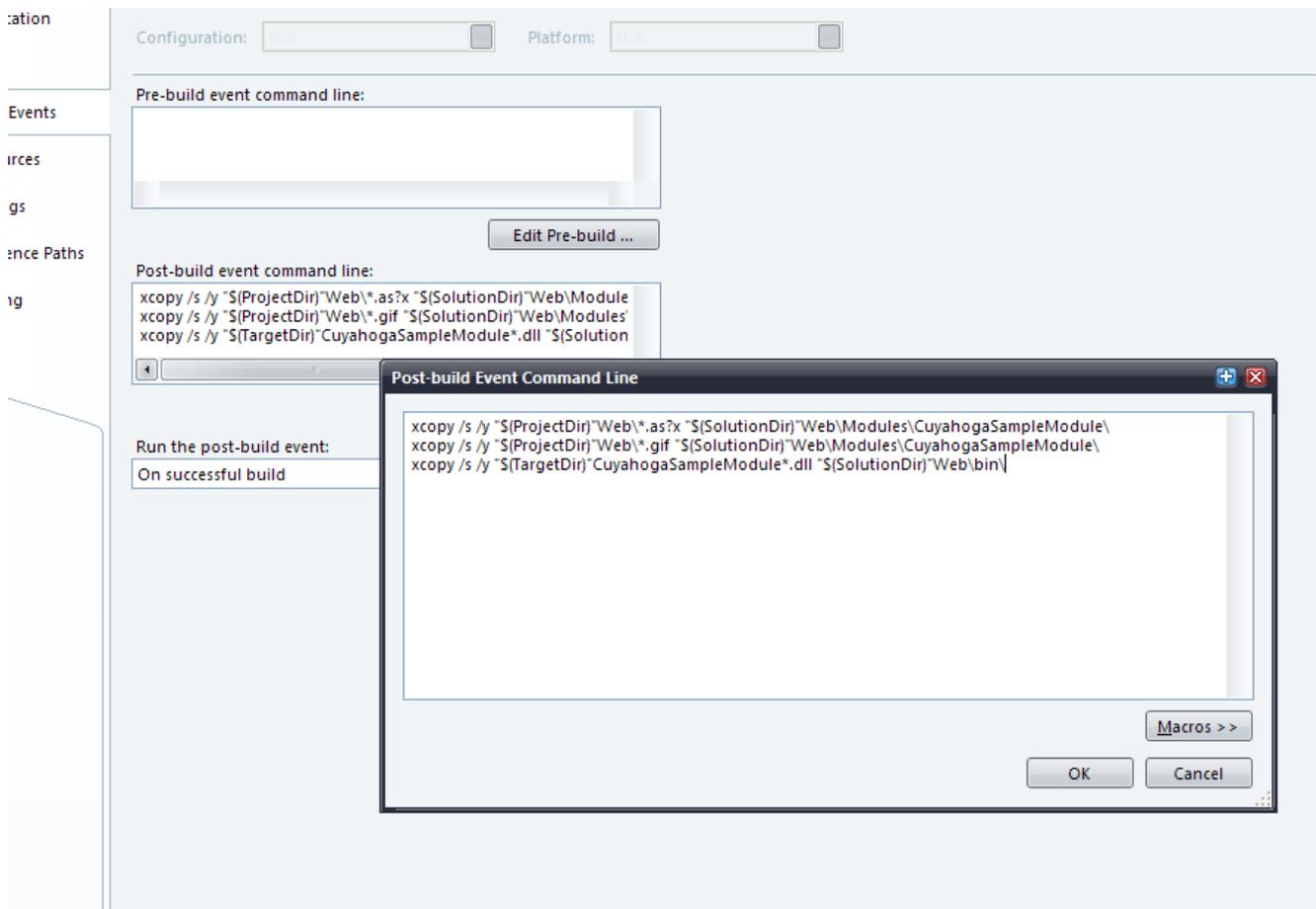
Choose 'ASP.NET Web Application' as the project type.

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:



The above event copies the resulting assembly and the .aspx, .ascx and .gif 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 *ModuleNameModule.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*.

```
using System;

using Cuyahoga.Core.Domain;

namespace Cuyahoga.Modules.Sample
{
```

```

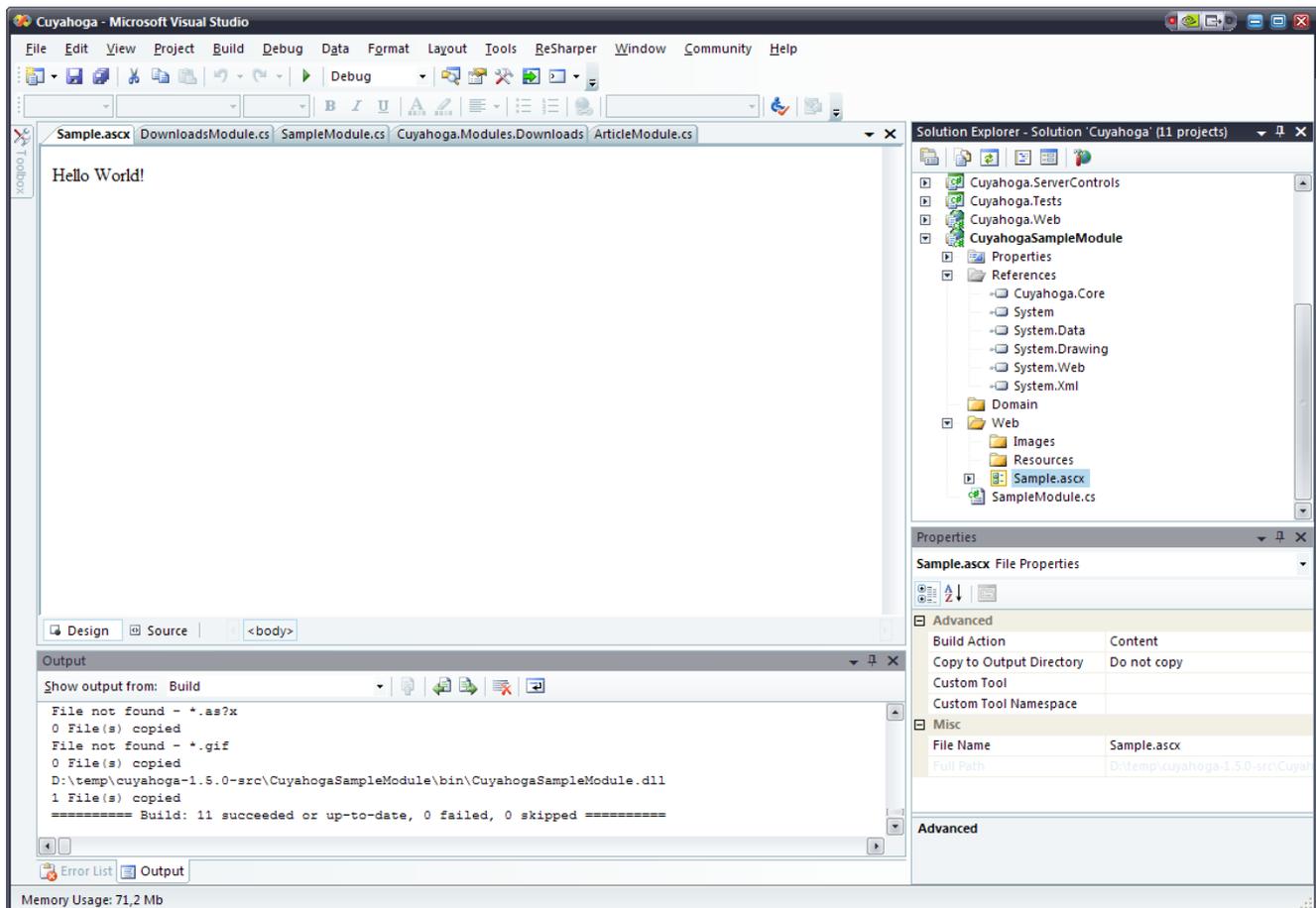
    /// <summary>
    /// The module controller class.
    /// </summary>
    public class SampleModule : ModuleBase
    {
        public SampleModule ()
        {
            //
            // TODO: Add constructor logic here
            //
        }
    }
}

```

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.



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

```
using System;
```

```

namespace CuyahogaSampleModule.Web
{
    /// <summary>
    /// The sample display control.
    /// </summary>
    public partial class Sample : Cuyahoga.Web.UI.BaseModuleControl
    {
        protected void Page_Load(object sender, EventArgs e)
        {

        }
    }
}

```

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

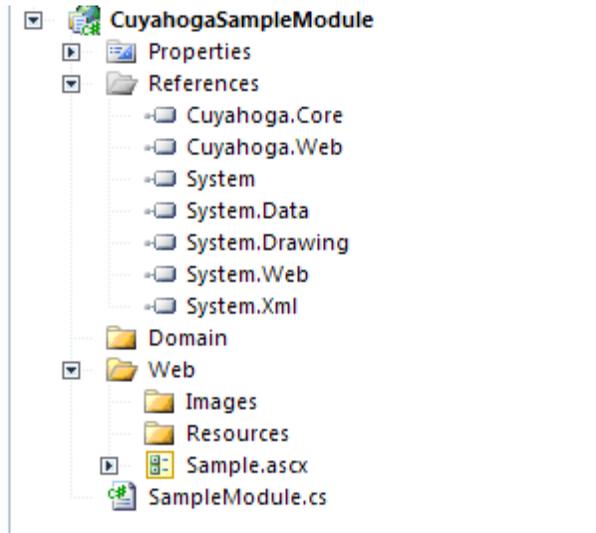
The *cuyahoga_moduletype* table

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

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

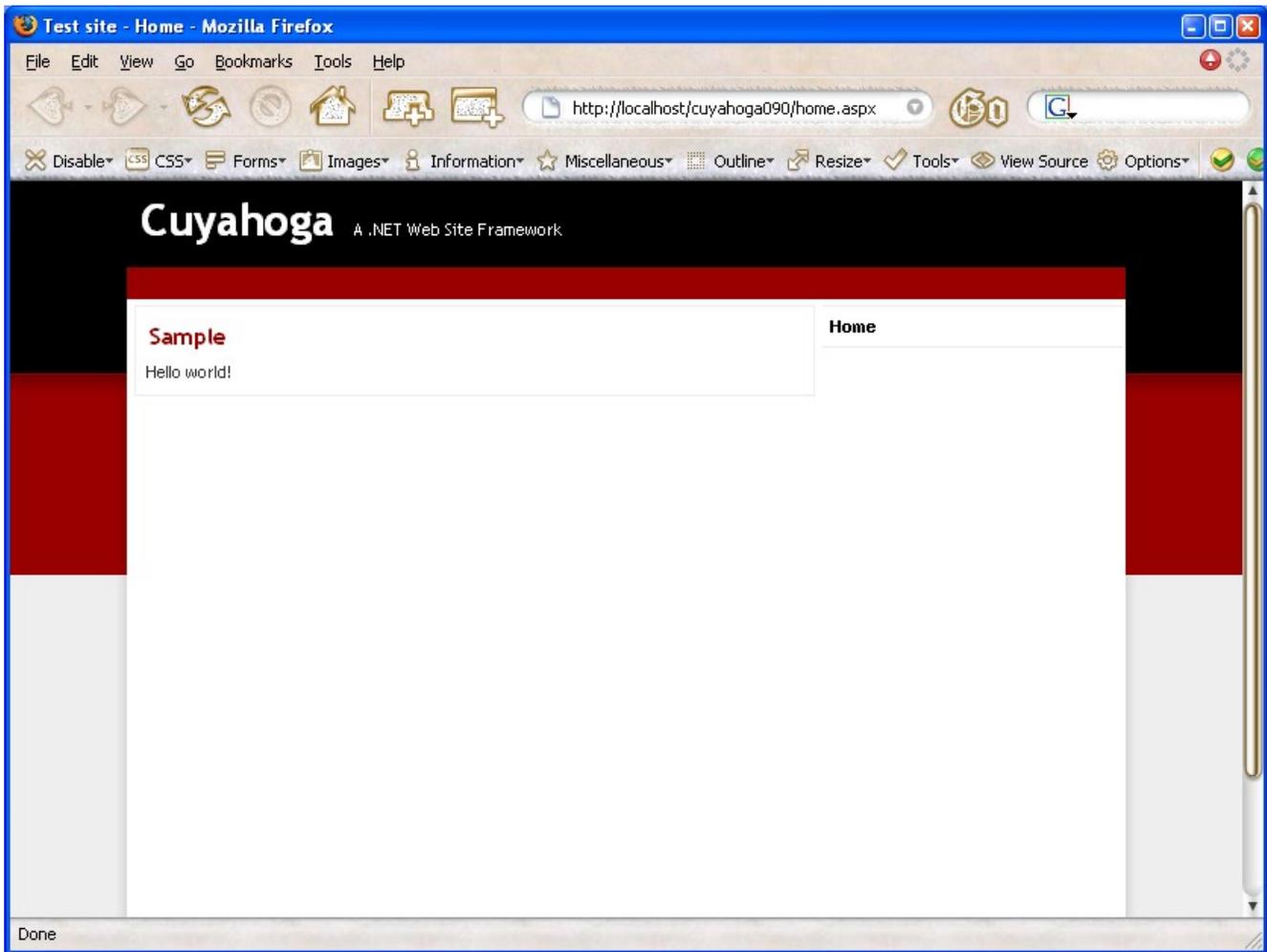
Running the first module

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



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:



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.

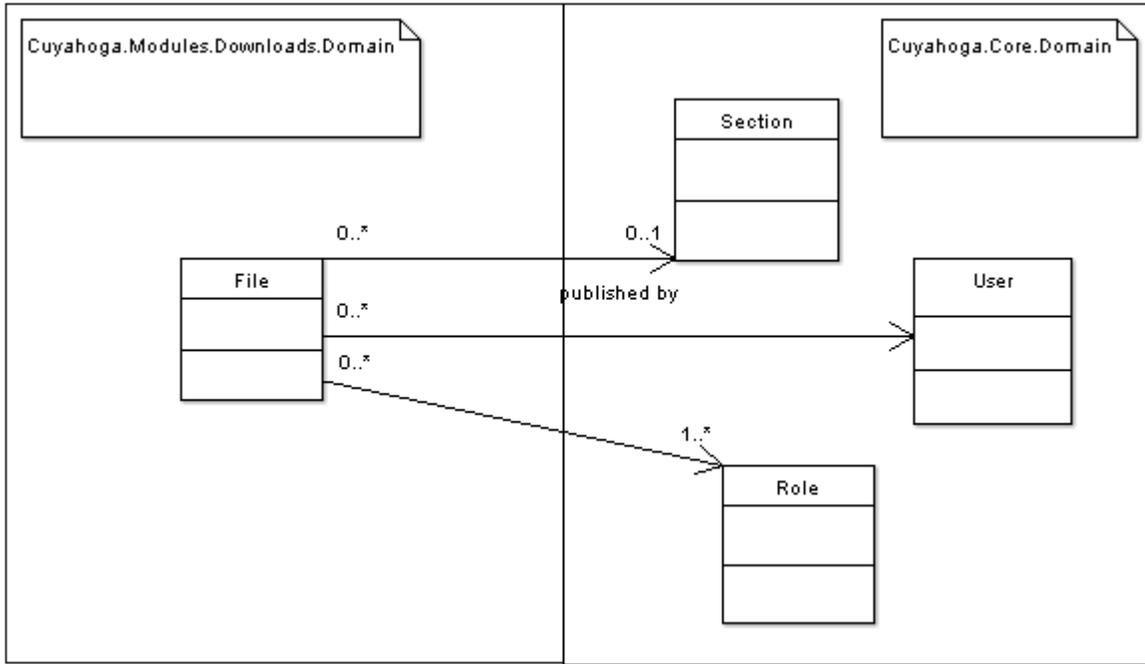
The domain

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.

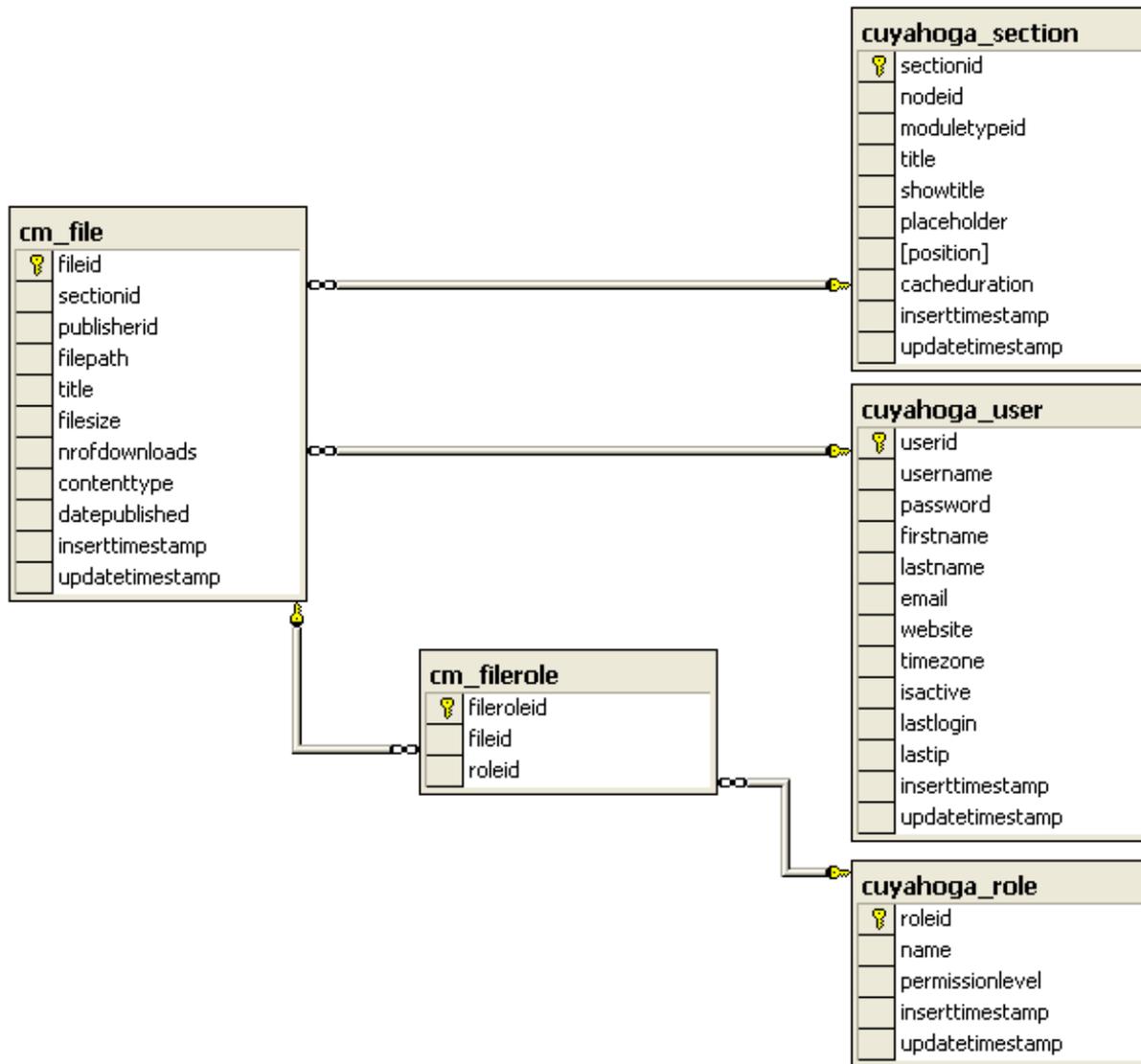
The requirements for the Downloads module are simple: users can download files and access to

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

```

<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 be 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. But don't worry, all you have to do is mark the module controller with the INHibernate module interface and Cuyahoga takes care of everything:

```

namespace Cuyahoga.Modules.Downloads
{
    /// <summary>
    /// The Downloads Module provides file downloads for users.
    /// </summary>
    public class DownloadsModule : ModuleBase, INHibernateModule
    {
        ...
    }
}

```

Just one rule here: the File class has to be in the same assembly as the module controller. When loading the module, Cuyahoga checks if the INHibernateModule interface is implemented. If so, it adds the module assembly to the NHibernate configuration.

Displaying objects

The *Web/Downloads.aspx* 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 or dao.

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()
{
    ISession session = this._sessionManager.OpenSession();
    string hql = "from File f where f.Section.Id = :sectionId order by f.DatePublished desc";
    IQuery q = session.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.

Note that in the above example, the NHibernate Session that does the database work is obtained via a SessionManager component that comes with the Castle NHibernate facility. This is new for Cuyahoga 1.5.0. All components for Database and File access, Email, etc. are externalized, so the module controller is much less coupled to infrastructure code. All components are glued together with dependency injection via the Castle Windsor container (see next section).

Leveraging services via dependency injection

From Cuyahoga 1.5.0, several internal and external components are available via dependency injection, provided by the Castle Windsor IoC container. It's too much to discuss all of this completely in this document. Check out the Castle website (<http://www.castleproject.org/container/index.html>) for more information.

All you have to know is what services are available and how to obtain a reference to such a service. The Cuyahoga services are defined in a configuration file, services.config (in the /Web/Config directory). Most of the times, you'll also want to use the Castle ISessionManager component for database access via NHibernate. To obtain a reference to such a component, simply put it as a parameter in the constructor of the module controller and at run-time, a nice instance of the component will be supplied by the system.

Again we take a look at the Downloads module how it is done:

```

/// <summary>
/// DownloadsModule constructor.
/// </summary>
/// <param name="fileService">FileService dependency.</param>
/// <param name="sessionManager">NHibernate session manager dependency.</param>
public DownloadsModule(IFileService fileService, ISessionManager sessionManager)
{
    this._sessionManager = sessionManager;
    this._fileService = fileService;
}

```

The Downloads module depends on two components: the Cuyahoga IFileService that stores physical files on the file system and the Castle ISessionManager for database access via NHibernate.

Using the PathInfo parameters to pass variables

A traditional way to pass parameters to a page request is with query strings. With Cuyahoga it is still possible to use these, but the majority of modules uses PathInfo variables to pass parameters. This makes urls slightly more search-engine friendly.

Now if the Cuyahoga page engine receives a request like

<http://servername/cuyahoga/23/section.aspx/download/45> it loads the page (Node) where the section with id 23 is located and calls the ParsePathInfo() method of the module controller that belongs to the

section with id 23. This method sets the module controller in a specific state, so that the rest of the module knows what (not) to do. For example, if a Downloads module instance retrieves the PathInfo '/download/45' like in the above url, it knows that it has to stream the file with id 45 to the browser.

Custom module settings

If a module has to be configurable per section, you can define configuration parameters in the *cuyahoga_modulesetting* table. These parameters automatically show up when editing a section in the site administration and the values are stored in the Settings collection of a section and later persisted in the *cuyahoga_sectionsetting* table. The settings can be used in different ways, but the preferred way is to override the ReadSectionSettings method in the module controller to store the module settings in strong typed variables (from DownloadsModule.cs):

```
public override void ReadSectionSettings()
{
    base.ReadSectionSettings();
    // Set dynamic module settings
    string physicalDir = Convert.ToString(base.Section.Settings["PHYSICAL_DIR"]);
    if (physicalDir != String.Empty)
    {
        this._physicalDir = physicalDir;
    }
    this._showPublisher = Convert.ToBoolean(base.Section.Settings["SHOW_PUBLISHER"]);
    this._showDateModified = Convert.ToBoolean(base.Section.Settings["SHOW_DATE"]);
    this._showNumberOfDownloads =
Convert.ToBoolean(base.Section.Settings["SHOW_NUMBER_OF_DOWNLOADS"]);
}
```

New in Cuyahoga 1.5.0 is the possibility to validate custom module settings. To do this, override the ValidateModuleSettings method in the module controller that is called after editing section properties:

```
/// <summary>
/// Validate module settings.
/// </summary>
public override void ValidateSectionSettings()
{
    // Check if the physical directory exists.
    // We need to do this based on the section setting because it might be possible that the related
section
// isn't saved yet.
if (base.Section != null)
{
    string physicalDir = Convert.ToString(base.Section.Settings["PHYSICAL_DIR"]);
    if (!String.IsNullOrEmpty(physicalDir))
    {
        CheckPhysicalDirectory(physicalDir);
    }
}
base.ValidateSectionSettings();
}
```

Module administration

Besides displaying content, a module also needs to offer possibilities to *manage* content. Editing content is done via separate .aspx pages. The relative link to the entry administration page is stored in the *cuyahoga_moduletype* table in the *editpath* column. For example, the Downloads module has 'Modules/Downloads/EditDownloads.aspx' as its entry edit page. When left blank, Cuyahoga assumes there is no module administration page.

If an authenticated user is allowed to edit a specific section, an 'edit' link to the module administration page automatically appears that points to the module administration entry page.

Building module administration pages is almost the same as building ordinary ASP.NET pages. Just

make sure that the code-behind page inherits `Cuyahoga.Web.UI.ModuleAdminBasePage`. This base class provides an environment where the related module controller, Section and Node are available. The `ModuleAdminBasePage` also provides generic facilities to update the search index when content is changed.

Make the module searchable

If a module needs its content indexed for full text searching, the module controller has to implement the `ISearchable` interface. Take a look at the Articles module (`ArticleModule.cs`) for an example.

The `ISearchable` interface has one method `GetAllSearchableContent()` and three events `ContentCreated`, `ContentUpdated` and `ContentDeleted`.

The `GetAllSearchableContent()` method returns all content for a specific module instance (related to a Section) and is called when the index is completely rebuilt from the administration interface. Because every module can have a different content structure, we need a unified structure to pass content to the search indexer because the indexer can't of course guess which content a module wants to have indexed. This is the `Cuyahoga.Core.Search.SearchContent` class. `GetAllSearchableContent()` has to return an array of `SearchContent` objects.

If a module controller implements `ISearchable`, the implemented events from the module controller are automatically handled by the `ModuleAdminBasePage`. The events all have the same `EventArgs`, `IndexEventArgs`, that contains a `SearchContent` object to be passed to the indexer. For example, when a new article is added in the Articles module, the module controller (`ArticleModule.cs`) saves an `Article` object, creates a `SearchContent` object from the `Article` object and raises the `ContentCreated` event with the `SearchContent` object as event data via `IndexEventArgs`.

Create an RSS feed for the module

To offer an RSS feed for a module, the module controller has to implement the `ISyndicable` interface. Again, the Articles module serves as an example how to accomplish this. The `ISyndicable` has just one method `GetRssFeed()`. This method returns a `Cuyahoga.Core.Util.RssChannel` object that contains everything for a complete RSS feed. The module itself is responsible for creating the `RssChannel` object.

At the time of writing (Cuyahoga 0.9), only RSS 2.0 is supported as a format for the feeds.

Language resources

If a module has to support different languages, you can create text strings in language resource files (the default .NET .resx format) that have to be placed in the `Module_Name/Web/Resources` directory. Like any resource file, the build action of these files has to be set to 'Embedded Resource'.

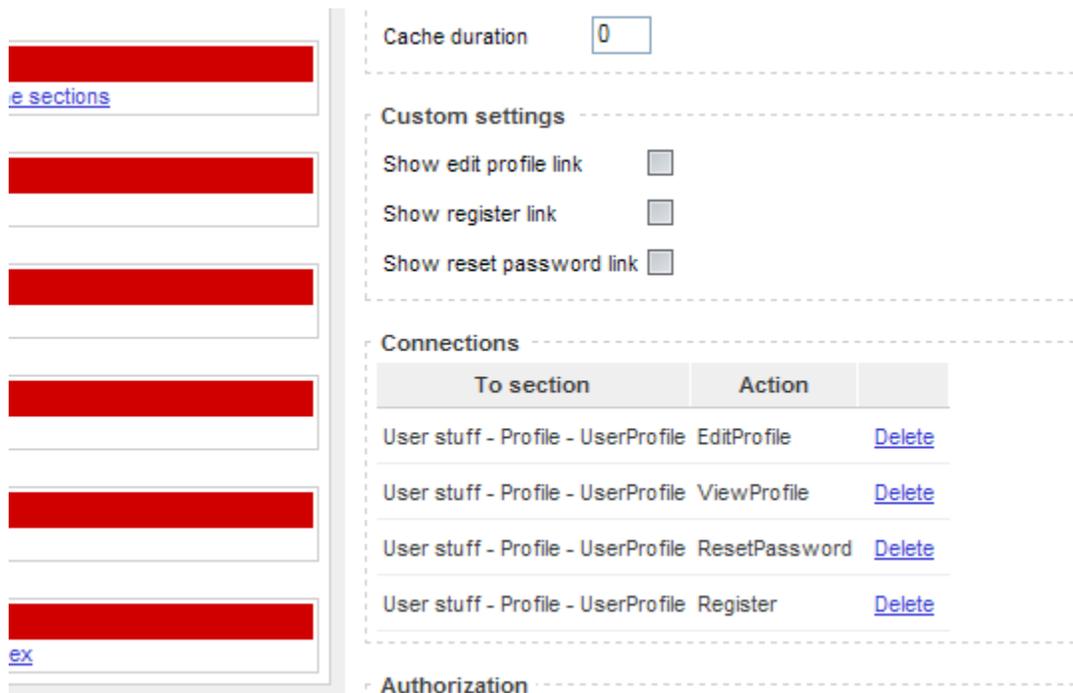
Only the display user controls (non-admin) support this localization. The `BaseModuleControl` that is the base class for all display user controls inherits from `LocalizedUserControl`. This base class provides a convenient `GetText()` method that accepts an identifier string and returns the localized string, depending on the current culture context. See the Downloads module for an example.

Inter-module communication

Sometimes, it's nice to be able link from one module to another. For example: Cuyahoga has a User

module with a login screen and some links to register as a new user or reset the password. The User module doesn't have this functionality itself and has to link a section that contains a Profile module. This is accomplished via two interfaces that have to be implemented by the Module Controller: IActionProvider and IActionConsumer.

IActionProvider has a method GetOutboundActions() that returns a list of actions that the module can link to. IActionConsumer has a method GetInboundActions() that returns a list of actions that a module is able to handle. Now in the section properties admin page it's possible to connect sections that have compatible actions (comparing outbound actions with inbound actions of all other sections):



When a connection between sections is made, the connected section is available via the Connections property of the section (from /Modules/User/User.ascx):

```
Section sectionTo = this._module.Section.Connections["Register"] as Section;
if (sectionTo != null)
{
    this.hplRegister.NavigateUrl = UrlHelper.GetUrlFromSection(sectionTo) + "/Register";
}
```

Just go and check out the User and Profile modules in the /Modules/User directory for an example how this works.

Deployment

The deployment of modules is a two step process:

1. Copy the required files to the deployment server
2. Run the database scripts to register the module in the Cuyahoga database and create the module-related tables.

The last step can be accomplished by just running a single.sql script on your database server that sets up the database, but in this example we'll set up a files structure so that the database part of the modules

can be installed, upgraded and uninstalled with the module installer. You can reach the module installer in the Cuyahoga site administration from 'Modules'.

Copy the files to the server

First create a subdirectory on the server in the Modules directory that is in the root of your web application and give it the same name as your module (column *name* in the *cuyahoga_moduletype* table). This is the place where the .aspx, ascx and other web files like images have to be copied. Of course we also need to deploy the compiled .dll to the server. This one goes into the /bin directory of the web application.

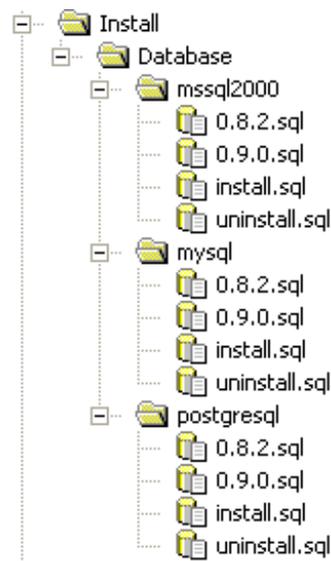
For example, when deploying the Downloads module, all files from the Web/ directory except the resource files are copied to the /Modules/Downloads/ directory on the server (preserving the directory structure) and the Cuyahoga.Modules.Downloads.dll file is copied to the /bin directory.

Database scripts for the module installer

Besides copying the web files, the database installer scripts also have to be copied to the server.

The module installer wants the database install scripts to be organized in a specific way. The layout is /Install/Database/Database_Type. For Database_Type you can choose either mssql2000, mysql or postgresql. Check the spelling, because the installer looks for the exact directory.

Again, we'll take the Downloads module as an example. It has the following installer directory structure which is copied to the /Modules/Downloads/ directory:



At least a *install.sql* file is required. The module installer specifically looks for this file. This file contains the create statements for the module tables, the module definition data and a record for the *cuyahoga_version* table (SQL Server example):

```
/*
 * Module tables structure
 */
CREATE TABLE cm_file(
fileid int identity(1,1) NOT NULL CONSTRAINT PK_file PRIMARY KEY,
sectionid int NOT NULL,
publisherid int NOT NULL,
```

```

filepath nvarchar(255) NOT NULL,
title nvarchar(100) NULL,
filesize int NOT NULL,
nofdownloads int NOT NULL,
contenttype nvarchar(50) NOT NULL,
datepublished datetime NOT NULL,
inserttimestamp datetime DEFAULT current_timestamp NOT NULL,
updatetimestamp datetime NOT NULL)
go

[...more module tables stuff ...]

/*
 * Module definition data
 */
DECLARE @moduletypeid int

INSERT INTO cuyahoga_moduletype (name, assemblyname, classname, path, editpath, inserttimestamp,
updatetimestamp) VALUES ('Downloads', 'Cuyahoga.Modules.Downloads',
'Cuyahoga.Modules.Downloads.DownloadsModule', 'Modules/Downloads/Downloads.ascx',
'Modules/Downloads/EditDownloads.aspx', '2005-05-15 14:36:28.324', '2004-05-15 14:36:28.324')

SELECT @moduletypeid = Scope_Identity()

INSERT INTO cuyahoga_modulesetting (moduletypeid, name, friendlyname, settingdatatype, iscustomtype,
isrequired) VALUES (@moduletypeid, 'SHOW_PUBLISHER', 'Show publisher', 'System.Boolean', 0, 1)

[. more modulesettings..]
/*
 * Version
 */
INSERT INTO cuyahoga_version (assembly, major, minor, patch) VALUES ('Cuyahoga.Modules.Downloads', 0,
9, 0)

```

Optionally, you can add an *uninstall.sql* script that removes the module tables from the database and removes the module definition data and the version record. The module installer will show the uninstall option when it finds the *uninstall.sql* file.

Again optional are the *.sql* scripts for upgrading. These need to follow the naming convention *major_version.minor_version.patch.sql*. The script with the lowest version number has to create every database object for that particular version (equally to the *install.sql* for that version). The files with the higher numbers have scripts that upgrade the database to that version.

Important: To use the upgrade features it is very important that the version record is updated with every upgrade script *and* that the assembly version is maintained properly. The module installer reads the version number in the database for a specific module and then analyzes the current module assembly version. If there is a difference between those two versions, all upgrade scripts, from the database version until the current assembly version, are executed.