Advanced Developer Training

August 2014
# Table of Contents

Advanced Developer Training ........................................................................................................... 3

Introduction ...................................................................................................................................... 3

Getting Started ............................................................................................................................... 3

What You’ll Learn ............................................................................................................................. 3

What You’ll Need .............................................................................................................................. 3

How to Get Set Up ............................................................................................................................ 3

User Generated Content .................................................................................................................. 4

Feedback ......................................................................................................................................... 4

Lab – Adding Feedback to a Content Definition .............................................................................. 6

Website Users ................................................................................................................................... 9

Lab – Creating a Website User Type in UGC ................................................................................... 10

Lab – Implementing a Login Page ...................................................................................................... 11

Comments ...................................................................................................................................... 14

Lab – Add Comments to a Content Definition ................................................................................. 15

Lab – Saving and Rendering Comments .......................................................................................... 17

Caching and Performance ................................................................................................................ 22

Output Caching ................................................................................................................................. 22

Caching in UCG ................................................................................................................................. 24

Agility Dependency Cache ................................................................................................................ 26

Accessing Volatile Resources .......................................................................................................... 27

Customizing Agility Input Forms .................................................................................................... 32

Ways to Customize Your Forms ....................................................................................................... 32

Note on Custom HTML Input Forms ................................................................................................ 33

Input Form API ................................................................................................................................. 34

Content Import API .......................................................................................................................... 35

Lab – Importing Content using a Console Application ...................................................................... 36

REST API ......................................................................................................................................... 40

REST API Setup ............................................................................................................................... 40
The URL Schemes

Lab – Getting Content via the REST API
Advanced Developer Training

Introduction
Now that you have concluded the Intermediate Developer Training, it is assumed you know the ins and outs of Web Content Management (WCM) and basic concepts of User Generated Content (UGC). These concepts will not be covered again in this training, but their understanding will be required in order to follow along and complete the labs within the Advanced Developer Training.

Getting Started

What You’ll Learn
In this course, we will be focusing on building on our understanding of UGC, introduction to the REST API, maintaining site performance, custom input forms, and the Content Import API.

What You’ll Need
To complete the labs in this training course, you’ll need to have a Sandbox Sample Site setup for you and have a copy of the site source code. Our current Sandbox Sample Site is a .NET 4.0 site using ASP.NET MVC 4.

You will need to have access to an Integrated Development Environment (IDE) such as Visual Studio. We recommend using Visual Studio 2013 – there is a free version Visual Studio Express for Web which is sufficient.

Finally, you’ll need a web browser with an internet connection to access the Agility Content Manager. The Agility Content Manager supports all modern browsers.

How to Get Set Up
You may use the same Sandbox Sample Site you used for your Developer 101 Training / Intermediate Developer Training. Having completed the Developer 101/Intermediate Training is not a requirement – although it is recommended if you are new to Agility or unfamiliar with concepts being covered here.

If you haven’t done so already, contact our sales team to get a Sample Site set up. We’ll send you a download link where you can get the source code for your sample site. Ensure you also have received your FTP credentials to where you can deploy code changes.

Once you’ve got the source files, select the .sln Solution File to open the project in Visual Studio. The site will automatically be configured to connect to your Sample Site instance and will be running in Development Mode using IIS Express. Click “Start Debugging” or “Start without Debugging” to run your site locally in your favourite web browser.
User Generated Content

Feedback

The User Generated Content Feedback API provides the ability to store "flag" based data that you may need from your users. Some examples would be Likes, Votes, Ratings, Approval/Disapproval, etc. The API is built to be lightweight and fast to add and retrieve data on the fly. It is also built generically so that you can attach feedback data to both Content Definitions (WCM) and UGC Definitions (UGC).

The API is just an extension of the existing UGC API and is used in the same fashion, except it has its own methods and object arguments.

Additional information outside of this training on the Feedback API (such as all JS and C# API Calls) can be found in the Feedback API Reference on the Developer Site.

Connecting with WCM and UGC Definitions

Any definition (WCM/UGC) in Agility can have feedback types associated to it. Enabling feedback types on your definition allows you to be able to use the Feedback tab on individual content items or records to view all feedback data collected from your website.
Feedback Types
- Likes
  - A counter of how many times the content item has been liked
  - Uses a BooleanFeedback (true/false) object to save
- Voting
  - A counter of how many votes this content item has received
  - Usually used for ranking content items or for contests
  - Uses a BooleanFeedback (true/false) object to save
- Approve / Disapprove
  - A counter of how many times an item has been approved or disapproved (similar to Like/Dislike)
  - Uses a BooleanFeedback (true/false) object to save
- Ratings
  - Can count rating sum and average
  - Uses a RatingFeedback object to save

Feedback Objects
BooleanFeedback Object
- Used for saving Likes, Voting and Approve / Disapprove
- View BooleanFeedback Developer API Reference for info on its properties
RatingFeedback Object
- Used for saving Ratings
- View RatingFeedback Developer API Reference for info on its properties
FeedbackSearchArgs Object
- Used for searching and retrieving feedback data for all Feedback Types including Likes, Voting, Approve / Disapprove, and Ratings
- View FeedbackSearchArgs Developer API Reference for info on its properties
Lab – Adding Feedback to a Content Definition

In this lab, we will be adding a Like component to our existing Products definition in Agility. This will allow website visitors to Like a product. You will need the Sample Site open in Visual Studio as well as Agility open in a browser.

1. In Agility, navigate to and open the “Product” Content Definition in Settings > Content Definitions

2. In the General Settings, check the box labelled “Likes” and click Save & Close.

3. In Visual Studio, open the existing view at the path “~/Views/ProductDetailsModule.cshtml” and replace the contents of the view with the following:

```csharp
using Agility.Web.Plugins
@model MVC4SampleSite.ViewModels.ProductDetailsModule

<div class="row">
    <div class="col-lg-12">
        @Model.Product.Title
        @if (Model.Category != null) {
            <small>@Model.Category.Title</small>
        }
    </div>

    <div class="like" data-content-id="@Model.Product.ContentID" data-reference-name="@Model.Product.ReferenceName">
        <span class="count">(0)</span>
        <a href="javascript:;" class="like-button">Like</a>
    </div>
</div>

<div class="row">
    <div class="col-md-8">
        @if (Model.Product.MainImage != null) {
            <img class="img-responsive" src="@Model.Product.MainImage.URL">
        }
    </div>

    <div class="col-md-4">
        <h3>@string.Format("{0:C}", Model.Product.Price)</h3>
        @Html.Raw(Model.Product.Details)
    </div>
</div>

<script type="text/javascript">
    @Html.DelayedScriptExecutionWrapperBegin()
    Modules.InitProductLike($(".like"))
    @Html.DelayedScriptExecutionWrapperEnd()
</script>
```
Note the additional “like” div and the script initialization function at the bottom.

4. Next, add the following JavaScript function to the file located at “~/Scripts/Site/Module.js”:

```javascript
Modules.InitProductLike = function ($like) {

    var renderLikes = function () {
        // set the like count
        var search = new Agility.UGC.API.FeedbackSearchArg();
        search.ReferenceName = $like.data('reference-name');
        search.ContentType = Agility.UGC.API.FeedbackContentType.WCM;
        search.SubmissionType = Agility.UGC.API.FeedbackSubmissionType.Like;
        // array of ID's to look up
        search.RelatedContentIDs = [$like.data('content-id')];
        search.Action = Agility.UGC.API.AggregateTypes.Count;

        Agility.UGC.API.GetFeedbackAggregate(search, function (data) {
            $('.count', $like).text('(' + data.ResponseData[0].Result + ')');
        });
    }

    $('.like-button', $like).click(function () {

        var LikeFeedback = new Agility.UGC.API.BooleanFeedback();
        LikeFeedback.ReferenceName = $like.data('reference-name');
        LikeFeedback.ContentType = Agility.UGC.API.FeedbackContentType.WCM;
        LikeFeedback.RelatedContentID = $like.data('content-id');

        // save the like
        Agility.UGC.API.SaveLike(LikeFeedback, function (data) {
            if (data.ResponseType == Agility.UGC.API.ResponseType.OK) {
                // re-render likes
                renderLikes();
            }
        });
    });

    // render like count on load
    renderLikes();
};
```

5. In Visual Studio, ensure your site is running locally and navigate to a product’s details page in your browser and click the Like link we added and watch the count increase.
6. In Agility, navigate to that same product content item and click on the Feedback Stats tab. You should see the same number of likes that were displayed on the website.
Website Users

The UGC concepts and labs that we’ve looked at so far are based on anonymous data, where anyone can submit content via the JavaScript API into your Agility instance. In a more functionally advanced project, you may want to provide users with ability to create profiles for themselves and then log in. UGC allows for this with the concept of Website Users – a more specialized version of a UGC table.

These user records can be used solely for login purposes, however any other User Content saved by the user (while logged in) will also store the user ID in the CreatedBy property (of the saved record) so that you can keep a track of who creates and edits records.

Creating a Website User Type is a similar process to creating a User Generated Content Definition, but you are presented with a couple of extra options (username and password) specifically for the login workflow. These fields are then used to interact with the user type via specific API calls, such as Authenticate, IsAuthenticated, RetrievePassword, ChangePassword and LogOut.

For additional information and API Reference check out the User Profile API on the Developer Site.

C# vs JavaScript

When dealing with Website Users, it’s tempting to place all your user workflow in server-side code (C#) because of security concerns. However, this doesn’t have to be the case as only a logged in user can access their own profile record, and in fact JavaScript is still the recommended way to handle website user workflows.

There are also some additional challenges with using C# and Website Users including managing output caching – so you don’t see user information from another user! And server performance as your server would be doing all the heavy lifting.

In JavaScript, you don’t have to do any post-backs, manage output caching, or put extra stress on your server. Use the JavaScript API wherever possible.
Lab – Creating a Website User Type in UGC

In this lab, you’ll set up a website user type. We’ll work only in Agility, so you’ll just need your browser. At the end of the lab, you’ll have a Website User Type that you can add users to and implement a login page for.

1. In Agility, navigate to Settings > Website User Types, and then select New Content Definition

2. Type the display name as “Profile”.

3. Select the Form Builder tab.

4. Click Add Field, and type the label as “Email Address”. Change the type to “Email”, and check the checkboxes for “Required”, “Unique”, “Is a Login field” and “Is an Alert field”. This field will represent our user’s username, and we therefore want it to be a unique field. Setting it to be a login field means that the API will know which field to check when we’re performing the login later on. Click on OK & New to create another field.

5. Give this new field a label of “Password”, set the Type to “Password”, and ensure the “Required” checkbox is checked. Click on OK & Close, and then Save & Close to finish creating the Website User Type.

6. Now select the “Website Users” link from the top of the page in Agility. Click on Profile to open the table that you just created. Click on New to create a new user.

7. Enter any email address, preferably one that you own and have easy access to. Enter the Password as “password”

   Email Address:* russell@agilitycms.com
   Password:* password

8. Now click on Save. You’ll see that the password automatically becomes encrypted.

   Email Address:* russell@agilitycms.com
   Password:* ZbgKNYgrJoQxJbiarLIbFA==

9. We now have a working website user type that we can build a login page for.
Lab – Implementing a Login Page

In this lab, we’ll reference the **Website User Type** that you just created in the previous lab to create a Login Module in our website. We’ll be using both Agility in the browser and our sample site in Visual Studio. At the end of the lab, you’ll have a simple login form with the ability to reset a user’s password.

1. Firstly, let’s create a Module and add it to a page so we can invoke a partial view that will contain our HTML and JavaScript for our login page.

2. In Agility, create a **New Module Definition** called “Login”. For this exercise, you don’t need to enter any properties as we are solely focusing on testing login functionality. For the Output Template, select “Partial View” and set the path to “~/Views/Modules/Login.cshtml”.

3. Get your latest strongly-typed API by clicking **Download API** and updating it in your website project.

4. Create the view “~/Views/Modules/Login.cshtml” and enter the following:

```csharp
@using Agility.Web.Plugins
@model MVC4SampleSite.Models.Module_Login

<div id="LoginForm" style="margin: 100px">
    <div>
        <label for="txtEmailAddress">Email Address:</label>
        <input type="text" id="txtEmailAddress" />
    </div>
    <div>
        <label for="txtPassword">Password:</label>
        <input type="password" id="txtPassword" />
    </div>
    <div>
        <a href="#" id="aSubmit">Submit</a>
    </div>
</div>

<script type="text/javascript">
    @Html.DelayedScriptExecutionWrapperBegin()
    Modules.InitLogin($('#LoginForm'))
    @Html.DelayedScriptExecutionWrapperEnd()
</script>
```

5. Next, add the following JavaScript function to the existing Modules.js found at “~/Scripts/Site/Modules.js”:

```javascript
using Agility.Web.Plugins
model MVC4SampleSite.Models.Module_Login
</div>

<script type="text/javascript">
    @Html.DelayedScriptExecutionWrapperBegin()
    Modules.InitLogin($('#LoginForm'))
    @Html.DelayedScriptExecutionWrapperEnd()
</script>
```

6. Next, add the following JavaScript function to the existing Modules.js found at “~/Scripts/Site/Modules.js”:
Modules.InitLogin = function ($module) {
    $("#aSubmit").bind("click", function () {

        var username = $("#txtEmailAddress").val();
        var password = $("#txtPassword").val();

        //login
        Agility.UGC.API.Authenticate("Profile", username, password, true, function (data) {
            if (data.ResponseType == Agility.UGC.API.ResponseType.OK) {
                //no errors occurred
                var authToken = data.ResponseData;

                if (authToken != null) {
                    //login success
                    alert('Login successful, auth token returned, and cookie set')
                    $("#txtEmailAddress").val("");
                } else {
                    //login fail
                    alert('Invalid email or password');
                }
            } else {
                //some sort of error occurred
                alert(data.Message);
            }
            $("#txtPassword").val("");
        });
    });
    return false;
};

This script will take the supplied values and check them against our Website Users Type “Profile” via the Authenticate method. The API takes five parameters:

- **websiteUserTypeName** – this is the website user type definition that you are authenticating against, this is “Profile” in our case.

- **login** – this is the value of the field that has been checked as “Is login field”. We set this to the EmailAddress field in our case, so we’ll be expecting the user to supply an email address

- **password** – the user’s password, in plain text – the encryption functionality is handled by the API

- **persistCookie** – this is a boolean indicating whether the API should set the authentication state in a cookie for future reference

- **callback** – a function to process the authentication result
6. Add this “Login” module to a new page in Agility.

7. In Visual Studio, build your project, and run it locally. Navigate to the page you just created to verify the login form.
Comments
An additional component to UGC, Comments can be enabled on WCM or UGC definitions. Similar to Feedback types, they can relate to content items or records directly.

When Comments are enabled a new UGC Comment Record Type will be created to store the comments. Comments use the same API calls as standard UGC records for saving (SaveRecord) and retrieving records (GetRecord/SearchRecords).

An additional helper method has been added to help search for comments as well and produce comment hierarchies when comments are expected to be threaded. This new call is "SearchComments".

Comments can be combined with Website Users to allow for logged in users to submit comments, while keeping track of the Website User that created each comment – otherwise, comments will be created anonymously.

For more information around Comments API references please visit the UGC Comments API Reference on the Developer Site.
Lab – Add Comments to a Content Definition

In this lab, we will be adding Comments to the existing WCM Content Definition “Product”. This will allow comments to be associated with each Product content item. At the end of this lab, we will have the architecture required to take the next steps in building our commenting UI. This lab will only require Agility open in a browser, looking at your Sample Site.

1. In Agility, navigate to Settings > Content Definitions and open Product.
2. Check the checkbox labelled “Comments” and select Threaded. Click Save to save the content definition.
3. After you save the definition, click on Customize Comments within the Edit Content Definition dialogue.

![Edit Content Definition Dialogue](image-url)
4. A Schema window should be displayed for your comments.

5. Uncheck the box labelled “Requires Moderation”. This will disable moderation for our comments.

6. Click on the Form Builder tab. Take note of all the default fields being set. Here, you can optionally add any other fields you want to capture when visitors submit comments.

7. Note there are also Notifications and Alerts available you setup.

8. Click on the API Access tab. Click Add Access.

9. For System Name select “Website” and for Access Level select “Create”.

10. Leave all other settings as defaults and click Save & Close to finish creating your comments schema. Click Save & Close again to save your content definition.

11. In Agility, navigate to User Content (in the main menu) and click on the Comments tab.

12. You should see your Comments list displayed in the list. Here, moderators can view all comments related to products.

![User Content & Comments](image)

13. Additionally, if you navigate to the “Products” list in Digital Content and open a content item, it will have a new tab called Comments where you view only comments related to that content item.
Lab – Saving and Rendering Comments
In this lab, we will be adding a basic UI for rendering and saving comments submitted on our Product Details pages. At the end of this lab, you will have a functional, albeit basic UI for rendering and submitted comments using the UGC API. You will require the Sample Site open in Agility as well as Visual Studio.

1. In Visual Studio, navigate to the view “~/Views/Products/ProductDetailsModule.cshtml” and replace its content with the following (next page):
@using Agility.Web.Plugins
@using Agility.Web.Extensions
@model MVC4SampleSite.ViewModels.ProductDetailsModule

<div class="row">
  <div class="col-lg-12">
    <h1 class="page-header">
      @Model.Product.Title
      @if (Model.Category != null)
      {
        <small>@Model.Category.Title</small>
      }
    </h1>

    <div class="like" data-content-id="@Model.Product.ContentID" data-reference-name="@Model.Product.ReferenceName">
      <span class="count">(0)</span>
      <a href="javascript:;" class="like-button">Like</a>
    </div>
  </div>
</div>

<div class="row">
  <div class="col-md-8">
    @if (Model.Product.MainImage != null)
    {
      <img class="img-responsive" src="@Model.Product.MainImage.URL">
    }

    <div class="comments" data-content-id="@Model.Product.ContentID" data-comments-recordtype="@Model.Product.CommentsRecordTypeName()">
      <div class="form-group">
        <label>Name:</label>
        <input type="text" id="name" />
      </div>
      <div class="form-group">
        <label>Comment:</label>
        <textarea id="comment-input"></textarea>
      </div>
      <button id="submit-comment" class="btn btn-default">Submit</button>
    </div>
  </div>
  <div class="col-md-4">
    <h3>@string.Format("{0:C}", Model.Product.Price)</h3>
    @Html.Raw(Model.Product.Details)
  </div>
</div>

<script type="text/javascript">
  @Html.DelayedScriptExecutionWrapperBegin()
  Modules.InitProductLike($('.like'))
  Modules.InitProductComments($('.comments'))
  @Html.DelayedScriptExecutionWrapperEnd()
</script>
Note the addition of the “.comments” div. This contain our UI for the comment submission as well as rendering the published comments themselves.

2. In Visual Studio, navigate to the JavaScript file Modules.js located in “~/Scripts/Site/Modules.js” and add the following (next page) function to the file:
Modules.InitProductComments = function ($comments) {
    var cacheKey = "CommentsFor_" + $comments.data('content-id');
    var renderComments = function () {
        $('.comments-list', $comments).html('');
        var commentSearchArg = {
            RelatedContentID: $comments.data('content-id'), //ID you are pulling comments for
            PageSize: 10,
            RecordOffset: 0,
            SortedField: "CreatedOn",
            SortDirection: "asc",
            RecordTypeName: $comments.data('comments-recordtype'),
            CacheKey: cacheKey
        };
        Agility.UGC.API.SearchComments(commentSearchArg, function (data) {
            if (data.ResponseType == 0) {
                //Render Comment Data
                //data.ResponseData.Records
                //data.ResponseData.TotalRecords
                var html = ""
                if (data.ResponseData != null && data.ResponseData.Records.length > 0) {
                    $.each(data.ResponseData.Records, function () {
                        html += "<li class='comment'>"
                        html += this.Comment;
                        html += "<div>Submitted By:" + this.Name + "</div>"
                        html += "</li>"
                    });
                }
                $(html).appendTo($('.comments-list', $comments));
            } else {
                alert(data.Message)
            }
        });
    };
    renderComments();

    $('#submit-comment').click(function () {
        var record = {
            ID: -1,
            RecordTypeName: $comments.data('comments-recordtype'),
            RelatedContentID: $comments.data('content-id'),
            Comment: $('#comment-input', $comments).val(),
            Name: $('#name', $comments).val() 
        };
        Agility.UGC.API.SaveRecord(record, function (data) {
            if (data.ResponseType == Agility.UGC.API.ResponseType.OK) {
                alert('Comment saved.')
                renderComments();
            } else {
                alert(data.Message)
            }
        }, cacheKey);
    });
    renderComments();
}
Here, we are using the JavaScript Comments API to search for comments relating to this product. Note there is a CacheKey parameter being set in the SearchComments call. Setting this cache key allows us to use that same cachekey as a parameter in our SaveRecord call. When the SaveRecord call is made, it will also clear the cache for any search results using the same cache key. This allows us to immediately re-render the comments after submission without having to wait for UGC’s built-in caching to expire.

3. In a browser, navigate to a product details page in the sample site and submit a comment. You should immediately see your comment appear after it is re-rendered. Note that we had disabled content moderation for this comments definition. Your UI should look something like this:

```
Name: 

Comment: 

• This is my comment
  Submitted By: James Vidler
• Another comment
  Submitted By: James Vidler
• And Another comment...
  Submitted By: James Vidler
```
Caching and Performance

Output Caching

By default any page that is generated by Agility is output cached. This provides a great performance improvement for your website by allowing a page that has already been rendered once to simply return the same rendered HTML code.

Settings

Output Caching can be turned on and off within the Agility Content Manager in the Settings > Caching section. By default the output cache timeout is set to 300 seconds (5 minutes). This value can be increased or decreased depending on your needs. The higher the value that is set the more stale your website content may be and the lower the value the lower the performance gain.

Please note that when a website is being developed and the site is set to run in Development Mode output cache is not enabled. Output cache is only turned on when a site is not in Development Mode (i.e. Live Mode).

Publishing Pages and Content

Whenever a page or is published in Agility, Agility will attempt to clear the output cache for the affected page. The same goes for modules and content within those modules. The Agility.Web.dll maintains relationships contained within your content structures to ensure that cache is cleared when a change is made and published down to your website. In most cases you will receive all the performance gain without having stale data as long as you are not adding custom data to a page from an exterior data source like a database.

Page URL’s and Custom Output Cache

By default there is a new output cache for each page URL within your site. This does NOT include different query string values such as:


OR


By default, only the page path is used to create the cache key.

We run into a problem when we build custom logic on the server side which loads data based on querystring, a cookie, session variable, etc. To explain this scenario I will use the example of a Profile Update Page.

Say we have the url:

http://www.MyAgilitySite.com/Profile.aspx
This page will on the server side read in the currently logged in users User ID and load their specific profile. By default with output cache the first user will load the page and see a valid profile page. If a second user loads this page within the 5 minute output cache timeout they will see the profile of the first user to load the page. This of course is a major issue and needs to be handled by the developer.

**Handling Querystrings and Output Caching**

If you are using querystrings and those querystrings impact what HTML is outputted for the current URL, you will need tell the Output Cache to vary by parameters. One way to do that is including the following within your controller:

```
```

This will treat every querystring key and value as part of the output cache key.

**Setting a Custom Output Cache String for Your Site**

There may be cases where you want to set the custom cache string for your Output Caching in your website.

You can do this by adding the following code to your `Global.asax`:

```csharp
public override string GetVaryByCustomString(HttpContext context, string custom) {
    /*
     * Handle OutputCache
     * AgilityCacheControl is a special "VaryByCustom" value that is added in the Agility Controller
     */
    if (string.Compare(custom, "AgilityCacheControl", true) == 0)
    {
        string s = Data.GetAgilityVaryByCustomString(context);
        s = string.Format("{0}.{1}", s, context.Request.Url.Host);
        return s;
    }
    return base.GetVaryByCustomString(context, custom);
}
```

This allows you to manipulate the cache string (s) to include any value you wish. A specific example could be including the value of a user’s cookie that may influence the behaviour of various modules and functionality across your website.
Caching in UCG
The UGC cache system will automatically cache search results to ensure UGC doesn’t have performance issues which can bog down your website during high load read and writes. This helps with cases such as UGC contests where a large number of people are submitting content for a short period of time.

Caching In Agility Content Manager
Calls made with an Admin API Key (e.g. from Agility Content Manager) are NEVER cached and saving records in Agility does NOT clear cache from the website.

Applying Custom Cache Keys
There are cases where you cannot wait for UGC to churn its cache and need to force it show you the latest search results. Using custom cache keys enables you to specifically manage the cache for search results from UGC – rather than forcing the entire UGC table to clear its cache you can simply clear the cache for subset of items you are currently working with. A specific use case for this is when you are saving comments for a website, once you save the comment you want to re-render the comments list so that the user can see their comment has been posted. Using a cachekey, you can ensure that the search results for those comments are cleared once the comment is saved.

SearchRecords & SearchComments
The SearchRecords and SearchComments API methods have a parameter for a custom cache key, so the website can store data for X minutes in cache without it being kicked out.

ClearCache
The ClearCache API call takes a cacheKey parameter and clears the cache.

SaveRecord
The SaveRecord method has a parameter that will enable a custom cacheKey to be cleared.

What Happens without Custom Cache Keys
If you do NOT specify a custom cache key on your calls to SearchRecords, the system will follow these rules for cache:

Sliding Scale
Content is cached for 1 minute on a sliding scale. This mean if the cache is accessed within 1 minute, its potential time in cache will be extended by another minute.

Cache Validity
Cached Content is checked for validity every 1 minute. This means that we will only check in the database to see if a cache entry has expired every 1 minute (not on every request), so cache will be less likely to be cleared from memory.
UGC Caching Example

Normally, data in UGC is cached based on the UGC Definition it came from for about a minute. This means that changes to data may not appear in the system for about a minute by default.

If you want to take control of the cache on a given set of records in UGC, you can do this by using custom cache keys.

For instance, if you are showing a list of videos for a given user and you want that list to be updated when the user adds a new one, this is how you would do that:

- **Pick a cache key that is related to the user and uniquely identifies the set of records.**
  - For instance, a good cache key might be "[UserID]_Videos".
  - This ensures that a different cache key will be used for each user, and it will be unique to the videos list as well.
  - Note that Agility takes care of any searching, paging or sorting parameters.
  - You only need to worry about your set of records as a whole.

- **Use the cache key when make calls to SearchRecords**
  - Set the CacheKey value on the SearchArg object that you pass to the SearchRecords call.

- **Use the cache key when you make calls to SaveRecord on records in the videos list.**
  - There is an optional 3rd parameter to the SaveRecord method which is the same custom cache key that you used.
  - When you do this, the cached items for that user will be cleared.

- **Use the ClearCache method if necessary.**
  - Pass the same cache key string to ClearCache at any time to clear the cache on demand. You might want to do this if you are deleting the user or some other operation where you want to remove the cache.

Remember, if you do not use a custom cache key, your UGC data will be cached on the server for about a minute on a sliding scale, meaning, if you don’t change the data for a given UGC content definition, it will stay in cache.
Agility Dependency Cache

The Agility Dependency Cache is designed to help improve performance by allowing you to cache Agility content and set a dependency on the content itself. This means that the cache will get reset when the content is changed, either by an edit or a new item.

An additional feature of the AgilityDependencyCache is that the content will be cached in the HttpContent.Cache only if the website is in “Live” mode. If the website is in development or preview mode the content will be cached in the HttpContent.Items.

Usage

Agility.Web already handles lots of caching for you and there’s no need to cache an entire content list or content item. The AgilityDependencyCache should come into play however if you are performing some complex filtering on content lists or perhaps several content lists and then combining them into an object. Also, the more content items a list may contain, the heavier the filtering operation is on your server. You may want to cache the result and then set a dependency on the Agility content list(s) your result object contains.

The algorithm for caching objects using the AgilityDependencyCache should look something like this:

```csharp
// Check the cache to see if we already have an item added
string cacheKey = string.Format("AgilityContent.{0}", contentReferenceName);
var content = (AgilityContentItem)Agility.Web.Usils.AgilityDependencyCache.Select(cacheKey);

if (content == null)
{
    // Nothing found in the cache, get the content from the Agility data store in the usual way
    content = Data.GetContentView(contentReferenceName);

    // Add the data into the cache ready for next time we want to access it
    Agility.Web.Usils.AgilityDependencyCache.Insert(cacheKey, content, new List<string> { contentReferenceName }, DateTime.Now.AddHours(1));
}
```
Accessing Volatile Resources

There are a few things we can do as developers to alleviate stress on our servers when we access volatile resources. This can mean anything like REST services, database calls, or anything that might provide a bottle neck.

I would say the only thing that doesn't fall into this category is file system access, but even in those cases you might want to use cache to hold the object in memory if it isn't too large.

The Problem

When you access a slow resource, it causes a bottle neck: everything in your web request has to wait for that resource to respond before it can continue, and if you have multiple requests accessing that particular web page, your server could see a CPU spike, or even crash, as it juggles all those queued threads.

Most often what happens in this case is the web service or database server you are accessing will get progressively slower as it tries to deal with the extra load. In the case of some services, like the Twitter or Facebook API you may even get denied service.

Solution 1 – Cache and a Single Thread

One way around this issue is to ensure only a single thread accesses the resource at a time using an exclusive lock. You could also use a semaphore to limit the access to only a few threads as well, but we’ve found that a single thread tends to work best here.

Pseudo-code

- Resource is requested
- Determine a cache key that identifies the parameters used to access the resource
Check for the object in cache
  o If found, return it
• Start a critical section (lock())
  o Check cache again
    - If found return it
  o Access the resource
  o Put the result in cache for a specific amount of time
  o Return
• End the critical section

What happens here is that only a single thread will get into the critical section at a time, meaning that the external resource won't have too much load on it.

The trick here is to check if the object is in cache as the first operation inside the locked critical section, since it may have been put into cache by another thread while the current thread was waiting to get into the section.

The problem with this approach is that it doesn't help when the resource goes down entirely.

Solution 2 – Cache and a Single Thread plus File System

We can extend Solution 1 by adding a more persistent storage of the object on the file system, so that if the object is not in cache, we can use it from the file system until it expires, or if the external resource is not available.

The downside here is that the resource's timeout needs to be short enough to ensure we don’t kill our own web server with a huge request queue waiting for it to response.

Changes are **bolded** from Solution 1.

Pseudo-code

• Resource is requested
• Determine a cache key and filename that identifies the parameters used to access the resource
• Check for the object in cache
  o If found, return it
• **Check for the object in the file system**
  o If found, check the last write time - make sure it isn't too old
    - If not too old, return it
    - Else save it to temp variable
• Start a critical section (lock())
  o Check cache again
    - If found return it
  o Access the resource **with a reasonable timeout**
*If it errors out, use the temp variable from above if it's set*
  o Write the object to the file system.
  o Put the result in cache for a specific amount of time
  o Return it

End the critical section

You can see from the bold parts that we don’t have to change much from Solution 1 to add file system support. Note the italicized line, though, as it could become a bottleneck for this logic if the timeout is too long. On the flip side, if you pick too short of a timeout, you may never get a response from the resource at all.

If you are returning binary or string data, you can save it straight to file, but if you are working with objects, you probably want to serialize it, and we recommend binary serialization, as it is much faster than XML or JSON. However, if you already have a good serialization mechanism in place for JSON or XML from a web service, I would stay with that and just save the text data in the file, as it doesn’t introduce a second serialization scheme into things.

**Solution 3 - Offline Resource Access**

If you have a finite and well defined set of resources that your site needs to access regularly, there is no point using Solution 1 or 2 and having web requests tied up waiting for those requests to finish.

You may as well access those resources offline in another process or in a specific worker thread, saving the results to the file system. Even better, if you have a web farm, you can propagate the files to each server on the farm from the single offline process. This is usually best done in a windows service. You could also spawn a thread inside your website code to run this from, though, from the Global.asax, but run some tests first to make sure the app pool isn’t being shut down by inactivity and killing your thread.

**Pseudo-code - Offline Process**

- Start timer to kick off process on a fixed time span.
- On timer interval:
  o Determine list of resources from config file (or possibly hard coded)
    ▪ Loop each resource item if more than one
    ▪ Determine file path for the resource on this machine
    ▪ Access resource
      ▪ On success, save the result to file system
      ▪ On error, write to error log
  o When loop is complete
    ▪ If any errors occurred, send an email reporting them
    ▪ If in a web farm, copy the file to the other servers

**Pseudo-code - Website**
• Resource is requested
• Determine a cache key and filename that identifies the parameters used to access the resource
• Check for the object in cache
  o If found, return it
• Start a critical section (lock())
  o Check cache again
    ▪ If found return it
  o Access the file that contains the resource
  o Put the result in cache
    ▪ Use a sliding interval, so it stays in cache as long as it’s being regularly requested
    ▪ Use a CacheDependancy on the filepath so the cache is cleared when the file is overwritten
  o Return the result
• End the critical section
Code Sample: Accessing a Volatile Web Resource

```csharp
public static object _myCacheLock = new object();

public static string MyResourceGetterMethod()
{
    string url = "http://www.google.com);

    try
    {
        //try to get the thing from cache first
        string cacheKey = "MyCachedObjectKey";
        string result = HttpContext.Current.Cache[cacheKey] as string;
        if (result == null)
        {
            //critical section: ensure only one thread can get here at a time...
            //see msdn documentation for lock(), Mutex and Semaphore for other ways to do this
            lock (_myCacheLock)
            {
                //check for cache one more time in case it was put into cache while we were
                //waiting for the lock to clear
                result = HttpContext.Current.Cache[cacheKey] as string;
                if (result == null)
                {
                    //do the web request
                    WebClient webClient = new WebClient();
                    result = webClient.DownloadString(url);

                    //put this in cache for 5 minutes absolute (change this as needed)
                    HttpContext.Current.Cache.Add(cacheKey, result, null,
                                                DateTime.Now.AddMinutes(5), Cache.NoSlidingExpiration, CacheItemPriority.Normal, null);
                }
            }
            return result;
        }
    }
    catch (Exception ex)
    {
        //do something with this error
        throw new ApplicationException(string.Format("Could not access url {0}'", url), ex);
    }
}
```
Customizing Agility Input Forms

Up until this point, we have used the default input forms for Module and Content definitions (WCM) and UGC Definitions. Agility identifies your properties and their types and renders the HTML form to the content editor.

Now, there cases where you may want to have finer control over your input forms and any scripts that might be running within those forms. Some use cases for this includes:

- Adding additional HTML to the input form to provide contextual help
- Adding scripts that will manipulate data or validation in the input form
- Getting data from other content lists to display or validate against
- Integrations with other third-party APIs using JavaScript

Ways to Customize Your Forms

There are several ways to customize your input forms:

Adding a Custom Section

- You can add Custom Sections as a field/property type in WCM and UCG definitions dialogue
- Allows you to add a small snippet of custom HTML to an input form (just like regular property/field) while still using the Agility System Generated Form

Using Custom Scripts

- You can add JavaScript to the Custom Scripts tab in WCM and UGC definitions dialogue
- WCM definitions support the use of the Input Form API to hook into events within the input form as well as get other WCM content
- Both WCM and UGC definitions can utilize the UGC API to get other UGC content

Using a Custom HTML Input Form

- WCM and UGC definitions allow you to set a custom HTML input form where you have full control over the input form HTML
- Any properties that exist on the definition must have a field in the HTML form
- Editing properties on the definition will require you to manually update the form accordingly
Note on Custom HTML Input Forms

Try to avoid using these unless there is no other alternative. Maintaining custom HTML Input forms can be cumbersome as well as prevent you from receiving features around new or improved input controls released by the Agility product team.
Input Form API

The Input API consists of an API for Input Controls as well as Events that you can use in WCM Custom Scripts.

Input Controls
There are a series of method calls that are available which allow you to pull Agility WCM content within your input form. See the Developer Site: Input Form API for Input Controls for full API references.

Events
You can hook into events such as OnLoadComplete, OnBeforeSave, OnBeforePublish, OnAfterPublish, etc. to integrate your custom JavaScript into core components of the input form. For full API reference please see Developer Site: Input Form API Events.
Content Import API

When you need to save web content into Agility for an import, you can use the Content Import API (C#) to do so. The Content Import API can be found within Agility.Web. It’s a JSON API specifically created to allow you to save items into the Digital Content section of Agility. This can be useful for doing content imports at the start of a project, or for keeping Agility content in sync with an outside system. It is not meant to be used on a regular basis to replace the functions of the content manager.

Typically, the Content Import API is used in a console application or windows service to either get, save, search, upload, or delete content.

For full API reference, please visit [Developer Site: Content Import API](#).
Lab – Importing Content using a Console Application

In this lab, we will demonstrate using the Content Import API to import some additional “Products” into the existing content list “Products”. You will need Visual Studio in order to create and run the console application and a browser to view the imported products in Agility. You will required the Sample Site open as in Visual Studio so that you can copy some setting from the web.config over. It assumed you are familiar with creating new projects in Visual Studio including console applications.

1. In Visual Studio, create a new Project. Select Console Application as the project type.

2. Next, we need to add in the additional components so that the console application can connect to your Agility content. Start by adding a new Reference to the same Agility.Web.dll the sample site is using (or make a copy).

3. In you App.config, add the following code:

```xml
<configuration>
  <configSections>
    <!-- Agility.Web Config Group -->
    <sectionGroup name="agility.web">
      <section name="settings" type="Agility.Web.Configuration.Settings, Agility.Web" allowLocation="true" allowDefinition="Everywhere" restartOnExternalChanges="false" requirePermission="false" />
    </sectionGroup>
  </configSections>
  <agility.web>
    <settings applicationName="Insert Website Name Here (Import)" keepUrlPathOnForcedChannel="true" developmentMode="true" contentCacheFilePath="c:\AgilityContent\SampleMVC\" redirect404ToDefaultLanguage="false">
      <websites>
        <add websiteName="Insert Website Name Here" securityKey="Insert Website Security Key Here" />
      </websites>
      <trace traceLevel="Verbose" logFilePath="c:\AgilityLogs\SampleMVC\SampleMVC.log" emailErrors="false" /
      <errorTraceTypes>
        <add name="HTTP_Dangerous" typeName="System.Web.HttpException, System.Web, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b03f5f7f11d35a7a" exceptionPropertyName="Message" exceptionPropertyValueContains="A potentially dangerous " traceLevel="Verbose" />
      </errorTraceTypes>
      </trace>
    </settings>
  </agility.web>
</configuration>
```

Replace the “Insert Website Name Here” and “Insert Website Security Key Here” with the correct values as they appear in the Sample Site’s web.config. This will setup your authentication.
4. Next, add a reference to **Newtonsoft.Json** so we can use that library to serialize and deserialize JSON. If you do not have this reference handy, you can install it via Nuget by using the Package Manager Console command: `Install-Package Newtonsoft.Json`

5. Copy and paste the following code into your `Program` class:
static void Main(string[] args)
{
    //build your content item
    var product = new
    {
        Title = "Sample Product 1",
        ProductCategoryID = 1027, // "Processors"
        ProductCategoryTitle = "Processors",
        Price = 99,
        Summary = "This is a sample summary",
        Details = "This is a sample details"
    };

    //upload an image - replace the path with a real image on your computer
    Stream s = new FileStream(@"C:\temp\400x250.gif", FileMode.Open);
    string filename = "400x250.gif";
    string contentType = "image/gif";
    var uploadRetStr = ServerAPI.UploadMedia("ImportedImages", filename, contentType, s);
    var uploadRetObj = JsonConvert.DeserializeObject<APIResult<dynamic>>(uploadRetStr);
    if (uploadRetObj.IsError)
    {
        //handle error
    }
    else
    {
        //build attachment
        int mediaID = uploadRetObj.ResponseData.MediaID;
        string mediaUrl = uploadRetObj.ResponseData.Url;
        string thumbnailUrl = uploadRetObj.ResponseData.ThumbnailUrl;
        int size = uploadRetObj.ResponseData.Size;

        var attachments = new[] {
            new {
                originalName = mediaUrl,
                mimeType = contentType,
                fileSize = size, // file size
                managerID = "ListingImage",
                AssetMediaID = mediaID // media id of uploaded file
            }
        };

        //serialize the objects
        string contentItemStr = JsonConvert.SerializeObject(product);
        string attachmentsStr = JsonConvert.SerializeObject(attachments);

        //save the content
        var saveRetStr = ServerAPI.SaveContentItem(-1, //if updating an item, pass content item here.
                                                   "Products",
                                                   "en-us",
                                                   contentItemStr, attachmentsStr);
        var saveRetObj = JsonConvert.DeserializeObject<APIResult<int>>(saveRetStr);
        if (saveRetObj.IsError)
        {
            //handle error
        }
        else
        {
            int contentID = saveRetObj.ResponseData;
        }
    }
}
6. Replace the image path for a real image that exists on your machine.

7. Build the project and run the console app by clicking Start in Visual Studio. It should complete without error.

8. Navigate in Agility to your “Products” list and verify your product was imported. Also note that image you uploaded should be present in Media & Documents as well as attached to the “Listing Image” field of the product that you imported.
REST API

There are many cases where you may need to interact with your website content outside of your website code or through any Content Import context. Cases include mobile apps, traditional software applications, other websites etc. Using the REST API, you can easily integrate your Agility content into any client that has the ability to make calls over the web.

The REST API uses specific URL schemes to request data, and your webserver will return the appropriate content in a JSON format.

Content that can be retrieved via the REST API include:

- Content Lists
  - Returns a list
- Content Items
  - Returns a single content item
- Image Galleries
  - Returns a single gallery
- Settings
  - Default Language
  - Languages (list)
  - GlobalCss
  - TopScripts
  - BottomScripts
  - EnableOutputCache
  - OutputCacheSettings

*At the time of this writing, there is no current implementation to request Pages *

REST API Setup

All REST API requests need to be made directly to your web server which is assumed to be publicly accessible via a URL. There is no additional setup required. The default base API route is “~/api”, however this can be overridden by placing the following within your route table:

```csharp
ApiRouting.AddAgilityApiRoutes(routes, "custom-api-route");
```
The URL Schemes
Each request to the API must contain a URL scheme made up of the path which will dictate what type of content to retrieve as well as some required and optional querystring parameters.

**Required Querystring Parameters**
- **time** – Unix timestamp
- **hash** – SHA1 hash of the WebsiteName, SecurityKey, and time (Unix timestamp)

**Optional Querystring Parameters (Content Lists Only)**
- **sort** – i.e. “Title ASC”
- **filter** – standard DataView row filter syntax
- **skip** – integer
- **take** – integer (there is a maximum page size of 1000)

**Request Paths**

**Get Content List**
- /api/content/{languageCode}/{referenceName}

**Get Content Item**
- /api/content/{languageCode}/{contentID}

**Get Gallery**
- /api/galleries/{galleryID}

**Get Settings**
- /api/settings

**Sample Request:**
- http://yourwebsite.com/api/content/en-us/Blogs?sort=Title%20ASC&filter=Category%3DFun&skip=10&take=5&time=1411006077&hash=04323a2eff88e6b3e78eff0a2fdd10d0237a9715
Lab – Getting Content via the REST API

In this lab, we are going to use our existing **Console Application** that we created in the previous lab to demonstrate basic usage of the **REST API**. We are going to simply get our entire list of products and output the response in within the console window.

To complete this lab, you will need to have your Sample Site open in **Visual Studio**, as well as the **Console Application** open in another **Visual Studio** window.

You will also need to add a reference to **System.Net.Http** in order to make web requests.

1. In **Visual Studio**, in your **Console Application**, add a new reference to **System.Net.Http**.

2. Next, in your **Console Application**, comment out or delete the code that is currently in your main **Program** class.

3. Copy and paste following code into your **Program** class:
```csharp
//replace with your local site or stage server
string website = "http://localhost:59679";
string languageCode = "en-us";
string referenceName = "Products";
string websiteName = "James Sample MVC"; //replace with your site name
string securityKey = "insert security key here"; //replace with your securityKey

string url = string.Format("{0}/api/content/{1}/{2}", website, languageCode, referenceName);

//get unix timestamp
DateTime unixEpoch = new DateTime(1970, 1, 1, 0, 0, 0, DateTimeKind.Utc);
DateTime dt = DateTime.UtcNow;
long time = (long)(dt - unixEpoch).TotalSeconds;

//calculate hash
string hashSource = string.Join(".", new[] { websiteName, securityKey, time.ToString("d") });
SHA1 sha1 = SHA1.Create();
byte[] sha1Bytes = sha1.ComputeHash(Encoding.UTF8.GetBytes(hashSource));
StringBuilder sb = new StringBuilder();
foreach (byte b in sha1Bytes)
{
    sb.Append(b.ToString("x2"));
}
string hash = sb.ToString();

//set the required querystring params
string hashQ = string.Format("time={0}&hash={1}", time, hash);

//add the time and hash querystrings
url = string.Format("{0}?{1}", url, hashQ);

using (HttpClient client = new HttpClient())
{
    client.Timeout = new TimeSpan(0, 0, 60); //60 seconds
    var res = client.GetAsync(url);

    try
    {
        if (!res.Result.IsSuccessStatusCode)
        {
            dynamic err = Newtonsoft.Json.JsonConvert.DeserializeObject<dynamic>(content);
            Console.WriteLine(string.Format("An error occurred: {0}", err.Message));
            Console.ReadKey();
        }
        else
        {
            content = res.Result.Content.ReadAsStringAsync().Result;
            Console.WriteLine(content);
            Console.ReadKey();
        }
    } catch (Exception ex) {
        Console.WriteLine(string.Format("An error occurred: {0}", ex.ToString()));
        Console.ReadKey();
    }
}
```

4. Resolve any missing references you may have.

5. Run the console app. If everything is setup correctly, you should see something like this in the console output: