Daft Solutions

Daft.Server, Daft.lib

Daft.Server Architectural Overview And Programming Principals

Colin Manning
12/10/2009

Daft.Server Overview

Daft.Server is a middleware platform that provides access to the Canto Cumulus digital asset management (DAM) system. The platform utilises a RESTful web service approach to provide a common access layer to Cumulus (and potentially other DAM, Workflow or Content Management systems), and simplify the task of accessing Cumulus based repositories.

The Daft.Server is written in Java, and is deployed as a standard Apache Tomcat web application, with minimal configuration requirements. Via a set of client libraries, called Daft.lib (support for .NET, Java and PHP), it is possible to write high level software that acts as a client to the Daft.Server, to implement system integrations, content aggregation or web solutions/portals.

Why Use Daft.Server

Daft.Server is designed to receive requests and post data via simple HTTP Get, Put, Post and Delete messages. Therefore it provides a consistent and open platform for data communication between workflow, digital asset and content management systems using well understood and supported protocols.

The Daft.Server also abstracts the complexity of communicating directly with various systems by providing high level methods to access common functions. So for example instead of having to compose an SQL query to ask a DAM system for information, and then process the result set, using Daft.Server you construct a URL with a query name in the path, and some URL parameters if required, and the Daft.Server will return results in a standard structured form (JSON, or appropriate mime type if binary data requested). And if you use a Daft.lib API (e.g. Daft.lib for .NET), then you do not even have to construct the URL, as the API does all the work for you, and return suitable data structures for you.

Some Use Cases

1. System Integration
    a. A publishing company has a department for creating content to be published in print, on the web and to mobile devices. They utilize a workflow management system which supports multi-department content creation, and inclusion of advertisements, along with multi-channel delivery. However the workflow solution has minimal digital asset management capabilities – usually simple file system based organization of documents, images and articles.
    b. Daft.Server provides a plug-in for the workflow management system, that provides users of the workflow system a seamless way to access the Cumulus DAM system, including access to previews, asset upload, and check-in and check-out version control functions. Daft.Server also provides the ability to synchronize the workflow systems product structures with Categories, for context sensitive asset searching, and usage analysis.
    c. As a result, assets can be uploaded to Cumulus from various sources, and all the powerful features of Cumulus for gathering meta data from different file formats, along with fast and sophisticated query capabilities can be utilized. Also digital assets used in the publishing environment, can be re-purposed in the business environment (e.g. for marketing purposes), and Cumulus can provide a flexible archiving solution, again seamlessly accessible from the workflow system.
2. Content Aggregation

a. An e-Commerce company needs to provide dynamic access to information about products for sale via the company's web site. Product information and specifications exists in a Product Information Management (PIM) database, product photographs, including previews and variants are stored in an image database (Cumulus DAM). This content needs to be aggregated by a Content Management System (CMS) which is responsible for managing the dynamic web publishing, based on content, templates and end user queries.
b. Daft.Server provides a direct route into the DAM system via URLs, which can be embedded by the CMS in HTML "<img>" tags for example, for previews, variants etc.. The access to the DAM system is via unique product codes sources form the PIM system. Daft.Server can also deliver pre-build product templates constructed by the CMS and store them in the DAM system for faster retrieval of semi-static or static content.
c. As a result the CMS does not need to work intimately with the asset metadata, and potentially duplicate assets for use in dynamic publishing. Also the CMS can benefit from the storage of pre-built "page snippets", making publishing faster and can verify consistency of delivered content. The DAM system can in fact also be used to hold in a retrievable form snapshots of web content, which the DAM system can deliver on request.

Daft.Server 3.5

The current Daft.Server release is 3.5. This section outlines the architecture of the software, and the various components supplied. Note that if you wish to experiment with how various URLs function, we recommend using Firefox 3.x web browser, with the JSONView add-on. Browsers do not natively support content type of "application/json" by default, and so will request you save the returned content as a file – however JSONView in Firefox will present the returned JSON data in a clear readable format.

Daft.Server Configuration

The services listed below depend on various configuration settings which are defined in the Tomcat web application configuration, called web.xml. The Daft.Server comes with a heavily commented sample web.xml file, so we will not go into too much detail here. However in summary, here are the key concepts that are defined in web.xml, and are key to the workings of Daft.Server URLs.

1. Pool settings define how many Cumulus licenses are allocated to each catalog to be made available to the Daft.Server. Setting the license count to zero will result in one license being used – we can't do anything without one license. The pool-collection-count defines how many database collection clones will be made from each license – depending on the use cases, setting this value high or low may improve Cumulus server response times.
2. Catalog aliases (<catalog>) define the information required to reference a connection to a Cumulus catalog, including server name, username, password, and security settings). A connection pool is created for each catalog alias the first time a catalog alias is referenced RESTfully.
3. Catalog views (<view>) define views on a Cumulus catalog – pretty much as you might do with an SQL database view. The view name is referenced in various RESTful URLs. A view can be defined for one catalog, or if common to all catalogs, can be defined just once for all catalogs.
4. Named queries (<query>) provide a convenient way to make a Cumulus query via a RESTful URL. While the Metadata Service supports passing Cumulus queries directly as a URL parameter, this can be complex, and so if there are common queries, then they can be defined here, and referenced by name in the RESTful URL. Also you can specify parameters to the query, and simply pass those parameters via URL parameters, making it very flexible to construct dynamic queries.

Daft.Server Services

The following services are currently supported.

Preview Service

The Preview Service provides image previews from assets stored in Cumulus. Note that as the Preview Service utilizes the Cumulus preview mechanism, previews can be generated for any file format for which Cumulus can generate a preview (pretty much anything). Previews can be returned in different formats, and there are a number of different types of previews supported (see examples below).

Previews are stored in the Daft.Server persistent preview cache, so once generated (on first fetch, or pre-generated), download speed for previews is very fast.

If a named preview does not exist, then it can be created by specifying the relevant dimension in the URL, and from then on, only the preview name need be referenced. Resending the dimensions will not result in a re-generation, unless the "force" parameter is specified in the URL.

There are two special types of previews, which are not cached:

1. Thumbnail preview returns the thumbnail as generated by Cumulus and stored with the asset record.
2. Full preview returns a full preview of the asset, which of course may be quite large and time consuming to generate.

Here are some example URLs:

1. http://myserver:8083/daft/preview/sample/250/small?size=80
    a. Return a preview for asset with Cumulus record id 250 called small which is 80x80 pixels – will crop the edges to give best fit based on aspect ration
2. http://myserver:8083/daft/preview/sample/250/medium:120&format=jpg&compressionLevel=5
    a. Return a preview called medium, with a maximum size (width or height) of 120 pixels. Preview will be a JPEG (the default anyway), with compression level set to 5 (must be 1-10)
3. http://myserver:8083/daft/preview/sample/250/slide?w=300&h=200&format=png
    a. Return a preview called slide with specified width and height, and image format will be PNG
4. http://myserver:8083/daft/preview/sample/250/topleft?t=0&l=0&w=400&h=400
    a. Return a preview called topleft which is an unscaled crop of the asset preview
5. http://myserver:8083/daft/preview/sample/250/ slide?w=300&h=20&force
    a. Replace the preview named slid for asset with Cumulus record id 250
6. http://myserver:8083/daft/preview/sample/250/clearcache
    a. Remove all cached previews for asset with Cumulus record id 250
7. http://myserver:8083/daft/preview/sample/250/ slide
    a. Return the preview for asset with Cumulus record id 250 called small. If asset is not found, then no preview will be returned

Metadata Service

Provides query functions. Typically returns JSON structured representation of query results, and uses views, as defined in the configuration to determine the data to return.

A very important thing to remember is that if you want to map returned JSON data to objects in a programming language such a. "Record_Name"), or else it is not possible to map to variable names. To support this, all Metadata Service URLs can optionally take a parameter "csharp", whose existence will result in spaces being replaced with underscores – so having Cumulus fields spaces and underscores will not work (a pretty bad idea anyway). This is not important for PHP, but is important for C# and Javascript for example.

Here are some example URLs:

1. http://myserver:8083/daft/metadata/overview/record_range?p1=1&p2=500&f=1&c=25
    a. Return the first 25 assets whose Cumulus ids are between 1 and 500 (the query record_range being defined in web.xml – as in the sample web.xml provided)
2. http://myserver:8083/daft/metadata/details/quicksearch?text=Oslo Football&csharp
    a. Return all assets that match the Cumulus Quick Search query containing works specified in the text parameter. (Cumulus Quick Search is a special Cumulus query that can be configured to search in various fields in a catalog, and is very popular

with Cumulus users). This query will ensure returned JSON field names do not have spaces, and so can map onto class objects in C# or Javascript for example.

3. http://myserver:8083/daft/metadata/details/?q=Categories is "$Keywords:Sport:Skiing"
    a. Return all assets assigned to the specified category.

Describing Metadata

The Metadata Service has some special methods that support describing the structure of the data that can be returned. So an application can first call this to understand what is possible, and us it to determine how to process or visualize returned data. This data is currently returned in the form of JSON or C# code snippet.

Upload Service

The Upload Service is responsible for uploading assets and/or metadata to Cumulus. It uses the standard multipart POST protocol used in standard HTML file upload. It also supports only uploading metadata, which if course is only relevant when updating an existing asset (for now at least – Cumulus now supports records without asset data, so that could be interesting).

Asset Service

The Asset Service supports features for downloading assets, and will in the future support asset check in and check out for version control.

Daft.FileMonitor

The Daft.FileMonitor (currently Windows only) is a program that watches one or more hot folders for files, and takes appropriate action as follows:

1. If a folder is configured to process assets, then the assets are queued for cataloging to Cumulus. The configuration can contain some standard meta data to apply to all files in the folder, as well as a set of pre-prepared previews that will be created in the Daft.Server preview cache once the asset is cataloged successfully in Cumulus.
2. If a folder is configured to process metadata, then the files (must be XML according to a given schema) are processed, and the metadata extracted. One field should be a reference to a file to catalog, or be the id of an existing asset in Cumulus to be updated. As in case 1 above, previews can be generated.

The file monitor triggers on file creation, but cataloging will not start until the file is fully readable. So it is event driven, but in the case of very large file will wait till the file is fully copied before trying to catalog.

If a file fails to catalog successfully, this is logged, and the file is copied to a fail folder. If successful, this is also logged, and the file is copied to a success folder.

If there are files in the monitored folders when the process starts, they are processed immediately.

The file monitor can be configured to watch sub folders, in which case files going into sub folders are treated as are files in the parent folder.

Daft.Server 4

The next major release of the Daft.Server platform is version 4. There are a number of major technical enhancements planned to the platform. The current product road map means that this release will be available in February 2010.

Daft.Security

The current release of Daft.Server has minimal security features. All security is based on the security permissions bound to the user assigned to the various catalog aliases defined in the configuration. While this provides extensive control via Cumulus permissions, and LDAP mapping, it does not provide security with regard to what is sent via the URLS. Therefore the following features will be implemented (some are already in incubation). Also it is possible to define catalog aliases to be read only n the current release.

1. Secure HTTP (HTTPS) access.
2. Local authentication via LDAP (suitable for portals and implementation of single sign on (SSO) implementation.
3. Session based security, requiring that access to a URL is only via an authenticated session. SO the lifecycle would be:
    a. Authenticate a user over HTTPS
    b. If user authenticated, an authenticated session id is returned
    c. Access to catalogs defined to be secure require an authenticated session id, otherwise “Forbidden” HTTP response returned.
    d. Sessions can be configured to time out in the usual way, so URL access to Secure catalogs will be forbidden if session has timed out
    e. Best practice will be for user to log out, at which the session will be destroyed in the server.

Daft.MCAL (Managed Central Asset Location)

Current Scenario

Cumulus provides three ways to store assets:

1. Copy files to some place on your file system, and let Cumulus create a record that references the file in this place.
2. Tell Cumulus to copy a file to a central asset location (CAL) and create a record that references the file in the CAL. The CAL should be in a file system location with limited permissions for normal users.
3. Tell Cumulus to copy a file to a special CAL, that Cumulus calls Vault. The Vault, copies the file, and provides a number of advanced functions, such as CAL management (e.g. automatic folder creation based on asset count or time), and version control.

In each of the above cases, the decision as to which storage strategy is used is either defined for all assets in a given catalog, or based on the settings in a Cumulus asset handling set (AHS).

The Current release of Daft.Server can direct how Cumulus behaves, by passing in an AHS to the cataloging process. However a very common use case is to use option 2, but unfortunately Cumulus does not provide any management capabilities, so you could end up with a single folder having hundreds of thousands or even millions of files, unless you manually control the process, which is not

ideal. In these cases, it is not often appropriate to use Vault, as Vault has an overhead in cataloging new assets and retrieving stored assets, and also the files stored in a Vault are not in an easily accessible form.

MCAL Approach

Daft.Server will continue to support the native approaches of Cumulus, if that is required. However a Managed Central Asset Location (MCAL) will be implemented, to provide the following features:

1.
1. Support multiple MCALs to provide load balancing and potentially fail over
2. Each MCAL can have a number of "Zones", each with its own rules and behaviors
3. Rules/Behavior for a Zone can include:
    a. Auto generate folders after a number of days
    b. Auto generate folders when a maximum number of files exist in current cataloging folder
    c. Define folders that are sources for Cumulus assets, catalog files once they arrive in these folders. Optionally have Cumulus process the folder on a timed basis, thus utilizing Cumulus fast folder cataloging via multi-threading.
    d. Define folders that are "hot", and so can trigger behavior for external systems via scripts, or some validation/checking of files before moving to other folders in a Zone (possibly in another MCAL, possibly external (e.g. FTP)
    e. Integrate with Daft.FileMonitor, and potentially supporting remote MCALs that can be online or offline.

From a Cumulus point of view, an MCAL is just any old file system, and so careful structuring of this architecture, can make it easier to hot swap in another Cumulus server for fail over support.

Preview Cache Management

The current release has only rudimentary support for managing the preview cache. So when an asset is updated, if the asset already has previews, it is the responsibility of the calling program to clear out the old cached previews. There is support via the RESTful URL to do this (and the Daft.lib client libraries can drive this behavior).

In version 4 it is proposed to implement automated cache management, whereby the preview cache is kept in sync with updates to Cumulus. There are two levels of complexity:

1. When an asset is updated via Daft.Server, then cache management is not so complicated to implement, as the update process has a handle on the Cumulus id for the asset.
2. When an asset is updated outside of Daft.Server, e.g. directly in Cumulus client or via some other integration, then it is more complex, and it will be required to implement a triggering mechanism to call back to Daft.Server when asset updates occur.

Note that only updates to the actual asset itself – not metadata is relevant .

Also a more sophisticated preview cache structure is to be implemented to avoid worst case scenarios where cache folders can end up with very many files. However it is still intended to keep the cache structure as simple as possible.

Optimize Connection Pooling

There are various improvements that we know we can make to connection pooling, and we are also in discussions with Canto to merge our pooling concepts with the Canto pooling option, so we expect to make improvements in throughput for high volume throughput in this area.

License Management

This is rather more to do with Daft Solutions administration, but from Version 4, only properly licensed servers will be supported.

Programming With Daft.lib

Any programming language that provides support for sending HTTP Requests, and receiving HTTP Responses can utilise the services provided by the Daft.Server. In such a scenario, the programmer is responsible for generating the URLs to be transferred, and processing the returned JSON or binary data.

Daft Solutions also supply a number of high level programming interfaces that provide a "native" interface for the programmer, and takes care of URL generation etc. I some cases, the API will also synchronize the JSON data to native classes (assuming a defined pattern for naming is maintained).

Daft.Server also has a method that can be used to describe the available data sets, and therefore the structure of data that will be returned. At the moment, there are two flavors one that returns a JSON description, and one that returns C# description.

1. http://myserver:8083/daft/sample/describe - returns a description of all views available in formatted as JSON.
2. http://myserver:8083/daft/sample/describe?csharpe - returns a description of all views available in C#. In this case, the C# code returned is exactly what is required to support polymorphic binding of requested data to C# classes when using the Daft.lib (.NET) API.
3. In both cases, you can also ask for the description of a single view – so:
    a. http://myserver:8083/daft/sample/overview/describe
    b. http://myserver:8083/daft/sample/overview/describe?csharp

Daft.lib (.NET)

Getting Started

The .NET flavor of Daft.lib is implemented in C#, currently using the .NET Framework 3.5. Development is undertaken using Visual Studio 2008.

We tend to add new methods to the API based on customer requirements. Also note that we may refactor certain aspects for version 4, but these changes will be clear and not architecturally different from the current 3.5 version.

The library depends on an external DLL (open source, royalty free and supplied with the Daft.lib DLL) that implements the JSON serialization to C# classes. We have the source code to this library, so can adapt or fix it if necessary.

Finally note that the Daft.lib is already in production use, and so it is has been tested in real life scenarios. Also note the Daft.FileMonitor is implemented using Daft.lib for .NET, and we provide the source code for this application as a reference implementation.

Implementation Details

In version 3.5, pretty much all the methods and behavior is encapsulated in a single C# class called "Metadata" in the namespace "Daft.lib".

The class uses reflection to provide polymorphic features. So for example if you make a query and specify the "details" view, then the API can map the returned data to a class that has the appropriate fields defined. The Metadata Service "describe" method described above, used with the "csharp" parameter will generate the C# code for you (note the classes generated are LINQ compatible, and so can be used directly in LINQ queries. These classes are subclasses of the class "CumulusAsset", which holds the core asset fields.

These classes are placed in t he namespace "Daft.lib.metadata.views" by default, but you can change this. Note however that the metadata class needs to know the name of this namespace on startup, as it runs the reflection code on this namespace. Therefore the constructor for the class "Metadata" takes this namespace as a parameter, and it is critical that you do not have any other classes in the namespace used – so do not embed these classes in another namespace rt things may not work – best to use the default generated by the Metadata Service "describe" method,

Sample Code

Source code for two applications are provided. Description of this code is beyond the scope of this document. Please contact Daft Solutions to walk through the sample code. Also note the code is provided only as an example as to how to use the Daft.lib (.NET) API, and is not guaranteed to function, and we accept no responsibility for use of this code by any organization for any purpose.

Do note however the Daft.FileMonitor is a supported application, and while relatively complex, it is a good example of real life usage of the API. We will continue to develop this application, and further document it, and will continue to make the source code openly available.

1. TestCumulusWebService
    a. This application is a very simple test application, that uses the Daft.lib (.NET) API to test various functions of the API. The main program is pretty hard coded, and assumes files and folders that exist – you need to adapt this. The primary functionality is encapsulated in a class called TestRest, so you can jump straight to this class to see how things function. The class file "test.cs" holds the classes auto generated by the Metadata Service "describe" method. You will find lots of examples of uploading, downloading, updating assets, and working with Cumulus categories also.
2. Daft.FileMonitor Library
    a. This application provides a set of functions to support monitoring folders on a file system, and uploading the files and metadata to Cumulus. It also shows how previews can be generated. This code needs to be called from either a Windows console application or service. We can provide the source code to do this if it is useful, but it is more important to examine the code in this library.