MFG225143

API Enhancements in Vault 2019

Paul Gunn
Autodesk

Learning Objectives

- Learn about new API functionality available in Vault 2019
- Discover how this functionality was used to implement Project Sync
- Understand how these capabilities can be used in a custom application
- See the code behind and understand how it works

Description

This class will cover new API functionality in Vault 2019 software. This functionality was added to support the Project Sync feature but was designed to be used by any client-side Vault customization. Topics include entity attributes (which can associate programmatic data with any Vault object) and scheduled jobs (which allow execution of custom jobs at a given time and cadence).

Speaker(s)

Paul Gunn has been a member of the Vault team since 2003. Throughout that time, he has been involved in the development of many features of the product including security, search, and replication with a focus on server-side functionality. Paul currently serves as a software architect for Vault.

Overview

Project Sync was one of the more significant features implemented in Vault 2019. It allows users to sync files between Vault and the Autodesk cloud services such as Fusion Team and BIM Docs. At a high level, files and folders are mapped to and associated with folders in these cloud locations. The Job Processor (leveraging the Autodesk Desktop Connector) is then responsible for doing the work of transferring files between these locations.

This feature mainly consists of client-side functionality.
- The administrative UI to define the folder mappings and synchronization schedules
- Commands to do ad-hoc upload and download of files within a given folder mapping
- The job processor job to handle the actual work of uploading and downloading the files

That said, this feature required server-side support to store the cloud mappings and other behavior - as well as the ability to schedule a recurring job for execution. In considering these requirements, the team decided to implement the server-side support in a sufficiently general way that it could be used by other features in the future. This turned out to be a wise decision, as a number of other internal projects have also begun to leverage this functionality successfully.

My goal here is to tell you about these new APIs and to demonstrate them in a sample application. My hope is that you, as developers customizing Vault to your own needs, will find this functionality useful in your own projects. I hope that you walk away thinking about and excited about the possibilities.

Entity Attributes

To support the association of cloud mappings with folders, we came up with the concept of entity attributes. As you may know, in Vault every type of object (files, folders, items, etc.) are referred to as entities. You may have seen this in using the Property Service, which often takes an Entity Class as a parameter (for example getting property definitions by entity class). An entity attribute is essentially application-defined data that can be associated with any Vault entity ID. For example, a folder ID, a file ID, a file master ID, etc.

Entity attributes are not directly user-visible, though an application may expose the information via some UI. They exist only for programmers. Additionally, they are not considered part of history. Standard properties require a new check-in to be associated with a file. This is not the case with entity attributes. It is possible to set entity attributes on entities that are read-only and not otherwise modifiable. So, for example, in Project Sync, it is possible to set entity attributes with information about the cloud mapping on a folder that is in a Release state

One key concept of entity attributes is the idea of the namespace. This should seem familiar if you have ever worked with XML. A namespace is an application-defined string which segregates that application’s attributes from those of other applications. We recommend as a best practice that you include your company name or another unique qualifier as part of your

namespace. Inside of that namespace, on a given entity, you can have as many attributes as you prefer. These are essential name/value pairs that are defined entirely by you.

The following Property Service methods are available for getting and setting entity attributes. I should point out that these (and all other methods and objects I'll be discussing) are documented in the Vault SDK.

```
EntAttr[] FindAllEntityAttributes(string namespc);

EntAttr[] FindEntityAttributes(string namespc, string attribute);

EntAttr[] GetEntityAttributes(long entityId, string namespc);

void SetEntityAttribute(long entityId, string namespc, string attribute, string val);
```

The 'EntAttr' currency is documented as follows in the SDK:

The following tables list the members exposed by EntAttr.

Public Properties

	Name	Description
	Attr	User-defined name of the attribute.
	Cloaked	Is the entity cloaked for the current user.
	EntityId	Entity tagged with this attribute.
	Val	Value of the attribute.

Scheduled Jobs

If you are already a Vault developer, you may be familiar with the Job Service. The Job Service exposes a number of different methods to support the operation of the job processor. Many of these methods you will seldom use, as the functionality is handled for you automatically by the job processor (e.g., the reservation and completion of jobs). However, if you have ever implemented a custom job (aside from ones queued automatically on lifecycle transition) then you have probably used the AddJob method:

```
Job AddJob(string type, string desc, JobParam[] paramArray, int priority);
```

As a brief review or introduction, I'll summarize how this works. The job type is a string that uniquely identifies your job. Similarly, to namespace above, we recommend that you include

your company name or some other qualifier. The job type is associated with your custom job implementation inside your vcet.config file. This configuration is a subject in itself, so I won't go into great detail. The sample code includes a vcet.config file and the links below will have resources on how this existing functionality works.

I will just briefly summarize the other parameters here. Description is a user-visible description of the job as can be seen in the job queue. ParamArray is a set of name value-pairs for passing information from the code adding the job to the job handler. A common parameter might be the entity Id that you are operating on. The final parameter is the priority, which is an indication of which jobs should be privileged over others. The priority, however, is not a guarantee of order execution (especially when you have multiple job processors) – so please do not count on it to ensure one job runs before another.

The new scheduled job functionality is supported by the following service methods:

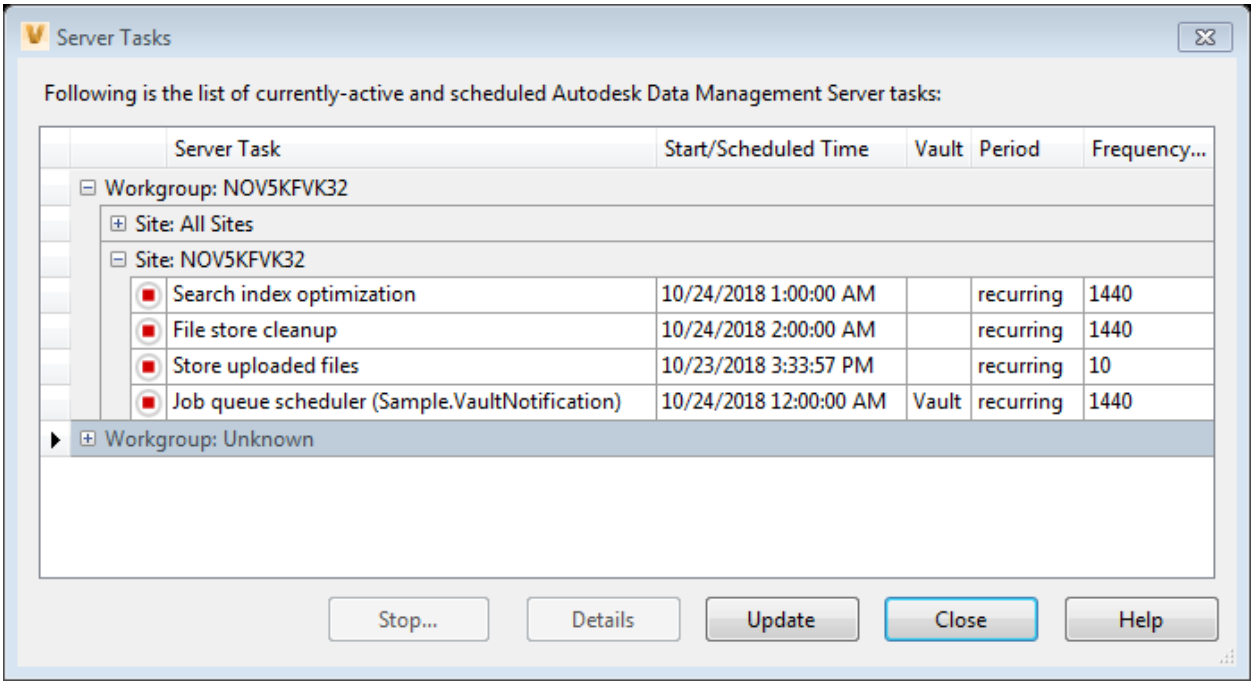
```
SchedJob AddScheduledJob(string type, string desc, JobParam[] paramArray, int priority,
System.DateTime execDate, int execFreqInMinutes);

void DeleteScheduledJob(long id);

SchedJob GetScheduledJob(long id);

SchedJob[] GetScheduledJobs();
```

I am going to focus here on the AddScheduledJob method, as the others are fairly self-explanatory. You may have noticed that the first four parameters are exactly the same as the existing AddJob method described above. This is intentional. When a scheduled job reaches its time of execution, a normal queued job is created with these settings and processed normally by the job processor.

The remaining two parameters determine the time and frequency of scheduling. The execDate supplies the date / time when the job should be first run. This could be immediate or sometime in the future. The execFreqInMinutes indicates the frequency at which the job should be run. One common usage would be 1440 minutes which is equivalent to daily execution. A frequency of zero indicates that the job should be run only once at the given date / time.

Once the job has been queued, neither the job processor or your job handler treats these types of jobs any differently than traditional jobs. In fact, the only way to know that a job was originally scheduled is that a `ScheduledJobID` parameter is available in the parameter list. Project Sync uses this information to delete the recurring job if the folder to sync had been deleted from Vault. Please note that the `ScheduledJobID` and the Job Id provided to your job handler will be different, as over time there can be multiple jobs queued from your scheduled job.

The scheduled job functionality relies on the same mechanisms as other background server tasks (such as nightly search index optimization). For the jobs to be queued, the Autodesk Job Dispatch Service should be running, and there can be up to a minute between creation of the scheduled job and queuing (assuming the job is for immediate execution). You can view the scheduled jobs along with other background server tasks in ADMS console.



The sample application

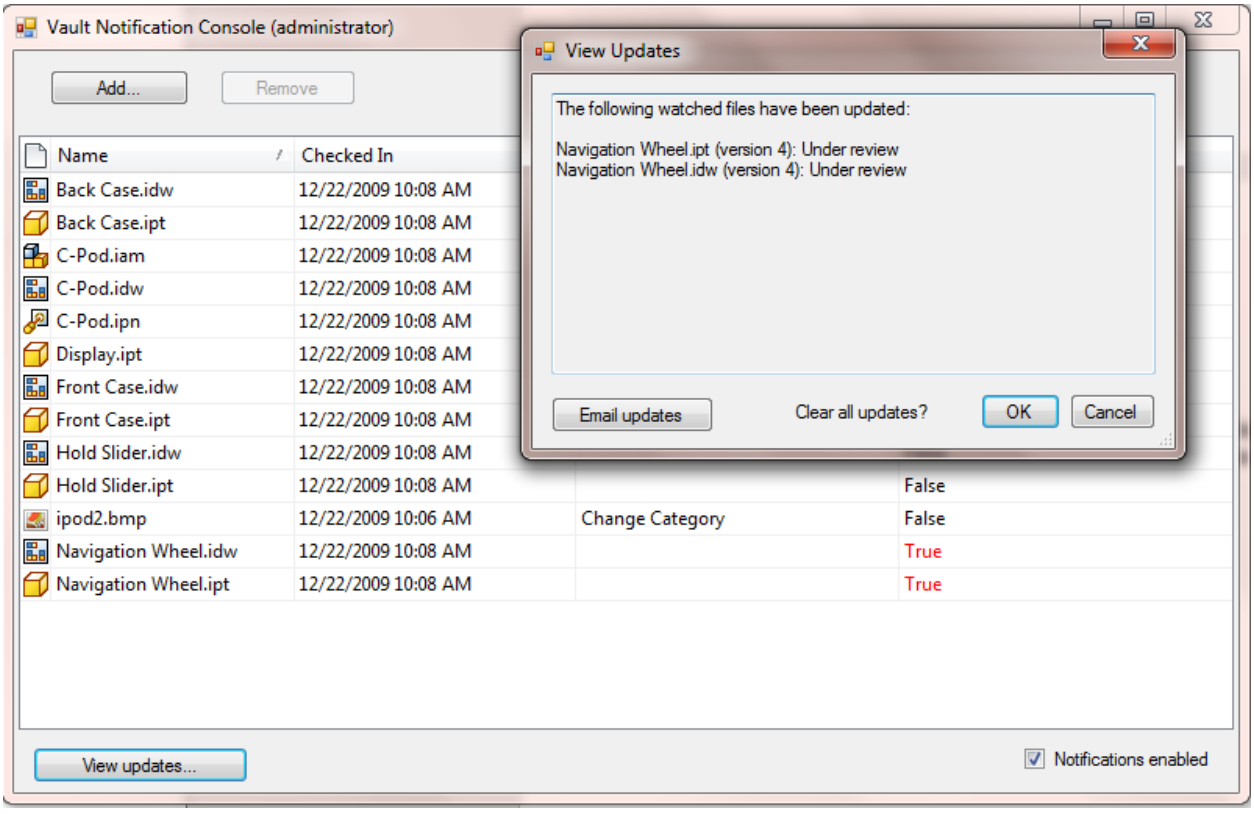
With this background in mind, we are ready to look at the sample application. I will give a brief description of the sample application before we dive into the code. The sample application allows the signed in user to create a notification list of files for which they would like change notifications. It consists of two parts:

(1) A Windows Forms application that supports adding and removing files, enabling / disabling the scheduled event, and being able to quickly view modifications that have been made.
(2) A job handler that sends out nightly email notifications with changes to files on the list. The job handler also automatically updates the notification list once notifications are sent out.

Please note that this coding project is an educational sample. As such, it has been implemented as simply as possible and lacks a number of desirable features:

- It assumes the email SMTP server is on 'localhost'
- It assumes the email addresses for vault user have been correctly configured
- A different scheduled job is created for each user, which could be inefficient
- The user must have permissions to create a scheduled job
- There are no options for including file dependencies and drawings

Below is a screenshot of the sample application.



In addition to the list above, another 'nice to have' would be the ability to add watched files via a Vault Explorer customization instead of using a separate application.

Reviewing the sample code

In discussing the code, I'm going to focus on the implementation of the notification list. Specifically: where the entity attribute methods are used to load and modify the notification list. In my view, this is the most interesting part of the code. While there is also an example usage of AddScheduledEvent, it is a one-liner and not very interesting.

How does the notification list work?

The notification list is implemented with a combination of two features. The entity attributes which we have discussed in some detail and a pre-existing feature known as Persistent Ids.

As a rule, it is not recommended to store file and other entity Ids across versions of Vault. Depending on new feature requirements, there is the possibility that these 'long' Ids could be modified as part of a data migration. Persistent Ids are string representations that are guaranteed to survive a data migration. With Persistent Ids, you can specify whether the Id represents a historical or tip iteration of the file – and that Persistent Id can at some later point be resolved to the actual Vault entity object.

In our particular case, we are generating historical Persistent Ids that resolve to a specific file iteration. These Persistent Ids are then added as Entity Attributes on the master Ids of files we want to get notification on. When we later resolve the Persistent Ids, we can simply check whether the resulting file objects are 'latest' or not. This gives us a quick indication of which files have been recently modified. To 'catch up' the notification list, we simply obtain Persistent Ids for the newer version of the file and re-add these as Entity Attributes.

How is the notification list loaded?

Here are the namespace and attribute name. You will note that the namespace is specific to the user logged into the application. This allows each user to have a separate notifications list.

```
private string AttributeNamespace
{
    get { return "Sample.VaultNotification." + UserName; }
}
private const string MostRecentAttribute = "MostRecentIterationId";
```

Here is the code for loading the notifications list:

```
public IEnumerable<IEntity> Load()
{
    var attributes = Connection.WebServiceManager.PropertyService.FindEntityAttributes
        (AttributeNamespace, MostRecentAttribute) ?? Enumerable.Empty<ACW.EntAttr>();

    attributes = attributes.Where(a => !a.Cloaked);

    if ( !attributes.Any() )
        return Enumerable.Empty<IEntity>();

    var resolvedIds =
Connection.PersistableIdManager.ResolvePersistableIds(attributes.Select(a => a.Val));
    return resolvedIds.Select( rid => rid.Value );
}
```

The first step is to get an array of all the attributes in the namespace (using `FindEntityAttributes`). We then filter out any entities that the user no longer has read access to. At that stage, if there are no attributes returned, we exit with an empty enumeration.

The val associated with the '`MostRecentIterationId`' attribute name is the Persistable Id for the historical iteration as of creation or last update. We pass an enumeration of these vals `ResolvePersistableIds` to obtain objects represented by those Persistable Ids. The `ResolvePersistableIds` methods returns a dictionary mapping the Persistable Ids to the resolved objects, so at this point we simply return an enumeration of those objects (the dictionary's Values).

One thing to note is that I personally make extensive use of Linq extensions like Select. This mechanism (though initially unfamiliar) avoids a lot of hand-coded For loops, etc. and in my opinion makes the programmer's intent much clearer. A reference for Linq and the specific extension methods used is available below.

How are files added to the notification list?

Here is the code for adding files to the notifications list:

```
public void Add(IEnumerable<IEntity> entities)
{
    var persistentIds = Connection.PersistableIdManager.GetPersistableIds(entities,
getLatest: false); // ensure we are getting historical IDs

    foreach (var current in persistentIds)
    {
        Connection.WebServiceManager.PropertyService.SetEntityAttribute
            (current.Key.EntityMasterId, AttributeNamespace,
              MostRecentAttribute, current.Value);
    }
}
```

This is essentially the opposite of the method we just looked at. Instead of loading and returning IEntity objects (in this case file iterations), we instead take an enumeration of IEntity objects as a parameter. At that point, we call `GetPersistableIds` which similarly returns a dictionary. Note that we are passing 'false' for getLatest, indicating that we want Persistent Ids for the specific historical iteration – rather than Persistent Ids that would always resolve to latest.

Given this dictionary of Persistent Ids, we simply iterate over the key/value pairs and create Entity Attributes for each using `SetEntityAttribute`. The main thing to note here is that we are creating the attribute on the file's master Id. That way, when we update the notifications list to represent latest versions, we will just be replacing the previous entry.

How are files removed from the notification list?

Here is the code for removing files from the notification list:

```
public void Remove(IEnumerable<IEntity> entities)
{
    foreach (var ent in entities)
    {
        Connection.WebServiceManager.PropertyService.SetEntityAttribute
            (ent.EntityMasterId, AttributeNamespace, MostRecentAttribute, null);
    }
}
```

This code is quite similar to the previous. The only real difference is that we are passing 'null' as the attribute's value. This has the effect of deleting the attribute, thereby eliminating the file from the notification list.

How is the notification list updated?

Here is the code of updating (catching up) the notification list:

```
public IEnumerable<IEntity> Update(IEnumerable<IEntity> entities)
{
    var modified = GetModified(entities);
    if( !modified.Any() )
        return Enumerable.Empty<IEntity>();

    var updatedFiles = Connection.FileManager.GetLatestFilesByIterationIds
       (entities.Select( e => e.EntityIterationId )).Values;
    Add(updatedFiles);
    return updatedFiles;
}
```

I am omitting the implementation of 'GetModified' here. Its role is to simply to filter and return IEntity files for which `IsLatestVersion` is false. That means that the IEntity represents an earlier version of the file and that there has been subsequent check-in activity since last time a notification was emailed out.

The logic then is to get all 'modified' entities. Assuming there are any, we get the latest file iteration for each via `GetLatestFilesByIterationIds`. Given this, we just have to use the existing 'Add' method previously discussed. The 'Add' method was written such that it will replace the pre-existing Entity Attribute if one exists.

Conclusion

This concludes our discussion and demonstration of these new API features. My hope is that you walk away from this class excited about the possibilities of using this functionality in your own projects. As mentioned, we at Autodesk have a number of in-development projects that use just this functionality. One key thing to note is that in Vault we have no secret server APIs. What we can use - you can use. Amaze us with what you can do.

Resources

The Vault software development kit
[Installed with Vault server] the SDK includes documentation and sample code

The Vault customization forum
https://forums.autodesk.com/t5/vault-customization/bd-p/301

Doug Redmond's video series on the Vault API (including implementing job handlers)
https://www.youtube.com/playlist?list=PLDA071BEB83F2713C&feature=view_all

.NET Linq functionality
Overview: https://www.oreilly.com/library/view/c-60-cookbook/9781491921456/ch04.html
Select: https://docs.microsoft.com/en-us/dotnet/api/system.linq.enumerable.select
Empty: https://docs.microsoft.com/en-us/dotnet/api/system.linq.enumerable.empty
Where: https://docs.microsoft.com/en-us/dotnet/api/system.linq.enumerable.where
Any: https://docs.microsoft.com/en-us/dotnet/api/system.linq.enumerable.any