

MFG468044

Hack the Vault Job Processor

Thomas Rambach
Corning

Learning Objectives

- Learn what is the Vault Job Processor
- Learn what are the limitations of the Job Processor
- Learn what you can you solve with this class
- Learn how to apply this class to real world examples

Description

Out of the box, the Vault job processor is a great way to keep your data updated. The job processor can update visualization files, create PDF files, and synchronize properties. The job processor also has many limitations. This course will show you how to hack your job processor through custom jobs to enable more-granular control of your design automation tools. Learn how to create a custom job-processor job, interact with the jobs by setting priorities for different job types, query the job queue, and add or remove jobs to the queue. With this additional functionality, you'll be able handle different types of files separately and take control of your Vault.

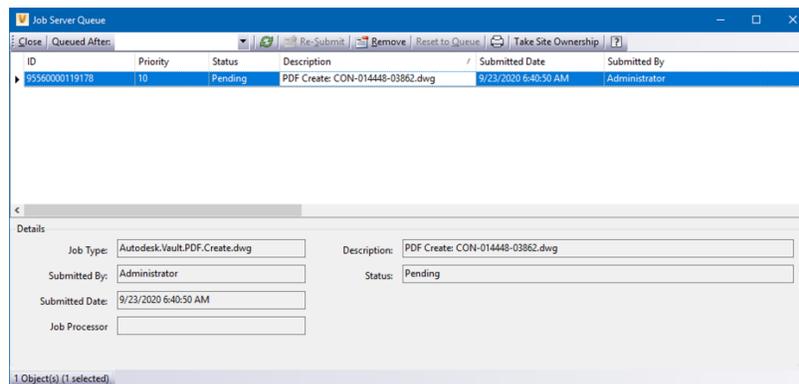
Speaker(s)

Thomas Rambach started his career in 1996 with Corning Optical Fiber division in Wilmington, NC as an Engineering Equipment Technician. Working with 2D CAD files, and diving into 3D with Mechanical Desktop, he hasn't looked back since. The next few years he spent working with some of the earliest releases of Inventor. From 2002-2006 moving to Flow Sciences as Drafting Manager where Vault and Productstream was adopted to manage day to day production activities. The next 11 years working for GE Nuclear in various capacities. First as a mechanical designer and then morphing roles into a software technical specialist. In 2017, he returned back to his roots at Corning with the role of CAD Systems Administrator where he manages CAD training, upgrades, user support, Vault sever infrastructure and a VR lab.

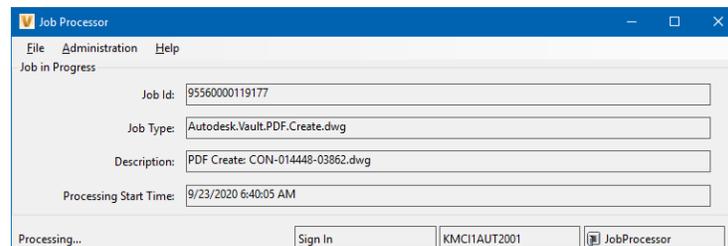
What is the Vault Job Processor?

“The Job Processor is a separate application that reserves queued jobs and pulls them from the job server to process them. Since the job processor is installed along with the Vault client, any workstation with the appropriate edition of Vault can be used to process jobs.”

Job Server Components:



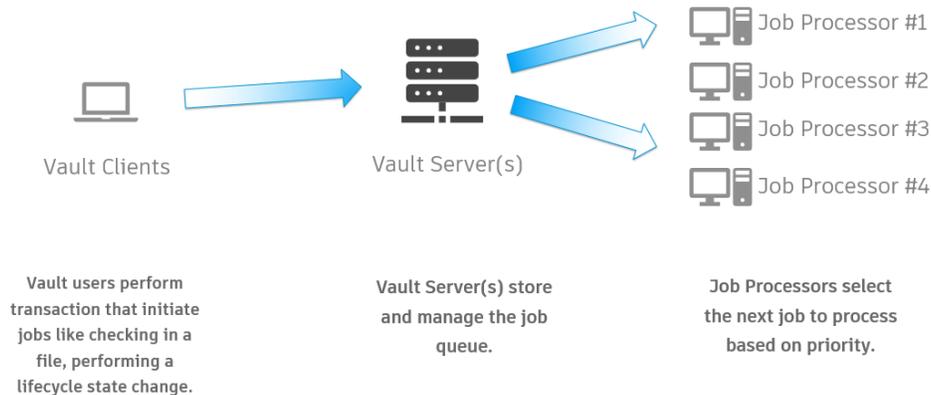
Vault Server: Job Processor Job Queue



Job Processor: Active Job

There's a lot of background activities that can occur in Vault. The Job processor is a separate application installed with the Vault client that processes queued jobs that are pulled from the vault server. The job processor can be installed and run on any workstation or server that you can install Vault on. So shown here is the job queue and an active job being processed by the job processor. Most of you familiar with Vault, should also be familiar with these two interfaces.

Typical Vault Job Processor Infrastructure:



The Vault clients feed the jobs into the vault job queue on the vault server. The job processors will reserve the next job to process based on priority of the submitted job.

How do you feed jobs to the Job Processor?

- On File Check-In

DWF Visualization files can be queued for creation on check-in of a file from within Inventor or AutoCAD.

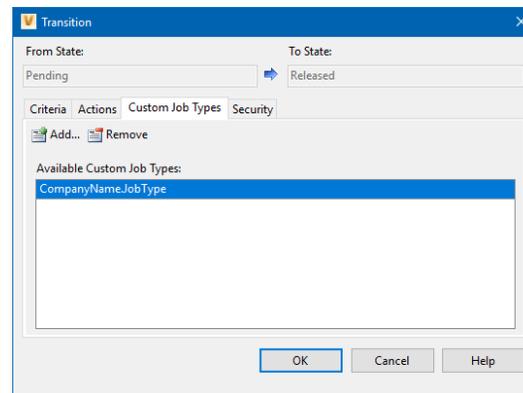
- On Lifecycle Transition

On lifecycle state change, actions can be performed using the job processor such as synchronizing properties, updating DWF, updating PDF.

- Manually

On launch of 'Update Visualization Attachment' a job will be created to update the DWF attachment or on manual selection of the PDF create command.

You can also add custom job types on lifecycle state transition



Custom jobs are added to the appropriate lifecycle transition as a custom job type. A developer creates the custom job type with matching name that the job processor runs.

Custom jobs are loaded from the Vault extension folder:

`%allusersprofile%\Autodesk\Vault[year]\Extensions\`

More Information:

<https://knowledge.autodesk.com/support/vault-products/learn-explore/caas/CloudHelp/cloudhelp/2020/ENU/Vault-Admin/files/GUID-A298690E-A937-4317-89C9-B04C9950DF2D-htm.html>

Creating your first custom Job Type

A great starting point with the basic framework necessary for creating your first custom job type is provided free by Marcus Koechl (Autodesk)

GitHub – Vault-Sample---InventorExportAnySampleJob

<https://github.com/koechl/Vault-Sample---InventorExportAnySampleJob/blob/master/Autodesk.VLTINVSrv.ExportSampleJob/JobExtension.cs>

What you'll need:

1. Download the GitHub project and place folder in Extensions directory.
2. Follow requirements on GitHub readme to use as-is.
3. Edit using Visual Studio. I recommend using VS2019.
4. Republish to Extensions folder on Job Processor.

** First: Use on development database using test data only ***

Vault has Limitations on How Jobs are Processed

- Default Vault Actions:
- Limited actions available. (Sync Properties, Update DWF, Update PDF)
- Custom Job Types:
- Runs on lifecycle transition (Ex: Review → Released).
- No way to differentiate by file type.
- Job will be created for all files that transition.
- Need to create different job processor extensions to handle different lifecycle transitions.
- Priority is always 100 and will run last after all other jobs are run in the queue.
- Jobs are processed in ascending priority order. 1 first, 2, 3, etc. up to 100 or more.

Solve the limitations

Breaking down the STEP Export Sample Job

```

34 namespace Autodesk.STEP.ExportSampleJob
35 {
36     public class JobExtension : IJobHandler
37     {
38         private static string JOB_TYPE = "Autodesk.STEP.ExportSampleJob";
39         private static Settings mSettings = Settings.Load();
40         private static string mLogDir = JobExtension.mSettings.LogFileLocation;
41         private static string mLogFile = JOB_TYPE + ".log";
42         private TextWriterTracelister mTrace = new TextWriterTracelister(System.IO.Path.Combine(
43             mLogDir, mLogFile), "mJobTrace");
44
45         #region IJobHandler Implementation
46         public bool CanProcess(string jobType)
47         {
48             return jobType == JOB_TYPE;
49         }
50     }

```

Look at JobExtension.cs. The method CanProcess returns the job type and determines if the extension is appropriate for the job being processed

You can see the JOB TYPE string, "Autodesk.STEP.ExportSampleJob" this is the value that you paste into the custom job action for the lifecycle transition. If the job processor picks up a matching job name, the job can be processed and the job is reserved for processing.

Execute the method that does the work

```

51     public JobOutcome Execute(IJobProcessorServices context, IJob Job)
52     {
53         try
54         {
55             FileInfo mLogFileInfo = new FileInfo(System.IO.Path.Combine(
56                 mLogDir, mLogFile));
57             if (mLogFileInfo.Exists) mLogFileInfo.Delete();
58             mTrace.WriteLine("Starting Job...");
59
60             //start step export
61             mCreateExport(context, Job);
62
63             mTrace.IndentLevel = 0;
64             mTrace.WriteLine("... successfully ending Job.");
65             mTrace.Flush();
66             mTrace.Close();
67
68             return JobOutcome.Success;
69         }
70         catch (Exception ex)
71         {
72             context.Log(ex, "Autodesk.STEP.ExportSampleJob failed: " + ex.ToString() + " ");
73
74             mTrace.IndentLevel = 0;
75             mTrace.WriteLine("... ending Job with failures.");
76             mTrace.Flush();
77             mTrace.Close();
78
79             return JobOutcome.Failure;
80         }
81     }
82 }
83

```

Execute will run the job and return the JobOutcome, success or failure from the mCreateExport method. Next you'll see the job is executed and mCreateExport is called which will return either a success or failure. The failures are logged to the job processor using the context.log statement.

Some job filters are built into the sample already

| | |
|--|---|
| <pre> 119 // only run the job for files 120 if (mEntClsId != "FILE") 121 return; </pre> | <p>Only runs the job against files. Not inadvertently applied to other entity types such as Items, Folders, etc.</p> |
| <pre> 123 // only run the job for ipt and iam file types, 124 List<string> mFileExtensions = new List<string> { ".ipt", ".iam" }; 125 ACM.File mFile = mMgr.DocumentService.GetFileById(mEntId); 126 if (!mFileExtensions.Any(n => mFile.Name.Contains(n))) 127 { 128 return; 129 } </pre> | <p>Filters the job to only run Part (.ipt) or Assembly (.iam) files.</p> |
| <pre> 132 // apply execution filters, e.g., exclude files of classification "Substitute" etc. 133 List<string> mFileClassific = new List<string> { "ConfigurationFactory", "DesignSubstitute", "DesignDocumentation" }; 134 if (mFileClassific.Any(n => mFile.FileClass.ToString().Contains(n))) 135 { 136 return; 137 } </pre> | <p>Further filters out any file classified as a ConfigurationFactory, DesignSubstitute or DesignDocumentation</p> |

Refer to the mCreateExport method. Allows to fine tune a custom job to only process files you want processed by a job that runs against all files being transitioned. There's already some filtered built into the sample, if the entity type is not a file, such as an item or folder... the job will not continue. This sample also filters to only run against files with the extension ipt or iam. Lastly, it skips any file classified as a configuration factory, design substitute or design document.

What other filters can be applied?

Lifecycle state

What if you want to handle the job differently if applied to multiple lifecycle state transitions? Instead of creating multiple extensions, you can create a lifecycle state filter.

Let's first lay some "context":

If you notice from the previous sample, the Execute method passes context and job to the mCreateExport method. Context is the Job Processor which houses the current connection. The job, is the job currently being processed:

```
//start step export
mCreateExport(context, job);
```



| | |
|---------|---|
| context | <ul style="list-style-type: none"> ▪ Connection ▪ Errors ▪ InventorObject |
| Job | <ul style="list-style-type: none"> ▪ Description ▪ Id ▪ JobType ▪ Params ▪ Priority ▪ VaultName |

The connection holds the keys to the kingdom

The connection allows access to all of the Vault services

You can use the various services to interact with Vault and perform actions on the file during the processing of the job.

| | |
|------------------|---|
| AdminService | Contains methods for manipulating users and groups. |
| AnalyticsService | Contains methods for Analytics within a vault |
| BehaviorService | Contains methods for manipulating behaviors. |
| CategoryService | Contains methods for manipulating categories. |

| | |
|---------------------------|---|
| ChangeOrderService | Contains methods for creating and manipulating change orders. |
| CustomEntityService | A collection of methods related to the Custom Entity entity type. |
| DocumentService | Contains methods for manipulating files and folders within a vault. |
| DocumentServiceExtensions | Contains more methods for manipulating files and folders within a vault. |
| ItemService | Contains methods for manipulating items. |
| JobService | Contains methods for manipulating the job queue. |
| LifeCycleService | Contains methods related to the lifecycle behavior. |
| NumberingService | Contains methods for retrieving and manipulating Numbering Schemes and configured Numbering Providers |
| PropertyService | Contains methods for manipulating properties on Entities. |

Lifecycle State Filter

Get the file to process by picking up the jobs entity ID:

```
using ACW = Autodesk.Connectivity.WebServices;
```

```
Connection connection = context.Connection;
```

```
Autodesk.Connectivity.WebServicesTools.WebServiceManager mWsMgr = connection.WebServiceManager;
```

```
ACW.File mFile = mWsMgr.DocumentService.GetFileById(mEntId);
```

Now get the file lifecycle state name:

```
mFile.FileLfCyc.LfCycStateName;
```

Pulling the job entity id, important to understand.... this is the files version id number, not the files master ID number. we can get the file using the ID number, from there we can pull in the files lifecycle state name.

What can you do now that you have the lifecycle state name?

```
String sLifecycleState = mFile.FileLfCyc.LfCycStateName;
switch (sLifecycleState)
{
    Case "Released":
        // Do something if a file is released
        break;
    Default:
        // Do something different if the file is not released
        break;
}
```

The lifecycle state filter, pulling the job entity id, important to understand.... this is the files version id number, not the files master ID number. we can get the file using the ID number, from there we can pull in the files lifecycle state name.

Category Filter

Get the file category and do something different based on the category name

```
String sFileCategory = mFile.Cat.CatName;
switch (sFileCategory)
{
    Case "Engineering":
        // Do something if a file is an engineering category
        break;
    Case "Document":
        // Do something different if the file is a document category
        break;
}
```

Like lifecycle state, we can do the same type of filter for categories. This is one more way to make your job processor do more with less where the single extension can handle multiple lifecycle states and categories. Again, we're using the referenced file, mFile and pulling in the category name.

Property Filter

There are 2 ways to get properties of a file.

1. Vault Properties : Retrieves the properties of a file as they are stored within Vault.
2. File Properties : If the Job Processor downloads the file for processing, the file properties can be retrieved directly for processing regardless of if they are indexed in Vault.

There really are 2 ways to retrieve properties of a file. Vault properties are the properties of the file as they are stored within Vault. And file properties, are if you job processor extension downloads the file for processing (as the example code does) the file properties can be retrieved directly for processing regardless if they are indexed in Vault. We'll cover vault properties today. Reading properties is fairly straightforward.

First you have to define the property definition for the property you want to retrieve:

```
// Define the Files Properties to Retrieve
PropDef[] docPropDefs = mWsMgr.PropertyService.GetPropertyDefinitionsByEntityClassId("FILE");
PropDef DescriptionDef = docPropDefs.Single(n => n.DispName == "Description");
```

Get the files properties for the specific property definitions:

```
fileProperties = PropertyService.GetProperties("FILE", new long[] { fFileID }, new long[] {DescriptionDef.Id,
AnotherProperty.Id});
```

Then get the property value if the property name matches the property name you want by cycling through them:

```
// Assign the Property Values
foreach (var key in fileProperties)
{
    string propDisplayName =
context.Connection.PropertyManager.GetPropertyDefinitionById(key.PropDefId).DisplayName;
    if (propDisplayName == DescriptionDef.DispName)
    {
        fileDescription = key.Val?.ToString() ?? "";
    }
}
```

First you have to define the property definition you want to retrieve. In this example, we're getting all the file property definitions and filtering to the property named "description."

This Works for both User defined properties and system properties. Once you retrieve the property definition, get the files properties for the specific property definitions.

Then cycle through all the properties retrieved for the file and match where the property name is equal and assign the property value to a string or other value type accordingly, in this case description.

Here we're setting the description to an empty value if the property is not found which can be useful for custom property definitions that may or may not be associated with every file in your vault.

Update a property

Get the files master ID:

```
mMasterID = mFile.MasterId;
```

Master ID = The ID# of the file regardless of version in the Vault.

File ID = The ID# of the specific file version in the Vault.

Update the file properties by passing in the property definition ID and the new property value:

```
context.Connection.WebServiceManager.DocumentService.UpdateFileProperties(new long[] { mMasterID },
    new PropInstParamArray[] { new PropInstParamArray() { Items = new PropInstParam[] { new
PropInstParam() { PropDefId = DescriptionDef.Id, Val = "New Description" } } } });
```

The JobService Allows Direct Manipulation of the Job Processor

| | |
|--------------------|--|
| AddJob | Adds a new job to the queue |
| AddScheduledJob | Adds a scheduled job with given execution date and frequency. |
| DeleteJobById | Deletes a job from the queue. |
| DeleteScheduledJob | Deletes the given scheduled job. |
| GetJobsByDate | Get all jobs from the queue queued on or after the specified start date. |
| GetJobQueueEnabled | Tells if the job queue is enabled. |
| GetScheduledJob | Gets information about the given scheduled job. |
| GetScheduledJobs | Gets information about all scheduled jobs. |
| ReserveNextJob | Reserve the next job in the queue |
| ResubmitJob | Resubmit a job of the specified Id to the queue. |

`SetJobQueueEnabled` Enables or disables the job queue.

`UpdateJobFailure` Inform the job queue that the client was unable to complete the job.

`UpdateJobSuccess` Inform the job queue that the job was successfully completed.

The job service allows for direct manipulation of the job processor. The job service in vault has several useful tools that can be used to interact with the job queue.

Add Job

You can add new jobs to the Job Processor with values you specify.

```
Public Function AddJob( _
    ByVal type As System.String, _  The Job Name. This sets what Job Processor extension will pickup the job.
    ByVal desc As System.String, _  Job description can be whatever you want to help identify it in the queue.
    ByVal paramArray() As JobParam, _  Job parameters are values passed into the job. This is powerful, you'll see.
    ByVal priority As System.Integer _  Job priority can be anything you want. 1-100 (actually 1-999).
) As Job
```

Add Job: Parameters

What are Job Parameters? MUST BE PASSED AS STRINGS

```
Job oJob = e.Context.Application.Connection.WebServiceManager.JobService.AddJob("CompanyName.JobType",
```

```
"Custom Job Name: " + mFile.Name, new JobParam[] { param1, param2, param3 }, 50);
```

Define parameters to pass. The sky is the limit:

```
JobParam param1 = new JobParam()
```

```
JobParam param2 = new JobParam();
```

```
param1.Name = "EntityId";
```

```
param1.Val = oLatestFileVer.Id.ToString();
```

```
param3.Name = "EntityClassId";
```

```
Param3.Val = "FILE";
```

You can pass in anything you like. Category Name, Lifecycle State, Properties, Etc. These are values that can be passed to your job as an array. These can be whatever you want without limits, but are also passed as strings that must be converted to other types as needed. We're creating 2 parameters and passing in the entity ID# and the entity class, of file. You could pass in anything you want, like file category, lifecycle state, or just a static value.

You can also pass in a job type parameter to allow your single job processor extension to process multiple job types:

```
JobParam param4 = new JobParam()

param4.Name = "JobType";

param4.Val = "UpdateProperties"
```

| | |
|---|--|
| <p>Get the Job Type Parameter</p> <pre>// Get the Job Type from the Parameters try { oJobType = job.Params["JobType"]; } catch (Exception ex) { oJobType = "Default"; }</pre> | <p>Process based on the Job Type value</p> <pre>// Update The File Properties Job if (oJobType == "UpdateProperties") { UpdateProperties(context, job); } // Release File Job if (oJobType == "ReleaseFile") { ReleaseFile(context, job); }</pre> |
|---|--|

Is the File Still Being Processed?

If you queue a job for a file of the same job name that's already in the queue, what happens?

An error will be returned when this happens. You can graciously handle the error and duplicate jobs would not be added.

```
try{

Job oJob = e.Context.Application.Connection.WebServiceManager.JobService.AddJob("CompanyName.JobType",
"Custom Job Name: " + mFile.Name, new JobParam[] { param1, param2, param3 }, 50);

}

catch (Exception ex)
{

// It's a dupe. I don't care about no stinkin' error

}
```

What if you want to really know if it's in the queue?

In this example, we get an array of 5000 of the jobs submitted in the last day (we know 5000 is a safe number that gets all jobs). We then see if the job has the same filename in the description.

```

Job[] oJobList = JobSvc.GetJobsByDate(5000, DateTime.Today.AddDays(-1));

if (oJobList != null)
{
    foreach (Job pJob in oJobList.ToArray())
    {
        if (pJob.Descr.Replace("Custom Job Name: ", "") == mFileName)
        {
            Log.WriteLog(fFileName + " job still in queue as Job#: " + pJob.Id.ToString() + " Status: " +
pJob.StatusMsg);
        }
    }
}

```

One challenge you may face when creating jobs through code is determining if the job is already in the queue and if it's still being processed. When creating a job where the job may already exist, an error would be generated and you can catch the error and handle it gracefully.

In other cases, you may want to query the job queue to determine if the job is still in the queue. In this example, we get up to 5000 jobs submitted in the last day. A value must be passed and in this example, we know we never have more than 5000. I like to filter the queue by specific job description as you can see we're pulling out the job name and matching the filename to find the match. There are multiple ways to accomplish this, this is my preferred method that works for me.

If you're developing an application and want to know if the job is really done, you could check the job queue but that's not always reliable.

For custom Jobs, it's helpful if result is apparent:

- Property is updated
- File is moved
- State is changed
- Etc.

Download my GitHub Template

Right-Click to Add to Job Queue and Process Extension

- Creates a menu command when selecting files to add them to the job queue.
- Contains a Job Processor extension to run the jobs.
- Use it as a starting point.
- Be creative.
- Don't screw up your data!

<https://github.com/cadtoolbox/MFG468044>