



# Rhozet<sup>®</sup> WFS

## Rhozet Workflow System API

Version 1.3

Rev A  
Manual Part No. RHZ-WFS-API-0001

## Disclaimer

Harmonic reserves the right to alter the equipment specifications and descriptions in this publication without prior notice. No part of this publication shall be deemed to be part of any contract or warranty unless specifically incorporated by reference into such contract or warranty. The information contained herein is merely descriptive in nature, and does not constitute a binding offer for sale of the product described herein. Harmonic assumes no responsibility or liability arising from the use of the products described herein, except as expressly agreed to in writing by Harmonic. The use and purchase of this product do not convey a license under any patent rights, copyrights, trademark rights, or any intellectual property rights of Harmonic. Nothing hereunder constitutes a representation or warranty that using any products in the manner described herein will not infringe any patents of third parties.

## Trademark Acknowledgments

© 2010 Harmonic Inc. All rights reserved. Harmonic, the Harmonic logo, Rhozet and the Rhozet logo are registered trademarks or service marks of Harmonic Inc. in the United States and other countries. Other company, product and service names mentioned herein may be trademarks of their respective owners. All product and application features and specifications are subject to change at Harmonic's sole discretion at any time and without notice.

## Documentation Conventions

This manual uses special formatting to call your attention to important information. The following conventions are used throughout this manual:

---

**IMPORTANT:** Important calls your attention to information that you need to be aware of before proceeding with the next step.

---

---

**NOTE:** Note calls your attention to additional information that you will benefit from heeding. It may be used to call attention to an especially important piece of information you need, or it may provide additional information that applies in only some carefully delineated circumstances.

---

In addition to these symbols, this manual uses the following text conventions:

- **Links:** indicates cross reference links.
- **Screen Output:** shows console output or other text that is displayed to you on a computer screen.
- **Italics:** used for emphasis, file names, and document references.

---

## History

Date	Version	Modification from Previous Version
03/26/2009	Draft 0.1	Initial Version
04/13/2009		Added Startup and Ui Configuration notes
05/18/2010	Released 1.0	Added sections from AParmar and Handoyo and API Reference
05/25/2010	Version 1.0	Added Job notes and some clarifications
06/21/2010	Versions 1.1	Modifications for 1.1 Release
06/28/2010	Version 1.1	Added WCF port reference and Ignored state update.
07/10/2010	Version 1.2	Added workflowQueueJob, added sequence for task fetching and updates
09/13/2010	Version 1.3	Added QueueWorkflowJobEx, QueueWorkflowTemplateJob, GetJobPageBySchedule, GetTaskPageBySchedule, GetTaskCount API Functions.
09/15/2010	Version 1.3	Added GetJobAssets API Function.
11/11/2010	Version 1.3	Added GetWatchFolderById Function.

## Table of Contents

<i>Disclaimer</i> .....	2
<i>Trademark Acknowledgments</i> .....	2
<i>Documentation Conventions</i> .....	3
<i>History</i> .....	4
TABLE OF CONTENTS.....	5
<b>CHAPTER 1: RHOZET WFS ARCHITECTURE</b> .....	<b>9</b>
1.1: TERMS AND ABBREVIATIONS.....	9
1.2: INTRODUCTION .....	9
1.3: WORKFLOW AND STATE .....	11
1.4: SERVER SIDE.....	12
<i>Processing Task State</i> .....	13
<i>Notes on Completed</i> .....	15
1.5: REDUNDANCY .....	15
<i>WFS Controller</i> .....	15
<i>Nodes</i> .....	16
1.6: CLIENT SIDE.....	16
<i>Carbon Server 3. x API</i> .....	16
<i>API based client</i> .....	17
<i>WFS Framework based client</i> .....	17
1.7: WCF INTERFACES.....	23
1.8: DATA STRUCTURES.....	23
<i>Rhozet.ApiLib.Fetcher</i> .....	24
<i>Rhozet.ApiLib.Software</i> .....	25
<i>Rhozet.ApiLib.Hardware</i> .....	25
<i>Rhozet.ApiLib.MachineStatus</i> .....	26
<i>Rhozet.ApiLib.Job</i> .....	27
<i>Rhozet.ApiLib.Task</i> .....	28
<i>Rhozet.ApiLib.ParentList</i> .....	29
<i>Rhozet.ApiLib.Parent</i> .....	29
<i>Rhozet.ApiLib.Child</i> .....	30
<i>Rhozet.ApiLib.DataObject</i> .....	30
<i>Rhozet.ApiLib.Asset</i> .....	30
<i>Rhozet.ApiLib.Preset</i> .....	31
<i>Rhozet.ApiLib.Module</i> .....	31
<i>Rhozet.ApiLib.ModuleScheme</i> .....	32
<i>Enumerations</i> .....	33
<i>Rhozet.ApiLib.FetcherType</i> .....	33
<i>Rhozet.ApiLib.JobType</i> .....	34
<i>Rhozet.ApiLib.JobStatus</i> .....	34
<i>Rhozet.ApiLib.TaskType</i> .....	35
<i>Rhozet.ApiLib.TaskStatus</i> .....	36
<i>Rhozet.ApiLib.PresetType</i> .....	37
<i>Rhozet.ApiLib.ModuleSubType</i> .....	37

Rhozet.ApiLib. ModuleType .....	37
1.9: CONSTRUCTING WORK - JOBS AND TASKS .....	38
<b>CHAPTER 2: API REFERENCE .....</b>	<b>41</b>
2.1: STATE.....	41
<i>Version</i> <i>GetVersion()</i> .....	41
<i>int</i> <i>ConnectionHealth()</i> .....	41
2.2: LOGS.....	41
<i>Guid</i> <i>StoreLog(ErrorLog log)</i> .....	41
<i>bool</i> <i>StoreLogs(IEnumerable&lt;ErrorLog&gt; log)</i> .....	41
<i>ErrorLog</i> <i>GetErrorLog(Guid id)</i> .....	42
<i>IEnumerable&lt;ErrorLog&gt;</i> <i>GetErrorLogbyReferenceId(Guid id)</i> .....	42
<i>IEnumerable&lt;ErrorLog&gt;</i> <i>GetErrorLogPage (int start, int count, long since)</i> .....	42
<i>IEnumerable&lt;ErrorLog&gt;</i> <i>GetErrorLogList (long since)</i> .....	42
2.3: TASK PROPERTY .....	42
<i>Task</i> <i>Property</i> contains the task results (for example, <i>Media Evaluate Task</i> ) and important data of the task which is completed.....	42
<i>Guid</i> <i>StoreTaskPropertyEx (TaskPropertyEx property)</i> .....	44
<i>bool</i> <i>StoreTaskPropertiesEx(IEnumerable&lt;TaskPropertyEx&gt; properties)</i> .....	44
<i>TaskPropertyEx</i> <i>GetTaskPropertyExById(Guid id)</i> .....	44
<i>IEnumerable&lt;TaskPropertyEx&gt;</i> <i>GetTaskPropertyExByReferenceId (Guid id)</i> .....	44
<i>IEnumerable&lt; TaskPropertyEx&gt;</i> <i>GetTaskPropertyExByRefNamSeq (Guid id, string name, int sequence)</i> .....	44
<i>IEnumerable&lt;TaskPropertyEx&gt;</i> <i>GetTaskPropertyExPage(int start, int count, long since)</i> .....	45
<i>IEnumerable&lt; TaskPropertyEx &gt;</i> <i>GetTaskPropertyExList (long since)</i> .....	45
2.4: PRESETS .....	45
<i>bool</i> <i>RemovePreset(Guid id)</i> .....	45
<i>Guid</i> <i>StorePreset(Preset Preset)</i> .....	45
<i>Preset</i> <i>GetPreset(Guid guid)</i> .....	46
<i>IEnumerable&lt;Preset&gt;</i> <i>GetPresetList(long since)</i> .....	46
<i>IEnumerable&lt;Preset&gt;</i> <i>GetPresetListByType(PresetType type, long since)</i> .....	46
<i>IEnumerable&lt;Preset&gt;</i> <i>GetPresetListByCategory(string category, PresetType type, long since)</i> .....	46
<i>IEnumerable&lt;Preset&gt;</i> <i>GetPresetPage(int start, int count, long since)</i> .....	46
<i>IEnumerable&lt;Preset&gt;</i> <i>GetPresetPageByType(PresetType type, int start, int count, long since)</i> .....	47
<i>IEnumerable&lt;Preset&gt;</i> <i>GetPresetPageByCategory(string category, int start, int count, long since)</i> .....	47
<i>IEnumerable&lt;Preset&gt;</i> <i>GetDistinctPresetTypes()</i> .....	47
2.5: MODULES .....	47
<i>bool</i> <i>RemoveModule(Guid id)</i> .....	47
<i>Guid</i> <i>StoreModule(Module Module)</i> .....	48
<i>Module</i> <i>GetModule(Guid id)</i> .....	48
<i>IEnumerable&lt;Module&gt;</i> <i>GetModuleList (long since)</i> .....	48
<i>IEnumerable&lt;Module&gt;</i> <i>GetModuleListBySubType (ModuleSubType subtype, long since)</i> .....	48
<i>IEnumerable&lt;Module&gt;</i> <i>GetModulePage (int start, int count, long since)</i> .....	48
<i>IEnumerable&lt;Module&gt;</i> <i>GetModulePageBySubType (ModuleSubType subtype, int start, int count, long since)</i> .....	49
<i>IEnumerable&lt;ModuleSubType&gt;</i> <i>GetDistinctModuleSubTypes (ModuleSubType subtype)</i> .....	49
2.6: MODULESCHEME .....	49
<i>bool</i> <i>RemoveModuleScheme (Guid id)</i> .....	49
<i>Guid</i> <i>StoreModuleScheme (ModuleScheme Module)</i> .....	49
<i>ModuleScheme</i> <i>GetModuleScheme (Guid guid)</i> .....	50
<i>IEnumerable&lt;ModuleScheme&gt;</i> <i>GetModuleSchemeList ()</i> .....	50
<i>IEnumerable&lt;ModuleScheme&gt;</i> <i>GetModuleSchemePage (int start, int count)</i> .....	50

2.7: FETCHERS .....	50
<i>bool EnableFetcher (string fetcherId, bool enabled)</i> .....	50
<i>bool EnableFetcher (bool enabled)</i> .....	51
<i>bool IsFetcherEnabled ()</i> .....	51
<i>bool IsFetcherEnabled (string fetcherId)</i> .....	51
<i>string RegisterFetcher (FetcherType type, TaskType[] roles)</i> .....	51
<i>string RegisterFetcher (FetcherType fetcher)</i> .....	51
<i>bool UpdateFetcherStatus ()</i> .....	52
<i>bool UpdateFetcherStatus (FetcherStatus fetcher)</i> .....	52
<i>Fetcher GetFetcherByld (string serviceld)</i> .....	52
<i>IEnumerable&lt;Fetcher&gt; GetFetcherList (long since)</i> .....	52
<i>IEnumerable&lt;Fetcher&gt; GetFetcherPage (int start, int count, long since)</i> .....	52
<i>bool RemoveFetcher (string fetcherId)</i> .....	53
<i>string GetFetcherConfig (string fetcherId)</i> .....	53
<i>bool StoreFetcherConfig (string fetcherId, string config)</i> .....	53
2.8: TASKS .....	53
<i>IEnumerable&lt;Task&gt; FetchTask(string fetcherId, int count)</i> .....	53
<i>Task FetchTaskByld(string fetcherId, guid taskId)</i> .....	54
<i>IEnumerable&lt;Task&gt; FetchTask(int count, int complexity)</i> .....	54
<i>IEnumerable&lt;Task&gt; FetchTask(int count)</i> .....	54
<i>Task FetchTaskByld(string fetcherId, guid taskId)</i> .....	54
<i>Task FetchTaskByld(guid taskId)</i> .....	55
<i>IEnumerable&lt;Task&gt; UpdateTaskStatus(IEnumerable&lt;Task&gt; task)</i> .....	55
<i>IEnumerable&lt;Task&gt; UpdateTaskStatus(string fetcherId, IEnumerable&lt;Task&gt; task)</i> .....	55
<i>IEnumerable&lt;Task&gt; GetTaskListByJobld(Guid id)</i> .....	55
<i>int GetTaskCount(TaskStatus status)</i> .....	55
<i>int GetTaskCount ( TaskType type)</i> .....	56
<i>int GetTaskCount(TaskStatus status , TaskType type)</i> .....	56
<i>int GetActiveTaskCount(TaskStatus status)</i> .....	56
<i>int GetActiveTaskCount(TaskStatus status, TaskType type)</i> .....	56
<i>IEnumerable&lt;Task&gt; GetTaskPage(int start, int count, long since)</i> .....	57
<i>Task GetTaskByld(Guid id)</i> .....	57
<i>IEnumerable&lt;Task&gt; GetTaskListByType(TaskType type, long since)</i> .....	57
<i>IEnumerable&lt;Task&gt; GetTaskPageByType(TaskType type, int start, int count, long since)</i> .....	57
<i>IEnumerable&lt;Task&gt; GetActiveTaskByMachine (string tcomputerName)</i> .....	58
<i>IEnumerable&lt;Task&gt; GetTaskPageBySchedule(int start, int count, long since, long startTimeMin, long     startTimeMax, long endTimeMin, long endTimeMax, TaskType type, IEnumerable&lt;TaskStatus&gt; status, ref int     totalTaskCount)</i> .....	58
2.9: JOBS .....	59
<i>Job QueueJob(Job job)</i> .....	59
<i>bool UpdateJobStatus(IEnumerable&lt;Job&gt; job)</i> .....	59
<i>Job GetJob(Guid id)</i> .....	59
<i>bool JobUpdate(Job job)</i> .....	59
<i>DashBoard GetJobDashboard()</i> .....	60
<i>IEnumerable&lt;Job&gt;GetJobPage(int start, int count, long since, JobStatus status)</i> .....	60
<i>IEnumerable&lt;Job&gt; GetJobPageByTaskStatusAndType(int start, int count, long since, JobStatus status,     TaskType tType, TaskStatus tStatus)</i> .....	60
<i>IEnumerable&lt;Job&gt; GetJobPageWithTaskAndStatus(int start, int count, long since, JobStatus status)</i> .....	60
<i>IEnumerable&lt;Job&gt; RemoveJobsWithStatusByTaskStatusAndType (JobStatus status, TaskType tType,     TaskStatus tStatus)</i> .....	61

---

<i>IEnumerable&lt;Job&gt; RemoveJobsWithStatus (JobStatus status)</i> .....	61
<i>IEnumerable&lt;Job&gt; GetJobPage (int start, int count, long since)</i> .....	61
<i>IEnumerable&lt;Job&gt; GetJobPageWithTask (int start, int count, long since)</i> .....	61
<i>int GetJobCount (JobStatus status)</i> .....	61
<i>bool RemoveJob (Guid guid)</i> .....	62
<i>bool RemoveJobs (IEnumerable&lt;Guid&gt; guid)</i> .....	62
<i>bool PurgeAllJobs ()</i> .....	62
<i>IEnumerable&lt;Job&gt; GetJobPageBySchedule(int start, int count, long since, long startTimeMin, long startTimeMax, long endTimeMin, long endTimeMax, IEnumerable&lt;JobStatus&gt; status, TaskType taskType, ref int totalJobCount)</i> .....	62
<i>Job QueueWorkflowJob(Guid watchGuid, string sourceUri, string targetUri)</i> .....	63
<i>Job QueueWorkflowJobEx(Guid workflowGuid, string sourceUri, string targetUri)</i> .....	63
<i>Job QueueWorkflowTemplateJob(string workflowTemplate, string sourceUri, string targetUri)</i> .....	64
<i>Asset GetJobAssets (Guid jobGuid, TaskType taskType, SubTaskType subTaskType)</i> .....	65
2.10: WATCHFOLDER.....	65
<i>bool StoreSnapshot(Guid id, Guid ssid_basedon, Guid ssid_new, string snap)</i> .....	65
<i>SnapshotObject GetSnapshot(Guid watchId)</i> .....	65
<i>String GetWatchConfigById(Guid watchId)</i> .....	65
<i>WatchConfigObject GetWatchFolderById (Guid watchId)</i> .....	66
<i>IEnumerable&lt;WatchConfigObject&gt; GetWatchConfigPage (int start, int count, long since)</i> .....	66
<i>bool StoreWatchConfig(Guid watchId, Guid templateId, string config)</i> .....	66
<i>bool RemoveWatchConfig(Guid watchId)</i> .....	66
2.11: CONFIGURATIONS.....	67
<i>bool StoreConfig(string name, string config)</i> .....	67
<i>bool RemoveConfig(string name)</i> .....	67
<i>IEnumerable&lt;ClientConfig&gt; GetConfigList(long since)</i> .....	67
<i>IEnumerable&lt;ClientConfig&gt; GetConfigPage (int start, int count, long since)</i> .....	67
<i>string GetConfig(string name)</i> .....	67
<i>DateTime GetJobManagerMachineTime()</i> .....	68

# Chapter 1: Rhozet WFS Architecture

## 1.1: Terms and Abbreviations

Term	Description
<b>Rhozet WFS Controller – WFS Controller</b>	The engine component.
<b>Rhozet Workflow Node – WFS Node</b>	The node component(s).
<b>Rhozet Workflow Manager – WFS Manager</b>	The user interface.
<b>Job</b>	A collection of linked dependent tasks.
<b>Task</b>	Single instance atomic workflow activity. Something that can be consumed by a single service.

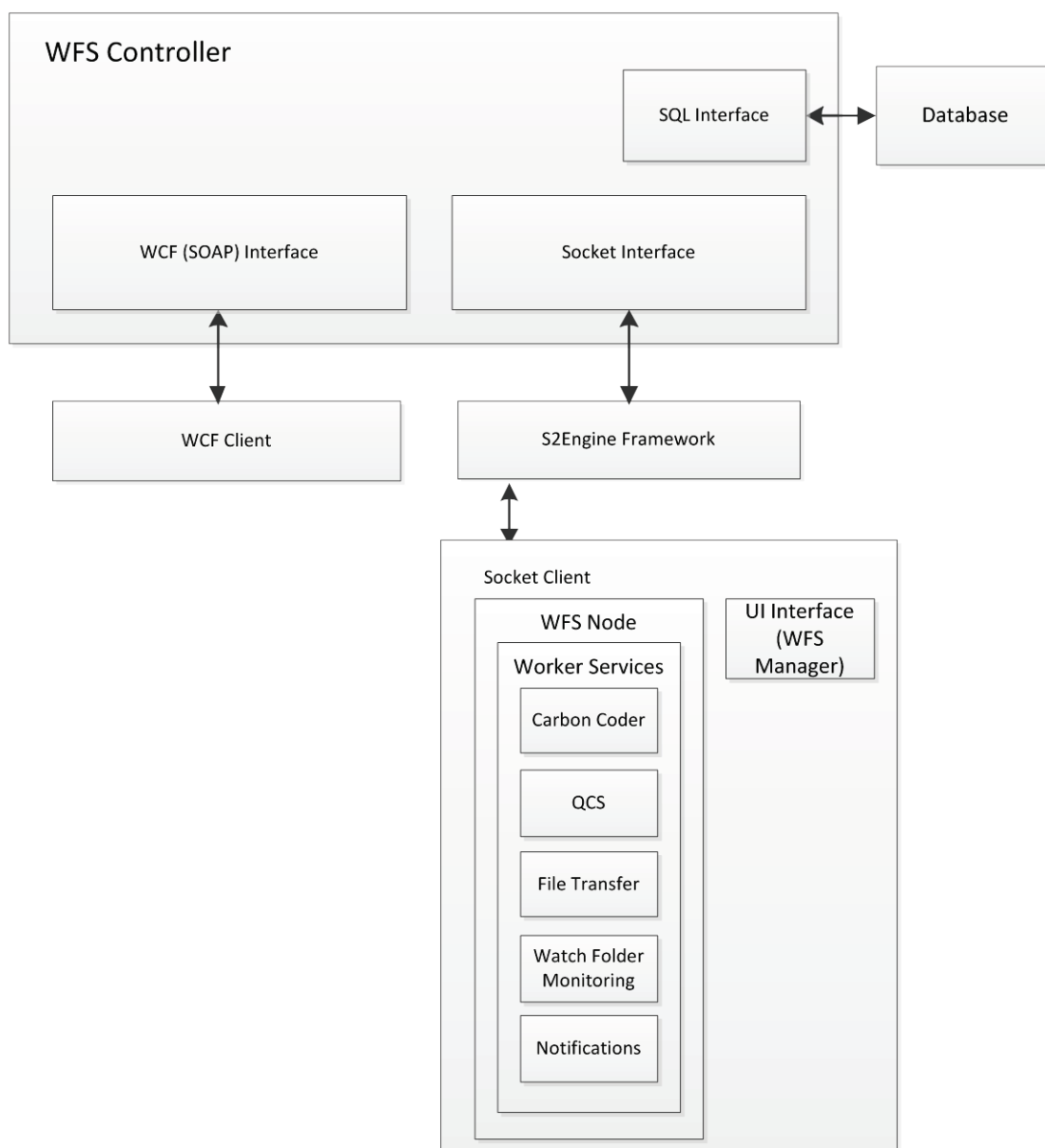
## 1.2: Introduction

The Rhozet Workflow System is client/server architecture for the distributed processing of inter-dependant tasks collected together and processed as a workflow.

Processing and storage for Jobs (a collection of tasks) is through the central and redundant Rhozet WFS Controller. The work is consumed through distributed nodes (as services) called Workflow Node(s). The worker nodes are discussed in more detail later but include, for example Transcoding, File Transfer and Quality Control.

The architecture includes two common sets of services – the Engine(WFS Controller)and the Node. Each Engine acts as a core consumer of Jobs and a core distributor of tasks in the process of workflow control, management and execution. The nodes act in a corresponding manner as consumers of tasks and thus, enable the distributed processing of this work across one or more nodes (as a farm).

The WFS Controller utilizes a central storage repository for both state and the data, which is currently stored in MS SQL Server (which in turn, can provide redundancy). Jobs, Tasks and so on are contained in a set of related tables, which make up the Rhozet Workflow Database.



**Figure 1 – Workflow System Architecture**

As shown in [Figure 1](#), you will see that all access points for the data are through WFS Controller. It functions as the central system for storing work and for responding to requests for work in a simple pull model approach thus distributing the load out to the edge of the infrastructure. To provide a level of redundancy, there can be two or more WFS Controllers.

---

## 1.3: Workflow and State

A WFS Controller controls the state by:

- Processing of State – for example, a job completed.
- Responding to requests for work – handing out tasks.
- Monitoring node states

Typically once a work item (Job) is stored, the tasks are processed and delivered to the nodes in the relevant priority/time order for consumption at the nodes.

A Job is stored through WFS Controller in the Database as a series of interrelated tasks. The nodes are continually calling in (when not busy) asking for work. The WFS Controller decides when a task is free to be distributed, and therefore consumed based on the state of prior related tasks.

A task is ready when its state is set to *Queued*. It will be in this state when one of two conditions arises:

- 1) When it's one of the start tasks – for example, it has no parents and therefore no dependencies.
- 2) When its parent(s) has been processed.

There are many cases where a task is considered processed. For example, a task is considered complete or it is Fatal. State references are explained later in this document.

For a task to be distributed to a node, the following conditions need to be met:

- 1) The task is in the *Queued* state (this only happens when its dependencies are complete or has no dependencies).
- 2) A Node calling for a task item is requesting work of the same “type” as the queued task. For instance a transcode task. (Nodes have roles – the role must match the task type in order for it to be distributed to the node).

Once a task is assigned to a Node then that task is locked until one of two things happen:

- 1) The task is completed and reported back to the WFS Controller.
- 2) The task is not completed and no status is given from the Node for the task. An example would be where the task is taken but the Node is disconnected from the service. For example, the WFS Controller thinks the Node has gone offline.

Although a task is locked in the database once distributed, it's continually being updated for at least one value called the “Last Updated” field. This field enables the WFS Controller to know that the task is still in progress. In addition, the WFS Controller may also be updating other data like “progress” and some of the “properties” of the task that are updates it receives directly from the Nodes.

---

If a task is not updated outside of the default timeout period (30 secs) then the task will be withdrawn from the Node and re-activated as “queued” in the database.

Depending on the Node, a task within an *Active* job typically transitions through several states. The main states are:

- Waiting – initial state
- Queued – has no parents or parents are completed
- Starting – the task is being worked on
- Reject – the QC task is completed with negative results
- Error – the node is unable to complete the work
- Fatal – no Node could complete the work
- Completed – would trigger subsequent dependent children
- Ignored – task is ignored if another workflow path is followed. For example, Transcode Error Email task will be ignored if transcode task is completed successfully.

Jobs also contain states. They are:

- Queued – the initial Job state
- Active – one or more of the jobs tasks are picked up for processing by a node
- Completed – all the required tasks are completed
- Fatal – all the required tasks failed
- Paused – a jobs state can be set to Paused when it is submitted (no fetcher can grab tasks from a Paused job until its state is changed to queued)

The key thing to remember from a clients perspective working through the API on tasks is that once the client has a task, it **must** provide an update to the WFS Controller on the task periodically so that the WFS Controller knows the task is still being processed. A client can do this forever if it so chooses but typically this is while the task progresses to complete. At this stage the client informs the WFS Controller that `Task.Status == Completed`.

This introduction provides an overview of the flow from the queued Job through to the distribution and completion of tasks. As mentioned earlier, tasks may be connected creating an array of relationships commonly termed the “Workflow”.

Later in this document, you will read about the API related functionality provided through a Dot Net Framework, a Dot Net API and a Windows Communication Foundation API as well as touch on an access point for Carbon Coder integration.

## 1.4: Server Side

The WFS Controller runs as a central Windows service or Services and is the consumer of the WFS API messages and serves as the contract for WFS Controller. The client API is the producer of the messages discussed in a later section.

---

The WFS Controller has several key roles that are all consumed through the API. They are:

- 1) Accepts and stores Jobs from the:
  - a. Dot Net WFS Controller framework
  - b. Dot Net 4.0 API
  - c. Older Carbon Server 3.x API clients.
- 2) Manages Presets from central storage.
  - a. Creation
  - b. Editing
- 3) Manages Watch Folder services.
  - a. Configure watch folders (or Workflows)
  - b. Process watch folder triggers
  - c. Provide watch Folder reporting services
- 4) Provides a source for tasks.
  - a. Distribute tasks to Nodes
  - b. Remove tasks from Nodes when not progressing
  - c. Provide task reporting services
- 5) Ensures the health of jobs, tasks and their nodes.
  - a. Change task states as they become completed
  - b. Watch over job progress
  - c. Monitor health of a Node
- 6) Edits system wide configuration data.
  - a. Enable/Disable nodes
  - b. Configure parameters per system or per Node

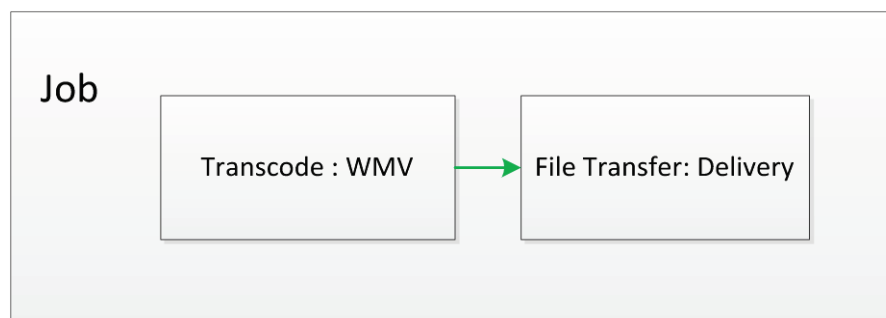
In addition to enabling the framework to store jobs and process tasks, WFS Controller handles the coordination of Watch Folders in the production of automatic workflow creation (Job creation).

WFS Controller also handles the editing, reading and storing of presets all accessed from the central storage. This is done by utilizing a central preset storage system that provides all clients access to a varied list of presets.

There are several other common tasks performed by WFS Controller which will be discussed along with the relevant API calls and how those work.

## Processing Task State

Jobs are stored through WFS Controller into the Databases and then processed for prioritized consumption of the nodes as and when the state of a particular task allows for it to be consumed.



**Figure 2 – Simple Job (Transcode to FTP)**

For example, as seen in [Figure 2](#), Transcode then FTP Transfersubmitted to the WFS manager. The job is stored as two related tasks with a parent child relationship.

The first task received is initially ready to be consumed because it has no parent. The client submitting this job need not be concerned with setting the correct state since the WFS Controller determines that no parent exists for this task and therefore, it may be consumed. The task is set to `Task.Status == Queued`.

The second Transfer task is not because it has a dependency on the completion of the transcode. Its state is therefore held in a wait (`Task.Status == Waiting`) state until the transcode task completes. There are many other possible use cases for the workflows and we discuss these later in the document.

---

**NOTE:** Completed means Completed, Fatal or Error states.

---

The task is consumed by the Node service (Carbon Coder instance for transcoding in this case) and eventually completed. The completed status is reported back to the WFS Controller that is able to update the status of the task in the database. This triggers a state change algorithm to run – noticing a change in state of the edge, it's able to then set the state of the dependant task to *Queued* – thus, freeing this second task up for fetching by another type of Node Service – a Transfer Service in this instance.

A second and slightly more complex sample is shown in [Figure 3](#). In this scenario, the flow for a successful completion is as [Figure 2](#). However, we have added one additional task, Notify. The role of this task is to email a predefined recipient(s). For example, if an error occurs in the transcode, the transcodes output state is Fatal. So, if the first task Completes successfully, the FTP will fire. If it fails, then the Notify will fire. This is one of the many examples of conditional state.

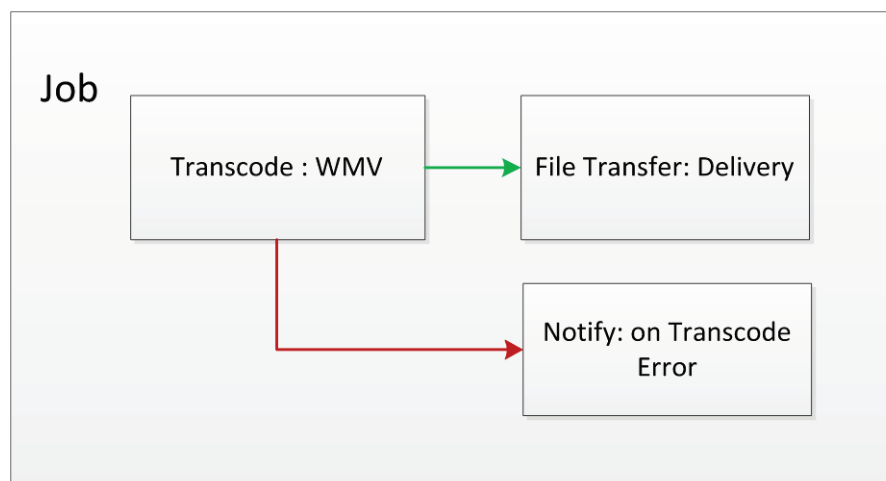


Figure 3 – Simple Job with Notify Task

## Notes on Completed

An Error state means it's worth retrying a task (up to a value configured in the WFS Controller – default=3). A Node reporting Error or a WFS Controller setting this state due to some progress issue allows the tasks to be retried on another Node when available.

Fatal, however, means the edge of the task should update state and trigger any children because this task is unable to complete successfully. A task would most likely go through the Error state three times on three different nodes before being transitioned to Fatal.

We commonly refer to the Node services requesting tasks as "Fetchers" and the process of grabbing is "Fetching" as defined in the documents and the API.

---

**IMPORTANT:** WFS Controller contains no state – most calls through the API are transactional and as such either succeed or fail.

---

## 1.5: Redundancy

### WFS Controller

This service is designed to run with one or more parallel services. The configuration of the WFS Controller enables the workflow controller group to function. When the currently active WFS Controller is unavailable, one of the online WFS Controllers from a group steps in. This redundancy from an API perspective is only available indirectly through the Dot Net based framework classes discussed later. Clients using the API directly would need to use local configuration and retry failed API calls using the WFS Controller Group connection details.

## Nodes

Nodes handle redundancy by having more than one instance of the same “Fetcher” type running on the nodes in the architecture. For instance, to ensure Watch Folders are redundant, we would typically set up two or more Watch Folder services in the network to handle an issue of any one of these services being disconnected (for whatever reason) from the system. This allows the other service to continue to satisfy calls. There is no reason you could not have all services running on all nodes to provide the maximum redundancy possible – although typically this is not necessary and this discussion falls outside the API and to the deployment and best practices guide. In large farms running all node types on all machines is potentially going to saturate the network with status updates – in this instance we would typically increase the timing of status messages to assist with network load.

WFS Nodes communicate to each WFS Controller in WFS Controller Group in a round robin fashion. Also, if the first WFS Controller in a group is too busy to respond or is shut down, WFS Nodes communicate to the second WFS Controller in a group and so on. This architecture is implemented in WFS socket based framework, and hence clients using the API directly would need to use their own load balancing technique.

## 1.6: Client Side

There are several methods of utilizing the WFS Controller from the client perspective:

- 3Dot Net API.
- 3.x API (Job Submission only).
- WCF interface
- Dot Net Framework

These interfaces and methods are discussed in the following sections and include a later reference section for the method details.

### Carbon Server 3. x API

Since the model for WFS is Pull – most 3.x API commands are modeled very differently. However, to provide some continuity from 3.x to this new API there is a job creation API called exposed on port 1301 (by default this may be changed). Any valid JobSubmit XML call to port 1301 will be translated into a Job and Task as defined in the new WFS framework and this call provides a response in terms of the Guid of the new Job which in turn is pulled down to a Fetcher for transcoding and processed.

---

**NOTE:** Whilst transcoding is of course fully supported, utilizing the older API for File Delivery, Notifications and so on is not – having said that, those functions may also simply just work. Also presets and their ID’s are held in a central database so any custom presets should be imported through Workflow Manager into the WFS Controller.

---

## API based client

This is the complete Dot Net API for working with the WFS Controller. Utilizing the Data Structures defined below, client software has every aspect of the architecture exposed to them for working with the services, creating jobs and tasks, working with presets as well as many other functional areas as in those features exposed through the WFS Manager. The WFS Manager is entirely built on this API.

## WFS Framework based client

The Framework for Workflow is also built solely on the API. The intent of the framework is to provide a low barrier of entry for third parties to write services that hook into the Engine providing Node services to the framework allowing the developers to focus on the business logic as opposed to spending too much time learning the extensive API.

Clients writing Node services or Fetchers are required to do several things to support integration into the framework. They are:

- 1) Require a Configuration file to provide some baseline default values (such as the core engines IP address). In order to get the WFS Controller Group auto discovered by WFS Node, In clientconfig.xml , parameter 'UseJMDisco' should be true, and WFS Controller and WFS Node should be up and running. In order to let fetcher decides which WFS Controller to connect to, this parameter should be set to false, and correct 'Primary' cotroller IP and 'Port' controller port parameter values should be specified.
- 2) Derive a class from the frameworks S2Engine class.
- 3) Implement the overrides required from step 2.

Here is a simple sample code from one of the services that shows how this all fits together. In this example, we will initialize a simple fetcher:

```
using System;
using Rhozet.ApiLib;
using Rhozet.JMLibrary;

//Worker Fetcher
namespace Rhozet.WorkerFetcher
{
    public class WorkerEngine : S2Engine
    {
        override protected void Initialize(JobManagerGroup group)
        {
            try
            {
                //Initilize the new fetcher with default worker config
                //the JM IP, port, fetcher type, and task types that
                //engine will to handle (values are from ClientConfig.Xml)
                string config = base.Initialize(new WorkerConfig(), group,
FetcherType.WorkerFetcher,
                new[] { TaskType.Pause }, 0);
                // Deserializing the config string into object
                // WorkerConfig derived from JmConfig
                _config = JmCommon.DeserializeToObject
                    (typeof(WorkerConfig), config) as WorkerConfig;
                //Setting the initial config to the fetcher
                if (null != _config)
                    JmPollFrequency = _config.JMPollFrequency.Value;
            }
            catch (Exception ex)
            {
                Log.Error("Trying to create Fetcher", ex);
            }
        }
    }
}
```

Initially we call the base. Initialize method indentifying the type of fetcher we want to create along with the type of Tasks we can process (Pause in this case).

A successful response provides us a configuration string (which in turn is deserialized into an object derived from a JmConfig class). JmConfig provides a base class for some common parameters and can be subclassed to provide additional Node configuration that is stored in the database.

When a new service is registered – the configuration provided in the call to initialize is setup as a default set for any subsequent fetchers of this type. Once a Fetcher (Node) is registered, the default configuration is copied to this unique instance and can be then later modified to suit this Fetcher.

There are then some primary methods that require Overrides in your client Fetcher class:

1) FetchTask (...)

- a. The response is normally one task that is ready to be processed by this Fetcher.

```
/// <summary>
/// Update the WE with your task status
/// This happens regardless of progress on the local task
/// </summary>
override protected void FetchTask()
{
    int count = MaxTasks - _list.Count;
    if (count > 0)
    {
        // fetch x number of tasks
        IEnumerable<Task> tasks = IJobManager.FetchTask(count);
        if (tasks != null)
        {
            using (lock(_list))
            {
                foreach (var t in tasks)
                {
                    _list.Add(t);
                }
            }
        }
    }
}
```

2) SendTaskStatus (...)

- a. Note – the response from this call could include tasks that have been re-allocated to another Fetcher (based on no progress coming from this Fetcher). Clients, therefore, should remove any processing of this task from the local system since the task is essentially re-allocated to an alternate fetcher or back in the *Queued* state.

```

protected override void SendTaskStatus()
{
    try
    {
        var tasks = GetLocalWorkerList();
        IEnumerable<Task> cancelledList = new List<Task>();
        if (tasks.Length > 0)
        {
            cancelledList = IJobManager.UpdateTaskStatus(tasks);
        }

        foreach(var t in cancelledList)
        {
            Lock (_list)
            {
                Manager._list.Remove(t.Guid
            }
        }
    }
    catch (Exception ex)
    {
        Log.Error("Failed updating task status.", ex);
    }
}

```

The WFS Controller requires a Fetcher client to provide two primary status reports (based on the PollFrequency in the Configuration). Failure to provide either of these can cause two possible issues:

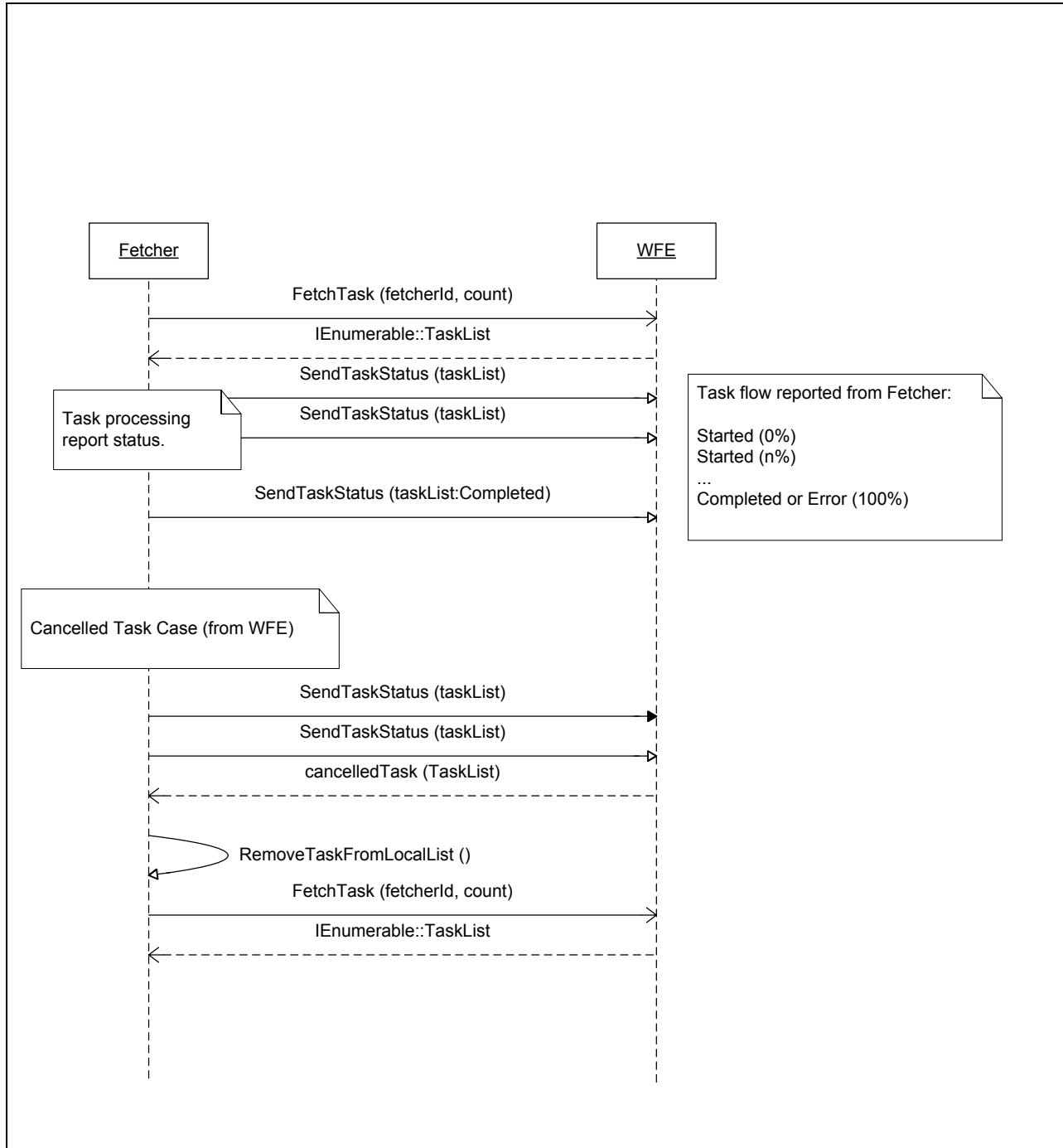
- 1) If the task status is not updated (regardless of how progress changes), then a timeout value on the WFS Controller may cause the task to be removed from this fetcher and allocated to another node.
- 2) The Fetcher Status (automatically handled by this framework) provides status updates on the health of this service. There is nothing for the client to do here since it is handled automatically.

A Fetcher has three choices to hand task state back to the WFS Controller:

- 1) Completed - everything worked ok and the task is complete.
- 2) Error - there was some problem in handling the task, but its non fatal. This triggers the task to be re-queued and handed off to another fetcher. Error count incrementing beyond 3 causes the task to change state to Fatal which means the task will now not be retried.
- 3) Ignored - reserved for when a fetcher Fetches the task, and then needs to hand this back. This is similar to error in that it's re-queued but its error count is not incremented.

The following calling sequence shows how to process task updates once they are "fetched". The WFS Controller only really cares that you send this message. The status and progress need not change but the status (event the exact same status) must be reported no later than 15 seconds apart. Otherwise, the task is taken from the fetcher and handed to another fetcher. Note how the second sequence here

shows how this works. If you receive cancelled tasks from the status update call, then you need to stop working on them because they no longer belong to this fetcher.



Lastly, to fully utilize the framework, the fetcher needs to update the configuration if it is being changed by the user. This is your chance to dynamically update any values used or defined in the framework configuration and shown to users from the WFS Manager in the configuration UI.

```
protected override void ProcessConfigurationChange(string config)
{
    var _config = JmCommon.DeserializeToObject
        (typeof(WorkerConfig), config) as WorkerConfig;
    //call parent framework with any relevant
    // change to the _config part
    base.ProcessConfigurationUpdate(_config);
    if (null != _config)
    {
        JmPollFrequency = _config.JMPollFrequency.Value;
    }
}
```

---

## 1.7: WCF Interfaces

The full API is now exposed on the central WFS Controller by default on ports 8731 and 8741 (Http and Tcp respectively). These values are configured in the application configuration for JMServices as follows:

```
<endpoint address="http://localhost:8731/Rhozet.JobManager.JMServices/"
binding="wsHttpBinding" contract="Rhozet.JobManager.IWfcJmServices"
name="JmHttpEndpoint">

<endpoint address="net.tcp://localhost:8741/Rhozet.JobManager.JMServices/"
binding="netTcpBinding" contract="Rhozet.JobManager.IWfcJmServices"
name="JmTcpEndpoint" />
```

Querying the service endpoint will provide the WSDL describing the contract provided by the WFS Controller. Aside from the calling protocol being Http/Tcp the interface is a mirror reference to the Dot Net API Reference described in this document so the WSDL is not displayed for brevity.

## 1.8: Data structures

After visiting the Framework for the WFS Controller, we need to review some of the more common data structures defined in the libraries and provided as part of the SDK.

This section defines the data structures defined both in the shared libraries that form part of the SDK as well as in the API.XSD definition file provided as part of the API. Clients requiring access to the data structures should add a reference to the APILib.dll provide during the installation in the ..\lib directory. The Actual API (methods and classes) are defined in the Rhozet.JMLibrary.dll.

Rhozet.ApiLib.dll is the library that holds the common classes to create job, task and fetcher. These classes are used by the actual API to create either a job submission engine (e.g.: Watch Folder Service) or fetcher engine (for example, Carbon Coder Service, Transfer Service, etc).

## Rhozet.ApiLib.Fetcher

The Fetcher structure describes the structure for a Node or Service, its properties and methods.

Name	Type	Description
<b>FetcherID</b>	System.String	Unique Name for identification of the fetcher
<b>FetcherType</b>	Rhozet.ApiLib.FetcherType	Enumerates the type of the fetcher during the initialization of the fetcher.  See FetcherType Enum for description.
<b>Version</b>	System.String	Fetcher version number.
<b>ActiveTaskCount</b>	System.Int32	Number of task that is currently active in the fetcher.
<b>Software</b>	Rhozet.ApiLib.Software	Software description of the Machine that is running the fetcher.
<b>Hardware</b>	Rhozet.ApiLib.Hardware	Hardware description of the Machine that is running the fetcher.
<b>Status</b>	Rhozet.ApiLib.MachineStatus	The status of the machine that is running the fetcher
<b>Enabled</b>	System.Boolean	Enables the fetcher
<b>WorkerRole</b>	List<TaskType>	Fetcher Role
<b>Configuration</b>	System.String	Config data of a Fetcher
<b>Property</b>	Rhozet.ApliLib.DataObject[]	Various property values of a Fetcher, list of unique key-value pairs.

### Rhozet.ApiLib.Software

Name	Type	Description
<b>Windows</b>	System.String	Windows OS description
<b>WindowsVersion</b>	System.String	Windows OS Version
<b>ComputerName</b>	System.String	Machine Unique ID
<b>DotNet</b>	System.String	Client installed DotNet Version
<b>CarbonCoder</b>	System.String	Carbon Coder Version (if installed)
<b>LicenseExpiry</b>	System.String	License expiry date
<b>TimeZone</b>	System.DateTime	Time zone information

### Rhozet.ApiLib.Hardware

Name	Type	Description
<b>Manufacturer</b>	System.String	Manufacturer of the computer
<b>Model</b>	System.String	Model of the computer

### Rhozet.ApiLib.MachineStatus

Name	Type	Description
<b>PhysicalMemory</b>	System.Int64	
<b>FreeMemory</b>	System.Int64	
<b>CpuUsage</b>	System.Int32[]	
<b>DiskSize</b>	System.String[]	
<b>FreeSpace</b>	System.String[]	
<b>Online</b>	System.Boolean	
<b>ServiceMemoryUsage</b>	System.Int64	
<b>ServiceCpuUsage</b>	System.Int32	
<b>ServiceThreads</b>	System.Int32	
<b>BackoffPeriod</b>	System.Int32	

## Rhozet.ApiLib.Job

Job class is an important part of the framework and provides an object that contains both tasks and their relationships.

Name	Type	Description
<b>Guid</b>	System.Guid	Unique identifier of Job
<b>Type</b>	System.ApiLib.JobType	Job Type
<b>FetcherID</b>	System.String	Fetcher that is currently (or previously) executing the Job
<b>Name</b>	System.String	Job Name
<b>Meta</b>	System.String	
<b>Status</b>	System.ApiLib.JobStatus	Job Status
<b>Priority</b>	System.Int32	Job Priority (1= low, 10= high)
<b>LastUpdate</b>	System.Int64	Time (in ticks) where job is last updated
<b>Created</b>	System.Int64	Time (in ticks) where job is created
<b>Started</b>	System.Int64	Time (in ticks) where job is stated
<b>Task</b>	List<Task>	List of tasks
<b>FileIndex</b>	System.String	Unique job index data
<b>StartTime</b>	System.Int64	Time (in ticks) when job should be started. Currently only being used by Carbon Capture Fetcher.
<b>EndTime</b>	System.Int64	Time (in ticks) when job should be completed. Currently only being used by Carbon Capture Fetcher.

## Rhozet.ApiLib.Task

An atomic unit of work contained within a Job that may have zero or more parent and zero or more child tasks. It also has a type that is used to distribute the task to a specific fetcher.

Name	Type	Description
<b>Guid</b>	System.Guid	Unique identifier of Task.
<b>Name</b>	System.String	Task Name.
<b>JobID</b>	System.Guid	Job Guid that holds the task.
<b>TaskType</b>	Rhozet.ApiLib.TaskType	Type of task.
<b>TaskStatus</b>	Rhozet.ApiLib.TaskStatus	Status of task.
<b>Progress</b>	System.Int32	Progress of task (0-100).
<b>Property</b>	Rhozet.ApiLib.DataObject[]	Task Properties, list of unique key-value pairs.
<b>ServiceID</b>	System.String	Fetcher that is currently(or previously) executing this Task .
<b>Image</b>	System.Byte[]	Current Image of the task.
<b>Parents</b>	Rhozet.ApiLib.ParentList	List of parents Guid and dependency of to the parents.
<b>Children</b>	Rhozet.ApiLib.Child[]	List of children and required status to move forward.
<b>Description</b>	System.String	Description of the task.
<b>LastUpdate</b>	System.Int64	Time (in ticks) where job is last updated.
<b>CheckInTime</b>	System.String	
<b>Asset</b>	Rhozet.ApiLib.Asset	Source and Targets file URIs.

Name	Type	Description
<b>Priority</b>	System.Int	Priority of a task. Currently used by Carbon Capture fetcher only.
<b>LastProgress</b>	System.Int	
<b>StartTime</b>	System.Int64	Time (in ticks) when job should be started. Currently only being used by Carbon Capture Fetcher.
<b>EndTime</b>	System.Int64	Time (in ticks) when job should be finished. Currently only being used by Carbon Capture Fetcher.

### Rhozet.ApiLib.ParentList

Name	Type	Description
<b>DependAllParents</b>	System.Boolean	If true, the task needs to have all parents meet the required status to proceed. Otherwise, it only needs one parent that meets the status to proceed.
<b>Parent</b>	Rhozet.ApiLib.Parent[]	List of parent Guid and required task status.

### Rhozet.ApiLib.Parent

Name	Type	Description
<b>ParentId</b>	System.Guid	Task Guid of the Parent
<b>RequiredStatus</b>	Rhozet.ApiLib.TaskStatus	The required task status for the parent to move to current task.  See TaskStatus Enum.

### Rhozet.ApiLib.Child

Name	Type	Description
<b>ChildId</b>	System.Guid	Task Guid of the Child
<b>RequiredStatus</b>	Rhozet.ApiLib.TaskStatus	The required task status for the current task to move to child.  See TaskStatus Enum.

### Rhozet.ApiLib.DataObject

Name	Type	Description
<b>Name</b>	System.String	Name of the property.
<b>Value</b>	System.String	Value of the property.

### Rhozet.ApiLib.Asset

Name	Type	Description
<b>Sources</b>	System.String[]	List of source URIs.
<b>Targets</b>	System.String[]	List of target URIs.

## Rhozet.ApiLib.Preset

Preset is a group of encoding parameters; they can be codec (transcode) or filter or Rhozet Quality Control System (QCS) checks.

In Rhozet WFS, transcode and filter preset saves the settings for a single codec or a filter. QC preset saves the settings for multiple quality checks.

Name	Type	Description
<b>Guid</b>	System.Guid	Preset Guid
<b>Name</b>	System.String	Preset Name
<b>Type</b>	Rhozet.ApiLib.PresetType	
<b>Description</b>	System.String	Preset description
<b>Category</b>	System.String	Preset Category
<b>ModuleList</b>	System.Guid[]	Array of Guids of all modules that the preset consists

## Rhozet.ApiLib.Module

Module contains a Scheme and a parameter set which has all the codec parameter settings. Transcode preset contains one Module, which is of Type 'Transcode', and SubType 'Common'.

Filter preset contains one Module, which is of Type 'Filter' and SubType can be 'Audio' or 'Video'.

QCS preset contains one Module of Type "Common", zero or more Modules of Type 'Video', 'Audio' or 'Container'. Rhozet Quality Control Module can be of Type 'QCComparative', which can be used only during post transcode QC tasks. If it is a type of 'QCNonComparative', which can be used during pre and post transcode QC tasks. User can create his/her own QCS Preset depending on his choice of quality checks. Refer to Rhozet QCS document for further explanation.

Name	Type	Description
<b>Guid</b>	System.Guid	Module Guid
<b>ModuleGuid</b>	System.Guid	ModuleGUID (Base preset/codec guid)
<b>Name</b>	System.String	Module Name

Name	Type	Description
<b>Description</b>	System.String	Module Description
<b>System</b>	System.Boolean	If System preset = 1, if user preset = 0
<b>Legacy</b>	System.Boolean	If Carbon Coder legacy preset = 1, if RPI based preset = 0 (All QC Presets are RPI based)
<b>SubType</b>	Rhozet.ApiLib.ModuleSubType	Common = 0, Video = 1, Audio = 2, Container = 3
<b>Type</b>	Rhozet.ApiLib.ModuleType	QCComparative = 0, QCNonComparative = 1, Transcode = 2, Filter = 3
<b>Parameter</b>	System.String	If Legacy preset, config data are stored. If RPI based preset, parameter xml is stored.
<b>Scheme</b>	System.String	See ModuleScheme

### Rhozet.ApiLib.ModuleScheme

RPI Scheme string value is stored in this data structure. Each Preset is associated with one or more Modules, where each module has its own scheme or can depend on the base preset's scheme.

Name	Type	Description
<b>Guid</b>	System.Guid	ModuleGUID or Guid of a Module
<b>Scheme</b>	System.String	If Legacy Preset, Config items are stored. If RPI based preset, RPI scheme xml is stored.

## Enumerations

### *Rhozet.ApiLib.FetcherType*

Enum
Undefined
WatchFolder
JobBuilder
JobEngine
JobManager
QCS
Review
ProfileManager
CarbonCoder
WorkerFetcher
Preset
CarbonCapture
Transfer
Connector
Monitor
Notifier

*Rhozet.ApiLib.JobType*

Enum
<b>Undefined</b>
<b>WatchFolder</b>
<b>Varied</b>

*Rhozet.ApiLib.JobStatus*

Enum
<b>Queued</b>
<b>Paused</b>
<b>Active</b>
<b>Fatal</b>
<b>Completed</b>
<b>Abort</b>

*Rhozet.ApiLib.TaskType*

<b>Enum</b>
<b>Undefined</b>
<b>Transcode</b>
<b>NotifySmtplib</b>
<b>NotifyUrl</b>
<b>NotifyProcess</b>
<b>Multiplex</b>
<b>Transfer</b>
<b>WatchTrigger</b>
<b>Pause</b>
<b>QC</b>
<b>Review</b>
<b>StartTask</b>
<b>NoopTask</b>
<b>EndTask</b>
<b>MediaEvaluate</b>
<b>Rehearsal</b>
<b>LiveCapture</b>
<b>ControlledCapture</b>

*Rhozet.ApiLib.TaskStatus*

Enum
Waiting
Queued
Error
Completed
Stopped
Starting
Started
Preparing
Fatal
Stopping
Pausing
Paused
Taken
Resuming
Hold
Reject
Ignored

*Rhozet.ApiLib.PresetType*

Enum
QC
Transcode
Filter

*Rhozet.ApiLib.ModuleSubType*

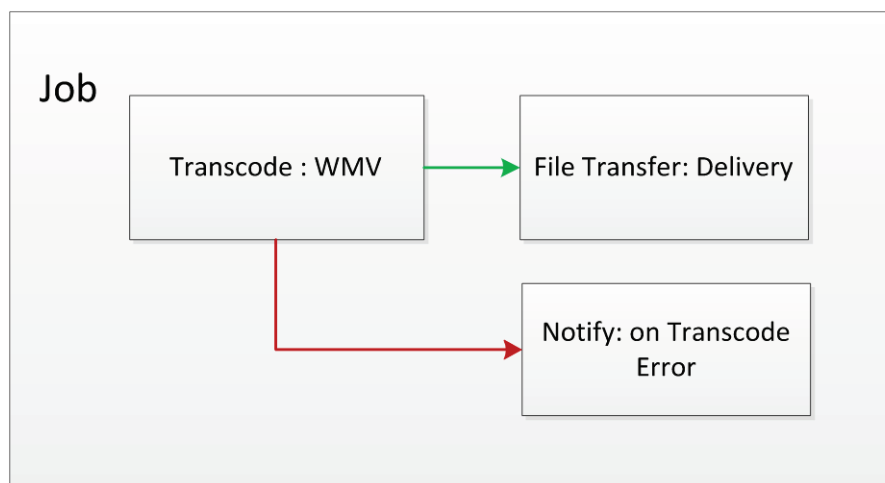
Enum
Common
Video
Audio
Container

*Rhozet.ApiLib.ModuleType*

Enum
QCComparative
QCNonComparative
Transcode
Filter

## 1.9: Constructing work - Jobs and Tasks

As previously mentioned Jobs are an atomic action of work. Jobs contain one or more tasks and task are linked together. Tasks can be linked together based on the outcome of previous tasks. So, for instance you can link a task to two tasks as shown in [Figure 4](#).



**Figure 4 – Jobs and Tasks**

The Job contains three tasks. Then child relationships can be set up based on the outcome of the Transcode task. For example, in our case above, Completed would in turn trigger FT delivery OR Failure would in turn trigger Notify by email.

Below is some sample code that explains the work required to set this up through the API:

```
var job = new Job { Name = "Transcode WMV", Task = new Task[3] };

job.Task[0] = new Task
{
    Name = "Transcode 123",
    Description = "MPEG 2 WMV",
    TaskType = TaskType.Transcode,
    Guid = Guid.NewGuid()
};
job.Task[0].AddTarget(@"\\Storage\MPEG2\Output");
job.Task[0].AddSource(@"\\Storage\MPEG2\File1.mpg");
job.Task[0].AddProperty("My private data property", "Some value");

//When transcode finishes, asset source and targets are populated on the
delivery fetcher
job.Task[1] = new Task();
job.Task[1] = new Task
{
    Name = "Delivery 123",
    Description = "MPEG 2 WMV",
    TaskType = TaskType.Delivery
};

job.Task[2] = new Task();
job.Task[2] = new Task
{
    Name = "Transcode 123",
    Description = "MPEG 2 WMV",
    TaskType = TaskType.Transcode,
    NotifyEmail = new NotifyEmailTask
    {
        From = "Admin@Rhozet.com",
        To = "Workflow@rhozet.com",
        Subject = "Job Error %SOURCE%"
    }
};
job.Task[0].AddChild(job.Task[1], TaskStatus.Completed);
job.Task[0].AddChild(job.Task[2], TaskStatus.Fatal);
```

This job can now use the `IJobLib` reference to `Submit(job)` so it's dropped into the processing queue on the database. The relevant exceptions are thrown if there is any issues submitting jobs and so the caller should handle those in case of error on submission.

All tasks in the job without a parent dependent will be queued immediately and subsequently distributed to a service that has the ability to process the "type" of task defined. So in the above case the transcode job would immediately be available to any fetcher that can "Transcode" tasks.

Once the task completes (either from status as Completed or Fatal) that task is closed and the WFS Controller processes the state. Depending on the state of the Transcode either the Transfer will occur OR the Email will be sent.

Note a fetcher never sets the state to Fatal. This is done on the processing side. If a fetcher that wishes to record a task has an error, then they set the state on the task and report this back as an Error. The WFS Controller will allow tasks to be processed and retried 3 times if a task generates an error like this. After three retries the task will become Fatal and as in this example follows a different path.

# Chapter 2: API Reference

## 2.1: State

### Version GetVersion()

This method returns the version number of WFS Controller.

Return Type	Arguments	Exceptions
Version	null	System.ArgumentNullException System.InvalidOperationException

### int ConnectionHealth()

This method returns an integer in the range of 0 to 100 showing how healthy the connection is.

Return Type	Arguments	Exceptions
int	null	

## 2.2: Logs

Logging within the framework is enabled through Log4Net. Each service contains a configuration file that can enable logging to a console, Event Log and in our case also to the WFS Controller through RhozetLogging services.

Utilizing logging from the code can be done through a simple `Log.Info("My info message")` in which case there is no need to utilize the methods below.

### Guid StoreLog(ErrorLog log)

This method stores an error log in the Database.

Return Type	Arguments	Exceptions
Guid	ErrorLog	System.ArgumentNullException System.InvalidOperationException System.NullReferenceException

### bool StoreLogs(IEnumerable<ErrorLog> log)

This method stores a list of error logs in the Database.

Return Type	Arguments	Exceptions
Guid	IEnumerable<ErrorLog>	System.ArgumentNullException System.InvalidOperationException System.NullReferenceException

### ErrorLog GetErrorLog(Guid id)

This method returns an error log based on the log ID.

Return Type	Arguments	Exceptions
ErrorLog	Guid	System.ArgumentNullException System.InvalidOperationException

### IEnumerable<ErrorLog> GetErrorLogbyReferenceId(Guid id)

This method returns a list of error logs based on the task ID.

Return Type	Arguments	Exceptions
IEnumerable<ErrorLog>	Guid	System.ArgumentNullException System.InvalidOperationException

### IEnumerable<ErrorLog> GetErrorLogPage (int start, int count, long since)

This method returns a list of error logs based on starting index and count of records. User can track the start index in this case.

Return Type	Arguments	Exceptions
IEnumerable<ErrorLog>	Guid	System.ArgumentNullException System.InvalidOperationException

### IEnumerable<ErrorLog> GetErrorLogList (long since)

This method returns a list of error logs based on a DateTime(ticks) value.

Return Type	Arguments	Exceptions
IEnumerable<ErrorLog>	long	System.ArgumentNullException System.InvalidOperationException

## 2.3: Task Property

Task Property contains the task results (for example, Media Evaluate Task) and important data of the task which is completed.

The below sample shows how to obtain the source media evaluate results:

```
private static void GetMediaEvaluateTaskResults()
{
var client = new JMServiceReference.WfcJmServicesClient("JmHttpEndpoint");

/// This API function returns the first 10 MediaEvaluate Tasks.

var mediaEvalTasks =
client.GetTaskPageByType(JMServiceReference.TaskType.MediaEvaluate, 0, 10,
0).ToList();

foreach(var task in mediaEvalTasks)
{
Console.WriteLine("Task Guid" + task.guidField);

foreach (var property in task.propertyField)
{
Console.WriteLine(property.nameField + ": " + property.valueField);
}
}
}
```

The below functions are useful for Pre-Transcode and Post Transcode QC task results.

### Guid StoreTaskPropertyEx (TaskPropertyEx property)

This method stores the TaskProperty in the Database.

Return Type	Arguments	Exceptions
Guid	TaskPropertyEx	System.ArgumentNullException System.InvalidOperationException System.NullReferenceException

### bool StoreTaskPropertiesEx(IEnumerable<TaskPropertyEx> properties)

This method stores a list of TaskProperties in the Database.

Return Type	Arguments	Exceptions
bool	IEnumerable <TaskPropertyEx>	System.ArgumentNullException System.InvalidOperationException

### TaskPropertyEx GetTaskPropertyExById(Guid id)

This method returns TaskProperty based on ID.

Return Type	Arguments	Exceptions
TaskPropertyEx	Guid	System.ArgumentNullException System.InvalidOperationException

### IEnumerable<TaskPropertyEx> GetTaskPropertyExByReferenceId (Guid id)

This method returns TaskProperty based on the Task ID.

Return Type	Arguments	Exceptions
IEnumerable <TaskPropertyEx>	Guid	System.ArgumentNullException System.InvalidOperationException

### IEnumerable< TaskPropertyEx> GetTaskPropertyExByRefNamSeq (Guid id, string name, int sequence)

This method returns a TaskProperty based on reference ID, name and sequence.

Return Type	Arguments	Exceptions
IEnumerable <TaskPropertyEx>	Guid id, string name, int sequence	System.ArgumentNullException System.InvalidOperationException

### **IEnumerable<TaskPropertyEx> GetTaskPropertyExPage(int start, int count, long since)**

This method returns a TaskProperty list based on starting index, count of records and DateTime. User should track index in this case. DateTime value “since” should represent a time span back in time. Default is zero for configured time span.

Return Type	Arguments	Exceptions
<code>IEnumerable&lt;TaskPropertyEx&gt;</code>	<code>int start, int count, long since</code>	<code>System.ArgumentNullException</code> <code>System.InvalidOperationException</code>

### **IEnumerable< TaskPropertyEx > GetTaskPropertyExList (long since)**

This method returns a TaskProperty list based on a DateTime value “since” which should represent a time span back in time. Default is zero for a configured time span.

Return Type	Arguments	Exceptions
<code>IEnumerable&lt;TaskPropertyEx&gt;</code>	<code>long since</code>	<code>System.ArgumentNullException</code> <code>System.InvalidOperationException</code>

## 2.4: Presets

This section covers the API calls related to storing, retrieving, updating and deleting of Presets.

### **bool RemovePreset(Guid id)**

This method removes a preset from Database based on preset ID.

Return Type	Arguments	Exceptions
<code>bool</code>	<code>Guid id</code>	<code>System.ArgumentNullException</code> <code>System.InvalidOperationException</code>

### **Guid StorePreset(Preset Preset)**

This method stores a Preset in the database, returning the GUID for the newly stored preset.

Return Type	Arguments	Exceptions
<code>Guid</code>	<code>Preset Preset</code>	<code>System.ArgumentNullException</code> <code>System.InvalidOperationException</code>

### Preset GetPreset(Guid guid)

This method returns a Preset based on the guid provided.

Return Type	Arguments	Exceptions
Preset	Guid guid	System.ArgumentNullException System.InvalidOperationException

### IEnumerable<Preset> GetPresetList(long since)

This method returns a list of presets based on the DateTime value.

Return Type	Arguments	Exceptions
IEnumerable<Preset>	long since	System.ArgumentNullException System.InvalidOperationException

### IEnumerable<Preset> GetPresetListByType(PresetType type, long since)

This method returns a list of presets based on PresetType and DateTime value.

Return Type	Arguments	Exceptions
IEnumerable<Preset>	PresetType type, long since	System.ArgumentNullException System.InvalidOperationException

### IEnumerable<Preset> GetPresetListByCategory(string category, PresetType type, long since)

This method returns a list of presets based on Category, PresetType and DateTime value.

Return Type	Arguments	Exceptions
IEnumerable<Preset>	string category, PresetType type, long since	System.ArgumentNullException System.InvalidOperationException

### IEnumerable<Preset> GetPresetPage(int start, int count, long since)

This method returns a list of presets based on starting index, number of records and DateTime value.

Return Type	Arguments	Exceptions
IEnumerable<Preset>	int start, int count, long since	System.ArgumentNullException System.InvalidOperationException

### **IEnumerable<Preset> GetPresetPageByType(PresetType type, int start, int count, long since)**

This method returns a list of presets based on PresetType, starting index, number of records and DateTime value.

Return Type	Arguments	Exceptions
<code>IEnumerable&lt;Preset&gt;</code>	<code>PresetType type, int start, int count, long since</code>	<code>System.ArgumentNullException</code> <code>System.InvalidOperationException</code>

### **IEnumerable<Preset> GetPresetPageByCategory(string category, int start, int count, long since)**

This method returns a list of presets based on Preset Category, starting index, number of records and DateTime value.

Return Type	Arguments	Exceptions
<code>IEnumerable&lt;Preset&gt;</code>	<code>string category, PresetType type, int start, int count, long since</code>	<code>System.ArgumentNullException</code> <code>System.InvalidOperationException</code>

### **IEnumerable<Preset> GetDistinctPresetTypes()**

This method returns a distinct list of preset types available in JM Library.

Return Type	Arguments	Exceptions
<code>IEnumerable&lt;PresetType&gt;</code>	null	<code>System.ArgumentNullException</code> <code>System.InvalidOperationException</code>

## 2.5: Modules

This section covers the API calls related to storing, retrieving, updating and deleting of Modules.

### **bool RemoveModule(Guid id)**

This method deletes a module from Database based on module guid.

Return Type	Arguments	Exceptions
<code>bool</code>	<code>Guid id</code>	<code>System.ArgumentNullException</code> <code>System.InvalidOperationException</code>

### Guid StoreModule(Module Module)

This method stores a Module in the Database. Returned is the guid of the new module stored.

Return Type	Arguments	Exceptions
Guid	Module Module	System.ArgumentNullException System.InvalidOperationException

### Module GetModule(Guid id)

This method retrieves a module from the Database based on the guid provided.

Return Type	Arguments	Exceptions
Module	Guid guid	System.ArgumentNullException System.InvalidOperationException

### IEnumerable<Module> GetModuleList (long since)

This method returns a module list based on the DateTime value.

Return Type	Arguments	Exceptions
IEnumerable<Module>	long since	System.ArgumentNullException System.InvalidOperationException

### IEnumerable<Module> GetModuleListBySubType (ModuleSubType subtype, long since)

This method returns a module list based on Module SubType and DateTime value.

Return Type	Arguments	Exceptions
IEnumerable<Module>	ModuleSubType subtype, long since	System.ArgumentNullException System.InvalidOperationException

### IEnumerable<Module> GetModulePage (int start, int count, long since)

This method returns a list of modules based on starting index, number of records and DateTime value.

Return Type	Arguments	Exceptions
IEnumerable<Module>	int start, int count, long since	System.ArgumentNullException System.InvalidOperationException

### **IEnumerable<Module> GetModulePageBySubType (ModuleSubType subtype, int start, int count, long since)**

This method returns a list of modules based on Module SubType, starting index, number of records and DateTime value.

Return Type	Arguments	Exceptions
<code>IEnumerable&lt;Module&gt;</code>	<code>ModuleSubType</code> subtype, <code>int</code> start, <code>int</code> count, <code>long</code> since	<code>System.ArgumentNullException</code> <code>System.InvalidOperationException</code>

### **IEnumerable<ModuleSubType> GetDistinctModuleSubTypes (ModuleSubType subtype)**

This method returns a distinct list of Module subtypes for a particular preset type.

Return Type	Arguments	Exceptions
<code>IEnumerable</code> <code>&lt;ModuleSubType&gt;</code>	<code>ModuleSubType</code> subtype	<code>System.ArgumentNullException</code> <code>System.InvalidOperationException</code>

## 2.6: ModuleScheme

This section covers the API calls related to storing, retrieving, updating and deleting of ModuleScheme.

### **bool RemoveModuleScheme (Guid id)**

This method removes a module scheme from the database based on the guid provided.

Return Type	Arguments	Exceptions
<code>bool</code>	<code>Guid</code> id	<code>System.ArgumentNullException</code> <code>System.InvalidOperationException</code>

### **Guid StoreModuleScheme (ModuleScheme Module)**

This method stores a Module Scheme to the database and returns the guid of the stored Module Scheme.

Return Type	Arguments	Exceptions
<code>Guid</code>	<code>ModuleScheme</code> Module	<code>System.ArgumentNullException</code> <code>System.InvalidOperationException</code>

### ModuleScheme GetModuleScheme (Guid guid)

This method gets a Module Scheme based on the guid provided.

Return Type	Arguments	Exceptions
ModuleScheme	Guid guid	System.ArgumentNullException System.InvalidOperationException

### IEnumerable<ModuleScheme> GetModuleSchemeList ()

This method gets a list of all Module Schemes stored in the database.

Return Type	Arguments	Exceptions
IEnumerable <ModuleScheme>	Guid guid	System.ArgumentNullException System.InvalidOperationException

### IEnumerable<ModuleScheme> GetModuleSchemePage (int start, int count)

This method returns a list of Module Schemes based on the starting index and number of records

Return Type	Arguments	Exceptions
IEnumerable <ModuleScheme>	int start, int count	System.ArgumentNullException System.InvalidOperationException

## 2.7: Fetchers

This section covers the API calls related to registering, updating status and retrieving of fetchers.

### bool EnableFetcher (string fetcherId, bool enabled)

This method can be used to enable or disable the fetcher from JM's perspective. If enabled is set to true, the fetcher gets enabled and can do FetchTask operations. If set to false, all FetchTask Operations are ignored by the fetcher.

Return Type	Arguments	Exceptions
bool	string fetcherId, bool enabled	System.ArgumentNullException System.InvalidOperationException

### bool EnableFetcher (bool enabled)

This method can be used to enable or disable the fetcher from JM's perspective. If enabled is set to true, the fetcher gets enabled and can do FetchTask operations. If set to false, all FetchTask Operations are ignored by the fetcher.

Return Type	Arguments	Exceptions
bool	bool enabled	

### bool IsFetcherEnabled ()

This method queries for the fetcher status as set in the database.

Return Type	Arguments	Exceptions
bool	null	

### bool IsFetcherEnabled (string fetcherId)

This method queries for the fetcher status as set in the database.

Return Type	Arguments	Exceptions
bool	string fetcherId	

### string RegisterFetcher (FetcherType type, TaskType[] roles)

This method registers a fetcher by Type and Role. Returned string is the configuration of the fetcher as stored in the database.

Return Type	Arguments	Exceptions
string	FetcherType type, TaskType[] roles	System.ArgumentNullException System.InvalidOperationException

### string RegisterFetcher (FetcherType fetcher)

This method registers a fetcher. Returned string is the configuration of the fetcher as stored in the database.

Return Type	Arguments	Exceptions
string	Fetcher fetcher	System.ArgumentNullException System.InvalidOperationException

### bool UpdateFetcherStatus ()

This method will set the Status of the fetcher as “Online” in the database.

Return Type	Arguments	Exceptions
bool	null	System.ArgumentNullException System.InvalidOperationException

### bool UpdateFetcherStatus (FetcherStatus fetcher)

This method will set the Status of the fetcher as “Online” in the database.

Return Type	Arguments	Exceptions
bool	FetcherStatus fetcher	System.ArgumentNullException System.InvalidOperationException

### Fetcher GetFetcherById (string serviceId)

This method returns a fetcher based on the provided serviceId.

Return Type	Arguments	Exceptions
Fetcher	null	System.ArgumentNullException System.InvalidOperationException

### IEnumerable<Fetcher> GetFetcherList (long since)

This method returns a list of all fetchers currently registered with the JM.

Return Type	Arguments	Exceptions
IEnumerable<Fetcher>	long since	System.ArgumentNullException System.InvalidOperationException

### IEnumerable<Fetcher> GetFetcherPage (int start, int count, long since)

This method returns a list of fetchers currently registered with the JM based on the starting index, number of fetchers and DateTime value.

Return Type	Arguments	Exceptions
IEnumerable<Fetcher>	int start, int count, long since	System.ArgumentNullException System.InvalidOperationException

### bool RemoveFetcher (string fetcherId)

This method removes the fetcher from the database.

Return Type	Arguments	Exceptions
bool	string fetcherId	System.ArgumentNullException System.InvalidOperationException

### string GetFetcherConfig (string fetcherId)

This method gets the fetcher configuration by name as stored in the database.

Return Type	Arguments	Exceptions
string	string fetcherId	System.ArgumentNullException System.InvalidOperationException

### bool StoreFetcherConfig (string fetcherId, string config)

This method stores a unique configuration for a given fetcher in the database.

Return Type	Arguments	Exceptions
bool	string fetcherId, string config	System.ArgumentNullException System.InvalidOperationException

## 2.8: Tasks

This section covers the API calls related to storing, retrieving and updating of Tasks.

### IEnumerable<Task> FetchTask(string fetcherId, int count)

This method fetches a range of tasks as specified by the fetcherID and count. It uses the fetcherID to look up the types registered for that fetcher, and then returns only the registered type of tasks.

Returned are all the queued (fetched) tasks numbering from zero to count.

Return Type	Arguments	Exceptions
IEnumerable<Task>	string fetcherId, int count	System.ArgumentNullException System.InvalidOperationException

### Task FetchTaskById(string fetcherId, guid taskId)

This method fetches a particular task as specified by the taskId.

Return Type	Arguments	Exceptions
<b>Task</b>	string fetcherId, Guid taskId	System.ArgumentNullException System.InvalidOperationException

### IEnumerable<Task> FetchTask(int count, int complexity)

This method fetches a range of tasks as specified by the count. It uses task complexity and returns only the types of tasks that meet the complexity value.

Returned are all the queued (fetched) tasks numbering from zero to count.

Return Type	Arguments	Exceptions
<b>IEnumerable&lt;Task&gt;</b>	int count, int complexity	System.ArgumentNullException System.InvalidOperationException

### IEnumerable<Task> FetchTask(int count)

This method fetches a range of tasks as specified by the count. It uses fetcherID to lookup the types registered and returns only the registered types of tasks.

Returned are all the queued (fetched) tasks numbering from zero to count.

Return Type	Arguments	Exceptions
<b>IEnumerable&lt;Task&gt;</b>	int count	System.ArgumentNullException System.InvalidOperationException

### Task FetchTaskById(string fetcherId, guid taskId)

This method fetches a particular task as specified by the taskId and fetcherId.

Return Type	Arguments	Exceptions
<b>Task</b>	String fetcherId, Guid taskId	System.ArgumentNullException System.InvalidOperationException

### Task FetchTaskById(Guid taskId)

This method fetches a particular task as specified by the taskId.

Return Type	Arguments	Exceptions
Task	Guid taskId	System.ArgumentNullException System.InvalidOperationException

### IEnumerable<Task> UpdateTaskStatus(IEnumerable<Task> task)

This method updates status and progress of a list of tasks which is required to prevent tasks from being reallocated to another fetcher.

It returns a list of tasks cancelled by JM. The fetcher should handle this list.

Return Type	Arguments	Exceptions
IEnumerable<Task>	IEnumerable<Task> task	System.ArgumentNullException System.InvalidOperationException

### IEnumerable<Task> UpdateTaskStatus(string fetcherId, IEnumerable<Task> task)

This method updates status and progress of a list of tasks which is required to prevent tasks from being reallocated to another fetcher.

It returns a list of tasks cancelled by JM. The fetcher should handle this list.

Return Type	Arguments	Exceptions
IEnumerable<Task>	string fetcherId, IEnumerable<Task> task	System.ArgumentNullException System.InvalidOperationException

### IEnumerable<Task> GetTaskListByJobId(Guid id)

This method returns list of tasks for a particular job as specified by the job ID.

Return Type	Arguments	Exceptions
IEnumerable<Task>	Guid id	System.ArgumentNullException System.InvalidOperationException

### int GetTaskCount(TaskStatus status)

This method returns a count of tasks for the given status.

Return Type	Arguments	Exceptions
int	TaskStatus status	

**int GetTaskCount ( TaskType type)**

This method returns a count of tasks for the given tasktype.

Return Type	Arguments	Exceptions
int	TaskType type	System.ArgumentNullException

**int GetTaskCount(TaskStatus status , TaskType type)**

This method returns a count of tasks for the given status and tasktype.

Return Type	Arguments	Exceptions
int	TaskStatus status, TaskType type	

**int GetActiveTaskCount(TaskStatus status)**

This method is meant for active jobs only. It returns a count of tasks for the given status for any jobs in active state.

Return Type	Arguments	Exceptions
int	TaskStatus status	

**int GetActiveTaskCount(TaskStatus status, TaskType type)**

This method is meant for active jobs only. It returns a count of tasks for the given status and tasktype for any jobs in active state.

Return Type	Arguments	Exceptions
int	TaskStatus status, TaskType type	

### **IEnumerable<Task> GetTaskPage(int start, int count, long since)**

This method provides a paged access to the tasks as specified by:

- start** Starting Index
- count** Number of tasks to be retrieved
- since** DateTime value in ticks which should represent a time span back in time. Default value for 'since' is zero for configured time span.

Maximum number of tasks retrieved by default is 100.

Return Type	Arguments	Exceptions
<code>IEnumerable&lt;Task&gt;</code>	<code>int start, int count, long since</code>	<code>System.ArgumentNullException</code> <code>System.InvalidOperationException</code>

### **Task GetTaskById(Guid id)**

This method retrieves a task as specified by the given ID.

Return Type	Arguments	Exceptions
<code>Task</code>	<code>Guid id</code>	<code>System.ArgumentNullException</code> <code>System.InvalidOperationException</code>

### **IEnumerable<Task> GetTaskListByType(TaskType type, long since)**

This method gets a list of tasks based on the task type and time span provided.

Return Type	Arguments	Exceptions
<code>IEnumerable&lt;Task&gt;</code>	<code>TaskType type, long since</code>	

### **IEnumerable<Task> GetTaskPageByType(TaskType type, int start, int count, long since)**

This method gets a paged list of tasks based on the task type, starting index, count and time span values provided.

Return Type	Arguments	Exceptions
<code>IEnumerable&lt;Task&gt;</code>	<code>TaskType type, int start, int count, long since</code>	

### **IEnumerable<Task> GetActiveTaskByMachine (string tcomputerName)**

This method gets a paged list of tasks based on the task type, starting index, count and time span values provided.

Return Type	Arguments	Exceptions
<code>IEnumerable&lt;Task&gt;</code>	<code>string computerName</code>	<code>System.ArgumentNullException</code> <code>System.InvalidOperationException</code>

### **IEnumerable<Task> GetTaskPageBySchedule(int start, int count, long since, long startTimeMin, long startTimeMax, long endTimeMin, long endTimeMax, TaskType type, IEnumerable<TaskStatus> status, ref int totalTaskCount)**

This method gets a list of tasks providing the following criteria:

- startTimeMin
- startTimeMax
- endTimeMin and endTimeMax
- list of taskStatus
- taskType

Those Tasks with startTime within the startTimeMin and startTimeMax, with endTime within the endTimeMin and endTimeMax, and task status similar to the list of provided task status will be returned sorted by the following priority:

1. startTime (earlier first)
2. task priority (higher first)
3. lastUpdateTime (earlier first)

totalTaskCount is the total number of tasks which gets calculated once the query with the above criteria is successfully executed.

Return Type	Arguments	Exceptions
<code>IEnumerable&lt;Task&gt;</code>	<code>int start, int count, long since, long startTimeMin, long startTimeMax, long endTimeMin, long endTimeMax, TaskType type, IEnumerable&lt;TaskStatus&gt; status, ref int totalTaskCount</code>	<code>System.ArgumentNullException</code> <code>System.InvalidOperationException</code>

## 2.9: Jobs

This section covers the API calls related to storing, retrieving and updating of jobs.

### Job QueueJob(Job job)

This method queues a new job and its' tasks to the job store. It returns the queued job along with the tasks. Job and task GUIDs are assigned by JM if not initially set in the call.

Return Type	Arguments	Exceptions
Job	Job job	System.ArgumentNullException System.InvalidOperationException

### bool UpdateJobStatus(IEnumerable<Job> job)

This method updates the status of a list of jobs.

Return Type	Arguments	Exceptions
bool	IEnumerable<Job> job	System.ArgumentNullException System.InvalidOperationException

### Job GetJob(Guid id)

This method gets the job based on the job ID provided.

Return Type	Arguments	Exceptions
Job	Guid id	System.ArgumentNullException System.InvalidOperationException

### bool JobUpdate(Job job)

This method updates job priority and/or status of a job.

Return Type	Arguments	Exceptions
bool	Job job	System.ArgumentNullException System.InvalidOperationException

## DashBoard GetJobDashboard()

This method returns a tightly bound structure of elements for the total number of tasks and jobs.

Return Type	Arguments	Exceptions
<code>Dashboard</code>	null	System.ArgumentNullException System.InvalidOperationException

## IEnumerable<Job>GetJobPage(int start, int count, long since, JobStatus status)

This method gets a paged list of jobs based on the starting index, count and time span specified. The list is also filtered by the Job status provided as parameter.

Return Type	Arguments	Exceptions
<code>IEnumerable&lt;Job&gt;</code>	int start, int count, long since, JobStatus status	System.ArgumentNullException System.InvalidOperationException

## IEnumerable<Job> GetJobPageByTaskStatusAndType(int start, int count, long since, JobStatus status, TaskType tType, TaskStatus tStatus)

This method returns a list of jobs based on the starting index, count and time span specified. The list is also filtered by JobStatus, TaskType and TaskStatus provided as parameters.

Return Type	Arguments	Exceptions
<code>IEnumerable&lt;Job&gt;</code>	int start, int count, long since, JobStatus status, TaskType tType, TaskStatus tStatus	System.ArgumentNullException System.InvalidOperationException

## IEnumerable<Job> GetJobPageWithTaskAndStatus(int start, int count, long since, JobStatus status)

This method gets a paged list of jobs based on starting index, count and time span specified. The list is filtered by JobStatus provided and includes the tasks associated with each job also.

Return Type	Arguments	Exceptions
<code>IEnumerable&lt;Job&gt;</code>	int start, int count, long since, JobStatus status	System.ArgumentNullException System.InvalidOperationException

### **IEnumerable<Job> RemoveJobsWithStatusByTaskStatusAndType (JobStatus status, TaskType tType, TaskStatus tStatus)**

This method removes all the jobs based on JobStatus, TaskType and TaskStatus provided.

Return Type	Arguments	Exceptions
bool	JobStatus status, TaskType tType, TaskStatus tStatus	System.ArgumentNullException System.InvalidOperationException

### **IEnumerable<Job> RemoveJobsWithStatus (JobStatus status)**

This method removes all the jobs of the given status.

Return Type	Arguments	Exceptions
bool	JobStatus status	System.ArgumentNullException System.InvalidOperationException

### **IEnumerable<Job> GetJobPage (int start, int count, long since)**

This method gets a paged list of jobs based on starting index, count and time span provided.

Return Type	Arguments	Exceptions
IEnumerable<Job>	int start, int count, long since	System.ArgumentNullException System.InvalidOperationException

### **IEnumerable<Job> GetJobPageWithTask (int start, int count, long since)**

This method gets a paged list of job along with their Tasks based on starting index, count and time span provided.

Return Type	Arguments	Exceptions
IEnumerable<Job>	int start, int count, long since	System.ArgumentNullException System.InvalidOperationException

### **int GetJobCount (JobStatus status)**

This method returns the total number of jobs present in the database for the given status.

Return Type	Arguments	Exceptions
int	JobStatus status	

### bool RemoveJob (Guid guid)

This method removes the job along with its tasks and error logs from the database as specified by job ID in the parameter.

Return Type	Arguments	Exceptions
bool	Guid guid	System.ArgumentNullException System.InvalidOperationException

### bool RemoveJobs (IEnumerable<Guid> guid)

This method removes all the jobs along with their tasks and error logs from the database as specified by the list of Guids in the parameter.

Return Type	Arguments	Exceptions
bool	IEnumerable<Guid> guid	System.ArgumentNullException System.InvalidOperationException

### bool PurgeAllJobs ()

This method purges all jobs along with related tasks and logs from the database.

Return Type	Arguments	Exceptions
bool	IEnumerable<Guid> guid	System.ArgumentNullException System.InvalidOperationException

### IEnumerable<Job> GetJobPageBySchedule(int start, int count, long since, long startTimeMin, long startTimeMax, long endTimeMin, long endTimeMax, IEnumerable<JobStatus> status, TaskType taskType, ref int totalJobCount)

This method gets a list of Jobs along with its task list providing the following criteria:

- startTimeMin
- startTimeMax
- endTimeMin and endTimeMax
- list of jobstatus
- taskType

Those Jobs with startTime within the startTimeMin and startTimeMax, with endTimes within the endTimeMin and endTimeMax, and job status similar to the list of provided job status will be returned sorted by the following priority order:

1. startTime (earlier first)
2. job priority (higher first)
3. lastUpdateTime (earlier first)

totalJobCount is the total number of jobs which gets calculated once the query with the above criteria is successfully executed.

Return Type	Arguments	Exceptions
<code>IEnumerable&lt;Job&gt;</code>	<code>int start, int count, long since, long startTimeMin, long startTimeMax, long endTimeMin, long endTimeMax, IEnumerable&lt;JobStatus&gt; status, TaskType taskType, ref int totalJobCount</code>	<code>System.ArgumentNullException</code> <code>System.InvalidOperationException</code>

### Job QueueWorkflowJob(Guid watchGuid, string sourceUri, string targetUri)

This method queues a new job providing the watch folder GUID, source URI along with complete file name and optional target/output folder path. This method queues the job formed by the watch folder configuration retrieved for a given watchGuid. This method uses TaskGraph library to generate the tasks and job for a given watchGuid. It returns the queued job along with the tasks. Job and task GUIDs are assigned by JM if not initially set in the call.

Return Type	Arguments	Exceptions
<code>Job</code>	<code>Guid watchGuid, string sourceUri, string targetUri</code>	<code>System.ArgumentNullException</code> <code>System.ArgumentException</code>

### Job QueueWorkflowJobEx(Guid workflowGuid, string sourceUri, string targetUri)

This method queues a new job providing the workflow template guid, source uri along with complete file name and optional target/output folder path. This method queues the job using default single file type watch options and the workflow template configuration retrieved for a given workflowGuid. This method uses TaskGraph library to generate the tasks and job for a given workflowGuid. It returns the queued job along with the tasks. Job and task GUIDs are assigned by JM if not initially set in the call.

Return Type	Arguments	Exceptions
<code>Job</code>	<code>Guid workflowGuid, string sourceUri, string targetUri</code>	<code>System.ArgumentNullException</code> <code>System.ArgumentException</code>

## Job QueueWorkflowTemplateJob(string workflowTemplate, string sourceUri, string targetUri)

This method queues a new job providing the workflow template xml string, source uri along with complete file name and optional target/output folder path. This method queues the job using default single file type watch options and the workflow template configuration. This method uses TaskGraph library to generate the tasks and job for a given workflow tasks. It returns the queued job along with the tasks. Job and task GUIDs are assigned by JM if not initially set in the call.

Return Type	Arguments	Exceptions
<a href="#">Job</a>	string workflowTemplate, string sourceUri, string targetUri	System.ArgumentNullException System.ArgumentException

Sample Workflow Template Structure (which has two transcode tasks) is as shown below:

```
<WorkflowTasks name="Default" guid="{3c2e318e-bfef-49d9-b0ad-ab5231f21f33}"
description="Default">
  <PreTranscodeTaskSet />
  <SourceFiltersTaskSet />
  <TranscodeSet>
    <Target>
      <Title>H.264 Exporter</Title>
      <ModuleGuid>{df227069-514d-4127-bb25-bf31d1e732aa}</ModuleGuid>
      <PresetGuid>{df227069-514d-4127-bb25-bf31d1e732aa}</PresetGuid>
      <Path> "Provide TargetPath Here" </Path>
      <Filename>%s</Filename>
      <Overwrite>1</Overwrite>
      <FilterTaskSet />
      <PostTranscodeTaskSet />
    </Target>
    <Target>
      <Title>Flash Exporter</Title>
      <ModuleGuid>{e868830b-3b2a-48a8-b5db-3943f29e7715}</ModuleGuid>
      <PresetGuid>{e868830b-3b2a-48a8-b5db-3943f29e7715}</PresetGuid>
      <Path>"Provide TargetPath Here" </Path>
      <Filename>%s</Filename>
      <Overwrite>1</Overwrite>
      <FilterTaskSet />
      <PostTranscodeTaskSet />
    </Target>
  </TranscodeSet>
</WorkflowTasks>
```

## Asset GetJobAssets (Guid jobGuid, TaskType taskType, SubTaskType subTaskType)

This method gets a list of sources and targets (in the form of Asset.Sources and Asset.Targets) of the job for the provided jobGuid and the tasks of taskType and subTaskType. Tasks of type Transcode do not have any subTaskType, hence user can specify the subTaskType to SubTaskType.Undefined.

Tasks of type 'Transfer' have sub task types 'Delivery' and 'Retrieval'. User can specify SubTaskType.Delivery or SubTaskType.Retrieval if he/she is using taskType 'Transfer'.

Return Type	Arguments	Exceptions
Asset	Guid jobGuid, TaskType taskType, SubTaskType subTaskType	System.ArgumentNullException System.InvalidOperationException

## 2.10: WatchFolder

This section covers the API calls related to storing, retrieving and updating watch folders.

### bool StoreSnapshot(Guid id, Guid ssid\_basedon, Guid ssid\_new, string snap)

This method stores a snapshot of the WatchFolder in the database if the ssid is valid.

Return Type	Arguments	Exceptions
bool	Guid id, Guid ssid_basedon, Guid ssid_new, string snap	System.ArgumentNullException System.InvalidOperationException

### SnapshotObject GetSnapshot(Guid watchId)

This method retrieves the snapshot for a watch folder as specified by the watch ID.

Return Type	Arguments	Exceptions
SnapshotObject	Guid watchId	System.ArgumentNullException System.InvalidOperationException

### String GetWatchConfigById(Guid watchId)

This method returns watch configuration for a specified watch folder as defined in the database.

Return Type	Arguments	Exceptions
string	Guid watchId	System.ArgumentNullException System.InvalidOperationException

### WatchConfigObject GetWatchFolderById (Guid watchId)

This method returns a watch folder object including its templateId and workflow template of the specified watchId.

Return Type	Arguments	Exceptions
WatchConfigObject	Guid watchId	System.ArgumentNullException System.InvalidOperationException

### IEnumerable<WatchConfigObject> GetWatchConfigPage (int start, int count, long since)

This method returns a list of watch configuration strings as specified by the starting index, count and time span values provided.

Return Type	Arguments	Exceptions
IEnumerable<WatchConfigObject>	int start, int count, long since	System.ArgumentNullException System.InvalidOperationException

### bool StoreWatchConfig(Guid watchId, Guid templateId, string config)

This method stores a new or updates an existing watch folder configuration string in the database.

Return Type	Arguments	Exceptions
bool	Guid watchId, string config	System.ArgumentNullException System.InvalidOperationException

### bool RemoveWatchConfig(Guid watchId)

This method removes the watch folder's configuration from the database as specified by the watchId.

Return Type	Arguments	Exceptions
bool	Guid watchId	System.ArgumentNullException System.InvalidOperationException

## 2.11: Configurations

### bool StoreConfig(string name, string config )

This method stores a fetcher's configuration to the database.

Return Type	Arguments	Exceptions
bool	string name, string config	System.ArgumentNullException System.InvalidOperationException

### bool RemoveConfig(string name)

This method removes the given fetcher's configuration from the database.

Return Type	Arguments	Exceptions
bool	string name	System.ArgumentNullException System.InvalidOperationException

### IEnumerable<ClientConfig> GetConfigList(long since)

This method gets a list of all the configurations stored in the database based on the time span value provided. Since is optional and if it is set to 0, a complete list of configurations is returned.

Return Type	Arguments	Exceptions
IEnumerable <ClientConfig>	long since	System.ArgumentNullException System.InvalidOperationException

### IEnumerable<ClientConfig> GetConfigPage (int start, int count, long since)

This method gets a paged list of the configurations based on starting index, count and time span provided.

Return Type	Arguments	Exceptions
IEnumerable <ClientConfig>	int start, int count, long since	System.ArgumentNullException System.InvalidOperationException

### string GetConfig(string name)

This method gets the configuration of the fetcher as specified by name.

Return Type	Arguments	Exceptions
string	string name	System.ArgumentNullException System.InvalidOperationException

## **DateTime GetJobManagerMachineTime()**

This method gets the primary WFS Controller's date time in UTC.

<b>Return Type</b>	<b>Arguments</b>	<b>Exceptions</b>
<b>DateTime</b>	<code>null</code>	System.ArgumentNullException System.InvalidOperationException



Harmonic Inc.  
4300 North First St.  
San Jose, CA 95134, U.S.A.  
T +1 408 542 2500  
F +1 408 490 6770

[www.harmonicinc.com](http://www.harmonicinc.com)  
[www.rhozet.com](http://www.rhozet.com)  
© 2010 Harmonic Inc. All rights reserved.  
Manual Part No. RHZ-WFS-API-0001