Episode 6.5 Advanced User Guide

161789

telestream

May 19, 2015

Copyrights and Trademark Notices

Third Party Library Notices

The following notices are required by third party software and libraries used in Episode. The software may have been modified by Telestream as permitted by the license or permission to use the software.

X264

Episode includes software whose copyright is owned by, or licensed from, x264 LLC.

SharpSSH2

SharpSSH2 Copyright (c) 2008, Ryan Faircloth. All rights reserved. Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

Neither the name of Diversified Sales and Service, Inc. nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

SQLite

The SQLite website includes the following copyright notice: http://www.sqlite.org/copyright.html. In part, this notice states:

Anyone is free to copy, modify, publish, use, compile, sell, or distribute the original SQLite code, either in source code form or as a compiled binary, for any purpose, commercial or non-commercial, and by any means.

Libxml2

Libxml2 by xmlsoft.org is the XML C parser and toolkit developed for the Gnome project. The website refers to the Open Source Initiative website for the following

licensing notice for Libxml2: http://www.opensource.org/licenses/mit-license.html. This notice states:

Copyright (c) 2011 xmlsoft.org

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in

all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

PCRE

The PCRE software library supplied by pcre.org includes the following license statement:

PCRE LICENCE

PCRE is a library of functions to support regular expressions whose syntax and semantics are as close as possible to those of the Perl 5 language. Release 8 of PCRE is distributed under the terms of the “BSD” licence, as specified below. The documentation for PCRE, supplied in the “doc” directory, is distributed under the same terms as the software itself. The basic library functions are written in C and are freestanding. Also included in the distribution is a set of C++ wrapper functions.

THE BASIC LIBRARY FUNCTIONS

Written by: Philip Hazel
Email local part: ph10
Email domain: cam.ac.uk
University of Cambridge Computing Service,
Cambridge, England.

Copyright (c) 1997-2010 University of Cambridge. All rights reserved.

THE C++ WRAPPER FUNCTIONS

Contributed by: Google Inc.

Copyright (c) 2007-2010, Google Inc. All rights reserved.

THE "BSD" LICENCE

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

* Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

* Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

* Neither the name of the University of Cambridge nor the name of Google Inc. nor the names of their contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Boost C++ Libraries

The Boost C++ Libraries supplied by boost.org are licensed at the following Web site: http://www.boost.org/users/license.html. The license reads as follows:

Boost Software License—Version 1.0—August 17th, 2003

Permission is hereby granted, free of charge, to any person or organization obtaining a copy of the software and accompanying documentation covered by this license (the “Software”) to use, reproduce, display, distribute, execute, and transmit the Software, and to prepare derivative works of the Software, and to permit third-parties to whom the Software is furnished to do so, all subject to the following:

The copyright notices in the Software and this entire statement, including the above license grant, this restriction and the following disclaimer, must be included in all copies of the Software, in whole or in part, and all derivative works of the Software, unless such copies or derivative works are solely in the form of machine-executable object code generated by a source language processor.

THE SOFTWARE IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT. IN NO EVENT SHALL THE COPYRIGHT HOLDERS OR ANYONE DISTRIBUTING THE SOFTWARE BE LIABLE FOR ANY DAMAGES OR OTHER LIABILITY, WHETHER IN CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Libevent

The libevent software library supplied by monkey.org is licensed at the following website: http://monkey.org/~provos/libevent/LICENSE. The license reads as follows:

Libevent is covered by a 3-clause BSD license. Below is an example. Individual files may have different authors.

Copyright (c) 2000-2007 Niels Provos <provos@citi.umich.edu> All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

3. The name of the author may not be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED.

IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

The FreeType Project

The FreeType Project libraries supplied by freetype.org are licensed at the following website: http://www.freetype.org/FTL.TXT. The license reads in part as follows:

Copyright 1996-2002, 2006 by David Turner, Robert Wilhelm, and Werner Lemberg

We specifically permit and encourage the inclusion of this software, with or without modifications, in commercial products. We disclaim all warranties covering The FreeType Project and assume no liability related to The FreeType Project.

Finally, many people asked us for a preferred form for a credit/disclaimer to use in compliance with this license. We thus encourage you to use the following text:

Portions of this software are copyright © 2011 The FreeType Project (www.freetype.org). All rights reserved.

Samba

Samba code supplied by samba.org is licensed at the following website: http://samba.org/samba/docs/GPL.html. The license is a GNU General Public License as published by the Free Software Foundation and is also listed at this website: http://www.gnu.org/licenses/. Because of the length of the license statement, the license agreement is not repeated here.

Ogg Vorbis

The Ogg Vorbis software supplied by Xiph.org is licensed at the following website: http://www.xiph.org/licenses/bsd/. The license reads as follows:

© 2011, Xiph.Org Foundation

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

•Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

•Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

•Neither the name of the Xiph.org Foundation nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

This software is provided by the copyright holders and contributors “as is” and any express or implied warranties, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose are disclaimed. In no event shall the foundation or contributors be liable for any direct, indirect, incidental, special, exemplary, or consequential damages (including, but not limited to, procurement of substitute goods or services; loss of use, data, or profits; or business interruption) however caused and on any theory of liability, whether in contract, strict liability, or tort (including negligence or otherwise) arising in any way out of the use of this software, even if advised of the possibility of such damage.

LibTIFF

The LibTIFF software library provided by libtiff.org is licensed at the following website: www.libtiff.org/misc.html. The copyright and use permission statement reads as follows:

Copyright (c) 1988-1997 Sam Leffler

Copyright (c) 1991-1997 Silicon Graphics, Inc.

Permission to use, copy, modify, distribute, and sell this software and its documentation for any purpose is hereby granted without fee, provided that (i) the above copyright notices and this permission notice appear in all copies of the software and related documentation, and (ii) the names of Sam Leffler and Silicon Graphics may not be used in any advertising or publicity relating to the software without the specific, prior written permission of Sam Leffler and Silicon Graphics.

THE SOFTWARE IS PROVIDED "AS-IS" AND WITHOUT WARRANTY OF ANY KIND, EXPRESS, IMPLIED OR OTHERWISE, INCLUDING WITHOUT LIMITATION, ANY WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

IN NO EVENT SHALL SAM LEFFLER OR SILICON GRAPHICS BE LIABLE FOR ANY SPECIAL, INCIDENTAL, INDIRECT OR CONSEQUENTIAL DAMAGES OF ANY KIND, OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER OR NOT ADVISED OF THE POSSIBILITY OF DAMAGE, AND ON ANY THEORY OF LIABILITY, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

zlib

The zlib.h general purpose compression library provided zlib.net is licensed at the following website: http://www.zlib.net/zlib_license.html. The license reads as follows:

Copyright (C) 1995-2010 Jean-loup Gailly and Mark Adler

This software is provided 'as-is', without any express or implied warranty. In no event will the authors be held liable for any damages arising from the use of this software.

Permission is granted to anyone to use this software for any purpose, including commercial applications, and to alter it and redistribute it freely, subject to the following restrictions:

1. The origin of this software must not be misrepresented; you must not claim that you wrote the original software. If you use this software in a product, an acknowledgment in the product documentation would be appreciated but is not required.

2. Altered source versions must be plainly marked as such, and must not be misrepresented as being the original software.

3. This notice may not be removed or altered from any source distribution.

Jean-loup Gailly, Mark Adler

LAME

The LAME MPEG Audio Layer III (MP3) encoder software available at lame.sourceforge.net is licensed under the GNU Lesser Public License (LGPL) at this website www.gnu.org/copyleft/lesser.html and summarized by the LAME developers at this website: lame.sourceforge.net/license.txt. The summary reads as follows:

Can I use LAME in my commercial program?

Yes, you can, under the restrictions of the LGPL. The easiest way to do this is to:

1. Link to LAME as separate library (libmp3lame.a on unix or lame_enc.dll on windows).

2. Fully acknowledge that you are using LAME, and give a link to our web site, www.mp3dev.org.

3. If you make modifications to LAME, you *must* release these modifications back to the LAME project, under the LGPL.

*** IMPORTANT NOTE ***

The decoding functions provided in LAME use a version of the mpglib decoding engine which is under the GPL. They may not be used by any program not released under the GPL unless you obtain such permission from the MPG123 project (www.mpg123.de). (yes, we know MPG123 is currently under the LGPL, but we use an older version that

was released under the former license and, until someone tweaks the current MPG123 to suit some of LAME's specific needs, it'll continue being licensed under the GPL).

MPEG Disclaimers

MPEGLA MPEG2 Patent

ANY USE OF THIS PRODUCT IN ANY MANNER OTHER THAN PERSONAL USE THAT COMPLIES WITH THE MPEG-2 STANDARD FOR ENCODING VIDEO INFORMATION FOR PACKAGED MEDIA IS EXPRESSLY PROHIBITED WITHOUT A LICENSE UNDER APPLICABLE PATENTS IN THE MPEG-2 PATENT PORTFOLIO, WHICH LICENSE IS AVAILABLE FROM MPEG LA, LLC, 6312 S. Fiddlers Green circle, Suite 400E, Greenwood Village, Colorado 80111 U.S.A.

MPEGLA MPEG4 VISUAL

THIS PRODUCT IS LICENSED UNDER THE MPEG-4 VISUAL PATENT PORTFOLIO LICENSE FOR THE PERSONAL AND NON-COMMERCIAL USE OF A CONSUMER FOR (i) ENCODING VIDEO IN COMPLIANCE WITH THE MPEG-4 VISUAL STANDARD ("MPEG-4 VIDEO") AND/OR (ii) DECODING MPEG-4 VIDEO THAT WAS ENCODED BY A CONSUMER ENGAGED IN A PERSONAL AND NON-COMMERCIAL ACTIVITY AND/OR WAS OBTAINED FROM A VIDEO PROVIDER LICENSE IS GRANTED OR SHALL BE IMPLIED FOR ANY OTHER USE. ADDITIONAL INFORMATION INCLUDING THAT RELATING TO PROMOTIONAL, INTERNAL AND COMMERCIAL USES AND LICENSING MAY BE OBTAINED FROM MPEG LA, LLC. SEE HTTP://WWW.MPEGLA.COM.

MPEGLA AVC

THIS PRODUCT IS LICENSED UNDER THE AVC PATENT PORTFOLIO LICENSE FOR THE PERSONAL AND NON-COMMERCIAL USE OF A CONSUMER TO (i) ENCODE VIDEO IN COMPLIANCE WITH THE AVC STANDARD ("AVC VIDEO") AND/OR (ii) DECODE AVC VIDEO THAT WAS ENCODED BY A CONSUMER ENGAGED IN A PERSONAL AND NON-COMMERCIAL ACTIVITY AND/OR WAS OBTAINED FROM A VIDEO PROVIDER LICENSED TO PROVIDE AVC VIDEO. NO LICENSE IS GRANTED OR SHALL BE IMPLIED FOR ANY OTHER USE. ADDITIONAL INFORMATION MAY BE OBTAINED FROM MPEG LA, L.L.C. SEE HTTP://WWW.MPEGLA.COM.

MPEG4 SYSTEMS

THIS PRODUCT IS LICENSED UNDER THE MPEG-4 SYSTEMS PATENT PORTFOLIO LICENSE FOR ENCODING IN COMPLIANCE WITH THE MPEG-4 SYSTEMS STANDARD, EXCEPT THAT AN ADDITIONAL LICENSE AND PAYMENT OF ROYALTIES ARE NECESSARY FOR ENCODING IN CONNECTION WITH (i) DATA STORED OR REPLICATED IN PHYSICAL MEDIA WHICH IS PAID FOR ON A TITLE BY TITLE BASIS AND/OR (ii) DATA WHICH IS PAID FOR ON A TITLE BY TITLE BASIS AND IS TRANSMITTED TO AN END USER FOR PERMANENT STORAGE AND/OR USE. SUCH ADDITIONAL LICENSE MAY BE OBTAINED FROM MPEG LA, LLC. SEE <HTTP://WWW.MPEGLA.COM> FOR ADDITIONAL DETAILS.

Limited Warranty and Disclaimers

Telestream, LLC (the Company) warrants to the original registered end user that the product will perform as stated below for a period of one (1) year from the date of shipment from factory:

Hardware and Media. The Product hardware components, if any, including equipment supplied but not manufactured by the Company but NOT including any third party equipment that has been substituted by the Distributor for such equipment (the "Hardware"), is free from defects in materials and workmanship under normal operating conditions and use.

Warranty Remedies

Your sole remedies under this limited warranty are as follows:

Hardware and Media. The Company will either repair or replace (at its option) any defective Hardware component or part, or Software Media, with new or like new Hardware components or Software Media. Components may not be necessarily the same, but will be of equivalent operation and quality.

Software. If software is supplied as part of the product and it fails to substantially confirm to its specifications as stated in the product user's guide, the Company shall, at its own expense, use its best efforts to correct (with due allowance made for the nature and complexity of the problem) such defect, error or nonconformity.

Software Updates. If software is supplied as part of the product, the Company will supply the registered purchaser/licensee with maintenance releases of the Company's proprietary Software Version Release in manufacture at the time of license for a period of one year from the date of license or until such time as the Company issues a new Version Release of the Software, whichever first occurs. To clarify the difference between a Software Version Release and a maintenance release, a maintenance release generally corrects minor operational deficiencies (previously non-implemented features and software errors) contained in the Software, whereas a Software Version Release adds new features and functionality. The Company shall have no obligation to supply you with any new Software Version Release of Telestream software or third party software during the warranty period, other than maintenance releases.

Restrictions and Conditions of Limited Warranty

This Limited Warranty will be void and of no force and effect if (i) Product Hardware or Software Media, or any part thereof, is damaged due to abuse, misuse, alteration, neglect, or shipping, or as a result of service or modification by a party other than the Company, or (ii) Software is modified without the written consent of the Company.

Limitations of Warranties

THE EXPRESS WARRANTIES SET FORTH IN THIS AGREEMENT ARE IN LIEU OF ALL OTHER WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, WITHOUT LIMITATION, ANY WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. No oral

or written information or advice given by the Company, its distributors, dealers or agents, shall increase the scope of this Limited Warranty or create any new warranties.

Geographical Limitation of Warranty. This limited warranty is valid only within the country in which the Product is purchased/licensed.

Limitations on Remedies. YOUR EXCLUSIVE REMEDIES, AND THE ENTIRE LIABILITY OF TELESTREAM, LLC WITH RESPECT TO THE PRODUCT, SHALL BE AS STATED IN THIS LIMITED WARRANTY. Your sole and exclusive remedy for any and all breaches of any Limited Warranty by the Company shall be the recovery of reasonable damages which, in the aggregate, shall not exceed the total amount of the combined license fee and purchase price paid by you for the Product.

Damages

TELESTREAM, LLC SHALL NOT BE LIABLE TO YOU FOR ANY DAMAGES, INCLUDING ANY LOST PROFITS, LOST SAVINGS, OR OTHER INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF YOUR USE OR INABILITY TO USE THE PRODUCT, OR THE BREACH OF ANY EXPRESS OR IMPLIED WARRANTY, EVEN IF THE COMPANY HAS BEEN ADVISED OF THE POSSIBILITY OF THOSE DAMAGES, OR ANY REMEDY PROVIDED FAILS OF ITS ESSENTIAL PURPOSE.

Further information regarding this limited warranty may be obtained by writing:
Telestream
848 Gold Flat Road
Nevada City, CA 95959

You can call Telestream at (530) 470-1300.

Part number: 161789

Publication Date: May 19, 2015

Contents

Preface

Support | Information | Assistance

Web Site. www.telestream.net/telestream-support/episode-6/support.htm

Support Web Mail. www.telestream.net/telestream-support/episode-6/contact-support.htm

Company and Product Information

For information about Telestream or its products, please contact us via:

Web Site. www.telestream.net

Sales and Marketing Email. info@telestream.net

Mail

Telestream
848 Gold Flat Road
Nevada City, CA. USA 95959

International Telestream Distributors

See the Telestream Web site at www.telestream.net for your regional authorized Telestream distributor.

We'd Like to Hear From You!

If you have comments or suggestions about improving this document, or other Telestream documents - or if you've discovered an error or omission, please email us at techwriter@telestream.net.

Audience and Assumptions

This guide is intended as a source of information for those who are planning, developing, or implementing automated digital media transcoding and integration solutions with Episode.

This guide is written with the assumption that you possess a general working knowledge of digital media processing, and of Episode. This guide also assumes you have a general knowledge of how to use command line and XML-RPC interfaces, and computer programming, as appropriate.

This guide does not describe how to use Episode in detail from Episode, the graphic user interface program. For information, see the Episode User's Guide.

How this Guide is Organized

This guide is organized into several high-level topics. Click on a heading below to jump to the topic:

Episode Overview

This topic introduces you to Episode's capabilities and its architecture and components, which are important to determining how best to approach a given automation or integration project; as well as concepts upon which Episode is built.

Creating Tasks, Sources, Workflows & Submissions

This topic describes how to create tasks and sources in the various interfaces. Likewise, the topic of creating workflows and submissions is described from a high-level perspective, taking into account the various interface distinctions.

Using Advanced Features

This topic describes Episode's advanced features, which are not available in the Episode GUI program, and can only be used with the CLI or XML-RPC API.

Using the Command Line Interface

This topic describes Episode's Command Line Interface (CLI). The CLI can be used to control Episode in an interactive command line environment, and also for lightweight automation of simple Episode tasks which can be accomplished without traditional programming, using batch files or scripting languages.

Using the XML-RPC Interface

This topic introduces you to Episode's XML-RPC interface—you'll learn how to access the XML-RPC documentation and the constraint XML files.

Episode Overview

This chapter describes the architecture, components, and major features of

Episode, from a system integrator/developer's perspective.

These topics are covered:

XML-RPC and CLI License Requirements

You can use the Episode XML-RPC and CLI interface without special licensing, but you need the appropriate license for the Episode features you are accessing. See the Episode Format Support document on the Telestream.net web site for details.

Note: When utilizing the CLI to execute unlicensed features in demo mode, add the `-demo` flag. In the XML-RPC interface, you can add `-demo` to *submitSubmisssion* and *submitBuildSubmission* to use unlicensed features in demo mode as well.

If you don't have the required licenses as described below, please contact your Telestream representative or contact Telestream directly—see Company and Product Information.

Note: You cannot execute an MBR task (Multi-bitrate) in the CLI unless no Episode license is active (you're using it in demo mode), or the Episode Engine license is active. In demo mode, MBR tasks watermark the output.

If you have any license activated other than the required ones, the MBR task halts with the error: Queued: No available license feature. De-activate the license, then use MBR in demo mode.

Episode Interfaces

There are several ways you can use Episode, by utilizing different interfaces. Each interface provides distinct advantages, exposes certain features, and is best-suited to certain applications.

- Graphic User Interface
- Watch Folder and Deployment Interface
- XML-RPC Interface
- Command Line Interface

The Episode graphic user interface program, implemented for both MacOS X and Windows, is described in detail in the Episode User's Guide.

Topics

Watch Folder and Deployment Interface

The watch folder and deployment interface is a file-based interface. This interface offers easy, file-based integration—no development is required.

You typically use the Episode GUI program to create your workflows with watch folders (for input file integration) and deployments (for output file integration) and then drop files into the watch folder for processing, and fetch output files from the watch folder for utilization.

XML-RPC and CLI Interfaces

The XML-RPC and CLI interfaces are available for both MacOS X and Windows. This guide provides an overview of these interfaces.

Note: For detailed information on the XML-RPC interface or the CLI, refer to the XMLRPC.html file or the CLI.html file on the Telestream.net web site. Links to these documents are also provided in the Episode Online Help, which can be accessed from the Episode Help menu.

XML-RPC Interface

The XML-RPC interface is a standard, language-agnostic, HTTP interface intended for use by integrating it into computer programs.

Note: For information about the XML-RPC standard, see www.xmlrpc.com.

The programmatic interface enables the most robust and flexible integration opportunities, and Telestream recommends that you utilize the XML-RPC interface when creating program-based integration solutions.

Command Line Interface

The Command Line Interface is primarily a user-driven method, for interacting with Episode by typing commands to perform specific tasks. The CLI can also be implemented in scripts and batch files—typically for lightweight automation tasks, where traditional programming is overkill.

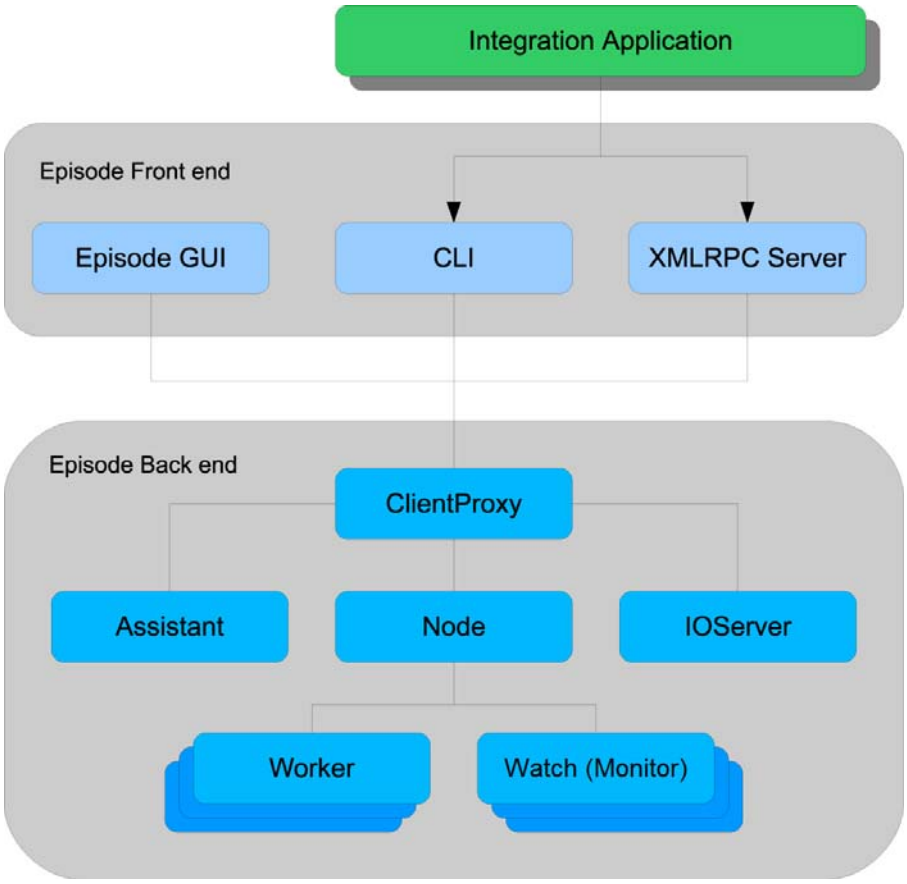
The CLI can be used interactively in the Command program in Windows and the Terminal application in MacOS.

Episode Architecture

Episode consists of a number of processes. These processes are divided into two groups: front-end and back-end processes. Front-end processes consist of user/integration interfaces, graphic user interface (GUI), and the Command Line Interface (CLI) and XML-RPC interface.

Back-end processes consist of those background processes which perform the work in Episode, depending on the usage and configuration of the Episode node(s) and cluster.

Figure 1. Episode front-end and back-end processes.



The background processes are always running by default on Windows, and started and stopped by default when the GUI (Episode.app) is started or quit on MacOS. In Episode for Windows, a number of Windows services are installed which are responsible for starting and stopping background processes. On MacOS, Episode uses *launchd* to run the processes.

Note: You can configure background services in the Episode GUI program. For details, see the User's Guide: Using Episode > Setting Preferences > Advanced.

Topics

Node

A *node* is the main background process in an Episode system. Its main functions are to schedule, distribute and execute jobs, serve the front-end submissions and requests, and maintain both the history database and the active database of jobs.

In a cluster, the node can take on the role as a master node, in which case it is responsible for communicating with and distributing jobs to other nodes in the cluster.

Worker

A *worker* is a process which is designed to execute one task, such as encoding a file, uploading a file to an FTP server, etc. It is a temporal process which executes exactly one task and terminates. A worker is always spawned by a node, and exits when the task is done.

Although a worker is not a background process, it is still a part of the Episode back-end. In a cluster, workers are spawned by the local node on command from the master node, and the worker always connects to the master node to receive its work description. It also receives key information about other nodes in the cluster, such as information on how to access files used in the task, files that may reside on other machines or shared storage. The worker also reports progress, logs messages and status back to the master node, which broadcasts them to all monitoring (connected) front-end processes.

Watch

The *watch* process *(*formerly called *monitor process*, now deprecated*)* is responsible for running one watch folder source configuration. It is, like a worker, a temporal process spawned by a node. Watch processes are not distributed in a cluster so all watches run on the master node. The watch reports file URLs back to the master node which takes appropriate actions, typically to spawn a new started workflow instance from the associated template workflow. The watch's logging messages are reported to the master node, which broadcasts them to all front-end processes.

IOServer

The *IOServer* process is used to enable file transfers and remote encoding, without requiring shared storage. See Shared Storage for how to optimize a clustered setup with shared storage.

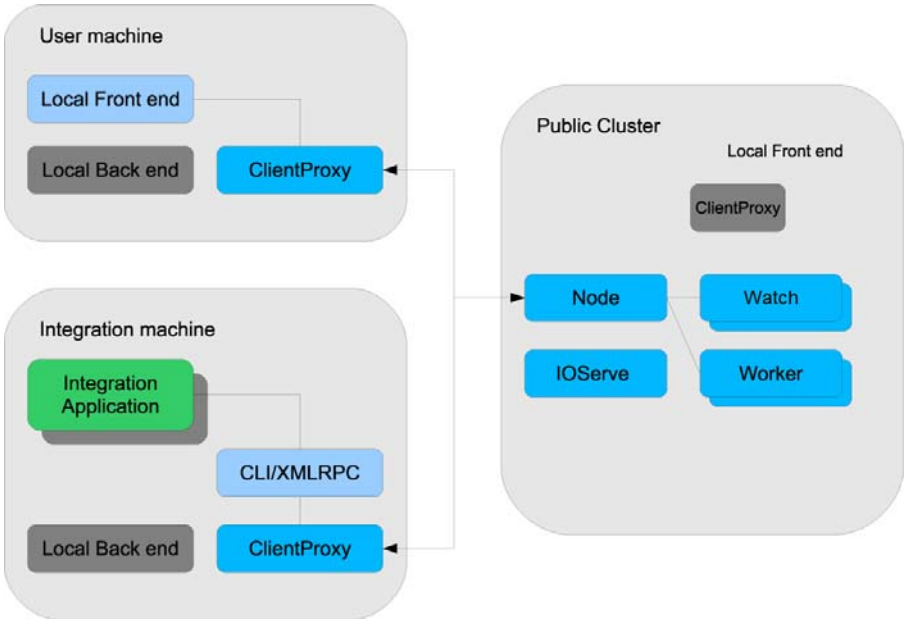
Assistant

The *Assistant* process performs common internal tasks for the Episode front-end such as browsing. It has no significant role in the system from the perspective of the end user.

ClientProxy

The *ClientProxy* process is the front-end's gateway to a node (or a cluster). It assists the front-end to create/read/write configuration files, build workflows, and prepare it all for submission to a node (local or remote). The ClientProxy is always the gateway for the local computer's front-end only, but can contact any remote public node—for example another node in cluster-mode.

Figure 2. Episode ClientProxy connections.



The ClientProxy keeps any connection alive after the first connection request by the front-end. ClientProxy gets status updates from the node it is connected to and caches history for a configurable time period (default: 6 hours). This is mainly for the purpose of integration status polling. For example, a finished job (successful or failed) is accessible for a reasonable time after it is finished without sending history requests to the node.

Episode Processes

This topic describes how to manage back-end processes on both MacOS and Windows, and how to configure them.

Topics

Managing Back-end Processes (MacOS)

On MacOS, Episode's background processes include:

- EpisodeNode
- EpisodeClientProxy
- EpisodeIOServer
- EpisodeAssistant
- EpisodeXMLRPCServer

These processes are launched by using `launchd (man launchd, man launchd.plist, man launchctl)`. When you start these processes via the CLI (and the Episode GUI client starts them), they generate plist files in the directory *~/Library/Application Support/Episode/* and start up. When you shut down these processes you (or the Episode GUI program does so automatically on exit), remove the launchd job by label.

Note: Be sure to supply the path to the command, and enclose it in quotes to permit spaces in the path. For example, from the root: *'/Applications/Episode.app/Contents/Resources/engine/bin/Episodectl' launch start*.

If the processes are *installed*—that is, symbolic links are created in *~/Library/LaunchAgents/*—the back-end processes are started when the user logs in. If you want the processes to launch when you start the computer, you have to manually copy or link the files into */Library/LaunchAgents/*.

It is a good idea to copy the files so a new `launchd` setting can be added to the plist file, the UserName directive that tells `launchd` which user to run the processes as, see `man launchd.plist` for more information.

Managing Back-end Processes (Windows)

On Windows, Episode's background processes include:

- EpisodeNode.exe
- EpisodeClientProxy.exe
- EpisodeIOServer.exe

- EpisodeAssistant.exe
- EpisodeXMLRPCServer.exe.

Each process has a corresponding Windows service installed. The processes are started and stopped via this service, either through the Windows Services control panel or through the Episode CLI, using these commands:

Note: Be sure to supply the path to the command, and enclose it in double quotes to permit spaces in the path. For example, from the root: *"C:\Program Files\Telestream\Episode 6\bin\episodectl.exe' launch start*.

- `episodectl.exe launch start`
- `episodectl.exe launch stop`
- `episodectl.exe launch list`
- `episodectl.exe launch restart`

Note: On a computer with UAC enabled, when attempting to start, restart, stop, or list services, Windows may display an error: "Failed to open service (access is denied". To resolve the problem, disable UAC.

Back-end Process Configuration

All back-end processes have a configuration file in XML format, except the temporal worker and watch processes. Some configuration options are either available in the Episode GUI program or configurable through the CLI, but most are not.

If a configuration setting is edited manually, the affected process has to be restarted in order for the change to take effect.

Table 1. Configuration File Directory by Operating System

Operating System	Configuration File Directory
MacOS	~/Library/Application Support/Episode/
Windows 7 & 8, Windows Server 2008 & 2012	C:\ProgramData\Telestream\Episode 6\

The processes that you may need to configure are the Node and the ClientProxy services, and in some cases the IOServer process.

Documentation for most settings is located directly inside the configuration files.

Documentation for CLI-configurable settings is available using these commands:

- `episodectl node -h`
- `episodectl proxy -h`
- `episodectl ioserver -h`

Note: Be sure to provide a fully-qualified path to the *episodectl* command, and use quotes (Mac OS X) or double quotes (Windows) if there are spaces in the path.

See the Episode User's Guide for information regarding configuration settings available in the Episode GUI program.

Episode Concepts and Components

To the user of the Episode graphic user interface program, Episode acts like a single application. This is a convenient ruse—Episode is functionally a collection of services and servers, utilized by Episode (the graphic user interface client program), to configure and operate Episode. As you can see, the term Episode refers not only to the graphic user interface client, but also the entire collection of services that comprise the Episode system.

In addition to Episode, you can utilize Episode system via other clients—programs that utilize the XML-RPC interface, plus the command line interpreter client. Understanding Episode concepts and components, along with an architectural understanding of how they relate, helps you get the most out of Episode.

Workflows, Tasks, and Sources

These components are the building blocks of Episode.

Topics

Workflows

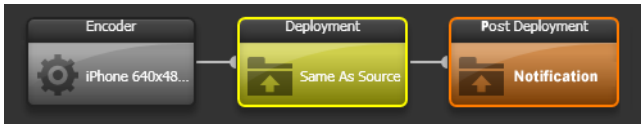
An Episode workflow is a collection of Episode tasks and task interdependencies.

Workflows, as described (and displayed) in the Episode User's Guide, are always comprised of a Source, Encode, and a Deployment task—this is the pattern always used in every workflow.

Figure 3. Episode workflow pattern as shown in the GUI.

From a system perspective, this is a bit of a misnomer. In actuality, the Source task is not actually a part of the workflow—it is a separate template (as defined) and process (when executing) that resolves the input dependency for the Encode task, and submits jobs to the actual workflow: the Encoder, Deployment task, and optional Post-deployment task, as defined in the target workflow.

Figure 4. Episode actual workflow pattern as used in an API.



Note: This distinction is important to understand and take into consideration when utilizing the APIs to implement Episode solutions and utilize them.

Tasks

A *task* in Episode is a specific unit of work to perform—for example, encode a file, or copy a file. Tasks exist in the context of a workflow, and have two states: a template (or definition), and a process, when executing.

A task can range from complex, such as encoding a file, to very simple, such as deleting a file. There are nine types of Episode tasks:

- Encode
- Transfer (Deploy in GUI)
- YouTube
- Execute
- Mail
- MBR
- Move
- Delete
- Localize

All tasks have a configuration, which describes how to perform the work. A Delete task, for example, must be provided a valid string, which identifies which file it should delete, while an Encoder task must be provided the format it should use to encode a file.

Tasks are always one of four types: Source, Encoder, Deployment, and Post-deployment Processing. Post-deployment Processing tasks are not exposed in the Episode GUI program; they can only be configured and used in an API.

Tasks may be independent of other tasks, or they may depend on other tasks.

Figure 5. Tasks are the building blocks of a workflow.

As an illustration, this example workflow is comprised of three tasks—a file localize task, an encode task, and a deployment task. Each of these tasks has a configuration specific to its task type.

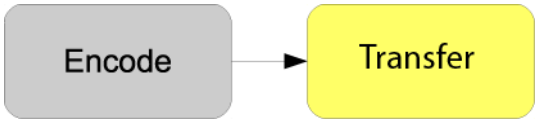
The encoder task has a configuration dependency—the URL of the localized input file. Similarly, the deployment task also has a dependency - the URL of the file created by the encode task. The encode and deployment tasks also have task dependencies - the previous task executing and exiting successfully.

Tasks that are not connected downstream of another task (such as this example's file task), may have unconnected run-time dependencies, which must be set and supplied externally. For example, if you have a watch folder source task, it creates a run-time dependency of a file for input. When the file is supplied (dropped into a folder), that dependency is resolved and a job is submitted.

The file task requires a fully-qualified path to the input file which it should localize. To supply this path, you could use an external monitor system via the XML-RPC interface, or you could call the file task from the command line interface to supply the required path, or the path could be supplied by Episode itself.

The order of task execution is controlled by task interdependencies. These can be the result of another task (for example, success or failure), or by a delivered value from another task—the URL of a produced file, for example. In the following figure, the Encode task delivers the URL of the encoded output file to the Transfer task.

Figure 6. Simplest Episode workflow.



When Episode is directed to process a workflow (for example, the user clicks the Submit button in the Episode client application) there is always an Episode Source accompanying it (this combination of source and workflow is referred to as an Episode Submission).

Sources

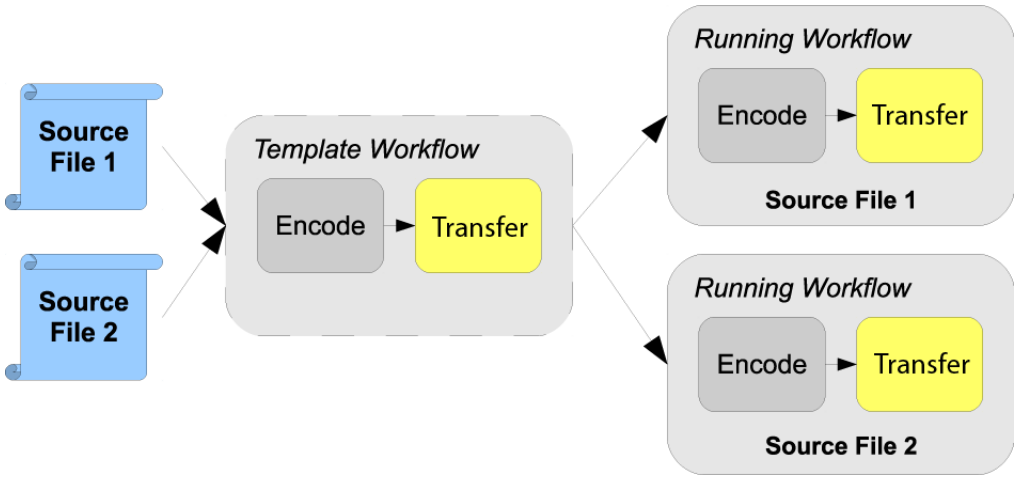
There are four types of Episode Sources:

- File List
- Watch Folder
- EDL
- Image Sequence

Ultimately, an Episode source specifies which file(s) the workflow should operate on and how it should interpret the files. For example, an Image Sequence source specifies that the files should be interpreted as frames in a movie, whereas a File List source specifies separate movies.

Episode sources always operate on template workflows. Template workflows can not run by themselves, because they have no source file to operate on. When an Episode Source operates on a template workflow, a started workflow (which contains the information about the source-file to work on), is created from the template workflow.

Figure 7. Episode template workflow spawning started workflows.



The tasks in the started workflow are then executed. Template workflows are displayed in the left panel of the Episode client application's Status window. Started workflows are displayed in the right panel of the Status window.

For most types of submissions, the template workflow exists only temporarily. For example, when an Episode Submission with a File List source is submitted:

1. The template workflow in the submission is created
2. For each file in the file list a started workflow is spawned
3. The template workflow is discarded
4. The tasks in the started workflows are executed.

For submissions containing watch sources, the template workflow exists as long as the watch folder exists. For each file the watch picks up, a started workflow is created.

Post-deployment Processing Tasks

Post-deployment tasks are also part of a workflow. These are optional, advanced feature tasks (such as email notification and execute tasks) that you can only define and execute via one of the APIs.

Variables

Sometimes it's desirable (or necessary) to add dynamic elements to a workflow. A basic dynamic example—and one which is part of every workflow by default—is to create an output name that is based on the name of the source file and the type of Encoder task used to encode the file.

The file-naming pattern in this example is a configuration in the Transfer task, which specifies how to construct the output file name. Variables may be used in a wide range of other task configurations. Examples include mail message construction, execute task environment variables and arguments, YouTube descriptions etc.

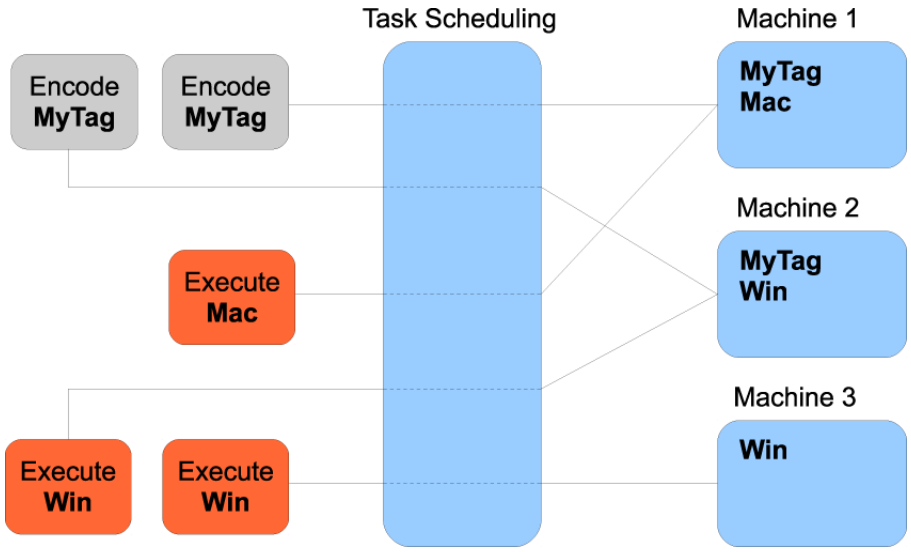
Execute `episodectl variables` for a description of all variables.

Episode Tags

The concept of *tags* in Episode is used to enable an easy way of controlling execution of tasks in a cluster. For example, you can use a tag to control which node, computer, or even group of computers a certain task should run on.

Tags are used primarily by the Execute task (or Script task), an advanced feature which is often dependent on the operating system, scripting software, or languages that are on the platform where the node is installed.

Figure 8. Tags are used to control workflow execution.



Nodes can only be configured using the CLI, directly on the target node; they can be configured on one or more machines in a cluster. Workflows (or tasks in a workflow) are then configured to only run on machines with a certain tag, or to *not* run on a machine with a certain tag (in both CLI and XML-RPC interfaces).

Execute `episodectl tags` for configuration directives and examples.

Creating Tasks, Sources, Workflows & Submissions

The purpose of this chapter is to functionally describe how to create tasks and sources in the various interfaces. Likewise, the topic of creating workflows and submissions is described from a high-level perspective, taking into account the various interface distinctions.

These topics are covered:

Notes: When executing a CLI command, be sure to supply the path to the command, and enclose it in double quotes to permit spaces in the path.

Be sure to provide a fully-qualified path to the *episodectl* command, and use quotes (Mac OS X) or double quotes (Windows) if there are spaces in the path.

For example, on Mac OS X, from the root: *'Applications/Episode.app/Contents/Resources/engine/bin/episodectl' launch start*.

On Windows, from the root: *"C:\Program Files\Telestream\Episode 6\bin\episodectl.exe" launch start*.

A folder is defined as a path ending with a path separator. On Windows, if you quote the string, you must either escape the backslash (\\) or use slash (/) as the last separator.

When using ! (exclamation) characters in bash arguments, they must be escaped, because bash parses the command before *episodectl* and will throw errors.

On Windows, you can only execute *episodectl launch* (and control the Episode system services) in the CLI if Windows UAC is disabled (turned off).

Creating Tasks

To create a task, you create a task configuration file. This file specifies what the task should do when it is executed. These configuration files are saved as .epitask files (a file with an epitask extension).

Note: Beginning with Episode 6.4, the *Uploader* task has been renamed *Transfer* in both the CLI and the XML-RPC interfaces, although the term *Uploader* still can be used, and remains backward-compatible.

These task files can be created in all interfaces with a few exceptions—see the tables below:

Table 2. Creating Tasks in the Episode GUI Program

Tasks	Command	Default Save Location
Encoder	New Task > New Encoder	OS X: ~/Library/Application Support/ Episode/User Tasks/Encoders/
	File > New > Encoder	Windows 7, 8, Server 2008, 2012: C:\ProgramData\Telestream\Episode 6\User Tasks\Encoders\
	Drag Encoder template into drop area	Windows 7, 8, Server 2008, 2012: C:\ProgramData\Telestream\Episode 6\User Tasks\Encoders\
Transfer	New Task > New Deployment	OS X: ~/Library/Application Support/ Episode/User Tasks/Deployments/
	File > New > Deployment	Windows 7, 8, Server 2008, 2012: C:\ProgramData\Telestream\Episode 6\User Tasks\Deployments\
	Drag folder into drop area	Windows 7, 8, Server 2008, 2012: C:\ProgramData\Telestream\Episode 6\User Tasks\Deployments\
	YouTube—Drag in YouTube template	Windows 7, 8, Server 2008, 2012: C:\ProgramData\Telestream\Episode 6\User Tasks\Deployments\

Table 3. Creating Tasks using the Episode CLI

Tasks	CLI Command	Default Save Location
Transfer	`episodectl task transfer`	Current working directory
YouTube	`episodectl task youtube`	
Execute	`episodectl task execute`	
Mail	`episodectl task mail`	
MBR	`episodectl task mbr`	

Table 4. Creating Tasks using the Episode XML-RPC Interface

Tasks	XML-RPC Method	Default Save Location
Transfer	`taskCreateTransfer`	File content returned in response
YouTube	`taskCreateYouTube`	
Execute	`taskCreateExecute`	
Mail	`taskCreateMail`	
MBR	`taskCreateMBR`	

For detailed information about these tasks, see the CLI documentation using the CLI command `episodectl task -h`.

Task configuration files are saved in XML format so they can be easily edited, although manual editing is not recommended unless necessary.

Some tasks can be created on-the-fly when performing a submission through the CLI or XML-RPC interfaces. For example, a destination (output) directory can be specified instead of a Transfer task file, in which case a default configuration will be created automatically for that destination directory.

Certain common configuration values, such as naming convention for the output file, have specific options in the submission commands. For example, the `--naming` option in the CLI and the naming property in the XML-RPC interface. These configuration names and values are also referred to as variables. See Variables for more information.

Setting Task Priority

Priority is only one of the parameters considered when the Node schedules tasks for execution. Other parameters are license requirements, platform requirements, user defined Tags, and a sequential number given to each workflow when it is submitted—that acts as a tie-breaker when everything else is equal. When priority and other requirements are equal, the sequence number makes it like a workflow queue: the first submitted workflow is the first to be distributed for execution.

Two different priorities can be configured prior to workflow submission: a task priority and a (template) workflow priority. The workflow priority is used as an initial task priority adjustment when the workflow is spawned (when the workflow and its tasks are created). It is possible to change the (template) workflow priority for a persistent workflow. That is, for a workflow attached to a watch folder source, but for spawned (started) workflows, the priority is a read-only constant value. After a workflow is spawned, the task(s) priority is the only priority that can be altered and it is the priority used when scheduling tasks for execution.

Two different priorities are implemented because it enables the user to decide which is more important—individual tasks (for example, a certain Encode task) or the source file, or where the source file came from. For example, a certain customer or a certain watch process.

XML-RPC and CLI Priority Commands

For workflows, priority is always set/configured at the time of submission. In the GUI you use the priority control.

In XML-RPC the priority option is available in the `submitBuildSubmission` and `submitSubmission` commands.

In the CLI, `--priority` is used. All creatable tasks (`taskCreateTransfer | taskCreateYouTube | taskCreateExecute | taskCreatemMail | taskCreateMBR`) have the `--priority` option.

Since there currently is no way to create Encode tasks using the CLI and not changeable via XML-RPC, there is a command for setting priority in an existing Encode epitask file: `episodectl.exe task set <path to existing task file> --priority <priority>`.

During run-time (after submission time/workflow spawning), the task(s) priority may be changed with the XML-RPC command `jobSetPriority`, and the CLI command `episodectl.exe job set-priority`, The initial task priority adjustment can be set on workflows attached to watch processes with the XML-RPC command `monitorSetPriority` and the CLI command `episodectl.exe watch-folder set-priority` (formerly `episodectl.exe monitor set-priority, now deprecated`).

Creating Sources

Episode supports several types of sources: File List, Watch Folder, EDL and Image Sequence. Except for EDL and Image Sequence sources, which are not available in the Episode GUI program, all sources can be created in all interfaces.

Note: Sources are saved in the Episode GUI program as .epitask files, although they are not strictly tasks by definition. In the CLI, sources are saved as files with the *.episource* file extension.

Table 5. Creating Sources using the Episode GUI Program

Sources	Command	Default Save Location
File List	Drag files into source drop area	MacOS X: ~/Library/Application Support/Episode/ User Tasks/Sources/
		Windows 7, 8, Server 2008, 2012: C:\ProgramData\Telestream\Episode 6\User Tasks\Sources\
Watch Folder	Drag folder into source drop area	Windows 7, 8, Server 2008, 2012: C:\ProgramData\Telestream\Episode 6\User Tasks\Sources\

Table 6. Creating Sources using the Episode CLI

Sources	Command	Default Save Location
File List	`episodectl source filelist`	Current working directory
Watch Folder	`episodectl source watch-folder`	
EDL	`episodectl source edl`	
Image Sequence	`episodectl source iseq`	

Table 7. Creating Sources using the Episode XML-RPC Interface

Sources	Command	Default Save Location
File List	`sourceCreateFileList`	File content returned in response
Watch Folder	`sourceCreateMonitor`	
EDL	`sourceCreateEDL`	
Image Sequence	`sourceCreateISEQ`	

Some sources can be created on-the-fly when performing a submission through the CLI or XMLRPC interfaces. For example, a list of source files will automatically create a File List source, and a directory could automatically create a default Watch Folder configuration for that directory.

Creating Workflows and Submissions

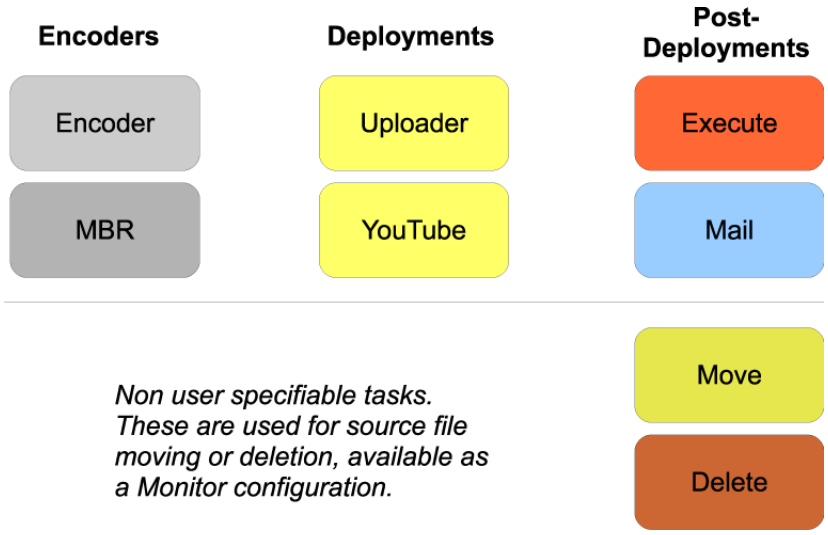
Workflows are created interactively in the Episode GUI program. Using the CLI and XML-RPC interface, they are created on-the-fly – that is, the workflow configuration is part of the submission command. The command in the CLI is `episodectl workflow submit`; in XMLRPC, it is `submitBuildSubmission`.

Note: When submitting a submission with *submitSubmission* (XMLRPC) or *episodectl ws -s...* (CLI), you can optionally override the source in the prebuilt submission with another provided source. The overriding source must be the same source type as the source in the prebuilt submission.

For example, if the prebuilt submission (the submission specified after *-s* in the CLI) has a *file-source*, it can only be replaced by another file-source (not a watch-, edl-, nor iseq-source).

Episode has three distinct groups of (user-specifiable) tasks: Encoders, Deployments, and Post-deployment tasks.

Figure 9. Encoder, Deployment, and Post-Deployment tasks

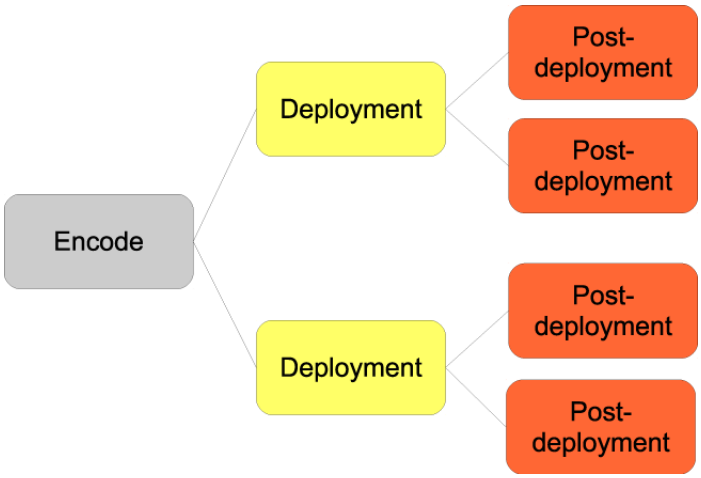


A workflow is built as a tree, branching out from Encoder actions to Deployment to Post-deployment actions. In the Episode GUI program, you specify a Deployment for each Encoder task. However, in CLI and XML-RPC, the default behavior is that you specify a Deployment for all Encoders.

During task execution, Deployments that are specified in the submit command are only executed after every Encoder in the submission has executed. Likewise, Post-deployment tasks in the submit only run after every Deployment in the submission has executed. Thus, depending on the number of Encoders or Deployments in the submit, the Deployments and Post-deployment tasks might be automatically replicated to the empty branches, for the workflow to execute correctly.

This effect of copying tasks should be taken into consideration when polling for status.

Figure 10. Workflows use a tree structure in CLI and XML-RPC.



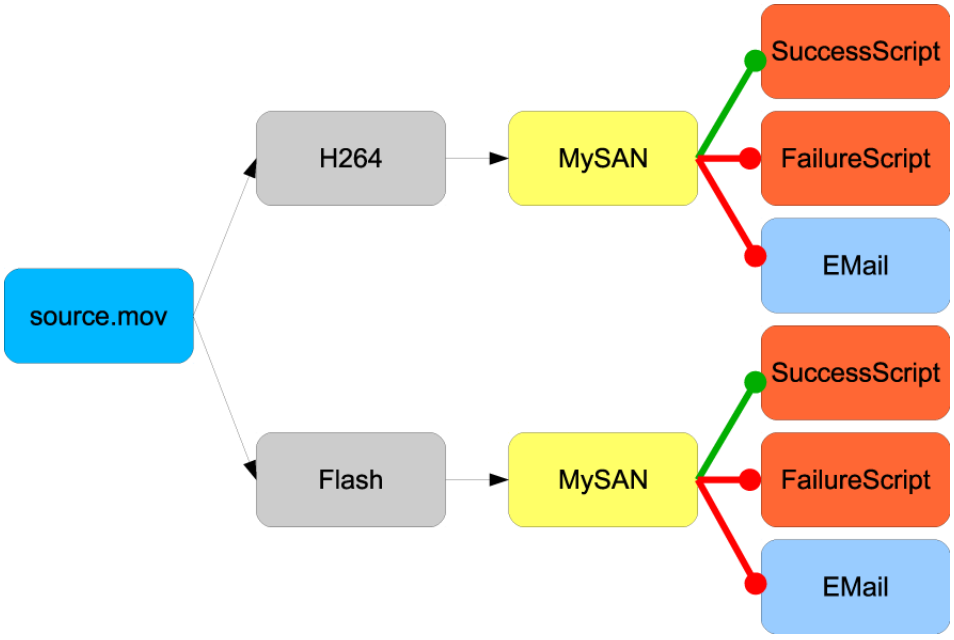
The execution of Post-deployment tasks are always controlled by the success or failure of a Deployment task. A Deployment task is passed a failure status if either the deployment fails or if the preceding Encode task fails. It is passed the success status only if both the preceding Encode task succeeds and the Deployment succeeds. In other words, a Post-deployment configured to run on success will only run if *all* preceding tasks succeeds and a Post-deployment configured to run on failure will run if *any* preceding task fails.

Below is an example CLI submission command (with options on separate lines for clarity only) with a typical workflow – two Encoders, a single Deployment task, and one Execute task that runs in case of failure and one in case of success. It also has a Mail task that sends an email in case of failure. The submission is accompanied by a single source file. Also, notice the copying/branching of the Deployment task and the Post-deployment tasks.

```
episodectl workflow submit
--file source.mov
--encoder H264.epitask Flash.epitask
--destination MySAN.epitask
--execute SuccessScript.epitask success FailureScript.epitask
failure
--mail EMail.epitask failure
```

This command produces a workflow like this in the Episode GUI program:

Figure 11. Example workflow.



After workflows are submitted, two kinds of IDs can be retrieved. One ID is the template workflow ID – the parent ID of the whole submission – from which any number of started workflows may be spawned. The other IDs are the individual started workflow IDs. The IDs can be used to obtain status about the submission's components, a group of workflows (parent-ID/template ID) individual workflows (started workflow ID) or the individual tasks within those workflows. The IDs may also be used to stop workflows.

Using Advanced Features

This chapter describes Episode's advanced features.

These topics are covered:

Notes: When executing a CLI command, be sure to supply the path to the command, and enclose it in double quotes to permit spaces in the path.

Be sure to provide a fully-qualified path to the *episodectl* command, and use quotes (Mac OS X) or double quotes (Windows) if there are spaces in the path.

For example, on Mac OS X, from the root: *'Applications/Episode.app/Contents/Resources/engine/bin/episodectl' launch start*.

On Windows, from the root: *"C:\Program Files\Telestream\Episode 6\bin\episodectl.exe" launch start*.

A folder is defined as a path ending with a path separator. On Windows, if you quote the string, you must either escape the backslash (\\) or use slash (/) as the last separator.

When using ! (exclamation) characters in bash arguments, they must be escaped, because bash parses the command before *episodectl* and will throw errors.

On Windows, you can only execute *episodectl launch* (and control the Episode system services) in the CLI if Windows UAC is disabled (turned off).

Advanced Features

Certain Episode features are termed *advanced features* and may be available only in the CLI or API, or may require the Episode Pro or Engine license. If you don't have the required license, please contact your Telestream representative, or contact Telestream directly—see Company and Product Information.

Workflow jobs using advanced features available only in the XML/RCP or CLI interface are displayed in the Episode graphic user interface's status window, but they cannot be displayed in the workflow editor—if you attempt to display them, Episode displays a dialog indicating they cannot be displayed.

For detailed information about the XML-RPC interface or the CLI, refer to the XMLRPC HTML or CLI HTML descriptions on the Telestream.net web site.

Advanced Sources

- **Image Sequence Input**. Enables you to submit image sequences, including DPX, TGA, TIFF, JPEG and PNG formats, or create watch folders to watch for image sequences and submit them to a workflow for transcoding.

Note: For a detailed list of supported image sequence formats, see the published Episode Format Support sheet at Telestream.net.

- **Edit Decision List (EDL) Conforming**. Enables you to create and submit an Episode EDL source which identifies a set of source files to be combined into a single output file. Each file in the EDL can be trimmed based on time-code or time.

When using EDL's as a source, your workflow must observe these constraints:

- You can't add intro/outro to encoders
- Both video and audio tracks must be present
- Encoder can not copy tracks
- Encoders must not have streaming enabled.

Advanced Encoding

- **Microsoft Smooth Streaming**. Enables you to create multi-bitrate Microsoft Smooth-Streaming packages for Web and Microsoft-compatible devices.
- **Apple HLS Streaming**. Enables you to create multi-bitrate segmented streaming packages for Web and Apple devices.

Advanced Post-Deployment Tasks

The following tasks can be executed from the Mac or Windows command line using the Episode command line interface. When entries contain spaces, remember to enclose them in single quotes for Mac (‘), double quotes for Windows (“). Also recall that Windows uses backslashes, Mac forward slashes. For details of CLI operation, please see Using the Command Line Interface (page 57).

- **Email Notification Task**. Enables you to send custom email notifications as part of your workflow, after the deployment task executes. See Table 8.
- **Execute Task**. Trigger user-written or 3rd party scripts (or programs) as part of your workflow to expand the functionality available in your workflow, after the deployment task executes.

Note: On Windows, Execute tasks sometimes do not function as expected. These failures may occur because of incorrect permissions, file extensions associated with the wrong application, or the task being run in a process spawned by a service running under the local system user. Using a variable such as %USERNAME% may also cause a failure. Lastly, the `--parse-progress` argument is not supported on Windows.

Table 8. Mail Notification Example CLI Commands

Mail Tasks	Enter these commands
E-mail on Job Success (**Note:** In Windows, leave out ./ and use back slashes in all paths)	Start from this directory: **Mac:** */Applications/Episode.app/Contents/Resources/engine/bin/* **Win:** *C:\Program Files\Telestream\Episode 6\bin*
Create mail task	./episodectl task mail
User name for outgoing mail	-u username@domain.com
Password for outgoing mail	-p PASSWORD
Server for outgoing mail	-s mailservername.domain.com
From mail sender address	-f username@domain.com
To mail address	-t username@domain.com
Mail subject (can use $variables)	--subject '$source.file$ encoded successfully'
Mail message	--message 'Task completed successfully' (Windows: use double quotes--" ".)
Name the epitask	--name ENCODE_SUCCESS
Save the epitask in...	-o /Users/myuser/Desktop/CLI/MailTask/mail-tasks/
E-mail on Job Failure (**Note:** In Windows, leave out ./ and use back slashes in all paths)	Start from this directory: **Mac:** */Applications/Episode.app/Contents/Resources/engine/bin/* **Win:** *C:\Program Files\Telestream\Episode 6\bin*
Create mail task	./episodectl task mail

Table 8. Mail Notification Example CLI Commands (continued)

Mail Tasks	Enter these commands
User name for outgoing mail	-u username@domain.com
Password for outgoing mail	-p PASSWORD
Server for outgoing mail	-s mailservername.domain.com
From mail sender address	-f username@domain.com
To mail address	-t username@domain.com
Mail subject (can use $variables)	--subject 'ERROR: $source.file$ encode failed' (Windows: use double quotes--" ")
Mail message	--message 'Task failed and needs attention' (Windows: use double quotes--" ")
Name the epitask	--name ENCODE_FAILED
Save the epitask in...	-o /Users/myuser/Desktop/CLI/MailTask/mail-tasks/

Table 8. Mail Notification Example CLI Commands (continued)

Mail Tasks	Enter these commands
CLI Workflow Commands (**Note:** In Windows, leave out ./ and use back slashes in all paths)	Start from this directory: **Mac:** */Applications/Episode.app/Contents/Resources/engine/bin/* **Win:** *C:\Program Files\Telestream\Episode 6\bin*
Submit a workflow	./episodectl ws
Choose an episubmission file (which includes source, encoder, and destination)...OR... Choose a source file	-s /Users/myuser/Desktop/CLI/MailTask/myworkflow.episubmission -f /Users/myuser/Desktop/CLI/MailTask/filename.mov
Choose a previously saved encode epitask	-e /Users/myuser/Desktop/CLI/MailTask/EncodeOP1a.epitask
Select destination directory for encoded file	-d /Users/myuser/Desktop/CLI/MailTask/output/
Select previously created epitask to send email when workflow is successful	-x /Users/myuser/Desktop/CLI/MailTask/mail-tasks/ENCODE_SUCCESS.epitask **success**
Select epitask to send email when workflow has failed	-x /Users/myuser/Desktop/CLI/MailTask/mail-tasks/ENCODE_FAIL.epitask **failure**
To see progress in the CLI	-wv
List available mail task options	./episodectl task mail -h

Advanced Clustering

A cluster consists of nodes (EpisodeNode process). A node is considered *private* when it is in default mode, and *public* when it is in cluster mode. When the node is public, it is remotely accessible for clients and other nodes that may be part of the same cluster. A cluster consists of one or many nodes, but clients can only communicate with the master node. In a low-volume implementation, even a one node cluster (on a dedicated computer) can be used to encode files for multiple clients running on desktop computers.

A cluster can be created either by using Bonjour or by specifying IP addresses or host names. The choice of method is mostly dependent on how dynamic a cluster should be. If computers are joined ad-hoc where participating computer can easily come and go, we suggest using Bonjour. If a cluster is mostly static—the cluster is made up of dedicated computers that are considered permanent over time, it's usually better to join them together by address.

Topics

Clustering Configuration

The node's configuration identifies it as a master or participant. It also specifies if it should use Bonjour to find a cluster master, publish itself on Bonjour (that is, be visible on the network), or contact a master node by address. You can manually edit the node's configuration file or use the CLI to configure the node at run-time. If the configuration is edited, the node process has to be restarted to pick up the new configuration.

There are six main clustering configuration settings in a node:

- Active—If the node is in cluster mode, i.e. public mode.
- Backup—If the node should be the master.
- Name—The name of the cluster to be a part of.
- Search—If Bonjour should search for the master of the cluster.
- Publish—If the node should publish itself on Bonjour.
- Hosts—The address of the master node of the cluster.

These configuration values are in the <cluster> element in the Node.xml file. For details, see Back-end Process Configuration.

```
<?xml version="1.0" encoding="UTF-8"?>
   <node-configuration version="11" format="untyped">
   ...
      <cluster>
         <active>no</active>
         <name>Episode Cluster</name>
         <backup>no</backup>
         <search>yes</search>
         <publish>yes</publish>
         <listen-port>40420</listen-port>
         <listen-interface>All</listen-interface>
         <listen-version>All</listen-version>
         <hosts>
            <host></host>
         </hosts>
         <dead-host-time>60000</dead-host-time>
         <stale-host-time>6000</stale-host-time>
      </cluster>
      ...
   </node-configuration>
   ...
```

To set up a cluster with the CLI, create a new cluster on the node you're using as master, with the command:

```
episodectl node create MyCluster
```

Now the configuration settings should look like this, and the node is ready to serve client requests:

```
<active>yes</active>
<name>MyCluster</name>
<backup>yes</backup>
<search>yes</search>
<publish>yes</publish>
<hosts>
   <host></host>
</hosts>
```

To determine what is published on Bonjour, execute `episodectl status clusters`.

To view the status of an individual node, execute `episodectl node info [address]` where address is the IP address or hostname of the node to contact (default: `local`).

To view the overall status of a cluster: Execute
`episodectl status nodes --cluster MyCluster`

or
`episodectl status nodes [address]`

where `address` is the IP or hostname of a node in the cluster.

If you want to join another node to the cluster, go to that computer and execute one of the following commands:

To use Bonjour to find the master, execute `episodectl node join MyCluster`

To specify the address to the master node, execute `episodectl node join --connect [address]` where address is the IP or hostname of the master node.

Use the configuration option `use-bonjour-IP-lookup` to control how IP addresses for Bonjour Episode nodes are resolved. If false (default), Episode expects the operating system to resolve the IP address using the hostname of the EpisodeNode found on Bonjour. If true, Episode resolves the IP address using the Bonjour service.

Setting `use-bonjour-IP-lookup` to true can resolve some connectivity issues, in particular ones where the user has restricted the EpisodeNode to only listen on specific network interfaces.

Avoiding Bonjour

When creating a cluster, execute the CLI episodectl command with these options:
`episodectl node create MyCluster --search no --publish no`
- or -
edit the configuration files manually to specify the Ethernet interface you want to use, and turn off Bonjour Lookup (see below.)

Then, restart the node using: `episodectl launch restart - node`.

When joining other nodes, add these options in the join command as well:
`episodectl node join --connect [master address] --search no --publish no`.

Note: In order to use the Episode graphic interface program on a node that does not employ Bonjour, the node has to be part of the cluster since you cannot connect by IP address.

Using a Specific Ethernet Interface

Enter the address of desired interface when joining nodes to the cluster. If you want the node to only accept incoming connections on a specific interface, you need to change the `<listen-interface>` setting in the Node.xml file. Since the node should always listen on the loopback interface too, that interface should be specified—separating them by a semicolon:

```
<?xml version="1.0" encoding="UTF-8"?>
<node-configuration version="11" format="untyped">
   ...
   <cluster>
      ...
      <listen-interface>lo0;en0</listen-interface>
      ...
   </cluster>
...
</node-configuration>
```

It is also possible to specify an IP address (which must be done on Windows):

```
<?xml version="1.0" encoding="UTF-8"?>
<node-configuration version="11" format="untyped">
   ...
   <cluster>
      ...
      <listen-interface>127.0.0.1;10.0.0.1</listen-interface>
      ...
   </cluster>
...
</node-configuration>
```

If the IOServer is used (instead of configuring a shared storage), you may do the same in its configuration file—IOServer.xml (see Back-end Process Configuration.)

Setting Bonjour IP Lookup to No

Finally, set the Bonjour IP lookup option to No, in the assistant.xml and node.xml files:

...
<use-bonjour-IP-lookup>no</use-bonjour-IP-lookup>
...

Shared Storage

If you are planning to use shared storage, you should configure the File Cache in the Episode GUI program, and also configure the `<resource-base-path>` in the Node.xml configuration file (see Back-end Process Configuration.) This cache path should be configured to point to the shared storage. Otherwise, Episode's IOServer will be used to access each nodes local file cache in a cluster.

```
<?xml version="1.0" encoding="UTF-8"?>
<node-configuration version="11" format="untyped">
   ...
   <node>
      ...
      <resource-base-path>/Path/to/Storage</resource-base-path>
      ...
   </node>
   ...
</node-configuration>
```

Due to the difference in how file resources are identified on Windows and MacOS file systems, it is not possible for Episode on Windows to identify a shared storage referenced in a MacOS manner as shared storage, and vice versa. If you want Episode to use shared storage between MacOS and Windows, you should use Named Storage instead.

Named Storage

The Named Storage feature allows you to define a storage location, such as a SAN, with a user-configurable name so that the same physical location can be used across Mac and Windows platforms even though the local path to that storage is different on each machine. Named Storage can be used within a cluster to permit access to files by multiple machines of either platform belonging to the cluster.

Named Storage is implemented using CLI commands, which means that you must have Episode Pro or Engine with API. For instructions in using the CLI, please refer to the *Episode Advanced User's Guide*. To access CLI help for instructions in using Named Storage, enter the following in the CLI:

```
Windows: episodectl ns --help
```

```
Mac: ./episodectl ns --help
```

Named Storage Simple Example

Windows Machine1 accesses a media location on a SAN using a windows path S:\

Mac Machine2 accesses the same location using a mac path /Volumes/MediaSAN/

In order for Episode to recognize both locations as the same physical storage, the CLI Named Storage feature must be used. You enter a CLI command on each machine that gives the physical location a name common to both machines. Then when that location is used, the system compares lists of named storage and matches them up so that the IO Server is not used and the files are moved directly from that storage. These are the commands you use for the two Windows and Mac example machines:

On Windows Machine1: `episodectl ns --add MediaSAN S:\`

On Mac Machine2: `./episodectl ns --add MediaSAN /Volumes/MediaSAN/`

Named Storage Cluster Example

You can also set up Named Storage to work with an Episode cluster, as this example illustrates. Adjust details shown in the example to fit your situation and network.

Note: Named storage must be defined on all machines before they join the cluster.

Starting Conditions

1. A network location is mounted on a Mac with the volume name "studioshares".
2. Note the folder level where the "root" of this mounted volume is located: smb://<servername>/<folder1>/<folder2>/studioshares/
3. Also note that once mounted, the path to this location on this machine is this: /Volumes/studioshares/

4. On Windows, you need to establish and note the full network path to this same location. In this example, “studioshares” is a shared folder on the server: \\<servername>\<folder1>\<folder2>\studioshares\

Named Storage Setup

1. On the Mac, define the named storage:

   ```
   ./episodectl node storage --add stgservices /Volumes/
   studioshares/stgservices/
   ```

2. On Windows, define the same named storage but use the full network path:

   ```
   episodectl node storage --add stgservices
   \\<server-name>\<folder1>\<folder2>\studioshares\stgser-
   vices\
   ```

Note: If there are any required user credentials for this server, add them as part of the path when defining the named storage:
`\\<user>:<password>@<servername>\<folder1>\<folder2>\studiosha res\stgservices\`

   The key detail to remember regarding the named storage defined path is that it must end in the same directory on all machines. In this case it’s “stgservices”.

3. Create the cluster.
4. Join or submit to cluster all client machines.

   The cluster should now be operational and the named storage accessible to all machines in the cluster.

Note: If you need to add new named storage to an existing cluster, you must take down the cluster first and ensure that all machines are working alone. Then you can add new named storage to each machine, create a new cluster, and join or submit to cluster all the machines that you want to include in the cluster.

Using the Command Line Interface

This chapter generally describes the Command Line Interface (CLI) for Episode.

The CLI is implemented on both Windows and MacOS; while use of the CLI is generally identical, accessing and running the CLI interpreter are different, and these differences are noted as appropriate.

Note: When utilizing the CLI to execute unlicensed features in demo mode, add the `-demo` flag. In the XML-RPC interface, you can add `-demo` to *submitSubmisssion* and *submitBuildSubmission* to use unlicensed features in demo mode as well.

Note: For license requirements, see XML-RPC and CLI License Requirements.

These topics are covered:

Notes: When executing a CLI command, be sure to supply the path to the command, and enclose it in double quotes to permit spaces in the path.

Be sure to provide a fully-qualified path to the *episodectl* command, and use quotes (Mac OS X) or double quotes (Windows) if there are spaces in the path.

For example, on Mac OS X, enter this path from the root (changed in Episode 6.5):
'/Applications/Episode.app/Contents/Resources/engine/bin/episodectl' launch start

On Windows, enter this path from the root:
"C:\Program Files\Telestream\Episode 6\bin\episodectl.exe" launch start

A folder is defined as a path ending with a path separator. On Windows, if you quote the string, you must either escape the backslash (\\) or use slash (/) as the last separator.

When using ! (exclamation) characters in bash arguments, they must be escaped, because bash parses the command before episodectl and will throw errors.

On Windows, you can only execute *episodectl launch* (and control the Episode system services) in the CLI if Windows UAC is disabled (turned off).

Starting the CLI Interpreter [Windows]

Before you can use the CLI interpreter or use the CLI in other ways on the Windows platform, the Client Proxy service must be running. Usually, you start all Episode services when your computer starts, even though you may not need them. By default, all Episode services are set to start up automatically when you install Episode. After installation, you should restart your computer to start all Episode services.

Based on your requirements, you can make sure your services are started by following these guidelines.

Starting Episode Services

The easiest way to start all Episode services is to start the Episode program:

Go to Start > All Programs > Telestream > Episode 6 > Episode 6.

When you start the Episode GUI program, all Episode services are started if they are not currently running. After starting Episode, you can stop the Episode GUI program if you choose; all Episode services remain running until explicitly stopped or the computer is shut down.

Note: Often, you'll keep Episode (the graphic user interface program) running so that you can use it to determine job status, refer to workflows, etc., as you interact with Episode via the CLI.

Other Alternatives

If your services are set to startup type Manual (or are not started), you can start them in the following ways:

- Start each Episode service manually in the Control panel
- Set each Episode service startup type to automatic in the Control panel
- Start each (or all) service using the CLI Launch command.

Starting Episode Control

Episode Control—the CLI Interpreter program—is installed by default in *C:\Program Files\Telestream\Episode 6\bin\episodectl.exe*.

If you installed Episode in another location, modify the commands below accordingly.

Note: This topic assumes you are familiar with the Command window and its features. If you're not familiar with the Command window features, read a Command window help document.

To start Episode Control, follow these steps:

Step 1 Click Start to display the Search Programs and Files text field. Enter `cmd` and press Enter to display the Command window.

Step 2 Navigate to the Episode bin folder, type the following, and press Enter:
`cd "C:\Program Files (x86)\Telestream\Episode 6\bin\"`
Quotes are necessary because of spaces in the path.

Step 3 To use the CLI, type *episodectl* along with your function and any arguments to execute the Episode command. For details, see Determining if Episode is Running.

Note: If your Episode services are not running, before proceeding, execute *episodectl launch start* with the proper arguments (see Return codes of processes in a UNIX-like environment do not display in the interpreter. To display the return code of the latest run process, enter *echo $? in Terminal.app.*).

Starting the CLI Interpreter [MacOS]

Before you can use the CLI interpreter or use the CLI in other ways, at least the Client Proxy services must be running. Usually, you'll start all Episode services, even though you may not need them. By default, all Episode services are set to startup type Automatic when you install Episode. After installation, you should restart your computer to start all Episode services.

Based on your requirements, you can make sure your services are started by following these guidelines.

Starting Episode Services

To start all Episode services, start the Episode application from the dock bar or go to Applications > Episode and double-click the Episode application.

When you start the Episode application, all Episode services are started, if they are not currently running. After starting Episode, you can stop Episode (the graphic user interface program) if you choose; all Episode services will remain running until explicitly stopped or the computer is shut down.

Note: Often, you'll keep Episode (the graphic user interface program) running so that you can use it to determine job status, refer to workflows, etc., as you interact with Episode via the CLI.

Other Alternatives

If your services are set to startup type Manual (or are not started), you can start them in the following ways:

- Start each Episode service manually in the Control panel
- Set each Episode service startup type to automatic in the Control panel
- Start each (or all) service using the CLI Launch command.

Starting Episode Control

Episode Control—the CLI Interpreter program—is installed in the Episode application bundle.

Note: This topic assumes you are familiar with Terminal and its features.

To start Episode Control, follow these steps:

Step 4 Open a Terminal window (Applications > Utilities > Terminal).

Step 5 Navigate to Episode's bin folder so you can execute the Episode Control program: */Applications/Episode.app/Contents/Resources/engine/bin/*.

Type the following command and press Enter:

```
cd /Applications/Episode.app/Contents/Resources/engine/bin
```

Step 6 In the bin folder, type the following with your function and any arguments to execute the Episode command: `./episodectl`

Note: If typing the full path is inconvenient you can add the directory to your PATH, or put a link to *episodectl* in one of the directories in your PATH.

Determining if Episode is Running

Before you submit jobs for encoding or to query an Episode node, make sure that Episode is running.

To determine that Episode is running on your local computer, execute one of these commands (for Windows, leave off the ./):

```
./episodectl launch list
./episodectl ll
```

In response, the system should display a list of the running Episode processes (on Windows, the PIDs are not shown):

EpisodeXMLRPCServer is running with PID 32420
EpisodeClientProxy is running with PID 32415
EpisodeAssistant is running with PID 32410
EpisodeIOServer is running with PID 32405
EpisodeNode is running with PID 32400

If Episode is not started, start it in one of two ways:

Start the Episode graphic user interface program

OR

In the CLI, execute one of these commands (for Windows leave off the ./):

```
./episodectl launch start
./episodectl ls
```

Using the CLI Interpreter

This topic describes generally how to interact with the CLI interpreter.

Executing Commands

To execute a command in Episode Control, execute Episode Control with the appropriate command and parameters. Make sure your command interpreter or terminal window is in the directory where Episode Control (Episodectl.exe) is located:

[Windows] *C:\Program Files\Telestream\Episode 6\bin*

[MacOS] */Applications/Episode.app/Contents/Resources/engine/bin/*

Enter the program name, followed by the command and parameters and press Enter to execute the command.

Note: In MacOS, precede the program name with ./ as in the following example:
`./episodectl node create --name HDCluster`

For Windows, the `./` should be left out.

Return Codes

Episode Control returns 0 when a command completes successfully, and returns 1 when most errors occur. When an error occurs, Episode Control returns an error message as well. Some commands return special return codes, which are described in the help page for the command.

Note: Return codes of processes in a UNIX-like environment do not display in the interpreter. To display the return code of the latest run process, enter *echo $?* in Terminal.app.

Displaying Episode Variables

To display the variables that can be set or read in conjunction with tasks, enter either of these two commands (for Windows, leave off the `./`):

```
./episodectl variables
./episodectl v
```

Displaying Episode Tags

To display the tags that can be used in conjunction with clusters, enter either of these two commands (for Windows, leave off the `./`):

```
./episodectl tags
./episodectl t
```

Executing Commands to a Cluster

The default target in the CLI is always the local node if nothing else is explicitly specified. You need to use `-c` with CLI commands when intended for cluster-wide execution— join, submit, watch-folder, status monitors, etc. Otherwise, the CLI will only execute the command in the local node.

Displaying CLI Help

To display help (man pages) in Episode Control, execute Episode Control with the command keyword `help`, or `whelp`. The `whelp` command displays the help text the full width of the console window. When displaying help on a command, you can specify the `-h` option. You can filter help contents by command or command and sub-command, as shown below.

Help Command Syntax

```
./episodectl help | whelp [<command>] | [<command>] [<sub command>] | all
```

Example (for Windows, leave off the `./`):

`./episodectl help all` returns the entire help set.

`./episodectl help watch-folder` returns the help text for the *watch-folder* command.

Writing Help to a Text File

To write help to a file, add `> <filename.txt>` to the command.

Example (for Windows, leave off the `./`):
`./episodectl help all > EpisodeCtl_Help.txt`

This command writes the entire help text to this text file: `EpisodeCtl_Help.txt`.

Episode 6.5 CLI Listing

The following pages include a complete listing of the CLI Help output. See Writing Help to a Text File on the previous page for details about how to output the Help to a text file.

TELESTREAM EPISODE COMMAND LINE INTERFACE

DESCRIPTION

The episodectl.exe program returns 0 on success and 1 for general errors. If 1 is returned an error message is also printed. Some commands have special return codes which are stated in their description.

SYNOPSIS

episodectl.exe help [<command>]
episodectl.exe help [<command>] [<sub command>]

If you want to get all help text you can write

episodectl.exe help all

You can also use whelp instead of help to get the description texts printed in the full width of the console.

episodectl.exe whelp [<command>] | [<command>] [<sub command>] | all

You can also use the -h option like this:

episodectl.exe [<command>] -h
episodectl.exe [<command>] [<sub command>] -h

episodectl.exe --version [<product>|<api>]

Print version information. If the optional argument product> or api> is specified, only the relevant version number is printed, without newline, suitable for program/script string comparison.

episodectl.exe variables
episodectl.exe v

An overview of "variables" that can be set and/or read to/from tasks.

episodectl.exe tags
episodectl.exe t

An overview of the "tags" concept and how to use them in a Cluster.

episodectl.exe priority

An explanation of how priority works in Episode.

episodectl.exe examples
episodectl.exe e

Introduction to the command line interface, aswell as a getting started guide and examples. This section is always printed wide.

episodectl.exe launch

launch start (ls)
launch restart (lr)
launch list (ll)
launch stop (lp)
launch install
launch uninstall

episodectl.exe node

node info (ni)
node cache (ncache)
node jobs (njobs)
node create (nc)
node join (nj)
node privatize (np)
node tag (nt)
node storage (ns)
node history (nh)
node log (nl)

episodectl.exe ioserver

ioserver add (ioa)
ioserver list (iols)
ioserver remove (ior)
ioserver log (iol)

episodectl.exe proxy

proxy defaults (pd)
proxy storage (ps)
proxy history (ph)
proxy log (pl)

episodectl.exe task

task transfer (tt, tu)
task youtube (ty)
task execute (tx)
task mail (tmail)
task mbr (tmbr)

task set (tset)

episodectl.exe source

source filelist (sfl)
source monitor (smon) deprecated

```
        source watch-folder(swf)
        source edl(sedl)
        source iseq(siseq)

    episodectl.exe workflow

        workflow submit (ws)
        workflow stop   (wp)
        workflow recall (wr)

    episodectl.exe status

        status clusters  (sc)
        status nodes     (sn)
        status workflows (sw, sws)
        status tasks     (st, sts)
        status monitors  (sm) Deprecated.
        status watch-folders  (swfs)

    episodectl.exe job

        job cancel       (jcan)
        job requeue      (jrq)
        job pause        (jpau)
        job resume       (jres)
        job set-priority (jprio)

    episodectl.exe monitor

        monitor start        (ms) Deprecated
        monitor list         (ml) Deprecated.
        monitor set-priority (mprio) Deprecated
        monitor stop         (mp) Deprecated
        monitor remove       (mr) Deprecated
        monitor log          (mg) Deprecated

    episodectl.exe watch-folder

        watch-folder start        (wfs)
        watch-folder list         (wfl) Deprecated
        watch-folder set-priority (wfprio)
        watch-folder stop         (wfp)
        watch-folder remove       (wfr)
        watch-folder log          (wfg)

    episodectl.exe util

        util analyze (ua, analyze)
```

COMMANDS AND SUB COMMANDS

launch

The launch command starts Episode services, stops Episode services and lists running Episode processes.

launch start [-n] [-i] [-a] [-c] [-x] [-j]

Start Episode process(es). Processes already running will not be restarted. If no process is specifically specified with an option, all processes will be started.

This command can also be specified as ls, i.e. episodectl.exe ls

-n
--node Start EpisodeNode.exe

-i
--ioserver Start EpisodeIOServer.exe

-a
--assistant Start EpisodeAssistant.exe

-c
--clientproxy Start EpisodeClientProxy.exe

-x
--xmlrpc Start EpisodeXMLRPCServer

-j
--jsonrpc Start EpisodeJSONRPCServer

launch restart [-n] [-i] [-a] [-c] [-x] [-j]

Restart Episode process(es). Processes that are not running will be started. If no process is specifically specified with an option, all processes will be restarted or started.

This command can also be specified as lr, i.e. episodectl.exe lr

-n
--node Restart EpisodeNode.exe

-i
--ioserver Restart EpisodeIOServer.exe

-a
--assistant Restart EpisodeAssistant.exe

-c
--clientproxy Restart EpisodeClientProxy.exe

-x
--xmlrpc Restart EpisodeXMLRPCServer

-j
--jsonrpc Restart EpisodeJSONRPCServer

launch list

List Episode processes.

This command can also be specified as ll, i.e. episodectl.exe ll

launch stop [-n] [-i] [-a] [-c] [-x] [-j]

Stop Episode process(es). If no process is specifically specified with an option, all processes will be stopped.

This command can also be specified as lp, i.e. episodectl.exe lp

-n
--node Stop EpisodeNode.exe

-i
--ioserver Stop EpisodeIOServer.exe

-a
--assistant Stop EpisodeAssistant.exe

-c
--clientproxy Stop EpisodeClientProxy.exe

-x
--xmlrpc Stop EpisodeXMLRPCServer

-j
--jsonrpc Stop EpisodeJSONRPCServer

node

With the node sub command, you can manage your local Node and get info about remote Nodes.

node info [<hostname/IP>]

Display some hardware info, licenses info, current status, and info about what is currently connected to this Node. Default host address is 127.0.0.1.

This command can also be specified as ni, i.e. episodectl.exe ni

NOTE: This will only work if the target node is in "Public" mode, i.e. a participant or master of a cluster. It will always work for the node on localhost regardless of mode.

node cache [--set <base path> [<sub dir>]]
[--set-default|--reset]
[--list]
[--clear]

Manage the File Cache of this node. The File Cache is used to by the tasks in the workflow, such as temporary files, encoded output files, or localized files.

It consists of two parts, a <base path> that must exist (<resource-base-path> in the configuration file) and a dynamically created <sub dir> sub directory (<resource-append-path> in confiuration file). There will be one more sub directory added to form the complete path, one for private mode called PrivateCache and one for cluster mode (public mode) called ClusterCache. Each workflow will have its own sub directory which will be deleted when a workflow is done.

The default location is on the local machine which means that the IOServer must be used in a cluster setup to access files on the different nodes in the cluster. If a shared storage is used, the File Cache should be configured to point to the shared storage, for example on Windows:
episodectl.exe node cache S:\ Episode
or OS X:
episodectl /Volumes/Storage/ Episode

If no command option is specified, the current location will be printed.

This command can also be specified as ncache, i.e. episodectl.exe ncache

--set Set the path to a new File Cache directory. This must be a local path. UNC paths are supported on Windows. The previous configured cache directory will be cleared.

--reset
--set-default Set the default paths.

--list List current contents in the cache. This should be empty. In cases when it's not empty and the node isn't currently working, the node was probably stopped while working or left a cluster while working.

--clear Clear the cache. Note: Do not do this while workflows are being processed on this machine

node jobs [--set <number of jobs>]
[--set-recommended [cpu|mem]]
[--run-master yes|no]
[--scheduling HB|LB|RR]
[--os-prio normal|low]
[--verify-io yes|no]
[--cluster-wide yes|no]

```
        [--retry <num>]
        [--encoding-retry <num>]


       Configure job related settings for this node, or a whole cluster in case of
       --verify-io and/or --os-prio. If no option is specified, the currently configured
       settings are displayed as well as recommendations for number of simultanious jobs.

       This command can also be specified as njobs, i.e. episodectl.exe njobs

        --set  Set number of simultaneous encoding jobs to <number of jobs>.

--set-recommended  Set a recommended number of jobs based on the hardware of this computer. If
                   the optional argument "cpu" is specified, a recommended value for CPU
                   intensive jobs is set. CPU intensive jobs are in this case jobs that are
                   predicted to use little memory, for example due to low resolution. The
                   optional argument "mem" is for if jobs are predicted to use a lot of memory,
                   for example HD material. If no argument is specified, a recommended value
                   for general cases is set.

  --run-master  If this node should run any jobs if/when it is a cluster master node. If
                there are more than 1 other nodes (worker/slave nodes) in the cluster, it is
                a good idea to consider turning the job running off on the master node. Note:
                Only encoding nodes require a license (i.e. nodes that have 1 or more
                encoding job slots configured). This means that you may put in a extra
                machine that is unlicensed to be the master node, and which is not running
                any jobs.

  --scheduling  Which scheduling (or job distribution) algorithm to use if/when this node is
                a cluster master node. HB is short for "Hardware Balanced", LB is short for
                "Load Balanced", and RR is short for "Round Robin". All three algorithms
                will take th number of configured job slots (available and total) into
                account and where Round Robin only takes that into account. Hardware
                Balanced will also take the individual cluster participants' hardware
                specifications into account. Finally, in addition to that, Load balanced
                will also look at the reported CPU and Memory load of each machine. The
                default interval for the nodes' "current load" (used in Load Balancing)
                reporting is 5 seconds. You can monitor that visually with this command episodectl.exe
                status nodes -c MyClusterName -w.

    --os-prio  If the jobs (EpisodeWorker processes) should be set with Normal or Low OS
                process priority. If you have a cluster where the participants may be
                computers that are performing other tasks or a person is currently working
                on, it is recommended to set this to Low. If this is the master node in a
                cluster and is configured to do encodings, it is highly recommended to set
                this to Low

   --verify-io  Advanced configuration. Perform IO verification in the job (EpisodeWorker).
                This is a security mechanism that, if a URL that is used in a job (for
                example file transfer) is a local URL/file (or resolves into a local
                URL/file), the local IOServer will be asked if this file was actually shared
                and thus belonged to a validly submitted job.

 --cluster-wide  If options --verify-io and --os-prio should be cluster-wide or configured
                locally on each node. If they are configured to be cluster-wide (on the
                master node), the master node will tell each job its own (the master's)
                confiuration instead of the local node's configuration, i.e. the node that
```

is actually running the task.

--retry How many times to retry failed non-encoding jobs. Default is 2.

--encoding-retry How many times to retry failed encoding jobs. Default is 0.

node create [-n] [<cluster name>]
[--search yes|no]
[--publish yes|no]

Create a public cluster. If you don't specify a name, the name previously configured will be used. If you haven't configured anything before, this is the default cluster name "Episode Cluster". If you specify an already existing cluster name and use Bonjour, one of the master nodes will re-join the cluster as a non-master.

This command can also be specified as nc, i.e. episodectl.exe nc

-n
--name Explicitly specify that this argument is the cluster name.

--search Use Bonjour to search for, and contact other nodes that belongs to the cluster with the same name. The default is "yes".

--publish Publish this node on Bonjour; making it visible to other nodes and clients. If Bonjour publication is turned off, other nodes that should participate in this cluster must specify the hostname/IP (with the --connect option) of this node to be able to join. Clients (such as episodectl) that want to submit jobs to this node or view status etc. must also specify this host's hostname or IP address with the --host option. The default is "yes".

node join [-n] [<cluster name>]
[--search yes|no]
[--publish yes|no]
[--connect <hostname/IP>]

Join a public cluster. If you don't specify a name, the name previously configured will be used. If you haven't configured anything before, this is the default cluster name "Episode Cluster". If joining by cluster name (as opposed to joining by address with option --connect), any previously configured host addresses will be cleared.

This command can also be specified as nj, i.e. episodectl.exe nj

-n
--name Explicitly specify that this argument is the cluster name.

--search Use Bonjour to search for, and contact other nodes that belongs to the cluster with the same name. The default is "yes".

--publish Publish this node on Bonjour; making it visible to other nodes and clients.

--connect Specify a hostname or an IP address to a node in a cluster. The IP address

may be directly to the master (recommended) or to a participant which will re-direct this node to the master node. When specifying a IP address, the cluster name will be inherited from the node you connect to, and thus the name will be ignored if you specify one.

node privatize [--publish yes|no]

Leave a cluster and make the node "private".

This command can also be specified as np, i.e. episodectl.exe np

--publish Publish this node on Bonjour; making it visible to other nodes and clients. NOTE: When a node is in private mode, no other client or node can connect it.

node tag [--add <tag> ...]
[--clear]
[--set-default|--reset]

Manage Tags on this node. If no option is specified, the currently set tags are shown as a space separated list. See tag section (episodectl.exe tags) for further information.

This command can also be specified as nt, i.e. episodectl.exe nt

--add Add one or more tags to the local node.

--clear Clear tags. This will remove all tags, including the default one.

--reset
--set-default Clear all user defined tags and set the default one which is a platform tag that is "Mac" or "Win".

node storage [--add <name> <path or url>]
[--remove <name>]
[--clear]

Manage named storages on this node. If no option is specified, the currently configured storages are shown. Named storages are useful if you have a mixed cluster of Windows and OS X machines, or if you submit jobs from clients that access a storage through a different path or URL than the submission target machine, either because of different platform or otherwise different mount point etc. If the access through a storage can not be resolved on a particluar machine (because it wasn't configured there), the Episode IOServer will be used for access instead. NOTE: If you configure storages on your machines, you must NOT use the --no-resource option when submitting jobs because that will disable the storage identification/resolving possibilities.

This command can also be specified as ns, i.e. episodectl.exe ns

--add Add one storage where <name> should preferrably be a unique (nick-)name for a particular storage on your entire network where Episode is used (see advanced note) and the same name must be used for the same storage on all machines where the storage is configured. <name> must not contain these characters '[]{}'.

The <path or url> should be a path or URL for how to reach a specific point on the storage from this machine, for example, if you have a Windows machine that accesses a storage via a UNC path \\server\share\share-object\ and you have a OS X machine that have access to the same share-point (or share-object) via the URL smb://server/share/share-object/ you should specify episodectl.exe node storage --add MyShareName \\server\share\share-object\ on the Windows machine and specify episodectl node storage --add MyShareName smb://server/share/share-object/ on the OS X machine. The important thing is that the relative paths, or sub directory structure inside the share-point is the same for all machines. Let's say that you have mounted that storage on the OS X machine in /Volumes/Storage/ (where "Storage" points to exactly the same point/directory as "share-object" in previous example) you should specify episodectl node storage --add MyShareName /Volumes/Storage/ on the OS X machine. Likewise, if you have mounted that share-point on Windows in S:\ you should specify episodectl.exe node storage --add MyShareName S:\ on the Windows machine.

NOTE: The named storage will be added to both the local Node and to the ClientProxy configuration. This is to enable both the Node to be a participant of a cluster where this storage is needed, and to enable this machine to be a pure client to one or more clusters where this storage is needed.

ADVANCED NOTE: If you have 2 or more different clusters where each cluster have different storages or different share-points on one storage, you should NOT use the same <name> for the storages/share-points beacuse that will disable the possibility for both clients and nodes (with the configured storage <name>) to "interact" with both/all clusters.

--remove Remove the storage named <name>.

--clear Clear/remove storage.

node history [--clear]
[--set-keep-time <number of days>]

Clear current history from database or re-configure the time to keep history.

This command can also be specified as nh, i.e. episodectl.exe nh

--clear Clear the current history from database.

--set-keep-time Set the number of days to keep history.

```
node log [--set-default|--reset]
        [--set-debug|--full]
        [--syslog yes|no [<level>]]
        [--file yes|no [<level>]] [-n <max-files>] [-s <max-size>]]
        [--tasks yes|no [<level>] [-n <max-files>]]
        [--watch-folders yes|no [<level>]] [-n <max-files>] [-s <max-size>]]


        Set logging settings for the Node, Tasks and Watch folders. The task and watch
        folder settings are "cluster-wide", meaning that only the master Node's
        configuration of these matters and it will log all tasks and watch folders
        throughout a cluster on the machine of the master Node. If no option is specified,
        the currently configured logging settings are shown. The <level> argument/parameter to
        some of the options should be a digit/number in the range 0..7 where 0 = Fatal, 1 =
        Alert, 2 = Critical, 3 = Error, 4 = Warning, 5 = Notice, 6 = Info, 7 = Debug.

        This command can also be specified as nl, i.e. episodectl.exe nl

 --set-default
     --reset  Set default logging settings, i.e. the settings for a fresh install. The
              default is to only log to ASL (Apple System Log) on OS X and Event Log on
              Windows with a verbosity of 5 (Notice).

  --set-debug
      --full  Set full debug logging settings. This is a significant overhead, only use
              this to try and solve problems.

    --syslog  If the Node should log to the system log and optionally a log verbosity
              level.

     --file  If the Node should log to a file and optionally a log verbosity level. The
              log files will be rotated when <max-size> is reached (configurable with sub
              option -s) and up to <max-files> will be created/rotated (configurable with
              sub option -n).

     --tasks  If the Tasks should log to file and optionally a log verbosity level. The
              sub option -n <max-files> configures how many Task files to save on disk
              before starting to clean older ones.

--watch-folders  If watch folders should log to file and optionally a log verbosity level.
              Each monitor will have its own log files that work just like for the Node,
              see option --file.
```

```
ioserver

With the ioserver sub command, you can manage shares currently shared in the local IOServer
and configure logging settings for the IOServer.


ioserver add <file or directory> ...
          [-i <id>]
```

[-e <hours>]

Add a share to the local IOServer. If no ID is specified, the CLI's ID will be used.

This command can also be specified as ioa, i.e. episodectl.exe ioa

-i
--id An ID for the share.

-e
--expire Optional automatic expiry time for the share specified in hours from current time. The default is the value 0 which means it will never expire, thus having to be removed manually.

ioserver remove -i <id>|--all

Remove a share from the local IOServer. If no ID is specified, the CLI's ID will be used, which will remove all shares added through the CLI.

This command can also be specified as ior, i.e. episodectl.exe ior

-i
--id An ID for the share to remove

--all Remove all shares

ioserver list

List shares on the local IOServer.

This command can also be specified as iols, i.e. episodectl.exe iols

ioserver log [--set-default|--reset]
[--set-debug|--full]
[--syslog yes|no [<level>]]
[--file yes|no [<level>]] [-n <max-files>] [-s <max-size>]]

Set logging settings for the IOServer. If no option is specified, the currently configured logging settings are shown. The <level> argument/parameter to some of the options should be a digit/number in the range 0..7 where 0 = Fatal, 1 = Alert, 2 = Critical, 3 = Error, 4 = Warning, 5 = Notice, 6 = Info, 7 = Debug.

This command can also be specified as iol, i.e. episodectl.exe iol

--set-default
--reset Set default logging settings, i.e. the settings for a fresh install. The default is to only log to ASL (Apple System Log) on OS X and Event Log on

Windows with a verbosity of 5 (Notice).

--set-debug
--full Set full debug logging settings. This is a significant overhead, only use this to try and solve problems.

--syslog If the IOServer should log to the system log and optionally a log verbosity level.

--file If the IOServer should log to a file and optionally a log verbosity level. The log files will be rotated when <max-size> is reached (configurable with sub option -s) and up to <max-files> will be created/rotated (configurable with sub option -n).

proxy

With the proxy sub command, you can manage your local ClientProxy.

proxy defaults [-d <path or url>]
[-c <cluster name>]
[--host <hostname or IP>]
[--guess yes|no]

The CLI uses the EpisodeClientProxy.exe process to do most things, and most importantly, use it to submit jobs and get status from. By configuring the Client Proxy with default values, you can change the behavior of your job submissions and omit certain options to episodectl.exe workflow submit. If no option is given, the currently configured defaults are shown.

This command can also be specified as pd, i.e. episodectl.exe pd

-d
--destination-dir Set a default destination path or URL. This will be used if option -d is omitted for the command episodectl.exe workflow submit. Default is the user's Desktop.

-c
--cluster Set a default target cluster name to submit to or check status on. This will be used if option -c is omitted for the commands episodectl.exe workflow ..., episodectl.exe status ..., and episodectl.exe watch folder .... This has precedence over a specified host. This may be an empty string, which is also default.

--host Set a default target host (hostname or IP address) to submit to. This will be used if option --host and -c are omitted for the commands episodectl.exe workflow ..., episodectl.exe status ..., and episodectl.exe watch folder .... Default is "127.0.0.1".

--guess "Guess if Shared Storage". This is a workflow configuration that tells the

tasks to "guess" if a file/directory is located on a shared storage. It bases its guess on if it has "non-native local access", i.e. if it has local access and the filesystem is non-native, or mounted. If it is, it is treated like a shared storage. This guess will be wrong if the computers in a cluster has (path wise) identical storages mounted on them that are not the same stoarage, for example Firewire/USB drives mounted at the same path with the same folder structure etc. Default is "yes".

proxy storage [--add <name> <path or url>]
[--remove <name>]
[--clear]

Manage named storages on this machine. If no option is specified, the currently configured storages are shown. Named storages are useful if you have a mixed cluster of Windows and OS X machines, or if you submit jobs from clients that access a storage through a different path or URL than the submission target machine, either because of different platform or otherwise different mount points etc. If the access through a storage can not be resolved on a particluar machine (because it wasen't configured there), the Episode IOServer will be used for access instead. NOTE: If you configure storages on your machines, you must NOT use the --no-resource option when submitting jobs because that will disable the storage identification/resolving possibilities.

This command can also be specified as ps, i.e. episodectl.exe ps

This command will do the exact same thing as episodectl.exe node storage so please see description of that command or use that command.

proxy history [--clear]
[--set-keep-time <number of hours>]

Clear current history from memory or re-configure the time to keep history in process memory. If you are submitting jobs to multiple target clusters, each history for each connection will be affected.

This command can also be specified as ph, i.e. episodectl.exe ph

--clear Clear the current history from memory (process memory of ClientProxy, not the Node).

--set-keep-time Set the number of hours to keep history. Default is 1 hour.

proxy log [--set-default|--reset]
[--set-debug|--full]
[--syslog yes|no [<level>]]
[--file yes|no [<level>]] [-n <max-files>] [-s <max-size>]]

Set logging settings for the ClientProxy. If no option is specified, the currently

configured logging settings are shown. The <level> argument/parameter to some of the options should be a digit/number in the range 0..7 where 0 = Fatal, 1 = Alert, 2 = Critical, 3 = Error, 4 = Warning, 5 = Notice, 6 = Info, 7 = Debug.

This command can also be specified as pl, i.e. episodectl.exe pl

--set-default
--reset Set default logging settings, i.e. the settings for a fresh install. The default is to only log to ASL (Apple System Log) on OS X and Event Log on Windows with a verbosity of 5 (Notice).

--set-debug
--full Set full debug logging settings. This is a significant overhead, only use this to try and solve problems.

--syslog If the ClientProxy should log to the system log and optionally a log verbosity level.

--file If the ClientProxy should log to a file and optionally a log verbosity level. The log files will be rotated when <max-size> is reached (configurable with sub option -s) and up to <max-files> will be created/rotated (configurable with sub option -n).

task

With the task sub command, you can create .epitask configuration files. The created file will be written in the current working directory unless --out <directory> is specified.

task transfer <url or path>
[--name <name>]
[--increment-filename yes|no]
[--try-link yes|no]
[--try-rename yes|no]
[--dest-filename <naming convention>]
[--dest-sub-dirs <name> ...]
[--dest-sub-dirs-ext <naming convention> ...]
[--re-create-source-sub-dirs yes|no]
[--post-dest-sub-dirs-ext <naming convention> ...]
[--post-dest-sub-dirs <name> ...]
[--priority <priority>]
[--tag <tag> ...]
[--inverse-tag <tag> ...]
[--format xml|simple-xml|ascii|bin]
[-o <directory>]
[--print-plain-path]

Create a Transfer .epitask configuration file where <url or path> is the destination directory for files. This directory must exist. <url or path> could also be the (case insensitive) keyword SAS which will configure the destination to "Same As Source".

The task of the Transfer is to transfer the encoded file from the Episode "Cache" to its final destination (this may of course just as well be a "download" depending on which computer it runs on and where its final destination is). This can be done in three (main) ways.

First way is to link the file (hard-link) to the destination. Notice that the cache directory will be removed when the workflow is done. This is the most effective way, as the file system only makes a new reference to the actual file data at the destination, in other words, it's the exact same physical data that once was wrtten to the cache. This is also preferable if more than one Deployment tasks are used because it is independent of in which order the tasks are executed, the file exists at the two locations at the same time until the whole workflow is done. Notice also that it's called "try-link", if linking fails due to that the target is not on the same physical device for example, a regular copy will be made instead. IF you're using a file system that doesn't support hard-links, renaming (or moving) the file is the second most effective way.

Renaming/Moving a file can also only be done on the same logical device (file system). It is implementation specific what actually is being done in case of different devices. A operating system copy could be made in which case no progress will be reported by the Transfer task and the job could time out. If more than one Deployment tasks are specified, for example one FTP and one local destination, the local Deployment could move the file (if executed first) so that the FTP Transfer task will have nothing to upload... It is therefore only recommended to use this if: You have a file system that does not support hard-links, you are only using one Deployment task, and you have your Episode File Cache (resource-base-path in Node.xml) and destination directory on the same logical device (file system).

The third way is of course to copy the file to its destination.

The default configuration is try-link=yes, try-rename=no. try-link has precedence over try-rename.

Output Directory creation

There are a number of directory creation options for the output. One option is to re-create a directory structure that "came" from (was set by) the source. The only source that currently sets a directory structure (sub directories) is a watch folder that is configured with recursive listing, i.e. a watch folder that looks for files in sub directories.
This is option --re-create-source-sub-dirs.

Then there are 2 versions of directory lists to create. The first version takes exactly one string of text per space separated list entry, a static text or one variable.
That is the --dest-sub-dirs option.

The other version takes a list of text strings, where each space separated list entry can contain both static texts and multiple variables.
That is the --dest-sub-dirs-ext option.

Finally, there are "post-" versions of the previous 2 options, where "post" means post source directory re-creation. Here is a example and its result:

A Watch Folder is set up to monitor a directory C:\Episode\ which contains a directory "MonitoredContent". The watch fpöder is created with the command:

episodectl.exe source watch-folder C:\Episode\ --recursive 1

Then we create a Transfer task with the following command:

episodectl.exe task transfer C:\EpisodeOutput\ --dest-sub-dirs '$dynamic.year.YYYY$' Project1 --dest-sub-dirs-ext '$dynamic.month.MM$-$dynamic.day.DD$' --re-create-source-sub-dirs yes

The workflow is set up with command:

episodectl.exe workflow submit --watch-folder Episode.episource -e H264.epitask -d EpisodeOutput.epitask

A file "SourceFile.mov" is copied into the "MonitoredContent" directory, which will produce the following output path

C:\EpisodeOutput\2012\Project1\03-12\MonitoredContent\SourceFile-H264.mov

If the post-versions are used instead, i.e. --post-dest-sub-dirs-ext and --post-dest-sub-dirs , the following path is created

C:\EpisodeOutput\MonitoredContent\03-12\2012\Project1\SourceFile-H264.mov

The order of directory creation is the same as the order the options are listed in the synopsis section above.

This command can also be specified as tu, i.e. episodectl.exe tu

--name A name for the task.

--increment-filename If "incremental filename" should be applied. This will make a listing of the destination directory and see if there are files named the same, and if there is, it will add " (n)" (where 'n' is one number higher than found) to the outfile. This MAY be a slight loss in performance depending on circumstances. Default is "yes".

--try-link If the task should try to link the file to its destination. Default is "yes".

--try-rename If the task should try to rename/move the file to its destination. Default is "no".

--naming
--dest-filename Insert a custom naming convention. The naming convention is specified as one string and you can insert variables. See variable section (episodectl.exe variables) for a list of the variables.

--dest-sub-dirs See Output Directory creation section above.

--dest-sub-dirs-ext See Output Directory creation section above.

--re-create-source-sub-dirs If the task should re-create a directory structure that "came" from (was set by) the source. The only source that currently sets a directory structure (sub directories) is a watch folder that is configured with recursive

listing, i.e. a watch folder that looks for files in sub directories. See Output Directory creation section above. The default is "no".

--post-dest-sub-dirs-ext See Output Directory creation section above.

--post-dest-sub-dirs See Output Directory creation section above.

--priority Set the priority of the task, default is 0. See episodectl.exe priority for more information.

--tag Set one or more tags on this task. See tag section (episodectl.exe tags) for further information.

--inverse-tag Set one or more inverse-tags on this task. See tag section (episodectl.exe tags) for further information.

--format Format of the ouput configuration file. The argument can be any of xml, simple-xml , ascii, bin. The default is xml which is an XML format with type information. There is a simple-xml format that is more readable and easier to use for manual editing but use this with caution because it may not be upgradable/usable when a new version of Episode is released. ascii is a more compact format and more efficient to work with (IO, storage etc) and still kind of readble but not recommended for editing. bin is a binary format and about the same size as ascii but not readble nor editable. ascii and bin are therefore only recommended for "automatic" use, i.e. from a program that both creates them and submits them.

-o
--out Specify the directory where the output file should be written. Default is current working directory.

--print-plain-path Print the path to the written configuration file without any other text or newline etc. Usable if a program/script should run this command and read back the path.

task youtube -u <username>
-p <password>
-t <title>
-d <description>
-c <category>
-k <keyword> ...
[--name <name>]
[--priority <priority>]
[--tag <tag> ...]
[--inverse-tag <tag> ...]
[--format xml|simple-xml|ascii|bin]
[-o <directory>]
[--print-plain-path]

Create a YouTube .epitask configuration file. Most options are mandatory.

The YouTube task will deploy the encoded file to YouTube.

This command can also be specified as ty, i.e. episodectl.exe ty

```
        -u
  --username  Username for YouTube account. Mandatory option.

        -p
  --password  Password for YouTube account. Mandatory option.

        -t
   --title  Title for uploaded video. This text could contain dollar variables, for
            example 'My encoded $source.filename$'. See episodectl.exe variables for
            more information. Mandatory option.

        -d
--description  Description for uploaded video. This text could contain dollar variables,
            for example 'My encoded $source.filename$'. See episodectl.exe variables for
            more information. Mandatory option.

        -c
  --category  Category for uploaded video. Mandatory option.
            Valid categories are: People  Film  Autos  Music  Animals  Sports  Travel
            Games  Comedy  News  Entertainment  Education  Howto  Nonprofit  Tech

        -k
  --keywords  One or more space separated keywords. Mandatory option.

    --name  A name for the task. The recipient's address will be used by default.

  --priority  Set the priority of the task, default is 0. See episodectl.exe priority for
            more information.

     --tag  Set one or more tags on this task. See tag section (episodectl.exe tags) for
            further information.

--inverse-tag  Set one or more inverse-tags on this task. See tag section (episodectl.exe
            tags) for further information.

   --format  Format of the ouput configuration file. The argument can be any of xml, simple-xml
            , ascii, bin. The default is xml which is an XML format with type
            information. There is a simple-xml format that is more readable and easier
            to use for manual editing but use this with caution because it may not be
            upgradable/usable when a new version of Episode is released. ascii is a more
            compact format and more efficient to work with (IO, storage etc) and still
            kind of readble but not recommended for editing. bin is a binary format and
            about the same size as ascii but not readble nor editable. ascii and bin are
            therefore only recommended for "automatic" use, i.e. from a program that
            both creates them and submits them.

        -o
    --out  Specify the directory where the output file should be written. Default is
            current working directory.

--print-plain-path  Print the path to the written configuration file without any other text or
            newline etc. Usable if a program/script should run this command and read
            back the path.
```

```
task execute <path>
        [--name <name>]
        [--content yes|no]
        [--args <arg> ...]
        [--env <name> <value> ... ...]
        [--parse-progress yes|no]
        [--progress-format <regexp>]
        [--progress-type percent|fraction]
        [--priority <priority>]
        [--tag <tag> ...]
        [--inverse-tag <tag> ...]
        [--format xml|simple-xml|ascii|bin]
        [-o <directory>]
        [--print-plain-path]


    Create a Execute .epitask configuration file where <path> is the path to a
    program/script.

    The Execute task can execute a program/script in two ways.

     - Execute a executable file referenced by a path.
     - Read the content of a script and later, when run, create a tempfile with the
    script code in the Episode "Cache" and execute that file.

    Executing a path is both platform dependent and may not be very suitable for
    distribution in a cluster. The execution (task scheduling/distribution) is very
    easily configurable through Tags (see episodectl.exe tags) so this may not be an
    issue. Reading in the script content is at least easily distributable but is
    probably still platform dependent and could consume memory and disk space etc if
    it's a large program.

    If the content is read, the file extension of the specified file is also saved and
    will be set on the file before execution in case of that the OS is basing its choice
    of interpreter by file extension.

    There is one environment variable that is always set by this task (in excess of
    variables configured with option --env), and that is the variable EPISODECTL in
    which the absolute path to the CLI executable file (episodectl.exe) is set.
    See episodectl.exe examples section about this task for more information.

    This command can also be specified as tx, i.e. episodectl.exe tx


    --name  A name for the task. A suitable name is generated if not specified.

  --priority  Set the priority of the task, default is 0. See episodectl.exe priority for
          more information.

   --content  If the content (code) of the program/script should be read and distributed.
          See description above. The default is yes.

    --args  The command line options and arguments that should be passed to the executed
          program. If you want to specify options to your program (and running this
          command from a command prompt), you may have to double-escape them like this
          episodectl.exe tx --args '"-x"' or the other way around. The parser for this
```

option will remove both single and double quotation marks.

--env The environment variables to set for the executed program. The arguments should be a space separated list of <name> <value> <name> <value> etc. See episodectl.exe variables for further info.

--parse-progress If the Execute task should read progress (by parsing stdout) from the executed program. This is mostly suitable for long running programs, or programs that makes a Deployment or file copying etc. Default is no.

--progress-format A regular expression to use to identify progress. The default regular expression looks like this: progress\[([\d\.]+)\] where you can see that it has a "capturing" definition (the parentheses) for digits '\d' and dots '\.' which must be present if you define your own. (You can leave dots out if you are using percent output, i.e. 0..100). If the default regexp is used, the executed program should output "progress[30]" or something like "progress[0.30000]" for reporting a progress of 30%, depending on which --progress-type is chosen. NOTE: Be sure to flush stdout for a successful feedback.

--progress-type If you want to output a progress value in percent between 0 and 100 as a integer value, or if you want to output a fraction value between 0.0 and 1.0.

--tag Set one or more tags on this task. See tag section (episodectl.exe tags) for further information.

--inverse-tag Set one or more inverse-tags on this task. See tag section (episodectl.exe tags) for further information.

--format Format of the ouput configuration file. The argument can be any of xml, simple-xml , ascii, bin. The default is xml which is an XML format with type information. There is a simple-xml format that is more readable and easier to use for manual editing but use this with caution because it may not be upgradable/usable when a new version of Episode is released. ascii is a more compact format and more efficient to work with (IO, storage etc) and still kind of readble but not recommended for editing. bin is a binary format and about the same size as ascii but not readble nor editable. ascii and bin are therefore only recommended for "automatic" use, i.e. from a program that both creates them and submits them.

-o
--out Specify the directory where the output file should be written. Default is current working directory.

--print-plain-path Print the path to the written configuration file without any other text or newline etc. Usable if a program/script should run this command and read back the path.

task mail [-u <username> [-p <password>]]
-s <server> [--port <port>]
-f <address>
-t <address> [-c <address>] [-b <address>]
-j <subject>
-m <message>
[--ssl yes|no]

```
    [--name <name>]
    [--priority <priority>]
    [--tag <tag> ...]
    [--inverse-tag <tag> ...]
    [--format xml|simple-xml|ascii|bin]
    [-o <directory>]
    [--print-plain-path]


   Create a Mail .epitask configuration file. Most options are mandatory.

   The Mail task sends a e-mail through a SMTP server. The default port for the mail
   submission is 587 which is the most common port for mail submissions.

   This command can also be specified as tmail, i.e. episodectl.exe tmail


      -u
--username  Username for outgoing mail server.

      -p
--password  Password for outgoing mail server.

      -s
 --server  The server to submit the mail to. The outgoing SMTP mail server. Mandatory
           option.

   --port  The port to connect to on the outgoing SMTP mail server. Default is 587.

      -f
   --from  Mail sender address. eg. EpisodeEngine@mycompany.com Mandatory option.

      -t
    --to  Mail receiver address. Mandatory option.

      -c
    --cc  Carbon Copy receiver address.

      -b
   --bcc  Blind Carbon Copy receiver address.

      -j
--subject  Message subject. This text could contain dollar variables, for example
           'Error: $source.filename$ failed to encode!'. See episodectl.exe variables
           for more information. Mandatory option.

      -m
--message  The message. This text could contain dollar variables, for example 'Error:
           $source.filename$ failed to encode!'. See episodectl.exe variables for more
           information. Mandatory option.

    --ssl  If TLS/SSL should be used. Default is "yes" and the task fails if a secure
           connection couldn't be set up.

   --name  A name for the task. The recipient's address is used by default.

--priority  Set the priority of the task, default is 0. See episodectl.exe priority for
```

more information.

--tag Set one or more tags on this task. See tag section (episodectl.exe tags) for further information.

--inverse-tag Set one or more inverse-tags on this task. See tag section (episodectl.exe tags) for further information.

--format Format of the ouput configuration file. The argument can be any of xml, simple-xml , ascii, bin. The default is xml which is an XML format with type information. There is a simple-xml format that is more readable and easier to use for manual editing but use this with caution because it may not be upgradable/usable when a new version of Episode is released. ascii is a more compact format and more efficient to work with (IO, storage etc) and still kind of readble but not recommended for editing. bin is a binary format and about the same size as ascii but not readble nor editable. ascii and bin are therefore only recommended for "automatic" use, i.e. from a program that both creates them and submits them.

-o
--out Specify the directory where the output file should be written. Default is current working directory.

--print-plain-path Print the path to the written configuration file without any other text or newline etc. Usable if a program/script should run this command and read back the path.

task mbr HTTPStreaming|SmoothStreaming|DASHStreaming
[--name <name>]
[--fragment-duration <duration>]
[--package-name <package name>]
[--priority <priority>]
[--tag <tag> ...]
[--inverse-tag <tag> ...]
[--format xml|simple-xml|ascii|bin]
[-o <directory>]
[--print-plain-path]

Create a MBR .epitask configuration file where the first argument is the type of packaging to perform. The three currently available values/packaging types are HTTPStreaming , DASHStreaming, and SmoothStreaming.

The MBR task creates Multi Bitrate output packages for Microsoft's Smooth Streaming, Dynamic Adaptive Streaming over HTTP (DASH) or Apple's HTTP Live Streaming. The package will be put in the output directory specified by the user. The input files to the MBR task must be one or more TIFO files.
TIFO is short for Telestream Intermediate Format and is available in the Episode GUI as a export format for the Encode task. Each TIFO file can contain one or more tracks of H264 video and AAC audio. The tracks are said to be unique depending on their bitrate. For HTTP Streaming the output will be a set of TS (Transport Stream) segments. Each source TIFO file will be packed into a separate set of TS segments. In the case of Smooth Streaming the output will be separate TS files where each uniquely identified track in the source files will be put into one output file.

This command can also be specified as tmbr, i.e. episodectl.exe tmbr

--name A name for the task. A suitable name is generated if not specified.

--fragment-duration Specify the fragment duration (in whole seconds). Default for HTTPStreaming is 10, for DASHStreaming is 2, and for SmoothStreaming 2.

--package-name Specify the package name. This name will be used as the file prefix for the final ouput files inside the output directory and in the manifest file or playlist file.

--priority Set the priority of the task, default is 0. See episodectl.exe priority for more information.

--tag Set one or more tags on this task. See tag section (episodectl.exe tags) for further information.

--inverse-tag Set one or more inverse-tags on this task. See tag section (episodectl.exe tags) for further information.

--format Format of the ouput configuration file. The argument can be any of xml, simple-xml , ascii, bin. The default is xml which is an XML format with type information. There is a simple-xml format that is more readable and easier to use for manual editing but use this with caution because it may not be upgradable/usable when a new version of Episode is released. ascii is a more compact format and more efficient to work with (IO, storage etc) and still kind of readble but not recommended for editing. bin is a binary format and about the same size as ascii but not readble nor editable. ascii and bin are therefore only recommended for "automatic" use, i.e. from a program that both creates them and submits them.

-o
--out Specify the directory where the output file should be written. Default is current working directory.

--print-plain-path Print the path to the written configuration file without any other text or newline etc. Usable if a program/script should run this command and read back the path.

task set <task configuration file>
[--name <name>]
[--tag <tag> ...]
[--inverse-tag <tag> ...]
[--remove-tag <tag> ...]
[--priority <priority>]

Set/Modify a existing <task configuration file> (path to a .epitask file). These settings are general task settings which apply to all tasks regardless of what kind of task it is.

This command can also be specified as tset, i.e. episodectl.exe tset

--name Re-name the task.

NOTE: This does not rename the actual file. It only changes the configured task name inside the file.

--priority Set or modify the priority of the task. See episodectl.exe priority for more information.

--tag Set one or more tags on this task. No previously set tag is removed. See tag section (episodectl.exe tags) for further information.

--inverse-tag Set one or more inverse-tags on this task. No previously set tag is removed. See tag section (episodectl.exe tags) for further information.

--remove-tag Remove one or more tags (or inverse-tags) from the task.

source

With the source sub command, you can create episource configuration files. The created file will be written in the current working directory unless --out <directory> is specified.

source filelist <url or path> ...
[--name <name>]
[--format xml|simple-xml|ascii|bin]
[-o <directory>]
[-p]

Create a file-list .episource configuration file.

This command can also be specified as sfl, i.e. episodectl.exe sfl

--name A name for the source.

-o
--out-dir Specify the directory where the output file should be written. Default is current working directory.

-p
--print-plain-path Print the path to the written configuration file without any other text or newline etc. Usable if a program/script should run this command and read back the path.

source watch-folder <url or path>
[--name <name>]
[--interval <seconds>]
[--safe-delay <seconds>]

```
[--pickup-present yes|no]
[--recursive <depth>]
[--ewi yes|no]
   [--ewi-wait 0..3600]
   [--ewi-num-waits 0..100]
[--iseq yes|no [--msni 0..1000000] [--sf 0..1000] [--fr <framerate>]]
[--udtim 0|1|2]
[--workflow-failure encode|deploy]
   [--move-source <url or path>]
   [--remove-source]
   [--stop-encoders]
[--workflow-success]
   [--move-source <url or path>]
   [--remove-source]
[--retry-delay-start <seconds>]
[--retry-delay-factor <factor>]
[--retry-delay-max <seconds>]
[--retry-max-attempts <attempts>]
[--file-name-contains-include <text> ...
[--file-name-contains-ignore <text> ...
[--file-name-is-include <name> ...
[--file-name-is-ignore <name> ...
[--file-name-begins-with-include <text> ...
[--file-name-begins-with-ignore <text> ...
[--file-name-ends-with-include <text> ...
[--file-name-ends-with-ignore <text> ...
[--file-extension-is-include <extension> ...
[--file-extension-is-ignore <extension> ...
[--directory-contains-include <text> ...
[--directory-contains-ignore <text> ...
[--directory-is-include <name> ...
[--directory-is-ignore <name> ...
[--max-size <bytes>]
[--min-size <bytes>]
[--max-mod-date <POSIX time>]
[--min-mod-date <POSIX time>]
[--format xml|simple-xml|ascii|bin]
[-o <directory>]
[-p]
```

Create a watch folder .episource configuration file where <url or path> should be a URL or path to an existing directory to be watched. A common configuration request is to re-create the watched directory structure (when recursive directory monitoring is configured) and that is an option on the Upload task, please see episodectl.exe task transfer -h for information about the option --re-create-source-sub-dirs.

This command can also be specified as smon, i.e. episodectl.exe smon

--name A name for the watch folder.

--interval Interval in seconds to check for directory changes (default is dependent on which watch folder plug-in that is used but typically between 2 and 5 seconds).

--safe-delay Safe delay in seconds. If no file change is discovered for this amount of

time, the file will be reported as ready. There is also an adaptive delay mechanism, see option --udtim. The default is 10 seconds.

--pickup-present If the watch folder should pick up files that are present when the watch folder is started (default is no).

--recursive Set a directory depth to which a recursive sub directory search should be made (default is 0, i.e. don't look in sub directories).

--ewi Encode While Ingest. If you use Telestream Pipeline for ingesting material, this option will encode the stream as it is written to disk. This basically means that the transcoding will be finished at about the same time as the ingest (if the transcoding is faster than, or equal to real-time). NOTE: This requires a stream source format. Also keep in mind that if 2-pass encoders are used, the second pass will never start until the first pass is finished, and the first pass will never finish before the ingest is finished. NOTE: Due to many SANs reporting wrong / duplicate files in any given listing, we advise against using EWI in combination with SAN filesystems (especially with image sequences).

--ewi-wait Number of seconds to wait before checking if a file has grown in size or in the case of Image Sequences, if a new file has arrived (at the end of the sequence). Default is 5 seconds.

--ewi-num-waits Number of times to wait. Default value is 5 which means that the total time to wait for a file-change detection, or in the case of Image Sequences, a new file at the end of the sequence has arrived, will default be 5 * 5 = 25 seconds.

--iseq If the watch folder should look for Image Sequences. Notice though that regular files, i.e. files where no sequence number is found, will be reported by the watch folder. The final sequence start, i.e. the first file (the file that will be reported) will have to arrive/be detected before --safe-delay "runs out". If this is used together with --ewi yes, the first file will be reported in the first "pass" of the watch folder, i.e. --safe-delay will have no effect. So to sum this up, if you use a scanner or other device that will guarantee that the first "true" file of the sequence will be detected first (or within the same watch folder pass), you can use --ewi yes. If you will transfer files some other way where the order of arrival is not guaranteed, you should NOT use "Encode While Ingest".

--msni This only affects option --iseq. See episodectl.exe source iseq -h.

--sf This only affects option --iseq. See episodectl.exe source iseq -h.

--fr This only affects option --iseq. See episodectl.exe source iseq -h.

--udtim This is short for "Update Detection Time Interval Multiplier". What it does is to measure the time between file update detections (UDTI) and Multiplies that time with a configurable number and then adds it to the "base safe-delay" specified by option --safe-delay. The new value is always based on the most reacent interval and could therefore go down. The need for this functionality has been observed on certain SANs. A value of 0 means no increase in safe delay (UDTI * 0 + safe-delay). A value of 1 is a proportinal adjustment (UDTI * 1 + safe-delay). A value of 2 is an exponential adjustment (UDTI * 2 + safe-delay). The default value is 1.

--workflow-failure Specify actions to be taken in the workflow in case of encoding or deployment failure. This will fulfill if ANY task fails (if multiple encoders or multiple deployments are present within the workflow). The argument to this option must be either "encode" or "deploy" and there are three "sub options" that could be used which are --move-source, --remove-source, and/or --stop-encoders. The valid combinations of these sub options are:
--move-source <url or path> --stop-encoders
--remove-source --stop-encoders
--stop-encoders
--move-source <url or path>
--remove-source

--workflow-success Specify actions to be taken in the workflow in case of deployment success. This will fulfill if ALL tasks succeeds (if multiple deployments are present within the workflow). There are two "sub options" that could be used which are --move-source or --remove-source where only either one of them may be specified.

--move-source Move the source file to <url or path> which should point to a existing directory.

--remove-source Remove the source file.

--stop-encoders Stop any non-finished encode tasks within the workflow (if multiple encoders were present within the workflow).

--dest-sub-dirs This is short for "Maximum Sequence Number Increase" which enables sequences with "gaps" in the sequence numbers. The default value is 1 which means that as soon as there is a missing number in a sequence, it will be treated as the end of the sequence (the sequence to encode).

--retry-delay-start If the watch folder fails to list the directory for some reason, it will try to list it again after specified amount of seconds (unless --retry-max-attempts is set to 0). The default is 20 seconds.

--retry-delay-factor For every retry to list the directory, the value specified with --retry-delay-start will be multiplied with this value until --retry-delay-max is reached. The default value is 2, i.e. the retry delay will be doubled.

--retry-delay-max The max amount of time in seconds that the retry delay will reach. The default value is 3600 seconds which equals 1 hour.

--retry-max-attempts The maximum number of tries to list a directory (that for some reason is un-listable) before giving up and stop monitoring. The default value is 40.

--file-name-contains-include One or more text strings that a filename should contain in order to be reported by the watch folder.

NOTE: This searches the file's name without the file extension.

--file-name-contains-ignore One or more text strings that a filename should contain in order to NOT be reported by the watch folder. In other words - if any specified text string is found in the filename, that file will be ignored by the watch folder.

NOTE: This searches the file's name without the file extension.

--file-name-is-include One or more file names that should be equal to found files in order to be reported by the watch folder.

NOTE: This matches the file's name without the file extension.

--file-name-is-ignore Ignore files where the file's name matches (one of) the specified names.

NOTE: This matches the file's name without the file extension.

--file-name-begins-with-include Report (include) files where the file name begins with (one of) the specified text string(s).

NOTE: This matches the file's name without the file extension.

--file-name-begins-with-ignore Ignore files where the file name begins with (one of) the specified text string(s).

NOTE: This matches the file's name without the file extension.

--file-name-ends-with-include Report (include) files where the file name ends with (one of) the specified text string(s).

NOTE: This matches the file's name without the file extension.

--file-name-ends-with-ignore Ignore files where the file name ends with (one of) the specified text string(s).

NOTE: This matches the file's name without the file extension.

--file-extension-is-include Report (include) files where the file extension is equal to (one of) the specified file extension(s).

--file-extension-is-ignore Ignore files where the file extension is equal to (one of) the specified file extension(s).

--directory-contains-include Report (include) files which are found inside a directory where the directory name contains (one of) the specified text strings. This is only applicable for watch folders that does recursive directory listings. If the directory depth is more than 1, all individual directory names will be checked, not just the "deepest" one (the one in which the file actually is located).

--directory-contains-ignore Ignore files which are found inside a directory where the directory name contains (one of) the specified text strings. This is only applicable for watch folders that does recursive directory listings. If the directory depth is more than 1, all individual directory names will be checked, not just the "deepest" one (the one in which the file actually is located).

--directory-is-include Same as --directory-contains-include except that the directory name must be equal.

--directory-is-ignore Same as --directory-contains-ignore except that the directory name must be equal.

--max-size The maximum size in bytes that a file must have in order to be reported by the watch folder.

--min-size The minimum size in bytes that a file must have in order to be reported by the watch folder.

--max-mod-date Files that have a modification date that is older than specified date will be reported and newer files will be ignored. This should be specified as a "Unix time" or "POSIX time". It is the number of seconds passed since 00:00:00 UTC, January 1, 1970. In the current timezone, the start time was Wed Dec 31 16:00:00 1969. The current POSIX time is 1432052485. See http://en.wikipedia.org/wiki/Unix_time for more information.

--min-mod-date Files that have a modification date that is newer than specified date will be reported and older files will be ignored. This should be specified as a "Unix time" or "POSIX time". It is the number of seconds passed since 00:00:00 UTC, January 1, 1970. In the current timezone, the start time was Wed Dec 31 16:00:00 1969. The current POSIX time is 1432052485. See http://en.wikipedia.org/wiki/Unix_time for more information.

--format Format of the ouput configuration file. The argument can be any of xml, simple-xml , ascii, bin. The default is xml which is an XML format with type information. There is a simple-xml format that is more readable and easier to use for manual editing but use this with caution because it may not be upgradable/usable when a new version of Episode is released. ascii is a more compact format and more efficient to work with (IO, storage etc) and still kind of readble but not recommended for editing. bin is a binary format and about the same size as ascii but not readble nor editable. ascii and bin are therefore only recommended for "automatic" use, i.e. from a program that both creates them and submits them.

-o
--out-dir Specify the directory where the output file should be written. Default is current working directory.

-p
--print-plain-path Print the path to the written configuration file without any other text or newline etc. Usable if a program/script should run this command and read back the path.

source edl --clip <url or path> [--in <time>] [--out <time>]
[--clip <url or path> [--in <time>] [--out <time>]]
[--clip <url or path> [--in <time>] [--out <time>]]
[...]
[--name <name>]
[--format xml|simple-xml|ascii|bin]
[-o <directory>]
[-p]

Create a EDL .episource configuration file. This command is a bit special because the order in which the options are specified is very important and the same options should be specified over and over again.

This is an example where a EDL is specified consisting of 3 source files. The first file is cut with seconds, the second file is not cut at all and the third file is cut with time code.
episodectl.exe source edl --clip file1.mov --in 1.00 --out 3.00 --clip file2.mov

```
        --clip file3.mov --in 00:00:01:01 --out 00:01:00:01

        Each segment in the list begins with the --clip option which takes one source file
        as argument/parameter. After each --clip <url or path> specification, exactly one --in
        and one --out may follow. Any of the --in/--out options may be left out in which
        case there will be no cutting in the beginning/end of the file. If the same source
        file should be used in multiple cuts, each cut must be specified with its own --clip
        option.

        This command can also be specified as sedl, i.e. episodectl.exe sedl


        --name  A name for the source.

        --clip  Specifies a clip in the list of clips where <url or path> is the URL or path
              to a source file.

         --in  Specify the in-point of the current --clip ... where <time> could be either a
              fractional number treated as seconds, or a time code in the format
              "HH:MM:SS:FF" and optionally followed by a frame rate definition with format
              "HH:MM:SS:FF@FR" where FR can have one of the following "values": 23.976, 24,
              25, 29.97, 30, 50, 59.94, 60.

         --out  Specify the out-point of the current --clip. See description of option --in.

       --format  Format of the ouput configuration file. The argument can be any of xml, simple-xml
              , ascii, bin. The default is xml which is an XML format with type
              information. There is a simple-xml format that is more readable and easier
              to use for manual editing but use this with caution because it may not be
              upgradable/usable when a new version of Episode is released. ascii is a more
              compact format and more efficient to work with (IO, storage etc) and still
              kind of readble but not recommended for editing. bin is a binary format and
              about the same size as ascii but not readble nor editable. ascii and bin are
              therefore only recommended for "automatic" use, i.e. from a program that
              both creates them and submits them.

            -o
      --out-dir  Specify the directory where the output file should be written. Default is
              current working directory.

            -p
--print-plain-path  Print the path to the written configuration file without any other text or
              newline etc. Usable if a program/script should run this command and read
              back the path.



  source iseq <url or path> ...
          [--msni 0..1000000]
          [--sf 0..1000]
          [--fr <framerate>]
          [--ewi yes|no]
             [--ewi-wait 0..3600]
             [--ewi-num-waits 0..100]
          [--name <name>]
          [--format xml|simple-xml|ascii|bin]
          [-o <directory>]
```

[-p]

Create a Image Sequence .episource configuration file. Each specified <url or path> should be a URL or path to the first image in a sequence, i.e. the first image in the sequence you want to encode, it could of course be an image in the middle of a sequence too. If you use images that lacks frame rate information, Episode will default to 25 fps. A desired frame rate may be set using the Advanced Frame Rate filter.

This command can also be specified as siseq, i.e. episodectl.exe siseq

--name A name for the source.

--msni This is short for "Maximum Sequence Number Increase" which enables sequences with "gaps" in the sequence numbers. The default value is 1 which means that as soon as there is a missing number in a sequence, it will be treated as the end of the sequence (the sequence to encode).

--sf This is short for "Safety Files" which controls how many files after any given file need to be present in the sequence before processing of said file begins. This does not apply to the first file in a sequence. This only has practical implications if "Encode While Ingest" is enabled. Setting this to 0 may significantly decrease the time the encoder has to wait when the end of the sequence is reached, but depending on file system and file format, may result in errors. Default is 1.

--fr Specifies which framerate to default to if the file header does not contain such info.

--ewi Enable/Disable "Encode While Ingest". This is suitable if you have a scanner or other device that writes new files over time. This will tell the sequence reader to wait for more images to arrive. This waiting time is configurable with the two options below. Default is "no". Keep in mind that if 2-pass encoders are used, the second pass will never start until the first pass is finished, and the first pass will never finish before the whole sequence is finished.

--ewi-wait Number of seconds to wait before checking if a new file has arrived. Default is 5 seconds.

--ewi-num-waits Number of times to wait. Default value is 5 which means that the total time to wait for a new file will default be 5 * 5 = 25 seconds.

--format Format of the ouput configuration file. The argument can be any of xml, simple-xml , ascii, bin. The default is xml which is an XML format with type information. There is a simple-xml format that is more readable and easier to use for manual editing but use this with caution because it may not be upgradable/usable when a new version of Episode is released. ascii is a more compact format and more efficient to work with (IO, storage etc) and still kind of readble but not recommended for editing. bin is a binary format and about the same size as ascii but not readble nor editable. ascii and bin are therefore only recommended for "automatic" use, i.e. from a program that both creates them and submits them.

-o

--out-dir Specify the directory where the output file should be written. Default is current working directory.

-p
--print-plain-path Print the path to the written configuration file without any other text or newline etc. Usable if a program/script should run this command and read back the path.

workflow

A Workflow consists of tasks usually forming a graph, tree, or "line" (branch). A subsequent task in a branch is run after the preceding one by two main conditions. Either that the preceding task sets one or more in-values or "variable" values on the subsequent task, which the subsequent task is dependent on to run, or that the subsequent task is triggered by a "result", or "status" of the preceding task, i.e. if it failed or succeeded.

A submitted workflow is always considered a "Template Workflow" from which any number of "Started Workflows" may originate. The started workflows are "Spawned" off a template by some in-value, usually a source file. If, for example a file source is used that consists of a file list of 3 files, 3 started workflows will be spawned from that source+workflow. If you have a watch folder source, each of the files that are reported by the watch folder will spawn off a started workflow from the template workflow that the watch folder source is attached to.

NOTE: When you save a .episubmission file in the GUI, it may or may not contain a source. In either case, the source can be overridden or inserted by the submit -s ... command described below.

If you do submissions and consider the entire submission as one job that can succeed or fail, you should use --id-out and poll status with status workflows -i ID. If you want to see status for the individual started workflows, you can use --ids-out and be able to poll statuses of all IDs with status workflows ID1 ID2 ... Even more detailed status information can be obtained about the individual tasks in the workflows by calls to status tasks.

```
workflow submit -e <encoder-file or directory-of-encoderfiles> ...
        [-f <sourcefile or directory-of-sourcefiles> ...]
        [--file-source <file-source-file>]
        [--edl <edl-source-file>]
        [--iseq <iseq-source-file>]
        [--watch-folder <watch-folder-source-file or directory-to-monitor>]
        [-d <directory or deployment-task-file>]
        [-x <Execute-file>|<script-file> [done|failure|success]]
        [-m <Mail-file> [done|failure|success]]
        [--mbr HTTPStreaming|SmoothStreaming|DASHStreaming|<MBR-file> <encoder-file
> ...]
        [--workflow-failure encode|deploy
           [--move-source <url or path>]
           [--remove-source]
           [--stop-encoders]]
        [--workflow-success
```

```
    [--move-source <url or path>]
    [--remove-source]]
[--naming <naming convention>]
[--split
    [--max-splits 2..32]
    [--split-on-gop-size 1..100]
    [--min-split-time 9..3600]]
[-n <name>]
[--workflow-name <workflow name>]
[--priority <priority>]
[--set-name <task-username> <variable> <value> ... ...]
[--set-type <task-typename> <variable> <value> ... ...]
[--set-list-name <task-username> <list-variable> <value> ...]
[--set-list-type <task-typename> <list-variable> <value> ...]
[--tag-name <task-username> <tag> ...]
[--tag-type <task-typename> <tag> ...]
[--tag-workflow <tag> ...]
[--inverse-tag-name <task-username> <tag> ...]
[--inverse-tag-type <task-typename> <tag> ...]
[--inverse-tag-workflow <tag> ...]
[-r <hours until expiration>]
[--no-resource]
[--id-out]
[--ids-out [<separator char>]]
[--watch-folder-id-out]
[-w [-v]]
[--host <hostname/IP>]
[--cluster <name>]
[--timeout <seconds>]
[--demo]
[-o [<directory>] [--format xml|simple-xml|ascii|bin] [-p]]
```

The above creates a one-shot workflow

```
workflow submit -s <submission-file or directory-of-submission-files> ...
    [-f <sourcefile or directory-of-sourcefiles> ...]
    [--file-source <file-source-file>]
    [--edl <edl-source-file>]
    [--iseq <iseq-source-file>]
    [--watch-folder <watch-folder-source-file or directory-to-monitor>]
    [--naming <naming convention>]
    [--split
        [--max-splits 2..32]
        [--split-on-gop-size 1..100]
        [--min-split-time 9..3600]]
    [-n <name>]
    [--workflow-name <workflow name>]
    [--priority <priority>]
    [--set-name <task-username> <variable> <value> ... ...]
    [--set-type <task-typename> <variable> <value> ... ...]
    [--set-list-name <task-username> <list-variable> <value> ...]
    [--set-list-type <task-typename> <list-variable> <value> ...]
    [--tag-name <task-username> <tag> ...]
    [--tag-type <task-typename> <tag> ...]
    [--tag-workflow <tag> ...]
    [--inverse-tag-name <task-username> <tag> ...]
```

```
        [--inverse-tag-type <task-typename> <tag> ...]
        [--inverse-tag-workflow <tag> ...]
        [-r <hours until expiration>]
        [--no-resource]
        [--id-out [<separator char>]]
        [--ids-out [<separator char>]]
        [--watch-folder-id-out [<separator char>]]
        [--id-group-separator [<separator char>]]
        [-w [-v]]
        [--host <hostname/IP>]
        [--cluster <name>]
        [--timeout <seconds>]
        [--demo]

    The above takes a submission configuration file


Submit Jobs to a Node. A Workflow instance will be created for every source file
specified.
If a directory is specified (ends with a '\'), all files in that directory will be
submitted.

There are a two ways to submit Jobs. Either by building a workflow "on the fly" (a
one-shot workflow) or by submitting a saved submission configuration file.

If option -e is used, a new "one-shot" Workflow will be built from the Encoder Task
configuration files (or Episode 5.x settings), destination(s) specified with option -d
.

If option -s is used, the specified files should be saved submission files (for
example a .episubmission file). If one or more directories are specified all
recognizable submission files found will be submitted.

This command can also be specified as ws, i.e. episodectl.exe ws


  --host  A host Node to operate on (default is 127.0.0.1). If this Node is a
          participant of a cluster but it is not the Master Node, we will
          automatically be redirected to the Master Node. You can configure a default
          host/IP to submit to with the command episodectl.exe proxy defaults --host
          <default host>.

    -c
--cluster  A Cluster name to operate on. This will try to find a Cluster participant
          via Bonjour. You can configure a default cluster to submit to with the
          command episodectl.exe proxy defaults -c <default name>.

--timeout  A timeout for the request. This command will connect to the Node through the
          ClientProxy and this timeout will be used in both those requests but it is
          unlikely that they will "add up". The default timeout is 30.

    -f
  --file  Specify one or more files as source files. You can also specify one or more
          directories in which case they should end with a slash. If a directory is
          specified, all files in that directory will be added to the file source
          list. In the case of option -s, this will override the source if one is
          already present.
```

--file-source Specify a File List configuration file as source. In the case of option -s, this will override the source if one is already present.

--edl Specify a EDL configuration file as source. In the case of option -s, this will override the source if one is already present.
NOTE: All $source.xxx$ variables (see episodectl.exe variables for more information) will be set with information about the first file in the EDL list.

--iseq Specify a Image sequence configuration file as source. In the case of option -s, this will override the source if one is already present.

--watch-folder Specify a Watch Folder configuration file as source or a directory to be monitored. If a directory to be monitored is specified, it should end with a slash ('/') in order to be identified as a directory (this could be a URL too, in which case episodectl.exe will not try to figure out if it's a directory or configuration file, that's why). In the case of option -s, this will override the source if one is already present.

-e
--encoder One or more Encoder Task configuration files or Episode 5.x setting files or a directory. If a directory is specified, that directory will be "searched" (recursively) for Encoder task configuration files and every Encoder task file found will be added to the workflow.

--mbr Specify either a multi bit rate (MBR) .epitask configuration (as <MBR-file>) file or a package type (HTTPStreaming, DASHStreaming, or SmoothStreaming) followed by any number of encoder epitasks which are to provide the encoded streams to the MBR task. This option can be supplied several times to generate a workflow with several packagings. <Note:> If the same encoder epitask appears in several packaging options, only one encoding will be performed and the resulting encoded stream will be shared between the MBR tasks (optimization) <Note:> You do not need to specify any encoders using the -e switch if --mbr is used.

-d
--destination A Destination to be used instead of the default one. The argument may be a Destination Task configuration file (see episodectl.exe task transfer -h on how to create one), a directory URL, or a directory path. If it is a directory path, it should end with a \. If no destination task or destination folder is specified, the default one will be used. The default one is configurable with the command episodectl.exe proxy defaults -d <default destination>.

-x
--execute Specify one or more Execute Task configuration files or script files to be run after the deployment (each deployment). If the specified file has the file extension "epitask" it's considered to be a Execute Task configuration file (see episodectl.exe task execute -h on how to create one) and if it has any other exension (or no extension), it is considered to be a script. If a script file is specified, a Execute task is created and configured with the content and file extension of that script (extension is for Windows where interpreter is chosen based on file extension). In other words, a copy of the script is sent for execution in the workflow. You can specify this option multiple times in order to run several Execute tasks either in case of previous task finish ('done'), previous task failure, or in case of

previous task success. The 'done', 'failure' or 'success' is per branch in a graph, i.e. if you specify two Encode tasks with option -e, and 2 Deployments with option -d, a total of four branches are created and each of these branches will have a copy of the done, failure and/or success Execute tasks. To specify one failure and two success scripts you may write: -x /path/to/Failure.epitask failure -x /path/to/Success1.epitask -x /path/to/Success2.epitask. The default is to run the Execute task in case of success. If a Encode + Deploy branch fails and the Execute task is configured to run on success, the status 8 or "Redundant" will be set on the Execute task. See example section (episodectl.exe examples) for further information.

-m
--mail Specify one or more Mail Task configuration files to be run after the deployment (each deployment). You may specify this option multiple times in order to run several Mail tasks either in case of previous task finish ('done'), previous task failure, or in case of previous task success. The 'done', 'failure' or 'success' is per branch in a graph, i.e. if you specify two Encode tasks with option -e, and 2 Deployments with option -d, a total of four branches are created and each of these branches will have a copy of the failure and/or success Mail tasks. To specify one failure and one success Mail tasks you may write: -m /path/to/FailureMail.epitask failure -m /path/to/SuccessMail.epitask. The default is to run the Mail task in case of failure. If a Encode + Deploy branch succeeds and the Mail task is configured to run on failure, the status 8 or "Redundant" will be set on the Mail task. See example section (episodectl.exe examples) for further information.

--workflow-failure Specify actions to be taken in the workflow in case of encoding or deployment failure. This will fulfill if ANY task fails (if multiple encoders or multiple deployments are present within the workflow). The argument to this option must be either "encode" or "deploy" and there are three "sub options" that could be used which are --move-source, --remove-source, and/or --stop-encoders. The valid combinations of these sub options are:
--move-source <url or path> --stop-encoders
--remove-source --stop-encoders
--stop-encoders
--move-source <url or path>
--remove-source

--workflow-success Specify actions to be taken in the workflow in case of deployment success. This will fulfill if ALL tasks succeeds (if multiple deployments are present within the workflow). There are two "sub options" that could be used which are --move-source or --remove-source where only either one of them may be specified.

--move-source Move the source file to <url or path> which should point to a existing directory.

--remove-source Remove the source file.

--stop-encoders Stop any non-finished encode tasks within the workflow (if multiple encoders were present within the workflow).

--set-name Set one or more variables on task(s) with specified name. The first argument is the user defined task name and following arguments should be

"variablename value variablename value" etc. See variable section (episodectl.exe variables) for further information.

--set-type Set one or more variables on task(s) with specified "type name". The first argument is one of the type names listed below and following arguments should be "variablename value variablename value" etc. See variable section (episodectl.exe variables) for further information. Available type names are:

Note for 6.2.x users: "Encode" was previously called "Encoder" and "Transfer" was previously called "Uploader". The old names still works but should be changed as soon as possible.

Encode
Transfer
YouTube
Execute
Mail
MBR

--set-list-name Set several values that makes up a list; on task(s) with specified name. See variable section (episodectl.exe variables) for further information.

--set-list-type Set several values that makes up a list; on task(s) with specified "type name". See variable section (episodectl.exe variables) for further information.

--tag-name Set one or more tags on task(s) with specified name. The first argument is the user defined task name and following arguments should be tags. See tag section (episodectl.exe tags) for further information.

--tag-type Set one or more tags on task(s) with specified "type name". The first argument is one of the type names listed below and following arguments should be tags. See tag section (episodectl.exe tags) for further information. Available type names are:

Note for 6.2.x users: "Encode" was previously called "Encoder" and "Transfer" was previously called "Uploader". The old names still works but should be changed as soon as possible.

Encode
Transfer
YouTube
Execute
Mail
MBR

--tag-workflow Set one or more tags on the whole workflow, every task will inherit these tags. See tag section (episodectl.exe tags) for further information.

--inverse-tag-name Set one or more inverse-tags on task(s) with specified name. The first argument is the user defined task name and following arguments should be tags. See tag section (episodectl.exe tags) for further information.

--inverse-tag-type Set one or more inverse-tags on task(s) with specified "type name". The first argument is one of the type names listed below and following arguments should be tags. See tag section (episodectl.exe tags) for further

information. Available type names are:

Note for 6.2.x users: "Encode" was previously called "Encoder" and "Transfer" was previously called "Uploader". The old names still works but should be changed as soon as possible.

Encode
Transfer
YouTube
Execute
Mail
MBR

--inverse-tag-workflow Set one or more inverse-tags on the whole workflow, every task will inherit these tags. See tag section (episodectl.exe tags) for further information.

--split Configure the Encoder(s) to do Split-and-Stitch. (This is the same as "--set-type Encoder sns yes")

--max-splits Specify maximum number of splits created. This only applies to option --split . The default value is 16.

--min-split-time Specify minimum duration in seconds for each split. This only applies to option --split. The default value is 30.

-r
--resource Specify an expiry time for IO shares. Default is 1 week.

--no-resource Don't convert files and directories into TSRC URIs, and don't share anything in the local IOServer. If you use this option, you must have configured all the nodes in a cluster to have access to the same storage at the same path. You can not use "Named Storages" if you use this option and you will most certainly get "Access Denied" errors if you don't turn off "IO Verificaition", see episodectl.exe node jobs -h for more information on that. This option is therefore not recommended.

-n
--name Name the submission. This does NOT have to be a unique name. If used together with option -s and multiple submission files, this will override the name on all submissions and name them the same.

--workflow-name Name the workflow. The workflow will default be named the same as the submission. This name can be used for job control through command episodectl.exe job ... -n <workflow name>.

-s
--submission Specifies that the input files are submission configuration files. If option -f, --edl, or --watch-folder is used in conjunction with this option, the source in the submission file will be replaced if one is already present.

--naming Insert a custom naming convention. The naming convention is specified as one string and you can insert variables. See variable section (episodectl.exe variables) for a list of the variables.

An example of the default naming convention would be specified as '$source.filename$-$encoder.name$'.If you're specifying this command in a shell, be sure to enclose it with single quotes as many shells interpret '$'

as a special character. (This is the same as "--set-list-type Transfer dest-filename '$source.filename$' - '$encoder.name$'")

--priority Set priority on the workflow. A higher value equals higher priority. The value can be any 64-bit integer, negative or positive (If you don't know what this means, it means very big numbers). The default priority is 0.

-w
--wait Wait until the started Workflows are finished. Program will exit with 0 if every job finished without failure and 1 if any of the jobs (started workflows) failed. NOTE: This option will have no effect if a watch folder source is used.

-v
--visual Get a visual (human readable) representation of the submitted workflow that is updated with intervals. NOTE: This is mostly useful for testing purposes. NOTE: This only applies to option --wait.

--id-out Output Template Workflow ID (from which any number of started workflows are created). This ID can be used to retrieve status through the command status workflows -i ... or status tasks -i .... If option -s is used, multiple Template IDs will be printed if multiple submission files were specified in which case you can optionally specify a separator character as argument, the default is a vertcal bar or "pipe" character ('|'). There are two special identifiers you can specify - "newline" or "LF" to get a newline separator.

--ids-out Output Started Workflow IDs. These IDs can be used to retrieve status with the commands status workflows ... or status tasks .... An optional separator character can be specified, default is a vertcal bar or "pipe" character ('|'). There are two special identifiers you can specify - "newline" or "LF" to get a newline separator.

--watch-folder-id-out Output Watch Folder ID if the source was a watch folder source. This ID can be used to control the watch folder with the episodectl.exe watch-folder command. If option -s is used, multiple Template IDs will be printed if multiple submission files were specified in which case you can optionally specify a separator character as argument, the default is a vertcal bar or "pipe" character ('|'). There are two special identifiers you can specify - "newline" or "LF" to get a newline separator.

--demo Submits the workflow with demo mode set (will add watermark)

-o
--out If you use episodectl.exe ws -e ... to build a workflow "on the fly", you can save that workflow to a .episubmission and later submit it through episodectl.exe ws -s .... The options --format and --print-plain-path are directly tied to this option (and the save functionality). The ouput filename will be generated unless option -n <name> is specified in which case <name> will be the filename and the extension .episubmission. The configuration file will be written to the current working directory unless the optional argument/parameter <directory> is specified. NOTE: Some things are not saved when you use this option. These include all variable override options --set-name --set-type --set-list-name --set-list-type and the utility option --split as well as workflow --priority and --demo. These options must be specified when the workflow is submitted. If this option is used from within a program/script, you can use option --print-plain-path to read/store the complete path in a variable.

```
    --format  Format of the ouput configuration file. The argument can be any of xml, simple-xml
              , ascii, bin. The default is xml which is an XML format with type
              information. There is a simple-xml format that is more readable and easier
              to use for manual editing but use this with caution because it may not be
              upgradable/usable when a new version of Episode is released. ascii is a more
              compact format and more efficient to work with (IO, storage etc) and still
              kind of readble but not recommended for editing. bin is a binary format and
              about the same size as ascii but not readble nor editable. ascii and bin are
              therefore only recommended for "automatic" use, i.e. from a program that
              both creates them and submits them. NOTE: This option only applies to option
              -o.

          -p
--print-plain-path  Print the path to the written configuration file without any other text or
              newline etc. Usable if a program/script should run this command and read
              back the path. NOTE: This option only applies to option -o.
```

```
workflow stop [<workflow id> ...]
              [-i <template id>]
              [--all]
              [--host <hostname/IP>]
              [-c <cluster name>]
              [--timeout <seconds>]
```

```
    Stops any running tasks and removes the workflow from the node's/cluster's "active
    state". The workflow will end up in the history. Started workflows are specified
    without any option and you can specify one Template workflow withy option -i.

    This command can also be specified as wp, i.e. episodectl.exe wp


      -i
     --id  Specify that this ID is a Template ID. Any workflows originating from this
           will be stopped and removed.

    --all  Stop all running workflows.

    --host  A host Node to operate on (default is 127.0.0.1). If this Node is a
            participant of a cluster but it is not the Master Node, we will
            automatically be redirected to the Master Node. You can configure a default
            host/IP to submit to with the command episodectl.exe proxy defaults --host
            <default host>.

      -c
  --cluster  A Cluster name to operate on. This will try to find a Cluster participant
            via Bonjour. You can configure a default cluster to submit to with the
            command episodectl.exe proxy defaults -c <default name>.

  --timeout  A timeout for the request. This command will connect to the Node through the
            ClientProxy and this timeout will be used in both those requests but it is
            unlikely that they will "add up". The default timeout is 30.
```

```
workflow recall [<workflow id>]
                [-i <template id>]
                [-n <name>]
                [--format xml|simple-xml|ascii|bin]
                [-o <directory>]
                [--print-plain-path]
                [--host <hostname/IP>]
                [-c <cluster name>]
                [--timeout <seconds>]


        Recall a submitted workflow from a node/cluster. This will write the recalled
        workflow as a submission file (.episubmission) including the source. The written
        file will be the same as if option --out was used in the command workflow submit ...
        --out and thus submittable again through option episodectl.exe workflow submit -s
        <recalled file>.

        If a <workflow id> is specified, the recalled data is the submission (and its template
        workflow) that spawned the started workflow with ID <workflow id>.

        This command can also be specified as wr, i.e. episodectl.exe wr


          -i
         --id  Specify that this ID is a Template ID.

          -n
        --name  Specify a name of the output file. This will not change the name of the
              actual submission/workflow. If no name is specified, the filename will be
              the same as the name of the recalled submission/workflow.

       --format  Format of the ouput configuration file. The argument can be any of xml, simple-xml
              , ascii, bin. The default is xml which is an XML format with type
              information. There is a simple-xml format that is more readable and easier
              to use for manual editing but use this with caution because it may not be
              upgradable/usable when a new version of Episode is released. ascii is a more
              compact format and more efficient to work with (IO, storage etc) and still
              kind of readble but not recommended for editing. bin is a binary format and
              about the same size as ascii but not readble nor editable. ascii and bin are
              therefore only recommended for "automatic" use, i.e. from a program that
              both creates them and submits them.

          -o
         --out  Specify the directory where the recalled workflow's submission file should
              be written. Default is current working directory.

          -p
--print-plain-path  Print the path to the written configuration file without any other text or
              newline etc. Usable if a program/script should run this command and read
              back the path.


        --host  A host Node to operate on (default is 127.0.0.1). If this Node is a
              participant of a cluster but it is not the Master Node, we will
```

automatically be redirected to the Master Node. You can configure a default host/IP to submit to with the command episodectl.exe proxy defaults --host <default host>.

-c
--cluster A Cluster name to operate on. This will try to find a Cluster participant via Bonjour. You can configure a default cluster to submit to with the command episodectl.exe proxy defaults -c <default name>.

--timeout A timeout for the request. This command will connect to the Node through the ClientProxy and this timeout will be used in both those requests but it is unlikely that they will "add up". The default timeout is 30.

status clusters [-p]
[-l [<separator>]]
[--timeout <seconds>]

Display info about the visible clusters on the local network by performing a Bonjour search. An asterisk character (*) indicates the Master node.

This command can also be specified as sc, i.e. episodectl.exe sc

-p
--private Include non-clustered nodes in the output, i.e. "Private" nodes.

-l
--list Print cluster names only, as a parse-friendly list with an optional <separator > character argument. The default is a "vertical bar", or "pipe" character ('|'). There are two special identifiers you can specify - "newline" or "LF" to get a newline separator.

--timeout A timeout for the request. This command will connect to the Node through the ClientProxy and this timeout will be used in both those requests but it is unlikely that they will "add up". The default timeout is 30.

status nodes [-c <cluster name>]
[--host <hostname/IP>]
[--timeout <seconds>]

Display info about the nodes in a cluster as well as current CPU and RAM usage. The configuration controlling the frequency of the updates is the <machine-metrics-update-interval> in the individual nodes' confiuration files. It is default set to 5 seconds (30 seconds before version 6.2) and you can change that interval for more frequent updates. The machine metrics information is also the base for the job scheduling algorithm called "Load Balancing" which will also be more accurate with shorter update intervals.

This command can also be specified as sn, i.e. episodectl.exe sn

--host A host Node to operate on (default is 127.0.0.1). If this Node is a
participant of a cluster but it is not the Master Node, we will
automatically be redirected to the Master Node. You can configure a default
host/IP to submit to with the command episodectl.exe proxy defaults --host
<default host>.

-c
--cluster A Cluster name to operate on. This will try to find a Cluster participant
via Bonjour. You can configure a default cluster to submit to with the
command episodectl.exe proxy defaults -c <default name>.

--timeout A timeout for the request. This command will connect to the Node through the
ClientProxy and this timeout will be used in both those requests but it is
unlikely that they will "add up". The default timeout is 30.

-w
--wait This will update and display the status with certain intervals. This will
not exit so you will have to manually stop the program.

status workflows [<id> ...]
[-i <template id>]
[--all [history [request]]]
[-s <separator char>]
[--status-out [text]]
[-v]
[-c <cluster name>]
[--host <hostname/IP>]
[--timeout <seconds>]

Get status for one or more Started Workflows. The IDs are supposed to have been
obtained from a call to workflow submit ... --ids-out unless option -i is used where
an id obtained from workflow submit ... --id-out should be specified. If neither of
these are specified, this is the default behaviour episodectl.exe sw --all.
Each status is printed in the order their ID was specified. A value of 0 means that
the workflow is finished and was successful. A value of 1 is a general error code. 2
means that the workflow failed. 3 means that it's still running and 4 means that
it's idle and can be treated as running. Idle is most often that some tasks in the
workflow are currently queued.

This command can also be specified as sw, i.e. episodectl.exe sw

-i
--id If you have obtained a template ID from a submission with option --id-out,
you can specify that to recieve status for all workflows originating from
that single template ID.
With this option this command will return special program return codes.

If ALL workflows originating from the specified ID has finished and was
successful, 0 is returned.

If ANY of the workflows has failed, 2 is returned.

If ANY of the workflows are still running and none has failed, 3 is returned.

If ANY of the workflows are idle and none has failed, 4 is returned.

Idle in this case is most often when tasks in the workflow are queued and can be treated as if the workflow is still running. This will only return status for workflows which have not yet expired from the ClientProxy's history. i.e. the ClientProxy will not query the Node's history for status.

This means that the returned status can change from, for example 2 (failed) to 3 or 4 IF a failed workflow expires from history before other workflows from the same template are still idle or running.

For example, if the ClientProxy has a configured history keep time of 1 hour, a submission is made at 12:00 which spawns 2 workflows, one of which immediately fails. The other one is (for some reason) queued (idle) for over 1 hour. At 13:00 the returned status will go from 2 to 3 because only the idle workflow is still in the history cache. It is therefore recommended to take immediate action on failure if this option is used!

--all This option lists all "active" workflows, i.e. status 3 or 4. The workflow ID will be printed aswell as the status. The format is "workflow-id=status-format".
You may optionally include the results of all historized workflows with the argument "history" in which case a plus-sign character is printed after the status which indicates that more info may be retrieved for this workflow. The history in this case is the history of the EpisodeClientProxy.exe process which is only in memory of that process and not stored on disk. This history cache-time is configurable in ClientProxy.xml.

You can add a second argument "request" to make a request to the node's database to see which workflows are still stored in the node's database. NOTE: Using "request" may be a significant overhead.

The result of the workflow, i.e. status 0 or 2, is saved as long as the node's database is not manually removed. What this means is that you can list workflow "results" (status) for workflows that have been cleaned out automatically after the configurated time to keep history. In other words, you will see a lot of workflows listed here that there is no way to get further info about than the actual status, or result. It's up to the user to maintain a description of these "Jobs" outside of the Episode system if needed.

-s
--separator Specify a separator character. The default is a newline character.

--status-out Print status instead of using return values. The printed number is the ASCII equivalent of the return codes. An optional argument "text" can be specified to print the status as text.

Status code	Text string
0	"Succeeded"
1	"Error" (CLI general error)
2	"Failed"

```
                3           "Running"
                4           "Idle"

        -v
    --visual  Shows a human readable overview output.

      --host  A host Node to operate on (default is 127.0.0.1). If this Node is a
              participant of a cluster but it is not the Master Node, we will
              automatically be redirected to the Master Node. You can configure a default
              host/IP to submit to with the command episodectl.exe proxy defaults --host
              <default host>.

        -c
    --cluster  A Cluster name to operate on. This will try to find a Cluster participant
              via Bonjour. You can configure a default cluster to submit to with the
              command episodectl.exe proxy defaults -c <default name>.

    --timeout  A timeout for the request. This command will connect to the Node through the
              ClientProxy and this timeout will be used in both those requests but it is
              unlikely that they will "add up". The default timeout is 30.



status tasks [<workflow id> ...]
            [-i <template id>]
            [--all [history|history-only [request]]]
            [--task-separator <separator char>]
            [--task-property-separator <separator char>]
            [--out [<key> ...]]
            [--filter-type <task type name> ...]
            [--filter-name <task user name> ...]
            [-w]
            [-v]
            [--host <hostname/IP>]
            [-c <cluster name>]
            [--timeout <seconds>]


        Get status for one or more tasks inside one or more workflows. There are two
        different separators to make it easy to parse the output.
        The first separator is for separating the individual tasks, and the other is for
        separating the properties of the task requested.

        Note for 6.2.x users, The synopsis and documentation for this command is completely
        new. The <key>s to option --out replaces all the old --xxx-out options as well as
        "variables". All the old options are still working but it is recommended to use the
        new --out option as soon as possible.


        This command can also be specified as st, i.e. episodectl.exe st

        If no option is specified, this is the default behaviour episodectl.exe st --all.


          -i
        --id  If you have obtained a template ID from a submission with option --id-out,
              you may specify that to recieve status for all tasks originating from that
```

single template ID. This will only return status for workflows which have
not yet expired from the ClientProxy's history. i.e. the ClientProxy will
not query the Node's history for status.

--all This will return status for all "active" (Idle, Queued, Running) tasks. If
argument "history" is specified, finished (historized) tasks will also be
shown.
The history in this case is the history of the EpisodeClientProxy.exe
process which is only in memory of that process and not stored on disk. This
history cache-time is configurable in ClientProxy.xml.
You can add a second argument "request" to make a request to the node's
database to retreive status for tasks that are still stored in the node's
database. NOTE: Using "request" may be a significant overhead. To view only
history, the history argument can be "history-only".

--task-separator
--ts Specify a separator character to be used to separate group of specified
"task properties". The default is a newline character.

--task-property-separator
--tps Specify a separator character to be used to separate each specified task
property. The default is a "vertical bar", or "pipe" character ('|'). There
are two special identifiers you can specify - "newline" or "LF" to get a
newline separator.

--filter-type Filter the output based on the name of the type of task. This is supposed to
be one of the task type names printed with option --name-out. Several type
names may be specified.

--filter-name Filter the output based on the user defined name of the task. Several names
may be specified.

-w
--wait Waits indefinitely (until the program is manually stopped) and polls status
for the tasks with a certain interval. This implies option -v which means
that it shows a human readable overview output.

-v
--visual Shows a human readable overview output. Running tasks' status is a progress
bar showing progress with a dash character '-' if it is not the last pass
and a equality character '=' when it is the last pass (The dash only applies
to Encode tasks configured with 2-pass/Multipass). Paused tasks are
indicated with double vertical bars, or pipe characters as a prefix to
either the status or the progress bar, e.g.
||[==========   ]
|| Queued
If used in conjunction with option --wait, the output is truncatated
vertically to fit the terminal/console window.

TIP: To get the best overview of Episode task status, get a computer with a
wide screen, create 2 terminal/console windows that are maximized vertically
and shares the screen horizontally. Then execute:
episodectl.exe st -c MyCluster --all -wv in the first window and
episodectl.exe st -c MyCluster --all history-only -wv in the second window.

--host A host Node to operate on (default is 127.0.0.1). If this Node is a
participant of a cluster but it is not the Master Node, we will

automatically be redirected to the Master Node. You can configure a default host/IP to submit to with the command episodectl.exe proxy defaults --host <default host>.

-c
--cluster A Cluster name to operate on. This will try to find a Cluster participant via Bonjour. You can configure a default cluster to submit to with the command episodectl.exe proxy defaults -c <default name>.

--timeout A timeout for the request. This command will connect to the Node through the ClientProxy and this timeout will be used in both those requests but it is unlikely that they will "add up". The default timeout is 30.

-o
--out This option takes one or more space separated <key>s, where each <key> is a keyword from the list below. Many keywords has the same name as the variables (episodectl.exe variables), without the surrounding dollar characters.

The output is printed in the same order as they are specified on the command line. If no option is specified, all <key>s listed below are printed in the format '<key>=value|<key>=value' etc (for default <separator char>). If all <key>s are printed, the order is undefined.

The keywords that outputs a time will print the time as a "Unix time" or "POSIX time". It is the number of seconds passed since 00:00:00 UTC, January 1, 1970. In the current timezone, the start time was Wed Dec 31 16:00:00 1969. The current POSIX time is 1432052485. See http://en.wikipedia.org/wiki/Unix_time for more information.

task.status.code
task.status.text

Current status code of the task as a number or as text, here is a list of each status:

Status code	Text string
0	"Idle"
1	"Queued"
2	"Submitted"
3	"PreStartFail"
4	"Running"
5	"Cancelled"
6	"Succeeded"
7	"Failed"
8	"Redundant"

task.scheduling-state

Current scheduling state of the task. A value of 0 means schedule (for execution, i.e. default), 1 means paused (do not schedule for execution).

task.progress

Current progress of the task as a integer value from 0 to 100

task.progress.float

Current progress of the task as a floating point value from 0.0 to 1.0

task.progress.mp

Multipass information. If multipass is not configured or the task is not an encoder task, the output will be "-". If multipass is enabled but it's not the last pass, the pass will be printed as a number followed by a "+"-sign, indicating there are more passes coming. When the encoder is doing the last pass, the pass will be printed as a number without a following "+"-sign. For example, when doing a 2-pass encoding, the output for the first pass will be "1+" and for the second pass "2".

task.id

The unique task ID inside the workflow. This ID can be used in calls to job cancel/requeue/pause/resume/set-priority together with the workflow ID, workflow.id.

task.name

The name of the task. In case of Split-and-Stitch or EDL tasks (dynamically created tasks), this name is a dynamically generated name suitable for human readbility, i.e a GUI. To identify a dynamically created task by (user defined) name, use task.user-name.

task.user-name

The user defined name of the task. In case of Split-and-Stitch or EDL tasks (dynamically created tasks), this name will be the same name as the "original" task name that created the task.

task.type-name

The type name of the task. Here is the current list of task type names:

Localize
Encode
MBR
Transfer
Move
Delete
YouTube

Execute
Mail

task.priority

The current priority of the task. This will initially be the same as workflow.priority but may be changed after the workflow is submitted with a call to job set-priority.
Developer note: 64-bit signed integer value.

task.message

The current message of the task. There is currently no definition of what this message may be, but in case of task failure, it will be a error message. In case of status Idle, it will tell why it is idle and so on. In a GUI, this may be displayed at all times.

task.node-id

The ID of the node that is currently running the task, or has the task queued, idle, etc. This ID is default configured to be the hostname of the machine. If the task is done, this will be the ID of the node that executed it (the last time in case of several attemps).

task.attempt

The current attempt to execute the task. This will be 0 if the task has not been run yet.

task.start-time

The time the task was scheduled for execution, i.e. started. If the task hasn't been started yet, this time will be 0. If this task is re-queued (manually, by Episode re-start, or by failure-re-run attempt) this will show the latest time it was started.

task.end-time

The time the task was finished. If the task hasn't finished yet, this time will be 0. If this task was re-queued (by failure-re-run attempt) this will show the latest time it failed.

workflow.status.code
workflow.status.text

Current status of the workflow as a status code or text.

Notice: This differs from the status of command status workflows, the reason is that the status is used as a program exit code in that

command and therefore needs to be altered, while that is not the case here.

Statuses:

Status code	Text string
0	"Idle"
1	"Running"
2	"Succeeded"
3	"Failed"

workflow.priority

Current priority of the workflow.
Developer note: 64-bit signed integer value.

workflow.end-time

The time the workflow finished. If the workflow hasn't finished yet, this time will be 0.

Additional (static) workflow information can be obtained through these keywords, see episodectl.exe variables for more information.

workflow.submission-name workflow.submission-time
workflow.submission-client workflow.submission-host
workflow.template-id workflow.id
workflow.spawn-value workflow.spawn-time
workflow.name workflow.seq-nr

Source information can be obtained through these keywords, see episodectl.exe variables for more information.

source.name source.type
source.url source.path
source.file
source.filename source.extension
source.parent-dir-name source.parent-dir-path
source.grandparent-dir-name source.grandparent-dir-path

If task.type-name equals Encode, the following keywords are available after the Encode task has finished successfully. If they are read under other circumstances the value will be empty. See episodectl.exe variables for more information.

encoder.name (same as task.user-name and always available)
encoder.input-duration-s encoder.input-duration-hms
encoder.input-pixel-dimensions
encoder.input-framerate encoder.input-samplerate
encoder.output-bitrate-kbps encoder.output-bitrate-mbps
encoder.output-duration-s encoder.output-duration-hms

encoder.output-pixel-dimensions
encoder.output-framerate encoder.output-samplerate

If task.type-name equals MBR, the following keywords are available after the MBR task has finished successfully. If they are read under other circumstances the value will be empty. See episodectl.exe variables for more information.

mbr.name (same as task.user-name and always available)
mbr.package-type mbr.package-name

If task.type-name equals Transfer, the following keywords are available after the Transfer task has finished successfully. If they are read under other circumstances the value will be empty. See episodectl.exe variables for more information.

deployment.name (same as task.user-name and always available)
deployment.dest-url deployment.outfile-path
deployment.outfile-name deployment.outfile-extension
deployment.outfile-file

If task.type-name equals YouTube, the following keywords are available.

deployment.name

If task.type-name equals Execute, the following keyword is available after the Execute task has finished successfully. If it is read under other circumstances the value will be empty.

execute.exit-code

The exit code of the program/script that was executed.

status watch-folders [--watch-folder-separator <separator char>]
[--watch-folder-property-separator <separator char>]
[--out [<key> ...]]
[-v]
[--host <hostname/IP>]
[-c <cluster name>]
[--timeout <seconds>]

Get status for all watch folders. There are two different separators to make it easy to parse the output.

The first separator is for separating the individual watch folders, and the other is for separating the properties of each watch folder.

This command can also be specified as sm, i.e. episodectl.exe sm

--watch-folder-separator
--ms Specify a separator character to be used to separate group of specified "monitor properties". The default is a newline character.

--watch-folder-property-separator
--mps Specify a separator character to be used to separate each specified watch folder property. The default is a "vertical bar", or "pipe" character ('|'). There are two special identifiers you can specify - "newline" or "LF" to get a newline separator.

-v
--visual Shows a human readable overview output.

--host A host Node to operate on (default is 127.0.0.1). If this Node is a participant of a cluster but it is not the Master Node, we will automatically be redirected to the Master Node. You can configure a default host/IP to submit to with the command episodectl.exe proxy defaults --host <default host>.

-c
--cluster A Cluster name to operate on. This will try to find a Cluster participant via Bonjour. You can configure a default cluster to submit to with the command episodectl.exe proxy defaults -c <default name>.

--timeout A timeout for the request. This command will connect to the Node through the ClientProxy and this timeout will be used in both those requests but it is unlikely that they will "add up". The default timeout is 30.

-o
--out This option takes one or more space separated <key>s, where each <key> is a keyword from the list below.

The output is printed in the same order as they are specified on the command line. If no option is specified, all <key>s listed below are printed in the format '<key>=value|<key>=value' etc (for default <separator char>). If all <key>s are printed, the order is undefined.

The keywords that outputs a time will print the time as a "Unix time" or "POSIX time". It is the number of seconds passed since 00:00:00 UTC, January 1, 1970. In the current timezone, the start time was Wed Dec 31 16:00:00 1969. The current POSIX time is 1432052485. See http://en.wikipedia.org/wiki/Unix_time for more information.

id

The unique watch folder ID.

name

User defined watch folder name.

url

The URL that this watch folder is monitoring.

workflow-id

The template workflow that the watch folder is targeting.

workflow-priority

The priority of the targeted template workflow.
Developer note: 64-bit signed integer value.

running

If the watch folder is currently running (monitoring). The printed value will be yes or no.

stop-reason

The reason why the watch folder is not running (stopped). If the monitor is running, this will be a empty value.

submission-time

The time this watch folder (and its template workflow) was submitted.

submission-name

The name of the submission.

submission-client

The client which made the submission.

submission-host

The host from which the submission came.

job

With the job commands, tasks and workflows can be operated on in run-time, i.e. after they've been submitted. All job commands have different options that takes IDs as arguments and here is a description of how to obtain these IDs

<workflow id>

episodectl.exe workflow submit ... --ids-out
episodectl.exe status workflows --all
episodectl.exe status tasks ... --out workflow.id
When printing everything with
episodectl.exe status tasks ... --out,
it is printed as "workflow.id=<workflow id>"

<task id>

episodectl.exe status tasks ... --out task.id
When printing everything with
episodectl.exe status tasks ... --out,
it is printed as "task.id=<task id>"

<template id>

episodectl.exe workflow submit ... --id-out
episodectl.exe status tasks ... --out workflow.template-id
When printing everything with
episodectl.exe status tasks ...,
it is printed as "workflow.template-id=<template id>"

<workflow name>

Specified through submission with
episodectl.exe workflow submit ... --workflow-name <workflow name>
episodectl.exe status tasks ... --out workflow.name

job cancel [-t <workflow id> <task id>]
[-w <workflow id>]
[-i <template id>]
[-n <workflow name>]
[--all]
[-c <cluster name>]
[--host <hostname/IP>]
[--timeout <seconds>]

Cancel a individual task, or remove one or more workflows in which case any associated tasks are cancelled. One of the options -t, -w, -i, -n, or --all must be specified and each option below describes the effects of the option. Commands for obtaining the various IDs are:

This command can also be specified as jcan, i.e. episodectl.exe jcan

```
        -t
    --task  Cancel and remove a individual task. The workflow that this task is a part
            of is not affected except for this task. If the task is in a running state,
            it is stopped first.

        -w
  --workflow  Cancel and remove a workflow. All running tasks (if any) are stopped before
            removal. This is the same as episodectl.exe workflow stop <workflow id>.

        -i
     --id  Cancel and remove all workflows originating from this ID. All running tasks
            (if any) are stopped before removal. This is the same as episodectl.exe
            workflow stop -i <template id>.

        -n
    --name  Cancel and remove all workflows that have the name <workflow name>. All
            running tasks (if any) are stopped before removal.

     --all  Cancel and remove all workflows. All running tasks are stopped before
            removal. This is the same as episodectl.exe workflow stop --all.

    --host  A host Node to operate on (default is 127.0.0.1). If this Node is a
            participant of a cluster but it is not the Master Node, we will
            automatically be redirected to the Master Node. You can configure a default
            host/IP to submit to with the command episodectl.exe proxy defaults --host
            <default host>.

        -c
  --cluster  A Cluster name to operate on. This will try to find a Cluster participant
            via Bonjour. You can configure a default cluster to submit to with the
            command episodectl.exe proxy defaults -c <default name>.

  --timeout  A timeout for the request. This command will connect to the Node through the
            ClientProxy and this timeout will be used in both those requests but it is
            unlikely that they will "add up". The default timeout is 30.



job requeue [-t <workflow id> <task id>]
        [-w <workflow id>]
        [-i <template id>]
        [-n <workflow name>]
        [--all]
        [-c <cluster name>]
        [--host <hostname/IP>]
        [--timeout <seconds>]


    Stop and re-queue one or more tasks.One of the options -t, -w, -i, -n, or --all must
    be specified and each option below describes the effects of the option.

    Notice that re-queuing tasks does not mean that another task is going to start
    without previously having either increased another task's priority, decreased the
    soon-to-be-re-queued task's priority, or new tasks have been submitted with higher
    priority, see explanation below.
```

There are many parameters taken into consideration when the Node schedules tasks for execution. First of all, a sequential number is given to each task that leaves the "idle" state and enters the "queued" state. Then there are priority and a number of requirements. Requirements includes license requirements, user defined requirements such as Tags, and finally platform requirements for certain encode formats. If many tasks meet the same scheduling requirements and have the same priority, the initial sequnce number will decide which task will be sheduled to run. Since the priority is the only scheduling parameter that can be changed after a submission is done, it must be changed on some task in order to avoid the "default" re-queue behavior that basically equals a re-start.

This command can also be specified as jrq, i.e. episodectl.exe jrq

-t
--task Re-queue a individual task. This will only affect a running task.

-w
--workflow Re-queue all running tasks in the workflow with ID <workflow id>.

-i
--id Re-queue all running tasks in workflows originating from this ID.

-n
--name Re-queue all running tasks in workflows that has the name <workflow name>.

--all Re-queue all running tasks.

--host A host Node to operate on (default is 127.0.0.1). If this Node is a participant of a cluster but it is not the Master Node, we will automatically be redirected to the Master Node. You can configure a default host/IP to submit to with the command episodectl.exe proxy defaults --host <default host>.

-c
--cluster A Cluster name to operate on. This will try to find a Cluster participant via Bonjour. You can configure a default cluster to submit to with the command episodectl.exe proxy defaults -c <default name>.

--timeout A timeout for the request. This command will connect to the Node through the ClientProxy and this timeout will be used in both those requests but it is unlikely that they will "add up". The default timeout is 30.

job pause [-t <workflow id> <task id>]
[-w <workflow id> [queued-only]]
[-i <template id> [queued-only]]
[-n <workflow name> [queued-only]]
[--all [queued-only]]
[-c <cluster name>]
[--host <hostname/IP>]
[--timeout <seconds>]

Pause one or more tasks.

Pausing a task means that its scheduling state is set in paused mode, meaning that every queued tasks can be paused, not just running tasks. If the task being paused is currently running, the actual process (EpisodeWorker.exe) is paused. If the task being paused is queued, it will not be scheduled for execution until it is resumed again.

Pausing running tasks does not free any job slots or other resources and is probably only useful for Desktop users to temporarily free CPU resources on the computer.

However, by pausing all non-running tasks, a cluster can be left to process all currently running tasks and concurrently be instructed to not schedule any new tasks, meaning that the cluster can be taken down for maintenance or re-configuration in a graceful way (no loss of semi-processed data) when the currently running tasks are finished, and any deployments and/or post-deployments are finished. When pausing is only done to queued tasks, all "branches" in the workflow (each Encoder has it's own branch) that has started will finish (since idle tasks are not paused), which means that there will not be any dependencies on temporary files (files in the Episode File Cache) when re-starting the cluster. When the cluster is brought back up, use episodectl.exe job resume --all to start task scheduling again.

One of the options -t, -w, -i, -n, or --all must be specified and each option below describes the effects of the option.

This command can also be specified as jpau, i.e. episodectl.exe jpau

-t
--task Pause a individual task, regardless of status.

-w
--workflow Pause all tasks in the workflow with ID <workflow id>. Only idle, submitted, running, or queued tasks will be affected. If the optional argument queued-only is specified, only queued tasks are paused.

-i
--id Pause all tasks in workflows originating from this ID. Only idle, submitted, running, or queued tasks will be affected. If the optional argument queued-only is specified, only queued tasks are paused.

-n
--name Pause all tasks in workflows that has the name <workflow name>. Only idle, submitted, running, or queued tasks will be affected. If the optional argument queued-only is specified, only queued tasks are paused.

--all Pause tasks in all active workflows. Only idle, submitted, running, or queued tasks will be affected. If the optional argument queued-only is specified, only queued tasks are paused.

--host A host Node to operate on (default is 127.0.0.1). If this Node is a participant of a cluster but it is not the Master Node, we will automatically be redirected to the Master Node. You can configure a default host/IP to submit to with the command episodectl.exe proxy defaults --host <default host>.

-c

```
    --cluster  A Cluster name to operate on. This will try to find a Cluster participant
            via Bonjour. You can configure a default cluster to submit to with the
            command episodectl.exe proxy defaults -c <default name>.

    --timeout  A timeout for the request. This command will connect to the Node through the
            ClientProxy and this timeout will be used in both those requests but it is
            unlikely that they will "add up". The default timeout is 30.



job resume [-t <workflow id> <task id>]
        [-w <workflow id>]
        [-i <template id>]
        [-n <workflow name>]
        [--all]
        [-c <cluster name>]
        [--host <hostname/IP>]
        [--timeout <seconds>]


      Resume one or more paused tasks.One of the options -t, -w, -i, -n, or --all must be
      specified and each option below describes the effects of the option.
      Resuming tasks will only affect tasks that are paused.

      See episodectl.exe job pause -h for a description of how to obtain the various IDs.

      This command can also be specified as jres, i.e. episodectl.exe jres


        -t
      --task  Resume a individual task. This will only affect a paused task.

        -w
    --workflow  Resume all paused tasks in the workflow with ID <workflow id>.

        -i
     --id  Resume all paused tasks in workflows originating from this ID.

        -n
      --name  Resume all paused tasks in workflows that has the name <workflow name>.

       --all  Resume all paused tasks.

      --host  A host Node to operate on (default is 127.0.0.1). If this Node is a
            participant of a cluster but it is not the Master Node, we will
            automatically be redirected to the Master Node. You can configure a default
            host/IP to submit to with the command episodectl.exe proxy defaults --host
            <default host>.

        -c
    --cluster  A Cluster name to operate on. This will try to find a Cluster participant
            via Bonjour. You can configure a default cluster to submit to with the
            command episodectl.exe proxy defaults -c <default name>.

    --timeout  A timeout for the request. This command will connect to the Node through the
            ClientProxy and this timeout will be used in both those requests but it is
            unlikely that they will "add up". The default timeout is 30.
```

```
job set-priority [-p] <priority>
                 [--whence <zero|original|current>]
                 [-t <workflow id> <task id>]
                 [-w <workflow id>]
                 [-i <template id>]
                 [-n <workflow name>]
                 [-c <cluster name>]
                 [--host <hostname/IP>]
                 [--timeout <seconds>]


      Set new priority on one or more tasks. One of the options -t, -w, -i, or -n must be
      specified and each option below describes the effects of the option. Commands for
      obtaining the various IDs are:

      This command can also be specified as jprio, i.e. episodectl.exe jprio

      Developer note: Priority is a signed 64-bit integer value.


    --whence  Set absolute value (zero) or adjust relative original or current priority. zero
              equals <priority>, original equals <priority> + task priority before workflow
              spawning, and current equals <priority> + current task priority. Default is original
              .

        -t
     --task  Set/adjust priority on a individual task.

        -w
  --workflow  Set/adjust priority on all tasks belonging to the workflow with ID "workflow-id
              ".

        -i
      --id  Set/adjust priority on all tasks belonging to any workflows spawned from
              this template ID.

        -n
    --name  Set/adjust priority on tasks belonging to any workflows with name <workflow-name
              >.

    --host  A host Node to operate on (default is 127.0.0.1). If this Node is a
              participant of a cluster but it is not the Master Node, we will
              automatically be redirected to the Master Node. You can configure a default
              host/IP to submit to with the command episodectl.exe proxy defaults --host
              <default host>.

        -c
 --cluster  A Cluster name to operate on. This will try to find a Cluster participant
              via Bonjour. You can configure a default cluster to submit to with the
              command episodectl.exe proxy defaults -c <default name>.

 --timeout  A timeout for the request. This command will connect to the Node through the
              ClientProxy and this timeout will be used in both those requests but it is
              unlikely that they will "add up". The default timeout is 30.
```

monitor

Monitors can be added to a Node or Cluster with the workflow submit command. With the monitor command, you can start, stop or remove these. You can also see log messages from them with the log subcommand which may be useful for debugging purposes.
With the monitor subcommands, the monitors are always attached to an existing Template Workflow (See workflow subcommand for a description of Template Workflows).
You can either control a monitor by its unique ID automatically given to it when attached, or you can choose to give it a name and reference it by that name.
The name must not be unique and if you reference monitors by name, all monitors with that name will be affected.
This means that you can control a group of monitors by giving them the same name.

monitor start [-i <monitor ID>] [-n <name>]
[--all]
[--host <hostname/IP>]
[--cluster <name>]
[--timeout <seconds>]

Start one or more existing monitor(s). This will have no effect if it's already running. One of the options -i, -n, or --all must be specified.

This command can also be specified as ms, ie. episodectl.exe ms

-i
--id A monitor ID to start.

-n
--name A monitor name to start. This may affect multiple monitors if they are named the same.

-a
--all Start all monitors that are not running.

--host A host Node to operate on (default is 127.0.0.1). If this Node is a participant of a cluster but it is not the Master Node, we will automatically be redirected to the Master Node. You can configure a default host/IP to submit to with the command episodectl.exe proxy defaults --host <default host>.

-c
--cluster A Cluster name to operate on. This will try to find a Cluster participant via Bonjour. You can configure a default cluster to submit to with the command episodectl.exe proxy defaults -c <default name>.

-w
--wait Do not exit program. This will print out any monitor log messages as they arrive. You have to manually stop the program.

--timeout A timeout for the request. This command will connect to the Node through the ClientProxy and this timeout will be used in both those requests but it is unlikely that they will "add up". The default timeout is 30.

monitor set-priority <priority>
[-i <monitor ID>]
[-n <name>]
[--all]
[--host <hostname/IP>]
[--cluster <name>]
[--timeout <seconds>]

Set priority on the template workflow that this monitor is attached to. The priority will be used as "initial task priority adjustment" for all workflows spawned from this template workflow (every file reported/submitted by this monitor. One of the options -i, -n, or --all must be specified.
Developer note: Priority is a signed 64-bit integer value.

This command can also be specified as mprio, ie. episodectl.exe mprio

-i
--id A monitor ID.

-n
--name A monitor name. This may affect multiple monitors if they are named the same.

-a
--all Set priority on all monitors.

--host A host Node to operate on (default is 127.0.0.1). If this Node is a participant of a cluster but it is not the Master Node, we will automatically be redirected to the Master Node. You can configure a default host/IP to submit to with the command episodectl.exe proxy defaults --host <default host>.

-c
--cluster A Cluster name to operate on. This will try to find a Cluster participant via Bonjour. You can configure a default cluster to submit to with the command episodectl.exe proxy defaults -c <default name>.

-w
--wait Do not exit program. This will print out any monitor log messages as they arrive. You have to manually stop the program.

--timeout A timeout for the request. This command will connect to the Node through the ClientProxy and this timeout will be used in both those requests but it is unlikely that they will "add up". The default timeout is 30.

monitor stop [-i <monitor ID>] [-n <name>]

```
        [--all]
        [--host <hostname/IP>]
        [--cluster <name>]
        [--timeout <seconds>]
```

Stop one or more existing monitor(s). This will have no effect if it's not running.
One of the options -i, -n, or --all must be specified.

This command can also be specified as mp, ie. episodectl.exe mp

```
   -i
 --id  A monitor ID to stop.

   -n
--name  A monitor name to stop. This may affect multiple monitors if they are named
        the same.

   -a
 --all  Stop all monitors that are running.

--host  A host Node to operate on (default is 127.0.0.1). If this Node is a
        participant of a cluster but it is not the Master Node, we will
        automatically be redirected to the Master Node. You can configure a default
        host/IP to submit to with the command episodectl.exe proxy defaults --host
        <default host>.

   -c
--cluster  A Cluster name to operate on. This will try to find a Cluster participant
        via Bonjour. You can configure a default cluster to submit to with the
        command episodectl.exe proxy defaults -c <default name>.

--timeout  A timeout for the request. This command will connect to the Node through the
        ClientProxy and this timeout will be used in both those requests but it is
        unlikely that they will "add up". The default timeout is 30.
```

```
monitor remove [-i <monitor ID>] [-n <name>]
        [--all]
        [--host <hostname/IP>]
        [--cluster <name>]
        [--timeout <seconds>]
```

Remove one or more existing monitor(s). The monitor(s) will be stopped (if running)
and permanently removed. The template workflow that this monitor is attached to will
also be removed. One of the options -i, -n, or --all must be specified.

This command can also be specified as mr, ie. episodectl.exe mr

```
   -i
 --id  A monitor ID to remove.

   -n
```

--name A monitor name to remove. This may affect multiple monitors if they are named the same.

-a
--all Remove all monitors.

--host A host Node to operate on (default is 127.0.0.1). If this Node is a participant of a cluster but it is not the Master Node, we will automatically be redirected to the Master Node. You can configure a default host/IP to submit to with the command episodectl.exe proxy defaults --host <default host>.

-c
--cluster A Cluster name to operate on. This will try to find a Cluster participant via Bonjour. You can configure a default cluster to submit to with the command episodectl.exe proxy defaults -c <default name>.

--timeout A timeout for the request. This command will connect to the Node through the ClientProxy and this timeout will be used in both those requests but it is unlikely that they will "add up". The default timeout is 30.

monitor log [-i <monitor ID>] [-n <name>]
[--host <hostname/IP>]
[--cluster <name>]
[--timeout <seconds>]

Recieve log messages from one or more existing monitor(s). This command will not exit, thus having to be manually stopped/breaked. If no option is given, all running monitors' log messages will be printed. The configuration variable controlling the level/verbosity of the messages is the one in the Node.xml configuration file in the 'monitors' section inside the 'logging' section. If you want to see debug output from a monitor with this command, you can use this command to set monitors to report at debug level: episodectl.exe node log --monitors no 7 (the 'no' is no, we don't want to log to file). You will have to either stop/start the monitor or restart the node after re-configuration. One of the options -i, -n, or --all must be specified.

This command can also be specified as mg, ie. episodectl.exe mg

-i
--id A monitor ID to recieve log messages from.

-n
--name A monitor name to recieve log messages from. This may affect multiple monitors if they are named the same.

--host A host Node to operate on (default is 127.0.0.1). If this Node is a participant of a cluster but it is not the Master Node, we will automatically be redirected to the Master Node. You can configure a default host/IP to submit to with the command episodectl.exe proxy defaults --host <default host>.

-c
--cluster A Cluster name to operate on. This will try to find a Cluster participant

via Bonjour. You can configure a default cluster to submit to with the
command episodectl.exe proxy defaults -c <default name>.

--timeout A timeout for the request. This command will connect to the Node through the
ClientProxy and this timeout will be used in both those requests but it is
unlikely that they will "add up". The default timeout is 30.

watch-folder

Watch folders can be added to a Node or Cluster with the workflow submit command. With the watch-folder command, you can start, stop or remove these. You can also see log messages from them with the log subcommand which may be useful for debugging purposes.
With the watch-folder subcommands, the watch folders are always attached to an existing Template Workflow (See workflow subcommand for a description of Template Workflows).
You can either control a watch folder by its unique ID automatically given to it when attached, or you can choose to give it a name and reference it by that name.
The name must not be unique and if you reference watch folders by name, all watch folders with that name will be affected.
This means that you can control a group of watch folders by giving them the same name.

watch-folder start [-i <watch folder ID>] [-n <name>]
[--all]
[--host <hostname/IP>]
[--cluster <name>]
[--timeout <seconds>]

Start one or more existing watch folder(s). This will have no effect if it's already running. One of the options -i, -n, or --all must be specified.

This command can also be specified as ms, ie. episodectl.exe ms

-i
--id A watch folder ID to start.

-n
--name A watch folder name to start. This may affect multiple watch folders if they
are named the same.

-a
--all Start all watch folders that are not running.

--host A host Node to operate on (default is 127.0.0.1). If this Node is a
participant of a cluster but it is not the Master Node, we will
automatically be redirected to the Master Node. You can configure a default
host/IP to submit to with the command episodectl.exe proxy defaults --host
<default host>.

-c
--cluster A Cluster name to operate on. This will try to find a Cluster participant

via Bonjour. You can configure a default cluster to submit to with the command episodectl.exe proxy defaults -c <default name>.

-w
--wait Do not exit program. This will print out any watch folder log messages as they arrive. You have to manually stop the program.

--timeout A timeout for the request. This command will connect to the Node through the ClientProxy and this timeout will be used in both those requests but it is unlikely that they will "add up". The default timeout is 30.

watch-folder set-priority <priority>
[-i <watch folder ID>]
[-n <name>]
[--all]
[--host <hostname/IP>]
[--cluster <name>]
[--timeout <seconds>]

Set priority on the template workflow that this watch folder is attached to. The priority will be used as "initial task priority adjustment" for all workflows spawned from this template workflow (every file reported/submitted by this watch folder. One of the options -i, -n, or --all must be specified.
Developer note: Priority is a signed 64-bit integer value.

This command can also be specified as mprio, ie. episodectl.exe mprio

-i
--id A watch folder ID.

-n
--name A watch folder name. This may affect multiple watch folders if they are named the same.

-a
--all Set priority on all watch folders.

--host A host Node to operate on (default is 127.0.0.1). If this Node is a participant of a cluster but it is not the Master Node, we will automatically be redirected to the Master Node. You can configure a default host/IP to submit to with the command episodectl.exe proxy defaults --host <default host>.

-c
--cluster A Cluster name to operate on. This will try to find a Cluster participant via Bonjour. You can configure a default cluster to submit to with the command episodectl.exe proxy defaults -c <default name>.

-w
--wait Do not exit program. This will print out any watch folder log messages as they arrive. You have to manually stop the program.

--timeout A timeout for the request. This command will connect to the Node through the

ClientProxy and this timeout will be used in both those requests but it is unlikely that they will "add up". The default timeout is 30.

watch-folder stop [-i <watch folder ID>] [-n <name>]
[--all]
[--host <hostname/IP>]
[--cluster <name>]
[--timeout <seconds>]

Stop one or more existing watch folder(s). This will have no effect if it's not running. One of the options -i, -n, or --all must be specified.

This command can also be specified as mp, ie. episodectl.exe mp

-i
--id A watch folder ID to stop.

-n
--name A watch folder name to stop. This may affect multiple watch folders if they are named the same.

-a
--all Stop all watch folders that are running.

--host A host Node to operate on (default is 127.0.0.1). If this Node is a participant of a cluster but it is not the Master Node, we will automatically be redirected to the Master Node. You can configure a default host/IP to submit to with the command episodectl.exe proxy defaults --host <default host>.

-c
--cluster A Cluster name to operate on. This will try to find a Cluster participant via Bonjour. You can configure a default cluster to submit to with the command episodectl.exe proxy defaults -c <default name>.

--timeout A timeout for the request. This command will connect to the Node through the ClientProxy and this timeout will be used in both those requests but it is unlikely that they will "add up". The default timeout is 30.

watch-folder remove [-i <watch folder ID>] [-n <name>]
[--all]
[--host <hostname/IP>]
[--cluster <name>]
[--timeout <seconds>]

Remove one or more existing watch folder(s). The watch folder(s) will be stopped (if running) and permanently removed. The template workflow that this watch folder is attached to will also be removed. One of the options -i, -n, or --all must be specified.

This command can also be specified as mr, ie. episodectl.exe mr

-i
--id A watch folder ID to remove.

-n
--name A watch folder name to remove. This may affect multiple watch folders if they are named the same.

-a
--all Remove all watch folders.

--host A host Node to operate on (default is 127.0.0.1). If this Node is a participant of a cluster but it is not the Master Node, we will automatically be redirected to the Master Node. You can configure a default host/IP to submit to with the command episodectl.exe proxy defaults --host <default host>.

-c
--cluster A Cluster name to operate on. This will try to find a Cluster participant via Bonjour. You can configure a default cluster to submit to with the command episodectl.exe proxy defaults -c <default name>.

--timeout A timeout for the request. This command will connect to the Node through the ClientProxy and this timeout will be used in both those requests but it is unlikely that they will "add up". The default timeout is 30.

watch-folder log [-i <watch folder ID>] [-n <name>]
[--host <hostname/IP>]
[--cluster <name>]
[--timeout <seconds>]

Recieve log messages from one or more existing watch folder(s). This command will not exit, thus having to be manually stopped/breaked. If no option is given, all running watch folders' log messages will be printed. The configuration variable controlling the level/verbosity of the messages is the one in the Node.xml configuration file in the 'watch folders' section inside the 'logging' section. If you want to see debug output from a watch folder with this command, you can use this command to set watch folders to report at debug level: episodectl.exe node log --watch-folders no 7 (the 'no' is no, we don't want to log to file). You will have to either stop/start the watch folder or restart the node after re-configuration. One of the options -i, -n, or --all must be specified.

This command can also be specified as mg, ie. episodectl.exe mg

-i
--id A watch folder ID to recieve log messages from.

-n
--name A watch folder name to recieve log messages from. This may affect multiple watch folders if they are named the same.

--host A host Node to operate on (default is 127.0.0.1). If this Node is a participant of a cluster but it is not the Master Node, we will automatically be redirected to the Master Node. You can configure a default host/IP to submit to with the command episodectl.exe proxy defaults --host <default host>.

-c
--cluster A Cluster name to operate on. This will try to find a Cluster participant via Bonjour. You can configure a default cluster to submit to with the command episodectl.exe proxy defaults -c <default name>.

--timeout A timeout for the request. This command will connect to the Node through the ClientProxy and this timeout will be used in both those requests but it is unlikely that they will "add up". The default timeout is 30.

util

There is currently only one utility command, analyze, with which you can retreive some basic source file information.

util analyze <url or path>
[-o [<key> ...]]
[-s <separator char>]

Analyze

This command can also be specified as ua or analyze, i.e. episodectl.exe ua or episodectl.exe analyze

-o
--out This option takes one or more space separated <key>s, where each <key> is a keyword from the list below. The values will be written in the same order as specified, separated by <separator char>.

duration

Media duration in seconds. Floating point value.

start-tc

Start time code in the format HH:MM:SS:FF. This will be present if time code information was found, this will be an empty value.

video.width

Number of horizontal pixels. Integer value. This will be present if a video track was present in the file. If no video track was found, this will be an empty value.

video.height

Number of vertical pixels. Integer value. This will be present if a video track was present in the file. If no video track was found, this will be an empty value.

video.frame-rate

Number of frames per second. Floating point value. This will be present if a video track was present in the file. If no video track was found, this will be an empty value.

video.bitrate

Bitrate on the video track. Floating point value. This will be present if a video track was present in the file. If no video track was found, this will be an empty value.

video.field-order

Field order for the video track. String value. It can be either unknown, top first, bottom first or progressive.

video.duration

VIDEO track duration. Floating point value. Can be compared to the video duration to find possible audio/video sync issues.

video.display-aspect-ratio-height

Display aspect ratio for height. Integer value. Not set for most containers.

video.display-aspect-ratio-width

Display aspect ratio for width. Integer value. Not set for most containers.

audio.channel-count

Number of audio channels. Integer value. This will be present if a audio track was present in the file. If no audio track was found, this will be an empty value.

audio.sample-rate

Number of audio samples per second. Integer value. This will be present if a audio track was present in the file. If no audio track was found, this will be an empty value.

audio.duration

Audio track duration. Floating point value. Can be compared to the video duration to find possible audio/video sync issues.

audio.bitrate

Bitrate on the audio track. Floating point value. This will be present if a audio track was present in the file. If no audio track was found, this will be an empty value.

-s
--separator Specify a separator character to be used to separate the output. The default is a "vertical bar", or "pipe" character ('|'). There are two special identifiers you can specify - "newline" or "LF" to get a newline separator.

TELESTREAM EPISODE VARIABLES

There are a number of variables available that can be used in file name generation, descriptions, script environment/arguments etc. Each task has a number of in-values, also called variables. Each task-variable has a name in the most simple case but could also be an ID or path that identifies a value inside a container. There are two kinds of containers - lists and "maps"/"dictionaries" (key-value containers). List elements are idenfied by square brackets and a index into the list starting at index 0, for example "args[0]" may refer to the first argument configurable in the Execute task. Keys in a key-value container are referred to by curly brackets, for example "env{MY_VAR}" may refer to a environment variable named MY_VAR, also configurable in the Execute task.

Then there are the values of these variables. The values may be a constant value set by default or by the user, or it could be a variable itself, identified by surrounding dollar signs, for example $source.filename$ which are pre-defined variables (by Episode) that are resolved some time during the execution of the workflow.

To sum it up, tasks have a number of variables that can be set before execution, or read after execution. These are identified by an ID that could just be a name, or may refer to items inside containers. The values of these variables may be a constant value, or a pre-defined "dollar-variable" value that is resolved by Episode.

Let's begin with the list of pre-defined variables (v1, v2 etc. refers to the Notes below):

```
$uuid.uppercase$              - A Universally unique identifier (uppercase).
$uuid.lowercase$              - A Universally unique identifier (lowercase).
$uuid$                    - This is the same as $uuid.uppercase$.

$source.name$                 - Name of the source used (user defined).
$source.type$                 - "file", "monitor", "edl", or "iseq".

$source.url$                  - URL of the source file.
$source.path$                 - Native platform path of the source file. v1.
$source.filename$             - Filename (without extension) of the source file.
$source.extension$            - File extension of the source file.
$source.file$                 - Filename including extension of the source file.
$source.parent-dir-name$      - Name of parent directory of the source file.
$source.parent-dir-path$      - Native platform path of parent directory of the
                          source file.
$source.grandparent-dir-name$ - Name of grandparent directory of the source file.
$source.grandparent-dir-path$ - Native platform path of grandparent directory
                          of the source file.

$workflow.submission-name$    - Name of submission (user defined).
$workflow.submission-time$    - Time of the submission. v3.
$workflow.submission-client$  - ID of the client that did the submission.
$workflow.submission-host$    - Host from which the submission came.

$workflow.template-id$        - The unique template workflow ID. v4.

$workflow.id$                 - The unique workflow ID. v4.
$workflow.name$               - The workflow name.
$workflow.spawn-value$        - The in-value that spawned the workflow. v5
$workflow.spawn-time$         - Time of workflow spawning (instance creation). v3.
$workflow.seq-nr$             - Workflow sequence number, set at spawn time.

$encoder.name$                - User defined Encoder task name.

$encoder.input-duration-s$    - Input duration in seconds (always two decimals).
$encoder.input-duration-hms$  - Input duration with format HH-MM-SS.
$encoder.input-pixel-dimensions$ - Input pixel dimensions with format WxH.
$encoder.input-framerate$     - Input frame rate (number of frames per second).
$encoder.input-samplerate$    - Input audio sample rate in Hertz.

$encoder.output-bitrate-kbps$ - Output bitrate in kbit/s (always two decimals).
$encoder.output-bitrate-mbps$ - Output bitrate in Mbit/s (always two decimals).
$encoder.output-duration-s$   - Output duration in seconds (always two decimals).
$encoder.output-duration-hms$ - Output duration with format HH-MM-SS.
$encoder.output-pixel-dimensions$ - Output pixel dimensions with format WxH.
$encoder.output-framerate$    - Output frame rate (number of frames per second).
$encoder.output-samplerate$   - Output audio sample rate in Hertz.

$mbr.name$                    - User defined MBR task name.

$mbr.package-type$            - "SmoothStreaming", "HTTPStreaming", or
"DASHStreaming"
$mbr.package-name$            - Name prefix of the output files.

$deployment.name$             - User defined Deployment task name.
```

```
$deployment.dest-url$            - URL of the outfile. v2.
$deployment.outfile-path$        - Native platform path of the outfile. v1, v2.
$deployment.outfile-name$        - Filename (without extension) of the outfile. v2.
$deployment.outfile-extension$   - File extension of the outfile. v2
$deployment.outfile-file$        - Filename including extension of the outfile. v2

$destination.url$                - Alias for $deployment.dest-url$.
$destination.path$               - Alias for $deployment.outfile-path$.
$destination.filename$           - Alias for $deployment.outfile-name$.
$destination.extension$          - Alias for $deployment.outfile-extension$.
$destination.file$               - Alias for $deployment.outfile-file$.

$dynamic.time$                   - Execution time of a task. v6 v3.
$dynamic.hr-time$                - Human readable execution time of a task. v6.
$dynamic.year.YY$                - Current year as two digits. v6.
$dynamic.year.YYYY$              - Current year as four digits. v6.
$dynamic.month.name$             - Current full month name. v6, v7.
$dynamic.month.short-name$       - Current abbreviated month name. v6, v7.
$dynamic.month.MM$               - Current month as two digits. v6.
$dynamic.day.name$               - Current full weekday name. v6, v7.
$dynamic.day.short-name$         - Current abbreviated weekday name. v6, v7.
$dynamic.day.DD$                 - Current day as two digits. v6.
$dynamic.hours.HH$               - Current hours as two digits. v6.
$dynamic.minutes.MM$             - Current minutes as two digits. v6.
$dynamic.seconds.SS$             - Current seconds as two digits. v6.
$dynamic.hostname$               - Hostname (of node) where a task is executed. v6.
$dynamic.node-id$                - Node ID (of node)  where a task is executed. v6.
$dynamic.platform$               - Platform where a task is executed. v6.
```

Note v1 - This is probably only useful when working with local files or UNC/Samba on Windows, i.e. not ftp monitors etc.

Note v2 - This is not applicable for YouTube deployment.

Note v3 - This is a "Unix time" or "POSIX time". It is the number of seconds passed since 00:00:00 UTC, January 1, 1970. In the current timezone, the start time was Wed Dec 31 16:00:00 1969.

Note v4 - This may for example be used in calls to episodectl.exe from within scripts.

Note v5 - The spawn value is often the URL of the source file, or in the case of a EDL source, it's the user defined name of the source, i.e. same as $source.name$.

Note v6 - The $dynamic.xxx$ variables will be resolved in run-time, when the actual task (that is configured with one of them) is executed on a certain node. The $dynamic.hostname$ and $dynamic.node-id$ will by default be the same, due to that the Node ID is default set to the hostname of the machine but could be configured to any (cluster-wide unique) ID. The $dynamic.platform$ will have one the the values "Win" or "Mac" (the same values used for the default platform "Tags").

Note v7 - Names according to the current locale.

These pre-defined variables may be used in different places due to where they are resolved and in what order the tasks are run.

The following variables are resolved prior to workflow execution (instantiation) and may be used everywhere in the workflow and read afterwards through the status tasks command:

$source.name$

$source.url$
$source.path$
$source.filename$
$source.extension$
$source.file$
$source.parent-dir-name$
$source.parent-dir-path$
$source.grandparent-dir-name$
$source.grandparent-dir-path$

$workflow.submission-name$
$workflow.submission-time$
$workflow.submission-client$
$workflow.submission-host$

$workflow.template-id$
$workflow.id$
$workflow.name$
$workflow.spawn-value$
$workflow.spawn-time$
$workflow.seq-nr$

The following variables are also resolved prior to workflow execution (instantiation) and may be used everywhere in the workflow but can not be read afterwards, their value is placed directly where they are specified:

$uuid.uppercase$
$uuid.lowercase$
$uuid$

The following variables are resolved when a task that is configured with one of them is actually executed on a node, their value is placed directly where they are specified and therefore not readble after workflow execution, i.e. readable by variable name.

$dynamic.time$
$dynamic.hr-time$
$dynamic.year.YY$
$dynamic.year.YYYY$
$dynamic.month.name$
$dynamic.month.short-name$
$dynamic.month.MM$
$dynamic.day.name$
$dynamic.day.short-name$
$dynamic.day.DD$
$dynamic.hours.HH$
$dynamic.minutes.MM$
$dynamic.seconds.SS$
$dynamic.hostname$
$dynamic.node-id$
$dynamic.platform$

The following variables are generated by the tasks and may be used either after the task has run or read after workflow execution through the status tasks command. That means that, for example, $encoder.output-duration-s$ is perfectly fine to use in a deployment task, since the deployment task is run after the encode task.

```
$encoder.name$
$encoder.input-duration-s$
$encoder.input-duration-hms$
$encoder.input-pixel-dimensions$
$encoder.input-framerate$
$encoder.input-samplerate$

$encoder.output-bitrate-kbps$
$encoder.output-bitrate-mbps$
$encoder.output-duration-s$
$encoder.output-duration-hms$
$encoder.output-pixel-dimensions$
$encoder.output-framerate$
$encoder.output-samplerate$

$mbr.name$
$mbr.package-type$
$mbr.package-name$

$deployment.name$
$deployment.dest-url$
$deployment.outfile-path$
$deployment.outfile-name$
$deployment.outfile-extension$
$deployment.outfile-file$
```

Now let's have a look at the task-variable IDs that you can set these pre-defined variable values on, or a constant value of choice. You have probably seen these "Task type names" mentioned before, but these are the available type names:
Encode, Transfer, YouTube, Execute, Mail.

And here is a list of recommended/documented variable IDs for each:

Encode

```
dest-file-extension      - Outfile extension.
sns                  - Do Split-and-Stitch? "yes"|"no".
sns-min-time             - A integer value (9..3600).
sns-max-splits           - A integer value (2..32).
sns-split-on-gop-size    - A integer value (1..100).
split-tag                - A tag to set on split-encode tasks. e1.
split-inverse-tag        - A inverse-tag to set on split-encode tasks. e1.
stitch-tag               - A tag to set on the stitch task. e1.
stitch-inverse-tag       - A inverse-tag to set on the stitch task. e1.
```

Note e1 - See episodectl.exe tags for a description of tags.

Transfer

```
dest-filename          - A list of values that forms the outfile name. d1
dest-filename[0]       - First part of outfile name.
dest-filename[1]       - Second part of outfile name.
dest-filename[n-1]     - n'th part of outfile name.
dest-sub-dirs          - A list of sub directories. d2, same logic as d1
dest-sub-dirs[0]       - First sub directory.
dest-sub-dirs[1]       - Second sub directory.
dest-sub-dirs[n-1]     - n'th sub directory.
increment-filename     - "yes"|"no". d2
try-link               - Try to hard-link the outfile. d3
try-rename             - Try to rename (move) the outfile. d3
```

Note d1 - This is a list of string values that are concatenated to form the final outfile name (without extension; the extension is taken from the encoded file). To set a list value, you should use the --set-list-name or --set-list-type option to workflow submit where the list items are space separated. Here is an example of how to specify the default naming convention for the outfile:
--set-list-type Transfer dest-filname $source.filname$ - $encoder.name$
Or the alternative:
--set-type Transfer dest-filname[0] $source.filname$ dest-filname[1] - dest-filname[2] $encoder.name$

Note d2 - See episodectl.exe task transfer -h for a description of this.

Note d3 - How to "transfer" the outfile from Episode's "Cache" directory to its final destination? See episodectl.exe task transfer -h for a description of this.

YouTube

```
username               - YouTube account username.
password               - YouTube account password.
meta-data{title}       - Title. y1.
meta-data{titles}      - Title. A list value. y1, y3.
meta-data{titles}[0]   - First piece of title. y1, y3.
meta-data{titles}[1]   - Second piece of title. y1, y3.
meta-data{titles}[n-1] - n'th piece of title. y1, y3.
meta-data{description} - Description. y1.
meta-data{descriptions} - Description. A list value. y1, y3.
meta-data{descriptions}[0] - First piece of description. y1, y3.
meta-data{descriptions}[1] - Second piece of description. y1, y3.
meta-data{descriptions}[n-1] - n'th piece of description. y1, y3.
meta-data{category}    - YouTube category. y2
meta-data{keywords}    - YouTube Keywords. A list value. y3.
meta-data{keywords}[0] - First YouTube Keyword.
meta-data{keywords}[1] - Second YouTube Keyword.
meta-data{keywords}[n-1] - n'th YouTube Keyword.
private                - If the movie is private "yes"|"no".
```

Note y1 - There is no support for inserting pre-defined variable values in the middle of a text. There are therefore 2 versions of both title and description, one version that could only hold a single variable, and 1 version that takes a list of strings. To construct a

title or description made up of both static text and variables, the list versions must be used, for example --set-list-type YouTube meta-data{descriptions} "My description of " "$source.filename$" "is ...".

Note y2 - The category should be one of these values: People, Film, Autos, Music, Animals, Sports, Travel, Games, Comedy, People, News, Entertainment, Education, Howto, Nonprofit, Tech.

Note y3 - To set a list value, you should use the --set-list-name or --set-list-type option to workflow submit where the list items are space separated. Here is an example of how to specify the keywords "world", "record", and "attempt"
--set-list-type YouTube meta-data{keywords} world record attempt
Or the alternative:
--set-type YouTube meta-data{keywords}[0] world meta-data{keywords}[1] record meta-data{keywords}[2] attempt

Execute

```
args              - Arguments to pass to program/script. A list value. x1
args[0]            - First argument.
args[1]            - Second argument.
args[n-1]           - n'th argument.
env               - Environment to set for program/script. A map value. x2
working-dir         - Working directory to set before execution. x3
parse-progress       - Parse/report progress?. "yes"|"no" x4
```

Note x1 - The arguments and/or options to pass to the executed program. To set a list value, you should use the --set-list-name or --set-list-type option to workflow submit where the list items are space separated. These two options will trim away both double and single quotation marks in order to support double escaping of passed options, i.e. if the executed program should recieve option -x, it has to be double escaped on the command line to avoid being treated as a regular option to workflow submit. Here is an example:
--set-list-type Execute args '"-x"' '$source.filename$'.

Note x2 - The environment variables to set for the executed program as a map value. To set map values, you use --set-type or --set-name and refer to the keys in the map with curly brackets {}. Here is an example, I have set the name "First Script" as a user-defined name for my task:
--set-name "First Script" env{UNIQUE_ID} 123 env{DURATION} '$encoder.output-duration-s$'

Note x3 - This should be a native platform/OS path and is therefore highly platform dependent.

Note x4 - See episodectl.exe task execute -h for more information.

Mail

```
username             - SMTP server account username.
password             - SMTP server account password.
server             - SMTP server address.
port             - SMTP server port.
```

```
from-mail          - Sender address.
from-mail          - Receiver address.
cc-mail            - Carbon Copy receiver address.
bcc-mail           - Blind Carbon Copy receiver address.
subject            - Message Subject. m1
body               - Message Body. m1
enable-ssl         - Use TLS/SSL.
```

Note m1 - Both the message subject and message body is specified as a list of strings. This is because any list item may be a dollar variable value. In the command episodectl.exe task mail --subject <subject> --message <message>, both <subject> and <message> are specified as ONE text string, and that string is then tokenized into a list. If you want to get a subject with the text "Episode Error: C:\path\to\MyFile.mov failed to encode!" C:\path\to\MyFile.mov was the variable $source.path$, you have to (in this "dynamic override" case) use: --set-list-type Mail subject 'Episode Error: ' '$source.path$' ' failed to encode!'. Notice the spaces in the surrounding texts.

TELESTREAM EPISODE TAGS

The concept of Tags is used to enable an easy way of controlling execution of tasks in a Cluster, i.e. on which node, or computer, or even group of computers a certain task should run on.

The need for this is primarily for the Execute task (or Script task) which is often dependent on the OS and/or what software and languages are installed on specific machines. But it can be used for other tasks too.

This is how it works. One or more tags can be set on each node in a cluster. One tag is set as default (unless explicitly removed) on each node and that is one of "Mac" or "Win" that could be used to control on which platform a certain task should run. Run episodectl.exe node tag to see tags currently set on the local node. You can also run episodectl.exe node info and look for "Tags:". If you already have a cluster set up, you can run episodectl.exe status nodes which will also show "Tags:" on each node in the cluster.

To add your own tags to the node (you can not configure a node remotely), use episodectl.exe node tag --add MyTag. Tags are case sensitive so be sure that you pay attention to that when specifying your tags.

When you have set up your tags on the nodes in your cluster, there are different tag options to episodectl.exe workflow submit which allows you to set tags on the tasks you're submitting. There are two kinds of tags you can set on tasks. A regular tag that means that the task will require that the tag is set on the machine in order to run there. The other kind of tag is a "inverse-tag". If this is set on a task it means that the task will require that the tag is NOT set on the machine in order to run there.

Since the platform tags "Mac" or "Win" are set by default, this is an example of the most basic usage:

For example, if you have a cluster with one Apple OS X machine and one Microsoft Windows machine. You have made a script file ("MyScript") with AppleScript code in it, this is how you should create a one shot workflow:

episodectl.exe ws -f ... -e ... -d ... -x MyScript --tag-type Execute Mac

Here is a more advanced example. If the Encoder task is configured with Split-and-Stitch, it will create a number of split tasks and a stitch task. It has therefore two "in-values" or variables named "split-tag" and "stitch-tag" that can be set to create these tasks with tags on them. It also has the two inverse-tag versions, "split-inverse-tag" and "stitch-inverse-tag".
For example, if you have 2 machines in your cluster that has fiber connections to your SAN and a bunch of other machines that don't. You want the 2 fiber connected machines to run all stitch tasks, but you do not want them to do any of the split encoding tasks. Begin with choosing and adding a tag for the stitch machines, let's say "Stitch". Now you have two options. Option 1 is to tag all remaining nodes with a "split tag" and set that as value on the Encoder's "split-tag" variable. Option 2 is to "inverse-tag" the split encoding tasks by setting the variable "split-inverse-tag".

Option 1 illustrated:

Tags on node 1: Mac Stitch
Tags on node 2: Mac Stitch
Tags on node 3: Mac Split
Tags on node 4: Mac Split
Tags on node 5: Mac Split

episodectl.exe ws -f ... -e ... -d ... --set-type Encoder stitch-tag Stitch split-tag Split

Option 2 illustrated:

Tags on node 1: Mac Stitch
Tags on node 2: Mac Stitch
Tags on node 3: Mac
Tags on node 4: Mac
Tags on node 5: Mac

episodectl.exe ws -f ... -e ... -d ... --set-type Encoder stitch-tag Stitch split-inverse-tag Stitch

If you want total control of the job distribution you may of course tag each computer with tags like Node1, Node2, Node3 etc and you may use the --tag-workflow or --inverse-tag-workflow options to workflow submit.

EXAMPLES FOR TELESTREAM EPISODE COMMAND LINE INTERFACE

General usage and command synopsis interpretation
==========

The Episode CLI has a set of commands, sub commands, and options. All commands consists of both a command and a sub command and there is often a short version of the command, typically consisting of two letters. For example, to start/launch the Episode Back End, the command episodectl.exe launch start is used, which also can be specified with the short version episodectl.exe ls.

Most commands have options. episodectl.exe understands "long options" and "short options". A long option is always specified with double dashes (--) and the option name immediately following the double dashes, for example a option named "priority" is specified --priority.

A short option is always a single letter and is always specified with a preceding single dash (-), for example the priority option may have a short version "p" which is then specified as -p. Since short options always are a single letter, multiple short options can be specified in "a row" or "a string" of letters, preceded by a single dash, for example the command status tasks has a option --wait with short version -w and a option --visual with short version -v. These two options are often used together to get a human readble, visual, and continuous status feedback in the terminal, and can be specified as -wv.

Most options also have "arguments" (parameters) which is indicated in a number of ways in the command synopsis documentation. Let's look at an example.

```
1    example command <required command argument>
2            --required-option <required option argument>
3            --required-option [<optional argument>]
4            [--optional-option <required option argument>]
5            [--optional-option [<optional argument>]]
6            [--optional-option [OptionalConstant]]
7            [--optional-option RequiredConstant|AnotherRequiredConstant]
8            [--optional-option RequiredConstant|<argument>]
9            [--option <keyword> ...]
10           [--option <key> <value> ... ...]
11           [--option <one time argument> <key> <value> ... ...]
```

In this example command, there are many options that are named the same which is unusual but actually do occur in Episode but not with different arguments like in this example. This is written to better illustrate the syntactic meaning of the options and the arguments.

In the first line, the <required command argument> is indicated by not having surrounding square brackets ([]) which indicates that something is optional, a option or argument. A argument that looks like this <argument> is printed differently on OS X and on Windows. On OS X, the argument is underlined, and on Windows it is surrounded by greater-than and less-than characters (<>). When a argument is indicated like this, it means that the argument should be defined by the user, it is often a path or URL or a keyword defined in Episode. In the help text for the command or option, that argument is often referenced in the same way as defined in the synopsis. In this example, the command text could refer to the required command argument like this:

"This command takes a required argument <required command argument>, where <required command argument> should be a path or URL to a existing file."

Line 2, 3, 4, and 5 should be self-explanatory.

In line 6, the "OptionalConstant" is a CLI defined word that should be entered exactly as the synopsis/documentation states, for example, if the documentation states [--all [request]], the valid option usages are --all or --all request.

In line 7, the vertical bar (|) placed between the two required constants indicates "or",

meaning that either "RequiredConstant" or "AnotherRequiredConstant" must be specified for the option. Notice that options also can be "or'd", for example [--set-default|--reset]. In that case, the options means the same thing. Options can also be "or'd" that do not mean the same thing, for example [-i <id>|--all]. In that case, either the option taking a <id> should be used, or the option --all, but not both, and in which case the documentation should clearly state how to use it!

In line 8, either the "RequiredConstant" or a user defined argument should be specified.

In line 9, the three dots indicates that more than one argument could be specified. It always refers to the previous argument so in this case it means that one or a list of <keyword >s can be specified.

In line 10, there are two sets of the three dots. This indicates that there is a pair of preceding arguments that can be continued. In this case, the pair <key> <value> can continue to be specified as, for example key1 value key2 value key3 value

In the last line, line 11, the dots refer to the <key> <value> pair, just like in line 10, and the <one time argument> should be specified once and only once as the first argument to the option.

About the examples
==========

To illustrate a command line prompt, the "$" character is used followed by a space. The actual command is also highlighted. This is a prompt and a command:

```
$ command
```

To illustrate the user's home directory, the "~" is prepended, like this "~$".

My username is "tomasa" which I will use when showing output from commands and scripts, for example

Output:

```
$ Path: /Users/tomasa/directory/file
```

To symbolize the path to the Current Working Directory, "[CWD]" is used.

Examples that involves script code are written in the Ruby programming language for its simple syntax and easy accessibility on all platforms, please see http://www.ruby-lang.org/en/ for more information about Ruby.

All scripts are available here: /Applications/Episode.app/Contents/MacOS/engine/API/ and can be executed like this (try it!)

```
$ ruby /Applications/Episode.app/Contents/MacOS/engine/API/scriptExample.rb
```

Notice: For simplicity, all commands and paths used in the examples are Unix/OS X-style. When the difference for Windows is big, a separate Windows section is present. The Windows PowerShell is a good shell to use on Windows because it has the most commonly used Unix commands as aliases for its regular commands, see http://technet.microsoft.com/en-us/library/bb978526.aspx and http://en.wikipedia.org/wiki/Windows_PowerShell.

Making the Episode CLI accessible from anywhere
==========

A nice thing to begin with is to make episodectl.exe accessible from any directory, without having to specify the complete path to it.
This can be accomplished in many many ways but let's only go through a couple. What is needed is to somehow make the location of episodectl.exe known to the command-line shell (the command-line interface provided by the Operating system).

OS X

The default shell on OS X is "Bash". Bash, as other Unix-shells, uses the built-in environment variable PATH to search for executable files, also called commands. Here is a quote from the description of PATH in the Bash Reference Manual - http://www.gnu.org/software/bash/manual/bashref.html#Bourne-Shell-Variables

> "A colon-separated list of directories in which the shell looks for commands."

There are a number of pre-defined paths already present in PATH which can be seen by executing the command (the dollar character is used to read the value of a variable)

```
$ echo $PATH
```

Example output:

```
$ /usr/bin:/bin:/usr/sbin:/sbin:/usr/local/bin
```

The quickest way of accomplishing what we want is to make a symbolic link to episodectl in one of the already defined directories in $PATH. Although I would recommend to continue to the next solution, here is a example of how to create the symbolic link

```
$ sudo ln -s /Applications/Episode.app/Contents/MacOS/engine/bin/episodectl /usr/bin/
```

Since the episodectl.exe is located in the directory /Applications/Episode.app/Contents/MacOS/engine/bin/ we can add this directory to the PATH variable so that Bash will look in this directory when trying to find the command "episodectl", but how do we do that?
Here is another quote from Bash Reference Manual - http://www.gnu.org/software/bash/manual/bashref.html#Bash-Startup-Files

> "When Bash is invoked [...], it first reads and executes commands from the file /etc/profile, if that file exists. After reading that file, it looks for ~/.bash_profile, ~/.bash_login, and ~/.profile, in that order, and reads and executes commands from the first one that exists and is readable."

First, check if one of those files is already present in the user's home directory. (The home directory is indicated by the "~" character, this is the directory the user is placed in when opening a Terminal window). These files begin with a dot (as in ".bash_profile") which means that they are "hidden" and not visible when doing a regular directory listing with ls. To also list files beginning with a dot, use the ls option -a.

```
~$ ls -a
```

There shouldn't be any of these files present unless some software has been installed that has automatically created one, such as for example MacPorts, X11, etc.

If there is one of these files present, edit that one, if there isn't we will create ".bash_profile ". You can use any text editor of choice as long as the output file is saved as pure text. Instead of using a text editor, we can simply execute the following command

```
~$ echo 'export PATH=$PATH:/Applications/Episode.app/Contents/MacOS/engine/bin/' >>
```

.bash_profile

If the file existed, the line will be appended to the file. If it didn't exist, it will be created.

The PATH=$PATH means that the old value of PATH is read into the new value of PATH before "our " path is appended to the list.

Now, when opening a new Terminal window, just typing ...
~$ epi[Hit TAB button]
... should result in
~$ episodectl

Congatulations, now we are ready to get to work!

Windows

We want to go to the Advanced tab in System Properties. Click on the button "Environment Variables...", locate the PATH variable in the list and click "Edit..." button. Add a semi-colon (;) at the end and type in (or copy/paste) the path where Episode is installed and add "\bin" at the end of that path.
Now, open a new Windows PowerShell window and type
~$ epi[Hit TAB button]

The shell on Windows is not case sensitive when performing "TAB completion", so it can be a bit hard to get to episodectl.exe, continue to hit the TAB button until you see
~$ episodectl.exe

Congatulations, now we are ready to get to work!

Controlling the back end processes

To start, restart, list, or stop the Episode back end processes, the launch ... commands are used. If a process' configuration file is changed, for example the Node.xml file, that process must be restarted in order to use the new configuration.

List processes
```
$ episodectl.exe launch list
$ episodectl.exe ll
```

Example output:
```
EpisodeXMLRPCServer  is running with PID 37518
EpisodeJSONRPCServer  is running with PID 37511
EpisodeClientProxy   is running with PID 37515
EpisodeAssistant     is running with PID 37512
EpisodeIOServer      is running with PID 37509
EpisodeNode          is running with PID 37506
$
```

Restart (or start) processes
```
$ episodectl.exe launch restart
$ episodectl.exe lr
```

Example ouput:
EpisodeNode restarted
EpisodeIOServer restarted
EpisodeAssistant restarted
EpisodeClientProxy restarted
EpisodeXMLRPCServer restarted
EpisodeJSONRPCServer restarted
$

Get Node information
==========

A very useful tool to verify the configuration, state, licenses, etc. in a node, local or remote, is to use the node info command.

Get information about the local Node
$ episodectl.exe node info
$ episodectl.exe ni

Get information about a remote Node. Notice that the remote node must be in "public" mode, i.e. a cluster master or a cluster participant
$ episodectl.exe ni 10.5.5.123
$ episodectl.exe ni server.domain.com

Setting up a environment for the examples
==========

Before we start going through the examples, it's very good to set up a directory structure that is easy to work with and that can be referenced to in the rest of the examples. Let's begin with creating a root directory that everything we create later is going to be placed in and referenced from. A good start is a directory called "EpisodeAPI" in the home directory.

Note: We will only operate in the user's home directory in these examples to avoid "permission denied" errors and it is not within these examples to describe how to get around such errors.

Create the root directory for examples, make sure you are in the home directory
~$ mkdir EpisodeAPI

Then go into that directory with
~$ cd EpisodeAPI

Tip! Try and get used to always using the TAB keyboard button when entering paths in the Terminal, because

* It speeds up navigation and finding target files
* It greatly reduces the risk of typing errors
* It automatically corrects paths with characters in it that needs to be preceded by a shell escape character, for example a path with a space in it, where the space character needs to be preceded by a backslash. In Windows PowerShell, quotation marks are automatically inserted around a path when

using the TAB button.

Try again and do this, back out of the directory again using ".." which means "parent directory"
$ cd ..
~$ cd Epi[Hit TAB button]

The shell should automatically (unless there is a directory in ~/ that begins with "Epi " already) fill in
~$ cd EpisodeAPI/

Make sure you are currently inside the ~/EpisodeAPI/ directory because we will create a few sub directories there now.

Create a directory for input (sources)
$ mkdir input

Create a directory for tasks
$ mkdir tasks

Create a directory for workflows
$ mkdir workflows

Create a directory for output
$ mkdir output

To be able to submit our first workflow, we need a Encode task. When testing things, it's good (in my opinion) to use a fast encoder and a short source file. In the application bundle, there is a API directory where there is a file called encoder.epitask that is going to be used in the examples, let's copy it over here...
$ cp /Applications/Episode.app/Contents/MacOS/engine/API/encoder.epitask .

Notice that the last dot (.) in the command means "current directory", which is used as the target for the copy command.
You are of course free to use whatever Encode task you want, the template Encode tasks that comes with Episode are located here
/Applications/Episode.app/Contents/Resources/templates/tasks/encodings/

Now we only need a source file. The only source file that I can refer to is a file called DefaultSource.mov which is located here
/Applications/Episode.app/Contents/Resources/DefaultSource.mov, it is not a pretty sight but it's not the purpose either. I'm sure you have a bunch of nice clips to use instead, why would you need Episode otherwise, right? Let's copy a source here that we can use later, and at the same time, rename it to source.mov
$ cp /Applications/Episode.app/Contents/Resources/DefaultSource.mov ./source.mov

Now we are ready to move on to the first submission!

Submit the first workflow

Submitting workflows is easy. It usually involves specifying tasks that have been created in advance which is demonstrated in the next section, but to get a feel for where we're going, we will do a simple submission so we can refer back to that in subsequent sections.
We begin with the most simple submission command, make sure you are in ~/EpisodeAPI/ created

in previous section
$ episodectl.exe workflow submit --file source.mov --encoder encoder.epitask --destination output/

For the rest of the examples, I will use the short versions, both for the command and for options that have a short version, like this
$ episodectl.exe ws -f source.mov -e encoder.epitask -d output/

Let's submit it again and watch the status of it. We use option --wait to tell episodectl to not exit and wait until the submitted workflow is done, and option --visual to get a human readable status output which is nice when doing things manually like now. Notice that the short options are used, and that they are specified as described in the first section
$ episodectl.exe ws -f source.mov -e encoder.epitask -d output/ -wv

Just for fun, try it again and type as little as possible, remember to always use the TAB button!
$ epi[Hit TAB] ws -f s[Hit TAB] -e e[Hit TAB] -d o[Hit TAB] -wv

As you may have noticed, we always have used relative paths in these examples. This is to easily being able to demonstrate usage when typing/submitting things manually. For real integrations with scripts/programs around the CLI, absolute paths are probably always used and the long options are preferable for readability and clarity.

Creating tasks

To be able to submit workflows to Episode, task configuration files must be used. The most important task is the Encode task which must be created through the Episode GUI application. All the other task types can be created through the CLI, either explicitly through the task ... commands or implicitly (automatically) when using the workflow submit command. For example, the only required configuration for a Transfer task is the URL to a directory to transfer a output file to, so a Transfer task may be implicitly created in the submit command by just specifying a URL or path like this --destination /path/to/destination/ instead of specifying a explicitly created task like this --destination /path/to/MyDeployment.epitask.

All tasks have a few common configurations ...

name
: All tasks have a name. The name can be used for referencing a task when submitting it, for example override a certain configuration on it. The name can also be used by other tasks in the workflow, for example in naming conventions, and it can be used in decisions in Execute tasks or in a integration through status output etc, etc.

priority
: Workflow-internal prioritization, see episodectl.exe priority for more information.

tag
inverse-tag
: Task distribution control, see episodectl.exe tags for more information.

... and a few common options for the output of the created task

format
Indicate what format to write the created task in (XML, binary, etc.). If dynamic task creation is performed by a program/script, it could be beneficial to write configuration files as binary to speed up parsing/reading and to reduce data size.

out
Specify in which directory the created configuration file should be written.

print-plain-path
Meant to be used when a program/script should read back the path of the created file.

task transfer

Create a Transfer task, no options
$ episodectl.exe task transfer ~/EpisodeAPI/output/
$ episodectl.exe tt ~/EpisodeAPI/output/

Output:
Task configuration written to: /Users/tomasa/EpisodeAPI/output.epitask
$

Create a transfer task with name "deployment", read back path into script. Notice This is Ruby script code, don't try to execute it con the command line!
f = IO.popen("episodectl task transfer ~/EpisodeAPI/output/ --name deployment -o ~/EpisodeAPI/ --format bin --print-plain-path")
output_path = f.gets
puts "'#{output_path}'"

Output:
'/Users/tomasa/EpisodeAPI/deployment.epitask'
$

Create a Transfer task with a custom naming convention
$ episodectl.exe tt ~/EpisodeAPI/output/ --dest-filename '$source.filename$-Encoded@$dynamic.hr-time$-Duration-$encoder.output-duration-hms$-Bitrate-$encoder.output-bitrate-mbps$Mbps'

Create a Transfer task for upload to a FTP server
$ episodectl.exe tt ftp://user:password@server.domain.com/directory/ --name MyFTP

Output:
Task configuration written to: /Users/tomasa/EpisodeAPI/MyFTP.epitask
$

Important When using deployments to external servers, always consider the potential failure due to server reliability, possible maintanance downtimes, possible overloading, etc, etc. Always consider using a backup deployment to some local storage or the local disk. For example, create a deployment called SafetyBackup.epitask which points to a local directory, then submit the workflow with the two deployments
$ episodectl.exe ws -f source.mov -e encoder.epitask -d MyFTP.epitask

SafetyBackup.epitask

Advanced Example

The naming convention, as a lot of other things, can be dynamically created when we submit the workflow. When specifying a text string consisting of variables, the string is tokenized so that each variable ends up in its own list item and any surrounding static text also ends up in its own list item in the task configuration. Let's demonstrate that. Let's say that we use the default naming convention, but between the source filename and the Encode task name, we want to put in a custom message. Here is a example of how the default naming convention looks in a XML configration file (the naming convention list items are highlighted)

```
<list name="dest-filename">
   <string>$source.filename$</string>
   <string>-</string>
   <string>$encoder.name$</string>
</list>
```

In the default naming convention, the only static text is the dash (-) between the source filename and the Encode task name. We could just substitute the dash with our custom message but it wouldn't look very nice so we want to have dashes on both sides of the message. Since the naming convention is tokenized (by the CLI) by dollar signs (as in Episode's variables), we need to specify a dummy variable to get a separate list item to substitute, for example

```
$ episodectl.exe tt ~/EpisodeAPI/output/ --dest-filename
'$source.filename$-$custom-message$-$encoder.name$' --name custom-name-deployment -o
~/EpisodeAPI/tasks/
```

Output:

```
Task configuration written to:
/Users/tomasa/EpisodeAPI/tasks/custom-name-deployment.epitask
$
```

Now, if we look in the created file, the naming convention should look like this (The key parts for substitution are highlighted)

```
<list name="dest-filename">
   <string>$source.filename$</string>
   <string>-</string>
   <string>custom-message</string>
   <string>-</string>
   <string>$encoder.name$</string>
</list>
```

Notice that the dollar signs have been removed. That's because it wasn't a recognized Episode variable.

Referencing a list item begins at index 0, so our custom message is now at index 2. We also need to know the name of the configuration we're affecting, in this case "dest-filename". There are four different options for setting values/variables in the submission command workflow submit. These are --set-name, --set-type, --set-list-name , and --set-list-type.
Let's test all of them. We begin with --set-name. For this option, we need to know the name of our task. When we created the task, we specified the name "custom-name-deployment" so here's how to submit the workflow with a custom message

inserted
$ episodectl.exe ws -f source.mov -e encoder.epitask -d tasks/custom-name-deployment.epitask --set-name custom-name-deployment dest-filename[2] "My Message"

If we have multiple Transfer tasks, for example, one that uploads to a FTP server, and the same configured naming convention, we can use the --set-type option to set the message on all Transfer tasks like this
$ episodectl.exe ws -f source.mov -e encoder.epitask -d tasks/custom-name-deployment.epitask --set-type Transfer dest-filename[2] "My Message"

The two list options are helper options for setting a complete list of values, separated by space characters. These two examples replaces the whole naming convention we configured earlier, i.e. these examples makes the previous configuration superflous
$ episodectl.exe ws -f source.mov -e encoder.epitask -d tasks/custom-name-deployment.epitask --set-list-name custom-name-deployment dest-filename '$source.filename$' - "My Message" - '$encoder.name$'

$ episodectl.exe ws -f source.mov -e encoder.epitask -d tasks/custom-name-deployment.epitask --set-list-type Transfer dest-filename '$source.filename$' - "My Message" - '$encoder.name$'

Here is a script with a more fancy message (requires previous steps) that you can run like this
$ ruby /Applications/Episode.app/Contents/MacOS/engine/API/CLI/TaskTransferSetNamingConv.rb

Output:
Outfile path: /Users/tomasa/EpisodeAPI/output/source-Created by tomasa-encoder.mov
$

task youtube

A YouTube task is probably pretty useless without the use of at least one variable. Let's create a YouTube task with the source filename as YouTube title and a description also containing the source filename. Let's call it "youtube" and place it in the tasks/ sub directory for easy reference in next example
$ episodectl ty -u MyName -p MyPass -t '$source.filename$' -d 'Here goes the description of $source.filename$' -c Howto -k Episode Example --name youtube -o tasks/

Output:
Task configuration written to: /Users/tomasa/EpisodeAPI/tasks/youtube.epitask
$

Advanced Example

To make the YouTube deployment a bit more useful, we can substitue the "meta-data" (title, description, category, keywords) when we submit a workflow. The YouTube task's configuration for "title" and "description" are list values called "titles" and "description s". The legacy single-string versions are still available, but the list versions have precedence and should be used even when using a single string of static text or a single variable. The list values makes it possible to have variables in the middle of static text which wouldn't be supported otherwise. When using options --title and --description, the CLI will split up dollar-variables and static text to form a list and configure the input configurations meta-data{titles} and meta-data{descriptions}. That means that the task created in the previous example will have 2 list entries configured for the description. To make the base task (the one that is used in submissions but with dynamically overrided values) more flexible, let's create a new one that has dummy text in a single list entry so we know that we always override the list values completely, even when using a single static text or a single variable

```
$ episodectl ty -u MyName -p MyPass -t placeholder -d placeholder -c Howto -k
placeholder --name youtube -o tasks/
```

Output:

```
Task configuration written to: /Users/tomasa/EpisodeAPI/tasks/youtube.epitask
$
```

This example demonstrates specifying a list of 3 values for the title, a single static text as description, a new category, and new keywords spcified as single list entries (as opposed to using --set-list-name for keywords).

```
$ episodectl.exe ws -f source.mov -e encoder.epitask -d tasks/youtube.epitask
--set-list-name youtube meta-data{titles} 'World record attempt. '
'$source.filename$' ' encoded with Episode' --set-list-name youtube
meta-data{descriptions} 'I will make a new world record attempt in transcoding'
--set-name youtube meta-data{category} Entertainment meta-data{keywords}[0]
Telestream meta-data{keywords}[1] Episode
```

Important When using deployments to external servers, always consider the potential failure due to server reliability, possible maintanance downtimes, possible overloading, etc, etc. Always consider using a backup deployment to some local storage or the local disk. For example, create a deployment called SafetyBackup.epitask which points to a local directory, then submit the workflow with the two deployments

```
$ episodectl.exe ws -f source.mov -e encoder.epitask -d youtube.epitask
SafetyBackup.epitask
```

Here is a script that demonstrates how you could interactively create the submission command

```
$ ruby
/Applications/Episode.app/Contents/MacOS/engine/API/CLI/TaskYouTubeInteractiveUsage.rb
```

task execute

The Execute task can be used to do pretty much anything, upload a file to some web storage, encode a file with Episode Engine 5.x, create output meta data, create archives, integrate with content management systems, etc, etc. Read the help page

for the Execute task (episodectl.exe tx -h) and make sure you understand the difference between running a program at a specified path, and reading in a script which can be distributed in a cluster.

The first example will show how we can create a simple "move source file" task. We will use the built in program /bin/mv to move/rename the file.

Important Never create a task that executes installed programs as "content", i.e. when using a OS program or other installed binaries, always create the task with option --content no.

Apart from obviously being platform dependent (to OS X), this task will not be able to move any URL, only locally accessible paths, so we will use the variable $source.path$ to send to /bin/mv. We also make sure that it is only executed on a OS X machine by tagging it with the built-in tag "Mac".

```
$ episodectl.exe tx /bin/mv --content no --args '$source.path$' /tmp/ --tag Mac
-o tasks/
```

Output:

```
Task configuration written to: /Users/tomasa/EpisodeAPI/tasks/mv.epitask
$
```

If we submit a workflow with our ~/EpisodeAPI/source.mov source file, it will be moved to /tmp/

```
$ episodectl.exe ws -f source.mov -e encoder.epitask -d output/ -x
tasks/mv.epitask -wv
```

Let's move it back here again...

```
$ mv /tmp/source.mov .
```

When working with scripts, it's always good to know what the data looks like that comes into a script. Try running the command

```
$ ruby
/Applications/Episode.app/Contents/MacOS/engine/API/scriptPrintARGV_ENV.rb Test
arguments
```

Now there should be a file called "argv_env.txt" in the directory ~/EpisodeAPI/. Read contents, for example

```
$ cat argv_env.txt
```

Sample output, a lot of environment variables have been excluded here:

```
Arguments:
'Test'
'arguments'

Environment:
SHELL=/bin/bash
USER=tomasa
HOME=/Users/tomasa
```

Let's create a Execute task with this script and set a few arguments and environment variables in it. To easily find the environment variables we set in the output from the script, I prepend "EPISODE_" to them which is of course not necessary

```
$ episodectl.exe tx
/Applications/Episode.app/Contents/MacOS/engine/API/scriptPrintARGV_ENV.rb --args
```

'$source.path$' '$deployment.outfile-path$' --env EPISODE_WORKFLOW_NAME
'$workflow.name$' EPISODE_SOURCE_DURATION '$encoder.input-duration-s$' -o tasks/

Output:
Task configuration written to:
/Users/tomasa/EpisodeAPI/tasks/scriptPrintARGV_ENV.rb.epitask
$

If we submit a workflow with this task and dynamically set some data, let's say a server hostname and a path
$ episodectl.exe ws -f source.mov -e encoder.epitask -d output/ -x tasks/scriptPrintARGV_ENV.rb.epitask --set-type Execute env{DYNAMIC_SERVER} my.server.com env{DYNAMIC_PATH} /dynamic/directory/ --workflow-name Amazing -wv

Content of ~/EpisodeAPI/argv_env.txt should look something like this:

Arguments
'/Users/tomasa/EpisodeAPI/source.mov'
'/Users/tomasa/EpisodeAPI/output/source-encoder (5).mov'

Environment:
SHELL=/bin/bash
HOME=/Users/tomasa
USER=tomasa
DYNAMIC_PATH=/dynamic/directory/
DYNAMIC_SERVER=my.server.com
EPISODE_SOURCE_DURATION=3.00
EPISODE_WORKFLOW_NAME=Amazing

A very important thing to notice here is that this task is only possible to run as a success task. That is because we have two input values that requires output from previous successful tasks, one from the the Encode task ($encoder.input-duration-s$) and one from the Transfer task ($deployment.outfile-path$). If we specify this Execute task to run on 'done' or 'failure' status, it will not run because it needs to be provided with the these values which it will not be if any of the previous tasks fail.

If status 'done' is chosen for a script, the status of the workflow's previous tasks may be checked inside the script with a call to episodectl.exe status tasks. The script /Applications/Episode.app/Contents/MacOS/engine/API/scriptStatistics.rb demonstrates that.

This script uses the pre-defined environment variable EPISODECTL to get the absolute path to the Episode CLI. It also uses the variable EPISODE_WORKFLOW_ID which should be configured with the Episode variable $workflow.id$ to be able to query the status for the Encode task. Let's create a Execute task
$ episodectl.exe tx /Applications/Episode.app/Contents/MacOS/engine/API/scriptStatistics.rb --env EPISODE_WORKFLOW_ID '$workflow.id$' -o tasks/

Output:
Task configuration written to:
/Users/tomasa/EpisodeAPI/tasks/scriptStatistics.rb.epitask
$

Let's try to submit a workflow with this task, use the encoder task as a source file, just to get an error.

```
$ episodectl.exe ws -e encoder.epitask -f source.mov encoder.epitask -d output/ -x tasks/scriptStatistics.rb.epitask done -wv
```

A file called "statistics.txt" should now be present in ~/EpisodeAPI/, let's check it's contents

```
$ cat statistics.txt
```

Output:

```
encoder.epitask failed to encode: initialization failed (no importer found)
source.mov encoded with 'encoder' in 7 seconds
$
```

If we want to do a script that does something that could take a long time, for example upload a file to some web storage or a FTP server, we might need to report progress from the script. Reporting progress is both visually nice and prevents the task from timing out due to lack of reporting. The default timeout is 10 minutes (which can be configured in the (Master-)Node's configuration file).

Let's begin with testing the progress reporting. There is a dummy progress reporting script located here /Applications/Episode.app/Contents/MacOS/engine/API/scriptDummyProgress.rb, test executing it

```
$ ruby /Applications/Episode.app/Contents/MacOS/engine/API/scriptDummyProgress.rb
```

Output (1 second delay for each line):

```
progress[0]
progress[10]
progress[20]
progress[30]
progress[40]
progress[50]
progress[60]
progress[70]
progress[80]
progress[90]
$
```

In order for Episode's Execute task to parse the scripts output, we must tell it to do so

```
$ episodectl.exe tx /Applications/Episode.app/Contents/MacOS/engine/API/scriptDummyProgress.rb --parse-progress yes -o tasks/
```

Submit a workflow, just to see that it works...

```
$ episodectl.exe ws -f source.mov -e encoder.epitask -d output/ -x tasks/scriptDummyProgress.rb.epitask -wv
```

There is a example script for uploading to a FTP server that demonstrates progress too. If you don't have a FTP server set up or one that you can use, you can always just check out the code. Let's copy the script over here to reduce commands a bit...

```
$ cp /Applications/Episode.app/Contents/MacOS/engine/API/scriptFTPUpload.rb .
```

This script uses the environment variables EPISODE_OUTPUT_PATH, FTP_SERVER,

FTP_USER, and FTP_PASS. The FTP_ variables have default values to 127.0.0.1 and anonymous user. If you have a FTP server on localhost with anonymous user access/upload rights, you can test the script with this command

```
$ env EPISODE_OUTPUT_PATH=~/EpisodeAPI/source.mov ruby scriptFTPUpload.rb
```

Otherwise you can of course set all variables, for exmaple

```
$ env EPISODE_OUTPUT_PATH=~/EpisodeAPI/source.mov FTP_SERVER=my.server.com
FTP_USER=MyUser FTP_PASS=MyPass ruby scriptFTPUpload.rb
```

As you can see, the script outputs a floting point progress between 0 and 1, e.g. progress[0.21157411376775]. This means that we must configure the Episode Execute task to parse that as a "fraction" as the progress type is called. Create the task

```
$ episodectl.exe tx scriptFTPUpload.rb --parse-progress yes --progress-type
fraction --env EPISODE_OUTPUT_PATH '$deployment.outfile-path$' FTP_SERVER
my.server.com FTP_USER MyUser FTP_PASS MyPass -o tasks/ --name FTPdeployment
```

Output:

```
Task configuration written to:
/Users/tomasa/EpisodeAPI/tasks/FTPdeployment.epitask
$
```

A common situation is the requrement for some output meta data. Here is a simple example of how to create a XML file with some meta data from a Execute task. Copy the example script over here...

```
$ cp
/Applications/Episode.app/Contents/MacOS/engine/API/scriptMetaDataGeneration.rb .
```

This script uses the environment variables EPISODE_OUTPUT_PATH, EPISODE_DURATION, EPISODE_SOURCE_FILENAME, and MY_DESCRIPTION. The EPISODE_ variables should be configured with Episode variable values, and MY_DESCRIPTION has a default value but should be specified at submission-time. The script is testable with the following command

```
$ env EPISODE_OUTPUT_PATH=~/EpisodeAPI/source.mov EPISODE_DURATION=01-23-45
EPISODE_SOURCE_FILENAME=source ruby scriptMetaDataGeneration.rb
```

This should have produced the file ~/EpisodeAPI/source.mov.xml, look at content

```
$ cat source.mov.xml
```

Output:

```
<?xml version="1.0" encoding="UTF-8"?>
<SomeMetaDataXML>
  <File>
    <Name>source</Name>
    <Description>No description supplied</Description>
    <Duration>01:23:45</Duration>
  </File>
</SomeMetaDataXML>
$
```

Now it's time to create the Execute task for the meta data generation

```
$ episodectl.exe tx scriptMetaDataGeneration.rb --env EPISODE_OUTPUT_PATH
'$deployment.outfile-path$' EPISODE_DURATION '$encoder.output-duration-hms$'
EPISODE_SOURCE_FILENAME '$source.filename$' --name MetaDataGenerator -o tasks/
```

Output:

```
    Task configuration written to:
/Users/tomasa/EpisodeAPI/tasks/MetaDataGenerator.epitask
    $

Now I think it's time to clear all the output files... it's a real mess in there at
the moment...
    $ rm output/*

And then submit a test workflow. Insert a custom description
    $ episodectl.exe ws -f source.mov -e encoder.epitask -d output/ -x
tasks/MetaDataGenerator.epitask --set-type Execute env{MY_DESCRIPTION} 'Nothing more
than a simple test' -wv

This should have produced the file ~/EpisodeAPI/output/source-encoder.mov.xml
    $ cat output/source-encoder.mov.xml

Output:
    <?xml version="1.0" encoding="UTF-8"?>
    <SomeMetaDataXML>
      <File>
        <Name>source</Name>
        <Description>Nothing more than a simple test</Description>
        <Duration>00:00:03</Duration>
      </File>
    </SomeMetaDataXML>
    $
```

task mail

```
The Mail task is probaly mostly useful for sending a notification if a encoding
should fail, let's create a Mail task with a nice error message. The default port
587 is used (please see http://en.wikipedia.org/wiki/Simple_Mail_Transfer_Protocol
about port 587). The automatically created name is also used, as well as using SSL.
The assumption in this example's message is that the Encode task failed (and not the
deployment).
    $ episodectl.exe tmail -u username -p password -s smtp.server.com -f
EpisodeEngine@mycompany.com -t episode.admin@mycompany.com -j 'Error: $source.file$
failed to encode' -m 'Please run this command to see why "episodectl status tasks
$workflow.id$ --filter-name $encoder.name$ -o task.message"' -o tasks/

Output:
    Task configuration written to:
/Users/tomasa/EpisodeAPI/tasks/episode.admin@mycompany.com.epitask
    $
```

task mbr

```
MBR is short for Multi Bit Rate and which currently supports Apple's HTTP Live
Streaming, DASH Streaming, and Microsoft's Smooth Streaming. A MBR workflow consists
```

of at least one MBR task and one Encode task (and a deployment...). To utilize the streaming technoligies better, at least 3 different Encode tasks with different bit rates should be used. Encode tasks can only be created in the Episode GUI application and should be configured to output the container format TIFO (Telestream Intermediate Format) which is the format that the MBR task can take as input.

Apple's HTTP Live Streaming and DASH Streaming can be produced on both Mac OS X and Microsoft Windows platforms, while Microsoft's Smooth Streaming can only be produced on the Windows platform. "Produced" in this context means that the MBR task configured to output Smooth Streaming must run on a Windows machine. The individual Encode tasks can of course run on either platform as usual.

We begin with creating three MBR tasks, one for each streaming technology. The default fragment duration for each is used. As package name, the source filename is specified

```
$ episodectl.exe tmbr HTTPStreaming --name HLS --package-name
'$source.filename$' -o tasks/
```

Output:

```
Task configuration written to: /Users/tomasa/EpisodeAPI/tasks/HLS.epitask
$

$ episodectl.exe tmbr SmoothStreaming --name MSS --package-name
'$source.filename$' -o tasks/
```

Output:

```
Task configuration written to: /Users/tomasa/EpisodeAPI/tasks/MSS.epitask
$

$ episodectl.exe tmbr DASHStreaming --name DASH --package-name
'$source.filename$' -o tasks/
```

Output:

```
Task configuration written to: /Users/tomasa/EpisodeAPI/tasks/DASH.epitask
$
```

Now we need some Encode tasks configured to output TIFO files as input for the MBR tasks. There are 3 Encode tasks configured for this in /Applications/Episode.app/Contents/MacOS/engine/API/ named "H264 1Mbit TIFO.epitask", "H264 768kbit TIFO.epitask", and "H264 512kbit TIFO.epitask". Let's copy them over here...

```
$ cp /Applications/Episode.app/Contents/MacOS/engine/API/H264* tasks/
```

When submitting a MBR workflow with multiple MBR tasks, the CLI will match the paths of the Encode tasks and build the workflow so that the same Encodes are only done one time and not one time for each streaming format. Let's try that, but before we submit the workflow, let's try something different to be able to view the result better, we set the number of jobs the node will execute down to 0 (zero)

```
$ episodectl.exe node jobs --set 0
```

Now we'll go ahead and submit the workflow. Notice that the --mbr option is used three times! We insert a custom naming convention here too

```
$ episodectl.exe ws -f source.mov --mbr tasks/HLS.epitask tasks/H264\ 1Mbit\
TIFO.epitask tasks/H264\ 768kbit\ TIFO.epitask tasks/H264\ 512kbit\ TIFO.epitask
--mbr tasks/MSS.epitask tasks/H264\ 1Mbit\ TIFO.epitask tasks/H264\ 768kbit\
TIFO.epitask tasks/H264\ 512kbit\ TIFO.epitask --mbr tasks/DASH.epitask tasks/H264\
1Mbit\ TIFO.epitask tasks/H264\ 768kbit\ TIFO.epitask tasks/H264\ 512kbit\
TIFO.epitask -d output/ --naming '$source.filename$-$mbr.name$' -wv
```

Output should look something like this (irrelevant columns are not present here...):

```
Filename          Task Name          Task Type
_____________________________________________________
    Running (0 of 0)

_____________________________________________________
    Queued (7 of 7)

source.mov        H264 1Mbit TIFO     Encode
source.mov        H264 512kbit TIFO   Encode
source.mov        H264 768kbit TIFO   Encode
source.mov        HLS                 MBR
source.mov        MSS                 MBR
source.mov        DASH                MBR
source.mov        output              Transfer
source.mov        output              Transfer
_____________________________________________________
    History (0 of 0)
```

Here we can see that there are only three Encode tasks, which will set their output files as input to both MBR tasks. You can of course put different encodes into each MBR format in the way you like. Let's get the work started again

```
$ episodectl.exe node jobs --set-recommended
```

Notice OS X: If you're on OS X, the Smooth Streaming task will display this message "Waiting for node with Windows operating system to become available". To get rid of the workflow, open up a new Terminal window or tab and simply stop/cancel all workflows. These commands will do that

```
$ episodectl.exe workflow stop --all
$ episodectl.exe wp --all
$ episodectl.exe job cancel --all
$ episodectl.exe jcan --all
```

Best practice

Specifying the rather long command line for submitting the MBR workflows can be avoided by dividing the commands into several steps. Since it's likely that the same workflow will be used over and over again with different source files, we can optimize things a bit. We will simply start with creating the workflow and save it to disk with a "dummy source" that is going to be replaced with a real source for all subsequent submissions we do. We will do the same submission command as earlier, but we will tell Episode to write it to disk (with a dummy source) instead of actually submitting it. To reduce data size and also speed up Episode's reading/parsing of the workflow each time we later submit it, we save it in binary format ("bin")

```
$ episodectl.exe ws -f DummySource --mbr tasks/HLS.epitask tasks/H264\ 1Mbit\
TIFO.epitask tasks/H264\ 768kbit\ TIFO.epitask tasks/H264\ 512kbit\ TIFO.epitask
--mbr tasks/MSS.epitask tasks/H264\ 1Mbit\ TIFO.epitask tasks/H264\ 768kbit\
TIFO.epitask tasks/H264\ 512kbit\ TIFO.epitask -d output/ --naming
'$source.filename$-$mbr.name$' --name MBRWorkflow -o workflows/ --format bin
```

Output:

```
Submission configuration written to:
```

/Users/tomasa/EpisodeAPI/workflows/MBRWorkflow.episubmission
$

The workflow is now ready to be used thousands of times with different source files, or even a different source type, a Monitor, a EDL, or a Image Sequence. Let's try a source file, notice that option --submission, short -s is used in the submission command to specify that a workflow (or submission file) is used as input
$ episodectl.exe ws -s workflows/MBRWorkflow.episubmission -f source.mov -wv

Very much easier, wasn't it?

Notice: When saving workflows for later submissions, some things are not saved. These include all variable override options --set-name, --set-type, --set-list-name, --set-list-type and the utility option --split as well as workflow --priority and --demo
.

task set

In this example, we will illustrate a very similar scenario as the documented examples in the episodectl.exe priority documentation. Please read that before going ahead with this example. We begin with copying the MBR tasks, rename them, and set priority on them. We pretend that the 1Mbit task is our most important broadcast version which we always want to run first, then we want to run the 768kbit task for the Web, and last, we want to run the 512kbit task as a archive version. Remember, this is just a illustrative example!
$ cp tasks/H264\ 1Mbit\ TIFO.epitask tasks/Broadcast.epitask

$ cp tasks/H264\ 768kbit\ TIFO.epitask tasks/Web.epitask

$ cp tasks/H264\ 512kbit\ TIFO.epitask tasks/Archive.epitask

$ episodectl.exe tset tasks/Broadcast.epitask --name Broadcast --priority 3

$ episodectl.exe tset tasks/Web.epitask --name Web --priority 2

$ episodectl.exe tset tasks/Archive.epitask --name Archive --priority 1

To better see what's going on, we once again set the number of jobs in the node to 0
$ episodectl.exe node jobs --set 0

Then, open up a new shell window (Terminal window), resize it to have at least 150 columns so we can see the status well. In the new window, execute the command
$ episodectl.exe status tasks --all -wv

Go back to the first window and submit a workflow with the default workflow priority (0)
$ episodectl.exe ws -f source.mov -e tasks/Broadcast.epitask tasks/Web.epitask tasks/Archive.epitask -d output/

Output should look something like this (irrelevant columns are not present here...)

Filename	Task Name	Task Type	Priority

```
    Running (0 of 0)

_____________________________________________________________________

    Queued (6 of 6)

source.mov      Broadcast       Encode          3
source.mov      Web             Encode          2
source.mov      Archive         Encode          1
source.mov      output          Transfer        0
source.mov      output          Transfer        0
source.mov      output          Transfer        0
```

Submit a new workflow with priority 100

```
  $ episodectl.exe ws -f source.mov -e tasks/Broadcast.epitask tasks/Web.epitask
tasks/Archive.epitask -d output/ --priority 100
```

Output should now look something like this

```
  Filename          Task Name         Task Type    Priority
_____________________________________________________________________

    Running (0 of 0)

_____________________________________________________________________

    Queued (12 of 12)

source.mov      Broadcast       Encode          103
source.mov      Web             Encode          102
source.mov      Archive         Encode          101
source.mov      output          Transfer        100
source.mov      output          Transfer        100
source.mov      output          Transfer        100
source.mov      Broadcast       Encode          3
source.mov      Web             Encode          2
source.mov      Archive         Encode          1
source.mov      output          Transfer        0
source.mov      output          Transfer        0
source.mov      output          Transfer        0
```

Now we want to do as the last example in the episodectl.exe priority documentation, to have the workflow-internal priority have effect "outside" the workflow, to have all Broadcast tasks always run first, then Web tasks, and then Archive tasks, regardless of workflow priority. We still want a granularity of 100 different workflow priorities to prioritize the source files or monitors with, so we must set a task priority of more than 100 for the Archive task and the difference between the task priorities must also be more than 100... we use larger and more even numbers...

```
  $ episodectl.exe tset tasks/Broadcast.epitask --priority 600

  $ episodectl.exe tset tasks/Web.epitask --priority 400

  $ episodectl.exe tset tasks/Archive.epitask --priority 200
```

Now, remove all the old workflows

```
  $ episodectl.exe wp --all
```

And submit again with priority 100, then let's say 50, and finally 0

```
  $ episodectl.exe ws -f source.mov -e tasks/Broadcast.epitask tasks/Web.epitask
tasks/Archive.epitask -d output/ --priority 100
```

$ episodectl.exe ws -f source.mov -e tasks/Broadcast.epitask tasks/Web.epitask tasks/Archive.epitask -d output/ --priority 50

$ episodectl.exe ws -f source.mov -e tasks/Broadcast.epitask tasks/Web.epitask tasks/Archive.epitask -d output/ --priority 0

Output should now look something like this (deployments doesn't really matter, so they've been taken away...)

```
Filename        Task Name       Task Type   Priority
_____________________________________________________
   Running (0 of 0)

_____________________________________________________
  Queued (18 of 18)

source.mov      Broadcast       Encode       700
source.mov      Broadcast       Encode       650
source.mov      Broadcast       Encode       600
source.mov      Web             Encode      500
source.mov      Web             Encode      450
source.mov      Web             Encode      400
source.mov      Archive         Encode       300
source.mov      Archive         Encode       250
source.mov      Archive         Encode       200
```

Creating sources

There are currently 4 types of sources, a file source (a single file or a list of files), a monitor source, a EDL source, and a iseq source (Image Sequence).

Before we start with examples, we need some more source files. Up until now, we have only used the ~/EpisodeAPI/source.mov file but let's find some more source files and copy them into our input/ sub directory. In the following examples, the source files will be referred to as "file1.mov", "file2.mov", "file3.mov", and "file4.mov".

One more thing, since we are working with sources now (and not specifically tasks), we can create a simple workflow so we don't need to specify the individual tasks anymore, let's create the most simple workflow with a dummy source that we will replace with newly created sources

$ episodectl.exe ws -f DummySource -e encoder.epitask -d output/ -o workflows/ --name simple

Output:

Submission configuration written to:
/Users/tomasa/EpisodeAPI/workflows/simple.episubmission
$

source filelist

The creation function for a file source, i.e. the command source filelist exists mostly for consistency. The only use case I can think of is to organize source files in saved lists for later use, i.e. later submissions. Anyway, create some source file lists for different customers as an example

```
$ episodectl.exe sfl input/file1.mov input/file2.mov --name BatchForCustomerA -o input/

Output:
    Source configuration written to:
/Users/tomasa/EpisodeAPI/input/BatchForCustomerA.episource
    $

    $ episodectl.exe sfl input/file3.mov input/file4.mov --name BatchForCustomerB -o input/

Output:
    Source configuration written to:
/Users/tomasa/EpisodeAPI/input/BatchForCustomerB.episource
    $
```

Then, a few days later when we have forgotten which files belonged to which customer, we can submit the lists we created for each of our customers "A" and "B".

```
    $ episodectl.exe ws --file-source input/BatchForCustomerA.episource -s workflows/simple.episubmission

    $ episodectl.exe ws --file-source input/BatchForCustomerB.episource -s workflows/simple.episubmission
```

source monitor

Directory monitoring is a very popular feature, often called "Watch folders" or "Hot folders", even though there is nothing special with the directory (or "folder") but rather a configuration in the software that is going to monitor the directory. The monitor configuration in Episode is quite extensive and may look a bit frightening due to all the different filtering options but don't panic, everything is optional except for the directory we want to monitor.

Let's begin by creating a few direcotries for testing

```
    $ mkdir monitor
    $ mkdir monitor/projectA
    $ mkdir monitor/projectA/temporary
    $ mkdir monitor/projectB
    $ mkdir monitor/failed
    $ mkdir monitor/processed
```

Let's create a simple monitor for the directory ~/EpisodeAPI/input/. We are always located in ~/EpisodeAPI/ in these examples. Since we will only deal with a reliable storage here (the local disk), and use small source files, we speed up the reporting time of the monitor a bit from the default by lowering the safe-delay.

```
    $ episodectl.exe smon monitor/ --safe-delay 3 -o input/
```

Output:
Source configuration written to:
/Users/tomasa/EpisodeAPI/input/monitor.episource
$

You may notice that the name of the monitor became "monitor", that is because the default name given to a monitor is the directory name of the directory it will monitor, which in the example here happened to be "monitor". If you created a monitor for /Users/Shared/, the default generated name would be "Shared".

Now we can submit this monitor together with our "simple" workflow we created earlier
$ episodectl.exe ws --monitor input/monitor.episource -s workflows/simple.episubmission

To verify that the monitor is up and running, we can use the command status monitors (sm), we use the option --visual (-v) to get a human readable output
$ episodectl.exe sm -v

Example output:

MONI-8A344F14-6191-4812-8887-1C4786A894A2

Name: monitor

Submission name: monitor

URL:
tsrc:/Users/tomasa/EpisodeAPI/monitor/#node-id,ClientProxy%40computer
Workflow ID: WOFL-5183AEA6-72C7-4292-87ED-340D574B4239

Workflow Priority: 0

Running: yes

Stop reason:

To test the monitor, we may be interested in a couple of things, first status of course but maybe also a log of what the monitor is doing... now we need two additional shell windows. In the first window, run episodectl.exe st --all -wv and in the second one, run episodectl.exe mg which is short for monitor log.

To be able to see any result, we can link or copy a file into the monitored directory
$ ln source.mov monitor/

Example output from monitor log command:
MONI-8A344F14-6191-4812-8887-1C4786A894A2 (Info): New file discovered:
source.mov
MONI-8A344F14-6191-4812-8887-1C4786A894A2 (Info): File source.mov seems ripe, checking if we can open it

MONI-8A344F14-6191-4812-8887-1C4786A894A2 (Info): File source.mov seems ripe, reporting as ready
$

When dealing with monitors, it's important to be able to detect when something is wrong. The monitor boolean status "running" (yes/no) only indicates that the monitor is running (as the name suggests), it will not indicate that it is having problems until it has given up and stops itself. Problems in this case means having a problem listing the directory it is configured to monitor, or a sub directory if it is configured to list recursively. It may be due to a non-responding server (SMB/FTP/EpisodeIOServer), the directory is not mounted, the monitor (which runs as the user running Episode) does not have permission to list the directory, or of course that the directory does not in fact exist at all.

So, how long is the monitor running with problems before giving up?

Actually quite long by default. There are a number of configurations for controlling this, they all begin with "retry-". --retry-delay-start, --retry-delay-factor, --retry-delay-max , --retry-max-attempts. "max-attmpts" is the number of times the monitor will retry listing the directory and is default set to 40. "delay-start" is the initial delay in seconds before doing the next attempt, default 20 seconds. "delay-factor" is the integer factor to multiply the current delay (initially delay-start) with, default 2, i.e. double the delay. "delay-max" is the maximum delay in seconds that the monitor will wait, default 3600 seconds, i.e. 1 hour. Since doubling the waiting time each retry, we will reach the maximum in 40, 80, 160, 320, 640, 1280, 2560, 5120 bam!, max reached in 8 retries which means that we have 32 (minus one for a initial immediate retry) retries left which means that the monitor will default run for well over 30 hours before giving up!

To see how the monitor behaves, we create a monitor that will fail a little quicker than the default configuration. We configure it to monitor a non-existing directory and lower the retry options a bit
$ episodectl.exe smon bogus/ --retry-delay-start 2 --retry-delay-factor 1 --retry-max-attempts 2 -o input/

Output:
Source configuration written to: /Users/tomasa/EpisodeAPI/input/bogus.episource
$

Before we submit this monitor, let's do a couple of things. First, all errors (and not just errors...) are by default logged to ASL (Apple System Log) on OS X, and to Windows Event Log. In Windows, the log is viewable through the Windows Event Viewer. On OS X, the log is easist accessed through the application Console.app. In the Terminal, you can use the command syslog to query the log in different ways, for example, to get all log messages sent by Episode, you can use the command ("Seq" means "Substring equals", see man syslog)
$ syslog -k Sender Seq Episode

We can also let the node create log files for the monitors, let's try that too
$ episodectl.exe node log --monitors yes

Now we are ready to submit the monitor
$ episodectl.exe ws --monitor input/bogus.episource -s workflows/simple.episubmission

This monitor should give up after about 6 seconds but error messages should show up in the logs during that time

Let's see how the log file looks now (you will have a different filename than shown below)
$ cat ~/Library/Logs/Episode/Monitors/bogus-\[A2F0BA0CA058\].log

Output:
[Some date] (INFO) Starting up Monitor Plug-In
[Some date] (WARNING) Error while trying to list directory '' (Failed to list /Users/tomasa/EpisodeAPI/bogus (No such file or directory)). 2 tries left
[Some date] (WARNING) Error while trying to list directory '' (Failed to list /Users/tomasa/EpisodeAPI/bogus (No such file or directory)). 1 tries left
[Some date] (WARNING) Error while trying to list directory '' (Failed to list /Users/tomasa/EpisodeAPI/bogus (No such file or directory)). 0 tries left
[Some date] (ERROR) Failed to list directory '' (Reached max retries, giving up)
[Some date] (ERROR) Failed to list directory '' (Reached max retries, giving up)

If watching monitor status now, the output should look something like this
$ episodectl.exe sm -v

Example output:

MONI-73C5E364-3370-4F1A-9BBD-DC5843AF2F04

Name: bogus

Submission name: bogus

URL: tsrc:/Users/tomasa/EpisodeAPI/monitor/#node-id,ClientProxy%40computer
Workflow ID: WOFL-3B244A04-DBCC-45A8-875D-4EAB0177F3AA

Workflow Priority: 0

Running: no

Stop reason: Failed to list directory '' (Reached max retries, giving up)

I think we're done with the bogus monitor now, remove it (using the monitor remove command and option --name with argument "bogus")
$ episodectl.exe mr -n bogus

Now let's try and use our project directories we created earlier. We want to monitor "projectA" and "projectB" directories. We also want to be able to have sub directories in the project directories that the monitor should ignore, the directory "temporary" is used to illustrate this. In addition to that we want the processed files to be moved into one of the directories "processed" or "failed", depending on the transcoding and deployment result. The monitor must be informed to not report any files in those directories either since that would make the monitor report files indefinitely, just like if a "Same as Source" deployment is used with a monitor, no good idea!

So, how do we tell the monitor to ignore those directories ("failed" and "processed",

and every project sub directory)?

Let's start with the project sub directories. We basically have three options, either we can set the "recursion depth" of the monitor to exactly 1 (which includes the project directories, but nothing deeper). The word and the monitor option --recursive in this case means recursively traverse down a directory structure for a specified number of recursions, i.e reach a specified depth in the structure. The second option is to add ignore-filters that mathces the sub directory names (or files). The third option is to add include-filters that matches the project directories. We will try option 1 and 2, but let's start with the most simple, recursion depth of 1. Then we must add ignore-filters for the directories "processed" and "failed" because they are also at directory depth 1.

```
$ episodectl.exe smon monitor/ --recursive 1 --workflow-failure deploy
--move-source monitor/failed/ --workflow-success --move-source monitor/processed/
--directory-is-ignore processed failed  --safe-delay 3 -o input/
```

Submit the monitor and a workflow

```
$ episodectl.exe ws --monitor input/monitor.episource -s
workflows/simple.episubmission
```

We copy the source.mov file into each directory, renaming them at the same time. Hopefully, only the project files are reported, and when the source files are moved, they should be ignored.

```
$ cp source.mov monitor/projectA/sourceA.mov; cp source.mov
monitor/projectB/sourceB.mov; cp source.mov monitor/projectA/temporary/temp.mov
```

Here is a example output from the episodectl.exe monitor log command we started earlier (removed some irrelevant lines)

```
MONI-CE48696A-D394-4F17-ADE3-3C0C1C856469 (Info): New file discovered:
projectA/sourceA.mov
MONI-CE48696A-D394-4F17-ADE3-3C0C1C856469 (Info): New file discovered:
projectB/sourceB.mov
MONI-CE48696A-D394-4F17-ADE3-3C0C1C856469 (Info): File projectA/sourceA.mov
seems ripe, reporting as ready
MONI-CE48696A-D394-4F17-ADE3-3C0C1C856469 (Info): File projectB/sourceB.mov
seems ripe, reporting as ready
MONI-CE48696A-D394-4F17-ADE3-3C0C1C856469 (Info): New file discovered:
processed/sourceA.mov
MONI-CE48696A-D394-4F17-ADE3-3C0C1C856469 (Info): File projectA/sourceA.mov
removed
MONI-CE48696A-D394-4F17-ADE3-3C0C1C856469 (Info): File processed/sourceA.mov
seems ripe, reporting as ready
MONI-CE48696A-D394-4F17-ADE3-3C0C1C856469 (Info): Ignore file:
processed/sourceA.mov (matching ignore: 'directory-is')
MONI-CE48696A-D394-4F17-ADE3-3C0C1C856469 (Info): New file discovered:
processed/sourceB.mov
MONI-CE48696A-D394-4F17-ADE3-3C0C1C856469 (Info): File projectB/sourceB.mov
removed
MONI-CE48696A-D394-4F17-ADE3-3C0C1C856469 (Info): File processed/sourceB.mov
seems ripe, reporting as ready
MONI-CE48696A-D394-4F17-ADE3-3C0C1C856469 (Info): Ignore file:
processed/sourceB.mov (matching ignore: 'directory-is')
```

Now, we try and use the filtering instead. Notice that filtering is case sensitive so if, for example, we want to allow project directories to be named "Project..." as well as "project..." we must specify both versions (notice though that in this

particular scenario, the previous option with limited recursion depth is preferable). If we specify that we will only include directories that contains the words "Project" or "project", we can skip the ignore directive for "processed" and "failed" since those names does not contain "project"... The next problem is to ignore the "temporary " directory. Since the directory filterings matches all directories in the realtive path from the monitored base path, i.e. matches both "projectA" and "temporary" in case of finding a file there, it will match the directory-contains-include filter. There are two options (if we don't want to use file name or extension filters), either to explicitly ignore the directory name "temporary" or change the directory structure to always place files for Episode in a sub directory with some specific name, "video", "Episode ", "transcode", for example.
First remove the old monitor, we remove all monitors for simplicity (command monitor remove option --all)

```
$ episodectl.exe mr -a
```

Create new monitor (overwrite the previous one)

```
$ episodectl.exe smon monitor/ --recursive 5 --workflow-failure deploy
--move-source monitor/failed/ --workflow-success --move-source monitor/processed/
--directory-contains-include project --directory-is-ignore temporary --safe-delay 3
-o input/
```

Submit the monitor and a workflow

```
$ episodectl.exe ws --monitor input/monitor.episource -s
workflows/simple.episubmission
```

Remove the old processed files...

```
$ rm monitor/processed/*
```

Copy the files again, which will overwrite the old files which the monitor will notice

```
$ cp source.mov monitor/projectA/sourceA.mov; cp source.mov
monitor/projectB/sourceB.mov; cp source.mov monitor/projectA/temporary/temp.mov
```

Here is a example output from the episodectl.exe monitor log command we started earlier (removed some irrelevant lines)

```
MONI-F0049321-6147-4A9A-A04C-91E3CF810E49 (Info): File
projectA/temporary/temp.mov updated
MONI-F0049321-6147-4A9A-A04C-91E3CF810E49 (Info): File projectA/sourceA.mov
updated
MONI-F0049321-6147-4A9A-A04C-91E3CF810E49 (Info): File projectB/sourceB.mov
updated
MONI-F0049321-6147-4A9A-A04C-91E3CF810E49 (Info): File
projectA/temporary/temp.mov seems ripe, reporting as ready
MONI-F0049321-6147-4A9A-A04C-91E3CF810E49 (Info): Ignore file:
projectA/temporary/temp.mov (matching ignore: 'directory-is')
MONI-F0049321-6147-4A9A-A04C-91E3CF810E49 (Info): File projectA/sourceA.mov
seems ripe, reporting as ready
MONI-F0049321-6147-4A9A-A04C-91E3CF810E49 (Info): File projectB/sourceB.mov
seems ripe, reporting as ready
MONI-F0049321-6147-4A9A-A04C-91E3CF810E49 (Info): New file discovered:
processed/sourceA.mov
MONI-F0049321-6147-4A9A-A04C-91E3CF810E49 (Info): New file discovered:
processed/sourceB.mov
MONI-F0049321-6147-4A9A-A04C-91E3CF810E49 (Info): File projectA/sourceA.mov
removed
MONI-F0049321-6147-4A9A-A04C-91E3CF810E49 (Info): File projectB/sourceB.mov
removed
```

```
    MONI-F0049321-6147-4A9A-A04C-91E3CF810E49 (Info): File processed/sourceA.mov
seems ripe, reporting as ready
    MONI-F0049321-6147-4A9A-A04C-91E3CF810E49 (Info): Ignore file:
processed/sourceA.mov (not matching include: 'directory-contains')
    MONI-F0049321-6147-4A9A-A04C-91E3CF810E49 (Info): File processed/sourceB.mov
seems ripe, reporting as ready
    MONI-F0049321-6147-4A9A-A04C-91E3CF810E49 (Info): Ignore file:
processed/sourceB.mov (not matching include: 'directory-contains')
```

source edl

```
    EDL is short for "Edit Decision List" and can be used for a number of things,
    concatenate clips, remove sections of clips, or insert clips into another clip, etc.
    Let's start with a simple concatenation example, concatenate four files and save the
    source in the input/ sub directory

    Create a EDL source
      $ episodectl.exe sedl --clip input/file1.mov --clip input/file2.mov --clip
    input/file3.mov --clip input/file4.mov -o input/ --name EDLConcat

    Output:
      Source configuration written to:
    /Users/tomasa/EpisodeAPI/input/EDLConcat.episource
      $

    Submit the EDL source with the simple workflow
      $ episodectl.exe ws --edl input/EDLConcat.episource -s
    workflows/simple.episubmission

    The "spawn value" of the spawned workflow will be the first file in the list, so for
    example the default naming convention will create a output file that is named "file1-encoder.mov
    " and the CLI's status will show "file1.mov" in the "Filename" column for all tasks in
    this workflow.

    This example will cut out portions of a single file, seconds are used as time
    specifications
      $ episodectl.exe sedl --clip input/file1.mov --in 1.0 --out 2.0 --clip
    input/file1.mov --in 5.2 --out 6.4 --clip input/file1.mov --in 9.5 -o input/ --name
    EDLCut

    This example will insert two seconds of file2.mov and file3.mov into the file1.mov
    at the spots cut out as in previous example
      $ episodectl.exe sedl --clip input/file1.mov --in 1.0 --out 2.0 --clip
    input/file2.mov --out 2.0 --clip input/file1.mov --in 5.2 --out 6.4 --clip
    input/file3.mov --out 2.0 --clip input/file1.mov --in 9.5 -o input/ --name EDLInsert
```

source iseq

```
    The Image Sequence source can take a list of image sequences. Each sequence is
```

always specified with the first file in the sequence that is going to be encoded.
Since it's beyond the scope of the examples to provide a testable case for image
sequences, I will use the path /path/to/ instead of the created path's for previous
examples

Create a ISEQ source with two sequences
    $ episodectl.exe siseq /path/to/sequence1/file_0001.dpx
/path/to/sequence2/file_0001.tga -o input/ --name TwoSequences

Output:
    Source configuration written to:
/Users/tomasa/EpisodeAPI/input/TwoSequences.episource
    $

Submit the ISEQ source with the simple workflow
    $ episodectl.exe ws --iseq input/TwoSequences.episource -s
workflows/simple.episubmission

Advanced Example

The Image Sequence source is basically a helper for URL creation. What is actually
run in the workflow is a URL constructed with the scheme file+iseq and other options
encoded into the query part of the URL. If you only use the actual start file of a
image sequence without other options, you can specify a URL with the "file+iseq"
scheme as a regualr file URL and use in other sources or directly in the workflow
submit command.

Here is an example of creating a EDL source that concatenates three image sequences
(the URLs have empty authority parts, hence the three slashes)
    $ episodectl.exe sedl --clip file+iseq:///path/to/sequence1/file_0001.dpx --clip
file+iseq:///path/to/sequence2/file_0001.dpx --clip
file+iseq:///path/to/sequence3/file_0001.dpx -o input/ --name ISEQ-EDL

Example of submitting a image sequence directly
    $ episodectl.exe ws -f file+iseq:///path/to/sequence/file_1234.dpx -s
workflows/simple.episubmission

Submitting workflows
==========

We have been submitting a large number of workflows in the previous examples so let's skip
the basic commands for doing so, and instead talk about the actual command options and the
important ones in particular.

There are bascically two versions of this command, one for building a workflow and
optionally submit it or save it, and one for submitting a pre-built workflow. Submitting
pre-built workflows lets you re-use commonly used workflows saved on disk and optionally
have the source substituted when submitting it. It also supports submitting multiple
workflows in one command and it is slightly more efficient and requires less cluttered
commands than re-building the workflow in each submit command.

Submitting pre-built workflows is done with option --submission, or -s for short.

Building workflows on the fly, and either submit them or save them for later use is done with options
--encoder or --mbr
--destination
--execute
--mail
--workflow-failure
--workflow-success

A accompanying source is always required when submitting workflows. When submitting pre-built workflows, the source is optionally replaced and for on-the-fly workflows, the source must always be specified. If building a workflow on the fly which is saved to disk instead of submitted, a "dummy source" can be specified and later substituted with a "real" source when submitted using option -s.

To illustrate the concept I talked about, I will use the MBR tasks created in the task mbr example section ("HLS.epitask" and "MSS.epitask") and the regular "encoder.epitask" file. Let's pretend that we have customers "A", "B", and "C". Customer A wants Apple's HTTP Live Streaming packages. Customer B wants Microsoft's Smooth Streaming packages, and customer C only wants our example "encoder.epitask" files.

Instead of building each workflow for each of our source files, we create/build the workflows one time and save them for later. We insert a dummy source called "DummySource" in each workflow. We create a default deployment by specifying the path to our "output" directory (as opposed to creating a Transfer task in advance). We also insert a custom naming convention by overriding the default one, using the helper option --naming instead of configuration variable insertion. Then we specify that we will save the workflows in the workflows/ directory instead of submitting them and name them appropriately. Finally we use the Episode disk file format "bin" to further optimize submission of the thousands of source files we (maybe) are going to submit!

We would like to do one more thing, and that is to place each customers' output in separate sub directories. To do that, we could have created a separate Transfer task for each customer in advance with the command task transfer and one of the directory creation options, for example, option --dest-sub-dirs .... We could also have used one of the configuration variable override/insertion options, for example, option --set-type Transer ... in the submit command if we submitted the workflow directly, but since we are saving the workflows, and since dynamic variable override directives are not stored in saved workflows, we will insert them when we actually submit them.

```
   $ episodectl.exe ws -f DummySource --mbr tasks/HLS.epitask tasks/H264\ 1Mbit\
TIFO.epitask tasks/H264\ 768kbit\ TIFO.epitask tasks/H264\ 512kbit\ TIFO.epitask -d output/
--naming '$source.filename$-$mbr.name$' --name CustomerA -o workflows/ --format bin

Output:
   Submission configuration written to:
/Users/tomasa/EpisodeAPI/workflows/CustomerA.episubmission
   $

   $ episodectl.exe ws -f DummySource --mbr tasks/MSS.epitask tasks/H264\ 1Mbit\
TIFO.epitask tasks/H264\ 768kbit\ TIFO.epitask tasks/H264\ 512kbit\ TIFO.epitask -d output/
--naming '$source.filename$-$mbr.name$' --name CustomerB -o workflows/ --format bin

Output:
   Submission configuration written to:
/Users/tomasa/EpisodeAPI/workflows/CustomerB.episubmission
   $
```

```
   $ episodectl.exe ws -f DummySource -e encoder.epitask -d output/ --name CustomerC -o
workflows/ --format bin

Output:
   Submission configuration written to:
/Users/tomasa/EpisodeAPI/workflows/CustomerC.episubmission
   $
```

Now we want to encode our "source.mov" file with one command and place the output in separate directories. How do we do that? We must use some variable, something that is different in each workflow. We have already named the workflows as we want the sub directories to be named so that seems to be the perfect fit! We could simply use the variable $workflow.name$ for the directory creation.

> Technical note: Workflows and Submissions
> ---
>
> A "workflow" and a "submission" seems very similar at first glance, but they differ technically. A "workflow" only contains tasks, but not a source. A workflow must always be submitted together with a accompanying source, which will result in a "submission ". What is always saved on disk is a submission and that's the reason for why we have to specify a "dummy source" when saving workflows like in this example (which are actually submission files).
>
> Since this distinction exists, there are also two different names we can use, the workflow name and the submission name. By default, if not explicitly specified, the workflow name becomes the same as the submission name. When submitting or saving workflows (submission files) the option --name is the name of the submission, and also the name of the output file (.episubmission file), while the option --workflow-name could be used to explicitly name the workflow, if need be.
>
> So, the two variables $workflow.submission-name$ and $workflow.name$ will by default have the same value, unless, as stated, the workflow is explicitly given a different name with option --workflow-name

Ok, let's submit the three workflows, override the dummy source, and insert the variable $workflow.name$ to be used for direcotry creation (If you're on OS X only, you might want to skip CustomerB)

```
   $ episodectl.exe ws -s workflows/CustomerA.episubmission
workflows/CustomerB.episubmission workflows/CustomerC.episubmission -f source.mov
--set-list-type Transfer dest-sub-dirs $workflow.name$
```

There is often a requirement to wait for the workflow to finish. Because of that, option --wait exists. As the option description states, the CLI will exit with code 0 if the workflow finishes successfully and code 1 if anything was not successful.

There is a script in the CLI example directory. The script could be executed with the same arguments as to the CLI. The script will add the option --wait and if anything fails, it will use the status tasks command to get and print a error message. Let's try to run the script with the encoder task as a source file, just to get an error. Also specify a bogus deployment to get another error

```
   $ ruby /Applications/Episode.app/Contents/MacOS/engine/API/CLI/SubmitWorkflowWaitExit.rb
ws -e encoder.epitask -f source.mov encoder.epitask -d output/ /Bogus/
```

```
Output:
  Task "Bogus" failed: "Failed to list /Bogus (No such file or directory)"
  Task "encoder" failed: "initialization failed (no importer found)"
  $
```

To be able to get the status or to enable post-submission (run-time) control of the workflow(s) we submit, we need to get some ID or IDs back from the submission command. Options --id-out, --ids-out, and --monitor-id-out are used for that purpose, just as the previous example script does. More examples of this in following sections.

Getting status

Status can be retrieved for individual tasks or for workflows. Querying status for tasks makes it possible to get very detailed information while querying workflow status is very efficient and only the actual status can be queried. Before showing examples of integration usage, let's look at the human readble status of the CLI, or "visual" status as the options are called.

We have used these options in previous examples but let's try it again. We begin with just using the option --visual (or -v for short) to the status tasks command (or st for short)

```
$ episodectl.exe st -v
```

This gives status for all tasks in "active" workflows, i.e. non-finished or "non-historized" workflows (which probably is just empty if trying now). The command by default uses the option --all if no IDs are specified. To be able to also see historized workflow's tasks, we must use the argument history to option --all, let's try that

```
$ episodectl.exe st --all history -v
```

Now, if previous examples were tested, within the last hour, it should give a decent list of tasks.

> Why within the last hour?
> ---
>
> When the ClientProxy is contacting a Node (submitting to, getting status from, etc.) it checks if it already have a open connection to that target and if it has, it also has that Node's information cached. If it does not have a open connection to the target Node, it will create one. When the connection to the target Node is open and active, all updates are broadcasted by the Node to each connected ClientProxy so the ClientProxy always has correct information to respond with to the CLI or to the XMLRPC server. The Node does not keep any history in memory, it only saves history to disk (if configured to save history at all), while the ClientProxy keeps history in memory for the configured history-keep-time amount of seconds, which default is 3600, or one hour.
>
> There is one more configurable time in the ClientProxy, a connection-keep-time. This time is also default 3600 seconds. When a connection times out, the saved history for it will also go away and therefore, the history-keep-time shold not be configured to be longer than the connection-keep-time. Both times are reset whenever there is a request on it from either the CLI, GUI, or XMLRPC server.

There is a option that can be used together with the visual status to get a continuous

status feedback with updates, option --wait (or -w for short), for example
$ episodectl.exe st --all history -w -v
$ episodectl.exe st -wv

To get the best total status overview of Episode is to open two Terminal windows, resize them so they share the full screen horizontally (i.e. splits the screen vertically) and in the left window, run
$ episodectl.exe st --all -wv

...and in the right window, run
$ episodectl.exe st --all history-only -wv

If you are on OS X, there is a example Apple script that sets up two Terminal windows with nice Episode-colors, fonts etc. and executes the above commands in them. It also pops up a list of clusters on the network for you to choose from. Open the script up in "AppleScript Editor" (which is default)
$ open /Applications/Episode.app/Contents/MacOS/engine/API/Apple/StatusTerminal.scpt

Then hit the "Run" button to test it. Notice that you can save it as a "Application" so you can double-click it later.

Here is a example script that pretends that a couple of source files are submitted together with a workflow. If any of the source files (any spawned workflow) fails, then all of the rest are aborted. The script requests the template workflow ID and polls status with that ID until it succeeds or fails. The script first passes all arguments to episodectl, so let's do a submission
$ ruby /Applications/Episode.app/Contents/MacOS/engine/API/CLI/PollTemplateStatusAbortOnFailure.rb ws -e encoder.epitask -d output/ -f source.mov source.mov

Example output successful run:
Idle at the moment...
Running...
Running...
Running...
Running...
Running...
Running...
Running...
Idle at the moment...
Running...
All finished successfully
$

Example output non-successful run:
Idle at the moment...
Running...
Something failed, aborting
$

The next example script uses status tasks command to kind of mimic the CLI's visual status output
$ ruby /Applications/Episode.app/Contents/MacOS/engine/API/CLI/StatusTasks.rb

Example output:
Filename Task Name Task Type Priority

```
 Status              Message

-------------------------------------------------------------------------------------------------------------------------------

 clip1.mov           encoder             Encode              5
[================== ]  Running
 clip1.mov           output              Transfer            5
Idle                 Waiting for...o transfer
 clip2.mov           encoder             Encode              5
[=============      ]  Running
 clip2.mov           output              Transfer            5
Idle                 Waiting for...o transfer
 clip3.mov           encoder             Encode              5
[------------------ ]  Running
 clip3.mov           output              Transfer            5
Idle                 Waiting for...o transfer
 clip4.mov           encoder             Encode              2
[----------------   ]  Running
 clip4.mov           output              Transfer            2
Idle                 Waiting for...o transfer
 clip5.mov           output              Transfer            2
Idle                 Waiting for...o transfer
 clip5.mov           encoder             Encode              2
[------------       ]  Running
 clip6.mov           output              Transfer            0
Idle                 Waiting for...o transfer
 clip6.mov           encoder             Encode              0
Queued               No free Enc... available
 clip7.mov           output              Transfer            0
Idle                 Waiting for...o transfer
 clip7.mov           encoder             Encode              0
Queued               No free Enc... available
 $
```

Using the XML-RPC Interface

This chapter describes Episode's XML-RPC interface.

The following topics are covered:

Note: When utilizing the CLI to execute unlicensed features in demo mode, add the `-demo` flag. In the XML-RPC interface, you can add `-demo` to *submitSubmisssion* and *submitBuildSubmission* to use unlicensed features in demo mode as well. For license requirements, see XML-RPC and CLI License Requirements.

Overview

The XML-RPC server is enabled by default and ready to use as a server for external integration. On its host node, however, it is a client to the Episode system and has the same role as the GUI client.

In the XML-RPC server, users may target nodes other than the local node—providing multiple ways to use the server to target other Episode nodes/clusters.

Episode uses Bonjour to find the XML-RPC servers and relate them to cluster and nodes, and targets different XML-RPC servers when targeting different clusters/nodes. This means that any cluster or private node having an active XML-RPC server is reachable.

Alternatively, you can use an XML-RPC server as the proxy for all calls to any Episode cluster. This XML-RPC server can run locally on the client (as long as Episode has been installed), on a dedicated server or on another server in one of the clusters that has been configured.

When sending method calls to the XML-RPC server, you can specify *target-node-info* to target clusters other than the local cluster/node where the XML-RPC server is running. In this case the XML-RPC server will only be able to target clusters and the local node, not other private nodes. This approach is easier from an implementation standpoint and may be the most intuitive way of starting XML-RPC interaction with Episode. All traffic will be routed through this server and if the integration is sensitive to network load or if the system relies on dedicated network setups for different clusters this option is probably not the best approach.

Restart the XML-RPC Service

If you should need to do so, you can restart the XML-RPC service using the Episode command line interface. To restart the service, open a terminal window (MacOS), or a command prompt window (Windows), and run the following command:

[MacOS]: /Applications/Episode.app/Contents/Resources/engine/bin/episodectl launch restart -x

[Windows 32-bit]: C:\Program Files\Telestream\Episode6\bin\episodectl launch restart -x

[Windows 64-bit]: C:\Program Files (x86)\Telestream\Episode6\bin\episodectl launch restart -x

Communicating with Episode via the XML-RPC API

To communicate with Episode via the XML-RPC API, you need to use an XML-RPC client library in your program.

The library you choose depends on (among other things), the language you're using to write your client programs.

XML-RPC libraries handle low-level HTTP request/response communications with the Episode XML-RPC server, and package method calls and returns into standardized XML-RPC message structures so they can be easily integrated with your program, in the language of your choice.

If you are not familiar with developing XML-RPC-based client programs, please see http://www.xmlrpc.com for information on the XML-RPC standard.

The following XML-RPC client libraries have been tested with Episode:

- **Redstone XML-RPC Library:** http://xmlrpc.sourceforge.net/
  - Language: Java
  - Platform: N/A (Independent)
  - License: LGPL
- **Cocoa XML-RPC Framework:** http://github.com/corristo/xmlrpc
  - Language: Objective C
  - Platform: MacOS, iOS
  - License: MIT
- **XML-RPC.NET:** http://www.xml-rpc.net/
  - Language: .NET
  - Platform: Microsoft Windows
  - License: MIT X11

Overview of XML-RPC File Structure

Episode XML-RPC API files use the elements described in this section.

Note: Other files may be referenced to define complex parameter structures as specified by an *inherit* attribute. These parameters expect a data structure that is defined in another constraint XML as their value. The name of the XML containing the constraint definition for these values is cited in a comment above the parameter's constraint tag.

Example

Each XML-RPC method is defined by a command element. The child nodes of the command element define the method's parameter and return structures. This structure consists of the following element hierarchy:

Figure 12. Typical XML-RPC method <command> element

```
<command ... <!-- The method --> >
  <send> <!-- The parameters -->
    <constraint ... >
      ...
    </constraint>
    ...
  </send>
  <reply> <!-- The returns -->
    <constraint ... >
      ...
    </constraint>
  </reply>
     ...
</command>
```

High-level Element Definitions

<command>: Defines a method.

Elements

<name>: The internal Episode method namespace

<send>: Defines the method's parameter structure

<constraint>: Defines a single key/value pair argument (hash map)

Attributes

property-name: Argument key

compact: Argument value data type

inherit: Argument value's inherited data type for complex data types

optional: Signifies whether or not this argument is required

<reply>: Defines method's return structure (hash map)

<option>: Defines one possible set of key-value pairs in an exclusive set

<constraint>: See above

Commands and Constraints

Command name attributes specify the internal method in the Episode namespace. The public XML-RPC method names are not the same. The public name is also the value of the <XMLRPC> element in command_doc.xml.

Method parameter and return structures always have an XML-RPC hash map (called a <struct> element) as their top level element. This <struct> element contains a set of key/value pairs that adhere to the constraint definitions for that method.

Constraints define the keys that will or can be present in the map, as well as the expected data type of their values. Complex value structures can be defined either using a multi-level 'compact' attribute, or using an 'inherit' attribute. See Data Types for more details.

Special Cases

There are a few special cases with optional constraints and the 'target-node-info' constraint. This parameter and 2 of its nested values are invisibly optional, even though they do not specify an optional attribute.

For any command that accepts the 'target-node-info' complex data structure parameter, it can always be omitted. If omitted, the Client Proxy service will always direct the call to the local host.

Also, when building a target-node-info structure, the *iid* and *persistent* values in the target-node-info map can also be omitted. These values are used by Episode internally, and suitable defaults will be generated automatically if they are omitted.

For an example of the 'target-node-info' argument structure, see the Inherited complex data structures section.

Option Sets

<option> element sets can be found in both parameter and return structure definitions. These elements imply that only one of the structures in that set of <option> elements can or will be present.

Some definitions combine option sets with standard constraints.

An example of this can be found in the <reply> from the proxy.process.log.get command:

Figure 13. Constraint definition using option sets

```
<reply>
  <!-- Common options for all entities -->
  <constraint property-name="error" compact="type:string"
        optional="yes"/>
  <constraint property-name="log-to-file" compact="type:bool"
        optional="yes"/>
  <constraint property-name="log-to-file-report-verbosity"
        compact="range:int(0..7)" optional="yes"/>
  <option>
    <!-- Options for node|xmlrpc|io|proxy|assistant -->
    <constraint property-name="system-log" compact="type:bool"
```

```
        optional="yes"/>
    <constraint property-name="system-log-report-verbosity"
        compact="range:int(0..7)" optional="yes"/>
    <constraint property-name="log-directory"
        compact="type:string" optional="yes"/>
    <constraint property-name="stdout-stderr-re-direct"
        compact="type:bool" optional="yes"/>
    <constraint property-name="rotation-max-files"
        compact="range:int(1..)" optional="yes"/>
    <constraint property-name="rotation-max-size"
        compact="range:int(1024..)" optional="yes"/>
  </option>
  <option>
    <!-- Options for watch folders -->
    <constraint property-name="rotation-max-files"
        compact="range:int(1..)" optional="yes"/>
    <constraint property-name="rotation-max-size"
        compact="range:int(1024..)" optional="yes"/>
  </option>
  <option>
    <!-- Options for tasks -->
    <constraint property-name="max-files"
        compact="range:int(1..)" optional="yes"/>
    <constraint property-name="clean-interval"
        compact="range:int(5..604800)" optional="yes"/>
  </option>
</reply>
```

In this example, the three constraints at the top of the reply (*error*, *log-to-file*, and *log-to-file-report-verbosity*) are not part of the option set. The presence of these constraints follows the same rules as constraints in any other <send> or <reply> block. However, only one of the value sets contained in the following 4 <option> blocks can be present.

This means that in a <struct> returned from this method, the *error*, *log-to-file*, and *log-to-file-report-verbosity* keys could always be present. However, if the *rotation-max-files* key was also present, the only other key that could exist in the map would be *rotation-max-size* (because it is defined in the same <option> block as *rotation-max-files*). Any keys defined in other options blocks would not be allowed in this return.

Tag Name Mappings

The tag names used to define data structures in the constraint definitions can usually be directly mapped to XML-RPC message structure tags as follows:

Constraint tags: <send>, <reply>, <db>, <dbmv>.
<dbmv> is a unique case. See below for details.
XML-RPC message element: <struct> (hash map)

Constraint tag: <list>
XML-RPC message element: <array>

See Primitive Data Types for mappings of primitive data types as values.

Note: <dbmv> is a unique <struct> definition used in returns. These <struct> elements use a variable keyset, rather than a fixed keyset defined by constraints. In these cases, both the key, and its value contain data that is part of the return. Unless you obtained the key for which you are looking for a value in one of these <struct> elements in a previous call, you will need to iterate the pairs to retrieve the desired data, rather than specifying a key to lookup in the map.

An example of this can be seen in the clusters constraint for the return from the *proxy.network.info.bonjour* command:

Figure 14. Constraint definition using <dbmv> tags

```
<constraint property-name="clusters">
  <dbmv> <!-- key is cluster name -->
    <list>
      <db>
        constraint property-name="host" compact="type:string"/>
        <constraint property-name="host-IPv4"
          compact="type:string"/>
        <constraint property-name="host-IPv6"
          compact="type:string" optional="yes"/>
        <constraint property-name="port" compact="type:string"/>
        <constraint property-name="os" compact="type:string"/>
        <constraint property-name="id" compact="type:string"/>
        <constraint property-name="is-master"
          compact="type:bool"/>
        <constraint property-name="is-backup"
          compact="type:bool"/>
        <constraint property-name="num-nodes" compact="type:int"/>
        <constraint property-name="tsp-compatible"
          compact="type:bool"/>
      </db>
    </list>
  </dbmv>
</constraint>
```

In this case, the key for each pair in the returned <struct> is the cluster name string, and the value is an <array> of <struct> elements containing the system information values for each system in that cluster, as defined by the constraints. All usages of the <dbmv> element should be commented to specify the data that will be returned as the map's keyset.

Data Types

The data type of the values expected/returned by a parameter are defined in one of four ways:

- A compact attribute denoting a primitive data type
- A compact attribute denoting a complex data structure
- An inherit attribute denoting an inherited complex data type
- In-place in the XML as child nodes of the constraint element

Primitive Data Types

Primitive data types, like constraint child tags, can be directly translated to native XML-RPC data types and message elements. Below is a list of the compact attribute values for primitive data types, and mappings to their XML-RPC counterparts.

Table 9. Primitive data types in CLI and XML-RPC

XML-RPC Data Type	CLI Tag Name	XML-RPC Element
bool	Boolean	<boolean>
integer	Integer	<i4>
string	String	<string>
binary	Base 64 encoded	<base64>

Note: Episode's implementation of the XML-RPC server does not surround string values with <string> tags. The server will accept message with or without <string> tags around these values, but your client must be compatible with this message structure in order to properly communicate with the server.

In-place Complex Data Structure Definitions

Many constraints use in place definitions for complex data structure values. These structures are defined by a series of child nodes under the constraint element. These XML tag names can be directly translated to XML-RPC message elements using the mappings defined in Tag Name Mappings.

Here is an example of an in-place complex data structure definition:

Figure 15. Typical in-place complex data structure element

```
<constraint property-name="task-username-tags" optional="yes">
  <list>
    <db>
    <!-- This should be a user defined task name -->
      <constraint property-name="name" compact="type:string"/>
      <!-- The tag to set as a run requirement for the task -->
      <constraint property-name="tag" compact="type:string"/>
      <!-- This optional property indicates if the task should
        run or should NOT run on the specified tag (if the tag is
        present on the Node). The default is run (true). -->
      <constraint property-name="run" compact="type:bool"
optional="yes"/>
    </db>
  </list>
</constraint>
```

Using the information in the mappings section, we can build the XML-RPC message structure that would be sent for this parameter under the top level struct, adhering to this constraint definition:

Figure 16. XML-RPC argument structure

```
...
<member>
  <name>user-name-tags</name>
  <value>
    <array>
      <data>
        <value>
          <struct>
            <member>
              <name>name</name>
              <value>
                <string>Task Name</string>
              </value>
            </member>
            <member>
              <name>tag</name>
              <value>
                <string>sometag</string>
              </value>
            </member>
            <member>
              <name>run</name>
            <value>
              <Boolean>1</Boolean>
            </member>
          </struct>
        </value>
      </data>
    <array>
  </value>
</member>
...
```

Complex Data Structure Compacts

A compact can also specify a multi-level complex data structure. These compact values use a combination of constraint child node tag names, and primitive data type identifiers to specify a complex structure.

Here is an example of a complex compact value, and its translation to an XML-RPC message:

```
compact="type:list(type:string)"
```

This compact specifies a value consisting of a list of strings. We know that a list maps to an XML-RPC <array>, so the XML-RPC value structure for this argument would look something like this:

Figure 17. XML-RPC structure of a multi-level compact

```
<array>
  <data>
    <value>
      <string>Some String</string>
    </value>
      ...
  </data>
</array>
```

Inherited Complex Data Structures

Some constraints do not have a 'compact' attribute, but instead use an 'inherit' attribute. The inherit attribute denotes that this constraint expects a complex data structure for its value that is defined elsewhere. Constraints using an 'inherit' attribute should be commented with the location of the constraint definition for that complex structure.

We can see an example of this with the "target-node-info" constraint that is used by many of the commands:

Figure 18. Example of constraint using target-node-info data type

```
...
<!-- Info about target node to submit to - default localhost see
proxy-constraints.xml for description of target node info
structure -->
<constraint property-name="target-node-info"
     inherit="target-node-info"/>
...
```

Following the XML comment, we can find the definition of the target-node-info structure in 'proxy-constraints.xml':

Figure 19. Definition of target-node-info constraint

```
...
  <constraint property-name="target-node-info">
    <db>
      <constraint property-name="persistent"
         compact="type:bool"/>
      <constraint property-name="iid" compact="type:string"/>
      <!-- If neither host/port nor cluster is specified, the
local node is used regardless of its state. If it's a cluster
participant, get redirected to the master node. -->
      <constraint property-name="host" compact="type:string"
         optional="yes"/>
      <!-- If no port is specified, the default port is used -->
      <constraint property-name="port" compact="type:string"
         optional="yes"/>
      <!-- Try to find a node using bonjour -->
      <constraint property-name="cluster-name"
         compact="type:string" optional="yes"/>
      <constraint property-name="timeout" compact="type:int"
         optional="yes"/>
    </db>
  </constraint>
...
```

From here, we can treat any constraint specifying an 'inherit="target-node-info"' attribute as if it had an in-place complex data structure definition that matches that of the target-node-info constraint specified in another file.

Using the JSON-RPC Interface

This chapter describes Episode's JSON-RPC interface for direct programmatic control of the Episode feature set.

The following topics are covered:

Overview

The JSON-RPC interface is enabled by default and ready to use for external integration of Episode to other systems and software.

- The Episode JSON service starts on default **port 8080**.
- Available Episode JSON-RPC commands are essentially the same as the XML-RPC commands described in the chapter on Using the XML-RPC Interface.

JSON Basics

The JSON-RPC service conforms to JSON-RPC 2.0 Transport: HTTP. JSON, which stands for JavaScript Object Notation, provides a human readable data-interchange format for querying and controlling Episode. JSON uses a text format based on the JavaScript Programming Language. Although JSON is language independent, it follows conventions similar to the C-family of languages, including C, C++, C#, Java, JavaScript, Perl, Python, and others.

JSON employs two main data structures: *objects* and *arrays*.

Objects follow these conventions:

- Consist of an unordered list of name and value pairs.
- Begin with a left brace ({) and end with a right brace (}).
- Include a colon after each name (:).
- Separate name/value pairs with a comma (,).

Arrays follow these conventions:

- Comprise ordered collections of values.
- Begin with a left bracket ([) and end with a right bracket (]).
- Separate values by a comma (,).

Values can consist of a string in double quotes, a number, a true or false or null, an object, or an array. Nested values are permitted.

A string consists of a sequence of Unicode characters in double quotes, using backslash escapes. A character is represented as a single character string.

A number is very much like a C or Java number, except that the octal and hexadecimal formats are not used.

You can add whitespace between any pair of tokens.

For more information about the JSON standard, visit these web sites:

http://www.jsonrpc.org/

http://www.simple-is-better.org/json-rpc/transport_http.html

Programming Languages and Libraries

You can choose from many popular programming languages to work with the JSON-RPC API. To communicate with Episode via the JSON-RPC API, you will need to use a JSON-RPC client library in your program. The JSON web site includes lists of many available libraries:

http://www.jsonrpc.org/

JSON-RPC libraries handle low-level HTTP request/response communications with the Episode JSON-RPC service and package method calls and returns into standardized JSON-RPC message structures so they can be easily integrated with your program, in the language of your choice.

JSON-RPC File Structure

Episode JSON-RPC API files use the elements shown in the following definitions and examples. Also see the XML-RPC chapter for command descriptions.

High-level Element Definitions

The elements present in requests and responses are described below.

Note that every request must be made using the HTTP *POST* method with *Content-Type* set to *application/json*.

Request Elements

POST / HTTP/1.1: HTTP method required at the start of every request.

Host: Episode JSON-RPC server host address and port, such as localhost or 127.0.0.1.

Content-Length: Number of bytes in the content request/response.

Content-Type: application/json

Request Message: Described in the next topic below.

JSON Request Message Structure

Request Message: Includes a request, ID, and parameters in this pattern: {"jsonrpc":"2.0","method":"***request***","id":***number***,"params":{***param***}}

"jsonrpc": Always set to "2.0",

"method": Identifies the desired method to execute.

"id": Set to a unique id string or integer. If omitted, the request is treated as a notification and the server response is also omitted.

"params": Include any method parameters.

Response Elements

HTTP/1.1 200 OK: Response method and status—OK or error message.

Server: EpisodeJSONRPCServer identifies the responding service.

Connection: Status of the server connection.

Access-Control-Allow-Origin: Used by the client to enable cross-site HTTP requests. Asterisk (*) tells the client that is possible to access the server from any domain.

Access-Control-Allow-Headers: Indicates headers the Episode JSON service will accept.

Allow: Indicates methods the Episode JSON service will accept.

Content-Type: application/json; charset=UTF-8.

Content-Length: Number of characters in the response.

Response Message: Described in the next topic below.

JSON Response Message Structure

Response Message: Includes the request ID, the JSON version, and additional data in this format: {"id": 1, "jsonrpc": "2.0", "result": {"API": 2, "product": "6.5.0" } }

"id": Same unique id string or integer used in the request. If omitted in the request, the server omits it in the response also.

"jsonrpc": Always set to "2.0",

"result": Contains the method response. Present only if no error occurred.

"error": Contains an error object. Present only if an error occurred.

Example Requests with HTTP Headers and Responses

Each JSON-RPC method is defined by the structure shown in these examples:

Example getVersion

```
POST / HTTP/1.1
Host: localhost:8080
Content-Length: 58
Content-Type: application/json
{"jsonrpc":"2.0","method":"getVersion","id":1,"params":{}}

Response:
HTTP/1.1 200 OK
Server: EpisodeJSONRPCServer
Connection: close
Access-Control-Allow-Origin: *
Access-Control-Allow-Headers: Origin, X-Requested-With,
Content-Type, Accept
Allow: OPTION, POST
Content-Type: application/json; charset=UTF-8
Content-Length: 71
{"id": 1, "jsonrpc": "2.0", "result": {"API": 2, "product":
"6.5.0" } }
```

Example statusTasks2 with params

```
POST / HTTP/1.1
Host: localhost:8080
Content-Length: 74
Content-Type: application/json

{"jsonrpc":"2.0","method":"statusTasks2","id":3,"params":{"hist
ory":true}}

Response:
HTTP/1.1 200 OK
Server: EpisodeJSONRPCServer
Connection: close
Access-Control-Allow-Origin: *
Access-Control-Allow-Headers: Origin, X-Requested-With,
Content-Type, Accept
Allow: OPTION, POST
Content-Type: application/json; charset=UTF-8
Content-Length: 5006

{"id": 3, "jsonrpc": "2.0", "result": {"statuses": {"WOFL-
5DA24888-9AA5-406..
```

Program Examples

The following examples show test files created using the Ruby scripting language.

To run a Ruby script you will need to get Ruby from www.ruby.org.

Example Class for HTTP Calls—jsonrpc.rb file

The following jsonrpc.rb file is a small example JSON-RPC wrapper.

```
require 'net/http'
require 'json'

#  A class used to make JSONRPC calls over HTTP
class JSONRPC

  # Used to send a command on the server
  # string url for server
  # string method with server method
  # hash with params server parameters
  def self.call(url, method, params, id)
    @toSend = {
      "jsonrpc" => "2.0",
      "id" => id,
      "method" => method,
      "params" => params
    }.to_json
    return self.raw_post(url, @toSend)
  end

  # Used to execute a command on the server
  # string url for server
  # string method with server method
  # hash with params server parameters
  def self.notification(url, method, params)
    @toSend = {
      "jsonrpc" => "2.0",
      "method" => method,
      "params" => params
    }.to_json
    return self.raw_post(url, @toSend)
  end

  # Used to make raw json posts
  # string url for server
  # string json with arguments
  def self.raw_post(url, json)
    uri = URI.parse(url)
    http = Net::HTTP.new(uri.host, uri.port)
    req = Net::HTTP::Post.new(uri.path, initheader = {'Content-Type' =>'application/json'})
    req.body = json
    resp = http.request(req)
    return resp
  end

end
```

Example Test Version—jsonTestVersion.rb file

```
require "test/unit"
require 'net/http'
require 'pp'
require 'json'
require_relative 'jsonrpc'

class TestJSONVersion < Test::Unit::TestCase
  def setup

  end

  # Test case
  def test_version
    # Request id any identifier
    id = 1;

    # Make request url, method, params, id
    raw_response = JSONRPC.call("http://localhost:8080/", "getVersion", {}, id)

    # Parse json
    response = JSON.parse(raw_response.body)

    # Use pp to print response, uncomment line below
    # pp response

    # check if reply is ok
    assert(response.has_key?("jsonrpc"), "No key jsonrpc" )
    assert_equal("2.0", response["jsonrpc"], "Key jsonrpc MUST be '2.0'")
    assert_equal(id, response["id"], "Response id must be equal to the sent id")
    assert(response.has_key?("result"), "No result present in response")
  end

  # An error occurred on the server while parsing the JSON text.
  # -32600  Invalid Request The JSON sent is not a valid Request object.
  # -32603  Internal error  Internal JSON-RPC error.
  # -32000 to -32099  Server error  Reserved for implementation-defined server-errors.
  # -32601  Method not found  The method does not exist / is not available.
  def test_bugus
    id = "string id";
    res = JSONRPC.call("http://localhost:8080/", "dummyMethod", {}, id)
    response = JSON.parse(res.body)
    assert(response.has_key?("jsonrpc"), "No key jsonrpc" )
    assert_equal("2.0", response["jsonrpc"], "Key jsonrpc MUST be '2.0'")
    assert_equal(id, response["id"], "Response id must be equal to the sent id")
    assert(response.has_key?("error"), "No error present in response")
    assert_equal(-32601, response['error']['code'], "Expected error code -32601")
  end

  # -32700  Parse error Invalid JSON was received by the server.
  def test_parse_error
    res = JSONRPC.raw_post("http://localhost:8080/", "{{{\"d\":[osososososos], lpldpdl plp lpl
pldp lpdlp{{not valid json pp}}")
    response = JSON.parse(res.body)
    assert_equal(-32700, response['error']['code'], "Expected error code -32700")
  end

  def test_notification
    res = JSONRPC.notification("http://localhost:8080/", "getVersion", {})
    assert_equal(nil, res.body, "Expected empty response on notification")
  end

  def teardown
    #void
  end

end
```

Example Test Status Tasks2—jsonTestStatusTasks2.rb file

```
require "test/unit"
require 'net/http'
require 'pp'
require 'json'
require_relative 'jsonrpc'

class TestJSONStatusTasks2 < Test::Unit::TestCase
  def setup

  end

  # Test case
  def test_basic_statustasks2
    # Request id any identifier (string or number)
    id = 2;

    # Make request url, method, params, id
    raw_response = JSONRPC.call("http://localhost:8080/", "statusTasks2", {"history" => true}, id)

    # Parse json
    response = JSON.parse(raw_response.body)

    # Use pp to print response, uncomment line below
    # pp response

    # check if reply is ok
    assert(response.has_key?("jsonrpc"), "No key jsonrpc" )
    assert_equal("2.0", response["jsonrpc"], "Key jsonrpc MUST be '2.0'")
    assert_equal(id, response["id"], "Response id must be equal to the sent id")
    assert(response.has_key?("result"), "No result present in response")
    assert(response["result"].has_key?("statuses"), "No statuses present in result")
  end

  def teardown
    #void
  end

end
```

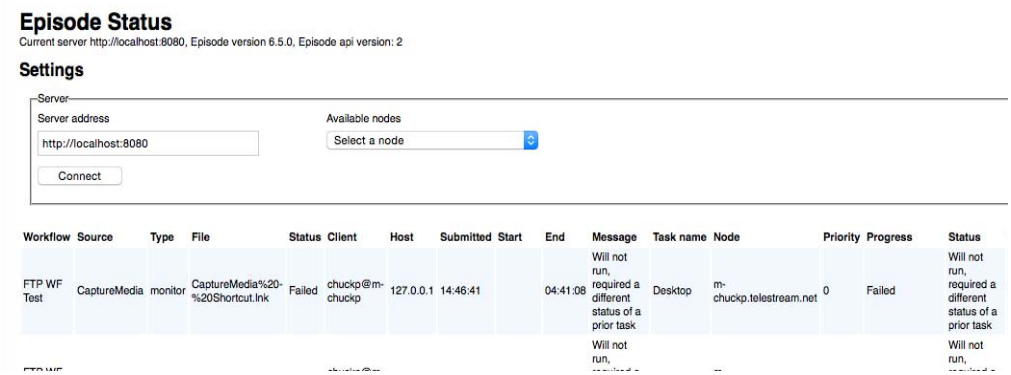
Demo Web Page with Job Monitoring

The JSON interface includes a Demo.html web page that provides a functioning job status monitoring feature.

To access the page:

1. Navigate to this location:
   - **Mac:** *Applications/Episode.app/Contents/Resources/engine/API/JSONRPC/HTML/demo.html*
   - **Win:** *C:\Program Files\Telestream\Episode 6\API\JSONRPC\HTML\demo.html*
2. Double-click the HTML file to open the Demo page in your default browser.
3. Enter the *server address* of your Episode installation.
4. *Select a node* from the Episode nodes listed in the *Available nodes* menu.
5. Click *Connect* to view the job status list.

Figure 20. Job Monitoring on the Demo Web Page



Index