Episode 7.4 Advanced User Guide

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: 221533

Publication Date: June 2017

Contents

Using the XML-RPC Interface 65

Using the JSON-RPC Interface 79

Preface

Support | Information | Assistance

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

Support Web Mail. www.telestream.net/telestream-support/episode/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 for those who are planning, developing, or implementing automated digital media transcoding and integration solutions with Episode.

This guide is written assuming that you possess a general working knowledge of digital media processing, of Episode, and that 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 user interface. For information about that, see the *Episode User 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 how to approach programming your own interface using the XML-RPC commands..

Using the JSON-RPC Interface

This topic explains the JSON interface and how to program your own interface to use the Episode JSON commands.

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
- JSON 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.

JSON Interface

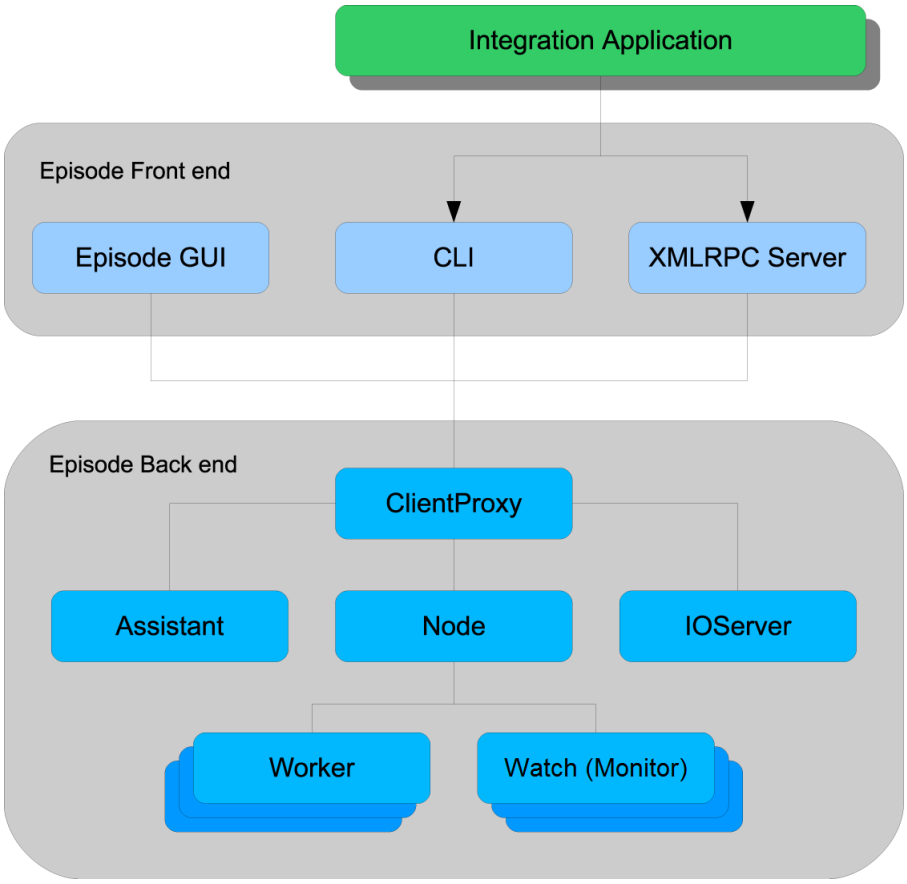
The JSON interface is very similar to the XML-RPC interface and uses many of the same commands, repackaged for the JSON framework. The last chapter in this guide covers the JSON interface.

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.

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.

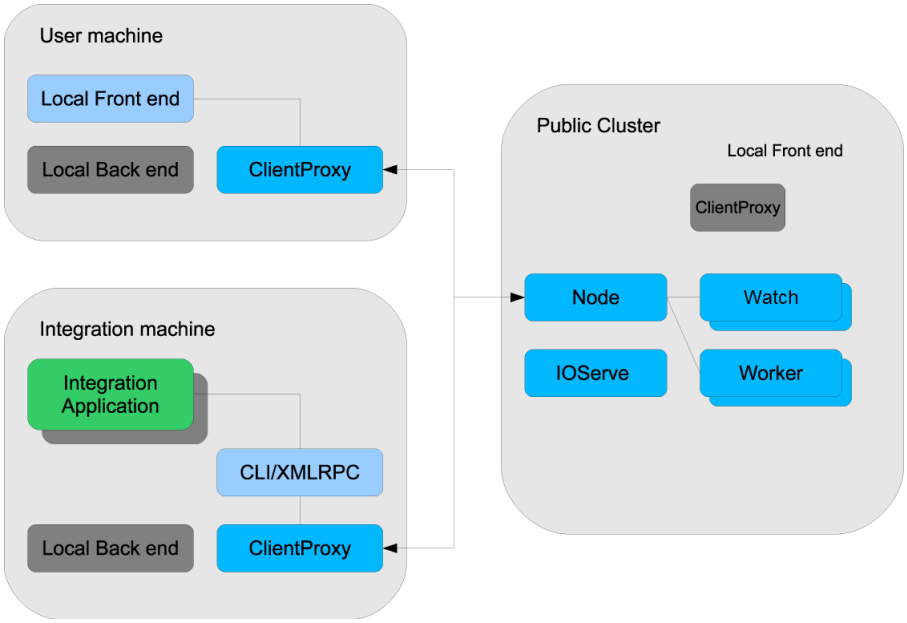
Assistant

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.

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 7\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.

Configuration File Directory by Operating System

Operating System	Configuration File Directory
MacOS	~/Library/Application Support/Episode/
Windows	C:\ProgramData\Telestream\Episode 7\

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.

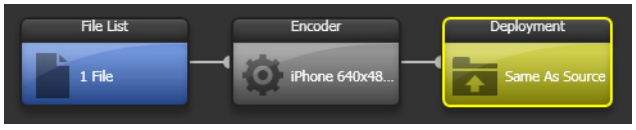
Topics

Workflows

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

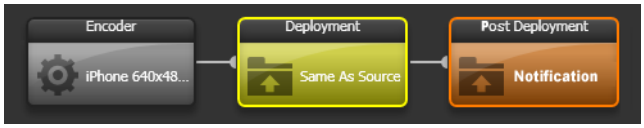
Workflows, as described (and displayed) in the Episode User's Guide, are always comprised of a Source, Encode, and a Deployment task—this is the pattern always used in every workflow.

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.

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. These are the types of Episode tasks:

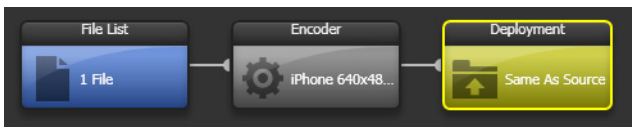
- Encode
- Transfer (Deploy in GUI)
- Execute
- Mail
- MBR
- Move
- Delete
- Localize

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.

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.

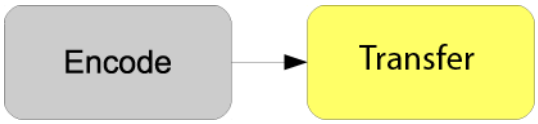
The encoder task has a configuration dependency—the URL of the localized input file. Similarly, the deployment task also has a dependency - the URL of the file created by the encode task. The encode and deployment tasks also have task dependencies - the previous task executing and exiting successfully.

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.

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

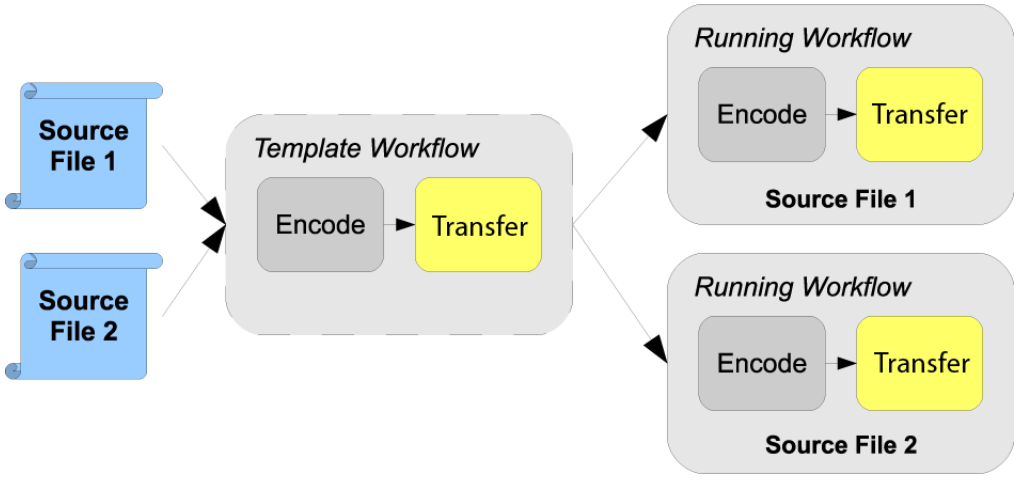
There are four types of Episode Sources:

- 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.

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, etc.

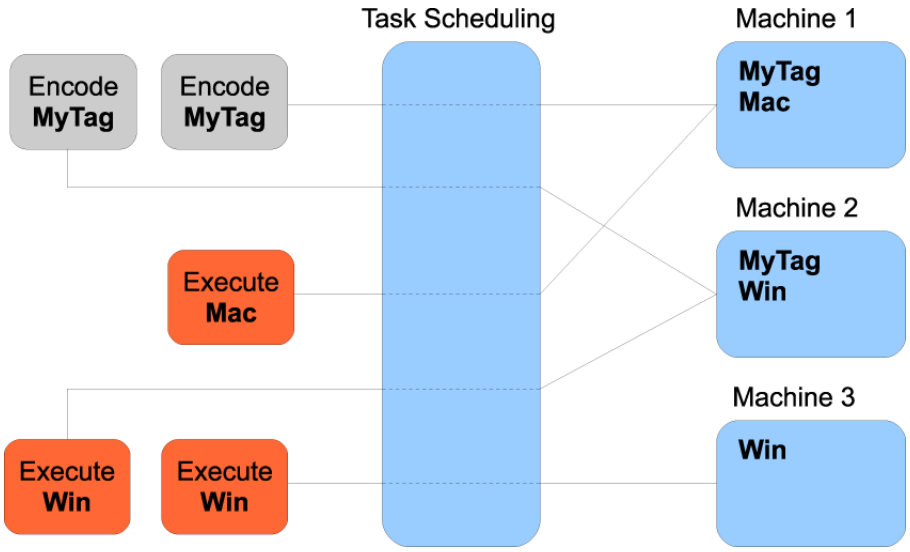
Execute `episodectl variables` for a description of all variables.

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.

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 7\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:

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: C:\ProgramData\Telestream\Episode 7\User Tasks\Encoders\
	Drag Encoder template into drop area	Windows: C:\ProgramData\Telestream\Episode 7\User Tasks\Encoders\
Transfer	New Task > New Deployment	OS X: ~/Library/Application Support/ Episode/User Tasks/Deployments/
	File > New > Deployment	Windows: C:\ProgramData\Telestream\Episode 7\User Tasks\Deployments\
	Drag folder into drop area	Windows: C:\ProgramData\Telestream\Episode 7\User Tasks\Deployments\

Creating Tasks using the Episode CLI

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

Creating Tasks using the Episode XML-RPC Interface

Tasks	XML-RPC Method	Default Save Location
Transfer	`taskCreateTransfer`	File content returned in response
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 | 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.

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: C:\ProgramData\Telestream\Episode 7\User Tasks\Sources\
Watch Folder	Drag folder into source drop area	Windows: C:\ProgramData\Telestream\Episode 7\User Tasks\Sources\

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`	

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

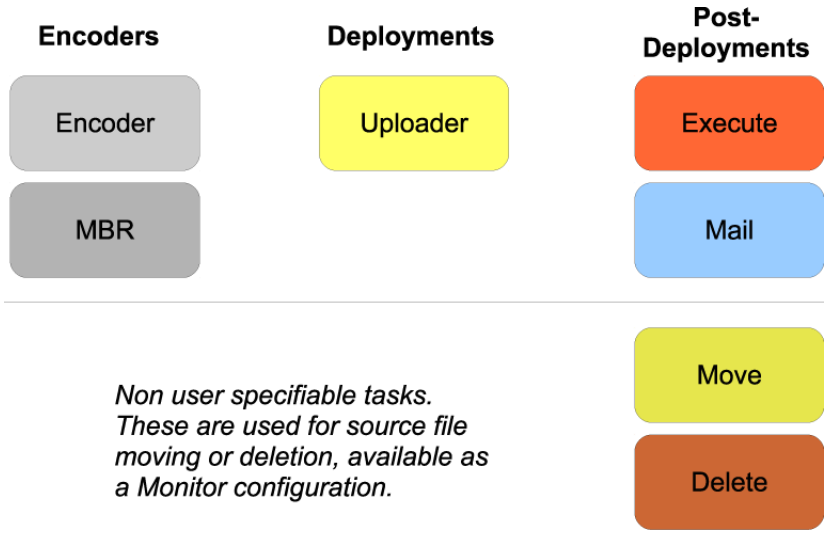
Workflows are created interactively in the Episode GUI program. Using the CLI and XML-RPC interface, they are created on-the-fly – that is, the workflow configuration is part of the submission command. The command in the CLI is `episodectl workflow submit`; in XMLRPC, it is `submitBuildSubmission`.

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.

Encoder, Deployment, and Post-Deployment tasks

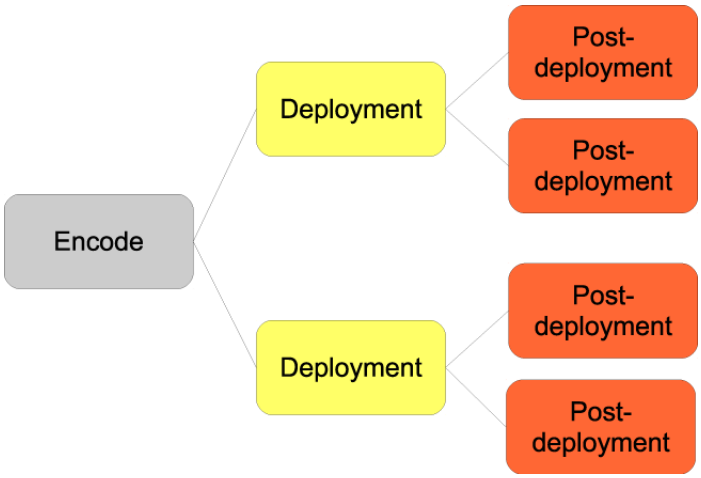


A workflow is built as a tree, branching out from Encoder actions to Deployment to Post-deployment actions. In the Episode GUI program, you specify a Deployment for each Encoder task. However, in CLI and XML-RPC, the default behavior is that you specify a Deployment for all Encoders.

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.

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



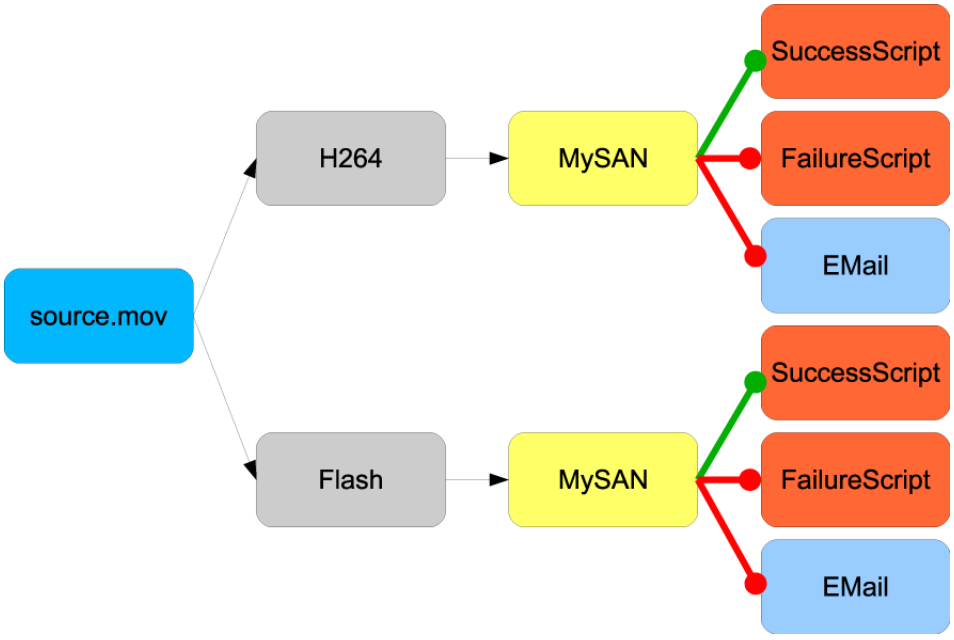
The execution of Post-deployment tasks are always controlled by the success or failure of a Deployment task. A Deployment task is passed a failure status if either the deployment fails or if the preceding Encode task fails. It is passed the success status only if both the preceding Encode task succeeds 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:

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 7\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 the table below.
- **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.

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 7\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 7\bin*
Create mail task	./episodectl task mail

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/

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 7\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 e`pisodectl 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 and is also available in the Episode Windows and Mac user interfaces. To access help for using Named Storage via the CLI, enter the following CLI command:

```
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>\studioshares\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 7):
'/Applications/Episode.app/Contents/Resources/engine/bin/episodectl' launch start

On Windows, enter this path from the root:
"C:\Program Files\Telestream\Episode 7\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 in Windows

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

Go to Start > All Programs > Telestream > Episode 7 > Episode 7.

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 in Windows

Episode Control—the CLI Interpreter program—is installed by default in *C:\Program Files\Telestream\Episode 7\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:

1. Click Start to display the Search Programs and Files text field. Enter `cmd` and press Enter to display the Command window.
2. Navigate to the Episode bin folder, type the following, and press Enter:
   `cd "C:\Program Files\Telestream\Episode 7\bin\"`
   Quotes are necessary because of spaces in the path.
3. To use the CLI, type *episodectl* along with your function and any arguments to execute the Episode command. For details, see Using the CLI Interpreter.

Note: If your Episode services are not running, before proceeding, execute *episodectl launch start* with the proper arguments (see Determining if Episode is Running).

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 in MacOS

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 in MacOS

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

To start Episode Control, follow these steps:

1. Open a Terminal window (Applications > Utilities > Terminal).
2. Navigate to Episode's bin folder so you can execute the Episode Control program:
   /Applications/Episode.app/Contents/Resources/engine/bin/.
3. Type the following command and press Enter:
   `cd /Applications/Episode.app/Contents/Resources/engine/bin`
4. 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 7\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`.

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\Episode7\bin\episodectl launch restart -x

[Windows 64-bit]: C:\Program Files (x86)\Telestream\Episode7\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:

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:

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:

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.

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:

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:

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:

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:

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':

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.

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.

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

Objects and Arrays

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

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. You can add whitespace between any pair of tokens.

Strings

A string consists of a sequence of Unicode characters in double quotes, using backslash escapes. A character is represented as a single character string.

Numbers

A number is very much like a C or Java number, except that the octal and hexadecimal formats are not used.

Binary Data

Binary data is treated as strings. When sending such a value as a parameter with JSON-RPC, it should be prefixed with "#bin#", without the quotation marks. Without the prefix, our JSON-RPC client will treat the value as a regular string.

Here is a small example of how a JSON command looks with a binary (base64) value. The value is truncated for ease.

```
{
"jsonrpc": "2.0",
"method": "submitBuildSubmission",
"id": 1,
"params":
{ "workflow-name":"json workflow", "file-
list":["C:\Users\Username\Videos\Sources\video.mov"],
"tasks":["#bin#PD94bWwgdmVyc2lvbj0iMS4wIiBlbmNvZGluZQo=..."],
"destination-dirs":["C:\Users\Username\Videos\Output\"] }
}
```

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 7\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.

Job Monitoring on the Demo Web Page