
Rapid API Programming Guide

For Rimage Software Development Kit 8.0



R I M A G E TM



Corporate Headquarters:

Rimage Corporation

7725 Washington Avenue South
Minneapolis, MN 55439
USA

800 553 8312 (toll free US)
Service: +1 952 946 0004 (International)
Fax: +1 952 944 6956

European Headquarters:

Rimage Europe GmbH
Albert-Einstein-Str. 26
63128 Dietzenbach Germany

Tel: +49-(0) 6074-8521-0
Fax: +49-(0) 6074-8521-21

CD and DVD Recording Software Disclaimer

This Product, Software, or Documentation may be designed to assist you in reproducing material in which you own the copyright or have obtained permission to copy from the copyright owner. Unless you own the copyright or have permission to copy from the copyright owner, you may be violating copyright law and be subject to payment of damages and other remedies. If you are uncertain about your rights, you should contact your legal advisor. If you are neither in possession of the copyright nor have authorization from the owner of the copyright, unauthorized copying of CDs and DVDs violates national and international legislation and can result in severe penalties.

Rimage Corporation reserves the right to make improvements to the equipment and software described in this document at any time without any prior notice. Rimage Corporation reserves the right to revise this publication and to make changes from time to time in the content hereof without obligation of Rimage Corporation to notify any person or organization of such revisions or changes.

This document may contain links to web sites that were current at the time of publication, but may have moved or become inactive since. This document may contain links to sites on the Internet that are owned and operated by third parties. Rimage Corporation is not responsible for the content of any such third-party site.

©2006, Rimage Corporation

Rimage™ is a trademark of the Rimage Corporation. SDK™ is a trademark of the Rimage Corporation. Dell® is registered trademark of Dell Computer Corporation. FireWire® is a registered trademark of Apple Computer, Inc.

All other trademarks and registered trademarks are the property of their respective owners.

Support information

US, Asia/Pacific, Mexico/Latin America

Rimage Corporation

7725 Washington Avenue South
Minneapolis, MN 55439
USA

Attn: Rimage Services

Telephone:

North America: 800-553-8312

Asia/Pacific, Mexico/ Latin America: 952-946-0004

Fax: 952-946-6956

When you contact Rimage Services, please provide:

- Unit serial number and software version.
- Functional and technical description of the problem.
- Exact error message received.

Europe

Rimage Europe GmbH
Albert-Einstein-Str. 26
63128 Dietzenbach Germany

Telephone: +49-(0) 1805-7462-43

Fax: +49-(0) 6074-8521-100

Rimage Europe Technical Website

<http://www.rimage.com/support.html>

Select the appropriate Support link to learn more. If you cannot find a solution on our website, email Rimage Services, Europe at support@rimage.de.

Learn more online

At www.rimage.com/support.html, you can experience Rimage's world-class Support and Services.

1. Select your product family.



2. Select your product.



3. Learn more on your product page.



From your product page you can access:

- Information about the latest software and firmware updates
- Product specifications
- Troubleshooting tips and FAQs
- The latest documents
- Printing tips
- Current firmware and driver downloads



Table of Contents

Introduction	1
Rapid API features	1
Getting started	2
Working with the API	3
Working with sessions	4
Client Session	4
SingleConnectionSession	4
Connection	4
Working with listeners	5
Listener callback options	5
Working with jobs	6
Creating a job	7
Setting job data	7
Jobs spanning multiple discs	8
Setting job parameters	8
Submitting a job	10
Durable Jobs	10
Monitoring jobs	10
Recovering Durable Jobs	11
Working with parameters	12
Working with servers	13
Working with server alerts	14
Working with server orders	14
Memory management	15
Enumerating objects	16
Exception handling	17
Customizing the API	19
Extending job classes	19
Job factory	21
Generating custom job XML	22
Customizing ImageOrder XML	23
Customizing ProductionOrder XML	24
XML writer factory	25
Generating custom server request XML	27
Samples	29
Deployment	31
C++ deployment	31
Build information	31
Required files and folders	31
.NET deployment	32
Build information	32
Required files and folders	32
Required .NET Assembly files	32
Appendix A – Client ID and Job ID uniqueness rules	33
Appendix B – Reference documentation	35



Introduction

Rimage has developed a new interface for programmers to integrate their software with the Rimage system quickly and easily. This interface is called the Rapid API, referred to in this document as Rapid API or simply the API.

- The Rapid API is included on the Rimage Software Development Kit (SDK) disc.
- The Rapid API is supported on Windows platforms only.
- The Rapid API supports C++ and all .NET languages.
- This document includes information about the Rapid API, which requires Rimage Software Suite version 8.0 or above.

Rapid API features

- The Rapid API eliminates the need for XML creation and parsing - an object oriented interface is presented.
- The Rapid API presents a job level interface encapsulating imaging and production orders – eliminating the need to administer order management details.
- The Rapid API offers the ability to set up either a single or multiple job listeners.
- The Rapid API offers the ability to set up connection listener and server listeners, called session listener.
- The Rapid API can make calls back to the application on the application's main user interface or UI thread or some other UI thread – eliminating problems with updating UI components from a non-UI thread.

 **Note:** .NET 2.0 generates an exception if a UI component is being updated on a non-UI thread.

- The Rapid API offers the ability to extend Job classes to implement functionality not provided by default.
- The Rapid API also offers the ability to customize Job XML to implement functionality not provided by default.



Getting started

 **Note:** Before you install the Rapid API, Producer Software Suite 7.3 or later must be installed on the PC connected to the Rimage autoloader (the Control Center).

1. Install the **Software Development Kit**.
2. Open sample project(s) and run sample program(s) to familiarize yourself with the Rapid API and the programming environment. Sample programs are located in C:\Program Files\RimageSDK\ApiSdk\Samples\RapidApi by default. Select the appropriate project type and open the project in Visual Studio 2005. Refer to [Samples](#) on page 29 for more information
3. Create a **program** to publish your first disc by implementing the following:

 **Note:** These steps are shown in the “Hello World”



Working with sessions

To begin working with the Rapid API, initiate a client session with the Rimage system. A session object is created and the application maintains a reference to only one session object. Through the session object, the application can manage jobs (create, submit, retrieve, etc.), connections, servers and server alerts, and set listeners.

Client Session

The **ClientSession** class is the base class for all sessions. This class is never instantiated directly. It exposes methods that are common to all types of sessions, such as job creation, enumerating over jobs, servers, and alerts, and setting job and session listeners.

Session type specific methods, such as submitting a job for processing, are exposed by **ClientSession** subclasses. A subclass' `GetInstance()` static method is called to get an instance of a session - this is the starting point of working with the Rapid API.

 **Note:** Currently Rapid API offers only one **ClientSession** subclass – **SingleConnectionSession**.

SingleConnectionSession

SingleConnectionSession is currently the only subclass of **ClientSession**. The Rapid API supports a connection to only one messaging server at any one time. The client application is free to connect and disconnect from any number of messaging servers on the network, but it can be connected to only one at a time.


To get an instance of a single connection session, use the static method `SingleConnectionSession.GetInstance()`. Rimage recommends the client application always accesses **ClientSession** and **SingleConnectionSession** through `SingleConnectionSession.GetInstance()` method. For example:

- **C++:** `SingleConnectionSession::GetInstance()->Connect()`
- **C#:** `SingleConnectionSession.GetInstance().Connect()`

 **Note:** `SingleConnectionSession` class inherits all the **ClientSession** methods which eliminates the need to hold onto a reference to **ClientSession** class.

Connection

Connection class encapsulates information about a connection to the messaging server and allows operations that are specific to this connection such as enumerating servers connected to the same messaging server or submitting a job to those servers. The **Connection** class also offers information methods about the connection itself, such as messaging host and port currently connected to, client ID, and Rimage system folder.

 **Note:** A **Connection** object is automatically created when `ConnectionSession.Connect()` method is called.



Working with listeners

The client application can asynchronously receive two types of events, job status events and session events. Job status events notify the application about the progress of a specific job. Session events include server alerts, connection drops, and server state changes.

1. The client application implements the listener interfaces.

```
class SampleUserJobListener : IJobStatusListener { }  
class SampleUserSessionListener : ISessionStatusListener { }
```


2. And gives the listener object references to the Rapid API.


```
SingleConnectionSession.SetDefaultJobListener()  
SingleConnectionSession.SetSessionStatusListener()
```

3. Statuses are asynchronously delivered to the application through the listener objects when they arrive.

4. An application can remove the listeners at any time.

```
SingleConnectionSession.RemoveDefaultJobListener()  
SingleConnectionSession.RemoveSessionStatusListener()
```

 **Note:** An application can also set up job status listeners on individual job objects. If the default listener and a specific job listener are set, then all statuses for that job will be received twice by the application – once by the default listener and once by the job specific listener.

 **Important!** Client application methods that implement the **ISessionStatusListener** and **IJobStatusListener** interfaces should not throw exceptions back to the API. If an exception is thrown, it is logged and the API will continue to process without notifying the user.

Listener callback options

Listener objects are called by the Rapid API on a thread. The client application has control over which thread is used when its listener objects are called by the API. The application specifies its intention when it sets up the listener. A listener can be called back on one of three different threads:

- Application's main UI thread – this is the most common option for UI applications. This eliminates a potential issue with updating UI components from a non-UI thread. Also, most UI applications have only one UI thread – the main application thread. To be called back on the main UI thread, setup the listener like this:

```
SingleConnectionSession.SetDefaultJobListener(IJobStatusListener,  
CallbackOnMainUIThread.True)
```

- An application's UI worker thread – in some cases an application will create a UI worker thread. The Rapid API needs the UI worker thread's ID and calls the listener back on this thread. To be called back on a UI worker thread, setup the listener like this:

```
SingleConnectionSession.SetDefaultJobListener(IJobStatusListener,  
CallbackUIThreadId)
```

- Rapid API thread – in cases when there is no UI thread, or the application will not update a UI component with status information, it may be more advantageous to receive listener events on a worker thread. This is recommended for console applications. To be called back on the Rapid API's thread, setup the listener like this:

```
SingleConnectionSession.SetDefaultJobListener(IJobStatusListener,  
CallbackOnMainUIThread.False)
```



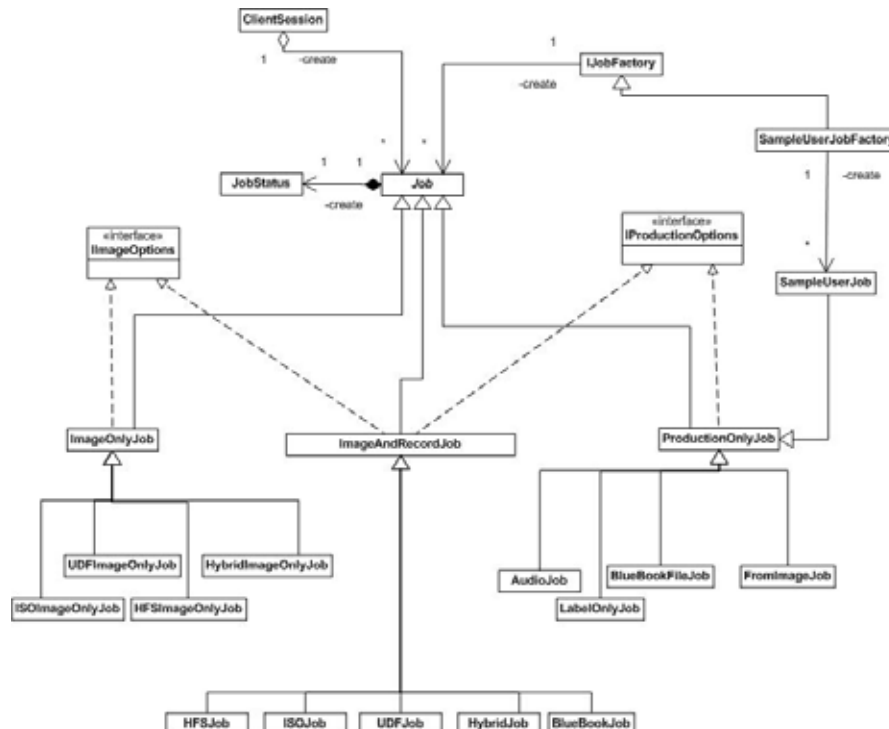
Working with jobs

The main purpose of the Rimage publishing system is to record discs with data and print labels on those discs. This process is facilitated by the Imaging and Production servers. Sometimes other applications are required, such as imaging data to an image file to be recorded on a disc at a later time, or printing a label on a previously recorded disc. Rapid API accommodates all of these scenarios by providing three basic types of Job classes.

- **ImageAndRecordJob** is the most common type of job that images and records data and optionally prints a label. This type of job utilizes both Imaging and Production servers.
- **ImageOnlyJob** is the job type that creates an image file to be recorded on a disc at a later time. This type of job utilizes only the Imaging server.
- **ProductionOnlyJob** this type of job exists for recording an existing image file on disc, printing a label on an existing disc, or to make an audio CD which requires no prior imaging. This type of job utilizes only the Production server.

The Job class hierarchy can be divided into three levels:

1. First level consists of the **Job** base class from which all job classes inherit. This class exposes methods common to all types of jobs, such as job ID and common parameters.
2. Second level consists of **ImageAndRecordJob**, **ImageOnlyJob**, **ProductionOnlyJob** – these classes encapsulate the types of Rimage jobs discussed above. This level of classes expose all methods necessary for a job type. These classes implement the **IImageOptions** interface, the **IProductionOptions** interface, or both interfaces.
3. Third level consists of specific job type classes which represent the most common types of Rimage jobs. These job classes have default parameter values to match their specific types. Refer to the [job parameter defaults](#) table on page 8 for more information. For example, the default media type for a **UDFJob** is DVDR, but the default media type for an **ISOJob** is CDR.





The above diagram also shows two key interfaces: **ImageOptions** and **IProductionOptions**. These interfaces specify the contract for jobs that are processed by Imaging server, Production server, or both.

ImageOptions interface exposes methods for specifying data to be imaged as well as settings which control how that data is to be imaged, for example, ISO Level 2 or UDF Version 1.02.

IProductionOptions interface contains methods that allow specification of information such as the label file and the list of audio or data tracks, to record on the disc. It also allows specification of production settings such as media type and number of copies.

Second level classes that implement the **ImageOptions** interface are **ImageAndRecordJob** and **ImageOnlyJob** because both of these job types involve the imaging of data.

Second level classes that implement the **IProductionOptions** interface are **ImageAndRecordJob** and **ProductionOnlyJob** because both of these job types use Production server to record a disc.

Creating a job

Job objects can be created from job classes of the second and third levels. The first level Job base class is declared abstract.

To create a specific type of job, use one of the following methods:

```
ImageAndRecordJob ClientSession.CreateImageAndRecordJob(JobType)
ImageOnlyJob ClientSession.CreateImageOnlyJob(JobType)
ProductionOnlyJob ClientSession.CreateProductionOnlyJob(JobType)
```

The JobType parameter tells the API which job to create. If the JobType parameter is specified as a third level type, such as UDFJobType, then the caller is required to cast the returned reference to the correct type. For example:

C++

```
UDFJob* job = (UDFJob*)SingleConnectionSession::GetInstance()->
CreateImageAndRecordJob(UDFJobType)
```

C#

```
UDFJob job = (UDFJob)SingleConnectionSession.GetInstance().
CreateImageAndRecordJob(JobType.UDFJobType)
```

Setting job data

After a job is created, setup the data and change any related settings. Communicate to the Rimage system which data to record on the disc. For **ImageAndRecordJob** and **ImageOnlyJob** this is accomplished with the AddParentFolder() method. An Editlist is specified by calling the AddEditList() method. Multiple parent folders and Editlists can be given to a job.

For a **ProductionOnlyJob** an image file must be specified with the AddDataTrack() method. Multiple data tracks can be recorded on a disc. For an audio job, AddAudioTrack() can be called multiple times.

The SetLabelFile() method is available on **ImageAndRecordJob** and **ProductionOnlyJob** to communicate to the API which label to print on the disc.

When a file is given to the API, (e.g. AddParentFolder()) a second parameter of type ConvertToUNC enum is passed in. This parameter tells the API whether or not to convert the file path to a UNC path so it is accessible from the network.



For example:

If C:\Rimage is the parent folder path, this folder is shared as Rimage, and the second parameter is **ConvertToUNC.True**, then the path \\<computername>\Rimage will be given to the server. This is useful when the Rimage servers are running on other computers on the network.

Some data, such as a parent folder or label file, have parameters that are specific to the data. Set and Get methods exist which take the *original* path given to the API as the first parameter and a key/value pair for the rest. For example:

```
void SetLabelFileParam(String labelFile, LabelFileParamType key, String value)
String GetLabelFileParam(String labelFile, LabelFileParamType key)
```

Refer to [Working with parameters](#) on page 12.

Jobs spanning multiple discs

If the data specified for an ImageAndRecordJob is too large for the specified media - more than 750MB for CDR, or more than 4.3GB for DVDR, etc. - Rimage Imaging and Production servers can produce multiple discs for that job, spanning the data across the discs.

To allow the Rimage system to span discs for a job, set the AllowSpanning property to true. By default this property is false.

C++

```
void ImageAndRecordJob::AllowSpanning(bool value)
bool ImageAndRecordJob::IsSpanning()
```

C#

```
ImageAndRecordJob.AllowSpanning { bool get(); void set(bool) }
```

Setting job parameters

The remaining job settings are divided into three categories:

- **Common job settings** are managed with SetJobParam() and GetJobParam(). These settings encompass parameters that are common to all job types.
- **Imaging job settings** are managed with SetImageParam() and GetImageParam(). These methods are specified by the **IImageOptions** interface and are implemented by the **ImageAndRecordJob** and **ImageOnlyJob** classes.
- **Production job settings** are managed with SetProductionParam() and GetProductionParam(). These methods are specified by the **IProductionOptions** interface and are implemented by the **ImageAndRecordJob** and **ProductionOnlyJob** classes.

Refer to [Working with parameters](#) on page 12.

Job parameter defaults

When a third level job class is created, parameters are set to the most reasonable default values for the created job type. The following table lists the default settings for all the third level jobs.

ImageAndRecordJob	Defaults
ISOJob	Image format = ISO Level 2, Media type = CDR
UDFJob	Image format = UDF Version 1.02, Media type = DVDR
HybridJob	Image format = ISO Level 2, HFS, Media type = CDR
BlueBookJob	Image format = ISO Level 2, Media type = CDR
HFSJob	Image format = HFS, Media type = CDR



ImageOnlyJob	Defaults
ISOImageOnlyJob	Image format = ISO Level 2
UDFImageOnlyJob	Image format = UDF Version 1.02
HybridImageOnlyJob	Image format = ISO Level 2, HFS
HFSImageOnlyJob	Image format = HFS
ProductionOnlyJob	Defaults
AudioJob	Media type = CDR
LabelOnlyJob	Media type = CDR
BlueBookFromImageJob	Media type = CDR
FromImageJob	Media type = CDR



Submitting a job

To submit a job for processing, the application simply calls

```
void SingleConnectionSession.SubmitJob(Job)
```

The Rapid API validates parameters set on the job for correctness before it sends the job out to the servers for processing. Validation includes:

1. Checking for incorrect parameter values.
2. Checking for incompatible parameter combinations. For example ISO and UDF format settings cannot coexist in a single job – it's either a UDF job or an ISO job.

Durable Jobs

To submit a job so its status can be recovered after a crash use

```
C++: SingleConnectionSession::SubmitDurableJob(Job)
```

```
.NET: SingleConnectionSession.SubmitDurableJob(Job)
```

This ensures that the client application using the Rapid API will never lose job statuses, even in case of a crash or network disconnect.

Refer to section *Recovering Durable Jobs* for more information.

Monitoring jobs

Each **Job** object holds onto a **JobStatus** object which represents the current status of the job at all times, even before a job is submitted. To access the **JobStatus** object call

C++

```
JobStatus Job.GetStatus()
```

To access the current state of the job call

C++

```
JobStateType JobStatus.GetState()
```

.NET

```
Job.Status
```

```
JobStatus.State
```

A job is in one of the following states at any one time

```
JobNotStarted
```

```
JobSubmittedForImaging
```

```
JobSubmittedForProduction
```

```
JobInProgress
```

```
JobCompleted
```


```
JobCancelled
```

```
JobFailed
```

JobStatus contains other information such as percent completed, error code and error message in case of failure, IDs of the servers processing the job, etc. The application is free to access the **JobStatus** object for a job at any time.



Another way to monitor job progress is to set up job listeners. If a listener is set up, a default listener or a job specific listener, the application will be notified asynchronously with a callback to the `IJobStatusListener.OnStatus()` method every time the status changes. The job ID and a reference to the **JobStatus** object for that job are passed into the callback method.

 **Important:** The application should not hold onto a Job Status reference passed into the callback. It should retrieve the JobStatus reference from the Job object every time.

Notes:

It is much more efficient to be notified through a listener versus polling for job status proactively.
Finished jobs can be removed from the Rapid API's cache by calling `ClientSession.RemoveFinishedJobs()`.

Recovering Durable Jobs

In case of an application crash or shutdown, or a network problem during job processing the flow of current job statuses will be interrupted. To recover missed job statuses the application needs to call

C++:

```
SingleConnectionSession.RecoverDurableJobs()
```

NET:

```
SingleConnectionSession.RecoverDurableJobs()
```

at next startup or reconnect to the Rimage system.

This will send stored job statuses to the client application. Rapid API will internally create all the recovered Job and JobStatus objects. If the client application sets up a default JobStatusListener before calling `RecoverDurableJobs()` then recovered job statuses will be sent to the listener.




Working with parameters

Throughout the Rapid API one encounters methods of this pattern

```
void SetSomethingParam(EnumType key,String value)
String GetSomethingParam(EnumType key)
```

The API groups similar parameters for **Job** or **Server** classes, into categories. Each category is represented by setter and getter methods as well as by an enumerated type. The enumerated type encapsulates the possible parameters for the category. For example the **ImageOptions** interface has the following methods using the **ImagingTargetParamType** enumerated type.

```
void SetImageParam(ImagingTargetParamType key, String param)
String GetImageParam(ImagingTargetParamType key)
enum ImagingTargetParamType {
    TargetImagingClusterParam
    TargetImagingServerParam
    LastImagingTargetParamType };
```

 **Note:** Every enumerated type ends with a delimiter value in the pattern: “Last<enum name>ParamType” – LastImagingTargetParamType in the above example. This enumerated value is for internal use only.

The value of the parameter, whether set or retrieved, is always represented by a string type (LPCTSTR in C++ and System.String in .NET). Some parameter values are just text, such as the **ImagingTargetParamType.TargetImagingServerParam** which requires the ID of a server.

Other parameters only accept values in a specific set, such as **FormatParamType.ISOLevelParam** which only accepts none, 1, or 2. Constants exist for all predefined values. For this example they are:

C++

```
FORMAT_ISO_NONE
FORMAT_ISO_9660_L1
FORMAT_ISO_9660_L2
```

.NET

```
JobParamValues.FormatISONone
JobParamValues.FormatISO_9660_L1
JobParamValues.FormatISO_9660_L2
```

Other parameters only accept true or false, such as **FormatOptionParamType.ForceUpperCaseParam**. Predefined constants exist for these values as well:

C++

```
RAPID_TRUE
RAPID_FALSE
```

.NET

```
ParamValues.True
ParamValues.False
```



Rapid API validates parameter values as soon as possible. If a parameter value for a specific parameter doesn't match the allowed set of values, an exception is thrown to the application. This pattern of setting and getting parameters allows the Rapid API to keep the number of class methods small.

Note: For a detailed explanation of all enumerated types and appropriate values, access the HTML help files located, by default, in C:\Program Files\RimageSDK\ApiSdk\doc.

Working with servers

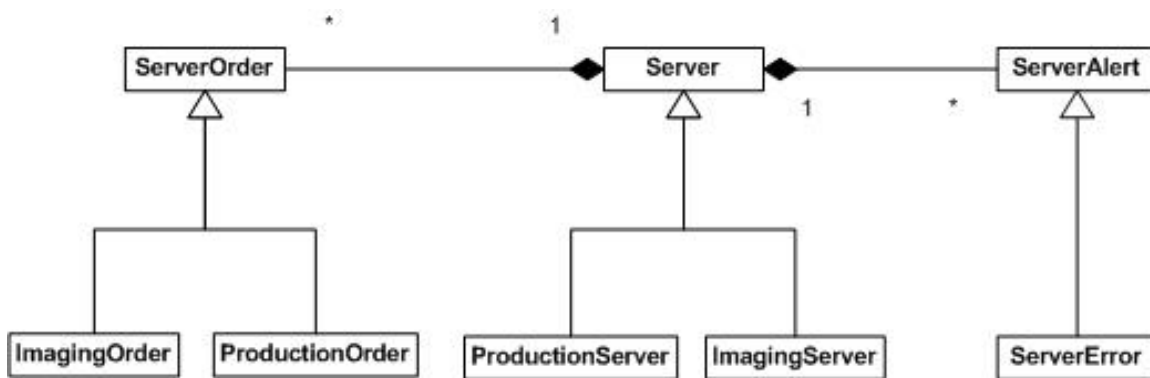
When the client application connects to a messaging server, the Rapid API starts collecting information about Rimage servers on behalf of the application. Server objects are never instantiated by the client application, instead they are created by the API when a specific server is detected online. The server objects represent either the Imaging or the Production servers available on the network. The application is notified of server state change events if a session status listener is set up.

The **Server** classes expose server related interfaces available to the client application. The application can pause and resume a server, retrieve server information, change server settings, and much more.

Server objects can be enumerated by initializing an iterator with either **ClientSession** or **Connection** objects. Refer to [Enumerating objects](#) on page 16.

Server objects also keep track of alerts currently active on a Rimage server. Alerts represent warning and error conditions, such as out of discs, recording failure, etc. Refer to [Working with Server Alerts](#) on page 14.

An application can also query a server for a list of orders currently in process on that server. Refer to [Working with server orders](#) on page 14.



Server is a base class for two concrete server types – **ImagingServer** and **ProductionServer**. The subclasses inherit all the **Server** methods as well as offering specific methods for each server type.

ProductionServer class extends the **Server** class and adds functionality specific to the Production server. It allows the caller to get information about disc input and output bins, recorders, printers, and to change Production specific settings.

ImagingServer class extends the **Server** class and adds functionality specific to the Imaging Server, such as retrieving Imaging server specific information.

Refer to [Working with parameters](#) on page 12.

Note: Offline servers can be removed from the Rapid API's cache by calling **ClientSession.RemoveOfflineServers()** or **Connection.RemoveOfflineServers()**.



Working with server alerts

A server alert represents a warning or an error condition encountered by a server. Most of the alerts require a user action, such as putting more discs in the input bin of an autoloader. Some alerts can be answered programmatically.

A `ServerAlert` object is attached to a **Server** object by the API when it is generated by a Rimage server. The application is notified of a new or acknowledged alert if a session status listener is set up.

`ServerAlert` objects can be enumerated by initializing an iterator with either a **ClientSession** or a **Server** object. Refer to [Enumerating objects](#) on page 16.

The typical application handles alerts in the following way:

1. A server sends out an alert.
2. The client application receives notification of the alert in `SessionStatusListener.OnServerAlert()` callback method.
3. The client application displays the alert in a dialog. The dialog shows buttons with corresponding strings received from `ServerAlert.GetNumReplies()` and `ServerAlert.GetReplyText()` methods.
4. An operator resolves the alert condition (places more blank discs in the input bin for example).
5. The operator clicks on one of the buttons which triggers a call to reply to one of the server's alerts. There are three methods to choose from to reply to a `ServerAlert`:
 - `ReplyWithActionId()` is the preferred method. This allows the caller to pass in a unique action ID for that specific reply. For a finite set of action IDs in the Rimage system, refer to `Server Alert Action IDs` located at `C:\Program Files\RimageSdk\Manuals\Programming Guides\Server Alert Action IDs Reference.pdf`. An Action ID can be retrieved from an alert by calling `GetReplyActionId(int index)`.
 - `ReplyWithText()` method allows the caller to pass in the same string as the one displayed on the button. The text string can be retrieved from an alert by calling `GetReplyText(int index)`.
 - `ReplyWithIndex()` method allows the caller to pass in the index of the reply, starting with index 0.
6. The server receives the reply, checks that the condition has been fixed, and sends out an alert acknowledgement.
7. Rapid API updates the status of the `ServerAlert` object.
8. The client application receives the alert acknowledgement through the `SessionStatusListener.OnServerAlert()` method with `AlertAcknowledged` value.

 **Note:** Acknowledged alerts can be removed from the Rapid API's cache by calling `Server.RemoveAcknowledgedAlerts()` or `ClientSession.RemoveAcknowledgedServerAlerts()`.

Working with server orders

Server classes allow you to work with any orders that are currently active on a certain server. These orders are submitted by a client application on the network, this may include orders submitted by client applications other than your own. Orders are represented by **ServerOrder** base class and **ImagingOrder** and **ProductionOrder** subclasses.

Server orders are attached to a **Server** object by the API after an application makes a call to `Server.RefreshInfo(ServerInfoRefreshType.RefreshActiveOrders)` method. Information that a client application retrieves from server orders is similar to the information obtained from a **JobStatus** object, such as ID, percent completed, copies completed, etc. **ServerOrder** objects can be enumerated by initializing an iterator with a **Server** object. You can also retrieve a list of orders that are pending to be processed. To do this call

C++:

```
Connection::InitPendingOrderIterator()
```

NET:

```
Connection.GetPendingOrders()
```

This call allows you to retrieve pending orders on a combination of specific connection and cluster. The rest of the parameters let you filter the list even further. You can specify a combination of the following filters

- **clientId** - limit the list to orders authored by a specific client application
- **serverId** - limit the list to orders targeted to a specific server
- **maxOrders** - limit the number of orders in the returned list
- **targetedOrdersOnly** - specifies if the returned orders are only ones targeted to a specific server or not targeted at all.

You can also call

C++:

```
Server::InitPendingOrderIterator()
```

.NET:

```
Server.GetPendingOrders()
```

to limit the list to orders targeted to the server object you're using. The parameters for this method have the same meanings as the parameters above.

Memory management

The caller never uses the new operator to create instances of any of the Rapid API's classes. To create a **Job** object call **ClientSession.CreateJob()**. A **JobStatus** object is created by the **Job** class, **Server** and **ServerAlert** objects are instantiated by the API, etc.

The application almost never calls the delete operator on the API objects because `Remove*()` methods exist on various classes. Examples include `ClientSession.RemoveAcknowledgedServerAlerts()` and `ClientSession.RemoveFinishedJobs()`. It is also possible to delete a specific finished job by calling `ClientSession.RemoveJob()`.

When an application disconnects from the Messaging Server, the **Server** and **ServerAlert** objects are removed. **Job** objects are not removed. The reason **Job** objects are not removed is that the same **Job** object can be processed on one messaging connection and then submitted for processing on a new connection if the application disconnects from the first connection and then connects to a different Messaging Server.



Enumerating objects

Rapid API instantiates objects on the caller's behalf and maintains lists of those objects internally. The API offers a way for the client application to retrieve references to those objects, such as a **Job** or **Server** object.


Methods exist on various classes to retrieve an object by its ID. Examples include `Connection.GetServer()` and


`Server.GetAlert()`. If the caller knows the ID of a certain object, such as an Alert ID or a Server ID, this is the fastest way to retrieve an object.

If however an application needs to list all objects of a certain type, such as **Jobs** or **ServerAlerts** for a specific **Server**, then it can use various classes to initialize an iterator and enumerate over the whole list.

 **Note:** The term **Enumerator** is used to match .NET terminology.

In C++, the caller initializes an **Iterator** of a specific type by using one of the `Init*Iterator()` methods on the classes that offer iterator functionality, such as **ClientSession**. Once an **Iterator** object has been initialized, the application can list all of the objects by calling `Iterator.Next()` in a loop. In C#, or any other .NET language, the call to a `Get*s()` method is made, such as `ClientSession.GetServers()`. This method returns an object which implements the `IEnumerable<T>` interface. This enables the use of the C# 'foreach' construct ('For Each' in Visual Basic). The enumerator can be obtained as well by calling `IEnumerable<T>.GetEnumerator()` method.

 **Note:** An initialized **Iterator** is like a snapshot in time. For example, if a new **Server** object is created by the API while the caller is enumerating **Server** objects, the new object will not be retrieved by the **Iterator**.

 **Important!** Calling one of the `Remove*()` methods to delete objects on one thread while enumerating the objects being removed on another thread can result in a null pointer or a null reference exception.

Example: Enumerating over Job objects

C++:

```
JobIterator iter;
SingleConnectionSession::GetInstance()->InitJobIterator(iter);
while ( iter.HasMore() )
{
    Job job = iter->Next();
}
```

C#:

```
foreach(Job job in SingleConnectionSession.GetInstance().GetJobEnumerator())
{
    // Work with the job object
    string jobId = job.JobId;
}
```



Exception handling

Most Rapid API methods throw an exception in case of an error, whether it's a connection related error or a parameter validation error. The type of the exception thrown is always **RimageException**. It is recommended that calls to the API methods are always wrapped in a try catch block as shown in this example:

C++

```
try
{
    SingleConnectionSession::GetInstance()->Connect(
        "client1", "localhost", "2506");
}
catch ( RimageException &re )
{
    // Rapid API error occurred
    LPCTSTR error = re.GetErrorMessage();
    int errorCode = re.GetErrorCode();
}
catch ( . . . )
{
    // anything else
}
```

C#


```
try
{
    SingleConnectionSession.GetInstance().Connect(
        "client1", "localhost", "2506");
}
catch ( RimageException re )
{
    // Rapid API error occurred
    string error = re.Message;
    int errorCode = re.ErrorCode;
}
catch ( Exception se )
{
    // anything else
}
```



Customizing the API

The functionality offered by the Rapid API is sufficient for most applications. There are times however when an application needs to do something different than what the API offers out of the box. Several options are available.

1. Customizing a job by extending one of the **Job** classes. For example, setting all job parameters in the constructor of the derived class to avoid polluting the application code.
2. Extending an **XMLWriter** class to generate custom XML. This is useful when the provided **Job** classes don't offer certain functionality, such as specifying a sub index for audio tracks.

 **Note:** The data and settings are converted to one or two XML orders before the job is submitted for processing.

3. Combination of 1 and 2.
4. Extending XML writer class to generate custom XML for a server request.

 **Note:** Almost all server operations are covered by the API out of the box.

Extending job classes

Only the second and third level **Job** classes can be extended by a client application. If an application needs functionality that is significantly different from what is offered by the third level classes, then it may make sense to extend the second level **Job** classes. Refer to [Working with Jobs](#) on page 6 for more information on the **Job** class.

 **Note:** Second level classes perform less validation than the third level classes because the job types are less specific at the second level.

When a **Job** derived class is extended by the client application, almost every method is available for override because most methods in the **Job** hierarchy are declared virtual.

A common reason for extending a **Job** class is to encapsulate the initialization in the subclass' constructor. Any data or parameter setting method can be called from the constructor of the derived class.

 **Note:** For reasons specific to the .NET Rapid API implementation, the derived class' constructor needs to be left blank. Instead, a virtual method Initialize() is available for this purpose.

Example: Extending a Job class

C++

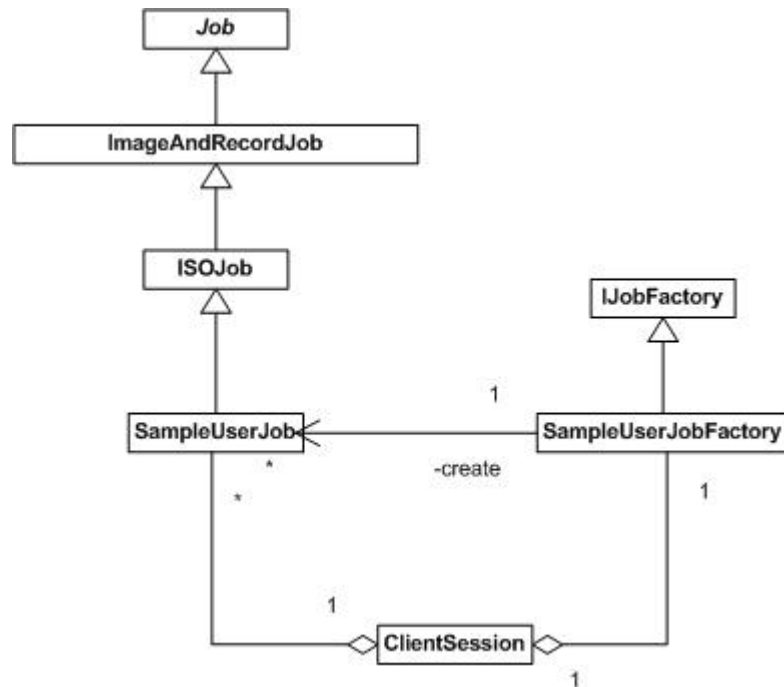
```
class SampleUserJob : public ISOJob
{
public:
    SampleUserJob()
    {
        SetImageParam(IsJolietParam, RAPID_TRUE);
        SetProductionParam(CopiesParam, "5");
        SetProductionParam(MediaTypeParam, MEDIA_TYPE_DVDR_DL);
    }
    virtual ~SampleUserJob() {}
};
```



C#

```
class SampleUserJob : ISOJob
{
    public SampleUserJob() {} // needs to be left blank
    public override void Initialize()
    {
        SetImageParam(FormatParamType.IsJolietParam, ParamValues.True);
        SetProductionParam(ProductionJobParamType.CopiesParam, "5");
        SetProductionParam(ProductionJobParamType.MediaTypeParam,
            JobParamValues.MediaTypeDVDR_DL);
    }
}
```

The UML diagram below illustrates the above example. The diagram also shows another key class involved in job customization, **IJobFactory** interface.



Job factory

Once you write your class you must instantiate it at runtime. To instantiate a custom class, Rapid API requires access to a **Job** object factory.


Rapid API internally uses a default job factory to create existing **Job** objects. To create a user defined **Job** object, the API requires a user defined **Job** factory. The client application implements the **IJobFactory** interface and gives the custom factory to the API.

When a `ClientSession.Create*Job()` method is called by the application, a job type is passed in. If the job type is one of the user types, the custom job factory is called instead of the default to create an appropriate object. Twenty-four custom job types can be created by an application. They are listed below:

- `UserImageAndRecordJobType1` through `UserImageAndRecordJobType8` for **Job** classes extended from the **ImageAndRecordJob** or its descendants.
- `UserImageOnlyJobType1` through `UserImageOnlyJobType8` for **Job** classes extended from the **ImageOnlyJob** or its descendants.
- `UserProductionOnlyJobType1` through `UserProductionOnlyJobType8` for **Job** classes extended from the **ProductionOnlyJob** or its descendants.

IJobFactory interface has two methods that need to be implemented

- `Job CreateJob(JobType)` creates the custom job by using a new operator.
- `bool RemoveJob(Job)` uses the delete operator to remove the custom job.

 **Note:** Unless there are native resources to release or delete, the `RemoveJob()` method doesn't need to do anything.

Example: Implementing a Job factory

C++:

```
class SampleUserJobFactory : public IJobFactory
{
public:
    SampleUserJobFactory() {}
    virtual ~SampleUserJobFactory() {}
    Job* CreateJob(JobType type)
    {
        if ( type == UserImageAndRecordJobType1 )
        {
            return new SampleUserJob();
        }
        return NULL; // not created by custom factory
    }
    bool RemoveJob(Job *job)
    {
        if ( job != NULL && job->GetJobType() == UserImageAndRecordJobType1 )
        {
            delete job;
            return true; // removed
        }
        return false; // not removed by custom factory
    }
};
SingleConnectionSession::GetInstance()->SetUserJobFactory(
new SampleUserJobFactory());
```



C#

```
class SampleUserJobFactory : IJobFactory
{
    public SampleUserJobFactory() {}
    public Job CreateJob(JobType type)
    {
        if ( type == JobType.UserImageAndRecordJobType1 )
        {
            return new SampleUserJob();
        }
        return null; // not created by custom factory
    }
    bool RemoveJob(Job job)
    {
        return true; // garbage collector will clean up the Job
    }
}
SingleConnectionSession.GetInstance().SetUserJobFactory(
new SampleUserJobFactory());
```

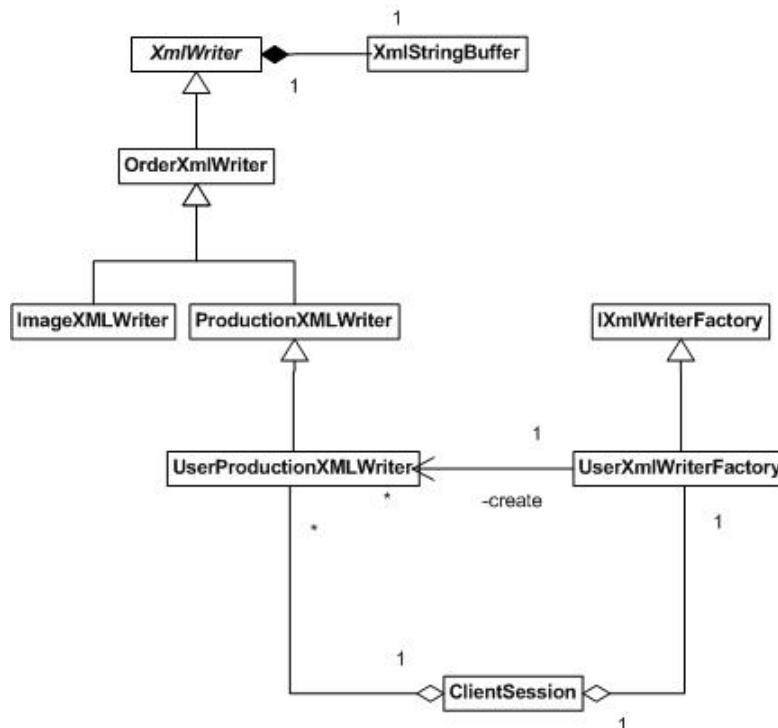
Generating custom job XML

Custom XML generation is available when more functionality is required beyond extending specific Job classes. The Rimage system processes every submitted job according to one or two XML orders – an Imaging order, a Production order, or both. Imaging orders are specified by the ImageOrder DTD and Production orders are specified by the ProductionOrder DTD.

 **Note:** DTDs are located in the Rimage system folder, C:\Rimage\XML by default.

The Rapid API allows you to extend the **ImageXmlWriter** class, the **ProductionXmlWriter** class, or both.

 **Note:** The API also allows custom **Job** classes to coexist with custom XML generation.



This diagram shows the XML writer class hierarchy. The base class is **XmlWriter**. This class exposes methods common to all types of DTDs such as setting and getting DTD version and path information.

It also exposes a virtual method called `FormatXml()` which is the entry point into XML generation. Subclasses of **XmlWriter** implement this method to generate XML specific to their type.

XmlWriter class implements the `FormatXml()` method in the following way:

```
void XmlWriter.FormatXml ()
{
    FormatDocType (GetStringBuffer ());
    FormatRootElement (GetStringBuffer ()); // implemented by a subclass
    FormatDetails (GetStringBuffer ()); // implemented by a subclass
    FormatCloseRootElement (GetStringBuffer ());
}
```

XmlWriter class has a protected member `GetStringBuffer()`. This method allows access to the **XmlStringBuffer** object which makes generating XML simple. **XmlStringBuffer** class offers methods such as `WriteStartElement()` and `WriteAttribute()` which take care of XML formatting details allowing the programmer to concentrate on the business logic. This buffer is passed into every formatting method.

Two **XmlWriter** subclasses - **ImageXmlWriter** and **ProductionXmlWriter** – are used by the Rapid API to generate job related XML. To customize XML generation for jobs, extend **ImageXmlWriter** and **ProductionXmlWriter**. The **ImageXmlWriter** class generates an ImageOrder XML string and the **ProductionXmlWriter** generates a ProductionOrder XML string.

Customizing ImageOrder XML

`ImageXmlWriter.FormatDetails()` writes elements and attributes which govern how the image is created. The body of the method looks like this:

```
void ImageXmlWriter.FormatDetails (XmlStringBuffer xmlStrBuffer)
{
    AddTarget (xmlStrBuffer);
    AddFormat (xmlStrBuffer);
    AddSource (xmlStrBuffer);
    AddOutput (xmlStrBuffer);
    AddRules (xmlStrBuffer);
    AddVolumeName (xmlStrBuffer);
    AddPVDInfo (xmlStrBuffer);
    AddControls (xmlStrBuffer);
    AddCustomize (xmlStrBuffer);
}
```

The `ImageXmlWriter.FormatDetails()` method is declared virtual which allows it to be completely overwritten. In most cases only a certain element needs to be changed. In those cases it is sufficient to overwrite only one of the methods that is called from the `FormatDetails()` method, such as `AddFormat()`. For example:

C++

```
class SampleUserImageXmlWriter : public ImageXmlWriter
{
public:
    SampleUserImageXmlWriter () {}
    virtual ~SampleUserImageXmlWriter () {}
    void AddFormat (XmlStringBuffer &xmlStrBuffer)
    {
        // custom code here
    }
};
```



C#

```
class SampleUserImageXmlWriter : ImageXmlWriter
{
    public SampleUserImageXmlWriter() {}
    protected override void AddFormat(XmlStringBuffer xmlStrBuffer)
    {
        // custom code here
    }
};
```

 **Note:** Refer to the ImageOrder DTD located at C:\Program Files\RimageSDK\ApiSdk\XML for the ImageOrder format details

Customizing ProductionOrder XML

ProductionXmlWriter.FormatDetails() writes elements and attributes which govern how the disc is recorded and what label to print. This is the same pattern used by **ImageXmlWriter**. The body of the method looks like this:

```
void ProductionXmlWriter.FormatDetails(XmlStringBuffer xmlStrBuffer)
{
    AddMedia(xmlStrBuffer);
    AddTarget(xmlStrBuffer);
    AddInOut(xmlStrBuffer);
    AddCustomize(xmlStrBuffer);
    AddActions(xmlStrBuffer);
}
```


The ProductionXmlWriter.FormatDetails() method is declared virtual which allows it to be completely overwritten. In most cases only a certain element needs to be changed. In those cases it is sufficient to overwrite one of the methods that is called from the FormatDetails() method, such as AddCustomize(). For example:

C++

```
class SampleUserProductionXmlWriter : public ImageXmlWriter
{
public:
    SampleUserProductionXmlWriter() {}
    virtual ~ SampleUserProductionXmlWriter() {}
    void AddCustomize(XmlStringBuffer &xmlStrBuffer)
    {
        // custom code here
    }
};
```

C#

```
class SampleUserProductionXmlWriter : ImageXmlWriter
{
    public SampleUserProductionXmlWriter() {}
    protected void AddCustomize(XmlStringBuffer xmlStrBuffer)
    {
        // custom code here
    }
};
```

 **Note:** Refer to the ProductionOrder DTD located at C:\Program Files\RimageSDK\ApiSdk\XML for the ProductionOrder format details.

XML writer factory

Once you write your extended XML writer class, you need to instantiate it at runtime. To instantiate a custom class, Rapid API requires access to an **XmlWriter** factory.

Rapid API internally uses a default XML writer factory to create existing **XmlWriter** objects. To create a user extended **XmlWriter** object, the API requires a user defined **XmlWriter** factory. The client application implements the **IXmlWriterFactory** interface and passes the custom factory to the API.

IXmlWriterFactory interface requires implementation of three methods:

- **CreateImageXmlWriter(JobType)** uses the new operator to create the custom **ImageXmlWriter** object. To specify the type of XML, a **JobType** is passed in to the **XmlWriter** factory.
- **CreateProductionXmlWriter(JobType)** uses the new operator to create the custom **ProductionXmlWriter** object. To specify the type of XML, a **JobType** is passed in to the **XmlWriter** factory.
- **RemoveWriter(XmlWriter)** uses the delete operator to remove the custom **XmlWriter** object.

 **Note:** Unless there are native resources to release or delete, the **RemoveWriter()** method doesn't need to do anything.

Example: Implementing an XML writer factory

C++

```
class SampleUserXmlWriterFactory : public IXmlWriterFactory
{
public:
    SampleUserXmlWriterFactory() {}
    virtual ~SampleUserXmlWriterFactory() {}
    ImageXmlWriter* CreateImageXmlWriter(JobType type)
    {
        if ( type == ISOJobType )
        {
            return new SampleUserImageXmlWriter();
        }
        return NULL; // not created by custom factory
    }
    ProductionXmlWriter* CreateProductionXmlWriter(JobType type)
    {
        if ( type == UDFJobType )
        {
            return new SampleUserProductionXmlWriter();
        }
        return NULL; // not created by custom factory
    }
    bool RemoveWriter(XmlWriter *writer)
    {
        if ( writer != NULL )
        {
            delete writer;
            return true; // removed
        }
        return false; // not removed by custom factory
    }
};
SingleConnectionSession::GetInstance()->SetUserXmlWriterFactory(
new SampleUserXmlWriterFactory());
```




C#

```
class SampleUserXmlWriterFactory : IXmlWriterFactory
{
    public SampleUserXmlWriterFactory() {}
    public ImageXmlWriter CreateImageXmlWriter(JobType type)
    {
        if ( type == JobType.ISOJobType )
        {
            return new SampleUserImageXmlWriter();
        }
        return null; // not created by custom factory
    }
    public ProductionXmlWriter CreateProductionXmlWriter(JobType type)
    {
        if ( type == UDFJobType )
        {
            return new SampleUserProductionXmlWriter();
        }
        return null; // not created by custom factory
    }
    public void RemoveWriter(XmlWriter writer)
    };
SingleConnectionSession.GetInstance().SetUserXmlWriterFactory(
new SampleUserXmlWriterFactory());
```

 **Note:** For more information on customizing job XML refer to the Rapid Custom sample project located at C:\Program Files\RimageSdk\ApiSdk\Samples\RapidApi

Generating custom server request XML

Server classes provide most of the functionality offered by the Rimage Imaging and Production servers. In cases where functionality is required that isn't provided or a newer server request DTD exists, custom server request XML can be generated. This is accomplished by extending the **XmlWriter** class and completely taking over XML generation. The extended **XmlWriter** class still has access to the **XmlStringBuffer** and all detailed formatting methods required to generate a valid XML string.

 **Note:** The two DTDs involved in server requests are `ProductionServerRequest` and `ImageServerRequest` located at `C:\Program Files\RimageSDK\ApiSdk\XML`.

Once the XML is generated, call `Server.ExecuteRequest(String)` to send it to the server for processing. When the server completes the request, it replies with XML formatted to conform to either the `ProductionServerReply` or the `ImageServerReply` DTD. Call the `Server.GetXml()` method to access this XML. The following example illustrates the flow:

C++

```
// generate xml
LPCTSTR requestXml = GenerateProductionRequest(); // user defined method
// get a reference to a server
Server *server =
SingleConnectionSession::GetInstance()->GetServer("HOST1_PS01");
// execute the request
try
{
    // this call can time out
    server->ExecuteRequest(requestXml);
}
catch ( RimageException &e)
{
    // handle the exception
}
// get the reply
LPCTSTR replyXml = server->GetXml();
// parse the reply xml
```

C#

```
// generate xml
string requestXml = GenerateProductionRequest(); // user defined method
// get a reference to a server
Server server =
SingleConnectionSession.GetInstance().GetServer("HOST1_PS01");
// execute the request
try
{
    // this call can time out
    server.ExecuteRequest(requestXml);
}
catch ( RimageException e)
{
    // handle the exception
}
// get the reply
string replyXml = server.GetXml();
// parse the reply xml
```



XmlWriter class requires the generated XML to include the DTD file path. This is accomplished by communicating the DTD name, version, and folder location of the DTD file to the API.

To generate a valid server request XML, overwrite two **XmlWriter** virtual methods `FormatRootElement()` and `FormatDetails()`. To write out the XML header for the request, overwrite the `FormatRootElement()` method.

 **Note:** `ProductionXmlWriter` and `ImageXmlWriter` classes write out the correct header automatically.


The following example shows how to generate the 'Get autoloader status' request.

C++

```
class SampleUserPSRequestXMLWriter : public XmlWriter
{
public:
    SampleUserPSRequestXMLWriter()
    {
        SetDtdName("ProductionServerRequest");
        SetDtdFileVersion("1.4");
        SetDtdFolderPath("c:\\rimage\\xml");
    }
protected:
    void FormatRootElement(XmlStringBuffer &xmlStrBuffer)
    {
        xmlStrBuffer.WriteStartElement(GetDtdName(), 0);
        xmlStrBuffer.WriteAttribute("ServerId", "SWRASKINREST_PS01");
        xmlStrBuffer.WriteAttribute("ClientId", "client1");
    }
    void FormatDetails(XmlStringBuffer &xmlStrBuffer)
    {
        xmlStrBuffer.WriteStartElement("GetServerStatus", 1);
        xmlStrBuffer.WriteAttribute("GetAutoloaderStatus", "true");
        xmlStrBuffer.WriteCloseElement("GetServerStatus");
    }
};
```

C#

```
class SampleUserPSRequestXMLWriter : XmlWriter
{
public SampleUserPSRequestXMLWriter()
{
    SetDtdName("ProductionServerRequest");
    SetDtdFileVersion("1.4");
    SetDtdFolderPath("c:\\rimage\\xml");
}
protected override void FormatRootElement(XmlStringBuffer xmlStrBuffer)
{
    xmlStrBuffer.WriteStartElement(GetDtdName(), 0);
    xmlStrBuffer.WriteAttribute("ServerId", "SWRASKINREST_PS01");
    xmlStrBuffer.WriteAttribute("ClientId", "client1");
}
protected override void FormatDetails(XmlStringBuffer xmlStrBuffer)
{
    xmlStrBuffer.WriteStartElement("GetServerStatus", 1);
    xmlStrBuffer.WriteAttribute("GetAutoloaderStatus", "true");
    xmlStrBuffer.WriteCloseElement("GetServerStatus");
}
}
```

 **Note:** For more information on customizing server request XML, refer to the Rapid Custom sample project located at `C:\Program Files\RimageSdk\ApiSdk\Samples\RapidApi`



Samples

The Rimage SDK installation includes sample projects for working with the Rapid API. By default these projects are placed in the C:\Program Files\RimageSdk\ApiSdk\Samples\RapidApi folder. The samples are broken into C++ and .NET (written in C#) samples. C++ samples are located under the \C++ subfolder and C# samples are in the \NET subfolder.

Three sample projects are available for each environment, they are:

- **Rapid Hello World** – this sample project is an introduction to the Rapid API. This sample takes the user through the most basic steps of connecting, creating and submitting a job, and disconnecting. This is meant to help you produce your first disc as quickly as possible.
- **Rapid Comprehensive** – this sample project includes examples of how to use the whole API. This includes setting up listeners, creating and customizing jobs, enumerating Rapid API objects, working with servers and alerts, and much more.
- **Rapid Custom project** – this sample project shows how to extend various classes to customize jobs and XML generation.

Deployment

C++ deployment

Build information

The Rapid API has been compiled using the Visual Studio 2005 VC8 compiler. Rimage DLLs include their version in the name of the file. The name/version has the following format:

```
<name>_<major>_<minor>_n_<interface>.dll
```

- Major version is seldom incremented, and only if a Rimage system undergoes a significant architectural change. For example, version 5.x to version 6.x – the Rimage system changed from file based to messaging/XML based.
- Minor version is incremented if a DLL is changed for a new release. Applications using this DLL will need to be rebuilt.
- “n” represents an internal build/bug fix version for a specific minor version. The actual File version of the dll will have a num in place of “n”. For example if the dll is named RmRapid_2_0_n_0.dll, the File version of this dll could be 2.0.26.1.
- Interface version represents iterations of the API itself. If the exported interface of the DLL itself is changed, this version is incremented and the applications using this DLL will need to be rebuilt.

 **Note:** The `_u` option indicates Unicode versions; no `_u` indicates non-Unicode versions.

Required files and folders

The following files and directories are required in C++ projects. Specify the paths and specify the .lib files (either Unicode or non-Unicode) as indicated.

Required DLL files (non-Unicode)

Installed by default in

```
C:\Program Files\RimageSdk\ApiSdk\bin.
```

```
RmRapid_2_0_n_1.dll
```

```
RmClient_8_0_n_5.dll
```

Required DLL files (Unicode)

Installed by default in

```
C:\Program Files\RimageSdk\ApiSdk\bin.
```

```
RmRapid_2_0_n_1_u.dll
```

```
RmClient_8_0_n_5_u.dll
```

VC 8.0 redistributables are required. Please refer to: [Microsoft Visual C++ 2005 Redistributable Package](#)

Required LIB files (non-Unicode)

```
RmRapid_2_0_n_1.lib
```

Required LIB files (Unicode)

```
RmRapid_2_0_n_1_u.lib
```

Required Include directory

Installed by default in C:\Program Files\RimageSdk\ApiSdk\include\rapid

Required #include statement

```
#include <RapidApiInclude.h>
```

(This file contains all headers files for the Rapid API)



.NET deployment

Build information

The Rapid API has been compiled using Visual Studio 2005 and .NET Version 2.0.

Rimage.Rapid.Api assembly implements the .NET API. This assembly can be used in any application written in a .NET supported language.

This assembly is strongly named, which among other things means that Common Run Time (CLR) takes the Assembly version of this assembly into account at load time.

Required files and folders

The following files are required in C#, VB.NET, or any other .NET project.

Required .NET Assembly files

Installed by default in C:\Program Files\RimageSdk\ApiSdk\bin.

`Rimage.Rapid.Api.dll`


The rest of the file list is identical to the Unicode list in C++ Required Files and Folders section.

Appendix A – Client ID and Job ID uniqueness rules

 **Note:** A *JobID* is automatically generated if you do not provide one.

Each *ClientID* and *JobID* must be unique. To ensure job uniqueness, Rimage makes the following recommendations:

- The *ClientID* must be used to connect to the Messaging Server (eMS). The Messaging Server requires *ClientID* uniqueness and will return an error when connected if a non-unique *ClientID* is detected. This ensures *ClientID* uniqueness.

 **Note:** Because more than one instance of an application can be run on one computer, the Rimage applications' *ClientID* is in the form of <HostName> + _ + <ApplicationInstanceld>.

- Integrators must ensure that the *JobIDs* they generate for a client application are unique. There is still a possibility that two clients will generate identical *JobIDs*, which is resolved as follows:
- Production and Imaging servers must take both *JobIDs* and *ClientIDs* into account to ensure order uniqueness internally to the server.

To reiterate:

- The *JobID* is unique in the client application's space.
- The *ClientID* is unique in the Messaging Server (eMS) space. The *ClientID* includes the <HostName> + _ + <ApplicationInstanceld>.

 **Note:** Alphanumeric character entries are typical for the *ClientID* and *JobID*. The period . and backslash \ must be excluded from the *ClientID* and *JobID* entries.



Appendix B – Reference documentation

For more information about Editlists, refer to the document located at:
C:\Program Files\RimageSdk\Manuals\Programming Guides\Editlists.pdf

For more information about label merge fields, refer to the document at:
C:\Program Files\RimageSdk\Manuals\Programming Guides\ Using Label Merge Fields.pdf

For more information on server alert Action IDs, refer to the document at:
C:\Program Files\RimageSdk\Manuals\Programming Guides\ Server Alert Action IDs Reference.pdf

For more information on CD Text, refer to the document located at:
C:\Program Files\RimageSdk\Manuals\Programming Guides\ Using CD Text.pdf