CUEScript Training Guide

RESTRICTED RIGHTS
Use, duplication, or disclosure by the Government is subject to restrictions as set forth in subparagraph (C)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013.

Manufacturer is:
Cleo Communications
4203 Galleria Drive
Rockford, IL 61111 USA
Phone: 815.654.8110
Fax: 815.654.8294
Email: sales@cleo.com
www.cleo.com

Support: 1.866.444.2536 or support@cleo.com

Cleo Communications reserves the right to, without notice, modify or revise all or part of this document and/or change product features or specifications and shall not be responsible for any loss, cost or damage, including consequential damage, caused by reliance on these materials.

This document may not be reproduced, stored in a retrieval system, or transmitted, in whole or in part, in any form or by any means (electronic, mechanical, photo-copied or otherwise) without the prior written permission of Cleo Communications.

©2008 CLEO COMMUNICATIONS ALL RIGHTS RESERVED. CLEO IS A REGISTERED TRADEMARK OF CLEO COMMUNICATIONS. ALL OTHER BRAND NAMES USED ARE TRADEMARKS OR REGISTERED TRADEMARKS OF THEIR RESPECTIVE COMPANIES.

©2008 CUESCRIPT IS A REGISTERED TRADEMARK OF CUESOFT.COM, INC

Table of Contents

Introduction 5

RMCS Architecture 6

Typical Scripting Scenario 7

1. Specialization 7
2. Script Sent to Communication Server 7
3. Script Runs on the Communication Server 7
4. Post-job Script 7

Scripts in RMCS 8

Adding New Scripts 8

Importing/Exporting Scripts 10

Specifying a Script for a Job 11

XML Primer 13

Elements/Tags 13

Attributes 13

Other XML Components 14

Comments 14

Empty Elements 14

Processing Instructions 14

Entities 15

CDATA Section 15

XML Namespaces 15

XML Documents as Nodes 16

XML Resources 17

XPath Primer 18

XPath Resources 19

Getting Started 20

Selecting Tools 20

Hello World 20

Using "format" 21

Scripting Language 22

Structure Tags 22

Variables 22

Data Types 23

Strings 23

Numbers 24

Node Lists 24

Arrays 25

Flow Control 25

<if> 25

<else> 25

CLEO CONFIDENTIAL. ©2008 CLEO Communications. All rights reserved. Cleo is a registered trademark of CLEO Communications

<switch> 26
<for> 26
<while> 27
<repeat> 27
<foreach> 27
<continue> and <break> 27
Functions **28**
Libraries **28**
Embedded Expressions **29**
Exception Handling **30**
CUEScript Internal Objects **32**
The $system Object **32**
Testing and Debugging your Script in RMCS **33**
Development Principles **33**
Testing and Debugging Process **33**
Testing as Job Server Job 34
Testing as a Communication Server Job 36
Testing on the Remote Server 39
Specialization **40**
Post Job Scripting **43**
Communication Server Scripts **44**
Working with Connections **45**
Initializing a connection 45
Connecting 45
Creating a New Connection 46
Setup a Listening Connection 46
Appliance Registration **48**
Sending Commands to the Appliance **49**
RMCS Events **49**
Using the Session Object **52**
Using Timers 52
Communicating with the Job Server 53
Transferring Files **53**
Job Server Scripts **56**
Remote Server Scripts **58**
RMCS Update **61**
Value Cache **63**

CLEO CONFIDENTIAL. ©2008 CLEO Communications. All rights reserved. Cleo is a registered trademark of CLEO Communications

Introduction

The Remote Machine Communication System (RMCS) is a highly scalable system for communicating with and managing remote computers. An important part of this system is a versatile XML-based scripting engine called the CUEScript Scripting Engine.

Having an in-depth understanding of CUEScript and how it works in RMCS will allow you to extend the functionality of RMCS to do what you need it to do. The objective of this training guide/course is to provide that foundation of understanding.

CLEO CONFIDENTIAL. ©2008 CLEO Communications. All rights reserved. Cleo is a registered trademark of CLEO Communications

RMCS Architecture

Figure 1 displays a picture of how the components that make up the RMCS system are connected. Note that all these components can exist on a single system or be spread across multiple machines.

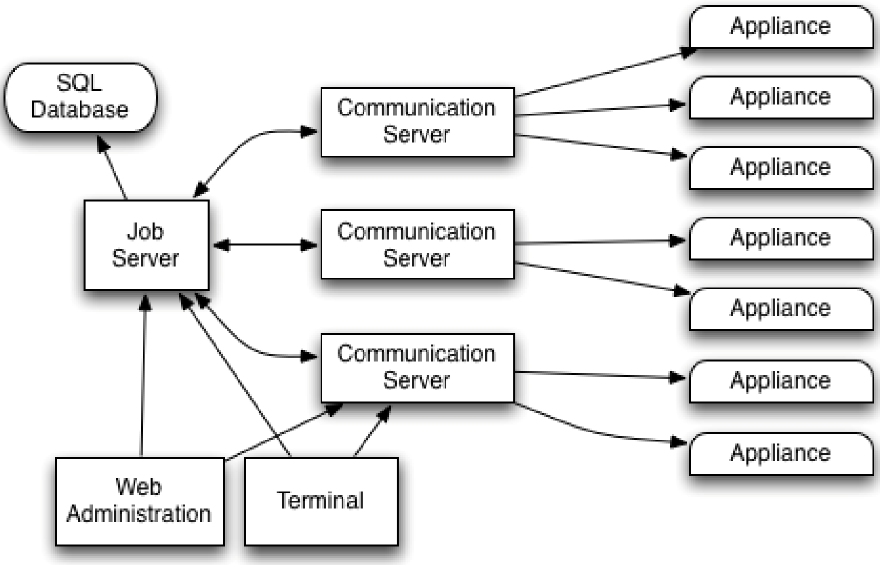


Figure 1 – RMCS Architecture Diagram

Job Server – This component is the core of the RMCS system. It manages run jobs, hosts a scheduler, sends configurations, and reports status. Specialization scripting occurs on the job server to prepare a job for running on the Communication Server. Stand-alone scripting jobs can also run on the Job Server. These are useful for automated configuration and maintenance tasks. Special scripting can also occur when a job is completed.

Communication Server – This is the workhorse where most of the work is done when connecting to an appliance. It handles connections, communications, and file transfers. The scripting that occurs on Communication Server is the most visible in the system.

Remote Server (Appliance) – This is the service running on remote machines that are communicated with. Generally appliances are communicated with, though they are capable of initiating communications with a Communication Server. Scripting that occurs on the Remote Server can be used for maintenance tasks and initiating connections. These scripts can also embed a schedule.

Web Administration – The web interface interacts with the job server and database to configure the RMCS system. Scripts can be created, edited and assigned to jobs using this interface.

CLEO CONFIDENTIAL. ©2008 CLEO Communications. All rights reserved. Cleo is a registered trademark of CLEO Communications

Terminal – The Terminal client application is used to connect to appliances for interactive sessions. Through the command interface it can also run local scripts (which act like macros for common tasks) and remote scripts (to kick off remote maintenance tasks).

Typical Scripting Scenario

Every job that runs as part of RMCS has a single script assigned to it. The most typical scripting scenario for the RMCS system is when a job runs on the Communication Server. When this kind of job runs, it creates a job instance item for each appliance as part of the job.

1. Specialization

As the job is creating each job instance item, it may specialize the script. If the script contains a "special" section, that section of script is run. It can be used to populate the script with phone numbers, communication settings and other database fields. Specialization is an optimization. It is only run once for a given job, script, and appliance. If the job, script, or appliance attributes change then specialization is re-run for that combination.

2. Script Sent to Communication Server

Once the job instance item is created, it is placed in a job queue where it waits for an available Communication Server resource. When a resource becomes available, it is removed from the queue and sent to the Communication Server.

3. Script Runs on the Communication Server

The Communication Server will then initialize a series of objects and begin running the script. The script can then use those objects to dial up or connect to an appliance, transfer files, or do whatever else is needed. When the script ends, the job server is notified with the exit status (success or error).

4. Post-job Script

When a job is complete (all the job instance items associated with that job have reported back with either success or error), the Job Server will look for a "post" script section in the original script to run. This section is run on the Job Server and can be used to reschedule the job, run a new job, or update database fields.

CLEO CONFIDENTIAL. ©2008 CLEO Communications. All rights reserved. Cleo is a registered trademark of CLEO Communications

Scripts in RMCS

Adding New Scripts

The place to add new scripts to RMCS is to use the Scripts page. After logging into RMCS, select Scripts from the Jobs menu.

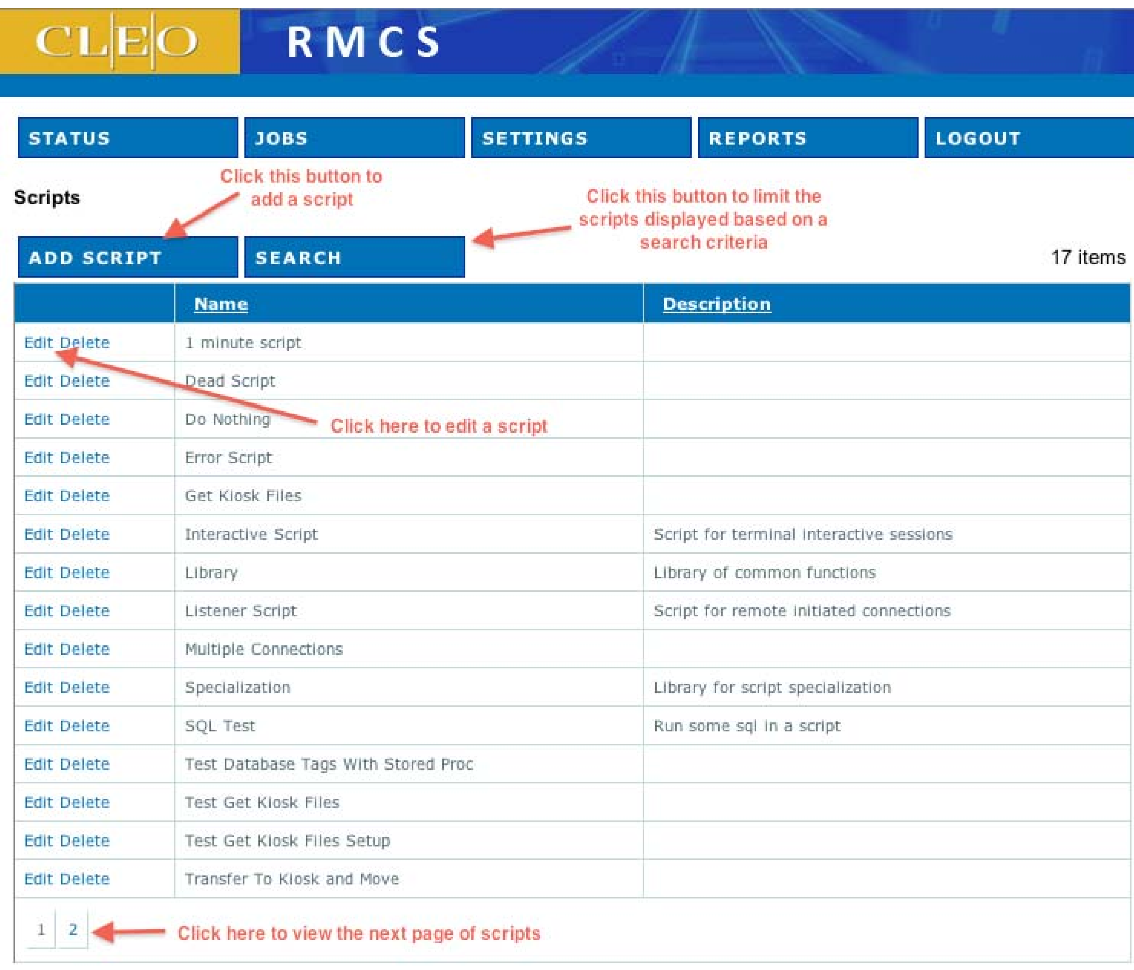


Figure 2 – Scripts Page

Click the “Add Script” button to add a new script.

CLEO CONFIDENTIAL. ©2008 CLEO Communications. All rights reserved. Cleo is a registered trademark of CLEO Communications

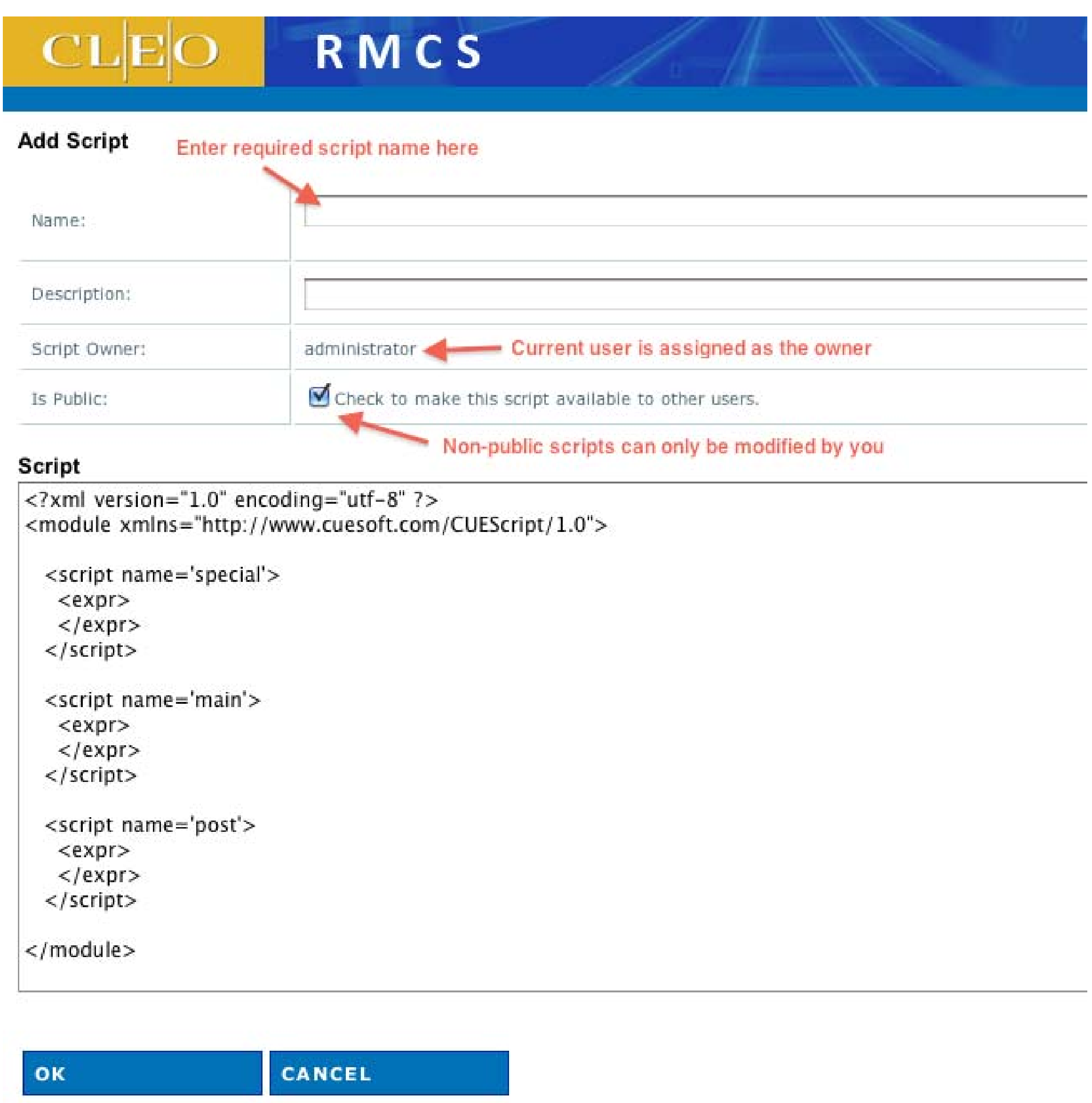


Figure 3 – Add Script Page

You must enter a script name. This name will show up in other drop-downs in the system when selecting a script. You can decide whether to make this script available to other users of the system. If the script is public, other users may be able to modify the script. If it is private, it won't be visible to other non-administrator users on the system.

You'll notice that a default script template has been created for you. You can start with this, or more likely start with another script that you've copied from somewhere else.

While you edit the scripts in the browser, it is not a very effective tool for doing so. Chances are that you'll find yourself copying and pasting scripts to and from a text or XML editor.

You can also download the CUEScript Editor (a free download from http://www.cuesoft.com/CUEScript/). This editor is designed for editing and testing your scripts. More on this later.

CLEO CONFIDENTIAL. ©2008 CLEO Communications. All rights reserved. Cleo is a registered trademark of CLEO Communications

Importing/Exporting Scripts

If you're only working with a single script, you can get away with copy and paste. If you're working with multiple scripts, you're better off using the import/export functionality.

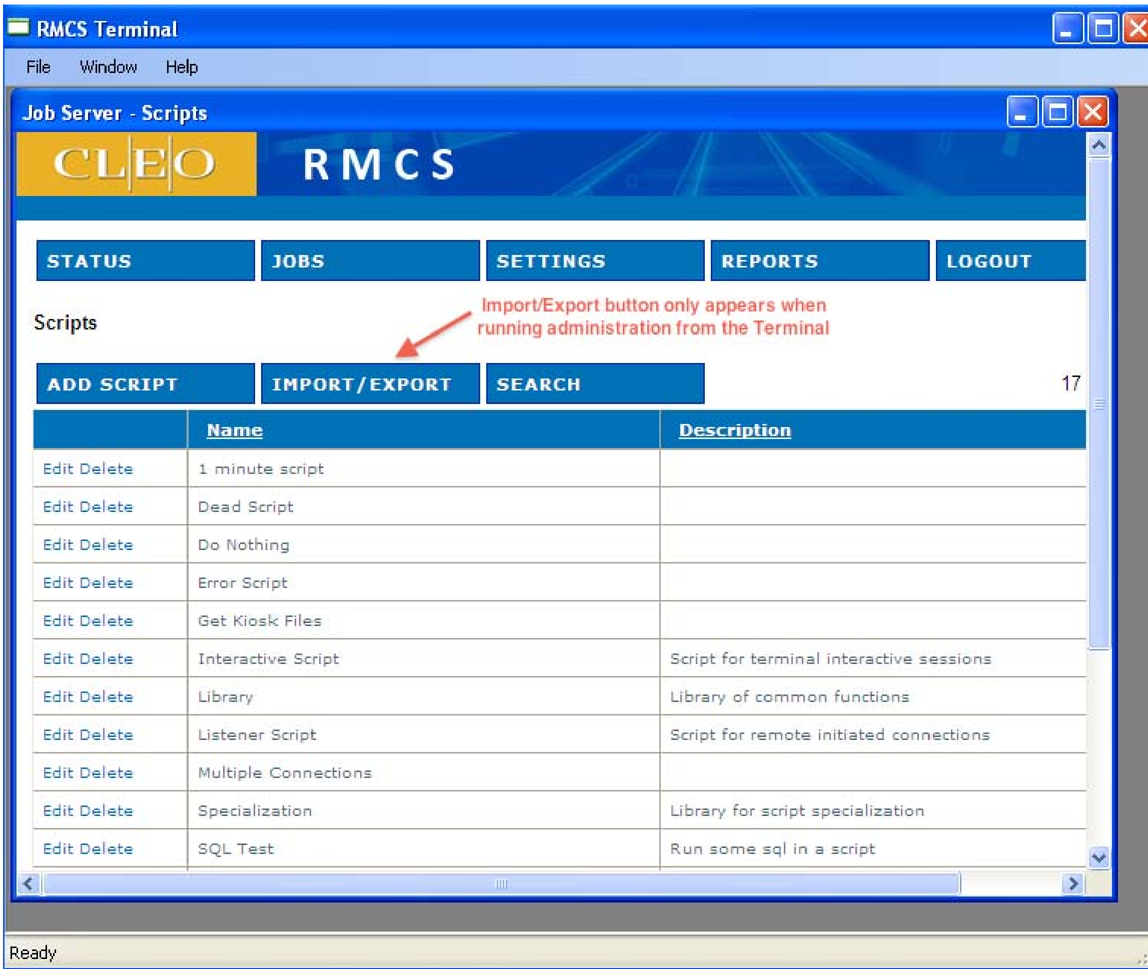


Figure 4 – Scripts Page inside Terminal application

Export allows you to save one or more scripts from RMCS to a folder on your local hard drive. Import will take those files from your local disk and add them or update them in RMCS.

In order import/export scripts, you must run the administrator from within the Terminal application (File | Administration…). You cannot do this from a regular browser. When you go to the Scripts page, you'll notice the addition of the "Import/Export" button. Clicking this button will bring up the import/export dialog.

CLEO CONFIDENTIAL. ©2008 CLEO Communications. All rights reserved. Cleo is a registered trademark of CLEO Communications

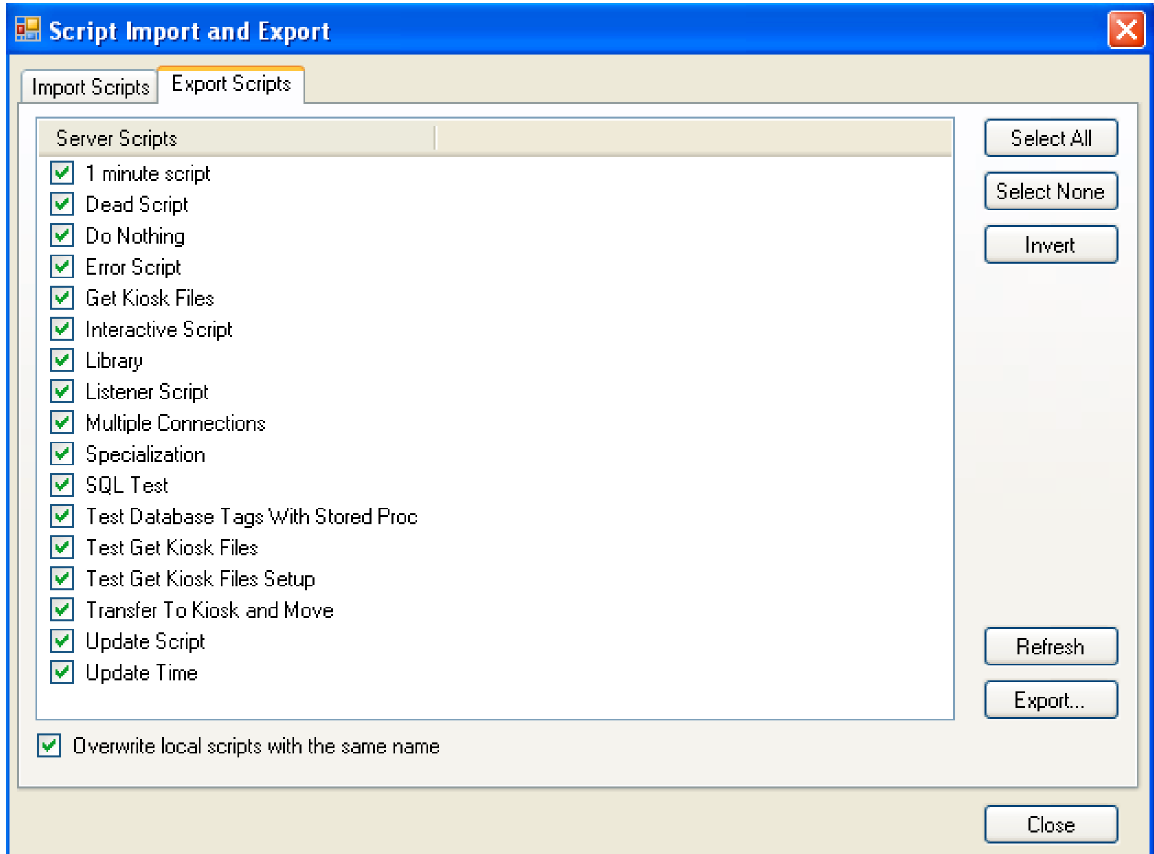


Figure 5 – Import/Export Dialog

Use the export tab to select the scripts to export and click the "Export…" button to select a folder to export the scripts to. Use the import tab to import scripts. On that tab, select the folder using the "Browse…" button and then select the scripts to import. Click the "Import" button to perform the import.

Both tabs have an "Overwrite…" check box. If checked, scripts with the same name will overwrite exiting scripts. If not checked and a script with the same name exists in a location, the import/export is skipped for that script.

Specifying a Script for a Job

To specify a script for a job, go to the Job Definitions page by selecting Definitions from the Jobs menu. Then when you edit a definition or add a definition by clicking the "Add Definition" button, you'll see a page like Figure 6.

In the Script drop down box, you'll see a list of your private scripts and any public scripts that are available.

CLEO CONFIDENTIAL. ©2008 CLEO Communications. All rights reserved. Cleo is a registered trademark of CLEO Communications

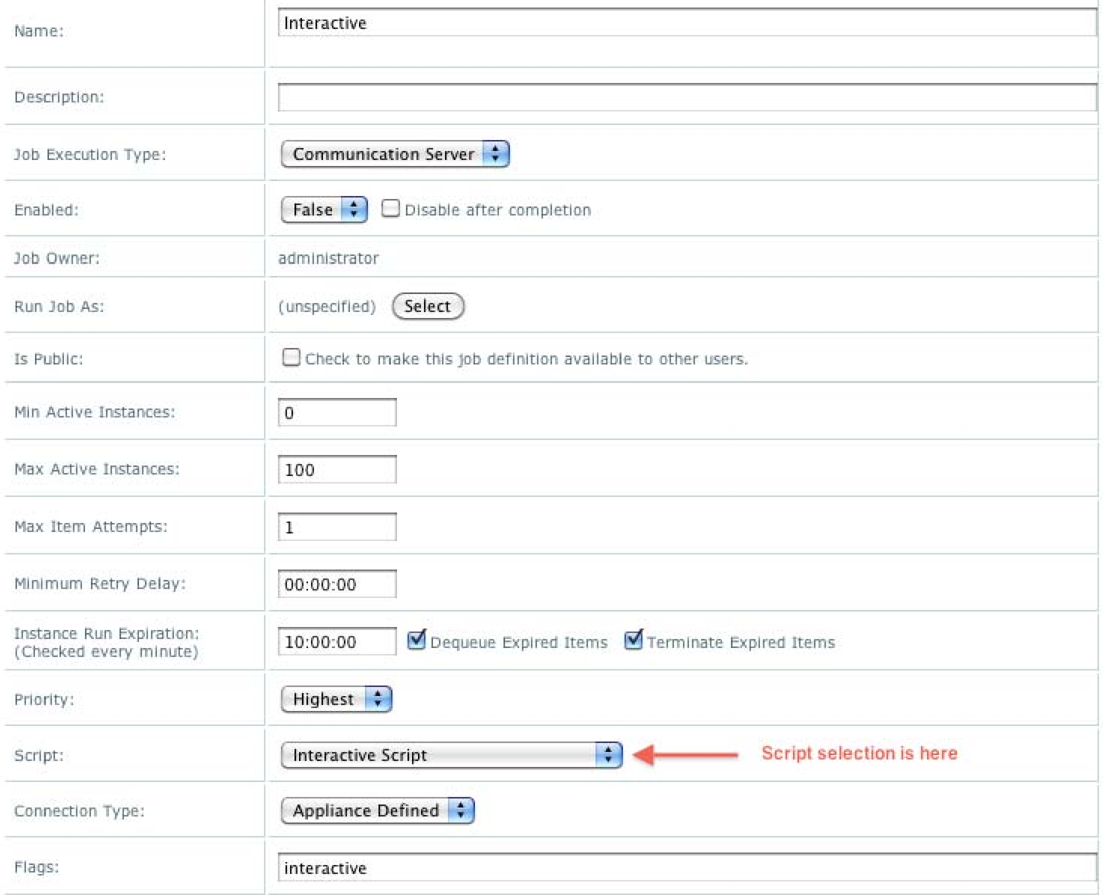


Figure 6 – Edit Job Definition Page

RMCS has a special job called "Interactive". This job definition MUST exist in order for the Terminal application to connect to appliances. The interactive script is a special script used to connect to an appliance. Once connected, it will wait for a Quit command before dropping the connection.

CLEO CONFIDENTIAL. ©2008 CLEO Communications. All rights reserved. Cleo is a registered trademark of CLEO Communications

XML Primer

XML is the markup language used for CUEScript. There are many misconceptions about XML. Probably the most common one is the idea that it is a programming language. It's not. XML is simply a way to markup data. The markup can be used to define structure.

Elements/Tags

XML is marked up using tags. Below is a sample XML document.

```
<?xml version="1.0"?>
<note>
    <to>Tove</to>
    <from>Jani</from>
    <heading>Reminder</heading>
    <body>Don't forget me this weekend!</body>
</note>
```

The <?xml version="1.0"?> is a special tag that identifies this as an XML document. It's not required but you'll often see it as the first line of an XML document.

- Every XML document can have only 1 root element. "note" is the root element of this document.
- Each element must have a start tag and an end tag. "<note>" is an example of a start tag and "</note>" is its corresponding end tag. End tags are like start tags but start with "</".
- Inside an element can be other elements or text. You can have any number of levels within an element.

Attributes

Elements can have attributes that provide more information about the element.

```
<bookstore>
  <book category="CHILDREN">
    <title>Harry Potter</title>
    <author>J K. Rowling</author>
    <year>2005</year>
    <price>29.99</price>
  </book>
  <book category='WEB'>
    <title>Learning XML</title>
    <author>Erik T. Ray</author>
    <year>2003</year>
    <price>39.95</price>
  </book>
</bookstore>
```

CLEO CONFIDENTIAL. ©2008 CLEO Communications. All rights reserved. Cleo is a registered trademark of CLEO Communications

In this example ‘category=”CHILDREN”’ is an example of an attribute.

- Attributes must be unique for an element but there is no limit to the number of attributes for an element.
- Attribute values can be enclosed in single or double quotes. Whichever type of quote you start with is the type of quote you must end with.
- Attribute values cannot span lines.

Other XML Components

```
<?xml version=”1.0”?>
<appliances>
  <!-- West coast appliances -->
  <applianceGroup zone=”PST”>
    <?display color=”red”?>
    <appliance number=”4158377985”>
      <city>Danville</city>
      <state>CA</state>
      <note><![CDATA[This appliance does not answer on Sundays!]]></note>
    </appliance>
  </applianceGroup>
  <!-- Mountain appliances -->
  <applianceGroup zone=”MST”/>
</appliances>
```

The example above lists several other components of XML.

Comments

You can insert comments into a document by surrounding them with <!-- and -->. Any text within the comment markup is ignored by whatever application using the data.

Empty Elements

If an element doesn’t have any child elements or text, it can be closed with a “/” instead of requiring an end tag.

For example:

```
<city></city>
```

can be represented as

```
<city/>
```

Processing Instructions

These are special instructions to an application. They are formatted using <?somedata?>. The use of these is application-specific. <?display color=”red”?> is an example of a processing instruction in the document above.

CLEO CONFIDENTIAL. ©2008 CLEO Communications. All rights reserved. Cleo is a registered trademark of CLEO Communications

Entities

You might be wondering how to represent < within the text data or a literal “ within an attribute.

The following example is not valid XML because the parser does not know that <10 is not the start of a tag.

```
<result>The answer is < 10.</result>
```

Entities are used to represent those special characters that are part of the markup. If you have data that uses one of these characters, use the corresponding entity.

Character	Entity
<	<
>	>
“	"
‘	'
&	&

The following is valid XML

```
<result>The answer is &lt; 10.</result>
```

CDATA Section

The CDATA section is a way to avoid having to use entities. You can use any characters within a CDATA section. The XML parser will ignore the meaning of any entities or special characters within “<!CDATA[“ and “]]>”.

CDATA sections cannot include nested CDATA sections or the string “]]>”.

XML Namespaces

In XML, a developer or author defines element names. This often results in a conflict when trying to mix XML documents from different XML applications.

For example, this XML carries HTML table information:

```
<table>
  <tr>
    <td>Apples</td>
    <td>Bananas</td>
  </tr>
</table>
```

And this XML carries information about a piece of furniture:

```
<table>
  <name>African Coffee Table</name>
  <width>80</width>
```

CLEO CONFIDENTIAL. ©2008 CLEO Communications. All rights reserved. Cleo is a registered trademark of CLEO Communications

```
  <length>120</length>
</table>
```

If these XML fragments were added together, there would be a name conflict since they both contain the <table> element. This problem is solved by adding a prefix to the element. The prefix must also be defined and this is done with a namespace attribute.

```
<root>

<h:table xmlns:h="http://www.w3.org/TR/html4/">
  <h:tr>
    <h:td>Apples</h:td>
    <h:td>Bananas</h:td>
  </h:tr>
</h:table>

<f:table xmlns:f="http://www.alswarehouse.com/furniture">
  <f:name>African Coffee Table</f:name>
  <f:width>80</f:width>
  <f:length>120</f:length>
</f:table>

</root>
```

You'll notice that "h:" prefix was added to the HTML table and the "f:" prefix was added to the furniture table. The xmlns:[prefix]="namespace" attribute defines the namespace for the prefix. The common convention is to define the namespace as a URI but it can be any string.

XML Documents as Nodes

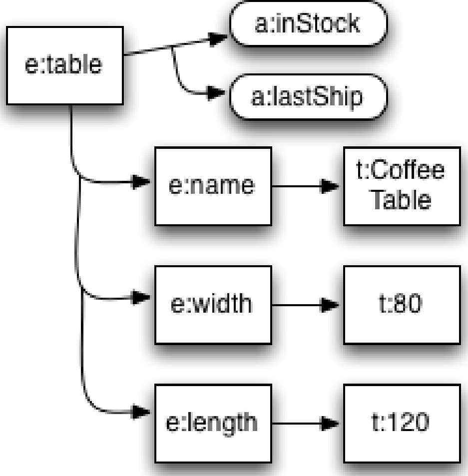
One of the nice features of XML is that it can be represented in computer memory as a tree-like node structure. Understanding this tree-like structure will help you understand XPath (in the next section) and how CUEScript will operate later.

Let's take the sample XML document:

```
<table inStock="true" lastShip="3/10/2010">
  <name>Coffee Table</name>
  <width>80</width>
  <length>120</length>
</table>
```

As a tree of nodes, it would be represented as:

CLEO CONFIDENTIAL. ©2008 CLEO Communications. All rights reserved. Cleo is a registered trademark of CLEO Communications



e: refers to an element node.

t: refers to a text node.

a: refers to an attribute node.

XML Resources

http://www.w3schools.com/xml/default.asp

http://www.learn-xml-tutorial.com/

CLEO CONFIDENTIAL. ©2008 CLEO Communications. All rights reserved. Cleo is a registered trademark of CLEO Communications

XPath Primer

XPath is a fairly complex syntax for describing one or more nodes in an XML tree. For CUEScript, you can use XPath to locate a node and then modify that node or sub-nodes. In this way, the script is self-modifying. This is done during specialization to insert appliance information into the script before it goes to a Communication Server.

```
<?xml version="1.0"?>
<project>
 <title>The Xpath project</title>
 <participants>
  <participant>
   <FirstName>Daniel</FirstName>
   <qualification>8</qualification>
   <description>Daniel will be the tutor</description>
   <FoodPref picture="dolores_001.jpg">Sea Food</FoodPref>
  </participant>
  <participant>
   <FirstName>Jonathan</FirstName>
   <qualification>5</qualification>
   <FoodPref picture="dolores_002.jpg">Asian</FoodPref>
  </participant>
  <participant>
  <FirstName>Bernadette</FirstName>
  <qualification>8</qualification>
   <description>Bernadette is an arts major</description>
  </participant>
  <participant>
  <FirstName>Nathalie</FirstName>
  <qualification>2</qualification>
  </participant>
 </participants>
 <problems>
  <problem>
   <title>Initial problem</title>
   <description>We have to learn something about Location Path</description>
   <difficulty level="5">This problem should not be too hard</difficulty>
  </problem>
  <solutions>
  <item val="low">Buy a XSLT book</item>
  <item val="low">Find an XSLT website</item>
  <item val="high">Register for a XSLT course and do exercices</item>
  </solutions>
  <problem>
   <title>Next problem</title>
   <description>We have to learn something about predicates</description>
   <difficulty level="6">This problem is a bit more difficult</difficulty>
  </problem>
  <solutions>
  <item val="low">Buy a XSLT book</item>
  <item val="medium">Read the specification and do some exercises</item>
  <item val="high">Register for a XPath course and do exercices</item>
  </solutions>
 </problems>
</project>
```

CLEO CONFIDENTIAL. ©2008 CLEO Communications. All rights reserved. Cleo is a registered trademark of CLEO Communications

With XPath, you specify a descriptive path to the node or nodes that you're looking for. The return value for an XPath expression can be a single node, a node list, a string, boolean, or number. The best way to understand this is to look at some examples.

```
/project
```

This returns a node list with just the root element in it. The slash (/) means to start root of the document. "project" matches elements under the document. Note that starting with "./" would start with the current node. This becomes relevant when the reference point is a node in the document.

```
/participant
```

would return no nodes since there is no element called participant under the document root.

```
//participant
```

returns all participant nodes in the document regardless of level.

```
//problem/title
```

returns just those title nodes under any problem node in the document.

```
//difficulty[@level]
```

returns the difficulty nodes have a level attribute.

```
//difficulty[@level='5']
```

returns the difficulty nodes whose level attribute is equal to five.

```
//difficulty[@level='5']/text()
```

returns the text of those nodes.

XPath Resources

http://www.w3schools.com/xpath/

http://edutechwiki.unige.ch/en/XPath_tutorial_-_basics

CLEO CONFIDENTIAL. ©2008 CLEO Communications. All rights reserved. Cleo is a registered trademark of CLEO Communications

Getting Started

Selecting Tools

To work with CUEScript you are going to want an XML editor or text editor. There are many fine professional or free XML editors available. An XML editor will check the structure of your XML document and can validate the document against the CUEScript schema. This will help you find common typing mistakes.

If you have Visual Studio, that can be used to edit XML documents as well. A text editor is fine too.

The CUEScript Studio editor is a free editor that can be downloaded from the CUESoft web site (http://www.cuesoft.com/cuescript). Besides being an editor, you can run and debug scripts that you write.

When it comes down to it, it's about your personal preference on what tools you'd like to use.

Hello World

Let's start with one of the simplest scripts:

```
<module xmlns="http://www.cuesoft.com/CUEScript/1.0">
  <script>
    <output>
            Hello World!
    </output>
  </script>
</module>
```

If you output this script in the CUEScript Editor, it will output the text "Hello World!". Let's take a look at this script in detail.

The "module" tag is essentially a wrapper tag. It's not required if you have a single script section, but is useful for grouping multiple script sections. An XML document can have only 1 root node, and the "module" tag is acting as that root.

The xmlns="http://www.cuesoft.com/CUEScript/1.0" defines the default namespace (the one without any prefix) as the CUEScript namespace. The CUEScript namespace declaration is required for all CUEScript documents.

The "script" tag is where execution begins.

The "output" tag simply outputs the contents to the output as specified for the CUEScript engine (in this case the output window). Note that the "output" tag is not supported in the RMCS system. Instead a $log variable is supplied to log output.

CLEO CONFIDENTIAL. ©2008 CLEO Communications. All rights reserved. Cleo is a registered trademark of CLEO Communications

The hello world script for RMCS would look like this:

```
<module xmlns="http://www.cuesoft.com/CUEScript/1.0">
 <script>
  <expr>
         $log.Info("Hello World!");
  </expr>
 </script>
</module>
```

The “expr” tag is a common tag which allows a series of statements to be specified as the child of that tag. “Info” is a method name on the $log object that displays informational text. Other method names on the $log object are “Error”, “Warning”, and “Debug”. For RMCS, anything logged using the $log object is stored with the job instance item when it exits.

Using “format”

The “format” function allows you to format strings using the .NET formatting convention. This will be especially useful for logging.

```
<module xmlns="http://www.cuesoft.com/CUEScript/1.0">
 <script name="main">
  <expr>
    $greeting := "Hello";
         $log.Info( format("{0} World!", $greeting) );
  </expr>
 </script>
</module>
```

A useful resource for formatting:

http://blogs.msdn.com/b/kathykam/archive/2006/03/29/net-format-string-101-c_2300_-strings.aspx

CLEO CONFIDENTIAL. ©2008 CLEO Communications. All rights reserved. Cleo is a registered trademark of CLEO Communications

Scripting Language

Structure Tags

The “module” and “script” tags make up the basic structure of a CUEScript.

```
<module xmlns="http://www.cuesoft.com/CUEScript/1.0">
	<script name="special">
		<stop/>
		<expr>
			$output.WriteLine("special section");
	</expr>
	</script>

	<script name="main">
		<expr>
			$output.WriteLine("main section");
		</expr>
	</script>

	<script name="post">
		<expr>
			$output.WriteLine("post section");
		</expr>
	</script>
</module>
```

The “module” tag is just a placeholder tag. It does not affect execution.

The “script” tag can have a “name” attribute that specifies the name of the script section. In most instances, the script engine will look the “main” script section and run that section.

The “expr” is common tag that you’ll use to include expressions.

The “stop” tag halts execution of the script (as a successful exit).

Variables

Variables can be defined anywhere in the script. Use the “var” tag to define a variable.

```
<module xmlns="http://www.cuesoft.com/CUEScript/1.0">
	<script>
		<var name="phone">"303-999-3969"</var>
		<var name="firstName" expr="'Alex'"/>
		<var name="lastName" expr="'Groenevelt'"/>
		<var name="fullName">$lastName ^ ", " ^ $firstName</var>
		<var name="age" expr="41"/>
		<expr>
			$birthDate := "7/23/69";
			$output.WriteLine( "Name: " ^ $fullName);
			$output.WriteLine( "Phone: " ^ $phone);
			$output.WriteLine( "Age: " ^ $age);
			$output.WriteLine( "Birth Date: " ^ $birthDate );
```

CLEO CONFIDENTIAL. ©2008 CLEO Communications. All rights reserved. Cleo is a registered trademark of CLEO Communications

```
                    </expr>
          </script>
</module>
```

The default scope for variables is at the tag level that they are defined at.

Variables defined within “expr” tags are scoped at the same level as the “expr” tag.

Adding the attribute scope=”global” to a “var” tag will give the variable global scope.

Data Types

Variables have types. Variables can hold any type of object but there are common types that you should be aware of.

Strings

You can define a string by placing the value in quotes.

```
<var name=”mystrvar” expr=”’my string’”/>
```

or

```
<expr>
  $mystrvar := “my string”;
</expr>
```

Use the ^ to concatenate strings as you’ve seen in the examples above.

Use the string() function to convert other values to a string type.

```
<expr>
  $age := 41;
  $ageAsString := string($age);
</expr>
```

For .NET objects stored, use the .NET ToString() member.

```
<expr>
  $sb := new( "System.Text.StringBuilder" );
  $sb.Append( "Strings" );
  $sb.Append( " are nice!" );
  $output.WriteLine( $sb.ToString() );
</expr>
```

There are several built in functions for performing basic string manipulation (string-length, substring, concat, starts-with, etc.)

The .NET String class is the core string type so any methods available to the string class are also available to you. See http://msdn.microsoft.com/en-us/library/system.string.aspx

The .NET StringBuilder class can also be used to manipulate strings. See http://msdn.microsoft.com/en-us/library/system.text.stringbuilder(VS.71).aspx

Numbers

There are 2 core numeric types native to CUEScript, integers and decimal numbers. Generally the value will be stored as an integer unless a decimal point is in the value.

```
$number := 10; <!-- integer -->
```

```
$number := 10.0 <!-- decimal -->
```

You can perform all the operations you'd expect to be able to perform on numbers (+, -, *, div). Division is performed using the keyword "div" since the "/" has other meanings in CUEScript expressions.

Use the number() and decimal() functions to convert values to integer and decimals respectively.

Node Lists

Most XPath expressions return node lists.

```
<s:module xmlns:s="http://www.cuesoft.com/CUEScript/1.0">
  <data>
    <appliance name="Concord"/>
    <appliance name="Walnut Creek"/>
    <appliance name="Oakland"/>
    <appliance name="Sacramento"/>
  </data>
  <s:script name="main">
    <s:expr>
      $appliances := //data/appliance;
      $output.WriteLine( "Found " ^ $appliances.Count ^ " items." );
      $output.WriteLine( "Second item is: " ^ $appliances[1].GetAttribute("name"));
      $output.WriteLine( "As a single node we got: " ^ tonode($appliances).GetAttribute("name"));
    </s:expr>
  </s:script>
</s:module>
```

Those lists are implemented as the .NET List<> object. "Count" will tell you the number of items in the list and the lists can be indexed as an array.

The tonode() method is a handy way of converting a node list to the first node in that list. This is useful when you know the resulting XPath query is only returning a single node.

CLEO CONFIDENTIAL. ©2008 CLEO Communications. All rights reserved. Cleo is a registered trademark of CLEO Communications

Arrays

Arrays are useful ways of constructing lists of data. You can define an array within script or build one dynamically using the .NET ArrayList object.

Refer to the CUEScript Developer's Guide for examples.

Flow Control

Flow control tags allow you to conditionally or repetitively run script.

<if>

The <if> tag is the most common conditional tag. If the condition specified in the test attribute it true, the contents of the <if> tag are run.

```
<if test="$connection.Connected()">
  <!-- run connection logic here -->
</if>
```

Use the not() operator to inverse the value of the test.

```
<if test="not($connection.Connected())">
  <!-- run disconnected logic here -->
</if>
```

<else>

The else tag is part of the <if> tag in that if the <if> test is false, the logic in the else tag is run. The <else> tag must immediately follow the </if> tag at the same level of the document as the <if> tag.

```
<if test="$connection.Connected()">
  <!-- run connection logic here -->
</if>
<else>
  <!-- run disconnected logic here -->
</else>
```

The <else> tag can also have a "test" attribute in which case the else portion is only run if the test is true. Multiple else cases can follow one another.

```
<if test="$age &lt; 18">
  <!-- person is a minor -->
</if>
<else test="$age >=18 and $age &lt; 30">
  <!-- person is young -->
</else>
<else test="$age >= 30 and $age &lt; 60">
  <!-- person is middle aged -->
</else>
<else>
  <!-- not gonna say -->
</else>
```

CLEO CONFIDENTIAL. ©2008 CLEO Communications. All rights reserved. Cleo is a registered trademark of CLEO Communications

<switch>

The switch tag is for matching an expression against multiple values to determine what action should be taken. The matching expressions are defined in the <case> tag. A <default> tag can be specified for an action if none of the case statements match the original expression.

```
<switch test="$connection.Connect()">
 <case value="0">
  <!-- Calls the registration check function.  It might throw. -->
  <return expr="CheckRegistration( $shouldFail )"/>
 </case>

 <case value="-1">
  <raise code="-1" explanation="Connect fail!"/>
 </case>

 <case value="-2">
  <expr>
   $command.WriteClientResponse( "Authentication failed.");
  </expr>
  <raise code="-2" explanation="Authentication failed!"/>
 </case>

 <case value="1">
  <expr>
   $log.Info( "Retrying connect." );
  </expr>
 </case>

 <default>
  <expr>
   $log.Error( "Unknown return code from connect" );
  </expr>
 </default>
</switch>
```

The "value" attribute of the <case> tags can include a constant or expression.

<for>

The <for> tag is most often used when you know exactly how many times to repeat an action. The <for> tag has 3 attributes: "init" can be specified to perform some initialization, "test" must be specified with an expression that returns a boolean result to determine if the looping should continue, "step" is run after each iteration of the loop and can be used to increment some value. Only the "test" attribute is required.

```
<for init="$i := 0" test="$i &lt; 10" step="$i := $i + 1">
 <!-- this will loop 10 times -->
</for>
```

You can use the <break/> tag to break out of a loop at any time.

CLEO CONFIDENTIAL. ©2008 CLEO Communications. All rights reserved. Cleo is a registered trademark of CLEO Communications

<while>

The <while> loop will continue to loop so long as the test attribute is true. The test is checked at the beginning of the loop.

```
<expr>
 $i := 0;
</expr>
<while test="not($connection.Connected) and $i lt; 5">
 <!-- try and connect -->
 <expr>
  $i := $i + 1;
 </expr>
</while>
```

<repeat>

The <repeat> loop is like the while except that the test condition is not checked until the end of the loop. Refer to the CUEScript Developer's Guide for syntax.

<foreach>

The <foreach> tag is used to iterate through each item in a collection. A collection can be array or list of nodes.

```
<s:module xmlns:s="http://www.cuesoft.com/CUEScript/1.0">
 <data>
  <appliance name="Concord"/>
  <appliance name="Walnut Creek"/>
  <appliance name="Oakland"/>
  <appliance name="Sacramento"/>
 </data>
 <s:script name="main">
  <s:foreach var="node" in="//data/appliance">
   <s:expr>
    $output.WriteLine( "found: " ^ $node.GetAttribute("name"));
   </s:expr>
  </s:foreach>
 </s:script>
</s:module>
```

<continue> and <break>

Continue is used to jump to the loop test condition without executing the remaining statements in the loop.

```
<while test="$filesLeft > 0">
 <if test="$system.Invoke( 'System.IO.File', 'Exists', $files[$filesLeft])">
  <expr>
   $filesLeft := $filesLeft - 1;
  </expr>
  <continue/>
 </if>
 <!-- transfer the file here -->
```

CLEO CONFIDENTIAL. ©2008 CLEO Communications. All rights reserved. Cleo is a registered trademark of CLEO Communications

```
</while>
```

Break is used to interrupt the loop. If you have loops inside of loops, you can break out of outer loops using the "level" attribute.

Functions

Functions are a way of grouping commonly executed script. Functions allow you to organize your scripts and improve readability. As we'll see in a little while, you can organize functions into libraries that can be referenced from more than one script.

```
<module xmlns="http://www.cuesoft.com/CUEScript/1.0">

	<function name="first">
		<expr>
			$output.WriteLine( "first 0: " ^$0);
			$output.WriteLine( "first 1: " ^$1);
		</expr>
	</function>

	<function name="second">
		<parameter name="fruit1" default="'kiwi'"/>
		<parameter name="fruit2" default="'banana'"/>
		<expr>
			$output.WriteLine( "second 0: " ^$fruit1);
			$output.WriteLine( "second 1: " ^$fruit2);
		</expr>
		<return expr="$fruit2"/>
	</function>

	<script>
		<expr>
			first( 'apple', 'orange' );
			$result := second( 'apple' );
			$output.WriteLine( "result: " ^ $result );
		</expr>
	</script>
</module>
```

Use the name attribute to name a function. If you don't specifically specify the name of function parameters, they are automatically named $0, $1, etc.

Use the <return> tag to send a result back from the function.

Libraries

Libraries allow you to collect functions that are commonly used in multiple scripts into their own scripts. These scripts are then referenced from your script using the <library> tag.

Libraries are best defined at the top of the script, right inside the script tag or at the module level. Libraries references are loaded when the script is loaded.

```
<library name="Utility" file="Library"/>
```

The "name" attribute specifies how the library is referenced from within your script. This can be any name. The "file" attribute refers to the name of the file holding the library functions. In the case of RMCS, the "file" is the name of the script holding the functions.

Here is how you might call a function in the library that is defined above:

```
<expr>
  Utility#MyFunction($somevalue);
</expr>
```

In RMCS, there is a library called "Library" with functions for making connections. There is also a library for specializing scripts for particular appliances called "Specialization".

Embedded Expressions

Embedded expressions are templates that let you insert expressions that evaluate to values in places. While you may not use this often, there are sometimes cases where this can come in handy.

An embedded expression uses {{ and }} to wrap the expression. These expressions are evaluated before other expressions.

Refer to the CUEScript Developer's Guide and look for the codes next to attributes as to whether embedded expressions are allowed for that attribute.

CLEO CONFIDENTIAL. ©2008 CLEO Communications. All rights reserved. Cleo is a registered trademark of CLEO Communications

Exception Handling

Exceptions are a way of handling errors in the system. You can recover from these errors to perform some alternate action.

```
<module xmlns="http://www.cuesoft.com/CUEScript/1.0">
		<script>
				<try>
						<expr>
								$foo := $bar;
								$output.WriteLine( "Success!" );
						</expr>
				</try>
				<catch>
						<expr>
								$output.WriteLine( $exception );
						</expr>
				</catch>
		</script>
</module>
```

In this first example, $bar is not defined so it will throw an exception. The $exception variable is set to the exception.

```
<module xmlns="http://www.cuesoft.com/CUEScript/1.0">
 <script>
  <try>
   <expr>
    $foo := 11
   </expr>
   <if test="$foo &lt; 10">
    <raise name="LessThanException" explanation="Foo less than 10"/>
   </if>
   <if test="$foo &gt; 10">
    <raise name="GreaterThanException" explanation="Foo greater than 10"/>
   </if>
  </try>
  <catch name="LessThanException">
   <expr>
    $output.WriteLine( "Less Than: " ^ $exception );
   </expr>
  </catch>
  <catch name="GreaterThanException">
   <expr>
    $output.WriteLine( "Greater Than: " ^ $exception );
   </expr>
  </catch>
 </script>
</module>
```

This second example demonstrates using internally named exceptions and the use of the <raise> tag to raise an exception in the code.

```
<module xmlns="http://www.cuesoft.com/CUEScript/1.0">
```

CLEO CONFIDENTIAL. ©2008 CLEO Communications. All rights reserved. Cleo is a registered trademark of CLEO Communications

```
  <script>
    <try>
      <if test="not($system.Invoke('System.IO.File', 'Exists', 'test.dat'))">
        <raise expr="$system.CreateNetObject('System.IO.IOException', 'File not there')"/>
      </if>
    </try>
    <catch>
      <expr>
        $output.WriteLine( $exception.Message() );
      </expr>
      <reraise/>
    </catch>
  </script>
</module>
```

This example shows raising a .NET exception object. Normally the exception stops in a <catch> block. You can re-raise the exception by calling <reraise/>. This will cause the exception to be caught in a parent <try><catch> block. If a parent block does not exist, the script will end with an exception state.

You can follow the last catch block in a series with a <finally> block for script that you want to run regardless of whether an exception occurs or not.

```
<try>
…
</try>
<catch>
…
</catch>
<finally>
  <expr>
    $log.Info( "end of procedure" );
  </expr>
</finally>
```

The Communication Server recognizes a special exception called the FatalException. When a Communication Server script exits with an Exception the normal retry rules apply. When it exits with a Fatal Exception, the Job Server will not retry that instance.

Here is an example of how to raise this type of exception:

```
<raise expr='new( "Cleo.Rmcs.CommunicationServer.FatalException" )'/>
```

CLEO CONFIDENTIAL. ©2008 CLEO Communications. All rights reserved. Cleo is a registered trademark of CLEO Communications

CUEScript Internal Objects

The $system Object

The system object is a built-in object with several convenience members and members for handling .NET object creation and nuances.

```
$system.Sleep(5000);
```

Call the "Sleep" member to wait the specified number of milliseconds.

```
$system.Wait( 5000);
```

The "Wait" call will wait until any event is fired. We'll discuss events when we cover Communication Server scripts.

```
$system.Wait( 10000, "OnConnected" );
```

This call waits for a specific event to be fired.

```
$newObj := $system.CreateNetObject( "System.Text.StringBuilder", "foobar" );
```

Use the "CreateNetObject" member to create a new .NET object. The first argument is fully qualified class name, and additional arguments are the arguments passed to the constructor.

```
$result := $system.Invoke( "System.IO.File", "Exists", "c:\temp\file.dat");
```

Use the "Invoke" member to call static members on .NET objects. The first argument is the fully qualified class name, the 2nd argument is the method name, and additional arguments are arguments passed to the method.

```
$newDocument := $system.CreateObject( "document" );
```

You can create a new script document in script. This might be useful to programmatically generate a new script and save that script out.

CLEO CONFIDENTIAL. ©2008 CLEO Communications. All rights reserved. Cleo is a registered trademark of CLEO Communications

Testing and Debugging your Script in RMCS

Writing a script that is performing complex functionality will naturally be complex. There are multiple approaches that can be taken when building a script. I'm going to recommend some principles and approaches that I think will work well for you.

Development Principles

1) Reuse existing scripts - If there is a script that is working and can be modified to do what you want, start with that. Copy the script, rip out the non-relevant pieces and then start building in the new functionality.

2) Take baby steps - Start with a simple script. Get that working. Add a little functionality. Get that working. Add a little more functionality. Get that working, and so on. I would strongly advise against writing the whole script at once and then try to get it working.

3) Use the $log object to log steps and data values. If the script runs at all, this information will be useful in knowing what happened. More on the $log object a little later.

4) Build and use library functions - If you a common task that can be moved into a library, do that. Other scripts can then use this proven code. Note that you should get the script working as a stand-alone script before moving functionality into a library. It'll be a lot easier to trouble-shoot the script if it's all in 1 place.

Testing and Debugging Process

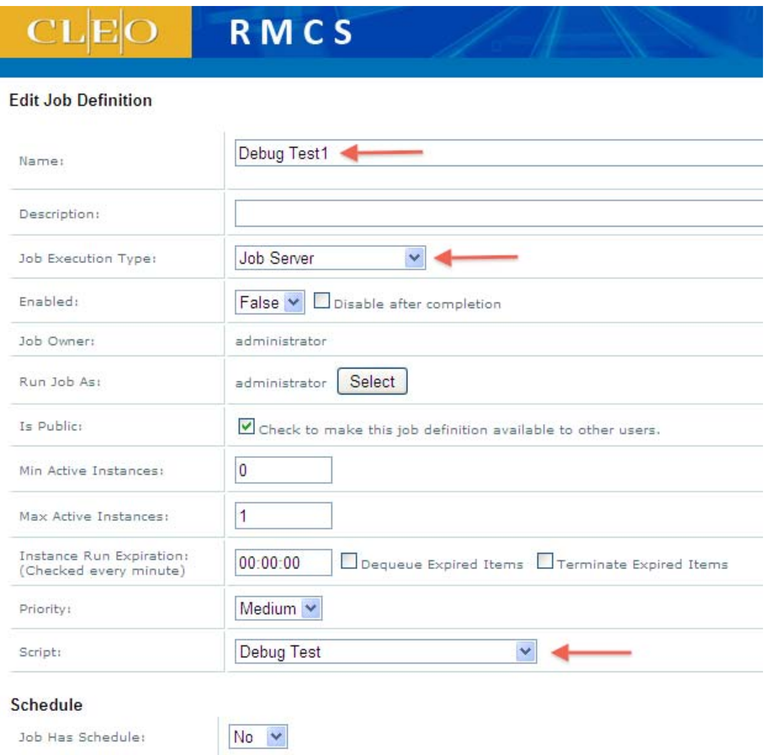
1) Whether starting from an existing script or building a new script, start by editing using CUEScript Studio or other XML editor. These editors can check for well-formedness and some simple syntax errors. (In CUEScript Studio, you can select Build | Check Script from the menu to check the script for errors)

2) If there are portions of script (or functions) that can be run independent of RMCS, test those using CUEScript Studio. CUEScript Studio will let you step line by line through your script.

3) Setup a job definition that uses your script for testing. If you're using a $connection object, you'll need to setup a single appliance to run against. If not, select Connection Type to None or run as a Job Server job.

Testing as Job Server Job

1) Setup a job definition, specifying the name, execution type, and script.

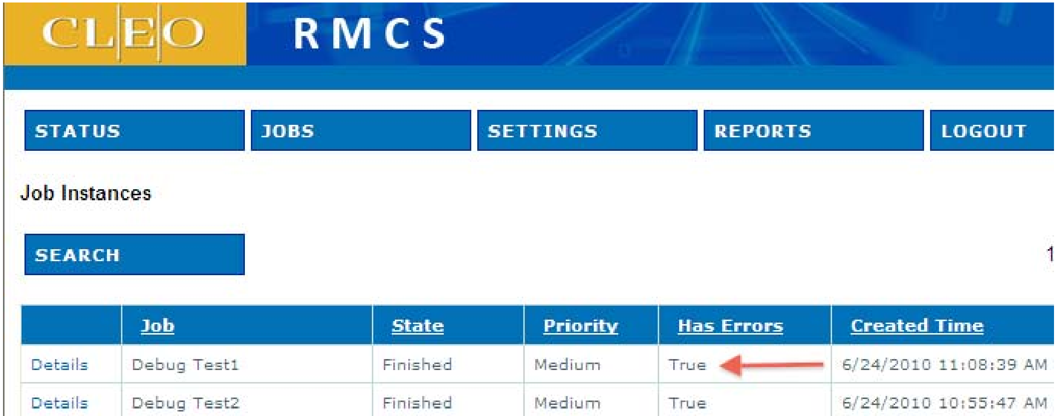


2) Run the job by clicking the Run link, and then clicking Run Now.

	Name	Enabled	Type	Has Sched.	Pend. Inst.	Last Run
Details Edit Delete Run	0 Error Job	False	Comm	True	N/A	7/24/08 08:
Details Edit Delete Run	0 Error Recovery	False	Comm	True	N/A	7/06/08 04:
Details Edit Delete Run	1 Minute Job	False	Comm	False	N/A	11/11/08 03
Details Edit Delete Run	Debug Test1	False	Job	False	N/A	6/24/10 10:
Details Edit Delete Run	Debug Test2	False	Comm	False	N/A	6/24/10 10:

CLEO CONFIDENTIAL. ©2008 CLEO Communications. All rights reserved. Cleo is a registered trademark of CLEO Communications

3) Go to the Job Instances page by selecting Jobs | Job Instances. You'll see the result of your job. If the script threw an exception, Has Errors will display as "true".



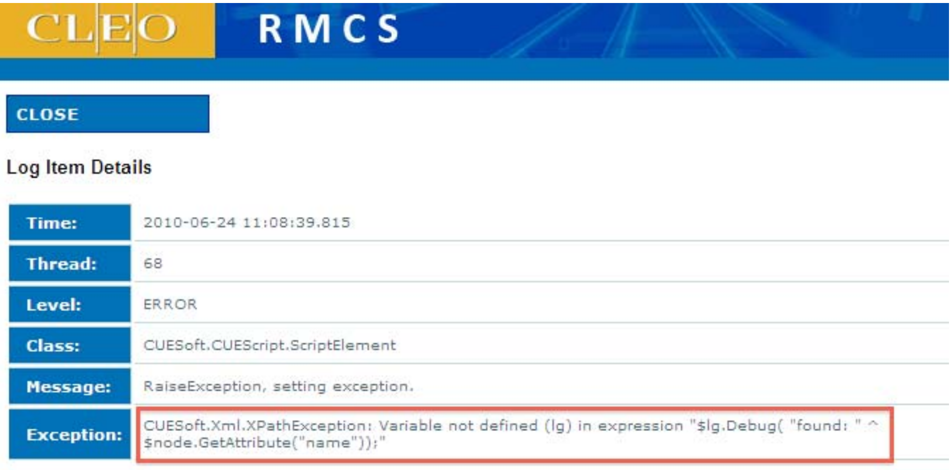
4) To view the reason for the exception, click Details, then click Log. This will show you the entire Job Server log during the time that the job ran. Note that logging items might include items from other Job Server jobs or Job Server activity. Exceptions will show up as an ERROR.

Log View of 6/24/2010 11:08:39 AM - 6/24/2010 11:08:44 AM
Job Item Instance: 3335 / Appliance:

Time	Thread	Level	Class	Message
11:08:39.503	63	DEBUG	JobServer	ProcessCommand(RmcsRunJob) started
11:08:39.503	63	DEBUG	JobDefinition	RunJob, in
11:08:39.503	63	DEBUG	JobDefinition	RunJob, inserted item into work queue, queueId: 3335, priority: Medium, jobInstItemId: 3335
11:08:39.534	63	DEBUG	JobDefinition	RunJob, out
11:08:39.581	64	DEBUG	JobServer	WakeUpCommServers, in
11:08:39.581	64	DEBUG	JobServer	SendCommand, sending command, address: vmxpdev2:7070, cmd: RmcsWakeup
11:08:39.581	64	DEBUG	JobServer	SendCommand, sending web request: http://vmxpdev2:7070/RmcsWakeup?hostAddress=VMXPDEV2&hostPort=7000
11:08:39.597	66	DEBUG	JobServer	RunJobServerJobs, in
11:08:39.722	64	DEBUG	JobServer	SendCommand, web response: <rmcs />
11:08:39.737	67	DEBUG	JobServer	ProcessCommand(RmcsGetWorkItem) started
11:08:39.737	66	DEBUG	JobServer	RunJobServerJobs, out
11:08:39.737	68	DEBUG	JobServer	ProcessCommand(RmcsUpdateWorkItemStatus) started
11:08:39.815	68	DEBUG	JobWorkItem	Execute(main), running for job: Debug Test1, instance: 3111
11:08:39.815	68	DEBUG	ScriptElement	Execute, cmd (script) in
11:08:39.815	68	DEBUG	ScriptElement	Execute, cmd (expr) in
11:08:39.815	68	DEBUG	ScriptElement	Execute, expr: $lg.Debug("found: " ^ $node.GetAttribute("name"));
11:08:39.815	68	ERROR	ScriptElement	RaiseException, setting exception.
11:08:39.815	68	DEBUG	ScriptElement	Execute, cmd (script) out

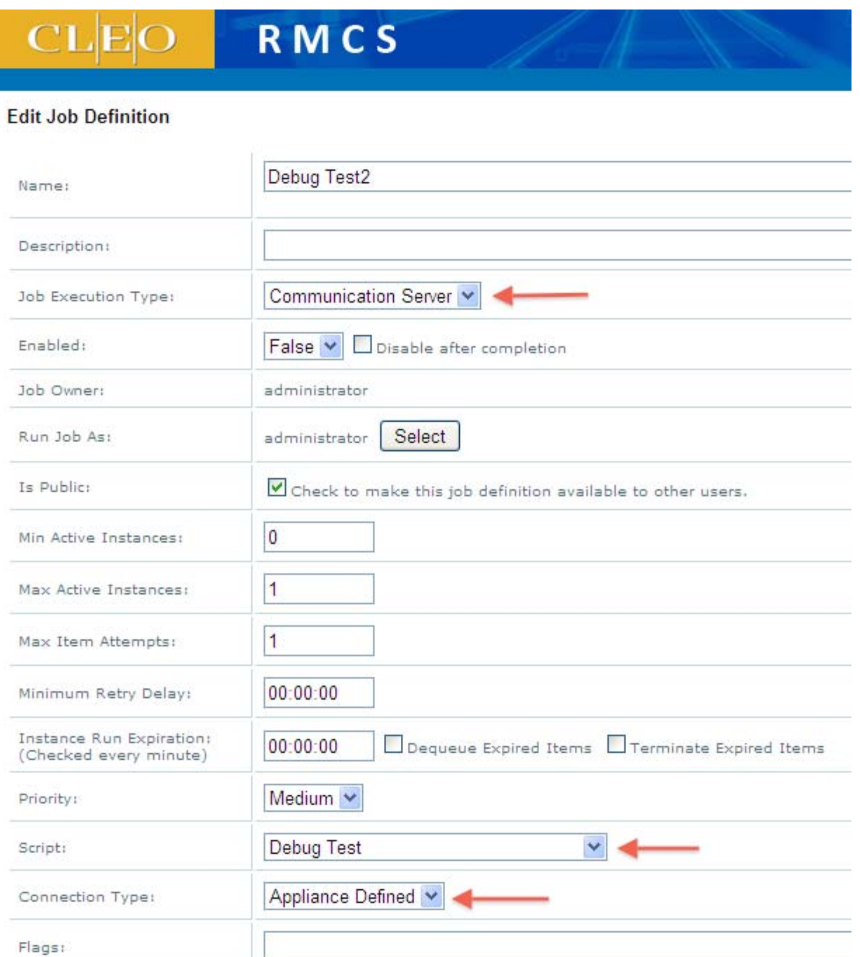
5) Clicking on the link will display the exception details. From this you can go back to your script to identify the problem.

CLEO CONFIDENTIAL. ©2008 CLEO Communications. All rights reserved. Cleo is a registered trademark of CLEO Communications



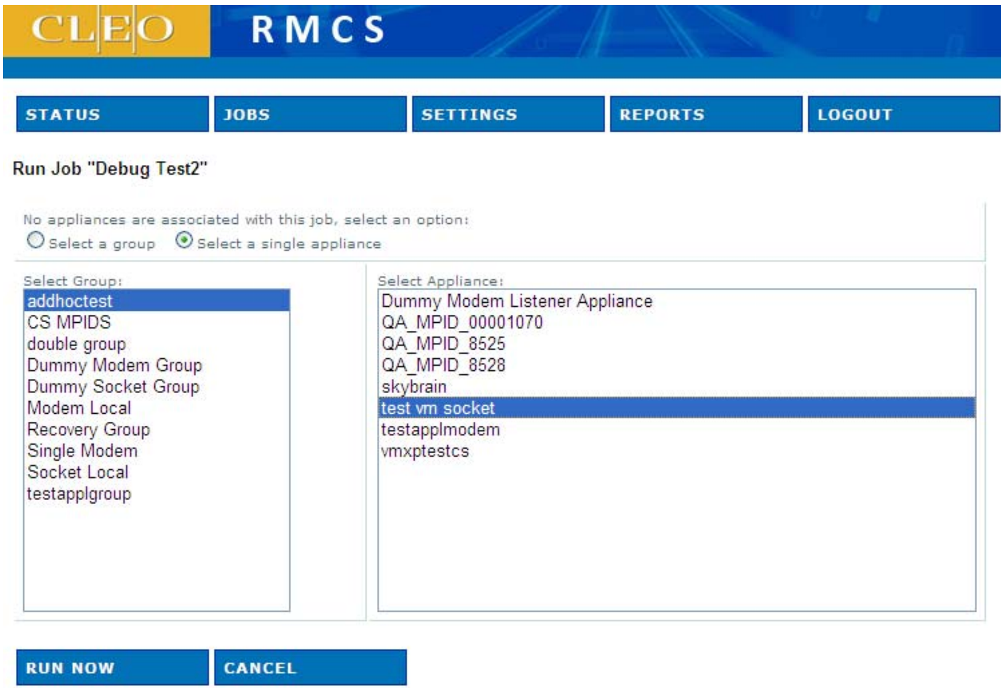
Testing as a Communication Server Job

1) Setup the Job Definition, specify Name, Communication Server as the Type, select your script, and select the Connection Type to "None" or "Appliance Defined" depending on whether you need a connection.

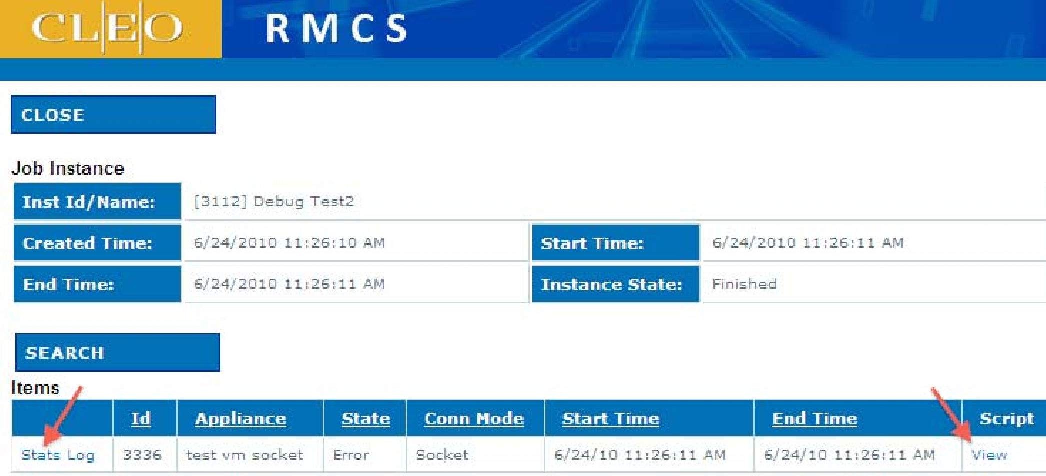


If you select "Appliance Defined", you can select an Appliance Group to run against (I'd suggest a group that has only 1 appliance in it during your testing.) or you can not specify an Appliance Group and you'll be prompted to specify a group or appliance when you run the job.

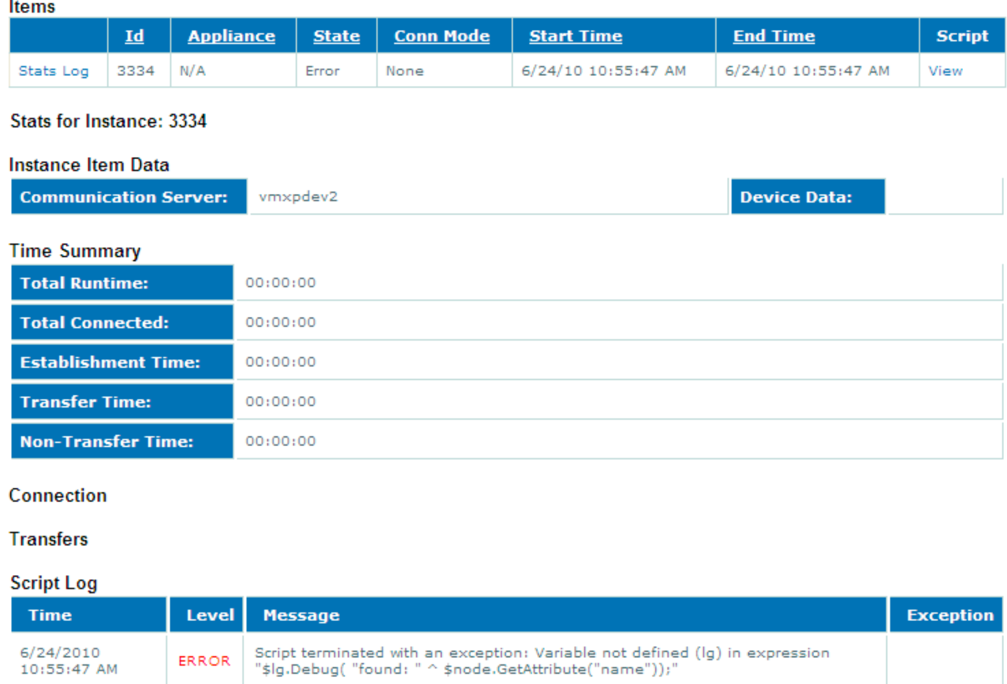
2) From the Definitions form, click Run to run the job, select the appliance and click Run Now.



3) Follow the same step 3 as testing for a Job Server Job. When you click details, you can click "Stats" to view the exception returned from the Communication Server. This will have your logging and errors in it. You can also click "View" to view the specialized script that ran for this job.



CLEO CONFIDENTIAL. ©2008 CLEO Communications. All rights reserved. Cleo is a registered trademark of CLEO Communications



Items

	Id	Appliance	State	Conn Mode	Start Time	End Time	Script
Stats Log	3334	N/A	Error	None	6/24/10 10:55:47 AM	6/24/10 10:55:47 AM	View

Stats for Instance: 3334

Instance Item Data

Communication Server:	vmxpdev2	Device Data:	

Time Summary

Total Runtime:	00:00:00
Total Connected:	00:00:00
Establishment Time:	00:00:00
Transfer Time:	00:00:00
Non-Transfer Time:	00:00:00

Connection

Transfers

Script Log

Time	Level	Message	Exception
6/24/2010 10:55:47 AM	ERROR	Script terminated with an exception: Variable not defined (lg) in expression "$lg.Debug("found: " ^ $node.GetAttribute("name"));"	

Testing on the Remote Server

Testing scripts on the Remote Server is a little more complicated. The only way view errors to look at the Remote Server logs. The best way to test for a Remote Server is to install a local socket-based version of the Remote Server on your local machine and use that as the appliance for testing.

You can then place the signed script directly in the scripts directory and have easy access to the logs.

Use the runscript.exe tool to manually run your script.

CLEO CONFIDENTIAL. ©2008 CLEO Communications. All rights reserved. Cleo is a registered trademark of CLEO Communications

Specialization

The best way to understand specialization is to walk through the script. Below is the interactive script.

```
<?xml version="1.0" encoding="utf-8" ?>
<module xmlns="http://www.cuesoft.com/CUEScript/1.0">
 <!-- **** Specialization - Start **** -->
 <script name="special" >
  <library name="Specialization" file="Specialization"/>
  <expr>
   Specialization#SpecializeScript();
   $scriptNode := tonode(//script[@name='special']);
   $scriptNode.ParentNode.RemoveChild( $scriptNode );
   $output.Write( $document.OuterXml );
  </expr>
 </script>
 <!-- **** Specialization - End **** -->

 <!-- **** Runtime - Start **** -->
 <script name="main">
  <library name="Utility" file="Library"/>

  <var name="initialization"/>
  <var name="location"/>
  <var name="port"/>

  <!-- Main script area -->
  <expr>$log.Debug('Interactive session script start.')</expr>
  <var name="quitEvent" expr="$system.CreateNetObject( 'System.Threading.ManualResetEvent', false() )"  scope="global"/>
  <try>
   <expr>
    $connection.Configure( "location=" ^ $location ^ ";port=" ^ $port ^ ";modemcodes=dialogic");
    $connection.Open();
    Utility#ConnectToRemote( false() );
    $log.Info( "mpid=" ^ $valueCache.GetValue( "mpid" ) );
    $command.WriteClientResponse( "Connected!" );
    $quitEvent.WaitOne();
   </expr>
  </try>
  <catch>
   <expr>
    $log.Error( 'Script exception.', $exception );
   </expr>
   <reraise/>
  </catch>
  <expr>$log.Debug('Interactive session script end.')</expr>

  <!-- Quit command event -->
  <event name="OnQuitScript">
   <expr>$quitEvent.Set()</expr>
  </event>

  <!-- OnDisconnect - connection was lost -->
  <event name="OnDisconnected">
```

CLEO CONFIDENTIAL. ©2008 CLEO Communications. All rights reserved. Cleo is a registered trademark of CLEO Communications

```
    <expr>$quitEvent.Set()</expr>
  </event>

  <!-- OnAuthenticationSuccess - display appl and register code for appliance -->
  <event name="OnAuthenticationSuccess">
    <expr>
      $command.WriteClientResponse( "Appliance Id: " ^ $security.ApplianceId() );
      $command.WriteClientResponse( "Appliance Reg Code: " ^ $security.ApplianceRegistrationCode() );
    </expr>
  </event>
 </script>
 <!-- **** Runtime - End **** -->

</module>
```

During specialization, a script section called "special" is located. If the script section does NOT exist, the entire script is returned as the script to be processed by the Communications Server. If a script section called "special" DOES exist, then the output (either via the $output object or the <output> tag) of that section is the script to be used by the Communications Server.

Here is the specialization logic for this script from the Specialization library.

```
<function name="SpecializeScript" xmlns="http://www.cuesoft.com/CUEScript/1.0">
 <var name="node" expr="tonode(//script[@name='main']/var[@name='initialization'])"/>
 <if test="not(isnull($node))">
  <expr>$node.SetAttribute( "expr", "'" ^ $appliance.Initialization ^ "'")</expr>
 </if>

 <var name="node" expr="tonode(//script[@name='main']/var[@name='kiosk'])"/>
 <if test="not(isnull($node))">
  <expr>$node.SetAttribute( "expr", "'" ^ $appliance.Name ^ "'" )</expr>
 </if>

 <var name="node" expr="tonode(//script[@name='main']/var[@name='location'])"/>
 <if test="not(isnull($node))">
  <expr>$node.SetAttribute( "expr", "'" ^ $appliance.Location ^ "'" )</expr>
 </if>

 <var name="node" expr="tonode(//script[@name='main']/var[@name='port'])"/>
 <if test="not(isnull($node))">
  <expr>$node.SetAttribute( "expr", "'" ^ string($appliance.Port) ^ "'")</expr>
 </if>
</function>
```

Objects available during specialization:

$scriptContext

$log

$appliance

$database

$refDatabase

 ©2008 CLEO Communications. All rights reserved. Cleo is a registered trademark of CLEO Communications

Post Job Scripting

For Communication Server scripts, you can specify a "post" script section that will be run when the job is completed. This script section is useful for updating database fields and/or rescheduling the job programmatically.

The special objects available during the post job script are:

$scriptContext (set to "jobServer")

$database

$refDatabase

$jobServer

$jobInstance

$log (reference to the Job Server's logging instance)

```
<script name='post'>
  <if test='$jobInstance.HasErrors()'>
    <!-- do something here -->

  </if>
</script>
```

CLEO CONFIDENTIAL. ©2008 CLEO Communications. All rights reserved. Cleo is a registered trademark of CLEO Communications

Communication Server Scripts

The Communication Server is where the bulk of your scripts are going to run. There are several objects that are exposed to these scripts.

These objects include:

Object	Description
$scriptContext	Set to the string "commServer".
$security	Of class Cleo.Rmcs.CommunicationServer.Security. This object holds data pertaining to appliance registration and is used to register an appliance. $security.JobApplianceId() - the appliance ID for the job. $security.JobApplianceRegistrationCode() - the registration code for the job's appliance. $security.ApplianceId() - the id of the connected appliance. This is known after a connection is made. $security.ApplianceRegistrationCode() - the registration code for the connected appliance. $security.HasRegistrationData() - returns true if the registration data is loaded. $security.CanRegister() - returns true if registration can occur. The company code cannot be empty in order to register. $security.IsRegistered() - returns true if registration has occurred in the past. $security.LoadRegistrationData() - load the registration data for this appliance. $security.Register() - registers the appliance by saving the value in the database and for the kiosk. See the section on appliance registration below.
$log	This is an instance of the log4net logging object. Items logged to this object are stored with the instance data when the job instance completes. Relevent methods for this call are: $log.Info("some data"); $log.Warning("some data"); $log.Error("some data"); $log.Debug("some data");
$connection	The connection object is created for the type of connection that is specified for the appliance. This will be either a modem connection or socket connection. See the section on Working with Connections below.
$command	This object is used to send commands to the remote appliance. See the section on Sending Commands to the Appliance for more info.

CLEO CONFIDENTIAL. ©2008 CLEO Communications. All rights reserved. Cleo is a registered trademark of CLEO Communications

$session	The session object can be used to get information about the session, setup timers, setup a listening connection and communicate with the Job Server.
$local	The local object is useful for performing some local file operations and working with INI files. See class Cleo.Rmcs.Server.Core.Local.
$fileTransfer	A reference to the file transfer processor. Use this object to transfer files between end points.
$attempt	This is the attempt to run this job instance. The first time, this value is 1.
$packetFactory	This object can be used to control the underlying protocol by setting error thresholds and maximum packet sizes. Generally there is no need to ever change these settings, but in the case of some troublesome kiosks, you may be able to manipulate these settings to increase your chance of success.

Working with Connections

The connection object is used to connect with an appliance.

Initializing a connection

Before you can connect to an appliance, you need to initialize the connection object with the settings it needs. This is done by calling Configure on the connection with the setting information it needs.

```
$connection.Configure( "location=" ^ $location ^ ";port=" ^ $port ^ ";modemcodes=dialogic");
```

Refer to the RMCS Developer's Guide for a list of settings and possible values.

Connecting

To actually connect, you call the $connection.Connect() member. The following function taken from the library demonstrates implementing retry functionality and handling of connect result codes.

```
<!-- ConnectToRemote( device, processo, retries = 3 ) -->
<!-- Attempt to connect to the remote server for the specified number of retries -->
<function name="ConnectToRemote" >
  <parameter name="shouldFail" default="true()"/>
  <parameter name="retries" default="3"/>

  <for init="$attempt := 0" test="$attempt &lt; $retries" step="$attempt := $attempt + 1">
  <switch test="$connection.Connect()">

    <case value="0">
      <!-- Return normally.  This is the only normal exit without throwing an exception. -->
      <!-- Calls the registration check function.  It might throw. -->
      <return expr="CheckRegistration( $shouldFail )"/>
    </case>

    <case value="-1">
      <raise code="-1" explanation="Connect fail!"/>
    </case>

    <case value="-2">
```

```
    <expr>
      $command.WriteClientResponse( "Authentication failed.");
    </expr>
    <raise code="-2" explanation="Authentication failed!"/>
  </case>

  <case value="1">
    <expr>
      $log.Info( "Retrying connect." );
    </expr>
  </case>

 </switch>
 </for>

 <!-- This function only returns normally when it connects.  It always throws on failure. -->
 <raise explanation="Connect fail, retries exceeded!"/>
</function>
```

There is no need to explicitly disconnect unless you plan on doing some processing after you finish with the connection. The connection is automatically disconnected when the script ends.

To explicitly disconnect, call:

```
$connection.Disconnect();
```

Creating a New Connection

The $connection object is not meant to be reused. If you want to create a new connection within the same script instance, you can call $session.GetNewConnection() to get a new connection object. This object will need to be initialized as before.

Setup a Listening Connection

The Communication Server can be setup to receive connections as well. It only needs to be told to wait for a connection. This is done by passing the $connection object to the $session.RequestListener call.

When a connection is established, the OnListenerConnect event is signaled.

Here is a sample listener script:

```
<script xmlns="http://www.cuesoft.com/CUEScript/1.0">
	<expr>$log.Debug('Listening session script start.')</expr>

	<var name="quitEvent" expr="$system.CreateNetObject( 'System.Threading.ManualResetEvent', false() )"
scope="global"/>
	<try>
		<expr>
			<!-- Subtract some destination time from now, and get ticks -->
			$session.AddTimer( "quittime", datetime( "15:00") );

			<!-- Make request for listening resource -->
			$session.RequestListener( $connection );

			<!-- Wait for connection, quit, or timeout -->
```

```
                        $quitEvent.WaitOne();

                        $session.ReleaseListener();
                </expr>
        </try>
        <catch>
                <expr>
                        $log.Error( 'Script exception.', $exception );
                        $session.ReleaseListener();

                </expr>
                <reraise/>
        </catch>
        <expr>$log.Debug('Listening session script end.')</expr>

        <!-- Quit command event -->
        <event name="OnQuitScript">
                <expr>$quitEvent.Set()</expr>
        </event>

        <!-- Timer command event -->
        <event name="OnTimer">
                <parameter name="sender"/>
                <parameter name="e"/>
                <expr>
                        $log.Debug( 'Got timer event: ' ^ $e.Name() );
                        $quitEvent.Set()
                </expr>
        </event>

        <!-- OnListenerConnect
                  ** We get a connection object back in case we need it
                -->
        <event name="OnListenerConnect">
                <expr>$log.Debug( 'Got listener connection.' );</expr>
        </event>

        <!-- OnSessionFinished - other side is done -->
        <event name="OnSessionFinished">
                <expr>
                        $log.Debug( 'Got session finished.' );
                        $quitEvent.Set()
                </expr>
        </event>

        <!-- OnDisconnect - connection was lost -->
        <event name="OnDisconnected">
                <expr>
                        $log.Debug( 'Got disconnect.' );
                        $quitEvent.Set()
                </expr>
        </event>

        <event name="OnCommand">
                <expr>$log.Debug('Got command ' ^ $eventArgs.Command )</expr>
        </event>

</script>
```

CLEO CONFIDENTIAL. ©2008 CLEO Communications. All rights reserved. Cleo is a registered trademark of CLEO Communications

Appliance Registration

Here is the registration function from the library. This is called by the connection logic when a successful connection is made.

```
<!-- CheckRegistration - Throws if registration fails.
1) If job appliance id is empty and kiosk appliance id is not empty then we need to get job appliance id.
2) If job appliance id is not empty and kiosk appliance id is not empty, if no match, disconnect
3) If job appliance id is not empty and kiosk appliance id is empty, register and push appliance id.
4) If appliance ids match and register codes don't match, reregister.
-->
<function name="CheckRegistration">
	<parameter name="shouldFail" default="true()"/>

	<if test="not($security.CanRegister)">
		<return/>
	</if>

	<if test="not($security.HasRegistrationData)">
		<if test="not($security.LoadRegistrationData)">
			<raise explanation="Unknown caller tried to connect."/>
		</if>
	</if>

	<if test="not(isnullorempty($security.ApplianceId)) and $security.JobApplianceId != $security.ApplianceId">
		<var name="msg">'Mismatched appliance ids, expected {{$security.JobApplianceId}}, has
{{$security.ApplianceId}}.'</var>
		<expr>
			$log.Error(		$msg );
		</expr>
		<if test="$shouldFail">
			<raise explanation="{{$msg}}"/>
		</if>
		<expr>
			$command.WriteClientResponse( $msg );
		</expr>
		<return/>
	</if>
	<if test="isnullorempty($security.ApplianceId) or $security.JobApplianceRegistrationCode !=
$security.ApplianceRegistrationCode">
		<expr>
			$security.Register();
			$msg := format( "Registered appliance '{0}' with code '{1}'.", $security.ApplianceId,
$security.ApplianceRegistrationCode );
			$log.Info( $msg );
			$command.WriteClientResponse( $msg );
		</expr>
	</if>
</function>
```

The goal of the registration process is to assure that the appliance we're communicating with is the one we're expecting to be communicating with. If the appliance ids match, it's the correct appliance. The registration codes are a good way of tracking when the appliance was first connected to.

CLEO CONFIDENTIAL. ©2008 CLEO Communications. All rights reserved. Cleo is a registered trademark of CLEO Communications

Sending Commands to the Appliance

The $command object is used to send commands to the remote appliance. These can be any of the remote commands specified in the RMCS Developer's Guide. There are two ways of sending commands: posting and sending.

Posting a command will send a command and immediately return without waiting for a response.

For example:

```
$command.Post( "rdel c:\temp\*.dat" );
```

Sending a command will send the command and wait for a response. Now it won't wait indefinitely. The default timeout for waiting for a response is 30 seconds. If the command times out waiting for a response a TimeoutException is thrown. You can change the timeout by setting $command.Timeout to the number of milliseconds to wait or by passing the timeout in milliseconds as the second parameter to the Send method.

```
$result := $command.Send( "rdir c:\pasta", 20000 );
```

The resulting object is an instance of SendCommandResponse. This object has several methods that can be used to access result values.

```
$result.Success()
```

is a boolean result of the success of the call.

```
$result.ErrorMessage()
```

is an error message if Success is false.

```
$result.DisplayText()
```

is the display text returned by the command (as you would see in the Terminal).

```
$result.Response()
```

is an XmlDocument containing the XML data returned by the command.

RMCS Events

Maintenance scripts that run on the Job Server, Terminal, or even Remove Server will run, perform some task, and then exit. However, scripts running on the Communication Server are often waiting for tasks to complete. Scripts can respond to events.

In CUEScript, an event is defined using the <event> tag.

```
<!-- Quit command event -->
<event name="OnQuitScript">
```

CLEO CONFIDENTIAL. ©2008 CLEO Communications. All rights reserved. Cleo is a registered trademark of CLEO Communications

```
        <expr>$quitEvent.Set()</expr>
</event>
```

A full list of events are found in the RMCS Developer's Guide. Because of the event model in the Communication Server, you will need to use .NET signals to stop and wait for events to occur. If you don't then scripts will exit before tasks can finish and events are received.

You can create a signal using the following call:

```
<var name="quitEvent" expr="$system.CreateNetObject( 'System.Threading.ManualResetEvent', false() )"  scope="global"/>
```

After you kick off tasks that will signal events (i.e. a file transfer that will signal when the transfer is complete), you can cause the script to wait using the following call:

```
$quitEvent.WaitOne();
```

In the event, call $quitEvent.Set() to set the signal. This will cause the WaitOne to statement to exit and continue.

Here is a portion of a script that transfers files and uses signals:

```
<var name="startEvent" expr="$system.CreateNetObject( 'System.Threading.ManualResetEvent', false() )"  scope="global"/>
<var name="quitEvent" expr="$system.CreateNetObject( 'System.Threading.ManualResetEvent', false() )"  scope="global"/>
<var name="successfulTransfer" expr="true()"  scope="global"/>
<expr>$log.Info('Get kiosk files script start.')</expr>
<try>
        <expr>
                $connection.Configure( "location=" ^ $location ^ ";port=" ^ $port);
                $connection.Open();
                connect( 2 );
                $log.Info( "Connected!" );
                $filesRemaining := 0;
                $countTries := 4;
        </expr>
        <while test="$countTries > 0">
                <try>
                        <expr>
                                $filesRemaining := $fileTransfer.GetRemoteFileCount($kioskRoot ^ '\*.*', 10000);
                                $countTries := 0;
                        </expr>
                </try>
                <catch>
                        <expr>
                                $countTries := $countTries -1;
                                $log.Warn( "Timeout trying to get count." );
                        </expr>
                        <if test="$countTries = 0">
                                <raise explanation="Failed to get files count."/>
                        </if>
                </catch>
        </while>
        <if test="$filesRemaining > 0">
                <expr>
                        <!-- Make sure that the destination directory exists -->
                        $session.CreateDirectory( $localRoot ^ $kiosk );

                        <!-- Transfer the file -->
```

```
				$fileTransfer.GetFiles( $kioskRoot ^ "\*.*", $localRoot ^ $kiosk );
			</expr>

			<expr>
				<!-- Wait for transfers to quit or some other quit operation -->
				$quitEvent.WaitOne();
			</expr>
			<if test="$successfulTransfer = false()">
				<!-- Raise exception -->
				<raise explanation="One or more files failed to transfer."/>
			</if>
			<else>
				<expr>$log.Info('Transfer succeeded.')</expr>
			</else>
		</if>
		<else>
			<expr>$log.Info('No files to transfer.')</expr>
		</else>
</try>
<catch>
		<expr>
			$log.Error( 'Script exception.', $exception );
		</expr>
		<reraise/>
</catch>
<expr>$log.Info('Get kiosk files script end.')</expr>

<!-- Quit command event -->
<event name="OnQuitScript">
		<expr>
			$successfulTransfer := false();
			$quitEvent.Set()
		</expr>
</event>

<!-- OnDisconnect - connection was lost -->
<event name="OnDisconnect">
		<expr>
			$successfulTransfer := false();
			$quitEvent.Set()
		</expr>
</event>

<!-- OnTransfersStopped - called when all transfers are completed -->
<event name="OnTransfersFinished">
		<if test="$filesRemaining &lt;= 0">
			<expr>
				$quitEvent.Set();
			</expr>
		</if>
</event>

<!-- OnTransferSuccess - called when an individual transfer has succeeded -->
<event name="OnTransferSuccess">
		<parameter name="sender"/>
		<parameter name="e"/>
		<expr>
			$log.Info( "Transfered file: " ^ $e.FilePath() );
			$filesRemaining := $filesRemaining - 1;
```

```
        </expr>
</event>

<!-- OnTransferError - called when an individual transfer fails -->
<event name="OnTransferError">
        <parameter name="sender"/>
        <parameter name="e"/>
        <expr>
                <!-- Signal a failed job so it will reschedule -->
                $log.Error( "Failed to transfer file: " ^ $e.FilePath() );
                $successfulTransfer := false();
                $filesRemaining := $filesRemaining - 1;
        </expr>
</event>
```

Like functions, events have parameter arguments. The first argument is the caller. You generally won't care about this. The second argument ($1) may have parameters that are relevant. This will depend on the event.

For example, the OnTransferSuccess second argument is a TransferStatsEventArgs object, which has several fields with information about the transfer including rate, time, status, file name, and bytes transferred.

Using the Session Object

The $session object performs tasks that are useful for Communication Server scripts.

```
$session.JobName()
```

returns the name of the job currently running.

```
$session.JobInstanceItemId()
```

returns the unique number of the job instance item running.

```
$session.SignalRemoteDisconnect()
```

tells the remote appliance to drop its connection. This can make for a cleaner break with socket connections.

Using Timers

Timers are useful for getting an event in a specified amount of time. For example, if a task hasn't completed in certain amount of time, you can decide to take some alternate action.

To add a timer, call the $session.AddTimer method.

```
$session.AddTimer( "mytimer", 300000 );
```

will create a timer called "mytimer" that will signal in 300000 milliseconds (5 minutes).

```
$session.AddTimer( "myothertimer", datetime( "20:00") );
```

will create a timer that will signal at 8:00pm.

When a timer comes due, it will signal an event.

```
<!-- Timer command event -->
<event name="OnTimer">
	<parameter name="sender"/>
	<parameter name="e"/>
	<expr>
		$log.Debug( 'Got timer event: ' ^ $e.Name() );
		$quitEvent.Set()
	</expr>
</event>
```

The name of the timer is available from the 2nd parameter. To remove a timer, call $session.RemoveTimer("mytimer") and pass in the name of the timer to remove. These timers will signal only once. To create a periodic timer, add a new timer in the OnTimer event.

Communicating with the Job Server

The Job Server is capable of performing several tasks through its HTTP command interface. These commands are enumerated in the RMCS Developer's Guide. These commands can be invoked from script using the $session. JobServerCommand method.

```
$result := $session.JobServerCommand( "rmcsgetscript", "name=Sample" );
```

The first argument is the command name. The second argument is a list of comma separated arguments to send to the command. The optional third argument is posted data. The result is an XML document containing the result.

Transferring Files

To transfer files from the Communication Server script, use the $fileTransfer object, which is an instance of the FileTransferProcessor class.

```
$fileTransfer.TransferActive
```

returns whether a transfer is currently active (boolean).

```
$fileTransfer.ActiveTransferCount
```

returns the number of active transfers occurring.

```
$fileTransfer.SetMaxDirectionalTransfers := 2;
```

CLEO CONFIDENTIAL. ©2008 CLEO Communications. All rights reserved. Cleo is a registered trademark of CLEO Communications

controls how many simultaneous transfers can occur at once per direction. The default is 1.

```
$fileTransfer.PutFiles( "c:\temp\*.dat", "c:\remoteData" );
```

is used to transfer files to a remote appliance.

```
$fileTransfer.GetFiles( "c:\remoteData\*.dat", "c:\temp" );
```

is used to transfer files from a remote appliance to a local drive.

```
$fileTransfer.WaitForStart();
$fileTransfer.WaitForStart(5000);
```

is used to wait for a transfer to start after calling GetFiles and PutFiles.

```
$fileTransfer.WaitForComplete();
$fileTransfer.WaitForComplete(120000);
```

is used to wait for transfers to complete once they've started.

```
$fileTransfer.CancelAllTransfers();
```

is used to cancel all transfers that are occurring.

```
$fileCount := $fileTransfer.GetRemoteFileCount( "c:\temp\*.dat" );
```

will return the number of remote files matching the path.

```
$fileCount := $fileTransfer.GetLocalFileCount( "c:\temp\*.dat" );
```

will return the number of local files matching the path.

```
<while test="$countTries > 0">
 <try>
  <expr>
   $filesRemaining := $fileTransfer.GetRemoteFileCount($kioskRoot ^ '\*.*', 10000);
   $countTries := 0;
  </expr>
 </try>
 <catch>
  <expr>
   $countTries := $countTries -1;
   $log.Warn( "Timeout trying to get count." );
  </expr>
  <if test="$countTries = 0">
   <raise explanation="Failed to get files count."/>
  </if>
 </catch>
</while>
<if test="$filesRemaining > 0">
 <expr>
  <!-- Make sure that the destination directory exists -->
  $session.CreateDirectory( $localRoot ^ $kiosk );

  <!-- Transfer the file -->
  $fileTransfer.GetFiles( $kioskRoot ^ "\*.*", $localRoot ^ $kiosk );
 </expr>
```

```
  <expr>
    <!-- Wait for transfers to quit or some other quit operation -->
    $quitEvent.WaitOne();
  </expr>
  <if test="$successfulTransfer = false()">
    <!-- Raise exception -->
    <raise explanation="One or more files failed to transfer."/>
  </if>
  <else>
    <expr>$log.Info('Transfer succeeded.')</expr>
  </else>
</if>
<else>
  <expr>$log.Info('No files to transfer.')</expr>
</else>
```

CLEO CONFIDENTIAL. ©2008 CLEO Communications. All rights reserved. Cleo is a registered trademark of CLEO Communications

Job Server Scripts

Job Server scripts are good for running scheduled maintenance tasks or scheduled configuration tasks.

For example, the following script can be used to update the flags settings for communication ports to control the allocation of resources.

```
<module xmlns="http://www.cuesoft.com/CUEScript/1.0">

<!--
  This script demonstrates how to set some device flags for communication servers
  and then pushing those settings to the communication server.

  Note: this does this for each comm server and makes no distinction between
modem and socket devices.  You'll need to add additional checks to make
         this more robust.
          -->

 <!-- **** Runtime - Start **** -->
 <script name="main">
  <!-- Main script area -->
  <expr>$log.Debug('Start of CommunicationServer set flags script.')</expr>

  <!-- Create an instance of the communication server manager -->
  <var name="commMgr" expr="$system.CreateNetObject('Cleo.Rmcs.Management.CommunicationServerManager',
$database )"/>

  <!-- This is the number of poerts to set -->
  <var name="portsToSet" expr="5"/>

  <!-- This is the flags value to set -->
  <var name="flagsToSet" expr="'Interactive'"/>

  <!-- Enumerate through each comm server -->
  <foreach var="commServer" in="$commMgr.GetCommunicationServers()">

    <!-- Enumerate through each device configuration for a comm server -->
    <foreach var="device" in="$commServer.GetDevices()">
      <var name="portIndex" expr="0"/>

      <!-- Enumerate through each device port in the device configuration -->
      <foreach var="port" in="$device.GetDevicePorts()">

        <!-- If port index exceeds ports to set, then break -->
        <if test="$portIndex &gt; $portsToSet">
          <break/>
        </if>
        <expr>
          <!-- Set flags and commit changes -->
          $port.SetFlags( $flagsToSet );
          $port.Commit();
          $portIndex := $portIndex + 1;
          $log.DebugFormat( "Port [{0}] set to flags [{1}]", $port.Name, $flagsToSet );
        </expr>
```

```
      </foreach>
     </foreach>
    </foreach>

    <!-- Update the comm server configuration -->
    <expr>
     $jobServer.Execute( "RmcsReloadCommCfg", "user=administrator;pwd=rmcsadmin" );
    </expr>

    <expr>$log.Debug('End of CommunicationServer set flags script.')</expr>
  </script>
</module>
```

CLEO CONFIDENTIAL. ©2008 CLEO Communications. All rights reserved. Cleo is a registered trademark of CLEO Communications

Remote Server Scripts

Remote scripts are those scripts that run on the appliance. A remote script can perform maintenance on the remote appliance, it can initiate a connection with a Communication Server, or perform some other function.

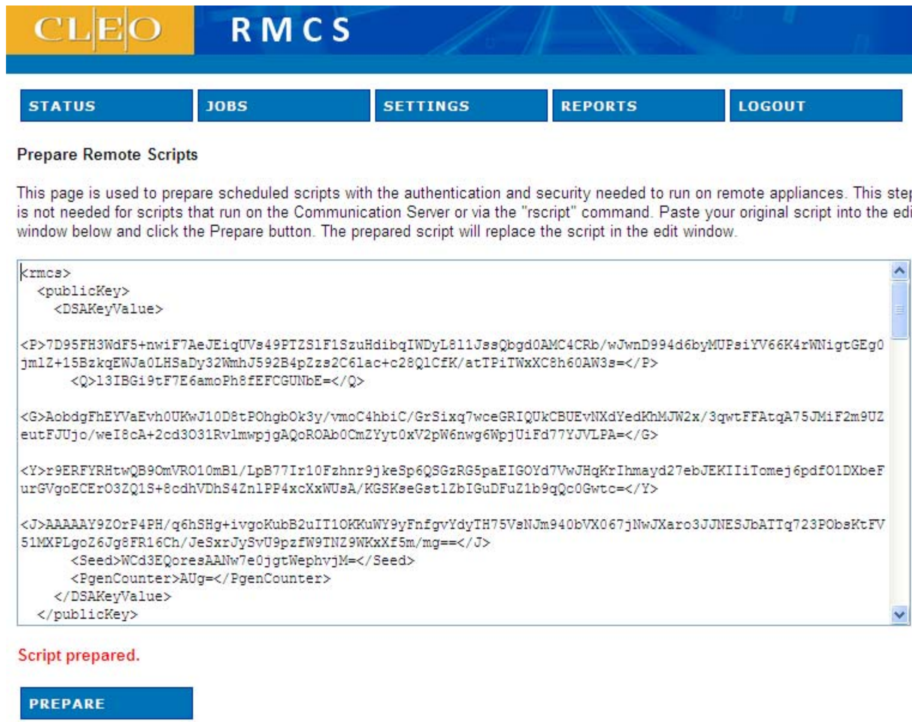
Remote scripts can be run several ways:

1) Manually - Using the "rscript" command or the "runscript.exe" executable.
2) In response to an event - A script section can be setup to run in response to the Remote Server service starting up, shutting down, or when a connection is established.
3) On a schedule - You can specify a schedule in the script to run on.

To create a remote script:

1) Edit the script as you normally would.

2) Sign the script by copying the script into the window on the "Remote Scripts" screen, click Prepare, then copy those contents to a new file.



The signed file will look like:

CLEO CONFIDENTIAL. ©2008 CLEO Communications. All rights reserved. Cleo is a registered trademark of CLEO Communications

```
<rmcs>
  <publicKey>
    <DSAKeyValue>

<P>7D95FH3WdF5+nwiF7AeJEiqUVs49PTZSlF1SzuHdibqIWDyL8l1JssQbgd0AMC4CRb/wJwnD994d6byMUPsiYV66K4rWNigtGEg
0jmlZ+15BzkqEWJa0LHSaDy32WmhJ592B4pZzs2C6lac+c28QlCfK/atTPiTWxXC8h60AW3s=</P>
      <Q>l3IBGi9tF7E6amoPh8fEFCGUNbE=</Q>

<G>AobdgFhEYVaEvh0UKwJ10D8tPOhgbOk3y/vmoC4hbiC/GrSixq7wceGRIQUkCBUEvNXdYedKhMJW2x/3qwtFFAtqA75JMiF2m
9UZeutFJUjo/weI8cA+2cd3O31RvlmwpjgAQoROAb0CmZYyt0xV2pW6nwg6WpjUiFd77YJVLPA=</G>

<Y>r9ERFYRHtwQB9OmVRO10mBl/LpB77Ir10Fzhnr9jkeSp6QSGzRG5paEIGOYd7VwJHqKrIhmayd27ebJEKIIiTomej6pdfO1DXbeFu
rGVgoECErO3ZQ1S+8cdhVDhS4ZnlPP4xcXxWUsA/KGSKseGstlZbIGuDFuZ1b9qQc0Gwtc=</Y>

<J>AAAAAY9ZOrP4PH/q6hSHg+ivgoKubB2uIT1OKKuWY9yFnfgvYdyTH75VsNJm940bVX067jNwJXaro3JJNESJbATTq723PObsKtFV
51MXPLgoZ6Jg8FR16Ch/JeSxrJySvU9pzfW9TNZ9WKxXf5m/mg==</J>
      <Seed>WCd3EQoresAANw7e0jgtWephvjM=</Seed>
      <PgenCounter>AUg=</PgenCounter>
    </DSAKeyValue>
  </publicKey>
  <schedule xmlns="http://www.cleo.com/Rmcs/Schedule/1.0">
    <!-- Tells the scheduler what section to run on startup -->
    <onstartup run="startup"/>
    <!-- Tells the scheduler what section to run on shutdown -->
    <onshutdown run="shutdown"/>
  </schedule>
  <cs:module xmlns:cs="http://www.cuesoft.com/CUEScript/1.0">
    <certificate type="Administrator" userId="administrator" loginId="1000" expires="6/29/2010 7:54:04 AM"
xmlns="http://www.cleo.com/Rmcs/Certificate/1.0">
      <groups/>
    </certificate>
    <tokens/>
    <cs:script name="startup">
      <cs:expr>
                                    $log.Debug( 'Startup script1 start' )
                        </cs:expr>
      <cs:try>
        <!-- Check if the kiosk is already registered -->
        <cs:if test="isnullorempty($appliance.RegistrationCode)">
          <cs:expr>
            <!-- Connect to the server -->
                                                            $connection.Open();
                                                </cs:expr>
          <cs:switch test="$connection.Connect()">
            <cs:case value="0">
              <cs:expr>
                <!--
                  Send 'register' command.
Commands sent from the remote script have to be either implemented as a "spider" command type in a .NET assembly or they
can be processed by the OnCommand event handler in the listener script.  This allows any command to be sent and processed
by the listener script.
The $eventResult should be set to the command result (as XML) to be returned to the remote script.

In this example, the listener script should process the register command to actually register the kiosk.
                                                                                  -->
                                    $system.Sleep( 30000 );
                                    $command.Send( "register" );
                                    $log.Info( 'Kiosk registered.' );
                        </cs:expr>
```

CLEO CONFIDENTIAL. ©2008 CLEO Communications. All rights reserved. Cleo is a registered trademark of CLEO Communications

```
        </cs:case>
        <cs:case value="-1">
          <cs:raise code="-1" explanation="Connect fail!"/>
        </cs:case>
        <cs:case value="-2">
          <cs:raise code="-2" explanation="Authentication failed!"/>
        </cs:case>
      </cs:switch>
    </cs:if>
    <cs:else>
      <cs:expr>$log.Info( 'Already registered.' )</cs:expr>
    </cs:else>
  </cs:try>
  <cs:catch>
    <cs:expr>
                                    $log.Error( 'Connection exception', $exception )
                            </cs:expr>
  </cs:catch>
  <cs:expr>
                            $log.Debug( 'Startup script1 end' )
                    </cs:expr>
 </cs:script>
 <cs:script name="shutdown">
  <cs:expr>
                            $log.Debug( 'Shutdown script1 start' )
                    </cs:expr>
  <cs:expr>
                            $log.Debug( 'Shutdown script1 end' )
                    </cs:expr>
 </cs:script>
 <Signature xmlns="http://www.w3.org/2000/09/xmldsig#">
  <SignedInfo>
   <CanonicalizationMethod Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-20010315"/>
   <SignatureMethod Algorithm="http://www.w3.org/2000/09/xmldsig#dsa-sha1"/>
   <Reference URI="">
    <Transforms>
     <Transform Algorithm="http://www.w3.org/2000/09/xmldsig#enveloped-signature"/>
    </Transforms>
    <DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
    <DigestValue>mbSW25nF9Syuw8pg8r6hy5+drRk=</DigestValue>
   </Reference>
  </SignedInfo>
  <SignatureValue>HnDp0fpJs53SB5b7vQduPVEuDJtCd4akZ1j//8ZYqdVDFZHW9MOUTA==</SignatureValue>
 </Signature>
</cs:module>
</rmcs>
```

3) Using the Terminal or a file transfer script, send the script file to the destination folder "**SCRIPT**". When the transfer is complete, the Remote Server will automatically load information about the script's scheduling into memory.

CLEO CONFIDENTIAL. ©2008 CLEO Communications. All rights reserved. Cleo is a registered trademark of CLEO Communications

RMCS Update

RMCS has the ability to update itself. This is useful when upgrading to different versions or adding new extension DLLs. The update process is relatively simple:

1) Create a zip file with the files to update. The zip file is unpacked in the Remote Server installation folder.
2) Transfer the zip file into the special "**UPDATE**" folder.
3) Send the "rmanage /update" call. This call will shut down the service, unpack any zip files in the update folder, and restart the service.

Here's a potential update script (minus the specialization section).

```
<!-- This section contains the main portion of the script -->
<script name="main" xmlns="http://www.cuesoft.com/CUEScript/1.0">
 <var name="initialization"/>
 <var name="kiosk"/>
 <var name="location"/>
 <var name="port"/>
 <var name="updateFolder" expr="'c:\rmcs\updates\'" scope="global"/>

 <!-- Main script area -->
 <var name="quitEvent" expr="$system.CreateNetObject( 'System.Threading.ManualResetEvent', false() )"  scope="global"/>
 <var name="successfulUpdate" expr="false()"  scope="global"/>

 <expr>$log.Info('Remote update script start.')</expr>
 <try>
  <!-- Make the remote connection -->
  <expr>
   $connection.Configure( "location=" ^ $location ^ ";port=" ^ $port);
   $connection.Open();
   connect( 2 );
   $log.Info( "Connected!" );

   <!-- Send the zip files from the CommServer to the remote. -->
   $fileTransfer.PutFiles( $updateFolder ^ '*.zip', "**UPDATE**" );
  </expr>

  <if test="not($fileTransfer.WaitForStart(10000))">
   <expr>
    $log.Info('No update transfer was initiated.')
    $successfulUpdate := false();
    $quitEvent.Set()
   </expr>
  </if>

  <expr>
   <!-- Wait for transfers to quit or some other quit operation -->
   $quitEvent.WaitOne();
  </expr>

  <if test="$successfulUpdate = false()">
   <!-- Raise exception -->
```

```
    <raise explanation="Unable to update remote."/>
   </if>
   <else>
    <expr>
     $log.Info( "Sending remote update command." );
     $command.Send( "rmanage /update" );
     $log.Info('Update succeeded.')
    </expr>
   </else>
  </try>
  <catch>
   <expr>
    $log.Error( 'Script exception.', $exception );
   </expr>
   <reraise/>
  </catch>
  <expr>$log.Info('Remote update script end.')</expr>

  <!-- Quit command event -->
  <event name="OnQuitScript">
   <expr>
    $quitEvent.Set()
   </expr>
  </event>

  <!-- OnDisconnect - connection was lost -->
  <event name="OnDisconnect">
   <expr>
    $quitEvent.Set()
   </expr>
  </event>

  <!-- OnTransfersFinished - called when all transfers are completed -->
  <event name="OnTransfersFinished">
   <expr>
    $quitEvent.Set();
   </expr>
  </event>

  <!-- OnTransferSuccess - called when an individual transfer has succeeded -->
  <event name="OnTransferSuccess">
   <parameter name="sender"/>
   <parameter name="e"/>
   <expr>
    $log.Info( "Transfered update file: " ^ $e.FilePath() );
    $successfulUpdate := true();
   </expr>
  </event>

  <!-- OnTransferError - called when an individual transfer fails -->
  <event name="OnTransferError">
   <parameter name="sender"/>
   <parameter name="e"/>
   <expr>
    $log.Error( "Failed to transfer update file: " ^ $e.FilePath() );
   </expr>
  </event>
 </script>
```

CLEO CONFIDENTIAL. ©2008 CLEO Communications. All rights reserved. Cleo is a registered trademark of CLEO Communications

Value Cache

The value cache is an in-memory store of name/value pairs on the remote server. When a connection is established with a remote appliance, the contents of the cache are transferred to the Communication Server during the initial handshaking. You can then query on these values to make a determination on what to do next.

The cache can be used to store the machine mpid, to store information about configuration, to hold data about the last connect time, or to hold version information.

The value cache is volatile, meaning that if the computer restarts or the service shuts down, its contents are lost. There are ways to persist the values of the cache between sessions.

The following example shows how to load the cache with the mpid value when the service starts.

```
<job xmlns='http://www.cleo.com/Rmcs/Schedule/1.0' xmlns:cs='http://www.cuesoft.com/CUEScript/1.0'>
 <schedule>
  <onstartup run='startup'/>
 </schedule>

 <cs:script name='startup'>
  <cs:expr>
   $result := $local.GetIniValue( "coinstarproperty", "mpid", "c:\pasta\pastaman.ini" );
   $valueCache.SetValue( "mpid", $result );
   $valueCache.SetValue( "startTime", datetime().ToString() );

   $log.Debug( 'Setting value mpid=' ^ $valueCache.GetValue( "mpid" ) )
  </cs:expr>
 </cs:script>
</job>
```

To build a persistent cache, you could load the cache from a file when the service starts and save the cache to a file when the service shuts down.

Here is an example of loading the cache from an XML file on startup.

```
<?xml version="1.0" encoding="utf-8" ?>
<job xmlns='http://www.cleo.com/Rmcs/Schedule/1.0' xmlns:cs='http://www.cuesoft.com/CUEScript/1.0'>
 <schedule>
  <onstartup run='startup'/>
 </schedule>

 <cs:script name='startup'>
  <cs:expr>
   $xmlFile := new( "System.Xml.XmlDocument" );
  </cs:expr>
  <cs:if test='$system.Invoke( "System.IO.File", "Exists", "c:\vcache.xml" )'>
   <cs:expr>
    $xmlFile.Load( "c:\vcache.xml" );
```

CLEO CONFIDENTIAL. ©2008 CLEO Communications. All rights reserved. Cleo is a registered trademark of CLEO Communications

```
    $values := $xmlFile.SelectNodes( "//value" );
   </cs:expr>
   <cs:foreach var='val' in='$values'>
    <cs:expr>$valueCache.SetValue( $val.GetAttribute( "name" ), $val.GetAttribute( "val" ) )</cs:expr>
   </cs:foreach>
  </cs:if>
 </cs:script>
</job>
```

To save the cache, you could have code like:

```
<?xml version="1.0" encoding="utf-8" ?>
<job xmlns='http://www.cleo.com/Rmcs/Schedule/1.0' xmlns:cs='http://www.cuesoft.com/CUEScript/1.0'>
 <schedule>
  <onshutdown run='save'/>
 </schedule>

 <cs:script name='save'>
  <cs:expr>
   <![CDATA[
   $xmlFile := new( "System.Xml.XmlDocument" );
   $xmlFile.LoadXml( "<values/>" );
   ]]>
  </cs:expr>
  <cs:foreach var='vpair' in='$valueCache'>
   <cs:expr>
    $element := $xmlFile.CreateElement( "value" );
    $element.SetAttribute( "name", $vpair.Key );
    $element.SetAttribute( "val", $vpair.Value );
    $xmlFile.DocumentElement.AppendChild( $element );
   </cs:expr>
  </cs:foreach>
  <cs:expr>$xmlFile.Save( "c:\vcache.xml" )</cs:expr>
 </cs:script>
</job>
```

From the Communication Server script, you could invoke the script section to save with a command like:

```
rscript /sec:save savevcache.xml
```