Accessing QueueMetrics through its XML-RPC interface

Loway

Accessing QueueMetrics through its XML-RPC interface

Loway

Table of Contents

List of Tables

Chapter 1. Document contents

> If you are starting with a new integration project, we suggest that you use the newer JSON APIs that are more powerful and easier to use.

This document details how to access and use the XML-RPC access functionality in Loway QueueMetrics. This makes it possible for your programs to leverage the power of QueueMetrics by calling a very simple API, with bindings available in nearly every programming language.

1.1. Revision history

- Nov 13, 2006: First draft

- March 22, 2007: Added PHP example

- April 13, 2007: "raw" blocks available

- May 11, 2007: real-time blocks and auth server

- Jun 11, 2007: multi-stint calls

- Nov 11, 2007: enter-queue position and schedule adherence

- Nov 10, 2008: Added Outcomes data blocks and external XML-RPC Authentication description

- Jan 14, 2009: Added support for JavaScript

- Feb 11, 2009: Moved to AsciiDoc format for ease of updating

- Feb 17, 2009: Added QM.findAudio methods

- Aug 07, 2009: Added QM.qareport and QM.qaformreport reports

- Aug 10, 2009: Added AgentsDO.AnsCallsCG for "Answered calls by custom group" reporting

- Dec 2, 2009: Added QM.qaformsummary report

- Jul 8, 2010: Added new blocks "Call Overview", "Traffic analysis", "Agent performance"

- Jul 21, 2010: Added QM.qaformgrading method

- Aug 20, 2010: Added new block "Inclusive Answered SLA"

- Oct 1, 2010: Referenced QM manual for possible data blocks.

- Dec 2, 2010: Added new QA related optional parameter, Added new QM.tskAddNote and QM.qacallstograde methods

- May 16, 2011: New method setActivationKey()

- Oct 6, 2011: Added new QM.tskAddTraining methods. Modified QM.tskAddNote in order to receive the processField

- Oct 7, 2011: Added new QM.broadcastMessage method.

- Oct 24, 2011: Added new QM.tskAddMeeting method.

- Nov 14, 2011: New method QM.authenticate()

- Nov 23, 2011: New method QM.insertRecordTag()

- March 6, 2012: New methods QM.getQueuesForAgent() and QM.getPauseCodesForAgent()

- November 15, 2012: Updated QM.qaformsummary report for non scoring questions

- Jan 21, 2013: New PHP library

- Nov 2, 2013: External data sources

- Jun 13, 2014: JSON API available

- Jun 01, 2015: Centralized CBT tracking

Chapter 2. What is XML RPC?

Wikipedia defines XML-RPC as:

XML-RPC is a remote procedure call protocol which uses XML to encode its calls and HTTP as a transport mechanism. It is a very simple protocol, defining only a handful of data types and commands, and the entire description can be printed on two pages of paper. This is in stark contrast to most RPC systems, where the standards documents often run into the thousands of pages and require considerable software support in order to be used.

This means that, whatever your programming language of choice, you can surely find an XML-RPC library for it; and once you have the library, connection to QueueMetrics is straightforward.

> QueueMetrics also offers a JSON API that is easier to work with and offers extended functionality. If you are starting with a new project, you should preferably use the JSON API.

2.1. Which functions does QueueMetrics export as XML-RPC?

QueueMetrics exports the full results of an analysis in XML-RPC, so you can access whatever information you feel you may need. Information is divided into blocks, i.e. sets of data that roughly correspond to the tables QM uses for its own output.

This means that you can build software that sits on top of QueueMetrics and uses its result as a starting point for further elaboration, e.g.:

- Visualizing results with complex graphs currently not supported by QueueMetrics

- Computing period comparison analyses (one period versus another period)

- Accessing agent presence data for payroll computation

Of course there are many possible scenarios where you might want to use such information.

The XML-RPC interface is available in all version of QueueMetrics starting from version 1.3.1.

> If you plan to use the XML-RPC, we suggest starting by installing the open XML-RPC Query Browser available at https://github.com/Loway/QueueMetricsXmlRpcBrowser so you can familiarize yourself with the interface right from your browser.

2.2. Example: accessing QueueMetrics from Python

In this example we'll see how easy it is to access QueueMetrics from a scripted language like Python. You can enter the following statements interactively using a Python IDE like IDLE, or make them a part of a larger program.

The following code connects to the XML-RPC port of a QueueMetrics instance running at http://qmserver:8080/qm130 and asks for a couple of tables, namely the distribution of answered calls per day and the Disconnection causes.

```
> import xmlrpclib
> server_url = 'http://qmserver:8080/qm130/xmlrpc.do';
> server = xmlrpclib.Server(server_url);
> res = server.QM.stats( "queue-dps", "robot", "robot","", "", "2005-10-10.10:23:12", "20

> res.keys()
['CallDistrDO.AnsDistrPerDay', 'result', 'KoDO.DiscCauses']

> res['result']
[['Status', 'OK'], ['Description', ''], ['Time elapsed (ms)', 3008], ['QM Version', '1.3

> res['result'][2][1]
3008
```

As you can see, it only takes four lines of Python code to connect to QueueMetrics and get all the results back!

2.3. Example: Accessing QueueMetrics from Java

This is an example functionally equivalent to the one above in Python, but it's written in Java using the Redstone XML-RPC client library.

```
import java.io.IOException;
import java.util.HashMap;
import java.net.URL;
import redstone.xmlrpc.XmlRpcClient;
```

```java
import java.util.ArrayList;

public class xmlRpcTestClient {
    public void perform() {
        String stUrl = "http://server:8080/qm130/xmlrpc.do";
        System.setProperty(
                "org.xml.sax.driver",
                "org.apache.xerces.parsers.SAXParser"
        );

        try {
            XmlRpcClient client = new XmlRpcClient( stUrl, false );

            ArrayList arRes = new ArrayList();
            arRes.add( "OkDO.AgentsOnQueue" );
            arres.add( "KoDO.DiscCauses" );
            arRes.add( "KoDO.UnansByQueue" );
            arRes.add( "DetailsDO.CallsKO" );

            Object[] parms = { "queue-dps", "robot", "robot",
                                "", "", "2005-10-10.10:23:12",
                            "2007-10-10.10:23:10",
                                "", arRes };

            Object token = client.invoke( "QM.stats", parms );
            HashMap resp = (HashMap) token;
            System.out.println( "Resp: " + resp );
        } catch ( Exception e ) {
            e.printStackTrace();
        }
    }

    public static void main(String[] args) {
        xmlRpcTestClient c = new xmlRpcTestClient();
        try {
            c.perform();
        } catch (Exception e ) {
            e.printStackTrace();
        }
    }
}
```

We'll have to explicitly set which XML parser to use; for the rest, the code looks very much alike the Python one.

2.4. Example: Accessing QueueMetrics from PHP

This example is based on PHP's *xmlrpc* module, available from http://phpxmlrpc.sourceforge.net/. The XML_RPC module that comes with PEAR does not support HTTP/1.1 and so it will likely not work with recent version of Apache Tomcat.

```php
<h1>A QueueMetrics XML-RPC client in PHP</h1>
<?
// import the library here
include "./xmlrpc.inc";

$qm_server = "10.10.10.10";   // the QueueMetrics server address
$qm_port   = "8080";          // the port QueueMetrics is running on
$qm_webapp = "queuemetrics";  // the webapp name for QueueMetrics


// set which response blocks we are looking for
$req_blocks = new xmlrpcval(array(
```

```php
                      new xmlrpcval("DetailsDO.CallsOK"),
                      new xmlrpcval("DetailsDO.CallsKO")
            ), "array");

// general invocation parameters - see the documentation
$params = array(
new xmlrpcval("queue-dps"),
                new xmlrpcval("robot"),
                new xmlrpcval("robot"),
                new xmlrpcval(""),
                new xmlrpcval(""),
                new xmlrpcval("2007-01-01.10:23:12"),
                new xmlrpcval("2007-10-10.10:23:10"),
                new xmlrpcval(""),
                $req_blocks
        );

$c = new xmlrpc_client("$qm_webapp/xmlrpc.do", $qm_server, (integer) $qm_port );
$c->return_type = 'phpvals'; // let client give us back php values instead of xmlrpcvals
$resp =& $c->send(new xmlrpcmsg( 'QM.stats', $params));

if (!$resp) {
    echo 'Communication error: ' . $cli->errstr;
    exit;
}
if ($resp->faultCode()) {
    echo 'Fault Code: ' . $resp->faultCode() . "\n";
    echo 'Fault Reason: ' . $resp->faultString() . "\n";
} else {
    $val = $resp->value();
    $blocks = XML_RPC_decode($val);

    // now we print out the details....
    printBlock( "result", $blocks );
    printBlock( "DetailsDO.CallsOK", $blocks );
    printBlock( "DetailsDO.CallsKO", $blocks );
}

// output a response block as HTML
function printBlock( $blockname, $blocks ) {
        echo "<h2>Response block: $blockname </h2>";
        echo "<table border=1>";
        $block = $blocks[$blockname];
        for ( $r = 0; $r < sizeof( $block ); $r++ ) {
                echo "<tr>";
                for ( $c = 0; $c < sizeof( $block[$r] ); $c++ ) {
                        echo( "<td>" . $block[$r][$c] . "</td>" );
                }
                echo "</tr>\n";
        }
        echo "</table>";
}
?>
```

In this next example, the PHP code is able to submit QA grading information in order to fill a specific form:

```php
<?php
    // import the library here
    include "./xmlrpc.inc";
```

```php
$qm_server = "10.10.10.10";   // the QueueMetrics server address
$qm_port   = "8080";           // the port QueueMetrics is running on
$qm_webapp = "queuemetrics";  // the webapp name for QueueMetrics

// set which response blocks we are looking for
$req_blocks = new xmlrpcval(array(
    new xmlrpcval("QualAssFormDO.FormStructure"),
    new xmlrpcval("QualAssFormDO.SectionValues"),
    new xmlrpcval("QualAssFormDO.Comments"),
    ), "array");

// set notes we want to update
$comments_block = new xmlrpcval(array(
    new xmlrpcval("This is a form note added with XMLRPC"),
    new xmlrpcval("This is the next form note added with XMLRPC"),
    ), "array");

// set QA infor scores to be
// Each item code should be defined.
// Each item will be associated to a score number
// or an empty string if N/A (only for Not Mandatory fields)
$grading_info = new xmlrpcval(array(
    'NMYN' => new xmlrpcval("", 'string'),
    'NMNUM' => new xmlrpcval("20", 'string'),
    'MNUM' => new xmlrpcval("60", 'string'),
    'MMUL' => new xmlrpcval("100", 'string'),
    ), "struct");

// general invocation parameters - see the documentation
$params = array(
    new xmlrpcval(""),
    new xmlrpcval("robot"),
    new xmlrpcval("robot"),
    new xmlrpcval("2010-06-16.12:32:00"),
    new xmlrpcval("3600"),
    new xmlrpcval("Test GUI items"),
    new xmlrpcval("1276684326.309"),
    $grading_info,
    $comments_block,
    $req_blocks
    );


$c = new xmlrpc_client("$qm_webapp/xmlrpc.do", $qm_server, (integer) $qm_port );
$c->return_type = 'phpvals'; // let client give us back php values instead of xmlrpc
$resp =& $c->send(new xmlrpcmsg( 'QM.qaformgrading', $params));

if (!$resp) {
    echo 'Communication error: ' . $cli->errstr;
    exit;
}

if ($resp->faultCode()) {
    echo 'Fault Code: ' . $resp->faultCode() . "\n";
    echo 'Fault Reason: ' . $resp->faultString() . "\n";
} else {
    $val = $resp->value();
    $blocks = XML_RPC_decode($val);
```

```
        // now we print out the details....
        printBlock( "result", $blocks );
        printBlock( "QualAssFormDO.FormStructure", $blocks );
        printBlock( "QualAssFormDO.SectionValues", $blocks );
        printBlock( "QualAssFormDO.Comments", $blocks );
    }

    // output a response block as HTML
    function printBlock( $blockname, $blocks ) {
        echo "\nResponse block: $blockname \n";
        $block = $blocks[$blockname];

        for ( $r = 0; $r < sizeof( $block ); $r++ ) {
            echo "\n";
            for ( $c = 0; $c < sizeof( $block[$r] ); $c++ ) {
                echo( $block[$r][$c] . "\t" );
            }
        }
    }
}
?>
```

As you can see, it is very similar to the other programming languages, and reading into the results is simply a matter of selecting the correct block and then accessing the data cell by row and column. The added complexity is due to the explicit error condition check and result printout.

2.5. Example: Accessing QueueMetrics from JavaScript

In order to access the XML-RPC interface from JavaScript, you need to use an adaptor library. We have been able to successfully use a library called JSON-XML-RPC that can be found at http://code.google.com/p/json-xml-rpc/ in a file called rpc.js

Generally speaking, a JavaScript client can make requests only against the same server that is serving the HTML page, therefore you need to install it on the same server as QueueMetrics, creating a separate webapp.

```
<html>
<head>
<title>javascript_client.html</title>
<script src="rpc.js"></script>
</head>
<body>
<h1>QueueMetrics JavaScript XML-RPC example </h1>
<script>
var server = "/DAILY/xmlrpc.do";

function run() {
  try {
    var service = new rpc.ServiceProxy( server, {
            asynchronous:false,
            protocol: "XML-RPC",
            methods: ["QM.stats", "QM.realtime", "QM.auth"]
    } );
    res = service.QM.stats( "q1", "robot", "robot","", "",
            "2005-10-10.10:23:12", "2009-10-10.10:23:10","",
            [ "KoDO.DiscCauses", "CallDistrDO.AnsDistrPerDay" ]
    );
    document.getElementById("RESULT").innerHTML = plotBlocks(res);
  } catch(e) {
    alert("Error raised: " + e);
  }
}


function plotBlocks( hmBlocks ) {
```

```
  s = "";
  for (var i in hmBlocks) {
    s += "<h2>Block: " + i + "</h2>";
    s += plotBlock( hmBlocks[i] );
  }
  return s;
}

function plotBlock( arBlock ) {
  s = "";
  for ( r =0; r < arBlock.length; r++ ) {
    s += "<tr>";
    for ( c = 0; c < arBlock[r].length; c++ ) {
      s += "<td>" + arBlock[r][c] + "</td>";
    }
    s += "</tr>";
  }
  return "<table border=1>"  + s + "</table>";
}
</script>

<input type="button" value="Click me!" onclick="run();" >
<div id="RESULT"></div>

</body>
</html>
```

As you can see, the code is actually very similar to the Python one. The only important difference here is that the names of the methods have to be explicitly declared.

Chapter 3. Understanding call parameters

There are a number of methods exported by the XML-RPC interface, and they are used for different reasons:

- QM.stats: get the main historical stats

- QM.realtime: get the real time stats (available since QM 1.3.5)

- QM.qareport: get the Quality Assessment statistics (available since QM 1.6.0)

- QM.qaformreport: get some Quality Assessment Forms details (available since QM 1.6.0)

- QM.qaformsummary: get Quality Assessment statistics related to the same form (available since QM 1.6.0)

- QM.qaformgrading: fill a QA form with the specified scores and/or comments (available since QM 12.01)

- QM.auth: use QueueMetrics as an auth server for third party software (available since QM 1.3.5)

3.1. The method QM.stats

This is the main method exported, its XML-RPC name being QM.stats. It takes a number of arguments, all of which must be supplied and must have the correct data type. They are:

1. *(String)* Queue name or names: the set of queues that must be included in the analysis. They must be separated by a "|" symbol if more than one queue is passed. The queue name is the internal Asterisk queue name.

2. *(String)* Access username: this one must be the user name of an active user holding the key ROBOT.

3. *(String)* Access password: this must be the clear-text password of the given user name

4. *(String)* Logfile - always leave blank.

5. *(String)* Period - always leave blank.

6. *(String)* Start of period. This must be written in exactly the format yyyy-mm.dd.hh.mm.ss (do not forget the dot between the date and the hour).

7. *(String)* End of period. Same format as start of period

8. *(String)* Agent filter - an agent's name, like "Agent/101" that must be the filter for all the relevant activity

9. *(List)* A list of the required analysis to be returned to the client. Each analysis name must be supplied as a String.

This call will start up a session in QueueMetrics, check if the user exists and has the privilege to run the report, run the analysis, prepare the required results and return them. At the end of the call, the QueueMetrics session is destroyed so no data is kept for further elaboration.

This means that it's usually the most efficient thing to do to request all needed response information at once, but it's wise to limit yourself to the minimum data set you will actually need, as each block takes up CPU and memory space to be marshaled between the native Java format, the intermediate XML format and the resulting client format.

It is also advisable to run large data set analysis at night time or when nobody is accessing the system, as they may take quite a lot of RAM and CPU, and this may slow down QueueMetrics for the other live users.

3.2. The method QM.realtime

This method is very similar to QM.stats but it is used to retrieve the real time stats. It must be called with the following parameters:

1. *(String)* Queue name or names: the set of queues that must be included in the analysis. They must be separated by a "|" symbol if more than one queue is passed. The queue name is the internal Asterisk queue name.

2. *(String)* Access username: this one must be the user name of an active user holding the key ROBOT.

3. *(String)* Access password: this must be the clear-text password of the given user name

4. *(String)* Logfile - always leave blank.

5. *(String)* Agent filter - an agent's name, like "Agent/101" that must be the filter for all the relevant activity

6. *(List)* A list of the required analysis to be returned to the client. Each analysis name must be supplied as a String.

The same suggestions that are given for *QM.stats* apply.

> Please note that there is a difference between results produced by the XML RPC realtime calls and the realtime statistics produced through the QueueMetrics GUI when the key *realtime.members_only* is equal to true. The difference is related to

the agents list shown. Being the list of queues, in the XML RPC call, specified by a list of names instead of a list of queue unique identifiers, it's not possible to correctly identify elementary queues from macro queues having the same name. In this situation the agent list will always be calculated as the sum of all agents associated to all elementary queues composing the macro queue, even if the macro queue has directly assigned agents.

3.3. The method QM.qareport

This method is very similar to QM.stats but it's used to retrieve Quality Assessment statistics. It must be called with the following parameters:

1. *(String)* Queue name or names: the set of queues that must be included in the analysis. They must be separated by a "|" symbol if more than one queue is passed. The queue name is the internal Asterisk queue name.

2. *(String)* Access username: this one must be the user name of an active user holding the key ROBOT.

3. *(String)* Access password: this must be the clear-text password of the given user name

4. *(String)* Start of period. This must be written in exactly the format yyyy-mm-dd.hh.mm.ss (do not forget the dot between the date and the hour).

5. *(String)* End of period. Same format as start of period

6. *(String)* Agent filter - an agent's name, like "Agent/101" that must be the filter for all the relevant activity

7. *(String)* The form name for wich you need to have information

8. *(String)* The grader type used to filter out graded forms. It could be any of the following values: "unknown", "agent", "grader", "caller". This parameter is optional and could not be present in the function call

9. *(List)* A list of the required analysis to be returned to the client. Each analysis name must be supplied as a String.

The same suggestions that are given for *QM.stats* apply.

3.4. The method QM.qaformreport

This method is very similar to QM.stats but it's used to retrieve raw information about Quality Assessment Forms. It must be called with the following parameters:

1. *(String)* Queue name or names: the set of queues that must be included in the analysis. They must be separated by a "|" symbol if more than one queue is passed. The queue name is the internal Asterisk queue name.

2. *(String)* Access username: this one must be the user name of an active user holding the key ROBOT.

3. *(String)* Access password: this must be the clear-text password of the given user name

4. *(String)* Start of period. This must be written in exactly the format yyyy-mm-dd.hh.mm.ss (do not forget the dot between the date and the hour).

5. *(String)* End of period. Same format as start of period

6. *(String)* Agent filter - an agent's name, like "Agent/101" that must be the filter for all the relevant activity

7. *(String)* The form name for wich you need to have information

8. *(String)* The grader type used to filter out graded forms. It could be any of the following values: "unknown", "agent", "grader", "caller". This parameter is optional and could not be present in the function call

9. *(List)* A list of the required analysis to be returned to the client. Each analysis name must be supplied as a String.

The same suggestions that are given for *QM.stats* apply.

3.5. The method QM.qaformsummary

This method is very similar to QM.stats but it's used to retrieve aggregated information about a specific Quality Assessment Form. It must be called with the following parameters:

1. *(String)* Queue name or names: the set of queues that must be included in the analysis. They must be separated by a "|" symbol if more than one queue is passed. The queue name is the internal Asterisk queue name.

2. *(String)* Access username: this one must be the user name of an active user holding the key ROBOT.

3. *(String)* Access password: this must be the clear-text password of the given user name

4. *(String)* Start of period. This must be written in exactly the format yyyy-mm-dd.hh.mm.ss (do not forget the dot between the date and the hour).

5. *(String)* End of period. Same format as start of period

6. *(String)* Agent filter - an agent's name, like "Agent/101" that must be the filter for all the relevant activity

7. *(String)* The form name for wich you need to have information

8. *(String)* The grader type used to filter out graded forms. It could be any of the following values: "unknown", "agent", "grader", "caller". This parameter is optional and could not be present in the function call

9. *(List)* A list of the required analysis to be returned to the client. Each analysis name must be supplied as a String.

The same suggestions that are given for *QM.stats* apply. The report counts the aggregated QA statistics on calls with timestamp included in the date range specified.

3.6. The method QM.qaformgrading

This method lets you fill a QA form through an RPC-XML call. It replies with the same raw information reported by the QM.qaformreport method and could replace it if QA parameters are empty when calling.

1. *(String)* Queue name or names: the set of queues that must be included in the analysis. They must be separated by a "|" symbol if more than one queue is passed. The queue name is the internal Asterisk queue name.

2. *(String)* Access username: this one must be the user name of an active user holding the key ROBOT.

3. *(String)* Access password: this must be the clear-text password of the given user name

4. *(String)* Start of period. This must be written in exactly the format yyyy-mm-dd.hh.mm.ss (do not forget the dot between the date and the hour).

5. *(String)* End of period. This is at least the number of seconds the call was in the waiting status (or the complete call time or a suitable number that comfortably contains the call like, for example, 3600).

6. *(String)* The form name for wich you need to have information

7. *(String)* The unique identifier for the call to be graded

8. *(String)* The grader type used to filter out graded forms. It could be any of the following values: "unknown", "agent", "grader", "caller". This parameter is optional and could not be present in the function call

9. *(Struct)* The list of QA items score supplied as (string,string) pairs. The list should contain all specific form items codes and their relative score. In order to specify N/A values for not mandatory items, an empty string should be specified. If the list is left empty, no QA score will be filled into the form

10. *(List)* A list of the notes to be filled in the form. Each note must be supplied as a String. If the list is empty, no new comments will be added to the form.

11. *(List)* A list of the required analysis to be returned to the client. Each analysis name must be supplied as a String.

The same suggestions that are given for *QM.stats* apply. The report counts the aggregated QA statistics on calls with timestamp included in the date range specified.

3.7. The method QM.qacallstograde

1. *(String)* Queue name or names: the set of queues that must be included in the analysis. They must be separated by a "|" symbol if more than one queue is passed. The queue name is the internal Asterisk queue name.

2. *(String)* Access username: this one must be the user name of an active user holding the key ROBOT.

3. *(String)* Access password: this must be the clear-text password of the given user name

4. *(String)* Start of period. This must be written in exactly the format yyyy-mm-dd.hh.mm.ss (do not forget the dot between the date and the hour).

5. *(String)* End of period. This must be written in exactly the format yyyy-mm-dd.hh.mm.ss (do not forget the dot between the date and the hour).

6. *(String)* The form name for wich you need to have information

7. *(String)* Agent filter - an agent's name, like "Agent/101" that must be the filter for all the relevant activity

8. *(List)* A list of constraints used by the engine to filter out all relevant forms (see below for the correct syntax)

9. *(List)* A list of the required analysis to be returned to the client. Each analysis name must be supplied as a String.

The constraints are a set of (key, values) pairs used by the engine to filter out the calls to be graded. The constraint list should be defined with the proper syntax in order to be correctly interpreted by QueueMetrics.

There are two types of constraints: the percentage values and the absolute values. They should be respectively specified through the suffixes "min" or "num".

The constraints are related to different categories:

- Individual agents: specified through the key AXG (like, for example: AGX_min or AGX_num)

- All calls: specified through the key AC (like, for example: AC_min or AC_num)

- Outcome code: specified through the key outcome followed by the outcome code and separated by an underscore character (like, for example: outcome_KN_num or outcome_KN_min)

- Agent group: specified through the key agroup followed by the agent group name and separated by an underscore character (like, for example: agroup_Default_min or agroup_Default_num)

3.8. The method QM.auth

This method is used to authenticate a username / password couple against the QueueMetrics server. This can be used by third-party software that does not want to keep its own separate user database but wants to use QueueMetrics' instead.

Call parameters:

1. *(String)* User name

2. *(String)* Password

Response

There is only one response block returned, named "auth", where the caller will retrieve all user data, including the live key set for that user.

3.9. The method QM.authenticate

This method is used to obtain the profile of a user given its login and password. The profile is made up of both login and - where applicable - agent information. It is possible to obtain the profile either of the very user you are calling (by knowing its login and password) or of a "deferred" separate user, if you have a user that would have the admin keys to view that information in the main GUI.

The deferred username works so that:

- If a deferred username is passed, and

- If the username/login pair are valid and the user has key USRADMIN and ROBOT

  - Then the User.* output for the deferred user is returned

  - If the user also holds the key USR_AGENT, the Agent.* output is returned as well

- The password-change function applies only to the user who logs in, not to the deferred user.

Call parameters

1. *(String)* User name

2. *(String)* Deferred user name, or blank

3. *(String)* Password

4. *(String)* New password, or blank if you do not want it changed.

Response

There is only one response block returned, named "output", where the caller will retrieve all user data, including the live key set for that user.

The output block has the format:

Column 0	Column 1	Explanation
user.user_id	173	Internal user-id
user.login	Agent/101	
user.real_name	John Doe	
user.class_name	AGENTS	
user.keys	USER AGENT XX YY	All computed keys, space-separated
user.n_logons	37	
user.last_logon	2011-10-08 12:34:56	
user.comment		Optional

Column 0	Column 1	Explanation
user.token	876543	Optional
user.email	me@home.it [mailto:me@home.it]	Optional
user.enabled	true	"true" or "false"
agent.id	86	Internal agent-id
agent.description	Agent J.D. (101)	The name displayed in reports
agent.aliases		A set of aliases (if present)
agent.location	Main	Blank if none
agent.group_name	Experienced agents	Blank if none
agent.current_terminal	Sip/1234	Blank or "-" if none
agent.vnc_url	http://1.2.3.4/vnc	Optional
agent.supervised_by	Demoadmin	Blank if none, or login of the supervisor
agent.xmpp_address	xmpp:101@myserver	XMPP chat address
agent.visibility_key		Optional
agent.payroll_code		Optional
password.changed	OK	OK if password was changed, blank otherwise

The following rules apply:

- Column zero contains the attribute

- Column one contains the value of the attribute (we supply a sample in the table above)

- Attribute names are not case sensitive

- If the user is also an agent, that is, there is an agent under the same name as the login, Agent attributes are passed.

- Blank attributes may or may not be present in the list of attributes

3.10. The method QM.findAudio

This method is used to retrieve a recorded file name from the QueueMetrics server. In order to retrieve the file name QM will invoke the currently configured Pluggable Modules to search within the current recording set. This can be used by third-party software that needs to retrieve audio recordings via HTTP.

Call parameters:

1. *(String)* User name (*)

2. *(String)* Password (*)

3. *(String)* Server

4. *(String)* Asterisk-ID (*)

5. *(String)* Call start (integer timestamp)

6. *(String)* Agent

7. *(String)* Queue

The username and password of a user with ROBOT access are mandatory.

If QM is on a clustered setup, the *Server* parameter must be passed to qualify the Asterisk call-id.

The *Asterisk-Id* is mandatory and is the one retrieved in the Call Details blocks.

Some PM may optionally require the *Call start*, *Agent* and *Queue* parameters; those are used for fuzzy matching of calls, e.g on an external storage. Most PMs that do an exact match do not need those parameters.

Response

There is only one response block returned, named "AudioFiles", where the caller will retrieve the filename of each recorded file and a URL to actually download the file.

The response may include zero or more files; it is well possible that multiple recordings be present for one call.

A sample PHP file that shows how to access the findAudio interface is supplied with QueueMetrics.

3.11. The method QM.insertRecordTag

This method is used to associated a new tag for a specific recording call file. This method could be used to retrieve the list of tags related to a specific call.

Call parameters:

1. *(String)* User name (*)

2. *(String)* Password (*)

3. *(String)* Server: the server name in a cluster setup or empty for a not cluster setup

4. *(String)* AsteriskId: the asterisk specific call id

5. *(String)* FileName: the recording filename associated to the tag to be inserted

6. *(String)* Time: the time (in seconds) where the tag will be placed. If empty, the tag will be placed at the beginning of the file

7. *(String)* Duration: the tag duration (in seconds). It could be empty.

8. *(String)* Message: the tag message. If empty, no tags will be added. This is useful for retrieve the list of tags associated to a specific call.

9. *(String)* Color: an hex color value in RGB form to be associated a tag. (from 000000 to FFFFFF)

Response

There is only one response block returned, named "TagRecords", where the caller will retrieve the list of tags associated to the specific *server* and *AsteriskId* parameters.

3.12. The method QM.tskAddNote

This method is used to insert a note task to a particular user. It's possible to specify the validity start and end date.

Call parameters:

1. *(String)* Access username: this one must be the user name of an active user holding the key ROBOT.

2. *(String)* Access password: this must be the clear-text password of the given user name

3. *(String)* Task to user: this is the login to the user where the task should be addressed

4. *(String)* Start of validity: This must be written in exactly the format yyyy-mm-dd.hh.mm.ss (do not forget the dot between the date and the hour). It could be empty.

5. *(String)* End of validity: This must be written in exactly the format yyyy-mm-dd.hh.mm.ss (do not forget the dot between the date and the hour). It could be empty.

6. *(String)* Task Process Field: This is an optional identifier defined as ProcessFamily/ProcessId to be associated to the task. Either ProcessFamily and/or ProcessId could be empty. The field could be empty.

7. *(String)* Task message: This is the message associated to the task

8. *(String)* Task notes: This is the optional note associated to the task

Response There is only the result response block. No other custom blocks will be returned as a result.

3.13. The method QM.tskAddTraining

This method is used to insert a training task to a particular user. It's possible to specify the validity start and end date, a title and an optional URL or ID that will be shown in the task detail page.

Call parameters:

1. *(String)* Access username: this one must be the user name of an active user holding the key ROBOT.

2. *(String)* Access password: this must be the clear-text password of the given user name

3. *(String)* Task to user: this is the login to the user where the task should be addressed

4. *(String)* Start of validity: This must be written in exactly the format yyyy-mm-dd.hh.mm.ss (do not forget the dot between the date and the hour). It could be empty.

5. *(String)* End of validity: This must be written in exactly the format yyyy-mm-dd.hh.mm.ss (do not forget the dot between the date and the hour). It could be empty.

6. *(String)* Task Process Field: This is an optional identifier defined as ProcessFamily/ProcessId to be associated to the task. Either ProcessFamily and/or ProcessId could be empty. The field could be empty.

7. *(String)* Task message: This is the message associated to the task

8. *(String)* Task notes: This is the optional note associated to the task

9. *(String)* Task reason: This is an optional reason associated to the task.

10. *(String)* Task Training Title: This is an optional task training title.

11. *(String)* Task URL: This is an optional URL (in the http://www.mydomain.com/path format) to be associated with the training task. It could be empty.

12. *(String)* Task ID: This is an optional ID associated to the task. If an URL if defined by the previous parameter, this field will be ignored. It could be empty.

13. *(String)* Training CBT Code: This is an optional code referring to a QueueMetrics known CBT item. If not empty, this field overrides the fields Task URL dna Task ID.

Response There is only the result response block. No other custom blocks will be returned as a result.

3.14. The method QM.tskAddMeeting

This method is used to insert a meeting task to a particular user. It's possible to specify the validity start and end date, a title, a date and an optional duration that will be shown in the task detail page.

Call parameters:

1. *(String)* Access username: this one must be the user name of an active user holding the key ROBOT.

2. *(String)* Access password: this one must be the clear-text password of the given user name

3. *(String)* Task to user: this is the login to the user where the task should be addressed

4. *(String)* Start of validity: This must be written in exactly the format yyyy-mm-dd.hh.mm.ss (do not forget the dot between the date and the hour). It could be empty.

5. *(String)* End of validity: This must be written in exactly the format yyyy-mm-dd.hh.mm.ss (do not forget the dot between the date and the hour). It could be empty.

6. *(String)* Task Process Field: This is an optional identifier defined as ProcessFamily/ProcessId to be associated to the task. Either ProcessFamily and/or ProcessId could be empty. The field could be empty.

7. *(String)* Task title: This is the title associated to the task

8. *(String)* Task message: This is the message associated to the task

9. *(String)* Task date: This is the meeting date. This must be written in exactly the format yyyy-mm-dd.hh.mm.ss (do not forget the dot between the date and the hour).

10. *(String)* Task duration: This is the optional meeting duration. If no duration is specified (or specified as 0) the meeting will have an undefined duration.

11. *(String)* Task notes: This is the optional note associated to the task

12. *(String)* Tasl reason: This is the optional reason associated to the task

Response There is only the result response block. No other custom blocks will be returned as a result.

3.15. The method QM.setActivationKey

This method is used to remotely change the configuration key of the QueueMetrics instance or to query the current configuration key.

As this operation is potentially critical, the user sending this request must hold the keys ROBOT and KEYUPDATE. We ship such a user named *keyupdater* in the default QM configuration but it has to be manually enabled. Make sure you change the password as well.

As the system must be restarted after setting the new key so that it is picked up (this is done automatically), the QM server may be unavailable for a few seconds during the restart phase and current user sessions may be forcibly terminated. It is therefore not advisable to run this command on a busy system with many users logged in.

Call parameters:

1. *(String)* Access username: this one must be the user name of an active user holding the keys ROBOT and KEYUPDATE.

2. *(String)* Access password: this must be the clear-text password of the given user name

3. *(String)* New key: this is the key that must be set. If blank, the current parameters are reported.

Response

The custom block "KeyResults" is filled with the following parameters:

- *KEY_status* : NOKEY if no new key is given, otherwise OK or ERROR depending on the success of the operation

- *KEY_plexId* : The server identifier

- *KEY_message* : A message explaining what went wrong

- *KEY_current_appl* : The name of the application

- *KEY_current_user* : The name of the user that the application is licensed to

- *KEY_current_exp* : The expiration date for the current key

Please note that when a new key is installed, the current user and expiration date are those of the system on which the key is being installed; you should get the new ones as soon as the system restarts (will usually take between 5 and 20 seconds).

3.16. The method QM.broadcastMessage

This method is used to insert broadcast messages that will be shown in the realtime broadcast message page.

Call parameters:

1. *(String)* Access username: this one must be the user name of an active user holding the key ROBOT.

2. *(String)* Access password: this must be the clear-text password of the given user name

3. *(String)* Message text: the text message to be broadcasted

4. *(String)* Queue Name: the optional queue name (it could be empty)

5. *(String)* Location Name: the optional location name (it could be empty)

6. *(String)* Supervisor Login: an optional supervisor login (it could be empty). The message will be broadcasted to people reporting to the supervisor login.

7. *(String)* Agent Login: an optional agent login (it could be empty). The message will be addressed to the specified agent.

8. *(String)* forEveryone: this could be "true" or "1" for messages to be broadcasted to everyone; "false" or "0" otherwise.

Response There is only the result response block. No other custom blocks will be returned as a result.

3.17. The method QM.getQueuesForAgent

Given an agent (like Agent/101) let the caller know on which queue(s) he's supposed to work, as per the configuration on the QM interface. For each queue, we get also back a "level", that is a penalty level, like 0, 1 or 2.

It will also return the composite queues the agent is known on and the level he's scheduled on them.

Call parameters

1. *(String)* User name

2. *(String)* Password

3. *(String)* Agent code (must exactly match the one present in QM)

Response

If the agent code is unknown, an exception is raised.

There is only one response block returned, named "output", where the caller will retrieve all user data, including the live key set for that user.

The output block has the format:

Column 0	Column 1	Column 2	Column 3
Agent/101	Q1	Queue 1	0
Agent/101	Q2	My Queue 2	2
Agent/101	Q3 Q4	Monitor 3 & 4	1

Explanation as follows:

- Column zero contains the agent code.

- Column one contains the queue or composite queue it is known for. These are a set of the queues as they are known in Asterisk. They are separated by either a space or a vertical pipe (|) symbol

- Column two contains the name that such queue(s) appear in the QM interface

- Column 3 contains the agent level, as per:

  - 0: Main (no penalty)

  - 1: Wrap

  - 2: Spill

3.18. The method QM.getPauseCodesForAgent

Given an agent (like Agent/101), the caller gets back a set of pause codes and their description as it would be visible to this user. As QM allows protecting pause codes with security keys (so that e.g. you can have some pauses visible by some users only) QM computes the set of allowed pause from the point of view of the agent.

Call parameters

1. *(String)* User name

2. *(String)* Password

3. *(String)* Agent code (must exactly match the one present in QM)

Response

If the agent code is unknown, an exception is raised.

There is only one response block returned, named "output", where the caller will retrieve all user data, including the live key set for that user.

The output block has the format:

Column 0	Column 1	Column 2	Column 3
Agent/101	03	Lunch break	PNB
Agent/101	17	E-mail	PB
Agent/101	26	Coffee break	NPNB

Explanation as follows:

- Column zero contains the agent code.

- Column 1 contains the pause code, as should be reported in Asterisk

- Column 2 is the description

- Column 3 is the pause type:

  - PB - pause is payable and billable

  - PNB - pause is payable but not billable

  - NPB - pause is not payable but billable (unlikely!)

  - NPNB - pause is neither payable nor billable

Chapter 4. Understanding results

The result of the analysis is made up of a Struct, i.e. an associative array similar to Java's HashMap or Perl's Hash, that maps block names to blocks.

Each block is a rectangular data container made up of an Array of Arrays of Strings. The outer Array contains the rows while the inner ones will hold column values.

In the response, you get all the requested blocks as of parameter #9 of the call, plus a block named response. The block response will contain:

- The status of the call: this should always be OK if no error was encountered. If this is not OK, the other blocks may be missing or meaningless.

- The version of QueueMetrics running in the server

- The time elapsed to run the report.

All other blocks will follow the convention that the inner Array #0 (the first row) will contain table headers, while actual data will be present from row 1 onwards. The table headers are the same you can see in the on-screen representation.

The following Python code will run a simple dump of all the results in a readable form:

```
for t in res.keys():
        print "===== Block " + t + ": ========"
        for r in range( len(res[t]) ):
                for c in range( len( res[t][r] )):
                        print res[t][r][c] ,
                        print "\t" ,
                print
```

The output will look like the following one:

```
===== Block result: ========
Status             OK
Description
Time elapsed (ms)   3008
QM Version          1.3.1


===== Block KoDO.DiscCauses: ========
Cause               N. Calls         ...
Caller abandon      46          83.6%
Agent dump          1           1.8%
Timeout (voicemail) 8           14.5%
```

As you can see, the block KoDO.DiscCauses has one full line of header on its first row.

Chapter 5. Using an external auth source for QueueMetrics

QueueMetrics can accept an external authentication source. By using a simple XML-RPC interface, it is possible to write third party scripts that are able to authenticate against a chosen authentication system (e.g. Kerberos, LDAP, a Microsoft domain server, etc).

5.1. Log on procedure

When an user logs in to QueueMetrics, QM checks to see if there is an XML-RPC auth source defined . QueueMetrics will execute a query to that third-party server passing along the user-id and the password given for authentication, plus a given service-id that will be defined in the configuration file.

The server will respond passing along a fixed set of data:

- An auth response (see below)

- A set of login information for that user (e.g. real name, email)

- The current class and user keys for that user

If the class passed along is empty, only the user keys will be computed. If the class is not empty, it will be searched by name on the QueueMetrics storage and it will be loaded. Trying to load a non-existent class will result in auth failure.

The auth server may return four different responses:

- Access allowed

- Access allowed with supplied user data

- Access is forbidden

- Access is fully delegated to QM

The following table explains the relationship between those states:

	Auth OK	Auth KO
User data from RPC	AUTHORITATIVE	FORBIDDEN
User data from QM	SUCCESSFUL	DELEGATED

If auth was SUCCESSFUL, the local QM database is checked for that user name. If such a user is present, the user details, class and key information are loaded from QM. If such a user is not present, the details are taken from the ones supplied via XML-RPC.

If auth was AUTHORITATIVE, the details are taken from the ones supplied via XML-RPC. Then they are copied to the local user database (with a random password) so that although the user cannot login manually, it is possible to decode the user name for all logged operations (e.g. Updating a queue). If a user with the same name is present, credentials are forcibly updated with the authoritative credentials.

If auth was FORBIDDEN, no other check is done and the user is rejected access.

If auth was DELEGATED, the standard QM logon procedure is done.

The whole procedure is totally transparent to the user, so they do not need to know which is the authority that grants or denies access.

5.2. XML-RPC call parameters

5.2.1. Method: QMAuth.auth

Table 5.1. Input parameters

Pos	Name	Type	Comment
0	System ID	String	As set in QM; if not set: blank
1	Username	String	
2	Password	String	

Table 5.2. Output block

Pos	Name	Type	Comment
0	Status code	String	One of: * A (authoritative) * S (success) * F (forbidden) * D (delegate) If other, behaves as D.

Pos	Name	Type	Comment
1	Real name	String	
2	Email	String	
3	Class name	String	Must match an existing class
4	User keys	String	

The following values are implied in QM:

- Enabled = yes

- Masterkey = no

The actual user data is only read by QM in case "A"; otherwise it's ignored, whatever is passed.

Tip: As a reference implementation, see the server that ships with QueueMetrics in the mysql-utils/xml-rpc/xmlrpc_auth_server.php file. It also contains an example of querying a LDAP server in PHP.

5.2.2. Configuration properties

A set of two new configuration properties control external auth sources in in QM:

Property	Description
default.authRpcServerUrl	If set, XML-RPC auth is used. Points to the URL of the auth server.
default.authSystemId	The system-ID for this QM license. Can be currently any user-chosen name for the system.

Chapter 6. Dynamic blocks in QueueMetrics

It is possible to have dynamic blocks generated externally that are included in QueueMetrics reports. These blocks can be then viewed or printed as any other native block.

QueueMetrics offers you three different APIs for providing external data blocks, so you can choose the method that works best for you:

- Plain HTML

- XML-RPC blocks

- JSON blocks

In any case, QueueMetrics will call the external server and provide it with the parameters of the current report. These parameters are set as follows:

- *queues*: The queue(s) that the report is running on

- *dtFrom*: Begin of report period

- *dtTo*: End of report period

- *agents*: The included set of agents

- *agent*: The agent bbeing filered on

- *locationId*: The ID of the current report location

- *agentGroupId*: The ID of the agent group

- *outcome*: The outcome being searched

- *callId*: The CallId to be searched

- *callerId*: The Caller ID of the call

- *waitDur*: The expected wait duration (as an interval, see below)

- *callDur*: The expected call duration (as an interval, see below)

- *disconnection*: The disconnection cause

- *enterPos*: The expected position when the call was queued (as an interval, see below)

- *attempts*: The number of attempts (as an interval, see below)

- *dnis*: The DNIS

- *ivr*: The IVR sequence

- *server*: The server id (for clusters)

- *supervisionOnly*: If the current report is filtered on the agents that the current user supervises

- *supervisorId*: The ID of the supervisor

- *tzOffset*: The timezone offset

- *joinMultiStint*: If multi-stint calls are to be joined or not

- *nonContig_dow*: A string with the days of week being considered. 1: Monday to 7: Sunday

- *nonContig_range1*: The first time-filtering range (as an interval, see below)

- *nonContig_range2*: The second time-filtering range (as an interval, see below)

- *queueSet*: The expanded set of queues included (as a pipe-separated sequence)

- *agentSet*: The expanded set of agents included

Parameters are passed only if they are present in your search query.

If the parameter is an interval, it will be passed as two separate items, one called MIN and the other MAX. For example, the *enterPos* parameter is passed as *enterPosMin* and *enterPosMax*.

String parameters starting with a ^ are to be interpreted as regular expression on the field.

the parameters *queueSet* and *agentSet* will include the actual items to be reported on. So you do not have to know which agent an user supervises, or which agents belong to a specific agent group.

6.1. Plain-HTML data sources (method XS01)

By specifying a *url* parameter to the data block XS01, the server specified will be called as an HTTP GET request.

The resulting page text should contain an HTML fragment that will be included in the resulting QueueMetrics page.

This is the most flexible method, as you can actually generate and include any kind of content, including images and JavaScript. The content shall be styled by the server generating it.

6.2. XML-RPC data blocks (method XS02)

The XS02 method will invoke an XML-RPC server specified as its *url* parameter. The server will be queried for a method called "QMDataSource.get" that receives a single Hash as its sole parameter. The hash contains report parameters as specified above.

The result shall be a List of exactly two element, containing:

- In position zero, a Table of all columns to be rendered.

- In position one, a Table containing data to be rendered (that might be empty, that is will contain an empty list)

We define a Table as a List of Lists of Strings.

So e.g. a data structure like:

```
Table 0:
 -----------------------
 |Column  1  |Style 1  |
 |Column  2  |Style 2  |
 |Column  3  |Style 3  |
 -----------------------


Table 1:
 ------------------------------------------
 |Value 1      |Value 2      |Value 3      |
 |...          |....         | ...         |
 ------------------------------------------
```

will be rendered in QueueMetrics as:

```
 +------------------------------------+
 |Column 1     |Column 2     |Column 3     |
 |-----------+-----------+-----------|
 |Value 1      |Value 2      |Value 3      |
 |...          |....         | ...         |
 +-----------+-----------+-----------+
```

The style for each column is a string like "width=150 type=text align=left" - see below for specification.

An example XML-RPC server written in PHP is available within the QueueMetrics samples under the name *extDataSource_xmlrpc.php*.

6.3. JSON data blocks (method XS03)

By specifying a *url* parameter to the data block XS03, the server specified will be called as an HTTP GET request.

The resulting page text should contain a JSON data structure that will be parsed in order to get column and data information.

```
{
    "headers": [
        { "title": "Column 1", "style": "width=150 type=text" },
        { "title": "Column 2", "style": "width=50 type=num" },
        { "title": "Column 3", "style": "width=50 type=text" }
    ],
```

```
    "data": [
        [ "a", "0", "c" ],
        [ "d", "1", "f" ]
    ]
}
```

In the example above, we have three columns - the first is a text column, the second an integer one, the third again some text. Please not that in any case the data table contains only text.

> If the returned data structure should not match the definition above, an error will be displayed.

An example JSON server written in PHP is available within the QueueMetrics samples under the name *extDataSource_json.php*.

6.4. Styling return values for XML-RPC and JSON data blocks

Styling information is passed as a string that allows the following parameters:

- *width*: The width in pixel for this column

- *type*: The type of the column.

  - *text*: Plain text

  - *ltext*: A localization key that will be substrituted for the current locale

  - *num*: An integer number

  - *decnum*: A decimal number, with the default decimal digits.

  - *perc*: A percentage. For example, "0.2345" will be rendered as "23.4%"

  - *html*: Some HTML text

  - *datetime*: A date and time, according to the current locale

  - *dtshort*: A shorter date and time, according to the current locale

  - *time*: A time, according to the current locale

  - *hhmmss*: An integer number of seconds expressed as hours:mins:secs. E.g. "68" will be expressed as "1:08".

- *align*: The textual alignment of the cell

  - left

  - right

  - center

Values to be rendered as dates and times must be passed in the format "2011-10-11.12:34:56".

> Column headers can be localized as well. Just include them between hashes, e.g. "clok_total_call_length".

Appendix A. Response block names

Each XML-RPC call will have its own admissible set of block names that can be asked for. Response blocks come in families, also known as Data Objects. Each response block name is made up of the Data Object name dot the method name, as in KoDO.DiscCauses. Remember that response block names are case sensitive.

A.1. Response block names for QM.stats

A complete list of possible QueueMetrics blocks is now maintained in the QueueMetrics User Manual, chapter 6 "Report Details". The User Manual can be obtained from the QueueMetrics website.

For every possible block there is a name, a description, a "shortcut code" for ease of identification and an XML-RPC code. That is the name of the block that has to be retrieved over XML-RPC.

> For example, if we want to access the "Disconnection Causes" block, we will look it up in the manual until we encounter "UN03 - Disconnection Causes".
>
> We see that its XML-RPC code is "KoDO.DiscCauses", so that is the name of the block we'll be asking for. Block names are case-sensitive, so make sure you are writing it as it is on the User Manual.

A.2. Response block names for QM.realtime

A.2.1. DataObject: RealtimeDO

Real-time information, as displayed in the main QM real-time page, using system defaults.

Method	Description
RTRiassunto	An overview table of the queues in use.
RTCallsBeingProc	Calls being processed in real-time
RTAgentsLoggedIn	Agents logged in and paused

A.3. Response block names for QM.qareport

A.3.1. DataObject: QualAssDO

Quality Assessment information, as displayed in the QA report page.

Method	Description
TrkCalls	Tracked calls per agent report
TrkCallsQ	Tracked calls per queue report
CallSupervs	Supervisors tracking calls report
Res1	Section 1 (as defined in the form) calls by agent report
Res1Q	Section 1 (as defined in the form) calls by queue report
Res2	Section 2 (as defined in the form) calls by agent report
Res2Q	Section 2 (as defined in the form) calls by queue report
Res3	Section 3 (as defined in the form) calls by agent report
Res3Q	Section 3 (as defined in the form) calls by queue report
Res4	Section 4 (as defined in the form) calls by agent report
Res4Q	Section 4 (as defined in the form) calls by queue report
Res5	Section 5 (as defined in the form) calls by agent report
Res5Q	Section 5 (as defined in the form) calls by queue report
Res6	Section 6 (as defined in the form) calls by agent report
Res6Q	Section 6 (as defined in the form) calls by queue report
Res7	Section 7 (as defined in the form) calls by agent report
Res7Q	Section 7 (as defined in the form) calls by queue report
Res8	Section 8 (as defined in the form) calls by agent report
Res8Q	Section 8 (as defined in the form) calls by queue report

Method	Description
Res9	Section 9 (as defined in the form) calls by agent report
Res9Q	Section 9 (as defined in the form) calls by queue report
Res10	Section 10 (as defined in the form) calls by agent report
Res10Q	Section 10 (as defined in the form) calls by queue report
AgentDetail	Tracked calls details for each defined agent

A.4. Response block names for QM.qaformreport and QM.qaformgrading

A.4.1. DataObject: QualAssFormDO

Quality Assessment information related to QA Forms.

Method	Description
FormStructure	The data structure of specified form
SectionValues	Raw QA values for each section in forms matching the query
Comments	Comments associated to forms matching the query

A.5. Response block names for QM.qaformsummary

A.5.1. DataObject: QualAssDO

Quality Assessment information related to QA Forms.

Method	Description
OverallAverageFormReport	Aggregated information for the overall specified form (scoring and not scoring questions included)
FormSummary	Aggregated information for the specified form (only scoring questions)
ScoringItemsFormSummary	Aggregated information for the specified form (only scoring questions, same as FormSummary)
NonScoringItemsFormSummary	Aggregated information for the specified form (only non scoring questions)

A.6. Response block names for QM.qacallstograde

A.6.1. DataObject: QAGradingDO

Quality Assessment information related to calls to be graded

Method	Description
qagExtendedProposals	Set of calls to be graded and related information

A.7. Response block names for QM.auth

The QM.auth call will return one single block named "auth".

This block contains the following information:

- UserName: the login name

- Status: OK if authentication was passed or ERR if it was not passed

- FullName: The user's full name

- Email: The user's email address

- Class: The name of the class the user belongs to

- Keys: The active key set of the user, that is, all keys given to the class plus or minus the keys that have been granted or revoked to this specific user

- Masterkey: If set to 1, this user has a Masterkey, so this user will pass each key ckeck

- NLogons: The number of logons the user has made. Each successful QM.auth call counts as a logon.

A.8. Response block names for QM.findAudio

The QM.findAudio call will return one single block named "AudioFiles".

This block contains the following information:

- Column 0: *Filename*. A file name of the audio file

- Column 1: *URL*. The download URL for this audio recording.

If no calls are found, the block has zero rows.

A.9. Response block names for QM.insertRecordTag

The QM.insertRecordTag call will return one single block named "TagRecords".

This block contains the following information:

- Column 0: *TagID*. A technical ID associated to each tag

- Column 1: *CallID*. The QueueMetrics Unique ID associated to the call (in the format asteriskId@server [mailto:asteriskId@server])

- Column 2: *Title*. The recording filename the tag is referring

- Column 3: *Color*. A decimal representation of the tag color

- Column 4: *Time*. The tag start time (in seconds)

- Column 5: *Duration*. The tag duration (in seconds) or 0 if no duration was specified

- Column 6: *Message*. The tag message

- Column 7: *Optilock*. Optilock field in the database. Reserved.

- Column 8: *UserID*. The technical user ID for the user who added the tag

- Column 9: *CreationDate*. The add operation timestamp

If no tags are found, the block has zero rows.

Chapter 7. Appendix II: A short list of XML-RPC libraries

To access QueueMetrics, you only need a library with Client capabilities. Server capabilities are not needed. The following list is by no means exhaustive of all available implementations:

Perl
The BlackPerl library is available at http://www.blackperl.com/RPC::XML/

Python
The xmlrpclib ships with any modern version of the language.

JavaScript
The JSON-XML-RPC library can be found at: http://code.google.com/p/json-xml-rpc/

JavaScript (2)
The Jsolait library also offers an XML-RPC module: http://jsolait.net/

Java
There are a lot of implementations available for Java, we recommend Redstone's LGPL library - http://xmlrpc.sourceforge.net/

C / C++
See http://xmlrpc-c.sourceforge.net/

C# / .Net
A connector is available at http://www.xml-rpc.net/

VisualBasic
A COM component that will work on most languages on the Windows platform: http://sourceforge.net/projects/comxmlrpc

PHP
The package *phpxmlrpc* can be downloaded from http://phpxmlrpc.sourceforge.net/

Ruby
The module *xmlrpc* is included in the Ruby Standard Library