IIQ Console Command Reference

IdentityIQ Version: 5.5

The information in this white paper has been merged into the IdentityIQ Administration Guide. Future updates to this content will be made only in that document; this white paper will not be updated for future releases.

This document is a command reference for the iiq console command line utility for IdentityIQ.

Table of Contents

Introduction

The IIQ Console is the command line utility for interfacing with IdentityIQ. It is a powerful tool that allows the user to view objects, execute workflows, import and export data, and much more. This document lists all the available console commands and describes how they can be used.

Launching the Console

The IdentityIQ Console (iiq console) is launched by executing the iiq.bat file found in the [IdentityIQ Installation Directory]/WEB-INF/bin directory. From a command prompt, launch the console with the command as shown for each operating system type:

Windows	iiq console
Unix	./iiq console -j

NOTE: The -j option turns on the JLine Java library for handling console input, enabling some ease-of-use functions such as command history recall. Command history recall is enabled in Windows without this library, so this parameter is not required in the Windows environment.

NOTE: The iiq console requires command line execution privileges, so System Administrator rights on the server where IdentityIQ is installed are required to run the iiq console.

The ">" prompt character indicates that the console is running and ready to accept commands.

Viewing the List of Commands

The help command displays a list of all commands available in the console along with a short description of each. At the command prompt, enter **help** or **?** to see this full list of available commands.

```
>help

Console Commands

?                   display command help
help                display command help
quit                quit the shell (same as exit)
exit                exit the shell (same as quit)
source              execute a file of commands
properties          display system properties
time                show how much time a command takes to run.
xtimes              Run a command x times.
about               show application configuration information
threads             show active threads
logConfig           reload log4j configuraton

Objects

dtd                 create dtd
summary             summarize objects
classes             list available classes
list                list objects
```

```
count                 count objects
get                   view an object
checkout              checkout an object to a file
checkin               checkin an object from a file
delete                delete an object
rollback              rollback to a previous version
rename                rename an object
import                import objects from a file
importExplanations    import explanations from a CSV file
export                export objects to a file
exportExplanations    export explanations to a CSV file
exportJasper          Exports only the jasperReport xml contained in a JasperTemplate
                      object.

Identities

identities            list identities
snapshot              create an identity snapshot
score                 refresh compliance scores
listLocks             list all class locks
breakLocks            break all class locks

Tasks

tasks                 display scheduled tasks
run                   launch a background task
runTaskWithArguments  launch a task synchronously with arguments
terminateOrphans      detect and terminate orphaned tasks
startScheduler        start the task scheduler
stopScheduler         stop the task scheduler
status                display system status

Certifications

certify               generate an access certification report
cancelCertify         cancel an access certification report
archiveCertification  archive and delete an access certification report
decompressCertification decompress an access certification archive

Groups

refreshFactories      refresh group factories (but not groups)
refreshGroups         refresh groups (but not factories)
showGroup             show identities in a group

Workflow

workflow              start a generic workflow
validate              validate workflow definition
workItem              describe a work item
approve               approve a work item
reject                reject a work item

Tests

test                  Run a mysterious test.
rule                  Run a rule.
parse                 Parse an XML file.
warp                  Parse an XML object and print the reserialization.
notify                Send an email.
authenticate          Test authentication.
simulateHistory       Simulate trend history.
search                Run a simple query
```

```
certificationPhase    Transition a certification into a new phase.
impact                Perform impact analysis
event                 Schedule an identity event
expire                immediately expire a workitem that has an expiration configured.  If
                      the workitem is type Event it'll also push the event forward with the
                      workflower.
connectorDebug        Call one of the exposed connector method using the specified
application.
encrypt
sql
hql
updateHql
date
shell
meter
compress
uncompress
clearEmailQueue       Remove any queued emails that have not been sent.
provision             Evaluate a provisioning plan.
lock                  lock an object
unlock                break a lock on an object
showLock              show lock details
clearCache            clear the object cache
```

Figure 1: List of Console Commands

Command Usage Syntax

The usage syntax for any console command that requires parameters can be determined by entering that command with no arguments.

```
> workflow
usage: workflow <name> [<varfile>]
```

Figure 2: Command Usage Syntax

NOTE: Command names are case sensitive and must be entered as shown in the command list. Parameters, however, are not case sensitive and will be matched to system objects, classes, etc. regardless of case.

Some commands take no arguments and will execute if entered at the command prompt. This table contains a list of the commands that require no arguments.

Command	Action
? or help	Lists all available console commands
quit or exit	Exits the console shell
classes	Lists all classes
refreshGroups	Refresh group indexes (Optional group name or ID can be specified)
refreshFactories	Refresh set of GroupDefinitions for a GroupFactory (Optional factory name or ID can be specified)
logConfig	Reloads log4j configuration from log4j.properties file
summary	Lists all classes and the count of objects of that class in the system
properties	Displays Java properties of the server where IdentityIQ is installed
about	Displays application configuration information
threads	Shows a list of active threads

tasks	Writes a list of all currently scheduled tasks, in a columnar layout, to the console (stdout)
identities	Writes the Name, Manager, Roles, and Links for each Identity in the system to the console (stdout)
startScheduler	Starts the task scheduler (optional parameters can be specified to start request scheduler)
stopScheduler	Stops the task scheduler
date	Displays the current system date/time and its UTIME (universal time) value (Optional UTIME parameter causes the command to display the date/time corresponding to the provided UTIME value.)
status	Reports current running status of the task and request schedulers
meter	Toggles metering on and off; while metering is on, the console reports some timing statistics for each command executed. Meter information is displayed after the results of each command as it is executed.
clearEmailQueue	Deletes all queued but unsent email messages
clearCache	Clears the IdentityIQ object cache

Syntax for Redirecting Command Output

Most of the commands report data or error messages to the console or standard out (stdout) for the system. The output for any command can be redirected to a file by specifying "**><filename>**" at the end of the command.

This example redirects the output from the get command to a file:

```
> get identity Adam.Kennedy > c:\output\AdamKennedyID.xml
```

Console Commands

In this document, the list of console commands is subdivided based on how frequently they are likely to be used in a production environment. There is a relatively short list of commands that most system administrators need to know well and will use fairly frequently; these are listed as **Commonly Used Commands**. Following that are the **Less Commonly Used Commands** – a set of commands that have some usefulness to a system administrator and may need to be run periodically (such as when working with SailPoint Support to resolve an issue), but are not ones that will be used with great regularity. Finally, a large number of the commands serve specific purposes that are useful to a developer but are not very useful for a system administrator in a production environment; these are grouped here as the **Seldom Used Commands**.

Commonly Used Commands

This section lists and documents the syntax and actions of the most commonly used console commands.

Help and ?

These two commands list all the available console commands.

Syntax	? help
Examples	> ? > help
Result	Lists all commands available in the console

Exit and Quit

These two commands exit the console shell, returning the user to the operating system command prompt.

Syntax	exit quit
Examples	> exit > quit
Result	Exits console shell and returns user to the operating system command prompt

Source

The **source** command runs commands from a script file. The commands on each line in the file will be executed by the console sequentially.

Syntax	source <filename>
Example	> source c:\data\cmdfile.txt
Result	Runs the console commands in the c:\data\cmdfile.txt file sequentially

List

The **list** command lists all objects of the specified class, constrained by any specified filter. If this command is specified without arguments, the command syntax is displayed, followed by a list of all available classes whose objects can be listed. This is helpful in locating objects within the system and in identifying object names to use as parameters on other commands.

Syntax	list <class name> [<filter>] filter: xxx – names beginning with xxx xxx* - names beginning with xxx *xxx – names ending with xxx *xxx* - names containing xxx
Example	> list application ent*
Result	Lists all application objects whose names begin with “ent”

Get

The **get** command displays the XML representation of the named object.

Syntax	get <class name> <object name or ID>
Example	> get identity Adam.Kennedy
Result	Displays the Adam.Kennedy Identity in XML format

NOTE: This command only displays the object to the console (stdout); it does not export the object. However, the output can be redirected to a file if the user has write access to the server's file system.

```
> get identity Adam.Kennedy > c:\output\AdamKennedyID.xml
```

Other alternatives for getting the XML representation of an object into a text file include:

- copying and pasting contents of this command's stdout into a text file
- retrieving the object's XML from the IdentityIQ Debug pages
- using the checkout command (described next) to write the XML representation of an object to a text file

Checkout

The **checkout** command writes a copy of the XML representation of the requested object to the specified filename. The file can be used for review or for moving objects from one environment to another (e.g. from the user acceptance testing environment to production). Organizations doing custom development on rules, workflows, etc. may use checkout to extract any of these objects to a file for modification.

Syntax	`checkout <class name> <object name or ID> <file> [-clean [=id,created…]]`
Example	`> checkout rule "Cert Signoff Approver" certrule.xml`
Result	Writes a copy of the Cert Signoff Approver rule's XML representation to the file certrule.xml

The **-clean** option can be used to remove all values that do not transfer between IdentityIQ instances, such as created and modified dates as well as globally unique ID values (GUIDs). Specifying the **–clean** option with no qualifiers (e.g. **–clean**) cleans the "id", "created", "modified", and "lastRefresh" attributes. The **–clean** option can also be used to explicitly clear specific fields by name; the fields to clear must be listed in a comma separated list (e.g. **–clean=id,modified**).

Checkin

The **checkin** command reads a file containing an object's XML representation and saves the object into the database. If the object is a workItem, the command invokes the workflower to process the workItem. If the object is a bundle (role) and the "approve" parameter is specified, a role approval workflow is launched. For all other object types (and for bundles that are submitted without the "approve" parameter), the object is saved into the database.

NOTE: The command's syntax parsing allows the "approve" parameter to be specified for any object but it will only affect the processing on Bundle objects.

Syntax	`checkin <filename> [approve]`
Example	`> checkin bobSmithID.xml` `> checkin newRole.xml approve`
Result	First example saves Identity Bob Smith, as represented by the XML in bobSmithID.xml, into to the database; overwrites existing or adds new record Second example launches an approval workflow for the bundle object represented by the XML in newRole.xml

NOTE: If an Import file (one beginning with a <Sailpoint> element) is specified as the input file for this command, only the first object in the file will be checked in; the rest will be ignored and a warning message will be displayed to the console (stdout).

Delete

The **delete** command deletes the named object and removes all of its owned, or subordinate, objects. In a production environment, this is generally not recommended unless specifically directed by SailPoint Support. **This action cannot be undone and should be used with extreme caution and only in rare circumstances** (e.g. to delete a corrupted object that has been determined to be unfixable through any other means)**.**

Syntax	`delete <class name> <object name or ID>`
Example	`> delete identity bob.smith`
Result	Removes Identity Bob Smith and all of his associated Link, Scorecard, EntitlementGroup, etc. objects from the system

NOTE: Wildcards can be used on the <object name or ID> argument:

- * - all objects of the specified class **(use with extreme caution!)**
- xxx* - all objects whose name or ID starts with xxx
- *xxx – all objects whose name or ID ends with xxx
- *xxx – all objects whose name or ID contains xxx
- xxx – a single object whose name starts with xxx (fails if more than one object's name starts with the specified characters)

Import

The **import** command imports objects into IdentityIQ from an XML file. This command can be used on a file that contains a Jasper report, a SailPoint import file, or an object of one of the standard object classes. The file contents are evaluated and processed based on the first tag in the file:

- <JasperReport>: Jasper report
- <SailPoint>: SailPoint import object; may contain multiple regular objects in one file as well as an ImportAction tag that directs how the contents of the file are processed (e.g. merge, include, execute, logConfig)
- Anything else: assumed to be a single regular object

Syntax	`import <filename>`
Example	`> import init.xml`
Result	Imports the contents of the file "init.xml" into the IdentityIQ database (This action is a normal part of the initialization process for IdentityIQ when it is first set up at a customer site.)

This is one of the most commonly used commands. Installations who manage their workflows and rules in an external source code control system, for example, use this command to bring changes to those objects into IdentityIQ once they have been modified in their external XML representations.

Export

The **export** command writes all objects of a given class to a specified filename. This is commonly used in gathering objects from IdentityIQ to deliver to SailPoint Support as resources in resolving tickets. It is also often used for moving sets of objects between environments and for managing objects outside of IdentityIQ (such as storing workflows and rules in a source code control system).

More than one class can be exported at a time to the same file by specifying all the desired class names as arguments to the command. If the export command is specified without any class names, all objects of all classes are exported to the specified filename.

Syntax	`export [-clean[=id,created...]] <filename> [<class name> <class name> …]`
Example	`> export –clean workflows.xml workflow` `> export IdLink.xml identity link`
Result	The first example exports the entire set of workflow objects from IdentityIQ to the file workflows.xml, removing values from the id, created, modified, and lastRefresh attributes. The second example exports all identities and links to a single file (IdLink.xml).

ListLocks

The **listLocks** command lists all locks held on any objects of the named class. At this time, Identity is the only class for which this command operates.

Syntax	`listLocks <class name>`
Example	`> listLocks identity`
Result	Lists the names of all Identities that are currently locked, as well as the locking user, the date/time the lock was established, and the date/time at which the lock expires for each. If the lock is expired, it is still shown in the list but is indicated as an expired lock.

BreakLocks

If a process is holding a lock but is unable to perform the required action, the lock may cause problems in other processes' performance as well. The **breakLocks** command can be used to release locks forcibly. (**NOTE**: The unlock command can be used to break a single lock whereas this command breaks all locks held on any object in the specified class.) At this time, Identity is the only class for which this command functions.

Syntax	`breakLocks <class name>`
Example	`> breakLocks identity`
Result	Releases all locks held on any identity object in the system and reports to the

	console (stdout) the identity name, lock holder, and UTIME value for the lock date/time and the lock expiration date/time.

NOTE: This command should be used with caution. Locks are useful in maintaining data integrity, and breaking them at the wrong time can potentially permit conflicting updates that may result in data corruption.

Rule

The **rule** command runs a rule defined in the system. The rule to run is specified as a command parameter. If any input variables must be passed to the rule, they must be entered in a variable file, specified as an XML Map; the file name is then also passed as a parameter to the command.

This command can be used for testing or executing existing system rules. It can also be used to run any beanshell code snippet against the IdentityIQ database: the code is created as a rule and loaded into the system and then executed from the console. Support sometimes uses rules like this to help with data cleanup.

Syntax	`rule <rule name or ID> [<varfile>]`
Example	`> rule "Check Password Policy" c:\data\pwdParams.xml`
Result	Runs the Check Password Policy rule, passing its input variables through the file c:\data\pwdParams.xml

Parse

The **parse** command validates an XML file. If it is in valid form and its tags match the IdentityIQ DTD, it runs successfully and no information is printed to the console (stdout). If any errors are encountered, a runtimeException is printed to the console describing the error.

Syntax	`parse <filename>`
Example	`> parse c:\data\newWorkflow.xml`
Result	Validates the XML in the file c:\data\newWorkflow.xml and reports any errors to the console (or reports nothing if it is valid)

Less Commonly Used Commands

These commands are not frequently used in a production environment. However, it is helpful to understand and be able to use them when the need arises.

DTD

The DTD command writes the IdentityIQ DTD (Document Type Definition) to the specified file.

Syntax	`dtd <filename>`
Example	`> dtd c:\DTD\identityIQ.dtd`
Result	Writes the IdentityIQ DTD to the file c:\DTD\identityIQ.dtd

Classes

The **classes** command lists all classes accessible from the console. These are frequently used as parameters to other commands so this list can be helpful in entering correct arguments on those commands.

Syntax	`classes`
Example	`> classes`
Result	Lists class names for all classes accessible to the console

Count

The **count** command returns a count of the objects of the specified class.

Syntax	`count <class name>`
Example	`> count identity`
Result	Displays the count of Identity objects in the system

ImportExplanations

The **importExplanations** command is used to import descriptions for managed attributes/entitlements from a CSV file.

Each line in the file is expected to contain these values:

<type>,<attribute>,<attribute display name>,<attribute value>,<description or explanation>,<managed attribute owner>,<requestable flag(True/False)>

Example:

Entitlement,memberOf,Administrators,"CN=Administrators,CN=Roles,DC=sailpoint,DC=com","System Administrators Group Member","Alan.Bradley",True

Syntax	`importExplanations <application name | connector FQN> <language code> <filename>`
Example	`> importExplanations ADAM default c:\data\explanations.csv`
Result	Imports the explanations for managed entitlements on the ADAM application from the file c:\data\explanations.csv

NOTE: The application can be identified by application name or by the fully qualified connector name. The language code is defined in the ManagedAttribute ObjectConfig; "default" is the default language code in the object config.

ExportExplanations

The **exportExplanations** command exports the managed entitlements' descriptions to a CSV file. This is useful for making mass changes to the explanations or for review them as a group by collecting all the explanations in one file.

Syntax	`exportExplanations <application name | connector FQN> <language code> <filename>`
Example	`> exportExplanations ADAM default c:\data\explanations.csv`

Result	Exports the explanations for managed entitlements on the ADAM application to the file c:\data\explanations.csv

NOTE: The application can be identified by application name or by the fully qualified connector name. The language code is defined in the ManagedAttribute ObjectConfig; "default" is the default language code in the object config.

Run

The **run** command starts execution of a task that requires and accepts no arguments. Three optional parameters can be specified for this command: trace, profile, and sync.

- **Trace** writes to the console (stdout) a trace of what happens as the task is run (depending on how the task's tracing code is written).
- **Profile** displays the timing of certain phases of the task (again, the details displayed depend on the task's profile code).
- **Sync** runs the task in synchronous execution mode, as opposed to scheduling it to run in background. If sync is specified, the control will return to the console only after the task has completed and any error messages will be written to the console. If sync is not specified, the task will be launched in the background and the results of the task will be viewable in the taskResults object (accessible from the console or from the UI under **Manage** -> **Tasks** -> **Task Results**).

Syntax	`run <task name> [trace] [profile] [sync]`
Example	`> run "Refresh Risk Scores"`
Result	Runs the Refresh Risk Scores task in background

RunTaskWithArguments

The **runTaskWithArguments** command starts execution of a task that requires arguments. These tasks are always run in synchronous execution mode. This can only be used for tasks that accept arguments of simple data types; specifying an object as an argument is not possible here.

Syntax	`runTaskWithArguments <task name> [arg1=val1 arg2=val2 …]`
Example	`> runTaskWithArguments "Identity Refresh" refreshLinks=True promoteAttributes=False`
Result	Runs the Identity Refresh task, refreshing Links for the Identities

RefreshFactories

The **refreshFactories** command can be specified with no arguments to refresh all group factories or with a specific GroupFactory name to refresh just that group factory. Refreshing the group factory means identifying the group values that define each of the groups (or GroupDefinition objects). It does not refresh the list of Identities that make up each group or the statistics gathered for each group – just the list of groups themselves.

Syntax	`refreshFactories [<factory name or ID>]`

Example	`> refreshFactories`
Result	Refreshes the GroupDefinition list associated with each GroupFactory in the system

RefreshGroups

The **refreshGroups** command refreshes the group indexes for all groups or for the specified group if one is named as a command argument. The group indexes are collections of statistics for identities that are part of the group; the statistics include number of members, number of certifications due for certification owners in the group, number of certifications owned and completed on time by members of the group, and risk score information for members of the group. RefreshGroups only applies to GroupDefinitions that are indexed (attribute indexed="True").

Syntax	`refreshGroups [<group name or ID>]`
Example	`> refreshGroups`
Result	Refreshes index information for all indexed groups

ShowGroup

The **showGroup** command shows the membership (Identities) of the group named as an argument to the command.

Syntax	`showGroup <group name or ID>`
Example	`> showGroup Finance`
Result	Lists all Identities who are members of the Finance group (i.e. meet the criteria specified in the Finance GroupDefinition object)

Workflow

The **workflow** command launches the workflow specified as a command parameter. If any input variables are to be passed to the workflow, they must be entered in a variable file, specified as an XML Map; the file name is then also passed as a parameter to the command. When the workflow is successfully launched, the XML for the workflowCase is printed to the console (stdout).

Syntax	`workflow <workflow name or ID> [<varfile>]`
Example	`> workflow "LCM Provisioning" c:\data\provFile.xml`
Result	Runs the LCM Provisioning workflow, passing its input variables through the file c:\data\provFile.xml

Validate

The **validate** command can validate a workflow or a rule. If any input variables are to be passed to the workflow or rule, they must be entered in a variable file, specified as an XML Map; the file name is then passed as a parameter to the command. If any validation errors are encountered, they are printed to the console (stdout). If no violations are found, nothing will be printed to the console.

Syntax	`validate <rule or workflow name or ID> [<varfile>]`
Example	`> validate "LCM Provisioning" c:\data\provFile.xml`
Result	Validates the LCM Provisioning workflow, passing its input variables through the file c:\data\provFile.xml, and displays any validation errors encountered to the console (or displays nothing if workflow is valid)

SQL

The **sql** command executes a SQL statement. It can execute SQL specified inline with the command or it can read the SQL from a file. Whether inline or in a file, only one SQL statement can be executed at a time. The output can be printed to the console (stdout) or can be redirected to a file. Select, update, and delete SQL statements can be executed. The update/delete abilities of this command should be used with great caution since these actions cannot be undone.

Syntax	`sql <sql statement> | -f <inputFileName>`
Example	`> sql "select * from sptr_identity" > c:\data\Identities.dat` `> sql -f c:\sql\SelectIdentities.sql`
Result	The first example executes the specified select statement and writes the results to the file c:\data\Identities.dat. The second example reads the SQL from the file c:\sql\SelectIdentities.sql, prints the SQL to the console (stdout), and displays the query results of the query to the console (stdout).

Provision

The **provision** command processes the specified provisioning plan for the specified identity but does not save the information. This is used to test a connector or to test a provisioning plan. If any errors are encountered, they are reported to the console. If the provisioning action would succeed, nothing is reported to the console.

Syntax	`Provision <identity name or ID> <provisioning plan filename>`
Example	`> provision Adam.Kennedy c:\data\provFile.xml`
Result	Tests the provisioning plan contained in c:\data\provFile.xml against Identity Adam.Kennedy and reports any exceptions to the console (or reports nothing if no errors are encountered)

Lock

The **lock** command obtains a persistence lock on an object. The object's class and ID or name must be specified. By default, the lock will be issued to the username "Console" but a different username can be specified in the command's lockName parameter. The lock automatically expires after 5 minutes.

Syntax	`lock <class name> <object ID or name> [<lockName>]`
Example	`> lock identity John.Smith`
Result	Obtains a persistence lock for Console on the identity record for John.Smith

NOTE: At this time, Identity objects are the only objects that can be locked. Attempts to specify a different object type in this command will result in a syntax error exception.

Unlock

The **unlock** command releases the lock on an object. The object's class and ID or name must be specified. If the object is not locked, the message "Object is not locked" is displayed. If it is locked, the lock is released and the message "Lock has been broken" is displayed.

Syntax	`unlock <class name> <object ID or name>`
Example	`> unlock identity John.Smith`
Result	Breaks the lock on the identity record for John.Smith

ShowLock

The **showLock** command lists the lock owner, locked date/time, and lock expiration date/time for a locked object. The object's class and ID or name must be specified to view its lock information. If the object is not currently locked, the message "Object is not locked" is displayed. If the lock has expired, the lock information is shown but is prefaced with the message "Lock has expired."

Syntax	`showLock <class name> <object ID or name>`
Example	`> showLock identity John.Smith`
Result	Displays lock information (owner, date/time, expiration date/time) on the identity record for John.Smith or displays "Object is not locked" if there is no lock on that object

Seldom Used Commands

These commands are most frequently used by developers for testing and will not often be used by a system administrator in a production environment.

Properties

The **properties** command displays system properties.

Syntax	`properties`
Examples	`> properties`
Result	Displays Java properties of the server where IdentityIQ is installed

Time

The **time** command reports how much time another command takes to run.

Syntax	`time <command>`
Example	`> time run "refresh risk scores"`
Result	Runs the run command (to run the "refresh risk scores" task) and then

	indicates how much time it took to run (most useful for long-running commands)

Xtimes

The **xtimes** command repeats a single command as many times as specified in the first argument. This command is primarily used for performance testing purposes; the tester may gain a better sense of how long a process takes by running it several times in succession than by running it once, particularly if it is short-running task.

Syntax	`xtimes <x> <command>`
Example	`> xtimes 3 run "refresh risk scores"`
Result	Runs the refresh risk scores task three times in a row

NOTE: This command can be combined with the time command to report timing statistics on the performance test. By specifying this command first (e.g. xtimes 20 time run <taskname>), the time taken for each command run is reported; by specifying the time command first (e.g. time xtimes 20 run <taskname>), the total time for all of the sequential runs is reported.

About

The **about** command displays IdentityIQ's application configuration information.

Syntax	`about`
Example	`> about`
Result	Lists application configuration specifics about the IdentityIQ instance (version, database, host, memory, etc.)

Threads

The **threads** command displays all active threads in the instance.

Syntax	`threads`
Example	`> threads`
Result	Lists the active threads

LogConfig

The **logConfig** command reloads the log4j configuration into the instance.

Syntax	`logConfig`
Example	`> logConfig`
Result	Reloads the log4j configuration from the log4j.properties file

Summary

The **summary** command lists all classes and the count of objects of each class. Changes in these counts for some objects (e.g. auditConfig) can alert the administrator to potential problems or areas of concern.

Syntax	`summary`
Example	`> summary`

Result	Lists class name and count of objects for each class in the system

Rollback

The **rollback** command can undo a change to a role by restoring a role from its BundleArchive object. BundleArchive objects are only created when role archiving is enabled for the installation. Role archiving tracks changes made to a role by storing the pre-modification state in a BundleArchive object when the Bundle object is updated to reflect the role's new configuration. This command only applies to the BundleArchive class.

Syntax	`rollback <class name> <object name or id>`
Example	`> rollback BundleArchive "Contractor-IT"`
Result	Restores the Contractor-IT role to the pre-modification state stored in its BundleArchive object

Rename

The **rename** command changes the name of an object from its existing name to the value specified in the <newname> parameter.

Syntax	`rename <class name> <object name or ID> <newname>`
Example	`> rename application ADAM ADAM-Production`
Result	Changes the name of the ADAM application to ADAM-Production

NOTE: The object can be found using its old Name or its ID value, but in either case, the <newname> value will be used to update the Name attribute for the object.

ExportJasper

The **exportJasper** command creates a JasperReport XML file from a JasperTemplate object in IdentityIQ. Jasper Report is a third party user interface for report writing. JasperReport XML is not compatible with IdentityIQ's XML so the JasperReport XML is wrapped in a JasperTemplate object when saved in IdentityIQ. To use the Jasper UI to make changes to a report format, the JasperTemplate must be exported to create a file that can be used directly with the Jasper UI.

Syntax	`exportJasper <filename> <JasperTemplate name or ID>`
Example	`> exportJasper c:\data\AggResRpt.xml AggregationResults`
Result	Exports the Jasper XML from the AggregationResults JasperTemplate object into the file c:\data\AggResRpt.xml

NOTE: The **import** command can be used to re-import a JasperReport object into the database; it will wrap the XML in a JasperTemplate as it is imported.

Identities

The **identities** command lists the Name, Manager, Roles, and Links for each Identity in the system. By default, this prints to the console (stdout) and can be difficult to read due to screen wrapping. If the output is redirected to a file, it is printed in the file in an easy-to-read columnar style.

Syntax	`identities`
Example	`> identities` `> identities > identities.txt`
Result	The first example writes the Name, Manager, Roles, and Links for each Identity in the system to the console (stdout). The second example redirects that information to the file identities.txt.

Snapshot

The **snapshot** command takes a snapshot of the named Identity as it exists at that moment and archives it in the database as an IdentitySnapshot object. This object is available to provide a historical record of the state of Identity objects at various points in time. Automatic snapshotting can be enabled and configured to create IdentitySnapshot objects at specified intervals or based on system activities (weekly, on aggregation change, etc.). The configuration of this feature must be considered carefully as it can negatively affect system performance. This snapshot command is used to generate a snapshot of a single Identity on demand.

Syntax	`snapshot <identity name or ID>`
Example	`> snapshot Alan.Bradley`
Result	Creates an IdentitySnapshot object for the Identity Alan.Bradley, capturing his Identity attributes, Bundles (roles), Entitlements outside roles, Links, and Scorecard information at that moment in time

Score

The **score** command refreshes the Identity score for the named Identity and updates that score in the database (on the Identity's Scorecard). This command is seldom used; score updates are more commonly executed through the IdentityIQ user interface.

Syntax	`score <identity name or ID>`
Example	`> score Alan.Bradley`
Result	Recalculates the risk scores for Alan.Bradley and updates his Scorecard with the new risk scores

Tasks

The **tasks** command lists the Name, State, Next Execution, and Cron string(s) for all currently scheduled tasks in the system.

Syntax	`tasks`
Example	`> tasks`

Result	All currently scheduled tasks are written in a columnar layout to the console (stdout)

TerminateOrphans

The **terminateOrphans** command sets the completion status of any open taskResult object to Terminated. While tasks are running, their taskResults should be in a "pending" state, but occasionally task results can become orphaned and remain in this non-completed state while the task has finished (or has otherwise been terminated). This command can be used to clean up those "orphaned" taskResults but it must only be executed when there are no tasks running on the application server; otherwise, taskResults for those actively running tasks will be terminated along with any orphaned results.

This command requires no arguments for execution but an artificial argument "please" has been added to prevent accidental running of this command.

Syntax	`terminateOrphans please`
Example	`> terminateOrphans please`
Result	Sets all open taskResults for the application server to a "terminated" status

StartScheduler

The **startScheduler** command can start the task scheduler, the request scheduler, or both, depending on the parameters specified.

- No parameter or "task": start task scheduler
- "request": start request scheduler
- "all": start both task and request scheduler

Syntax	`startScheduler [task | request | all]`
Example	`> startScheduler all`
Result	Starts the task and request schedulers (if they are not running)

StopScheduler

The **stopScheduler** command is used to stop the task scheduler.

Syntax	`stopScheduler`
Example	`> stopScheduler`
Result	Stops the task scheduler if it is running

Status

The **status** command shows the current status of the task scheduler and request scheduler – whether running or stopped.

Syntax	`status`

Example	`> status`
Result	Reports the current running status of the task and request schedulers

Certify

The **certify** command creates a manager certification for the specified manager. The certification is generated using the installation's default settings/parameters for a manager certification. This command is primarily used for testing purposes.

Syntax	`certify <manager identity name or ID>`
Example	`> certify Catherine.Simmons`
Result	Generates a manager certification for manager Catherine Simmons

NOTE: The command syntax help indicates that this command can generate an application-owner certification when an application is specified as a command argument (instead of a manager), but this feature has not been updated as the certification components of the product have changed over time. As a result, the application argument for this command is not currently usable.

CancelCertify

The **cancelCertify** command can be used to delete a certification object from the system. This command, however, is no longer recommended; instead, use the **delete** command (delete certification <certification name>) to delete a certification object.

Syntax	`cancelCertify <certification name or ID>`
Example	`> cancelCertify "Manager Access Review for William Moore"`
Result	Delete the named certification (if more than one certification object with the same name exists, this command will fail)

ArchiveCertification

The **archiveCertification** command archives the specified certification (creates a certificationArchive object from it) and deletes it as an active certification.

Syntax	`archiveCertification <certification name or ID>`
Example	`> archiveCertification "Manager Access Review for William Moore"`
Result	Creates a certificationArchive object that contains the named certification's data and delete the certification from the system

DecompressCertification

The **decompressCertification** command retrieves the named certificationArchive object and prints it to the console (stdout) in the normal Certification object's XML format. This is a friendlier view of the certification data than the Get command on the CertificationArchive would produce.

Syntax	`decompressCertification <certificationArchive name or ID>`

Example	`> decompressCertification "Manager Access Review for William Moore"`
Result	Prints the named certification archive to the console (stdout) in certification XML format

WorkItem

The **workItem** command displays certain details (owner, create date, expiration date) for the specified workItem.

NOTE: This command requires the workItem ID or name value as an input parameter. The workItem ID value (a long hexadecimal number) can be obtained through the console's "list workItem" command. The workItem Name is not the descriptive name for the workitem; it is another numeric value, assigned when the workItem is created, that can be found in the XML representation of each workItem through the Debug pages.

Syntax	`workItem <workItem ID or name>`
Example	`> workItem 40288f0132b155ad0132b58a4e3f018e`
Result	Displays the Owner, Created date, and Expiration date for the specified workItem

Approve

The **approve** command sets the specified workItem to a "Finished" state (indicating it was "approved"), adds any specified completion comments to the workItem, and submits the workItem to the workflower to move it to the next appropriate stage.

NOTE: This command requires the workItem ID or name value as an input parameter. The workItem ID value (a long hexadecimal number) can be obtained through the console's "list workItem" command. The workItem Name is not the descriptive name for the workitem; it is another numeric value, assigned when the workItem is created, that can be found in the XML representation of each workItem through the Debug pages.

Syntax	`approve <workItem ID or name> [<comments>]`
Example	`> approve 40288f0132b155ad0132b58a4e3f018e "Access approved"`
Result	Marks the specified workItem as approved, adds the comment "Access approved" to the workItem's completion comments, and submits the workItem for evaluation of the next appropriate step (another approval, provisioning, etc.)

Reject

The **reject** command sets the specified workItem to a "Rejected" state, adds any specified completion comments to the workItem, and submits the workItem to the workflower to move it to the next appropriate stage.

NOTE: This command requires the workItem ID or name value as an input parameter. The workItem ID value (a long hexadecimal number) can be obtained through the console's "list workItem" command. The workItem

Name is not the descriptive name for the workitem; it is another numeric value, assigned when the workItem is created, that can be found in the XML representation of each workItem through the Debug pages.

Syntax	`reject <workItem ID or name> [<comments>]`
Example	`> reject 40288f0132b155ad0132b58a4e3f018e "Access conflicts with AP data entry entitlement"`
Result	Marks the specified workItem as rejected, adds the comment "Access conflicts with AP data entry entitlement" to the workItem's completion comments, and submits the workItem for evaluation of the next appropriate step (another approval, etc.)

Test

The **test** command is used by SailPoint developers only and is not useful at customer installations. It is manipulated to perform tests of code segments during dev but cannot be modified in the field to test a specific installation's functionality; that would require modifying the IdentityIQ source code. This console command can be ignored.

Warp

The **warp** command parses an XML file to create an object and then displays the object's XML representation to the console (stdout). If it is not in valid form or its tags do not match the IdentityIQ DTD, a runtimeException is printed to the console describing the error.

Syntax	`warp <filename>`
Example	`> warp c:\data\newWorkflow.xml`
Result	Parses the XML in the file c:\data\newWorkflow.xml and displays the XML representation of the object to the console (unless errors are encountered, in which case it reports any errors to the console)

Notify

The **notify** command sends an email message to the specified recipient using the email template specified. This command does not accept any other parameters that can be passed to the template, so it can really only be used for templates whose messages don't rely on variable substitutions to build the content. This command is most often used for testing purposes.

NOTE: The "toAddress" argument to this command can contain an Identity name or ID or an actual email address. If it contains an Identity name or ID, the email address will be retrieved from the Identity record.

Syntax	`notify <emailTemplate name or ID> <toAddress>`
Example	`> notify Certification Alan.Bradley`
Result	Sends an email to Alan.Bradley's email address using the Certification email template

Authenticate

The **authenticate** command authenticates the provided username and password against the pass-through authentication source or the internal IdentityIQ records. If the values can be authenticated, no results are returned by the command. If the password is incorrect or the user name cannot be found, an error message will be displayed to the console (stdout) indicating the problem.

Syntax	`authenticate <username> <password>`
Example	`> authenticate Alan.Bradley s53n659#@5a!`
Result	Authenticates username Alan.Bradley and the provided password against the authentication source (pass-through or internal)

SimulateHistory

The **simulateHistory** command is used to generate a fake, randomly-generated group index or identity score history for one or more groups or Identities. This can be useful for generating test data in a development environment but is not useful in a production environment at all.

Syntax	`simulateHistory Identity|Group <group name or ID>|<identity name or ID>|all`
Example	`> simulateHistory Identity all` `> simulateHistory Group Finance`
Result	First example generates fake risk scorecards for all Identities in the system Second example generates fake groupIndex information for the Finance group

Search

The **search** command functions like a simplified SQL/HQL interface, looking up an object based on specified criteria. A single class name can be specified along with a list of all the attributes to be displayed from that class. Following the "where" keyword, search filters can be specified in name value sets. All filter values are used in a "like" comparison, so if the record's field value contains the specified value string, the record will be returned.

Syntax	`search <class name> [<Attribute name>…] where [<filter>…]` `filter: <attribute name> <value>`
Example	`> search identity name manager.name region where name kat`
Result	Returns the name, manager's name, and region for all Identities whose name contains the string "kat". e.g. Records for Katherine.Jones, John.Kato, and Tammy.Erkatz would all be returned by this search

CertificationPhase

The **certificationPhase** command transitions the specified certification to the specified phase. This command will fail if the certification is already on or past the requested phase.

NOTE: If the specified phase is not enabled for the given certification, the certification will be advanced to the next enabled phase after the requested phase. For example, if a certification has neither a Challenge n or a

Remediation phase enabled but the command requests that it be advanced to the Challenge phase, the certification will be advanced to the End phase instead.

NOTE: The certification will be sequentially advanced through all its enabled phases until it reaches or passes the requested phase. Any business logic that should occur during each phase transition (period enter rules, period end rules, etc.) will be executed during the phase advancement.

Syntax	`certificationPhase <certification name or ID> [Challenge | Remediation | End]`
Example	`> certificationPhase "Catherine Simmons Access Review" Challenge`
Result	Advances the "Catherine Simmons Access Review" certification from its current phase (Active) to the Challenge phase. If this review is not configured for a Challenge phase, it will be transitioned to the Remediation or End phase (depending on configuration).

Impact

The **impact** command reads an XML file containing a <Bundle> object and performs a role impact analysis for the role. The command first parses the XML to its object form. If that object is not a Bundle, the impact analysis is not performed.

Syntax	`impact <filename>`
Example	`> impact c:\data\ContractorRole.xml`
Result	Performs a role impact analysis for the Bundle object represented by the XML in c:\data\ContractorRole.xml

Event

The **event** command schedules a workflow to run, passing in an Identity name as an argument to the workflow. By default, the workflow is scheduled 1 second after the command is issued (effectively immediately) but a delay can be specified in seconds as a command argument.

Syntax	`event <identity name or ID> <workflow name or ID> [<seconds>]`
Example	`> event Catherine.Simmons "Identity Refresh" 60`
Result	Schedules an Identity Refresh workflow to run for Catherine.Simmons one minute (60 seconds) after the command is issued

ConnectorDebug

The **connectorDebug** command is used to test a connector or troubleshoot application aggregation issues. Its "method" parameter determine what is tested and how.

Syntax	`connectorDebug <application name or ID> <method> [<methodArgs>…]`

The specific syntax for each of the "methods" is shown below.

Method	test
Purpose	Test whether a connection can be established with the application through its connector
Syntax	`connectorDebug <application name or ID> test`
Example	`> connectorDebug ADAM test`
Result	If connection is established, returns "Test Succeeded." If an error occurs, displays the exception information to the console.

Method	iterate
Purpose	Iterate through the application's account or group records
Syntax	`connectorDebug <application name or ID> iterate [account|group (default = account)] [-q (for "quiet mode")]`
Example	`> connectorDebug ADAM iterate -q` `> connectorDebug ADAM iterate account`
Result	First example iterates all account records natively in the ADAM application and returns only the count of iterated objects and how many milliseconds it took to run. Second example iterates account records natively in the ADAM application and returns a ResourceObject representation of each account to the console.

Method	get
Purpose	Test whether a connection can be established with the application through its connector
Syntax	`connectorDebug <application name or ID> get account|group <nativeIdentity>`
Example	`> connectorDebug ADAM get account "CN=Willie.Gomez,DC=sailpoint,DC=com"`
Result	Returns the XML representation of the <ResourceObject> for that nativeIdentity on the application

Method	auth
Purpose	Test pass-through authentication against the specified application (The featuresString in its application definition must contain "AUTHENTICATION".)
Syntax	`connectorDebug <application name or ID> auth username password`
Example	`> connectorDebug ADAM auth administrator Pa$$w0rd`
Result	Returns "Authentication Successful" when user is authenticated or displays the exception message to the console if authentication fails

Encrypt

The **encrypt** command can be used to encrypt a string. This command is generally only useful for test purposes. It can be used to generate an encrypted password which can be passed in other console commands (such as the authenticate command).

Syntax	`encrypt <string>`
Example	`> encrypt MyPa$$w0rd`
Result	Returns the encrypted equivalent for the specified string

HQL

The **hql** command executes a search based on a Hibernate Query Language statement. The command syntax matches the **sql** command's syntax, but this command can only select data, not update it.

Syntax	`hql <hql statement> | -f <inputFileName>`
Example	`> hql "select name, manager.name from Identity" > c:\data\Identities.dat` `> hql -f c:\hql\SelectIdentities.hql`
Result	The first example executes the specified HQL select statement and writes the results to the file c:\data\Identities.dat. The second example reads the HQL from the file c:\hql\SelectIdentities.hql, prints the HQL to the console (stdout), and displays the query results of the query to the console (stdout).

Date

The **date** command shows the current date and time for the application server or it can show the date and time value for a specified utime (universal time) value.

Syntax	`date [<utime>]`
Example	`> date` `> date 1338820492484`
Result	The first example displays the command syntax and the current date/time and current UTIME value. The second example returns the date/time value for the specified UTIME value.

Shell

The **shell** command escapes out to the command line and runs the command specified. (This command does not currently work properly in a Windows environment but does work in UNIX).

Syntax	`shell <command line>`
Example	`> shell ls`
Result	Lists the contents of the UNIX file system directory from which the console was run

Meter

The **meter** command toggles metering on or off. While metering is on, the console reports some timing statistics for each command executed. Meter information is displayed after the results of each command as it is executed

Syntax	`meter`
Example	`> meter`
Result	Turns metering on when it is currently off and turns it off when it is currently on. When turned on, all subsequently issued commands will report timing

	statistics until metering is turned off again. Meter information displayed includes: number of calls, total number of milliseconds, maximum time for one call, minimum time for one call, and average time per call.

Compress

The **compress** command is designed to compress the contents of a file to a string that could be included within an XML element. It compresses the file and then encodes it to Base64 and writes that text to the specified output file. This resultant file could then be used in an XML element stored in the database. This has limited usefulness within IdentityIQ since no part of the application is designed to read these compressed strings, but custom rules could be used to process them as needed or they could simply be stored in the database to be retrieved and uncompressed for use by an external application at a later time.

Syntax	`compress <input filename> <output filename>`
Example	`> compress file1.txt file2.txt`
Result	Compresses the contents of file1.txt, encodes that into Base64, and writes the resultant text string to file2.txt

Uncompress

The **uncompress** command functions in exactly the opposite way from the **compress** command, taking a compressed, Base64-encoded file and returning its uncompressed format.

Syntax	`uncompress <input filename> <output filename>`
Example	`> compress file2.txt file3.txt`
Result	Reverses the compressing process to return the original, uncompressed version of the text, writing that to the file file3.txt

ClearEmailQueue

The **clearEmailQueue** command deletes all queued but unsent email messages from the IdentityIQ email queue. This includes any new messages that have not yet been sent and messages that have encountered sending problems that have prevented successful delivery.

Syntax	`clearEmailQueue`
Example	`> clearEmailQueue`
Result	Deletes all unsent emails from the email queue

ClearCache

The **clearCache** command removes objects from the Hibernate object cache. This can be used when debugging Hibernate issues.

Syntax	`clearCache`
Example	`> clearCache`
Result	Clears the Hibernate object cache

Document Revision History

Revision Date	Written/Edited By	Comments
June 2012	Jennifer Mitchell	Initial Creation (IdentityIQ Version 5.5)
Sep 2013	Jennifer Mitchell	Correct errors in RunTaskWithArguments syntax (no commas between arguments)