Developer Guide

MOCA

Release 2017.1.0.0

Last updated: 28 August 2017

Legal notice

Provide feedback on this document

JDA values your opinion and strives to ensure that the documentation you receive is clear, concise, and provides the appropriate information required for you to use each JDA application efficiently.

If you would like to provide feedback on this document, you can submit your questions or suggestions to the JDA Documentation Management team and they will be forwarded to the appropriate development teams for review and consideration in a future release.

In addition to the provided documentation, many additional resources are available to help you understand and work with your JDA applications. For more information on these resources, see the JDA Services website.

Table of Contents

Chapter 1. About This Guide

Introduction

This guide provides MOCA developers with:

- A description of MOCA
- Advanced information regarding MOCA
- Procedures on how to work and develop in conjunction with MOCA
- Procedures on how to deploy applications in a MOCA environment

This guide should be used as a supplement to the *MOCA Quick Reference Guide*.

Purpose of Each Section

This table explains the intended purpose of each section in this guide.

Note: Previous versions of this guide included a section on the MOCA Console. Information on the Console is now published in the *MOCA Console Guide*.

Use the information in…	To learn…
"Architecture" (on page 5)	About the: • MOCA mission • MOCA architecture • MOCA overview
"Configuration" (on page 11)	About: • The MOCA registry file • Environment variables • Registry includes • Hooks • Jetty • Remote method invocation
"Starting MOCA" (on page 47)	• How to bootstrap an environment • How to start the MOCA server: ◦ From the command line ◦ As a UNIX service ◦ As a Windows service
"Tracing and Logging" (on page 53)	• About trace messages and levels • How to enable and disable trace messages • How to configure appenders for log files

Use the information in...	To learn...
"MSQL" (on page 68)	• How to start MSQL. • How to use MSQL commands • How to execute local syntax in MSQL
"Local Syntax" (on page 73)	• How to execute commands • About: ◦ Operators and functions ◦ Variable replacement ◦ More complex local syntax usage

"Command Repository" (on page 99)	• About:
	  ◦ Component level definitions
	  ◦ Command definitions
	  ◦ Trigger definitions
	• How to initialize and version component libraries
	• How to document commands and trigger

"Developing Components and Applications in Java" (on page 118)	• How to create a Java component
	• About:
	  ◦ MocaResults interface
	  ◦ MocaContext interface
	  ◦ MocaConnection interface

"Developing Components and Applications in C" (on page 133)	• How to create a C component • About: ◦ RETURN_STRUCT structure ◦ mocaDataRes structure ◦ Commonly used srvlib functions
"Jobs" (on page 138)	• What a job is • How to configure jobs • How to monitor jobs • How to trace jobs • How jobs work in a cluster

Use the information in...	To learn...
"Tasks" (on page 145)	• What a task is • How to configure tasks • How to monitor tasks • How to trace, test and troubleshoot tasks • How tasks work in a cluster • What is special about thread-based tasks
"Socket Server Tasks" (on page 157)	• How to configure socket server tasks • About socket server task processing commands
"Service Manager" (on page 160)	• How to configure services • How to start and stop services
"Security" (on page 168)	• About session authentication, keys and domains • About administration and database passwords • How to hide sensitive data from traces • How to limit client access to commands
"Transaction Management Support" (on page 179)	• Basics about transaction management support • How to have third-party tools join a transaction • About known third-party tools that can join a transaction
"Clustering" (on page 181)	• How to configure clusters by configuring: ◦ Registry keys ◦ JGroups ◦ Role managers • About cluster modes • About configuration scenarios • How to troubleshoot cluster-related problems
"Caching" (on page 197)	• About caching provided by MOCA and Infinispan (third-party caching provider) • How to enable Infinispan for MocaCache • How to enable and disable transactional caches • How Infinispan caching can be used with hibernate for clustered/transactional second level cache • About examples on how to configure various settings in Infinispan • About resources for additional Infinispan documentation

Use the information in...	To learn...
"Asynchronous Execution" (on page 209)	• How to obtain the executor • How to submit callable tasks for pull notification • How to submit callable tasks for push notification • How to perform asynchronous execution in a clustered environment
"Web Services" (on page 215)	• How to define and build Web services • How to deploy MOCA web services using Spring MVC • About web services security • How to call your MOCA Spring-based web service endpoints
"Configurable Web Services" (on page 229)	• About configurable web services and how they differ from SOAP and Spring MVC web services • About actions and resources • How to configure the XML files for actions and resources • How to handle primary keys and compound primary keys • About calling configurable web services and their responses • About web service utilities
"Command Profiling" (on page 247)	• How to enable and disable command profiling • How to analyze command profile information

"Pooling Framework" (on page 251)	• How to use these pooling framework entities:
	  ◦ Builders
	  ◦ Pools
	  ◦ Validators
	• How to use optional dynamic proxies

"Localization" (on page 257)	• How MOCA Console localization works • How to configure the default locale • How to add a new locale to an existing instance
"Monitoring and Diagnostics in MOCA" (on page 260)	• About Monitoring and Diagnostics in MOCA and probes • How to view probe data • About MOCA probes, the monitoring MBean, and notifications
"Extensibility" (on page 297)	• How to extend database tables by adding user-defined columns • About user-defined column behavior • How to identify tables available for extension

Chapter 2. Architecture

Overview

Introduction

This section provides a basic understanding of MOCA. Specifically, it describes the:

- Objectives of MOCA.
- Topography of the JDA supply chain execution thin client architecture.
- Services and libraries provided by MOCA.

MOCA Mission Statement

The mission of MOCA is to provide JDA development staff with a technical infrastructure that permits and promotes the interoperability of components in a simple, effective and non-implementation specific manner.

The JDA product development strategy promotes the creation of thin clients. A thin client application is responsible for user interaction and is not responsible for significant transactional or computational activity. In this type of architecture, it is critical to have proper delegation of responsibilities, and the phrase "client/services" is used to depict this approach. In this architecture, MOCA resides between the client and the services layer enabling both sides to interact.

Service Components

A service component is an isolatable and individually deployable implementation of functionality that has a known interface. Users of the services are not exposed to the implementation details of the components; they just know what functional problem the service component solves and understand how to activate it.

In terms of MOCA, all service components are known by a name. The name is defined by a verb/noun clause (such as "move inventory", "allocate shipment location" and "rate shipment").

Objective of MOCA

The objective of MOCA is to provide the framework enabling thin clients to activate service components. MOCA does not specify implementation; therefore, as long as services can communicate with MOCA, the implementation and specific implementation technology is left to the component developers.

What Is Provided With MOCA

MOCA provides:

- The framework for the development and deployment of service components. (While this is really more of a philosophy than specific software deliverable, it is an important part of MOCA.)
- Mechanisms enabling developers to create and deploy MOCA servers. For example, MOCA provides a mechanism for the definition of components and binaries to convert them into a form that can be used by MOCA.
- Libraries and development support in the form of C header files and libraries and Java classes, enabling and aiding application and server-side development.

Thin Client Architecture

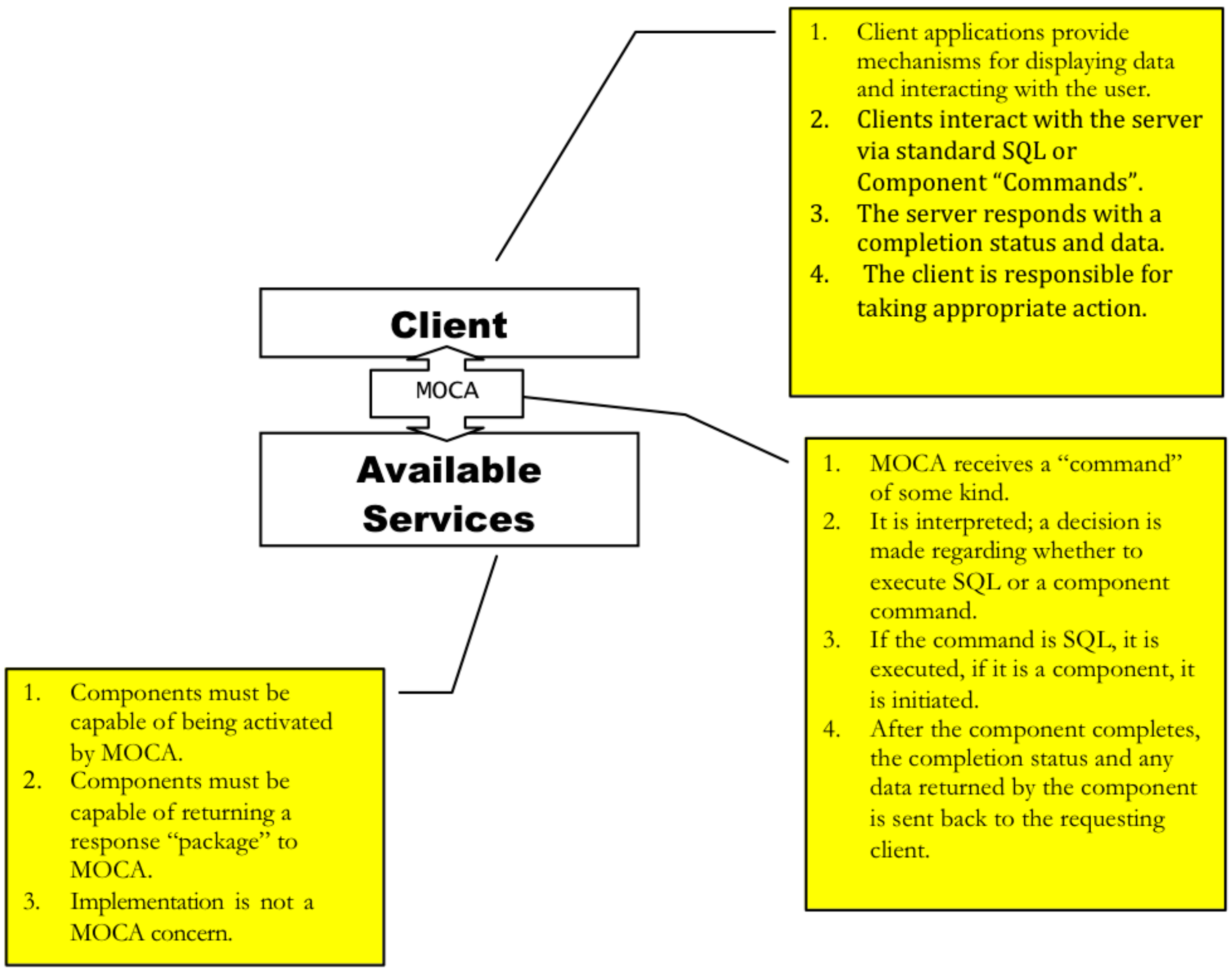
Description

The JDA approach to application development revolves around the idea of thin clients. Thin clients are applications whose responsibility is limited to user interaction (such as displaying information on a window and getting input from the user).

For example, internet/intranet applications are examples of thin clients; those applications run in the context of a browser that has virtually no ability to store data and very limited ability to store context information within the application. Applications running in the context of a browser must rely on a MOCA server to perform the bulk of the processing required by the application.

Implementation

This image is an example of how the MOCA framework enables thin clients to activate service components.



1. Client applications provide mechanisms for displaying data and interacting with the user.
2. Clients interact with the server via standard SQL or Component "Commands".
3. The server responds with a completion status and data.
4. The client is responsible for taking appropriate action.

1. MOCA receives a "command" of some kind.
2. It is interpreted; a decision is made regarding whether to execute SQL or a component command.
3. If the command is SQL, it is executed, if it is a component, it is initiated.
4. After the component completes, the completion status and any data returned by the component is sent back to the requesting client.

1. Components must be capable of being activated by MOCA.
2. Components must be capable of returning a response "package" to MOCA.
3. Implementation is not a MOCA concern.

Example: An Employee Database Using a MOCA-Based Application

This is an example of an employee database developed using a MOCA-based application. Users of the employee database should be able to view and maintain employee records. Simple maintenance programs such as the one described in this example are the standard program types of MOCA-based applications.

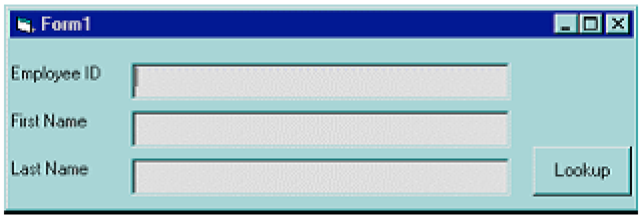
Note: MOCA provides some support for Visual Studio and other Windows development tools using the MMDA COM object. The example described would likely be built with Visual Basic or C# and the MMDA tool.

In this example, you need to develop an employee database that must store employee data (such as name, employee ID, hire date, salary, home address and phone number). A user of the employee database needs to be able to find and retrieve records meeting specific search criteria and then view the details of a record, modify the data in a record, and delete an existing record or create a new one.

In order to manage this type of database, you need to implement an "employee database management" application. In general, the application client would start up and initiate a connection to the MOCA server. If it could not start that connection, the application would not start. (In Visual Basic, this is done in the "form load" event or something similar.)

You use a typical data-driven application (DDA) since applications of this type generally have an initial lookup form. On the lookup form, the user can enter search criteria in one or more fields to perform a search. After the search is performed, a list of records that meet the specified search criteria appears in a grid. The user either works directly from that form, or accesses another form that displays more information for a selected record. From that second form, the user can modify the data in the record, delete the existing record or create a new one.

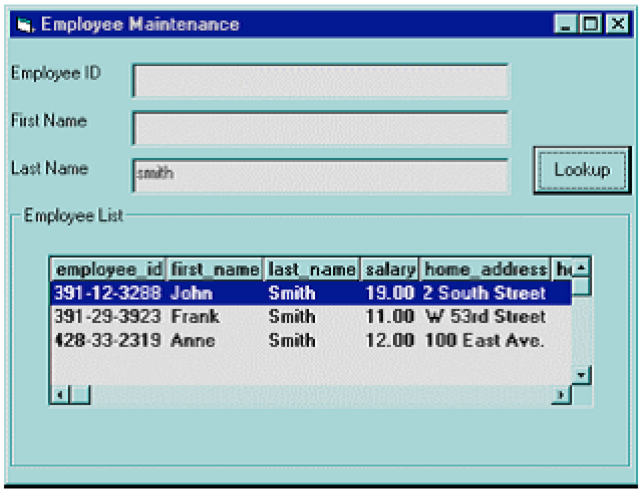
Query fields are provided for employee ID, first name and last name on the initial lookup form. The component that would do the query would implement the **list employees** command. It would take as arguments the fields from the employee database entity corresponding to the employee ID, first name and last name (for example, `employee_id`, `first_name`, and `last_name`).



After the user types **smith** in the `last_name` field on the lookup form and clicks the **Lookup** button, the client makes a call to the MOCA server sending it this command:

```
list employees where last_name = 'smith'
```

The MOCA server turns that command into a call to the appropriate component. The component responds with the requested records (MOCA components always return data to the caller in tabular form), the MOCA server sends the results back to the client, and the client displays the records that matched the search criteria in a grid.

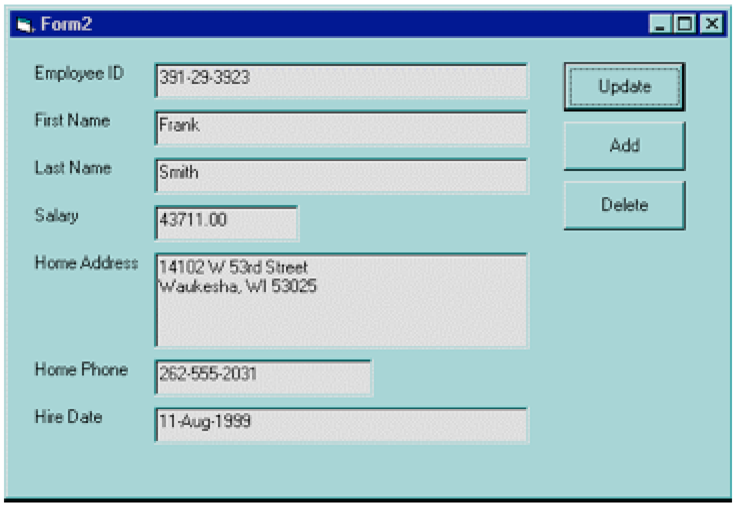


Employee ID	
First Name	
Last Name	smith

Employee List

employee_id	first_name	last_name	salary	home_address	h
391-12-3288	John	Smith	19.00	2 South Street	
391-29-3923	Frank	Smith	11.00	W 53rd Street	
428-33-2319	Anne	Smith	12.00	100 East Ave.	

In the grid the user selects the record for Frank Smith (employee ID 391-29-3923) and double-clicks the record to view the record details. The client application gathers all the information it can about that employee ID, and displays it in the main data entry form. Depending on the implementation, the client may choose to get more detailed information than the **list employees** command provided. Perhaps there is a **list employee details** command that takes only an `employee_id` argument, and it provides all the information the client needs to proceed (`list employee details where employee_id = '391-29-3923'`). Either way, the client application is now displaying the main data entry form.



Employee ID	391-29-3923
First Name	Frank
Last Name	Smith
Salary	43711.00
Home Address	14102 W 53rd Street Waukesha, WI 53025
Home Phone	262-555-2031
Hire Date	11-Aug-1999

After the user has modified the employee's record and clicks **Update**, some action must be taken to signal to the client application that it is time to submit the changes. The client application calls back to the MOCA server submitting this command:

```
change employee where employee_id= '391-29-3923' and first_
name='Frank' and last_name='Smith' and ...
```

If the user selects to delete the employee record, the client would have had to send a **remove employee** command instead. If the user had chosen to create a new employee record on the main data entry form, the client would have submitted this command:

```
add employee where employee_id ='...' and ...
```

Architectural Overview

Description

MOCA consists of several applications and processes. All of these applications and processes are used to serve MOCA requests that come from various client applications.

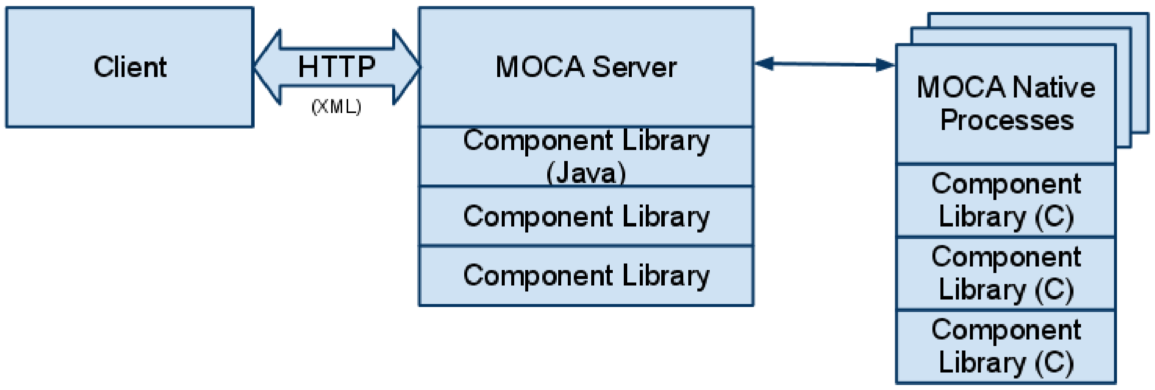
Definitions

This table provides definitions to common MOCA architectural terms that are used interchangeably but have distinct meanings.

Term	Definition
Service	A business capability exposed through a MOCA server.
Command	A client- or server-initiated request encoded as a text string that when executed in a MOCA server, causes components to be called.
Component	A unit of functionality implemented as a C function, Java method or local syntax command script. Each component is associated with a specific "command name" to allow it to be invoked using a command.

The MOCA Server

MOCA services are primarily served through the MOCA server. The MOCA server is a standalone Java application that listens for incoming HTTP requests, interprets a request as a MOCA command, executes the command and then returns the result to the caller. This image provides an example of this basic interaction.



A MOCA client (which can be a Web UI layer, a GUI library, an RF UI layer, a Web service layer or any other code that wants to initiate a service request) sends an HTTP request to the MOCA server. The MOCA server interprets the request, executes the requested components and returns the result to the caller.

MOCA Command Request

The MOCA command request consists of these three parts:

- A command that consists of a block of local syntax
- Arguments that are made available to the command processor
- Environment variables that are additional values that affect the way MOCA processes requests

Note: There are additional parts (such as authentication tokens and session keys) that are beyond the scope of this overview.

Command Syntax

The MOCA command language, known as local syntax, provides a mechanism by which service calls can be invoked using simple text-based commands similar to SQL. Every MOCA service request includes a command that can be any legal local syntax statement or sequence of statements.

Components can be written to take arguments. A client can include argument values in the syntax of a given request (such as in the where clause), or they can be passed by the client directly. Components can also be written to use environment variables. Environment variables are variables that are available to the MOCA server throughout the execution of a request, and effectively act like global variables. Examples of environment variables that may be passed to a MOCA server include locale and language, device ID, and user ID.

For more information about local syntax and how to effectively use it to execute commands, see "Local Syntax" (on page 73).

Chapter 3. Configuration

Introduction

This section describes the files (such as the RPTAB and registry file), environment variables and hooks used to configure the different parameters of an environment. For the registry file, each registry section and key/value pair are provided.

RPTAB File

The `rptab` file is a configuration file that is:

- Shared by all installed environments on a system.
- Used by the environment bootstrapping utilities to set the required environment variables. For more information on how to bootstrap an environment, see "Starting MOCA" (on page 47).

On Windows systems the `rptab` file is located in `%ALLUSERSPROFILE%\RedPrairie\Server\rptab` and contains a list of the installed environments as well as each environment's `LESDIR` path name, and optionally, the path name of the environment's registry file. This is an example of a Windows `rptab` file.

```
# ----------------------------------------------------------------------------
#
# SOURCE FILE: rptab
#
# DESCRIPTION: This file contains a list of RedPrairie environments installed
#              on the system. It is used by the RedPrairie environment
#              utilities.
#
# NOTE:     The registry file pathname field below is optional. If one is
#           not provided the rpset script will look for one using a
#           predefined set of rules looking in the LESDIR\data directory.
#
# FORMAT:   <EnvironmentName>;<Directory>[;<Registry File Pathname>]
#
# EXAMPLE: prod;c:\redprairie\prod
#
# ----------------------------------------------------------------------------

wm;c:\redprairie\wm\les;c:\redprairie\wm\les\data\registry
```

On Linux/UNIX systems the `rptab` file is located in `/etc/rptab`. The `rptab` file format for Linux/UNIX contains a list of the installed environments along with the log name of the owner of each environment, the environment's `LESDIR` path name, and optionally, a flag indicating if the environment should be automatically started when the system starts. This is an example of a Linux/UNIX `rptab` file.

```
# ----------------------------------------------------------------------------
#
# SOURCE FILE: /etc/rptab
#
# DESCRIPTION: This file contains a list of RedPrairie environments installed
#              on the system. It is used by the RedPrairie environment
#              utilities.
#
# FORMAT:  <EnvironmentName>:<Logname>:<Directory>:<Autostart?>
#
# EXAMPLE: prod:redprairie:/opt/redprairie/prod:Y
#
# ----------------------------------------------------------------------------

wm:wm:/opt/redprairie/wm/les:N
```

Registry File

Description

The registry file is an INI-formatted file with a number of sections, each consisting of one or more keys defined by a name/value pair. The registry file is used to configure different parameters of an environment (such as connecting to a database and security).

IMPORTANT: After changing the registry file, you must restart the server before the changes take effect.

Java System Properties

In addition to the proprietary MOCA registry, all MOCA registry keys are also implemented as Java system properties. This supports easier integration with third-party tools, such as Spring, that use Java system properties as a standard method for maintaining the tools' XML-based configurations. To reference a MOCA registry key using a Java system property, use the following format:

`moca.registry.<section_name>.<registry_key>`

Example: The following table illustrates a MOCA registry key as defined in the MOCA registry and its corresponding Java system property. Reading the `moca.registry.java.vmargs` Java system property in this example returns `-Xmx1G`.

MOCA registry format	Corresponding Java system property format
`[JAVA]vmargs=-Xmx1G`	`moca.registry.java.vmargs`

Java system property names are formatted as all lowercase letters and can be accessed when the MOCA server is running, or by a process or utility that calls `ServerUtils.setupDaemonContext`. Values that are encrypted are not decrypted; for example, a registry entry that is an encrypted password remains encrypted when read by the corresponding Java system property.

Example: Registry File Section Format

This is an example of the format of a section named "SERVER" with these three keys: "port", "rmi-port" and "memory-file". These keys could also be referred to as `server.port`, `server.rmi-port` and `server.memory-file`. This is the format to which registry keys are referred in this guide.

```
[SERVER]
port=4500
rmi-port=4501
memory-file=%LESDIR%\data\commands.mem
```

Service Section

The service section is used to configure how the server is started when it is started using `rp start` on Linux/UNIX platforms. This does not affect the Windows service. This is controlled through the new Service Manager for Windows see "Service Manager" (on page 160)

This table lists the valid keys for the service section.

Name	Type	Default Value	Description
`command`	command	`mocaserver`	Command that runs when the service starts. To enable full tracing, set this to `mocaserver -t* -o`.
`output`	pathname	`%LESDIR%`\log\mocaserver.log	File to which the standard output is written that comes from the server.

Cluster Section

The cluster section is used to configure the server when deployed in a cluster. For more information on how to define the JGroups registry keys, see "Clustering" (on page 181).

This table lists the valid keys for the cluster section.

Name	Type	Default Value	Description
`name`	string		Name of the cluster. Every node in a cluster must use the same name in order to communicate with each other.
`node-name`	string		Name of the node. A readable node name more clearly identifies nodes in the cluster, for example "MAIN_NODE".
`jgroups-xml`	pathname		Name of the JGroups XML-based configuration file, such as `jgroups-udp.xml` or `jgroups-tcp.xml`, that specifies how JGroups is configured for communication with the rest of the nodes on the cluster. This file must be located in the `%LESDIR%\data` directory.
`cookie-domain`	string		Domain name used when setting the domain for the console session cookie. **IMPORTANT**: This value must be set to a portion of the domain name that is consistent across all nodes in the cluster and must always start with a "."; otherwise, users are required to re-authenticate when switching between nodes when using the MOCA Console.

Name	Type	Default Value	Description
`role-manager`	string	`PREFERRED`	Type of role manager to control the ownership of jobs and tasks on the node. Three different types of role managers can be defined on each node in the cluster. These are the valid values: `FIXED`, `PREFERRED` (default) and `DYNAMIC`. For more information on the different role manager implementations, see "Clustering" (on page 181).
`roles`	list of strings		Comma-separated list of one or more roles that the role manager uses to determine the roles that a node owns when it starts.
`exclude-roles`	list of strings		Comma-separated list of one or more roles that the role manager excludes when determining the roles that a node owns when it starts.
`role-check-rate`	integer	`10`	Number of seconds that each node waits between each acquisition of a role. This wait time staggers the assignment of roles, so a single node does not acquire all roles and then lose them as other nodes start.
`async-runners`	integer	`10`	Number of thread runners that the server starts to handle the incoming asynchronous execution requests.
`async-submit-cap`	integer	`10000`	Maximum number of asynchronous execution requests that a node can handle at one time. Any callable task submitted beyond this limit blocks until a node can handle another callable task.
`jgroups-protocol`	string	`udp`	The communication standard to use for clustering. These are the valid values: • **udp** – User Datagram Protocol (UDP) communication protocol. Cluster membership is dynamically discovered using UDP multicast. • **tcp-mcast** – Transmission Control Protocol (TCP) communication protocol. Cluster membership is dynamically discovered using UDP multicast. • **tcp-hosts** – TCP communication protocol. Cluster membership is limited to a static pre-defined list of nodes. • **tcp-db** – TCP communication protocol. Cluster membership is dynamically discovered using a shared database. This option requires that the MOCA instance is configured to use a database.
`jgroups-bind-addr`	string		IP address to which JGroups can bind. This can be used when a static IP address is configured for the node's adapter. Starting with version 8.2.0.0, this key is mandatory for clusters that are set up using the MOCA registry settings.

Name	Type	Default Value	Description
jgroups-bind-interface	string		Interface to which JGroups can bind. It is recommended that you use this key when multiple network adapters are attached to the node (for example, eth0).
jgroups-bind-port	integer		Port that JGroups uses for communications. When jgroups-protocol is tcp-hosts, this value defaults to 7800. Otherwise, this value defaults to a dynamically assigned port.
jgroups-tcp-hosts [1]	string		Comma-separated list of all the names and JGroups bind port number of the nodes in the cluster, except the node currently being configured. For example, if you have nodes host1, host2 and host3 in your cluster, communicating on port 7800 and are currently configuring host3, then jgroups-tcp-hosts would be host1[7800],host2[7800].
jgroups-compress	string	false	Indicates whether JGroups is to compress messages between nodes. To enable compression, set the value of this key to true.
jgroups-mcast-addr [2]	string	228.6.7.8	IP address to use for multicast.
jgroups-mcast-port [2]	integer	46655	Port number to use for multicast.
jgroups-tcp-ping-timeout [1]	integer	5000	Timeout in milliseconds to wait for initial members.
jgroups-merge-timeout	integer	15000	Timeout in milliseconds to wait to complete a merge.
remote-retry-limit	integer	5	Number of retries for an Infinispan cache operation in response to a failure.

Name	Type	Default Value	Description
`exclude-process-tasks`	boolean	`false`	Removes all process based tasks from the cluster. Disabling process based tasks in the cluster reduces network traffic and improves cluster stability. If sharing caches between processes is required, then individual process based tasks can be added to the cluster by specifying the JVM argument – `Dmoca.cluster=true` or reconfiguring the task to thread may work as well. The use cases for cache sharing needs to be determined on a case by case basis, there's is no easy way to determine what process tasks should or should not be clustered.
`mode`	string	`moca`	The clustering architecture mode for the server. The "moca" clustering mode is enabled by default because it provides a much more stable cluster. It is possible to switch back to the old "infinispan" mode if any issues are found. Switching back to the old mode is highly discouraged.

1 Required if jgroups-protocol is set to tcp-hosts.
2 Only applies if jgroups-protocol is set to udp or tcp-mcast.

Server Section

The server section is used to configure server-based applications that consist of the MOCA server as well as other applications running in server mode.

This table lists the valid keys for the server section.

Name	Type	Default Value	Description
`url`	url		URL that clients use to connect to the environment.
`port`	integer	`4500`	Port number on which to listen if a `jetty.xml` file does not exist.
`rmi-port`	integer	`4501`	Port number for the RMI registry to use.
`classic-port`	integer		Port number used for the classic MOCA server. If not defined, then the classic MOCA listener is not started.
`mad-port`	integer		Port number used by the Monitoring and Diagnostics application. If not defined, then the port number is dynamically determined.
`native-process-port`	integer		Port used for communication with native processes. If not defined, then port number is dynamically determined.

Name	Type	Default Value	Description
`native-crash-port`	integer		Port used for communication with native processes. If not defined, then port number is dynamically determined.
`rmi-object-port`	integer		Port used by RMI. If not defined, then port number is dynamically determined.
`trace-file`	pathname		Path name of the trace file.
`trace-level`	string		Enabled trace level: • ***** – All trace levels are logged, with the exception of command profiling. • **Not defined** – Only levels equal to or higher (more specific) than `INFO` are logged.

Name	Type	Default Value	Description
`command-profile`	pathname		Indicates how you want to use command profiling to track usage and performance: • **No registry entry or a registry entry with no value** – Saves the command profile information and statistics in memory. If the `command-profile` registry key is not specified (no `command-profile=<Value>`) or it is specified, but left blank (`command-profile=`), command profiling is done in memory. Therefore, you can only analyze usage and performance between server restarts. • **`command-profile=<Path Name><File Name>`** – Saves the command profile information and statistics in a file, where the file is named <*File Name*>**--server.csv** and is stored in the directory identified by <*Path Name*> (for example, **%LESDIR%\log\mocaprofile**). This option lets you save and analyze usage and performance across server restarts. Profile data is only written to the file during server shutdown so the performance overhead is very low. • **`command-profile=off`** – Disables all command profiling, including in-memory command profiling. This option lets you eliminate the memory usage sustained in command profiling. For more information, see "Command Profiling" (on page 247).
`memory-file`	pathname	`%LESDIR%\data\commands.mem`	Path name of the command file.

Name	Type	Default Value	Description
`mailbox-file`	pathname		Path name of the mailbox file.
`prod-dirs`	list of pathnames		List of installed product path names, separated by the operating system path separator (for example, semicolon for Windows or colon for UNIX) and supplied in order of highest to lowest layered product. Sub-directories of products can also be added, such as `%DCSDIR%\webclient`.
`arg-blacklist`	list of strings		Comma-separated list of argument names, such as usr_pswd, whose value must not appear as clear text in trace files. For more information on traces and sensitive data, see "Security" (on page 168).
`min-idle-pool-size`	integer	`1`	The minimum number of idle native processes that are maintained in the pool. If the total number of idle and busy processes equals the `max-pool-size` value, then another process is not be spawned. Setting the value of `min-idle-pool-size` to 0 causes processes to only be created as requested. Setting this value equal to the `max-pool-size` value ensures that the pool always has that many processes in it.
`max-pool-size`	integer	`20`	Maximum size of the native process pool. If this maximum size is reached, then each request waits until a busy process becomes idle.
`classic-pool-size`	integer	`20`	Number of threads used for executing incoming socket requests on the MOCA classic protocol listener.
`classic-encoding`	string	`UTF-8`	Encoding used to communicate with classic clients. This must be valid Java encoding and must be the same as US-ASCII in the lower 7 bits. UTF-8 and the typical 8-bit ISO-8859-x encodings/charsets all qualify.

Name	Type	Default Value	Description
`classic-idle-timeout`	integer	`0`	Number of seconds a classic MOCA (socket) connection can be idle and still be connected to the server. If not defined, or defined as `0` (zero), connected sockets are kept indefinitely.
`process-timeout`	integer	`20`	Number of seconds the server waits for a process to start before throwing an error.
`max-commands`	integer	`10000`	Maximum number of requests that this native process can use before it is shut down.
`session-idle-timeout`	integer	`3600`	Number of seconds a MOCA session can be idle (no recent request) before it is automatically removed if it was holding on to some kind of server resource.
`session-max`	integer	`10000`	Maximum number of MOCA sessions that can be open at the same time for a server. If the maximum number is reached, then a new request tries to terminate the next idle session to be removed. If no idle sessions are available, then the request waits until one becomes available.
`query-limit`	integer	`100000`	Maximum number of rows that a query can return. If an SQL query returns more rows than the number specified by `query-limit`, an exception is thrown and the transaction rolls back. The value of `query-limit` can be set to a number between 1 and 2,000,000 inclusive. If you set `query-limit` to a number outside of the valid range of values (for example, 0), the MOCA server does not start and a message is added to the log stating that the query-limit value is not in the valid range. The maximum valid value of 2,000,000 is set to prevent out of memory errors that might occur on the server if all rows are read into memory.

Name	Type	Default Value	Description
`compression`	boolean	`false`	Indicates whether MOCA is to compress the server response. Set this key to `true` to enable compression of the server response.
`max-async-thread`	integer	`20`	Maximum number of asynchronous runner threads that are available to execute asynchronous callable tasks.
`console-default-locale`	string	`en-us`	Default locale for the MOCA Console (including the login page). This key must be a locale name as defined in the **locale-mapping.xml** file. For more information, see "Localization" (on page 257).
`row-accumulation-limit`	integer	`1000000`	Maximum number of rows that can be accumulated when piping together commands in local syntax. This value is used to prevent an excessively large number of rows from being accumulated which could cause performance or memory issues on the server. If the accumulation limit is reached, an exception is thrown. For more information (including a description of how to handle cases when you want to temporarily exceed the row accumulation limit), see "Row Accumulation Limit" (on page 96).
`mad-probing-enabled`	boolean	`true`	Indicates whether Monitoring and Diagnostics probes are enabled.
`inhibit-tasks`	boolean	`false`	Indicates whether to inhibit MOCA tasks. Turning this option on runs MOCA without tasks.
`inhibit-jobs`	boolean	`false`	Indicates whether to inhibit MOCA jobs. Turning this option on runs MOCA without jobs.
`support-zip-timeout`	integer	`2000`	Milliseconds to wait for custom support ZIP hooks to finish running. This timeout may be increased if there are custom support ZIP hooks that rely on expensive operations to complete.

Name	Type	Default Value	Description
`clean-war-deploy`	boolean	`true`	Indicates that MOCA is to delete the expanded web archive files (WARs) that are deployed by Jetty every time the MOCA server is shut down. This action eliminates potential disk space issues, but at the cost of a longer shut down time. The daemon thread that is spawned to delete the expanded WARs takes time and uses resources that cannot be used by other processes while the daemon thread is running. If a shorter shut down time is more important than a potential disk space issue or you have processes in place to delete the WARs, set this key to false.
`rpweb-url`	url		Required. Specifies the REFS (portal server) service URL to facilitate communication between MOCA and REFS systems.
`mass-index-interval`	integer		Milliseconds to wait between indexing executions before MOCA reindexes all hibernate tables that have an entity that is annotated with `@Index`. To disable indexing, do not include this MOCA registry key. If you include this MOCA registry key, but do not specify its value (`massindexinterval=`), indexing starts and stops without performing any indexing, which produces a line in trace files that states that the index is disabled.

Name	Type	Default Value	Description
`async-trace-close-wait-ms`	integer	`15000`	Max time in milliseconds that a call to disable a trace may block while waiting to ensure that all asynchronous log messages have been delivered. This setting should only be increased if the system is very busy, clients are losing messages at the end of a trace, and the log messages are critical to the operation. Increasing the value on a very busy system may introduce blocking for other areas that close traces such as tasks with a log configured. If MOCA detects that trace messages have been lost, it logs an error in the server trace.
`ws-cws-context`	string	`cws`	The default context path for Configurable Web Services. This determines the prefix for all CWS action paths.
`ws-max-upload-size-kb`	integer	`1000`	The maximum allowed request size (in kilobytes) before file uploads are refused by the RESTful web service framework.
`ws-base-url`	string		Optional. Defines the base path the system uses for all REST web resource links included in web service responses. By default, the system creates links using the scheme, host and port of the current request. This approach may not be adequate for customers with sophisticated deployments. This setting provides customers the option to deterministically define the base path to be use instead of mirroring the current request. The base path should contain the scheme, host and port (optional). For example: `https://prod.server.corp:8080`

Name	Type	Default Value	Description
`ws-expansion-url`	string		Optional. Defines the base path the system uses to expand REST web resource links. This configuration option is only available when ws-base-url is defined. By default, during web service link expansion the system performs a web service call using the base path defined by ws-base-url. If ws-base-url points to a load balancer or other network appliance it may be preferable for the system to expand links using a different base path (such as localhost) for improved performance. For example: `http://localhost:4500`
`console-base-url`	string		Optional. Defines the base path the system uses for the MOCA Console. The MOCA Console contains several flows that use HTTP redirects (302), which by default are executed using the current request’s scheme, host, and port. This setting provides customers the option to define the base path to be used instead of mirroring the current request. The base pat should contain the scheme, host, and port (optional). For example: `https://prod.server.corp:8080`
`process-watcher-threads`	integer	1	Optional. Defines the number of threads for accepting native process connections. Higher value improves resilience against non-native process connections, but also uses more resources. Value of 0 forces legacy non-threading behavior. Intended for situations when the box is experiencing a lot of nuisance socket connections, such as caused by running port security scans.
`process-id-timeout-millis`	integer	500	Optional. Timeout when reading a process id from new native process connection.

Name	Type	Default Value	Description
`process-accept-timeout-millis`	integer	100	Optional. Timeout when waiting to accept a new native process connection.
`user-auth-persistence-provider`	string	infinispan	Optional. Specifies the mechanism MOCA uses for persisting user authentication sessions. MOCA relies on session persistence to support shared logins across a MOCA cluster. These are the valid values: • **database** – Offers good performance and reliability. Requires MOCA to be configured with a database. • **infinispan** – MOCA uses an Infinispan replicated cache to maintain sessions across a cluster. This option offers the best performance and does not require a database, but it's less reliable (existing user logins not visible to all nodes of the cluster). Customers experiencing difficulties with their cluster should consider switching to the database option.
`oracle-limit-syntax`	string	cte	Optional. Specifies which algorithm MOCA uses by default to page SQL queries. Queries that explicitly specify an algorithm will use that algorithm over the one set by this registry setting. These are the valid values: • **classic** – For Oracle database. Uses a sub-query select for the page data. • **cte** – For Oracle database. Uses a common table expression for the main query (default). • **over** – For Oracle database. Uses analytical count over the whole data set. Note that this algorithm may not be compatible with some types of queries, so great care should be taken before selecting it for a production environment. See Local Syntax in the *MOCA Developer Guide* for more information about SQL query paging.

Database Section

The database section is used to configure connection information for the database.

This table lists the valid keys for the database section.

Name	Type	Default Value	Description
`username`	string		Username used to connect to the database.
`password`	string		Password used to connect to the database.
`dba-username`	string		Username used by the `installsql` and `dbupgrade` utilities to connect to the database. If not defined, then the `username` parameter is used.
`dba-password`	string		Password used by the `installsql` and `dbupgrade` utilities to connect to the database. If not defined, then the `password` parameter is used.
`url`	url		URL used to connect to the database. URLs are specific to a database driver.
`driver`	class name		Class name of the JDBC driver used to connect to the database. If not defined, then database support is disabled.
`min-idle-conn`	integer	`1`	The minimum number of idle database connections that are maintained in the pool. If the total number of idle and busy connections equals the `max-conn` value, then another connection is not created. Setting the value of `min-idle-conn` to 0 causes connections to only be created as requested. Setting this value equal to the `max-conn` value ensures that the pool always has that many connections in it.
`max-conn`	integer	`100`	Maximum number of database connections that can exist in the database connection pool.
`pool-validate-on-checkout`	boolean	`true`	Indicates that MOCA is to verify the validity of a connection from the database connection pool before providing it to a request. Connections can become stale (invalid) if issues occur, such as network outages or database server restarts. This key provides the following behavior: • **Key is true**: Each transaction that requires a database connection submits a simple query to check if the connection is valid. If the connection is not valid, then it is removed from the pool and the process is repeated until a valid connection is found. Setting this key to true improves the stability of the server at a small performance cost (one additional, simple query per transaction). • **Key is false**: Since no validation is performed, a request can potentially receive a stale connection that results in the request failing. However, the stale connection is then removed from the pool.
`conn-timeout`	integer	`30`	Number of seconds before attempts are made to get a connection from the database connection pool.

Name	Type	Default Value	Description
login-timeout	integer	30	Number of seconds before attempts are made to establish a new database connection.
isolation	string		Isolation level to use. This is only valid for SQL Server databases. These are valid trace levels: read_committed, read_uncommitted, repeatable_read, serializable, snapshot and default.
dialect	class name		Name of the class used for translation of SQL. If a dialect is not specified, then it is derived from the driver parameter. These are valid class names: com.redprairie.moca.server.db.translate.OracleDialect and com.redprairie.moca.server.db.translate.SQLServerDialect.
web-statement-timeout	integer	0	Timeout for SQL statements that are called only though a web service. Timeout of 0 indicates no timeout.

Java Section

The Java section is used to define the configuration of Java VMs.

This table lists the valid keys for the Java section.

Name	Type	Default Value	Description
vm	pathname	java	Name of the Java executable used for the main server and all 64-bit applications.
vmargs	string		Arguments passed to the Java VM.
native-vmargs	string	"-Xmx128m"	Arguments passed to the Java VM.
vmargs.*appname*	string		Arguments passed to the Java VM when launching the application *appname*.

Security Section

The security section is used to configure the security attributes of the environment, including which environments can interact with each other and how users are authenticated externally through LDAP.

This table lists the valid keys for the security section.

Name	Type	Default Value	Description
domain	string		Domain to which the environment belongs. This should be a single word that uniquely identifies the system.
trusted-domains	list of		Space-separated list of domains that are

Name	Type	Default Value	Description
	domains		allowed to access the environment. This is useful for allowing a login on one environment to execute remote commands on another environment.
`ldap-url`	url		URL used to connect to the LDAP server. Multiple semi-colon separated URLs can be provided.
`ldap-bind-dn`	string		Distinguished name of the user to bind to the LDAP server.
`ldap-bind-password`	string		Password of the user defined by the `ldap-bind-dn` key. The LDAP bind password can be hashed in the registry. Generate the hash using `mpasswd -l`.
`ldap-auth-type`	string		Authentication type used when authenticating users. These are the valid values: "SIMPLE" or "DIGEST-MD5".
`ldap-referrals`	string		Behavior used when the LDAP server issues a referral to another LDAP server. These are the valid values: "IGNORE" or "FOLLOW".
`ldap-uid-attr`	string		Name of the LDAP attribute that contains the user's login name. Typically, this is "sAMAccountName" for ActiveDirectory servers and "uid" for other LDAP servers.
`ldap-role-attr`	string		Name of the LDAP attribute that contains the list of roles to which the user has been assigned.
`allow-legacy-sessions`	boolean	`true`	Indicates whether legacy (socket) connections are allowed in the server.
`session-key-idle-timeout`	integer	`1800`	Number of seconds a client connection can be idle before automatically forcing a re-authentication request.
`remote-timeout`	integer	`86400`	Number of seconds a remote connection can last before automatically being cancelled.
`admin-user`	string		Administrative user that can be used to log in to the MOCA Console or to support remote access by Java Management Extensions (JMX). Typically, you should only use the administrative user for a MOCA server instance that is not associated with a database or database access is not possible. When using a MOCA server instance that is associated with a database, you should instead use standard

Name	Type	Default Value	Description
			authentication to authenticate the user against the database. For more information, see the "Administrative password" section in "Security" (on page 168).
`admin-password`	string		Hashed password associated with `admin-user`. A new hashed password can be generated using the mpasswd utility.
`ws-xss-protection-enabled`	boolean	false	When enabled, indicates that the MOCA web service framework interrogates inbound and outbound web services for cross-site scripting (XSS) content. If XSS content is detected in a request, the system errors and returns a 400 Bad Response HTTP status. If XSS content is detected when writing the response, the system stops writing and terminates the request.
`size-limit-http-request`	integer	0	(0=unlimited) Maximum size limit of an http request that MOCA can handle.
`size-limit-read-file`	integer	0	(0=unlimited) Maximum size limit of a regular file read by MOCA code into memory.
`size-limit-log-file`	integer	0	(0=unlimited) Maximum size of a log file read by Moca code into memory.
`size-limit-alert-file`	integer	16777216	(16777216 =16M) Maximum size of EMS alert file read by MOCA into memory.
`size-limit-code-file`	integer	16777216	(16777216=16M) Maximum size of source code file read by MOCA into memory.
`size-limit-email-attachment`	integer	268435456	(268435456=256M) Maximum size of email attachement that MOCA can handle.
`size-limit-string_identifier-bridge`	integer	0	(0=Unlimited) Maximum size of string identifier bridge.
`line-length-limit-process-output`	integer	65536	(65536=64K) Maximum line size when reading output from a process. If line length exceeds this value, it is split into smaller chunks.

Server Mapping Section

The server mapping section is used to provide a mechanism for mapping aliases (usually host/port pairs) used for remote execution to a connection URL.

This table lists the valid key for the server mapping section.

Name	Type	Default Value	Description
alias	url		Mapping from a server alias to a URL. This is usually used to map classic *host:port* pairs to a URL.

Environment Section

The environment section is used to configure environment variables through the registry. The environment variables defined in this section are used when processes that are looking for environment variables cannot find them in the actual running environment. Environment variables defined in the registry file can be thought of as default environment variables.

Note: Environment variables in the registry file do not take effect until a process starts; consequently, setting an environment variable such as `PATH` is not effective.

This table lists the valid key for the environment section.

Name	Type	Default Value	Description
name	string		Environment variable name/value pair.

Event Management Section

The Event Management section is used to configure the Event Management spooler.

This table lists the valid keys for the Event Management section.

Name	Type	Default Value	Description
`url`	url		URL used to connect to the Event Management server.
`spool-dir`	pathname		Directory from which the files are read.
`processed-dir`	pathname		Directory to which the files are moved after successful processing.
`bad-dir`	pathname		Directory to which the files are moved that cannot be successfully processed.

MOCARPT Section

The MOCARPT section is used to configure the Reporting server.

This table lists the valid keys for the MOCARPT section.

Name	Type	Default Value	Description
`app-server`	url	`server.url`	URL used to connect to the MOCA server. This is typically used in a clustered environment in order to load balance report requests.
`report-server`	url		URL used to connect to the Reporting server.

Name	Type	Default Value	Description
			Note: If Reporting is installed in the same environment, this value should not be set.
`archive-folder`	pathname	`%LESDIR%\report-archive`	Path name of the directory for archived reports.
`cache-folder`	pathname	`%LESDIR%\report-cache`	Path name of the directory for cache files.
`db-date-time-format`	string	`#yyyy-mm-dd Hh:Nn:Ss#`	Date and time format used for the archive database. These are the typical values for each database engine: • **Access** – #yyyy-mm-dd Hh:Nn:Ss# • **Oracle** – 'yyyymmddHhNnSs' • **SQL Server** – 'mm/dd/yyy Hh:Nn:Ss'
`db-init-command`	string		Command to execute when a connection to the archive database is established. This is primarily used to set up date and time formats. These are the typical values for each database engine: • **Access** – *none* • **Oracle** – `alter session set NLS_DATE_FORMAT='YYYYMMDDHH24MISS'` • **SQL Server** – *none*
`default-locale-id`	string	`US_ENGLISH`	Locale ID used if it is not passed to the Reporting server components. The locale ID is used when translating report text through the multi-language system.
`ems-attachment-format`	string	PDF	Format to which to export reports for attaching to Event Management events.

Name	Type	Default Value	Description
ems-folder	pathname	%LESDIR%\report-ems	Path name for Event Management attachments.
lib-folder	pathname	%LESDIR%\report-libs	Path name of the directory containing DLLs or shared libraries to load.
failure-log-path	pathname	%LESDIR%\log\report-failure-logs	Path name of the directory for report failure log files.
failure-log-keep-days	integer	1	Number of days of report failure logs to keep. The Reporting server automatically purges report failure logs when writing new ones.
local-db-connection	string	Provider=Microsoft.Jet.OLEDB.4.0;Data Source=$LESDIR\db\mocarpt.mdb	Database connection string for the archive database. This should be an OLEDB or Windows DSN connection name.

Monitoring Section

The Monitoring section is used to configure settings related to Monitoring and Diagnostics functionality.

This table lists the valid keys for the Monitoring section.

Name	Type	Default Value	Description
csv-reporter-enabled	boolean	true	Indicates whether the Monitoring and Diagnostics CSV Reporter is enabled. The CSV Reporter periodically saves Monitoring and Diagnostics probe data to the local file system in CSV format for historical monitoring purposes.
csv-reporter-directory	pathname	%LESDIR%/data/csv_probe_data	Directory to which the CSV Reporter writes probe data CSV files. This directory should only be used by the CSV reporter. Additionally, the MOCA server process must have write access to this directory.

Name	Type	Default Value	Description
`csv-reporter-keep-hours`	integer	`72`	Number of hours that the CSV Reporter should retain probe data CSV files. Any probe data CSV files with a time stamp that has aged past the keep period defined by `csv-reporter-keep-hours`, are automatically deleted from the `csv-reporter-directory`. **Note**: To archive a probe data CSV file before it is deleted, specify a directory for the `csv-reporter-archive-directory` key.
`csv-reporter-archive-directory`	pathname		Directory to which the CSV Reporter can optionally archive probe data CSV files: • If `csv-reporter-archive-directory` is not defined, archiving is not enabled (default). • If `csv-reporter-archive-directory` is defined, archiving is enabled. If enabled, archiving occurs before a probe data CSV file is deleted from the `csv-reporter-directory` directory based on the `csv-reporter-keep-hours` time period. Archiving consists of compressing the probe data CSV files into ZIP files and storing them in the `csv-reporter-archive-directory` directory. This directory should only be used by the CSV Reporter. Additionally, the MOCA server process must have write access to this directory.
`csv-reporter-archive-keep-hours`	integer	`168`	Number of hours to retain archived probe data ZIP files on the file system. Archive probe data ZIP files with a time stamp that has aged past the keep period defined by `csv-reporter-archive-keep-hours`, are automatically deleted from the `csv-reporter-archive-directory` directory.

Name	Type	Default Value	Description
csv-reporter-support-default-hours	integer	24	Number of hours of probe data CSV files that should be included in the support ZIP file. This data is useful for debugging purposes to gain a historical perspective of the system. For more information, see the "Support file" section in the *MOCA Console User Guide*.
jolokia-auth-model	string	simple	MOCA authentication model for Jolokia connections. These are the valid values: • **simple**: a fast, stateless protocol that is limited to the MOCA Console admin user. Simple is the recommended value. • **persisted**: the normal MOCA authentication protocol that allows advanced logins like LDAP and setting up cookies. **Note**: The persisted auth scheme may create a lot of user sessions in the MOCA system. This may have an effect on the performance of the system, especially if clustering is configured.

Registry File Template

This is a registry file template that contains every registry section and key/value pair.

```
#------------------------------------------------------------------------
#
# SOURCE FILE: registry
#
# DESCRIPTION: RedPrairie environment registry file.
#
#------------------------------------------------------------------------

[SERVICE]
command=<command line>
output=<file name>

[CLUSTER]
name=<string>
node-name=<string>
cookie-domain=<string>
jgroups-xml=<pathname>
role-manager=<string>
roles=<list of strings>
exclude-roles=<list of strings>
role-check-rate=<integer>
async-runngers=<integer>
async-submit-cap=<integer>
```

```
jgroups-protocol=<string>
jgroups-bind-addr=<string>
jgroups-bind-interface=<string>
jgroups-bind-port=<integer>
jgroups-compress=<string>
jgroups-tcp-hosts=<list of strings>
jgroups-mcast-addr=<string>
jgroups-mcast-port=<integer>
jgroups-tcp-ping-timeout=<integer>
jgroupd-merge-timeout=<integer>
remote-retry-limit=<integer>
exclude-process-tasks=<boolean>
mode=<string>

[SERVER]

url=<url>
port=<integer>
rmi-port=<integer>
classic-port=<integer>
trace-file=<pathname>
trace-level=<string>
command-profile=<pathname>
memory-file=<pathname>
mailbox-file=<pathname>
prod-dirs=<list of pathnames>
arg-blacklist=<list of strings>
min-idle-pool-size=<integer>
max-pool-size=<integer>
classic-pool-size=<integer>
classic-encoding=<string>
classic-idle-timeout=<integer>
process-timeout=<integer>
max-commands=<integer>
session-idle-timeout=<integer>
session-max=<integer>
query-limit=<integer>
compression=<boolean>
max-async-thread=<integer>
console-default-local=<string>
row-accumulation-limit=<string>
mad-probing-enabled=<boolean>
inhibit-tasks=<boolean>
inhibit-jobs=<boolean>
support-zip-timeout=<integer>
clean-war-deploy=<boolean>
rpweb-url=<url>
```

```
mass-index-interval=<integer>
async-trace-close-wait-ms=<integer>
ws-cws-context=<string>
ws-max-upload-size-kb=<integer>
classic-oracle-limit-syntax=<boolean>
ws-base-url=<string>
ws-expansion-url=<string>
console-base-url=<string>

[DATABASE]
username=<string>
password=<string>
dba-username=<string>
dba-password=<string>
url=<url>
driver=<string>
min-idle-conn=<integer>
max-conn=<integer>
pool-validate-on-checkout=<boolean>
conn-timeout=<integer>
login-timeout=<integer>
isolation=<string>
dialect=<string>
web-statement-timeout=<integer>

[EMS]
url=<url>
spool-dir=<pathname>
processed-dir=<pathname>
bad-dir=<pathname>

[ENVIRONMENT]
<name>=<string>

[SERVER MAPPING]
<alias>=<url>

[SECURITY]
domain=<string>
trusted-domains=<domain list>
ldap-url=<url>
ldap-bind-dn=<string>
ldap-bind-password=<string>
ldap-auth-type=<string>
ldap-referrals=<string>
ldap-uid-attr=<string>
ldap-role-attr=<string>
```

```
allow-legacy-sessions=<boolean>
session-key-idle-timeout=<integer>
remote-timeout=<integer>
admin-user=<string>
admin-password=<string>
ws-xss-protection-enabled=<boolean>

[JAVA]
vm=<pathname>
vmargs=<string>
vm3=<pathname>
native-vmargs=<string>
vmargs.<appname>=<string>

[MOCARPT]
app-server=<url>
report-server=<url>
archive-folder=<pathname>
cache-folder=<pathname>
db-date-time-format=<string>
db-init-command=<string>
default-locale-id=<string>
ems-attachment-format=<string>
ems-folder=<pathname>
lib-folder=<pathname>
failure-log-path=<pathname>
failure-log-keep-days=<integer>
local-db-connection=<string>

[MONITORING]
csv-reporter-enabled=<boolean>
csv-reporter-directory=<pathname>
csv-reporter-keep-hours=<integer>
csv-reporter-archive-directory=<pathname>
csv-reporter-archive-keep-hours=<integer>
csv-reporter-support-default-hours=<integer>
```

Environment Variables

Description

The following table lists the environment variables that can be used to configure or alter the runtime behavior of an environment.

Name	Purpose
`MOCA_DMALLOC`	Enables the C dynamically allocated memory debugger.

Name	Purpose
`MOCA_ENVNAME`	Name given to the environment.
`MOCA_REGISTRY`	Path name of the registry file.
`MOCA_TRACE_LEVEL`	Enabled trace level: • ***** = All trace levels run, with the exception of command profiling.

MOCA_DMALLOC Options

MOCA provides a conditional wrapper around the `malloc( )`, `calloc( )`, `realloc( )` and `free( )` system calls that can be enabled by setting the `MOCA_DMALLOC` environment variable. When this variable is set, MOCA intercepts these system calls and maintains a list of all nodes of memory allocated along with information regarding each node of memory. Using this information, MOCA can detect most issues related to dynamic memory including leaks, overwrites and double frees.

The following table lists the options that can be set using the `MOCA_DMALLOC` environment variable.

Name	Purpose
`on`	Turns on dmalloc (the C dynamically allocated memory debugger).
`deadbeeflen`	Defines length of the "dead beef" value that is written to the end of each allocated memory block. For example: `MOCA_DMALLOC=on,deadbeeflen=100`
`noabort`	Does not abort if a call to `malloc()`, `calloc()` or `realloc()` fails.
`nofreenull`	Complains if the application passes NULL to `free()`.
`nomemset`	Does not `memset()` memory blocks using the "dead beef" string.
`outfile`	Writes messages to the given output filename rather than to `stdout`. For example: `MOCA_DMALLOC=on,outfile=dmalloc.log`
`time`	Tracks the allocation time of each memory block rather than a call sequence number.
`trackcalloc`	Causes dmalloc to abort on the nth call to `calloc()`. For example: `MOCA_DMALLOC=on,trackcalloc=2348`
`trackmalloc`	Causes dmalloc to abort on the nth call to `malloc()`. For example: `MOCA_DMALLOC=on,trackmalloc=1026`
`trackrealloc`	Causes dmalloc to abort on the nth call to `realloc()`. For example: `MOCA_DMALLOC=on,trackrealloc=104`

Registry Includes

Description

You can include registry files in your MOCA registry. This lets you reuse entries across multiple MOCA-based environments that use the same entries.

IMPORTANT: This is only available for development environments.

Example: This code is an example of a registry include.

```
# include<$MOCADIR/data/registry.import>
```

Rules

There are no restrictions on file names. However, do not type text (other than spaces) in addition to the text listed in the example above.

You may add a registry include to a registry that is also a registry include in a different registry.

Registry includes are read in and applied in the order that they are listed in the registry. For example, if you type the registry include code at the bottom of the registry, the associated registry is included last.

Example

This is an example of a registry include file, the registry file that references the registry include file and the resulting registry entries.

Note: The registry is truncated for readability.

Example: This example illustrates the code for a registry include file that is stored in the $MOCADIR/data directory and named registry.import.

$MOCADIR/data/registry.import

```
[SERVER]
memory-file=$MOCADIR/data/commands.mem
prod-
dirs=$HOME/dev/tools/:$EMSDIR:$SALDIR:$MCSDIR:$MOCADIR:$MOCADIR/test
url=http://localhost:4500/service
port=4500
compression=true
rmi-port=4501
classic-port=4550
```

Example: This example illustrates the code for the registry file that includes registry.import.

$LESDIR/data/registry

```
#include<$MOCADIR/data/registry.import>

[JAVA]
vm=$HOME/Downloads/jre1.6.0_30/bin/java
vmargs=-Xmx512m -Djava.net.preferIPv4Stack=true
native-vmargs=-Djava.net.preferIPv4Stack=true -XX:-
HeapDumpOnOutOfMemoryError

[SERVER]
classic-port=5000
```

Example: This example illustrates the resulting registry entries. The registry.import entries are included in the appropriate section ([SERVER]) of the registry.

$LESDIR/data/registry

```
[JAVA]
vm=$HOME/Downloads/jre1.6.0_30/bin/java
vmargs=-Xmx512m -Djava.net.preferIPv4Stack=true
native-vmargs=-Djava.net.preferIPv4Stack=true -XX:-
HeapDumpOnOutOfMemoryError

[SERVER]
memory-file=$MOCADIR/data/commands.mem
prod-
dirs=$HOME/dev/tools/:$EMSDIR:$SALDIR:$MCSDIR:$MOCADIR:$MOCADIR/test
url=http://localhost:4500/service
port=4500
compression=true
rmi-port=4501
classic-port=5000
```

Hooks

MOCA Hooks

MOCA persistence hooks are coded using Java. Hooks are implemented using Spring-based XML configuration. Each `data` directory under the list of directories defined by the `server.prod-dirs` registry key is searched for a file named `hooks.xml`. If more than one file is listed, `hooks.xml` files that appear earlier in the list take precedence over files that appear later in the list.

Hooks are supported by providing a definition that has a class that implements one of the hooks' interfaces. For more information, see "Supported Hooks" (on page 40). Each type of hook either allows one or unlimited hooks on the MOCA server. This is done depending on the usual behavior of the hook. An example is `MessageResolver` that is used to provide custom error message text replacement. There can only be a single hook called for this since you would not want multiple message translations done on the same message.

Example:
This is an example of a Spring-based XML configuration file for `MessageResolver`.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean class="com.redprairie.mcs.hooks.MlsTextMessageResolver"/>
</beans>
```

This bean would be used since it implements the `MessageResolver` interface shown by the naming of the class.

Supported Hooks

This table lists the supported hooks' interface names, whether single or multiple hooks can be registered at a time and a brief description of their usage.

Interface	Single/Multiple	Usage Description
`MessageResolver`	Single	Called when an exception is raised to the caller to translate the message.
`DispatchActivityHook`	Single	Called when MOCA returns from executing a command from a client to log statistics of the command.
`QueryHook`	Single	Called when SQL is executed, and before and after results are returned. This hook must be very heavily evaluated before its use is implemented.
`MocaServerHook`	Multiple	Called when a server restart is requested. This is different from a restart of the Windows Service since it lets the server stay online.
`SupportHook`	Multiple	Called when a server support zip is generated. This adds the generated data to the support zip. For more information, see the "Support File Application Hooks" section of the *MOCA Console User Guide*.

MessageResolver Hook

The `MessageResolver` hook lets classes implementing its interface perform error message translation. Only a single `MessageResolver` hook can be registered at once, and it is called when an error code is returned to the caller.

```
public interface MessageResolver {
    /**
     * Looks up the given key in the message catalog.
     *
     * @param key the message key to look up in the message catalog.
     * @return a locale-specific message for the given message key.
     */
    public String getMessage(String key);
}
```

DispatchActivityHook

The `DispatchActivityHook` hook lets classes implementing its interface log specific command result statistics. Only a single `DispatchActivityHook` hook can be registered at one time, and it is called when a command is finished executing.

```
public interface DispatchActivityHook {
    /**
     * Called when a command is finished executing. Note that this method is
     * called outside of a normal MOCA transaction. Any database changes that
     * happen inside this hook must be done with commands that run inside their
     * own transaction.
     *
     * @param duration
     * @param command
     * @param env
     * @param errorCode
     * @param rowCount
     */
    public void activity(long duration, String command, Map<String, String> env, int
errorCode, int rowCount);
}
```

QueryHook Hook

The `QueryHook` hook lets classes implementing its interface intercept SQL statement execution and change the actual SQL statement. In addition, the result set can be manipulated before it is returned to the MOCA server using the hook.

IMPORTANT: If the `QueryHook` is not implemented correctly, there can be a negative impact to both the stability and performance of the system.

```
/**
 * A hook used to modify MOCA SQL queries.
 */
public interface QueryHook {
    /**
     * The query advisor interface. This interface is used by the MOCA SQL
     * adapter to allow modification of incoming queries as well as results. The
     * same advisor instance will be used for modifying the query as well as its
     * results, so if the implementation wants to retain information between
     * calls, it can.
     *
     * All of the methods of this interface are allowed to throw MOCAException
     * (not SQLException), to indicate an error that occurred with the query or
     * results. This can include application-level errors, and the correct error
     * code will be returned to the caller.
     */
    public interface QueryAdvisor {
        /**
         * Called before execution of a query. This method is actually called
         * before SQL translation, which means it will be in the "MOCA" database
         * dialect (simplified Oracle SQL).
         *
         * @param sql the query as passed to the MOCA SQL engine. This query may
         *            contain parameters (bind variables). This parameter will
         *            never be null.
         * @param bindList a list of bind variables to be used in this query.
         *         This parameter might be null.
         * @return a query to be used instead of the passed-in query. This
         *         method must not return <code>null</code>.
         * @throws MocaException if an error occurs.
         */
        public String adviseQuery(String sql, BindList bindList) throws MocaException;

        /**
         * Called after columns have been defined on the results object. The
         * <code>res</code> parameter is a mutable results object, and should be
         * modified in place. It is assumed that no columns are being removed
         * from the results, and that reordering of columns will not be done.
         *
         * @param res the results object to be operated on.
         * @throws MocaException if an error occurs.
         */
        public void adviseMetadata(EditableResults res) throws MocaException;

        /**
         * Called after each row is populated on the results object. The
         * <code>res</code> parameter is a mutable results object, and should be
         * modified in place. When the results object is passed to this method,
         * its current row (and edit row) are pointing to the last row in the
         * results, and it is assumed that the same is true when the method
         * completes. In particular, it must be possible to call
         * <code>addRow</code> on <code>res</code> after completion of this
         * method.
         *
         * @param res the results object to e operated.
         * @throws MocaException if an error occurs
         */
        public void adviseRowData(EditableResults res) throws MocaException;
    }

    /**
     * Returns an instance of a query advisor. The advisor returned is only used
     * for the execution of a single query and is not retained for additional
     * calls.
     *
     * @return an instance of the QueryAdvisor interface. This method must not
     *         return <code>null</code>.
     */
    public QueryAdvisor getQueryAdvisor();
}
```

MocaServerHook Hook

The `MocaServerHook` hook provides a method of notification when the server is restarted from the Console or by the **restart server** command that is available in the Development Tools component library. Restarting the server retains the running of the servlet but restarts important areas of the system, including the Native Process Pool as well as the Task and Job Managers. Developers that need to add an additional restart item to this list can implement this interface and put it into the `hooks.xml` file.

Example:
This is an example of the appropriate usage of the MocaServerHook hook that flushes the Hibernate second-level cache.

```
public interface MocaServerHook {
    /**
     * This method will be invoked as a callback whenever the
administrative
     * command to restart the MOCA server has been called.
     */
    public void onRestart();
}
```

Enable a Hook

Under some circumstances, it might be necessary to enable a MOCA hook.

To enable a hook:

1. Make sure that you have a hooks.xml in your %PRODUCTDIR%/data directory.
2. Add a bean to the hooks.xml with your class that *implements the hook interface*.

Example:
This is an example of a hooks.xml file with an enabled DispatchActivityHook.

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">
<bean class="com.redprairie.product.hooks.MyDispatchActivityHook"/>
</beans>
```

Disable a Hook

Under some circumstances, it might be necessary to disable a MOCA hook.

To disable a hook:

1. Set the hook in the LES hooks.xml to a default hook. MOCA gives a default (null) hook for DispatchActivityHook and MessageResolver.
2. If you are disabling a hook type other than DispatchActivityHook or MessageResolver, you must implement the hook type's default Hook. Implement the interface and have the methods do nothing.

Example:
This is an example of the NullDispatchActivityHook.

```
public class NullDispatchActivityHook implements DispatchActivityHook
{

    @Override
    public void activity(long duration, String command, Map<String,
String> env, int errorCode, int rowCount) {
        // Do nothing
    }
}
```

Example:
This is an example of the default MessageResolver.

```
public class DefaultMessageResolver implements MessageResolver {

    // @see
com.redprairie.moca.server.dispatch.MessageResolver#getMessage
(java.lang.String)
    @Override
    public String getMessage(String key) {
        return null;
    }
}
```

Example:
This is an example of a %LESDIR%\hooks.xml file with a disabled DispatchActivityHook and MessageResolver.

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">
    <bean
class="com.redprairie.moca.server.dispatch.NullDispatchActivityHoo
k"/>
    <bean class="com.redprairie.moca.server.DefaultMessageResolver"/>
</beans>
```

Jetty

Description

An optional Jetty configuration file named `jetty.xml` can be used to configure Jetty, the embedded servlet container that is used by the MOCA server. The directories listed in the `server.prod-dirs` registry key are searched in order for a `jetty.xml` file, and the first file found in that directory list is used.

Developers should not need any of the additional Jetty functionality provided by using a `jetty.xml` file. However, users that want to use HTTPS should copy the sample `jetty.xml` file from `%MOCADIR%\samples\data` to their `%LESDIR%\data` directory as the basis for configuring a Jetty `ServerConnector` to support HTTPS.

Note: Contact Product Development for additional information on how to configure a MOCA server to use HTTPS.

Servlet Handler Configuration

The servlet container for the default /service and /console endpoints can both be configured through the `jetty-handler.xml` file. Similar to configuration using the `jetty.xml` file, MOCA searches the directories listed in the `server.prod-dirs` registry key, in the sequence listed. MOCA then uses the first found instance of the `jetty-handler.xml` file for the configuration.

There is a distributed example of the `jetty-handler.xml` file in the `%MOCADIR%\samples\data` directory. The example has a block that illustrates the configuration for a Denial of Service filter. This filter is useful in systems that have too many requests come in at the same time from a particular client; it lets you throttle connections if they are producing too many requests per second.

The `jetty-handler.xml` file follows the standard Jetty configuration .xml file format. For more information, see the Jetty/Reference/jetty.xml syntax documentation website.

The configuration xml is used to configure a given instance of a `org.eclipse.jetty.servlet.ServletContextHandler`. See the Jetty Javadoc for the list of operations can be performed on `org.eclipse.jetty.servlet.ServletContextHandler`.

Remote Method Invocation

Remote Method Invocation (RMI) is currently only configured through the `server.rmi-port` registry key. For more information, see "Registry File" (on page 12).

Chapter 4. Starting MOCA

Description

This section explains how you start a MOCA server on Linux/UNIX and Windows platforms.

Bootstrapping the Environment

Description

With the exception of starting the MOCA server as a Windows service, you must bootstrap your environment before trying to start the MOCA server. The steps to do this depend on whether you are working in an environment that was built from source code (a development environment) or one that was installed from binaries (an installed environment).

Development Environments

Development environments are bootstrapped the same regardless of the platform (Linux/UNIX or Windows). The `rpset` script takes a single argument, the environment name, and uses it to look up information about the environment in the `rptab` file, which is then used to perform the actual bootstrapping of the environment.

- On Linux/UNIX platforms the `rpset` script is located in the `/usr/local/bin` directory tree and is in the user's search path.
- On Windows platforms the `rpset` script is distributed with the Development Tools and is in a user's search path. This example shows the bootstrapping of an environment named "moca-dev" on a Windows platform.

```
c:\dev:> rpset moca-dev

Setting up the environment for the "moca-dev" instance...
Setting environment for using Microsoft Visual Studio 2008 x86 tools.

        Instance: moca-dev

        Registry: c:\dev\trunk\moca\data\registry

             URL: http://localhost:4500/service

   Database User: dev
             URL: jdbc:sqlserver://localhost;databaseName=dev

             LES: c:\dev\trunk\env
            MOCA: c:\dev\trunk\moca

moca-dev C:\dev\trunk\env>
```

Installed Environments

Installed environments are bootstrapped differently on Linux/UNIX and Windows platforms. Installed Windows environments are bootstrapped differently because the Development Tools are not available for installed environments.

You bootstrap an installed environment on Linux/UNIX platforms using the same steps as is taken for development environments. For more information, see "Development Environments" (on page 47).

On a Windows platform, a batch file `env.bat` that sets a number of environment variables is used for installed environments. The `env.bat` file is generated by the MOCA `servicemgr` tool and typically is located in `%LESDIR%` (but could exist anywhere under the install directory tree). Any changes to the environment require a new `env.bat` file to be generated, so it is suggested that you generate a new one if you are unsure that the current file is up to date.

This example generates a new `env.bat` file for an environment named "prod", copies it to `%LESDIR%` and then calls it to bootstrap an environment.

```
C:\> cd \redprairie\prod\moca\bin
C:\redprairie\prod\moca\bin> servicemgr -e prod dump

Looking for environment prod in rptab... found
Looking for a registry file...
Checking c:\redprairie\prod\les\data\registry.prod ... not found
Checking c:\redprairie\prod\les\data\registry... found
Creating env.bat file for environment...

    Environment: prod
       Registry: c:\redprairie\prod\les\data\registry

Done

C:\redprairie\prod\moca\bin> move env.bat \redprairie\prod\les
C:\redprairie\prod\moca\bin> cd \redprairie\prod\les
C:\redprairie\prod\les> env

set MOCA_ENVNAME=prod
set MOCA_REGISTRY=c:\redprairie\prod\moca\data\registry
.
.
.
set
CLASSPATH=c:\redprairie\prod\moca\build\classes;c:\redprairie\prod\mo
ca\lib\*;c:\redprairie\prod\moca\javalib\*;

c:\redprairie\prod\moca\build\classes;c:\redprairie\prod\moca\lib\*;c
:\redprairie\prod\moca\javalib\*;.

C:\redprairie\prod\les>
```

Starting the MOCA Server from the Command Line

Description

If you want to start the MOCA server from the command line using parameters from the registry file and default values for everything else, you can simply run `mocaserver` from the command line. During development, however, there are a number of ways that you may want to start the MOCA server. Running `mocaserver -h` displays the full command line usage. Some examples of the most common scenarios are to redirect the output to a log file and to use different port numbers. Command line arguments can be combined as necessary.

Redirecting Output to a Log File

Both during development and while troubleshooting an issue, you can enable tracing and redirect log messages to a file. The `-t` command line argument enables tracing, and the `-o` command line argument provides an output file to which to redirect messages.

```
mocaserver -t -o mocaserver.log
```

Note: The `-t` argument still accepts a parameter for backwards compatibility, but the argument is ignored; tracing is enabled regardless of the argument (if any) that is included after the `t`. See "Tracing and Logging" (on page 53).

Using Different Port Numbers

The port number that the MOCA server listens on is configured using this precedence:

1. The value provided by the `-p` command line argument
2. The `jetty.xml` (if one exists in the `data` directory under the list of path names defined by the `server.prod-dirs` registry key value)
3. The `server.port` registry key value
4. The default value of "4500"

The server port number needs to match the url port. To override url registry setting you can use -u command-line argument.

If you are trying to start more than one MOCA server in the same environment, then you also need to make sure that each mocaserver uses unique listening ports.

- The `-p` command line option can be used to change the server port number
- The `-r` command line argument can be used to change the RMI server's port number
- The `-P` command-line option is used to change the classic port.

The MOCA server also opens several additional listening ports, but those ports are usually dynamic and don't need to be overriden from command-line.

```
mocaserver -p 9900 -r 9901 -P 9902 -u http://testbox:9900/service
```

Disabling Tasks and Jobs

With tracing turned on, the output from tasks and jobs can fill an output file quickly. If you are trying to troubleshoot an issue unrelated to a task or job, you can disable the task or job using the `-J` and `-T` command line options.

```
mocaserver -JT
```

Starting the MOCA Server as a UNIX Service

An `rp` script is installed under `/usr/local/bin` on the Linux and UNIX platforms that can be used to start and stop the MOCA server. The script starts the MOCA server in the background using the UNIX **nohup** command so logging out does not shut down the process, and all output is redirected to `$LESDIR/log/mocaserver.log`.

```
rp start
```

Starting the MOCA Server as a Windows Service

Description

The MOCA server can be configured to run as a Windows service using the `servicemgr.exe` tool that should be present in your `%MOCADIR%\bin` directory after MOCA is installed or built. Descriptions of what functionality the `servicemgr.exe` supports follow.

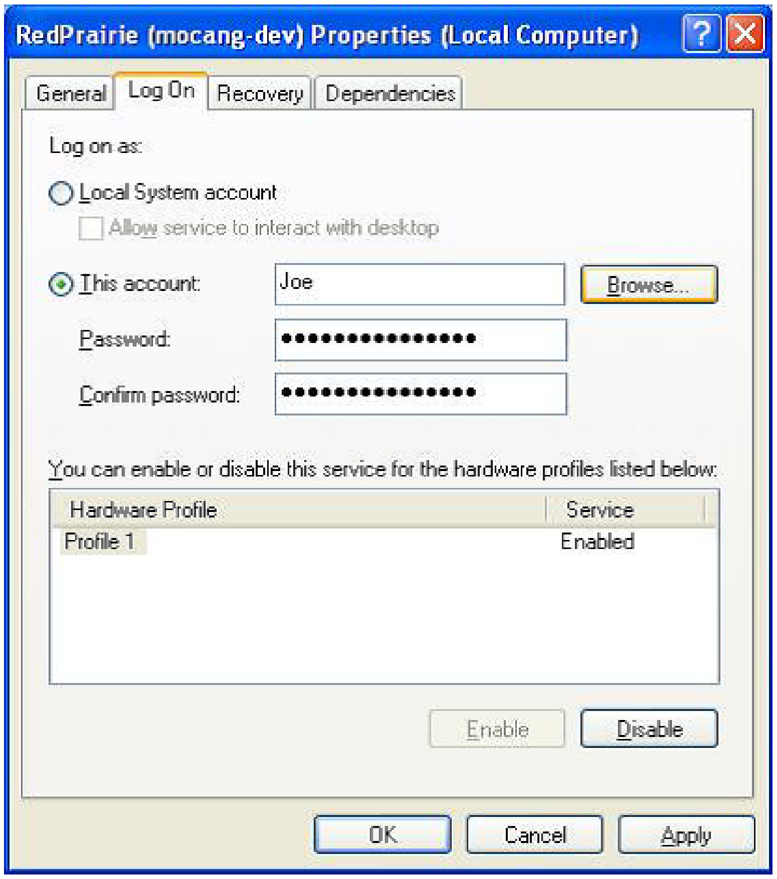
Installing/Uninstalling an Instance as a Windows Service

The MOCA server can be installed as a Windows service using this syntax: `servicemgr -e` *environment_name* `install`.

After installing the MOCA server as a Windows service, it is visible in the Windows Services tool. To view the MOCA server in the Services tool, from the **Start** menu, display the list of programs, then select **Administrative Tools**, and then select **Services**. Then on the Services window, scroll to the MOCA server. For example: MOCA Server (*environment_name*)

Modifying the "Log on as" User for the Service

By default the service runs under the Local System Account. You can change this setting by modifying the log on settings for the MOCA server using the Windows Services tool. To modify the log on settings, on the Services window, select the MOCA server, right-click the MOCA server name, and then from the shortcut menu select **Properties**. On the Properties window that appears, click the **Log On** tab and in the **This account** box, type a valid username and in the **Password** box type a password and confirm the password as shown in this image.



Log on as:

Local System account

Allow service to interact with desktop

[✓ This account:] Joe

Password: •••••••••••••••

Confirm password: •••••••••••••••

You can enable or disable this service for the hardware profiles listed below:

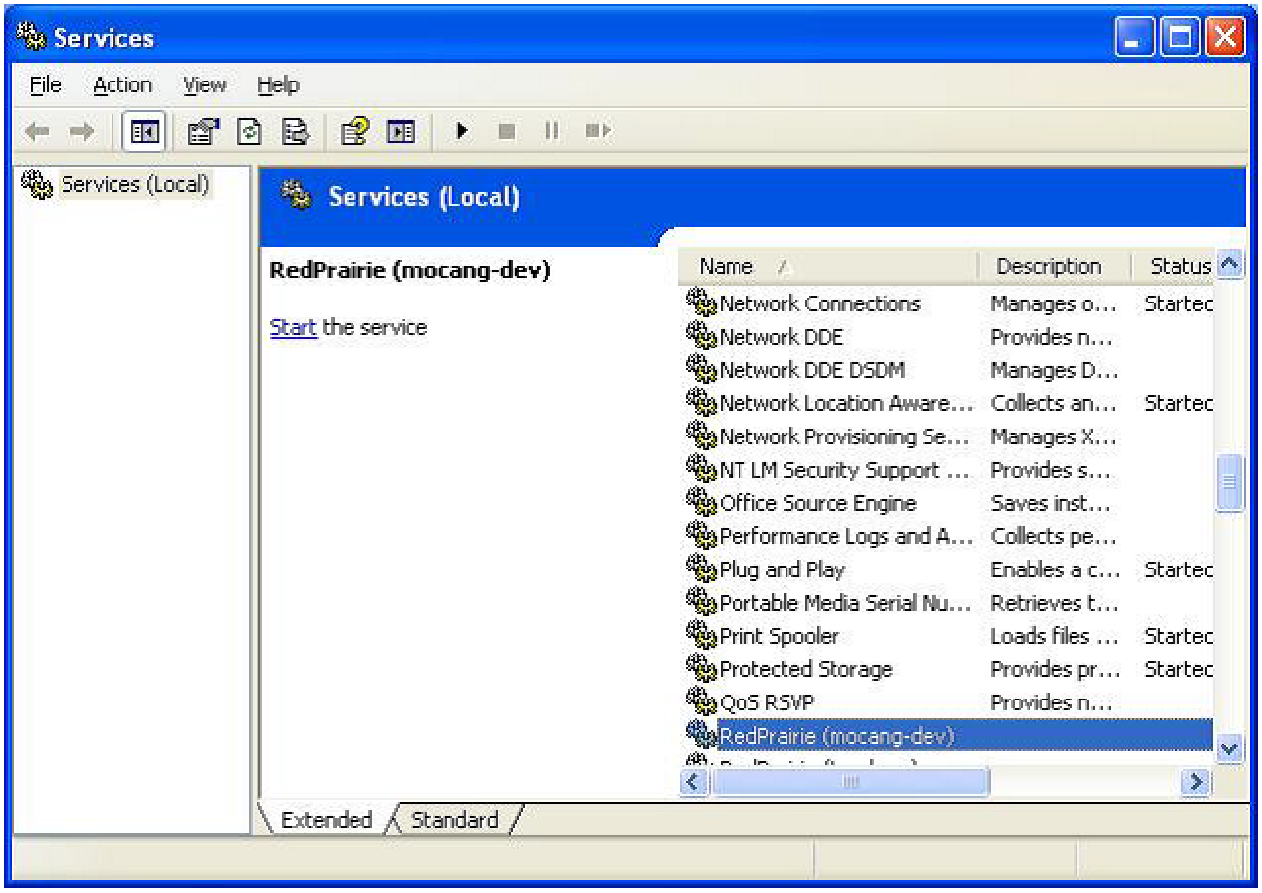
Hardware Profile	Service
Profile 1	Enabled

The `service.command` registry key value can be used to change the command that is started when the Windows service starts and provides a way, for example, to pass command line arguments to the `mocaserver` executable. The `service.output` registry key can be used to redirect output to a log file. By default output is not redirected to any log files. For more information on the `service.command` and `service.output` registry key values, see "Configuration" (on page 11).

Starting and Stopping an Installed Windows Service

You can start or stop the installed Windows service by either using the Windows Services tool or from the command line.

To start or stop the service using the Service tool, on the Services window, in the grid, select the service, and then click **Start** or **Stop** the service. This image is an example of the Services tool window.



RedPrairie (mocang-dev)

Start the service

Name	Description	Status
Network Connections	Manages o...	Startec
Network DDE	Provides n...	
Network DDE DSDM	Manages D...	
Network Location Aware...	Collects an...	Startec
Network Provisioning Se...	Manages X...	
NT LM Security Support ...	Provides s...	
Office Source Engine	Saves inst...	
Performance Logs and A...	Collects pe...	
Plug and Play	Enables a c...	Startec
Portable Media Serial Nu...	Retrieves t...	
Print Spooler	Loads files ...	Startec
Protected Storage	Provides pr...	Startec
QoS RSVP	Provides n...	
RedPrairie (mocang-dev)		

To start the service from a command line, open a command line prompt window and type this command:

```
net start MOCA.<environment_name>
```

To stop the service from a command line, open a command line prompt window and type this command:

```
net stop MOCA.<environment_name>
```

Event Messages

The Service Manager outputs event messages when:

- A user successfully starts an application.
- A startup attempt is unsuccessful.
- The registry file is read.
- An application is shut down.

You can view messages on the Windows Event Viewer. To open the Event Viewer, from the **Start** menu, display the list of programs, then select **Administrative Tools**, and then select **Event Viewer**. On the Event Viewer window, double-click the event log that you want to view. Event messages are displayed in the log as an Application type, and the Source is the name of the service. For example: MOCA.*environment_name*

Chapter 5. Tracing and Logging

Introduction

MOCA uses the Apache Log4j 2 logging facility to support logging and tracing. This logging facility allows fine-grained and configurable logging in addition to supporting custom layouts and appenders. For more information on Log4j 2, see the Apache Log4j 2 website.

Trace Messages

Every trace message written by MOCA consists of a pre-defined set of fields that provide information in addition to the message itself. This is an example of a set of messages written by the default configuration to a log file.

Example Trace

```
2013-10-23 11:55:13,495 DEBUG [94  eff4] CommandDispatcher [0]
Dispatching command... []
2013-10-23 11:55:13,495 DEBUG [94  eff4] CommandDispatcher [0] Server
got: noop []
2013-10-23 11:55:13,496 DEBUG [94  eff4] DefaultServerContext [0]
Parsing command...  []
2013-10-23 11:55:13,496 DEBUG [94  eff4] DefaultServerContext [0]
Parsed command []
2013-10-23 11:55:13,497 DEBUG [94  eff4] DefaultServerContext [0]
Executing... []
2013-10-23 11:55:13,497 DEBUG [94  eff4] DefaultServerContext [1]
Looking up command: noop []
2013-10-23 11:55:13,497 DEBUG [94  eff4] DefaultServerContext [1]
Executing built-in command: noop []
2013-10-23 11:55:13,497 DEBUG [94  eff4] MocaTransactionManager [0]
Commit  []
2013-10-23 11:55:13,497 DEBUG [94  eff4] MocaTransactionManager [0]
No current transaction []
2013-10-23 11:55:13,498 DEBUG [94  eff4] CommandDispatcher [0]
Returning 0 row(s)  []
```

This table describes the fields (in order) of a trace message.

Field	Description
Time Stamp	Date and time when the message was written. For asynchronous logging, this is the time when the message was first logged; the time when the message could actually be written to the destination may be much later.
Level	Abbreviation of the Log4j 2 level of the message. See "Trace Levels" (on page 54).
Thread Id	Thread ID from which the message initiated.
Session Id	Last four characters of the session from which the message initiated. A session can span multiple threads.
Logger Name	Name of the logger from which the message initiated.

Field	Description
Stack Level	Level of the stack from which the message initiated.
Message	Actual message.
Blank identifier	A set of brackets for custom identifiers

Trace Levels

MOCA defines a set of trace levels and, similarly, Log4j 2 defines levels to show the importance of the message. The Log4j 2 levels are `TRACE`, `DEBUG`, `INFO`, `WARN`, `ERROR` and `FATAL`. By default the MOCA server only writes messages for MOCA-based products from levels `INFO` and higher. Messages for other third-party products are only written for levels `WARN` and higher. Messages initiated from the `misLogError`, `misLogWarning`, and `misLogInfo` C functions are written at the `ERROR`, `WARN`, and `INFO` logging levels, respectively. Messages initiated when one of the MOCA trace levels are enabled, are written at the `TRACE` and `DEBUG` levels.

With the move to Log4j 2, MOCA eliminated all MOCA trace levels, except *, thus simplifying the use of MOCA. Any value that is passed in as the trace level argument, other than `null`or the empty string, enables tracing. Passing `null` or the empty string disables tracing.

The following table lists the text that is written to the log file to represent each Log4j 2 level and describes the level.

Level	Description
`DEBUG`	General purpose DEBUG message. `DEBUG` is the default level used for any logging done by MOCA.
`INFO`	Informational message that is displayed when tracing is disabled. Informational messages are not errors, but provide details, such as the port number of the running MOCA server.
`WARN`	Warning message that indicates that something may be impeding the operation of the component or server.
`ERROR`	Error message that indicates that a segment of the system has encountered an issue that caused the system to not work properly. However, the system is still able to continue running.

Enabling Logging or Tracing

MOCA supports the following types of logging:

- **Server logging**: Includes trace messages that are logged from the entire MOCA server. This type of logging can be enabled by passing the `-t` command line argument to the MOCA server; for example, `mocaserver -t`. Alternatively for server-wide tracing, changes can be made to the `logging.xml` file in **%LESDIR%**.
- **Session logging**: Only includes trace messages that are logged for a single session. Session logging can be turned on and off using the **set trace** command or, from the JDA SCE client, by selecting **Tracing** from the **Tools** menu. Enabling session logging is useful when debugging an issue that is specific to one session; for example, debugging the failure of the loading of a particular form in the JDA SCE client.

IMPORTANT: If you need to make changes to the logging configuration, make those changes to the **%LESDIR%/data/logging.xml** file. MOCA merges all of the **logging.xml** files in the application directories into a single **runtime-logging.xml** file, which is used to configure Log4j 2. Do not edit the **%MOCADIR%/data/logging.xml** file.

Server Tracing

To enable MOCA tracing, run `mocaserver` with the `-t` argument. This command and argument writes `DEBUG` level and higher log messages to the MOCA standard output. To save this logging to a file, you can pass the `-o` argument with a log file name or pipe the standard output to a file. The `-t` argument used to be `-t*` in previous releases, so starting with the 2013.2 release, any character after the `t` is ignored for backwards compatibility.

Example: The following commands are examples of how to start the MOCA server with tracing to a file.

Server Trace Example

```
mocaserver -t -o /tmp/mylocation/mylog.log
mocaserver -t 2>&1 > /tmp/mylocation/mylog.log
```

If you are starting MOCA as a service on Windows, output is written to the following files by default:

- `%LESDIR%/log/<application name>.<environment name>.YYYY-MM-DD.log`
- `%LESDIR%/log/<application name>.<environment name>-stdout.YYYY-MM-DD.log`
- `%LESDIR%/log/<application name>.<environment name>-stderr.YYYY-MM-DD.log`

If you need to enable tracing or add a custom appender for MOCA and you cannot start it manually, then logging should be configured directly in the logging.xml files. For more information about running MOCA as as service, see "Service Manager" (on page 160).

To enable tracing via XML configuration, edit the **%LESDIR%/data/logging.xml** file and override the `com.redprairie` logger to have a `DEBUG` level. You can also add a file appender at the same time, which would direct the output to a file instead of the MOCA standard output.

Example: The following XML code is an example of a **%LESDIR%/data/logging.xml** configuration to enable MOCA server tracing and output it to a file.

%LESDIR%/data/logging.xml - Configuration for Server Logging Full Example

```
<?xml version="1.0" encoding= "UTF-8" ?>
<configuration packages="com.redprairie.moca.server.log" monitorInterval="30">
    <appenders>
        <File name="MYTRACEFILE" fileName="${env:LESDIR}/log/trace.log">
            <PatternLayout>
                <pattern>%d{DEFAULT} %-5p [%-3T %-4.4X{moca-session}] %c{1} [%X{moca-
stack-level}] %m []%n</pattern>
            </PatternLayout>
        </File>
    </appenders>
    <loggers>
        <asyncLogger name="com.redprairie" level="DEBUG">
            <appender-ref ref="MYTRACEFILE"/>
        </asyncLogger>
    </loggers>
</configuration>
```

You can also set up a daily rolling file appender with compressed archives, which stores the MOCA output in 24 hour compressed chunks. Storing the log file information this way can be helpful when diagnosing difficult issues that require long-term logging. To disable the compression feature, delete `.gz` from the end of the `filePattern` attribute. For other rollover behavior options, see the Apache Log4j 2 documentation.

Example: The following XML code is an example of a **%LESDIR%/data/logging.xml** configuration to roll the output of the mocaserver log and compress it automatically.

%LESDIR%/data/logging.xml - RollingFile Server Logging Full Example

```
<?xml version="1.0" encoding= "UTF-8" ?>
<configuration packages="com.redprairie.moca.server.log"
monitorInterval="30">
    <appenders>
<RollingFile name="TraceFileAppender"
fileName="${env:LESDIR}/log/server.log"
filePattern="${env:LESDIR}/log/mocaserver-%d{yyyy-MM-dd-
HH}-%i.log.gz">
    <PatternLayout>
        <pattern>%d{DEFAULT} %-5p [%-3T %-4.4X{moca-session}] %c{1}
[%X{moca-stack-level}] %m []%n</pattern>
    </PatternLayout>
    <Policies>
        <TimeBasedTriggeringPolicy interval="24" modulate="false"/>
    </Policies>
</RollingFile>
</appenders>
    <loggers>
        <asyncLogger name="com.redprairie" level="DEBUG">
            <appender-ref ref="TraceFileAppender"/>
        </asyncLogger>
    </loggers>
</configuration>
```

Third-Party Tracing

Trace messages issued from third-party software (such as Hibernate, Spring, and Jetty) can also be captured by configuring an appropriate logger in the **%LESDIR%/data/logging.xml** file. For example, to enable JGroups tracing, add a logger for the `org.jgroups` package.

Example: The following XML code is an example of a **%LESDIR%/data/logging.xml** configuration to enable third-party logging for JGroups.

%LESDIR%/data/logging.xml - 3rd Party Tracing Full Example

```
<?xml version="1.0" encoding= "UTF-8" ?>
<configuration packages="com.redprairie.moca.server.log"
```

```
monitorInterval="30">
    <loggers>
        <asyncLogger name="org.jgroups" level="DEBUG"/>
    </loggers>
</configuration>
```

To enable third-party logging to a file, you must configure a logger and a corresponding custom appender in the **%LESDIR%/data/logging.xml** file. For example, the following XML configuration enables JGroups logging to a file, and Hibernate logging to a file that is limited to 10MB (a number of previous versions are kept as well). Since additivity is set to`false` on both loggers, neither of them log to the main MOCA output.

Example: The following XML code is an example of a **%LESDIR%/data/logging.xml** configuration to enable third-party logging to a file and a rolling file.

%LESDIR%/data/logging.xml - 3rd Party Appender Full Example

```
<configuration packages="com.redprairie.moca.server.log"
monitorInterval="30">
    <appenders>
        <File name="MYTRACEFILE" fileName="${env:LESDIR}/log/jgroups-
trace.log">
            <PatternLayout>
                <pattern>%d{DEFAULT} %-5p [%-3T %-4.4X{moca-session}]
%c{1} [%X{moca-stack-level}] %m []%n</pattern>
            </PatternLayout>
        </File>
        <RollingFile name="MYROLLINGFILE"
fileName="${env:LESDIR}/log/hibernate-trace.log"
filePattern="${env:LESDIR}/log/hibernate-trace-yyyy-%d{MM-dd-
yyyy}-%i.log">
            <PatternLayout>
                <pattern>%d{DEFAULT} %-5p [%-3T %-4.4X{moca-session}]
%c{1} [%X{moca-stack-level}] %m []%n</pattern>
            </PatternLayout>
            <Policies>
                <SizeBasedTriggeringPolicy size="10MB"/>
            </Policies>
        </RollingFile>
    </appenders>
    <loggers>
        <asyncLogger name="org.jgroups" level="DEBUG"
additivity="false">
            <appender-ref ref="MYTRACEFILE"/>
        </asyncLogger>
        <asyncLogger name="org.hibernate" level="DEBUG"
additivity="false">
            <appender-ref ref="MYROLLINGFILE"/>
        </asyncLogger>
    </loggers>
</configuration>
```

For reference, the following table lists other commonly used third-party products and their associated package names.

Product	Package	Usage
Google	`com.google`	Guava Java collections library.
Hibernate	`org.hibernate`	Object-relational mapping library. For more information, see "Caching" (on page 197).
Infinispan	`org.infinispan`	Clustered caches. For more information, see "Caching" (on page 197).
Jetty	`org.eclipse.jetty`	Web server. For more information, see "Web Services" (on page 215).
JGroups	`org.jgroups`	Clustering. For more information, see "Clustering" (on page 181).
MSSQL	`com.microsoft`	SQL Server JDBC library.
Quartz	`org.quartz`	Job scheduling. For more information, see "Jobs" (on page 138).
Spring	`org.springframework`	Dependency injection and web services. For more information, see "Web Services" (on page 215).

Developers can also enable messages conditionally from their own packages by using the package or class names of their own code. For example, to enable messages from Transportation Management components, you could add an entry to the `logging.xml` file with a logger name of com.redprairie.tm.`components` and define the level accordingly.

Jobs and Tasks Tracing

The easiest way to enable tracing for a job or thread-based task is to enable it in the MOCA Console on the Jobs or Tasks pages. These pages allow the user to get `DEBUG` or `INFO` level logging for a job or task and optionally output the log messages to a file. For more information, see "Jobs" (on page 138) and "Tasks" (on page 145).

Sometimes you want to get `DEBUG` level tracing from a job or task, but you are afraid that `DEBUG` level tracing creates too much logging and fills the disk. To help with this scenario, it is possible to configure a custom appender for a job or task, which allows you to specify an appender that has a size-based or time-based rollover policy.

IMPORTANT: To configure a custom appender for a job or thread-based task, you must define a custom appender and a route with a key that matches the file name configured for the job or task.

The following is an example of a custom appender for a job. You notice that there are some other XML elements required to make this work due to the fact the the **%LESDIR%/data/loggging.xml** file is being merged with other XML files to produce the final XML configuration file. The parent XML elements for the route should be the same as those in the **%MOCADIR%/data/logging.xml** file.

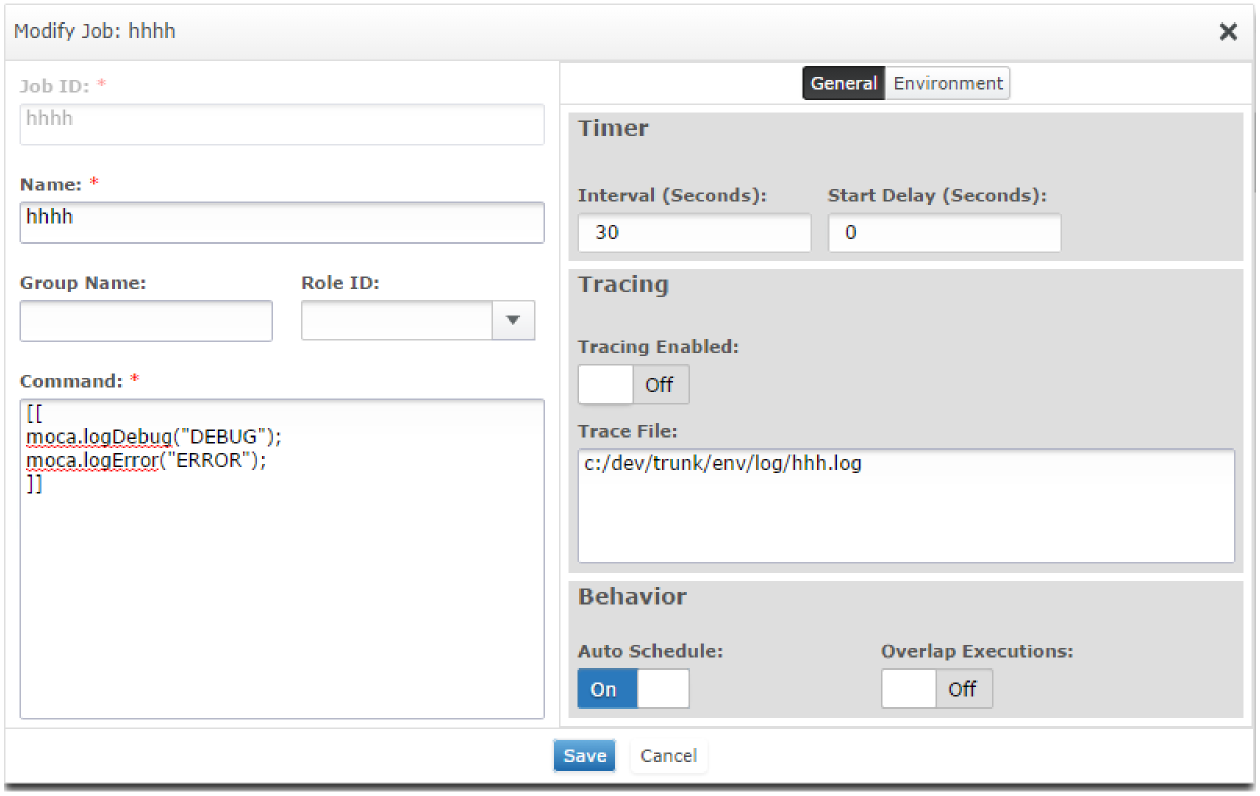
Note: The `key` of the `Route` must exactly match the `log_file` of the job definition in the database to have a custom appender. This means that when you configure the trace file for the job in the MOCA Console, the value must be the same as the `Route`'s `key` letter-for-letter. Technically, these two strings do not have to be a path to a file, since the actual file name for the log comes from the appender's `filename` and `filePattern` attributes. However, it is good practice to use an actual path here so that if the custom appender configuration is removed, then the job or task is still properly configured to generate logging to a valid location.

Example: The following XML code is an example of a **%LESDIR%/data/logging.xml** configuration for a custom rolling file appender for a job.

%LESDIR%/data/logging.xml - Job Appender Full Example

```
<configuration packages="com.redprairie.moca.server.log"
monitorInterval="30">
    <appenders>
        <MocaRouting name="RoutingAppender">
            <Routes pattern="$${ctx:moca-trace-file}">
                <Route key="c:/dev/trunk/env/log/hhh.log">  <!-- The
"key" here must match the job's/task's "log_file" EXACTLY -->
                    <RollingFile name="MYROLLINGFILE"
                              fileName="${env:LESDIR}/log/hhhh.log"
                           filePattern="${env:LESDIR}/log/hhhh-%d{MM-
dd-yyyy}-%i.log.gz">
                        <PatternLayout>
                            <pattern>%d{ISO8601} %-5p [%-3T %-4.4X
{moca-session}] %c{1} [%X{moca-stack-level}] %m []%n</pattern>
                        </PatternLayout>
                        <Policies>
                            <SizeBasedTriggeringPolicy size="10MB"/>
                        </Policies>
                        <DefaultRolloverStrategy max="20"/>
                    </RollingFile>
                </Route>
            </Routes>
        </MocaRouting>
    </appenders>
</configuration>
```

Example: The following image illustrates the configuration in the MOCA Console of the job that corresponds to the XML configuration for the job's rolling file appender.


Modify Job: hhhh

General | Environment

Job ID: * hhhh

Name: * hhhh

Group Name:

Role ID:

Command: *

```
[[
moca.logDebug("DEBUG");
moca.logError("ERROR");
]]
```

Timer

Interval (Seconds): 30

Start Delay (Seconds): 0

Tracing

Tracing Enabled: Off

Trace File: c:/dev/trunk/env/log/hhh.log

Behavior

Auto Schedule: On

Overlap Executions: Off

Client Tracing

With Log4j 2, client tracing is either enabled or disabled. You cannot specify a level of tracing when initiating a client trace. If you set the trace level to a non-zero positive number, you enable all tracing. Also, starting with the 9.0.0.0 release all logging and tracing is asynchronous. To handle asynchronous client tracing, MOCA only guarantees that all messages will be in the client trace file after the trace is closed; that is, after the `set trace where activate = 0` command is executed. There is special logic in MOCA to make this guarantee, and beyond closing the trace, the client cannot expect any or all of the trace messages to actually be in the trace file yet (depending on system load).

MOCA Activity Logger

A MOCA activity logger logs information about all running commands. It can be added to the core MOCA server log file or a separate log file, similar to other loggers using XML configuration.

The following information is logged by the activity logger:

- Run duration of a command
- Actual command
- Runtime environment
- Error code returned
- Number of returned rows

Example: The following text is an example of the logging generated by the activity logger.

```
2013-10-23 11:55:13,498 DEBUG [94  eff4] Activity [0] command="noop"
duration="3" env="{WEB_CLIENT_ADDR=127.0.0.1, USR_ID=super, WEB_
SESSIONID=b592b264cdf9eff4, MOCA_APPL_ID=msql, LOCALE_ID=US_ENGLISH}"
error="0" rowCount="0" []
```

Example: The following XML code is an example of the configuration for generating a separate MOCA activity log file.

%LESDIR%/data/logging.xml - Activity Logger Configuration Full Example

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration packages="com.redprairie.moca.server.log"
monitorInterval="30">
    <appenders>
        <File name="ActivityAppender"
fileName="${env:LESDIR}/log/activity.log">
            <PatternLayout>
                <pattern>%d{DEFAULT} %-5p [%-3T %-4.4X{moca-session}]
%c{1} [%X{moca-stack-level}] %m []%n</pattern>
            </PatternLayout>
        </File>
    </appenders>
    <loggers>
        <asyncLogger name="com.redprairie.moca.Activity"
level="INFO">
            <appender-ref ref="ActivityAppender"/>
        </asyncLogger>
    </loggers>
</configuration>
```

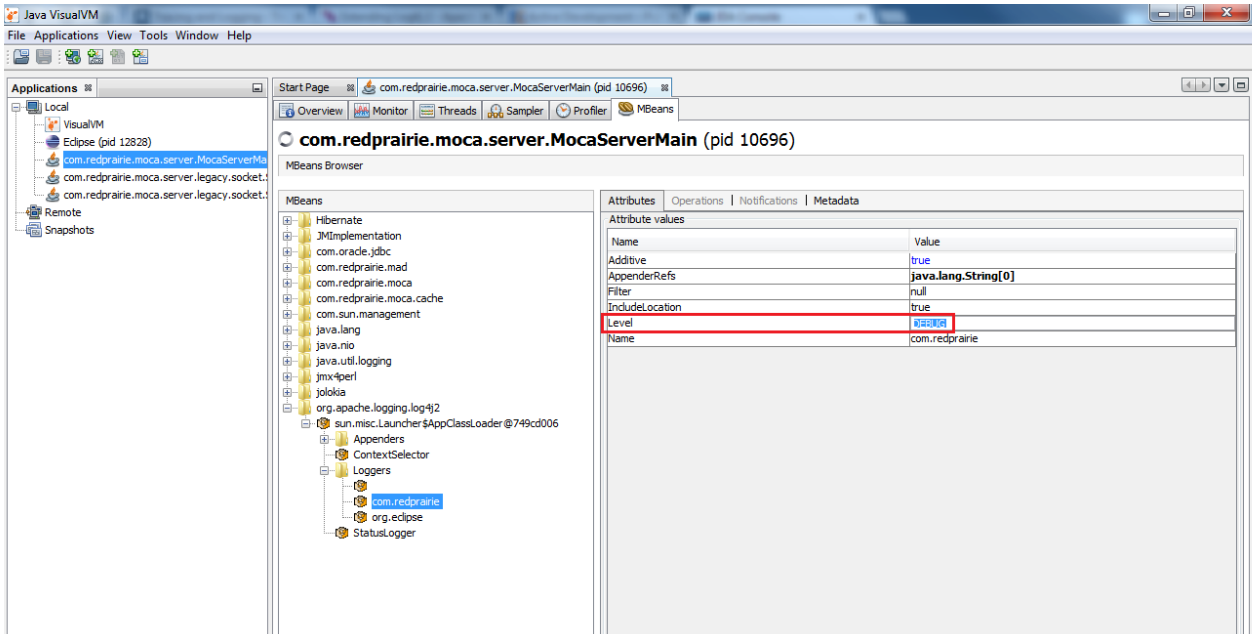
Run-Time Logging Level Configuration

You can change Log4j 2 logging levels at run time, which allows you to enable low-level tracing without restarting the application. You may want to do this type of tracing to troubleshoot an issue that might no longer be present if you restart the application. You can use one of the following methods to change the logging level at run time for a logger:

- **Editing the runtime-logging.xml**: %LESDIR%/data/runtime-logging.xml is the logging configuration for MOCA. This method of run-time configuration is more flexible than other methods. You can make a simple change, such as changing the levels on a logger, or a complex change, such as redesigning the whole configuration. Log4j 2 listens for changes to this file and updates the configuration automatically after a change is made.

  Note: Log4j 2 relies on file timestamps to read in new configurations. If you are copying over an older version of runtime-logging.xml, make sure to update the last-modified date on the file, such as with the touch command.

- **Configuration change through JMX**: Log4j 2 exposes a JMX MBean `org.apache.logging.log4j2:type=sun.misc.Launcher$AppClassLoader` that allows changes to the configuration. This method is limited, but is easier to use if you know how to use JMX tools. To change the logging level, find the appropriate logger and edit the `Level` field. Make sure that you correctly spell the level.



Name	Value
Additive	true
AppenderRefs	java.lang.String[0]
Filter	null
IncludeLocation	true
Level	DEBUG
Name	com.redprairie

Frequently Asked Questions

How do I prevent logging from filling up the disk?

If server logging is configured to be very verbose, it is possible that it could fill up the disk eventually. To prevent this, MOCA logging can be configured to be split into chunks and compressed. To do this, you must override the console appender that normally writes the logging to standard output.

Remember that editing the main MOCA logging XML is not allowed, so the work has to be done in **%LESDIR%/data/logging.xml**.

Example: The following example is an example of a **%LESDIR%/data/logging.xml** configuration to save all server output in 24 hour increments and compress them.

%LESDIR%/data/logging.xml - Rolling Server Logging Full Example

```
<configuration packages="com.redprairie.moca.server.log"
monitorInterval="30">
    <appenders>
        <MocaRouting name="RoutingAppender">
            <Routes pattern="$${ctx:moca-trace-file}">
                <Route key="${ctx:moca-trace-file}">
                    <RollingFile name="SERVERAPPENDER"
fileName="${env:LESDIR}/log/server.log"
filePattern="${env:LESDIR}/log/server-%d{yyyy-MM-dd-HH}-%i.log.gz">
                        <PatternLayout>
                            <pattern>%d{DEFAULT} %-5p [%-3T %-4.4X
{moca-session}] %c{1} [%X{moca-stack-level}] %m []%n</pattern>
                        </PatternLayout>
                        <Policies>
                            <TimeBasedTriggeringPolicy interval="24"
modulate="false"/>
```

```
                        </Policies>
                    </RollingFile>
                </Route>
            </Routes>
        </MocaRouting>
    </appenders>
</configuration>
```

This change makes it so that all MOCA server output as well as third-party logging is being saved to disk through Log4j. The rolling policy can be adjusted as needed by adjusting the appender. See the Log4j documentation for more information on the rolling policies.

How do I use asynchronous logging?

Starting with the 9.0.0.0 release, asynchronous logging for MOCA is enabled by default. It is still possible to use synchronous logging, although this should rarely be used. Asynchronous logging provides much better performance for the application and much higher logging throughput. To enable asynchronous logging, change any synchronous logger's configuration to be asynchronous. This change can be performed by using the `<asyncLogger>` elements for XML configuration instead of the old `<logger>` elements. If you want to make the root logger asynchronous as well, you should use the `<asyncRoot>` XML element instead of the old `<root>` element.

Example: The following example is an example of an XML configuration for an asynchronous logger.

Asynchronous Loggers

```
<asyncLogger name="com.redprairie.my.logger" level="ALL"
additivity="false">
    <appender-ref ref="AsyncAppender"/>
</asyncLogger>
```

Asynchronous logging can also be achieved by having a synchronous logger with an asynchronous appender that allows the appender to collect data in a buffer before flushing it to the file. An example of an asynchronous file appender is the `<RandomAccessFile>` appender. This approach offers similar performance to having an asynchronous logger with a synchronous appender. There is no performance improvement to have both an asynchronous logger and an asynchronous appender. Instead, there are now two queues for the messages to make it to their destination.

Example: The following is an example of an XML configuration with a synchronous logger and an asynchronous appender.

Random Access Logger

```
<?xml version="1.0" encoding= "UTF-8" ?>
<configuration packages="com.redprairie.moca.server.log"
monitorInterval="30">
    <appenders>
        <RandomAccessFile  name="AsyncAppender"
fileName="${env:LESDIR}/log/mylog.log">
            <PatternLayout>
                <pattern>%d{DEFAULT} %-5p [%-3T %-4.4X{moca-session}]
%c{1} [%X{moca-stack-level}] %m []%n</pattern>
            </PatternLayout>
        </RandomAccessFile>
    - </appenders>
    <loggers>
        <logger name="com.redprairie.moca.my.logger" level="ALL"
additivity="false">
            <appender-ref ref="AsyncAppender"/>
        </logger>
    </loggers>
</configuration>
```

When setting up configuration, care should be taken to use the correct case for the XML elements such as `<asyncLogger>`. Log4j 2 allows some flexibility with the case, but the merging of multiple XML configurations into one configuration by MOCA can be affected if there are different cases for the XML elements.

Note: In general, it is possible to configure all loggers to be asynchronous by setting the `Log4jContextSelector` system property to `org.apache.logging.log4j.core.async.AsyncLoggerContextSelector`. However, this approach is not recommended with MOCA since mixing the current approach with this property slows down all logging in the system.

How do I avoid issues with a stale runtime-logging.xml?

The **runtime-logging.xml** file is rebuilt every time that MOCA is run. Certain processes choose not to rebuild **runtime-logging.xml** when they run. In these cases, **runtime-logging.xml** is rebuilt if any of the component **logging.xml** files are older than **runtime-logging.xml**. This means that **runtime-logging.xml** should always be up to date. The only case when **runtime-logging.xml** is not up to date is when it is manually added or updated, its modified date is newer than any of the component **logging.xml** files, and then a **logging.xml** file is modified. To fix this issue, run anything that forces the rebuild of **runtime-logging.xml**, such as mocaserver.

Where can I learn more about the Log4j 2 architecture?

See the Log4j 2 architecture web page. To understand logging, you need to understand the Log4j 2 concepts, such as the logger hierarchy that allows fine tuning of the logging messages that are produced by the application. In most JDA SCE applications, the logger name is the full class name (package and class name) of the Java class from which logging messages are being executed. This naming convention supports a natural hierarchy of loggers across package namespaces. For example, in MOCA all database-related classes exist under the `com.redprairie.server.db` package. Therefore, specifying the package name as a logger, logging is included from all of the classes in the package because of the logger hierarchy and the package naming scheme used for loggers.

How do I debug a server-wide issue using logging and tracing?

See the "Trace Messages" (on page 53) section. The following description is a common use case:

When dealing with a server-wide issue, you may want to configure logging so that specific third-party application or JDA SCE specific messages are logged. To do this, use the`<loggers>` element inside of the `logging.xml` file. By adding loggers, you can fine tune the messages that are logged in your server-side trace and log. The following example further illustrates this configuration.

Example: I am running in a clustered environment and I am noticing that the first two nodes in my cluster start up fine but the third node does not join my cluster.

To help debug the issue with the clustered environment, you most likely want to enable logging for the third-party library that handles clustering: JGroups. To enable logging for JGroups, add a new logger to the `<loggers>` element in the logging.xml file.

Jgroups Logger

```
<logger name="org.jgroups" level="INFO"/>
```

JGroups uses a package-based naming scheme for its loggers where all loggers are named using a name under `org.jgroups` (for more information, see the Log4j 2 architecture web page). Therefore, the above logger logs any JGroups-related messages at an `INFO` level to the server trace file. This can be fine tuned by changing the level value. A logger level of`DEBUG` provides more detailed information than the `INFO` level. Changes to the `<loggers>` section should only be made if you want server-wide logging for the particular logger configuration. For session-specific logging, see the next section.

IMPORTANT: Enabling JGroups logging with a level of `DEBUG` creates very large amounts of trace and log messages. Consider this fact when leaving this logger enabled for long periods of time.

What is additivity?

If you define a logger for Infinispan and give it a file appender, the Infinispan log statements are included in both the file and the main output. This behavior is the result of a property in Log4j2 called additivity. Loggers with additivity enabled log to their parent loggers in addition to their own appenders. Additivity is enabled by default. This property can be disabled on any logger so that logging messages are only directed to the logger itself.

Example: The following code illustrates the XML configuration for disabling additivity on a logger.

Additivity Example

```
<logger name="org.infinispan" level="all" additivity="false">
    <appender-ref ref="InfinispanFileAppender"/>
</logger>
```

For more information about additivity, see the Log4j 2 documentation.

Does log4j support environment variable expansion?

XML-based file configuration supports environment variable expansion, but requires the environment variable reference to be of the form `${`*env:varname*`}`. Although environment variable references can be used for any value, it is most commonly used to define a value for the `file` parameter in `FileAppender` elements.

What is the MocaRouting appender?

The MocaRouting appender is an extension of the Log4j 2 standard routing appender. In contrast to the standard routing appender (which does not close file appenders until shutdown), the MocaRouting appender closes file appenders during runtime, and was created to support client tracing. The MocaRouting appender lets client tracing start a log file and close the handler during runtime. It also provides the route mechanism by which jobs and tasks handle logging.

How do I debug a session-specific issue involving a third-party library using logging and tracing?

By default MOCA ships with a `logging.xml` configuration file that enables `DEBUG` level tracing for all JDA SCE loggers when session tracing is enabled (for example, from the JDA SCE client). However, there may be cases that require obtaining logging messages from third-party libraries at the `INFO` or lower levels. Instead of changing the `<loggers>` section which dictates server-wide tracing, change the `<PackageFilter>` section. MOCA distributes the following `PackageFilter` configuration in the `logging.xml` file.

Package Filter Example

```
<PackageFilter includePackages="com.redprairie">
    <ThreadContextMapFilter onMatch="ACCEPT" onMismatch="NEUTRAL">
      <KeyValuePair key="moca-trace-level" value="*"/>
    </ThreadContextMapFilter>
</PackageFilter>
```

The above package filter performs the following tasks:

- Checks a thread local map to determine whether the **moca-trace-level** is *. This **moca-trace-level** indicates that tracing is enabled for the session thread.
- Checks that the logging message came from a certain set of packages specified by the `includePackages` comma-separated list. This check is similar to the Log4j 2 logger hierarchy, so if a log message comes from **com.redprairie.test** it is included in the log and trace, but messages from **org.some.third.party.logger** are not included.

Example: I am trying to load a form on the JDA SCE client but I am encountering Hibernate exceptions.

We know that the issues are most likely related to Hibernate (a third-party tool). Therefore, to start our investigation, we may want to enable logging for Hibernate to get more information about the cause of the issue. We only see this issue when working with this one specific form or request. This indicates that we want to have session-specific logging for Hibernate since server-wide logging for Hibernate might add too much unnecessary information to the log file.

To enable session-specific logging for hibernate, add the **org.hibernate** package to the `includePackages` setting in the `<PackageFilter>` configuration element of the `logging.xml`file. The following code illustrates the new `<PackageFilter>` configuration.

```
<!-- For session tracing capture debug level messages from RedPrairie
(com.redprairie) and Hibernate (org.hibernate) loggers -->
<PackageFilter includePackages="com.redprairie,org.hibernate">
    <ThreadContextMapFilter onMatch="ACCEPT" onMismatch="NEUTRAL">
      <KeyValuePair key="moca-trace-level" value="*"/>
    </ThreadContextMapFilter>
</PackageFilter>
```

After enabling this new configuration, we can wait 30 seconds, enable tracing on the client, and try to load our form or make our request again, which produces a trace that includes Hibernate messages.

In older versions of MOCA, there were more specific trace levels. How can I mimic this in the current version?

Older versions of MOCA that did not use Log4j 2 could enable specific trace levels for MOCA. These levels included: `FLOW`, `SQL`, `SERVER`, `MANAGER`, `PERF`, and `SARGS`. This let a user enable tracing for specific levels based on needs. For example, if a user wanted to see a trace of all the SQL statements being executed by the server, the user could enable the `SQL`trace level by running MOCA using the –`tS` argument. However, more recent versions of MOCA use Log4j 2, and can only enable and disable tracing by using the `–t` argument when running MOCA.

However, there are techniques for mimicking the same functionality for a server-wide trace even though the trace levels are no longer used. To do this, add loggers the same way you add loggers for the server-wide logging configuration using the `<loggers>` configuration element in the `logging.xml` file. The following table lists the loggers for each trace level that can be added to mimic the old behavior.

Trace Level	Logger Configuration
FLOW	<logger name="com.redprairie.moca.server.Flow" level="DEBUG"/>
SQL	<logger name="com.redprairie.moca.server.db" level="DEBUG"/>
SERVER	<logger name="com.redprairie.moca.server" level="DEBUG"/>
MANAGER	><logger name="com.redprairie.moca.server.Manager" level="DEBUG"/> <logger name="com.redprairie.moca.server.dispatch" level="DEBUG"
PERF	<logger name="com.redprairie.moca.server.Performance" level="DEBUG"/>
SARGS	<logger name="com.redprairie.moca.server.Argument" level="DEBUG"/>

Why does logger overriding not work?

The algorithm for merging XML files is sensitive to order of elements because of the way that both XML files are traversed concurrently. Make sure that you are following the general layout of the XML configuration in %MOCADIR%/data/logging.xml. If you are adding a logger and overriding another logger at the same time, make sure that the overriding logger is placed first in the list of loggers. The easiest way to determine if the configuration is correct is to run MOCA and inspect %LESDIR%/data/runtime-logging.xml (MOCA rebuilds the file from the respective logging.xml files on startup).

Chapter 6. MSQL

Description

This section describes the use of MSQL.

MSQL is an interactive command processor that you can use to execute arbitrary local syntax, including commands, SQL and Groovy scripts. MSQL runs in one of these two modes:

- **Client** – In client mode, MSQL connects to and initiates requests against a running MOCA server.
- **Server** – In server mode MSQL acts as a standalone server application.

Starting MSQL

Description

Prior to starting MSQL you need to bootstrap your environment. The steps to do this are different for each platform. On Linux and UNIX platforms logging in and calling `rpset <environment name>` sets up your environment. On Windows platforms this requires running the `env.bat` script.

The number of command line options for MSQL are limited and most developers start MSQL in either a client application or a standalone server application. This is the command line usage for MSQL.

```
moca-dev :>msql -h
Usage: msql [ -hv ] [ -a <url> | -S | -M ]
                    [ -u <user id> ] [ -w <password> ]
                    [ @<pathname> ]
-a <url>       URL to connect to in client mode
-u <user id>   Login user id for client mode
-w <password>  Login password for client mode
-S             Run in server mode
-M             Run in server mode (multi-threaded)
-h             Show help
-v             Show version information
@<pathname>    Path name of script file to process
```

Starting MSQL in Client Mode

In client mode MSQL establishes a connection to a running MOCA server using the URL provided in the `-a` command line argument. If a URL is not provided, but a `MOCA_REGISTRY` environment variable is set, the `server.url` registry key value is used. The URL can be one of these forms: `http[s]://<hostname>:<port>/service` or `hostname:port`. If the URL is of the form `hostname:port`, MSQL connects using the legacy protocol.

When connecting to a MOCA server, the user's credentials must first be authenticated before attempting to execute any commands or local syntax. If the `-u <username>` and `-w <password>` command line arguments are provided, they are used to authenticate the user; otherwise, MSQL prompts the user for the username and password.

This is an example for starting MSQL in client mode.

```
c:\ > msql -a http://localhost:4500/service
```

```
MSQL 2010.1.1 - Wed Apr 28 18:18:35 2010
```

```
Copyright (c) 2002-2010 RedPrairie Corporation.  All rights reserved.

Connecting to service: http://localhost:4500/service

Login: super
Password: ******

MSQL>
```

Starting MSQL in Server Mode

In server mode MSQL runs as its own server application and establishes a dedicated database connection so the MOCA server is not required to be running. MSQL can support multi-threaded Java commands if necessary through the `-M` command line option.

The `-S` command line option executes C and COM component libraries directly within the MSQL process. This prevents multi-threaded Java commands from calling C commands at the same time. In addition, if a C command causes a crash, MSQL itself will crash.

The `-M` command line option executes C and COM component libraries in a separate pool of native processes. This allows multi-threaded Java commands to call C commands simultaneously from each Java thread. Any C commands crashing in a native process return an error code to MSQL rather than causing it to crash.

This is an example for starting MSQL in server mode.

```
c:\ > msql -S

MSQL 2010.1.1 - Wed Apr 28 18:22:34 2010

Copyright (c) 2002-2010 RedPrairie Corporation.  All rights reserved.

Running in server mode...

MSQL>
```

MSQL Commands

MSQL supports a number of built-in commands that can be used from the MSQL command prompt. This table describes the built-in commands.

Command	Description
`Q[UIT]`	Closes MSQL.
`DESC <table name>`	Describes the database table name by calling the **describe table** command.
`ED[IT] <#>`	Opens an external editor editing the history element of the specified number. If no number is specified, then the last command is edited.
`L <#>`	Lists the specified command in the history indexed by its position. If a number is not provided, then the last command is listed.
`H[ISTORY] <#>`	Lists all the commands in the history up to the specified position. If a number is not specified, then the entire history is listed.
`/`	Executes the last command entered.

Command	Description
`/ <#>`	Executes the command in the history indexed by its position.
`@ <pathname>`	Executes commands from the given script file path name.
`! <cmd>`	Executes a shell command in the current working directory.

MSET Commands

In addition to the built-in commands, the behavior of MSQL can be modified using special built-in **MSET** commands. This table describes the **MSET** commands.

Command	Description
`MSET AUTOCOMMIT ON | OFF`	Enables/disables auto commit.
`MSET ENVIRONMENT <name>=<value>`	Sets an environment variable when running in client mode. Setting environment variables is not supported when running in server mode.
`MSET SPOOL <pathname>`	Sends all output to a spool file.
`MSET SPOOL OFF`	Stops any current spooling.

Entering and Executing Commands

Commands can be entered at the MSQL prompt after authenticating your credentials. MSQL recognizes two different types of commands: built-in MSQL commands and local syntax. MSQL commands are executed simply by pressing the Enter key, and local syntax commands can span multiple lines and are executed by entering "/" at an MSQL prompt.

This code is an example of entering and executing commands.

```
MSQL> desc comp_ver

Column         Nullable  Type      Length  Char_Length  Byte_Length
-------------  --------  --------  ------  -----------  -----------
BASE_PROG_ID   NOT NULL  NVARCHAR  256     256          512
COMP_MAJ_VER   NOT NULL  INT       (null)  (null)       (null)
COMP_MIN_VER   NOT NULL  INT       (null)  (null)       (null)
COMP_BLD_VER   NOT NULL  INT       (null)  (null)       (null)
COMP_REV_VER   NOT NULL  INT       (null)  (null)       (null)
COMP_FILE_NAM  NOT NULL  NVARCHAR  256     256          512
COMP_PROG_ID   NOT NULL  NVARCHAR  256     256          512
COMP_TYP       NOT NULL  NVARCHAR  1       1            2
COMP_FILE_EXT  NOT NULL  NVARCHAR  3       3            6
COMP_NEED_FW   NULL      INT       (null)  (null)       (null)
LIC_KEY        NULL      NVARCHAR  100     100          200
GRP_NAM        NOT NULL  NVARCHAR  40      40           80
```

```
MSQL> get os var where var = 'MOCA_REGISTRY'
   1  /

Executing... Success!

value
-------------------------------
c:\dev\trunk\moca\data\registry

(1 Rows Affected)

MSQL> quit
```

Enabling and Disabling Autocommit

By default all local syntax commands that execute successfully (return a status code of 0) result in a commit being issued automatically by MSQL. Local syntax commands that do not execute successfully (return any non-0 status code) result in a rollback being issued. This behavior can be disabled with the **mset autocommit off** command. When autocommit is disabled, you need to execute a **commit** or **rollback** command. In addition, the MSQL prompt is changed when autocommit is disabled to make it clear that MSQL is not automatically committing or rolling back transactions.

This code is an example of disabling autocommit.

```
MSQL> mset autocommit off

(autocommit off) MSQL> do loop where count = 5
(autocommit off)    1  |
(autocommit off)    2  [insert into mytable values (@i)]
(autocommit off)    3  /

Executing... Success!

(0 Rows Affected)

(autocommit off) MSQL> rollback
(autocommit off)    1  /

Executing... Success!

(0 Rows Affected)
```

Spooling Output

The input and output of an MSQL session can be captured by spooling it to a log file. The MSQL command `mset spool <pathname>` directs all input and output to the given path name. Spooling can be turned off again by issuing the **mset spool off** command.

This code is an example of spooling output to a log file.

```
MSQL> mset spool msql.log

MSQL> publish data where a = 1
   1  /

Executing... Success!

a
---
1

(1 Rows Affected)

MSQL> quit

c:> dir msql.log

 Volume in drive C is unlabeled      Serial number is 12ba:087e
 Directory of  C:\

 4/28/2010  21:41             142  msql.log
              142 bytes in 1 file and 0 dirs    4,096 bytes
allocated
   132,243,038,208 bytes free
```

Chapter 7. Local Syntax

Description

This section describes the concepts related to local syntax.

Local syntax is the MOCA scripting language that is used to execute commands on a MOCA server. With local syntax, you can:

- Execute commands with or without arguments.
- Execute SQL statements against a database.
- Execute Groovy scripts.
- Write local syntax that includes all of the above, including conditional logic and exception handling.

Command Execution

Local syntax that is not enclosed in brackets is executed as a command. A command is a noun and a verb phrase that defines a task to be performed. An example of a commonly used MOCA command is: publish data. You can further define a local syntax command by adding a where clause and arguments. On a MOCA server, command definitions are stored in the Command Repository. When a command is executed, the command line interpreter refers to the Command Repository to determine how and when to execute the command. For more information about the command repository, see the "Command Repository" (on page 99).

This code is an example of running the **publish data** command to which a where clause and two arguments were added.

```
MSQL> publish data where a = 1 and b = 2
   1  /

Executing... Success!

a    b
---  ---
1    2

(1 Rows Affected)
```

SQL Statement Execution

Local syntax that is enclosed in brackets is executed as an SQL statement against the database engine. The MOCA server supports multiple database engines, including Oracle and SQL Server. Because each database vendor supports a slightly different dialect of SQL, the server translates the given SQL statement into an SQL statement that is appropriate for the database engine being used. The implication of this is that the translation of some SQL constructs that are specific to one of the database engines may not be supported by the server. In general, the server supports Oracle's SQL dialect with some exceptions.

This code is an example of syntax that is enclosed in brackets and is executed as an SQL statement.

```
MSQL> [
   1   select base_prog_id, comp_maj_ver, comp_min_ver, comp_bld_ver,
comp_rev_ver
```

```
   2     from comp_ver
   3    where grp_nam = 'MOCA'
   4  ]
   5  /

Executing... Success!

base_prog_id            comp_maj_ver  comp_min_ver  comp_bld_ver
comp_rev_ver
----------------------  ------------  ------------  ------------  ---
---------
MocaClient.Command      2010          1             0             2
RedPrairie.MOCA.Client  2010          1             0             9

(2 Rows Affected)
```

Inline Groovy Script Execution

Local syntax that is enclosed in double-brackets is executed as a Groovy script. The MOCA server automatically imports the `com.redprairie.moca.*` and `com.redprarie.moca.util.*` classes so if you write Groovy scripts you do not need to import these from their Groovy scripts. While the server fully supports Groovy scripts of any size and complexity, you need to consider the advantages and disadvantages of writing a Groovy script versus spending the extra time to write an equivalent Java method. For simple functionality whose implementation is likely to remain unchanged, a Groovy script is preferred. For more complex functionality (or code that could be refactored), writing a Java method is preferred.

This code is an example of a Groovy script.

```
MSQL> [[
   1   value = moca.getRegistryValue('server.url');
   2  ]]
   3  /

Executing... Success!

value
-----------------------------
http://localhost:4500/service

(1 Rows Affected)
```

External Groovy Script Execution

Groovy scripts can also be included from the contents of a given filename by using a reference to the file containing the Groovy script. MOCA searches for filename under each of the directories defined by the `server.prod-dirs` registry key value and appending `/src/groovy` to each directory.

This code is an example of external groovy script execution.

```
MSQL> [[ @get-server-url.groovy ]]
   1  /

Executing... Success!

value
-----------------------------
http://localhost:4500/service

(1 Rows Affected)
```

Operators

MOCA supports a number of operators that can be used within local syntax. Operators can be applied to literal values and variables, and can be used in conditional tests. For more information, see "Variable Replacement" (on page 83).

Note: For a full list of operators supported within local syntax by the MOCA server, see the *MOCA Quick Reference Guide*.

This code is an example of the use of operators.

```
MSQL> publish data where sum = 4 + 5
   1  /

Executing... Success!

sum
---
9

(1 Rows Affected)

MSQL>  publish data where  full_name = 'John' || ' ' || 'Doe'
   1  /

Executing... Success!

full_name
---------
John Doe

(1 Rows Affected)
```

Functions

A rich set of built-in functions are also provided and can be used within any local syntax command. As with operators, functions can be applied to literal values and variables as well as nested within other functions.

Note: For a full list of functions supported within local syntax by the MOCA server, see the *MOCA Quick Reference Guide*.

This code is an example of functions that are supported.

```
MSQL> publish data where result = max(1, 10)
   1  /

Executing... Success!

result
------
10

(1 Rows Affected)

MSQL> publish data where length = length('Hello, world')
   1  /

Executing... Success!

length
------
12

(1 Rows Affected)

MSQL> publish data where today = sysdate()
   1  /

Executing... Success!

today
-------------------
2010-04-30 10:40:35

(1 Rows Affected)

MSQL> publish data where result = '[' || lower(rtrim('HELLO, WORLD
')) || ']'
   1  /

Executing... Success!

result
--------------
[hello, world]
```

```
(1 Rows Affected)
```

Command Streams

Description

Perhaps the most important aspect of any command to understand is that it is essentially a black box that accepts zero or more arguments and publishes a tabular result set. Regardless of the language in which a command is implemented, the MOCA server passes the implementation of that command and any arguments, and then the command publishes a result set. Even "simple" Java, C and COM commands that return only a status code are still publishing a result set, which happens to be empty.

The real power of MOCA becomes more apparent when commands are chained together, and the result set published from one command is used to provide the arguments to another command. Just as important, the arguments passed to one command can be made available to other downstream commands. The argument values and published data for a command are placed on the "stack" and become the "context" made available to other downstream commands. Commands can be chained together in different ways to determine what the context looks like for commands. For example, commands can be chained together with a "|" (command pipe) character to allow a downstream command to have full access to the context provided by a command upstream of it. In addition, a ";" (semicolon) character can be used to chain commands together providing a clean context to a downstream command.

Command Pipes

The command pipe (|) character provides the ability to execute a stream of commands where each downstream command has full access to the context provided by its upstream commands. Each command separated by pipes has access to published data and arguments from upstream commands through the replacement of variable references with each variable's value. Commands separated by pipes are sometimes called groups.

This code is an example of how the published data from the **do loop** command can be referenced from the **publish data** command downstream. The "@i" in the where clause of the **publish data** command is the simplest form of "variable replacement". In this example the "@i" variable reference is replaced with the value of the variable "i" from the context. MOCA supports a number of forms of variable replacement. For more information, see "Variable Replacement" (on page 83).

```
MSQL> do loop where count = 3
   1  |
   2  publish data where value = @i
   3  /

Executing... Success!

value
-----
0
1
2

(3 Rows Affected)
```

As shown in this example, there are three rows published from the command stream, because downstream commands are executed once for each row published by an upstream command. Because the **do loop** command published three rows, the **publish data** command was in turn called three times. Downstream commands are always called at least once as long as the upstream command was successful. So even if **do loop** command did not publish any rows, the **publish data** command still would have been executed once.

Command Separators

The semicolon (;) character provides the ability to execute a sequence of commands with a new context. Commands separated by semicolons are sometimes called streams.

Each command separated by semicolons executes independently of each other; that is, they do not have the context from upstream commands available for reference. Piping commands together can build a large context consisting of many stack levels that consumes memory. If a command can run in a clean context, then using a command separator rather than a pipe is both preferred and more efficient.

This code is an example of two **publish data** commands executing independently of each other. Unlike the previous example using pipes, the "@a" variable reference evaluates to `null` because the second **publish data** command executes in a clean context.

```
MSQL> publish data where a = 1;
   1  publish data where a = @a and b = 2
   2  /

Executing... Success!

a     b
---  ---
(nu  2

(1 Rows Affected)
```

Commands separated by semicolons can still have access to some shared upstream context.

This code is an example of a shared upstream context. The "@a" variable reference evaluates to 1 because both of the streams have access to the published "a" value. The "@b" variable reference evaluates to null because the second stream does not have access to the "b" value published by the first stream.

```
MSQL> publish data where a = 1
   1  |
   2  { publish data where b = @a; publish data where r = @b and s =
@a }
   3  /
Executing... Success!
r     s
---  ---
(nu  1
(1 Rows Affected)
```

Command Concatenation

The ampersand (&) character concatenates (links together) the result sets from a group of commands in a command stream.

This code is an example of where an ampersand concatenates commands.

```
MSQL> publish data where a = 1 and b = 2
   1  &
   2  publish data where a = 10 and b = 20
   3  /

Executing... Success!

a    b
---  ---
1    2
10   20

(2 Rows Affected)
```

Command Grouping

Braces ({}) can be used to group commands together. Any commands inside of braces get grouped together as one logical command, so command with braces can be put directly into a command stream. For example, if you want several commands to execute independently of each other, but they need to execute at the end of a piped stream of commands in the context that command stream provides, then you would use braces.

This code is an example of using braces (two of the previous examples are combined in this example). The second **publish data** command in the command grouping still has access to the value of the "@i" variable reference from the **do loop** command, but does not have access to the value of the "@a" variable reference.

```
MSQL> do loop where count = 3
   1  |
   2  {
   3      publish data where a = @i
   4      ;
   5      publish data where a = @a and b = @i
   6  }
   7  /

Executing... Success!

a      b
-----  -----
(null) 0
(null) 1
(null) 2

(3 Rows Affected)
```

Result Set Redirection

A result set can be "packed" into a variable using redirection. The contents of a packed (or sub) result set cannot be displayed using a tool like MSQL, but any commands that reference it do have access to its contents using the MOCA API. MSQL displays the contents of a sub result set by displaying the string "(res)".

This code is an example of redirection.

```
MSQL> do loop where count = 1000 >> subres
   1  /

Executing... Success!

subres
------
(res)

(1 Rows Affected)
```

Conditional Command Execution

Commands can be conditionally executed using an IF-ELSE statement.

Note: Although braces are not required in an IF-ELSE statement, their use is encouraged as best practice.

This code is an example of an IF-ELSE statement.

```
MSQL> publish data where a = 1
   1  |
   2  if (@a)
   3  {
   4      publish data where result = 'A is not null'
   5  }
   6  else
   7  {
   8      publish data where result = 'A is null'
   9  }
  10  /

Executing... Success!

result
-------------
A is not null

(1 Rows Affected)
```

Exception Handling

All commands that do not complete successfully raise exceptions. However, exceptions can be caught using either a try-catch statement or a catch clause. Exceptions that are caught allow execution to continue and cause the status code from the command to be changed to 0 so that the database transaction is committed. Exceptions that are not caught cause execution of the current command to stop immediately. An optional finally block can be added to a try-catch statement that is always executed, regardless of any success or error condition that occurs. Simple catch statements do not support a finally block. The results of a finally block are thrown away.

This code is an example that shows both a more complicated try-catch-finally statement, as well as a simple catch statement.

```
MSQL> try
   1  {
   2      [update mytable set col1 = 10 where col1 = 20]
   3  }
   4  catch (-1403)
   5  {
   6      publish data where result = 'Not found'
   7  }
   8  finally
   9  {
  10      publish data where result = 'Finally'
  11 }
  12 /

Executing... Success!

result
---------
Not found

(1 Rows Affected)

MSQL> [update mytable set col1 = 10 where col1 = 20] catch(-1403)
   1  |
   2  publish data where result = 'Continuing on despite an
exception'
   3  /

Executing... Success!

result
----------------------------------
Continuing on despite an exception

(1 Rows Affected)
```

Remote and Parallel Command Execution

Remote Command Execution

A command can be executed against remote MOCA servers using the `remote` keyword, which begins a distributed transaction across the local and remote systems. A two-phase commit is performed on both the local and remote systems if both the remote command and any local commands are successful. A rollback is performed on both the local and remote systems if either the remote command or any local commands fail. If any part of the two-phase commit fails the local transaction is rolled back.

Example: `remote(system) command`

The *system* is a string of this format: `http[s]://`*hostname*`:`*port*`/service#`*timeout*.

Note: Remote execution to 2009.2 and previous MOCA servers is supported using the *hostname*`:`*port*`#`*timeout* format.

Parallel Command Execution

Multiple commands can be executed against remote MOCA servers in parallel with the `parallel` keyword.

Examples:

```
parallel('system1,[system2,[...]]')command
```

```
parallel('http://moca1:4500/service,http://moca2:4500/service') ...
```

Each *system* is in the same format as that used for remote command execution. Rollbacks are performed on systems on which the command failed, and commits are performed on systems on which the command succeeded.

In Transaction Parallel Command Execution

Multiple commands can be executed against remote MOCA servers in parallel within the same transaction by using the `inparallel` keyword, which begins a distributed transaction across the local and all remote systems. A two-phase commit is performed on the local and all remote systems if all remote command and local commands are successful. A rollback is performed on the local and all remote systems if any remote commands or any local commands fail. If any part of the two-phase commit fails the local transaction is rolled back.

Examples:

```
inparallel('system1,[system2,[...]]')command
```

```
inparallel('http://moca1:4500/service,http://moca2:4500/service') ...
```

Each *system* is in the same format as that used for remote command execution.

Variable Replacement

Description

Many of the previous examples used a simple form of variable replacement to reference the value of a variable from the context for use in the current command. A variable reference of the form `@`*variable* in local syntax is replaced with the value of that variable from the context, but there are many more forms of variable replacement that you can use to write powerful local syntax to solve complex problems. Variable replacement is not limited to just simple local syntax. The MOCA server also supports variable replacement within SQL statements.

These variable replacement forms are supported within local syntax by the MOCA server and are described following in this guide:

- `@`*variable*
- `@+`*variable*
- `@*`

These forms are used in almost every command implemented in local syntax.

Note: For a complete list of variable replacement forms supported within local syntax by the MOCA server, see the *MOCA Quick Reference Guide*.

@variable Form

A `@`*variable* variable reference in local syntax is replaced by the value of *variable*. It is the most simplistic of the variable reference forms and is easy to understand.

This code is an example of where the **do loop** command publishes a single incrementing column named "i", which in turn is re-published by the **publish data** command.

```
MSQL> do loop where count = 3
   1  /

Executing... Success!

i
---
0
1
2

(3 Rows Affected)

MSQL> do loop where count = 3
   1  |
   2  publish data where value = @i
   3  /

Executing... Success!
```

```
value
-----
0
1
2

(3 Rows Affected)
```

@+variable Form

The `@+`*variable* variable reference is replaced with "*variable*=*value*". If the variable is not found in the context, then its reference is instead replaced with "`1=1`". If the variable's value was set with an operator other than "=", then that operator is used instead.

This code is an example of some original local syntax followed by what that local syntax becomes after the variable replacement is performed.

The local syntax:

```
publish data where lodnum = 'A19376262'
|
[
 select *
   from invlod
  where @+lodnum
]
```

Becomes:

```
publish data where lodnum = 'A19376262'
|
[
 select *
   from invlod
  where lodnum = 'A19376262'
]
```

@* Form

The `@*` variable reference is slightly more complicated and can cause confusion. References to `@*` are replaced by every variable in the where clause of the current command invocation, in order and in the same format as that used for `@+`*variable*.

Products commonly implement what is generically referred to as a "list" command, which is usually implemented by an SQL statement of varying complexity. An example of one of these list commands would be the Warehouse Management list work zones command, which is a simple SQL statement that selects all columns from the "zonmst" database table.

This code is an example of the implementation of the **list work zones** command.

This is the implementation of the **list work zones** command:

```
[
 select * from zonmst
```

```
  where @*
  order by wh_id, wrkzon
]
```

Executing this **list work zones** command:

```
list work zones
    where wrkare = 'AREA1'
      and wh_id = 'WM1'
```

Results in this SQL statement being executed by the MOCA server:

```
[
 select * from zonmst
  where wrkare = 'AREA1'
    and wh_id = 'WM1'
  order by wh_id, wrkzon
]
```

To further this example, if the standard **list work zones** command was updated so that it requires a "wrkzon" argument, you would need to change the where clause of the SQL statement to explicitly include the "wrkzon" argument using `@+`*variable* in addition to using the `@*` variable reference. This is because any reference of a variable that was provided in the where clause of the current command invocation is removed from the list of variables that `@*` "knows" about.

This code is an example of the modified `list work zones` implementation that shows the change to require a "wrkzon" argument along with the modified where clause in the SQL statement so that the "wrkare" still exists in the SQL statement's where clause:

```
if (@wrkare is not null)
{
    [
     select * from zonmst
      where @+wrkare
        and @*
      order by wh_id, wrkzon
    ]
}
else
{
    set return status
        where status = 714
          and message = 'A ''wrkare'' argument is required'
}
```

Directives

Directives are used to change the normal behavior of variable replacement and are specified by appending "#*directive_name*" to a variable reference. Directives are not used very often but are important to understand.

Note: For a complete list of supported directives, see the *MOCA Quick Reference Guide*.

This code is an example of a directive being used. The `#keep` directive is used the most often and gives you an alternative way to modify the example from the previous section that ensured that a "wrkare" argument was passed to the **list work zones** command. The `#keep` directive tells the MOCA server to keep the variable in the list of variables that are expanded if a `@*` variable reference is used. Knowing this, you can change the IF-ELSE statement in the previous example and no longer need to explicitly include the "wrkzon" argument using `@+`*variable* in the SQL statement's where clause.

```
if (@wrkare#keep is not null)
{
    [
     select * from zonmst
      where @*
      order by wh_id, wrkzon
    ]
}
else
{
    set return status
        where status = 714
          and message = 'A ''wrkare'' argument is required'
}
```

Type Casts

Type casts are used to coerce a variable reference to a specific type and are only valid in SQL statements. You specify type casts by appending ":*type_cast_name*" to a variable reference. If both a directive and type cast are used on the same variable, the directive must occur before the type cast.

Note: For a complete list of supported type casts, see the *MOCA Quick Reference Guide*.

Use of the date type cast is more complex. Variable references that include a date type cast are replaced with a value that is dependent on the context of the variable reference.

This code is an example of the date type cast.

References in the context of:

```
[
 select ... where column = @varname:date
]
```

Are replaced with:

```
[
 select ... where column = to_date(varname:date, 'YYYYMMDDHH24MISS')
]
```

And references in the context of:

```
[
 select ... where @+varname:date
]
```

Are replaced with:

```
[
 select ... where to_char(varname, 'YYYYMMDDHH24MISS') = 'value of
varname'
]
```

Database Table Qualifiers

Database table qualifiers are used to avoid ambiguity in multiple table joins when using an `@+`*variable* variable reference by prefacing *variable* with the table name.

This code is an example of a database table qualifier.

```
[
 select lodnum, subnum, wh_id, stoloc
   from invlod, invsub
  where @+invlod.ins_dt > sysdate - 7
]
```

SQL Hints

Description

SQL hints give you a method of taking control over some aspects of the MOCA server's SQL parser. All SQL hints should be used sparingly, and you should be aware of exactly what the SQL hints are doing and the consequences when deciding to use a hint over the default behavior.

Note: For a complete list of supported hints, see the *MOCA Quick Reference Guide*.

SQL Conversion

By default, all SQL statements are converted to syntax that is appropriate for the native database engine. In some cases, you may want to make use of functionality provided by the native database engine that the MOCA server does not intrinsically support.

This code is an example of the noconv hint that provides a hint to the server to not convert the given SQL statement.

```
[
 /*noconv*/
 select ...
  from ...
 where ...
]
```

Auto-Binding

The MOCA server automatically replaces literals within SQL statements with bind variables where possible to increase the likelihood of the database engine finding a pre-parsed version of a given SQL statement in its cache. There are instances, however, where passing a literal to the database engine is actually beneficial to passing a bind variable.

This code is an example of the nobind hint that gives you a way to toggle the server's default auto bind behavior on and off.

```
[
 select *
   from invlod, invsub
  where @+invlod.ins_dt > sysdate - 7
    and invlod.lodnum = invsub.lodnum
    and invlod.prmflg = /*#nobind*/ 1 /*#bind*/
    and invlod.ins_dt > sysdate - 7
]
```

SQL profiling

The #profile hint labels SQL statements so that they can be distinguished from other SQL statements in the MOCA server command profiler. The MOCA server command profiler provides performance statistics for all executed MOCA commands, and JDA supply chain execution application issued and manually issued SQL statements. SQL profiling provides more flexibility in tracking SQL statements (for example, separating out more complex SQL statements for benchmarking or ignoring simple pass-through SQL statements). In the MOCA server command profiler, SQL statements are categorized by command path and then by profile.

Example: The following code is an example of applying the SQL profile hint, MOCA Retrieve Tasks, to the SQL command, `select * from task_definitions`.

```
[
/*#profile = MOCA Retrieve Tasks */
select * from task_definitions
]
```

Limit (paging and totaling)

#limit description

The #limit hint defines the subset of rows that you want the MOCA SQL engine to return and (optionally) the total number of rows without the limit applied. The #limit hint provides simpler client side paging so that the client application can display a certain number of rows starting with a specified row number. MOCA handles this by wrapping the specified SQL statement with row limit criteria and modifying the query so that at the database level the query only returns the requested number of rows (row limit). The optional total number of rows lets the client application calculate (and typically display) the appropriate number of pages. The algorithm that is used for paging can be configured with the #limitalg hint (see below).

The #limit hint is formatted as follows:

```
[
/*#limit=<Start_Row>,<Row_Limit>,<Calculate_Total> */
<Query>
]
```

Where:

- **<Start_Row>**: Number of the row that starts the subset of rows, where the first row is 0, the second row is 1, and so on. <Start_Row> can be an integer or integer-type variable. The maximum value of <Start_Row> is limited to Java's maximum integer size which is 2^31-1 (approximately 2.1 billion).

- **<Row_Limit>**: Total number of rows to return in the subset. <Row_Limit> can be an integer or integer-type variable. Setting <Row_Limit> to 0 indicates that there is no limit. For more information, see "Dynamically turning #limit on and off" (on page 93).
- **<Calculate_Total>**: (Optional.) Indicates whether to include the value of the total number of rows returned by <Query> without the limit applied. <Calculate_Total> can be a Boolean (`true` or `false`) value or Boolean-type variable. When <Calculate_Total> is `true`, MOCA modifies <Query> to also calculate the total number of returned rows without the limit applied. The total is not included as a column in the result set but can be extracted from the result set using a MOCA Java API (`MocaUtils.getPagedTotalRowCount`) or by casting the results to the MOCA `PagedResults` interface and using `PagedResults.getTotalRowCount()`. For more information, see "PagedResults interface" (on page 92).

  IMPORTANT: Enabling <Calculate_Total> adds a count of all possible rows based on the where clause, which could result in a potential performance loss. You should only enable <Calculate_Total> when you must know the total number of rows in addition to limiting the rows to a certain subset. A typical use case for this scenario is a client control that must determine the number of total possible rows that exist so that a corresponding paging control is correctly generated.

- **<Query>**: SQL statement used to query the database for the rows of information. <Query> must conform to the following restrictions and limitations:
  - **Top-level queries only**: The #limit hint only applies to the top-level query and should be entered at the start of the query. The #limit hint cannot be used in sub-queries or to limit individual union statements.
  - **No `rownum`**: <Query> cannot use `rownum`.
  - **No common table expressions**: <Query> cannot use common table expressions since MOCA internally uses them to achieve paging for the query and some database engines do not support recursive common table expressions. For more information, see the Wikipedia website and search for common table expressions.
  - **No duplicate columns**: Since <Query> cannot return duplicate columns, avoid the use of `select *` when joining multiple tables.
  - **Special format for unions with order by**: If <Query> uses union with an order by clause, it should be written in the following format:

```
[
/*#limit=@startRow,@rowLimit,@calculateTotal */
select *
  from (
         <query1>
         UNION
         <query2>
       ) results
  order by ...
]
```

Example: The following code is an example of a page that starts with the first row returned by the query and includes 25 rows.

```
/* Page 1 */
[
/*#limit=0,25 */
select * from task_definition
]
```

Example: The following code is an example of a page that starts with the 26th row returned by the query and includes 25 rows.

```
/* Page 2 */
[
/*#limit=25,25 */
select * from task_definition
]
```

Example: The following code is an example of a page that starts with the 26th row returned by the query, includes 25 rows (row limit), and includes the total number of rows without the limit applied.

```
/* Page 2 - with total */
[
/*#limit=25,25,true */
select * from task_definition
]
```

#limitalg description

The #limitalg hint can be used to choose the algorithm that is used for a paged query. This hint will only be respected if the total row count is requested and if the #limit hint is detected.

The #limitalg hint is formatted as follows:

```
[

/*#limit=<Start_Row>,<Row_Limit>,<Calculate_Total>*/ /*#limitalg=<Algorithm>*/

<Query>

]
```

The following table describes the available algorithms.

Algorithm	Details	Example
classic	Affects Oracle only. This is the classic MOCA approach that uses a select for the count of the total rows and a sub-query select for the page. Performs best on simple queries, but works well on large data sets.	`SELECT moca_subquery__.*,` `(SELECT Count(*)` `FROM (SELECT job_definition_exec.*` `FROM job_definition_exec)) moca_total_rows__` `FROM (SELECT job_definition_exec.*` `FROM job_definition_exec)`

Algorithm	Details	Example
		`moca_subquery__` `WHERE rownum <= 25`
cte	Affects Oracle only. Uses a common table expression for the main query. Then gets the total row count from the main query and another select for the page data based on the main query view. This approach may perform poorly on some expensive queries as well as on queries with large data sets.	`WITH moca_mainquery__` `AS (SELECT job_definition_exec.*` `FROM job_definition_exec)` `SELECT moca_subquery__.*,` `(SELECT Count(*)` `FROM moca_mainquery__) moca_total_rows__` `FROM (SELECT row_.*,` `ROWNUM rownum_` `FROM moca_mainquery__ row_` `WHERE ROWNUM <= 25) moca_subquery__`
over	Affects Oracle only. Uses an analytical function to get the total row count over the whole data set. Works better on some expensive queries on and on large data sets than the cte algorithm. This algorithm has a limitation that queries must qualify the star (*) in the select statement with the database table. For example:- `select job_definition_exec.* from job_definition_exec`	`SELECT *` `FROM (SELECT Count(*)` `over () moca_total_rows__,` `job_definition_exec.*` `FROM job_definition_exec)` `WHERE ROWNUM <= 25`

Example: The following code is an example of a page that starts with the first row returned by the query and includes 25 rows.

```
/* Page 2 - with total. Use the "classic" algorithm.*/

[

/*#limit=25,25,true */ /*#limitalg=classic*/

select * from task_definition

]
```

PagedResults interface

When the #limit hint is used, the underlying Java interface that is returned is `PagedResults` (com.redprairie.moca.PagedResults). The `PagedResults` interface is an extension of the `EditableResults` interface and includes additional methods to access the values of <Start_Row>, <Row_Limit>, and <Calculate_Total>. For more information, see "#limit description" (on page 88).

The following code specifies the `PagedResults` interface:

```
/**
 * Results that are returned from a query that uses the
 * #limit hint to specify that row limiting and pagination should be used.
 * This result set optionally includes an additional total
 * row count that is the number of rows that would be returned
 * had row limiting NOT been used. For example, for a query that
 * normally returns 100 rows but is limited to rows 25-50, {@link #getRowCount()}
 * returns 25 but {@link #getTotalRowCount()} returns 100 which is the number
 * of rows had the limit not been applied. Additionally, the PagedResults interface
stores
 * information about the start row and row limit that was specified when the
 * result set was generated.
 *
 */
public interface PagedResults extends EditableResults {

    /**
     * Gets the start row (zero-based index) that was specified
     * to generate the paged results. The returned value is 0 if a start
     * row was not specified.
     * @return The start row
     */
    public int getStartRow();

    /**
     * Gets the row limit that was specified when generating this paged
     * results. 0 indicates no limit.
     * @return The row limit
     */
    public int getRowLimit();

    /**
     * Gets the total number of possible rows had row limiting
     * not been used. The total number of rows is optionally calculated
     * depending on whether it was specified as one of the #limit hint's arguments.
     * If the total rows were not calculated, the returned value is -1.
     * @return The total number of rows or -1 if total row calculating
     *         was not enabled.
     */
    public int getTotalRowCount();

    /**
     * Indicates whether the total row count was calculated for this
     * result set when using the #limit hint. If this value is true, then
     * {@link #getTotalRowCount()} returns the total number
     * of possible rows (without the limit applied).
     * @return Whether total row calculating was enabled or not.
     */
    public boolean isTotalRowCountCalculated();

}
```

Example: The following code is an example of using the #limit hint and accessing the original #limit values using the `PagedResults` interface (executed in Groovy).

```
[
/*#limit=0,10,true */
select * from task_definition
] >> res
|
[[
    // Cast the results to the PagedResults interface. This code is safe, provided that
the limit hint is also used.
    PagedResults pagedRes = (PagedResults) res
    actualRowCount = pagedRes.getRowCount() // The subset of rows that are actually
returned.
    totalPossibleRowCount = pagedRes.getTotalRowCount() // The total number of rows that
could be returned if the results were not limited.
    startRow = pagedRes.getStartRow() // Returns 0 which is the value of the start row
specified in the limit hint.
    rowLimit = pagedRes.getRowLimit() // Returns 10 which is the value of the row limit
specified in the limit hint.
]]
```

Example: The following code is an example of using the #limit hint without calculating the total number of rows. In this case, the `PagedResults.getTotalRowCount()` returns -1 and `PagedResults.isTotalRowCountCalculated()` is false.

```
[
/*#limit=0,10,false */
select * from task_definition
] >> res
|
[[
    // Cast the results to the PagedResults interface. This code is safe, provided that
the limit hint is also used.
    PagedResults pagedRes = (PagedResults) res
    isTotalRowCountCalculated = pagedRes.isTotalRowCountCalculated() // Returns false
because the total row count was not calculated by the #limit hint.
    totalRowCount = pagedRes.getTotalRowCount() // Returns -1 because the total row count
was not calculated.
    actualRowCount = pagedRes.getRowCount() // The subset of rows that are actually
returned.
    totalPossibleRowCount = pagedRes.getTotalRowCount() // The total number of rows that
could be returned if the results were not limited.
]]
```

Dynamically turning #limit on and off

As indicated in Limit paging and totaling, the #limit hint also supports variables so that the rows being limited can be dynamic. This feature provides the capability to turn the #limit hint off by setting both <Start_Row> and <Row_Limit> to 0. MOCA interprets a <Row_Limit> of 0 as meaning no limit. By also setting <Start_Row> to 0, MOCA returns all possible rows.

Example: The following code is an example of using variables to turn the #limit hint off.

```
publish data
    where rowLimit = nvl(@rowLimit, 0)
      and startRow = nvl(@startRow, 0)
      and calculateTotal = nvl(@calculateTotal, false)
|
[
/*#limit=@startRow,@rowLimit,@calculateTotal */
select * from task_definition
]
```

Dynamically turning the #limit hint on and off is useful in commands. You can code your commands so that they take values that are then passed to the #limit hint. If the variables are not on the stack, then the #limit hint becomes `/*#limit=0,0,false */` and is turned off.

Example: You added the code from the previous example to the MOCA command `list tasks` and encountered the following situations:

- **The #limit variables are not on the stack**: The #limit hint becomes `/*#limit=0,0,false */` and is turned off. The query is not limited and runs as normal by selecting all of the rows in the `task_definition` table.
- **The #limit variables are on the stack**: The #limit hint is turned on and the query returns only the appropriate subset of rows from the `task_definition` table. For example, the following code returns 50 rows from the `task_definition` table starting with the 11th row:

```
list tasks where startRow = 10 and rowLimit = 50 and calculateTotal =
true
```

NotFoundException and total number of rows

Requesting a starting row number that is not available (for example, requesting starting row number 501 from a result set that only contains 100 rows), results in a NotFoundException (-1403) and, if requested, the total number of rows.

Example: The following code is an example of requesting a starting row number that is not available. There are only 100 rows in the task_definition table.

```
[
/* Catching the -1403 exception in this example so we can access the
results to see the total possible row count */
/*#limit=500,10,true */
select * from task_definition ]
] catch (-1403) >> res
|
[[
    // Cast the results to the PagedResults interface. This code is
safe, provided that the limit hint is also used.
    PagedResults pagedRes = (PagedResults) res
    actualRowCount = pagedRes.getRowCount() // Returns 0 because the
501st row was requested but does not exist.
    totalPossibleRowCount = pagedRes.getTotalRowCount() // Returns
100 because it is the total number of possible rows that could be
returned by the query.
]]
```

Forced Join Order

The "+ ordered_all" hint forces a specific join order on the query or sub-query. Specifically, the join of tables will be performed in the exact order as they appear in the query. Without this hint, the database if free to pick a join order for the query. This hint can be used to force a join order on all supported databases, or one of the database-specific variants can be used to force a join order only on a particular database. Since SQLServer doesn't support the join order hint the same way that Oracle does, MOCA automatically translates the Oracle-style hint into a "FORCE ORDER" option when running against SQLServer.

Example: The following code is an example of applying the SQL forced join order hint to a SQL query that performs a join of multiple tables.

```
SELECT /*+ ordered_all */ foo.*,
                                  inside.things,
                                  inside.stuffs
FROM foo
       JOIN (SELECT /*+ ordered_all */ bar.things,
                                              wer.stuffs,
                                              tor.branch
       FROM bar
              JOIN wer
              ON foo.wert = wer.wert
              JOIN tor
              ON wer.tert = tor.tert) inside
       ON foo.foot = inside.foot
WHERE inside.werk = 'test'
```

Note that the join order can be specified for the top-level query as well as any sub-queries. It's also possible to perform ordered joins only a specific type of database by using a more specific hint.

The following table describe the forced join order hints.

Hint	Description
+ ordered_all	Forces ordered joins on all supported databases.
+ ordered	Forces ordered joins **only** on Oracle. Has no effect when running against another database.
+ ordered_mssql	Forces ordered joins **only** on Microsoft SQLServer. Has no effect when running against another database.

Command Hints

Command hints enable you to control certain aspects of the MOCA command execution engine similar to SQL hints. The following code illustrates the generic command hint syntax:

```
/*#<hint_key>=<optional_hint_value> */
command here
```

Note: Because mbuild validates command hint syntax, if you misspell a command hint or use the number symbol (#) immediately after the `/*` in a code comment (for example, `/*###This is a code comment.*/`) mbuild issues a warning. You can eliminate the warning for code comments by adding a space after the asterisk and before the number symbol (for example, `/* ###This is a code comment.*/`) or eliminating the use of the number symbol (for example, `/* This is a code comment.*/`).

MOCA provides a predefined command hint that is described in the following section.

Row Accumulation Limit

In 2014.1, MOCA added the concept of a row accumulation limit to prevent the accumulation of a dangerously large number of rows from piped local syntax commands. For example, the following code returns 10 million rows:

```
do loop where count = 10000000
|
publish data where arg1 = 'someValue'
```

Typically, you do not want to run code that returns 10 million rows because of the memory usage associated with accumulating that many rows or the potential of encountering an out of memory error that terminates your MOCA instance. However, even if you run such code, MOCA enforces a row accumulation limit. When the accumulated number of rows reaches the limit, an exception occurs. By default, MOCA is distributed with the row accumulation limit set to 1 million which is configurable by changing the MOCA registry key **server.row-accumulation-limit**.

If you need to run code that accumulates a large number of rows that might exceed the row accumulation limit, you should use the paging functionality (limit SQL hint). In the rare case that you need to accumulate a number of rows that is greater than the row accumulation limit without using paging, you can disable the row accumulation limit by adding the command hint **nolimit-accumulation** to your command. The following code illustrates the addition of the **nolimit-accumulation** command hint:

```
/*#nolimit-accumulation */
do loop where count = 10000000
|
publish data where arg1 = 'someValue'
```

Place this hint at the top of a local syntax command or on the first line of an adhoc request. Adding this hint disables the row accumulation limit for the extent of the associated command stream.

Reserved Words

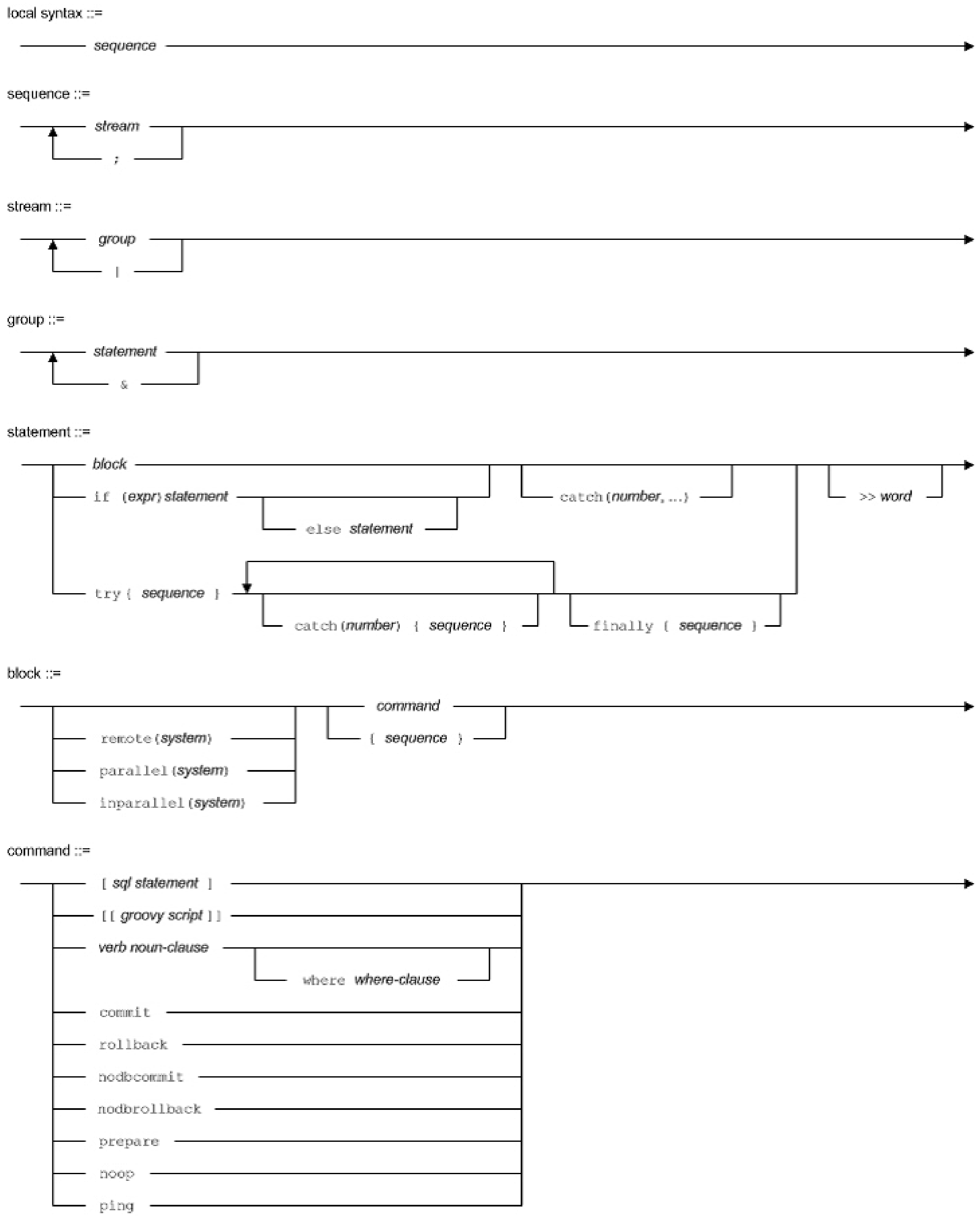
The following are reserved words in local syntax and should not be used as a command or argument name:

- `and`
- `if`
- `not`
- `remote`
- `catch`
- `inparallel`

- `null`
- `try`
- `else`
- `is`
- `or`
- `where`
- `finally`
- `like`
- `parallel`

Local Syntax Grammar

The following image is the definition of MOCA's local syntax grammar.



Chapter 8. Command Repository

Introduction

Description

This section describes the command repository. The command repository is a directory structure that consists of files that contain the definitions for component levels, commands and triggers. Together these definitions are used to determine what, how and when to execute different commands. Component level, command and trigger definition files are XML-based and their content is defined in this section.

Directory structure

The root folders of the command repository directory structure are defined in the `server.prod-dirs` MOCA registry key. Each root folder contains a subfolder named **src**, which contains a subfolder named **cmdsrc**. Each **cmdsrc** subfolder contains component level definition files, and subfolders for each component level that contain the command and trigger definition files:

```
<PROD_DIR>
|___src
    |___cmdsrc
        |____<Component_Level>.mlvl
        |____<Component_Level>
             |____<Command>.mcmd
             |____<Trigger>.mtrg
```

IMPORTANT: The **src** and **cmdsrc** folder names are hard-coded in `mbuild`.

Component Level Definition Files

Description

Component level definition files are used to specify the component libraries that implement commands. A component library is a JAR file, DLL or shared library that implements methods and functions that match the definition of commands specified in command definition files. The MOCA server loads these libraries at run time and calls the methods and functions when a command is executed. A command defined in a component library at a higher level takes precedence over the same command in a component library at a lower level. Each directory containing component level definition files should be in the list of path names defined by the `server.prod-dirs` registry key.

All component level definition files have these rules:

- The filename must have an .mlvl suffix (for example, `MOCAbase.mlvl`)
- Only one component level definition is allowed per file

Local Syntax Libraries

Although it is not common, component libraries can consist of nothing more than a set of local syntax commands. The implementation of a local syntax command is contained directly within that command's command definition file, so a JAR file, DLL or shared library is not technically required to support component libraries consisting of only local syntax commands. However, you are required to provide an implementation of versioning methods or functions if the component library is part of the standard product suite. Following that rule, local syntax component libraries are nothing more than a component library whose JAR file, DLL or shared library contains only the necessary methods or functions to support the required versioning of the component library. For more information, see "Initializing and Versioning Component Libraries" (on page 109).

Java Component Libraries

Java component libraries require a package element to be specified in the component level definition file. The package defines a Java package that contains one or more classes that implement the commands specified in the component library's command definition files. The JAR file that contains the package must exist somewhere in the list of path names defined by the `CLASSPATH` environment variable. In addition, the MOCA server looks for a `ComponentLibrary` class in the package that provides methods for initializing the component library, and providing version and product information. For more information, see "Initializing and Versioning Component Libraries" (on page 109).

This code is an example of a component level definition file for a component library named "JavaSample" whose commands are implemented in the Java package `com.redprairie.les.javasample`. This component library has a sort sequence of 10000, which defines its precedence level relative to other component libraries.

```
<component-level>
 <name>JavaSample</name>
 <description>Sample Java Methods</description>
 <sort-sequence>10000</sort-sequence>
 <package>com.redprairie.les.javasample</package>
</component-level>
```

C Component Libraries

C component libraries can contain C and simple C functions, and can be recognized by component level definition files that specify a library element. C functions return a pointer to a `RETURN_STRUCT` containing a status code, message and result set, and simple C functions return a `long` representing a status code. A status code other than `eOK` (0) is considered an error. The DLL or shared library that contains the C functions must exist in the list of path names defined by the `PATH` environment variable on Windows or the corresponding shared library search path on Linux and UNIX systems. C component libraries also support initialization and versioning by implementing a set of C functions within the DLL or shared library. For more information, see "Initializing and Versioning Component Libraries" (on page 109).

This code is an example of a component level definition file for a component library named "MOCAbase" whose commands are implemented in the C DLL `MOCAbase.dll` or shared library `MOCAbase.so`.

```
<component-level>
 <name>MOCAbase</name>
 <library>MOCAbase</library>
 <description>MOCA Base Component Library</description>
 <sort-sequence>0</sort-sequence>
</component-level>
```

COM Component Libraries

COM component libraries contain COM methods, and their component level definition file contains a program-ID element. COM component libraries must implement their classes within the program ID. As with Java, C and COM component libraries should also implement a set of methods to initialize and version the component library.

This code is an example of a component level definition file for a component library named COMSample whose commands are implemented in the COM DLL `COMSample.dll`.

```
<component-level>
 <name>COMSample</name>
 <description>Sample COM Methods</description>
 <sort-sequence>10000</sort-sequence>
 <program-id>RedPrairie.COMSample</program-id>
</component-level>
```

Mixing Languages in Component Libraries

It is possible to define component levels that house commands implemented in more than one language. For example, you can define a component level consisting of commands written in Java and C by specifying both a "package" and "library" element in the same component level definition file. The MOCA server references JAR files found in the class path for the commands implemented in Java and the appropriate DLL or shared library for commands implemented in C. Component levels cannot contain commands implemented in C and COM all at once because Windows does not support a single DLL containing code from those technologies.

When multiple languages are mixed within the same component level the initialization and versioning methods or functions only need to be provided by one of the languages.

This code is an example of a Warehouse Management component level definition file that contains commands that are implemented in C using the `DCSint` DLL or shared library, and in Java using the `com.redprairie.wmd.intrinsic` package. In addition, a number of commands are implemented in local syntax.

```
<component-level>
<name>DCSint</name>
<description>WMD Intrinsic Server Functions</description>
<package>com.redprairie.wmd.intrinsic</package>
<library>DCSint</library>
<sort-sequence>700</sort-sequence>
</component-level>
```

XML Element Descriptions

Component level definition files are XML-formatted. This table describes each element that can exist in a component level definition.

Element	Description
`name`	Name of the component level. A component level name is required.
`description`	Description of the component level. A component level description is optional.
`library`	DLL or shared library name, minus the filename extension. If a library is not provided, the component library name is used. **Note**: If "library" is not specified, and the component level specifies a Java package or COM program ID, no DLL is loaded, and any defined C components are not executed.

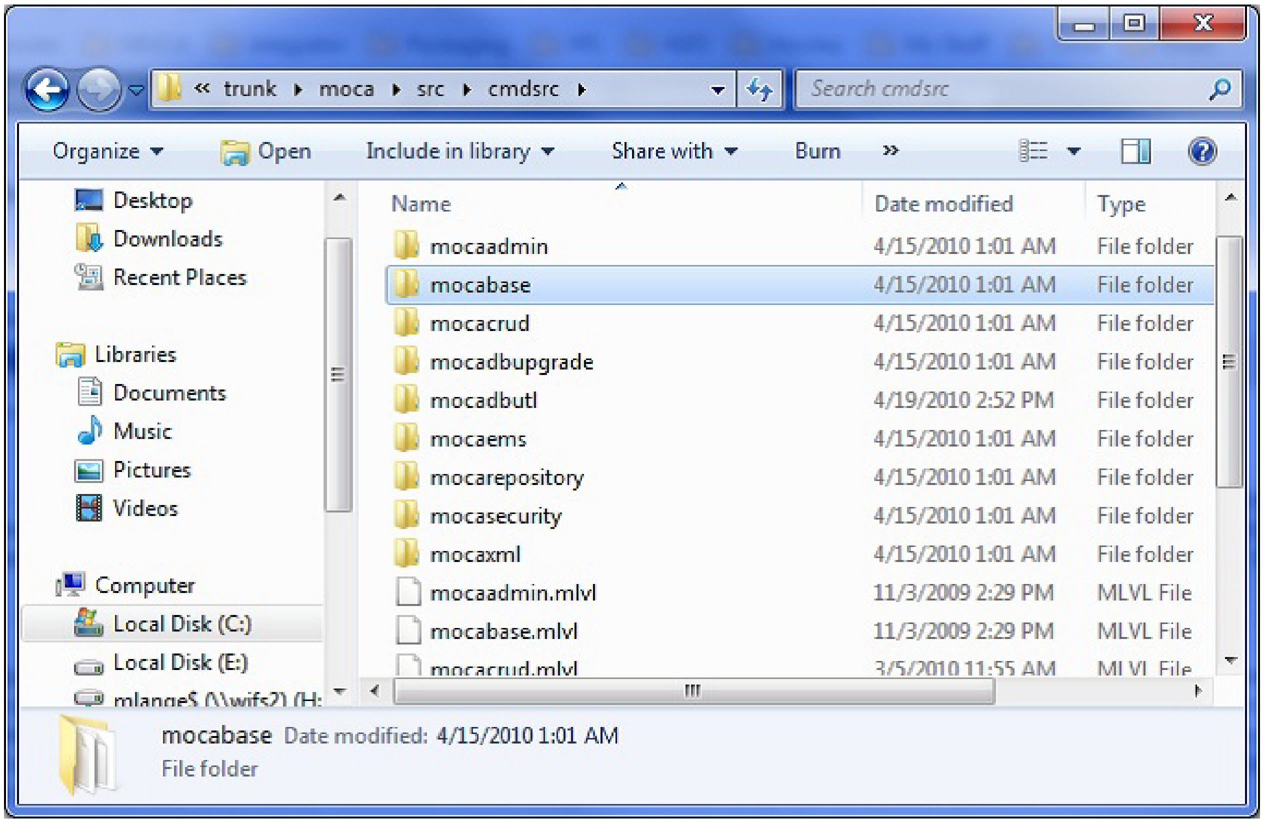
Element	Description
`directory`	Sub-directory name in which this component level's command and trigger definition files reside. If a directory is not provided, the filename of the component level definition file minus the .mlvl extension is used.
`sort-sequence`	Sort sequence of this component level. Component levels with higher sort sequences take precedence over component levels with lower sort sequence numbers. A sort sequence is required.
`program-id`	Program ID where this component library's COM methods reside. This is required for component libraries that implement COM methods. It should not be provided for any other type of component library.
`package`	Package where this component library's Java component classes reside. This is required for component libraries that implement Java methods. It should not be provided for any other type of component library.
`editable`	Indicates whether the component libraries' commands or triggers can be changed using Server Command Maintenance. These are the valid values: yes and no (default).
`version`	Version of the component level definition file. This is optional and can be used as a container for a revision control system ID or version.

Command Definition Files

Description

Command definition files are used to define the different commands that make up a component library. Support exists for these different types of commands: local syntax, Java methods, C functions, simple C functions and COM methods.

All command definition files should reside in a directory at the same level as the component level definition file and are named with the name of the component level definition file minus the ".mlvl" extension.



Name	Date modified	Type
mocaadmin	4/15/2010 1:01 AM	File folder
mocabase	4/15/2010 1:01 AM	File folder
mocacrud	4/15/2010 1:01 AM	File folder
mocadbupgrade	4/15/2010 1:01 AM	File folder
mocadbutl	4/19/2010 2:52 PM	File folder
mocaems	4/15/2010 1:01 AM	File folder
mocarepository	4/15/2010 1:01 AM	File folder
mocasecurity	4/15/2010 1:01 AM	File folder
mocaxml	4/15/2010 1:01 AM	File folder
mocaadmin.mlvl	11/3/2009 2:29 PM	MLVL File
mocabase.mlvl	11/3/2009 2:29 PM	MLVL File
mocacrud.mlvl	3/5/2010 11:55 AM	MLVL File

All command definitions files have these rules:

- The filename must have a .mcmd suffix (for example, `pick_inventory.mcmd`)
- Only one command definition is allowed per file

Local Syntax Commands

Local syntax commands are distinguished by a "type" element with a value of "Local Syntax". Local syntax commands can contain references to other commands, execute SQL or execute a Groovy script.

This code is an example command implemented in local syntax that executes SQL. For more information on how to use and develop local syntax commands, see "Local Syntax" (on page 73).

```
<command>
 <name>list appointment limits</name>
 <description>This command is used to list entries in the appointment
limits table.</description>
 <type>Local Syntax</type>
 <local-syntax>
  [
   select *
     from appt_limit
    where @+start_tim:date
      and @+end_tim:date
      and @*
    order by arecod, start_daycod, start_tim
  ]
 </local-syntax>
</command>
```

Java Methods

Java methods are distinguished by a "type" element with a value of "Java Method". They are also the only command type that must include a "class" element in addition to a "method" element. One or more "argument" elements can also be defined and are used to determine the signature of the Java method to call.

This code is an example command named `calculate ucc128 check digit`. This command is implemented in Java by method `calculateCheckDigit` within the class `Ucc128Utils` and takes a single argument (which is a string).

```
<command>
 <name>calculate ucc128 check digit</name>
 <description>Calculate a ucc-128 check digit.</description>
 <type>Java Method</type>
 <class>Ucc128Utils</class>
 <method>calculateCheckDigit</function>
 <argument name="tag" alias="tag_id" datatype ="string">
  UCC-128 tag
 </argument>
</command>
```

C and Simple C Functions

C and simple C functions can be recognized by a "type" element with a value of "C Function" or "Simple C Function" in their command definition files. As with Java commands, one or more "argument" elements can be defined and are used to determine the prototype of the C function to call.

This code is an example of a C function that implements the list library versions command. The C function `mocaListLibraryVersions` is called with a single string argument when the command is executed.

```
<command>
 <name>list library versions</name>
 <description>List component library versions.</description>
 <type>C Function</type>
 <function>mocaListLibraryVersions</function>
 <argument name="category" datatype ="string">
  Component library category.
 </argument>
</command>
```

COM Methods

COM methods are distinguished by a "type" element with a value of "COM Method". One or more arguments can be defined using the same "argument" element as with other command definitions and are used to determine the signature of the COM method to call.

This code is an example of a command implemented in COM named `scan serial number`. This command is implemented by the COM method `usrScanSerialNumber` and takes one string argument.

```
<command>
 <name>scan serial number</name>
 <description>Scan a product's serial number.</description>
 <type>COM Method</type>
 <method>usrScanSerialNumber</function>
 <argument name="usrid" alias="user_id" datatype ="string">
  User id.
 </argument>
</command>
```

XML Element Descriptions

This table describes each element that can exist in a command definition.

Element	Description
`name`	Name of the command. A command name is required
`description`	Description of the command. A command description is optional.
`type`	Type of the command. These are the valid values: Local Syntax, C Function, Simple C Function, COM Method and Java Method.
`local-syntax`	Local syntax that the command executes. Local syntax is required for commands implemented by local syntax but should not be provided for any other type of command.
`function`	Name of the C function that implements the command. A function name is required for commands implemented by a C function but should not be provided for any other type of command.
`class`	Name of the Java class that implements the command. The class is assumed to be located in the package named in this command's component library. This element is required for Java methods but should not be provided for any other type of command.
`method`	Name of the Java or COM method that implements the command. A method name is required for commands implemented by a Java or COM method but should not be provided for any other type of command.
`authorization`	Tree of child XML elements that are associated with controlling access to this command. See "Command Authorization" (on page 115) later in this chapter. **Child Element** / **Description** `option` — User role option that is allowed to execute this command.
`security-level`	Client security level. These are the valid values: OPEN, PUBLIC, PRIVATE, ADMIN, SQL, SCRIPT, REMOTE and ALL.
`insecure`	Indicates whether this is insecure. Insecure commands can be executed even if the user has not logged in yet. These are the valid values: yes and no (default).
`disable`	Indicates whether this should be disabled. These are the valid values: yes and no (default).

Element	Description
`argument`	One element per defined argument. Each argument element contains attributes that describe the argument. Every argument to command implemented by C functions and COM methods are required to be defined. Commands implemented by local syntax do not require arguments to be defined, and any arguments that are defined are for documentation purposes only. See the table that follows for a description of the arguments.
`read-only`	Indicates whether the command definition file or any of its commands and triggers can be changed using the cmdlib API. These are the valid values: yes and no (default).
`documentation`	Tree of child XML elements that document the command. For more information, see "Documenting Commands and Triggers" (on page 111).
`version`	Version of the command definition file. This is optional and can be used as a container for a revision control system ID or version.
`transaction`	Transaction setting for the command. This allows for this command to run in a separate transaction. These are the valid values: required (default) and new.

This table describes each attribute of the `argument` element that can exist in a command definition.

Argument Attribute	Description
`name`	Name of the argument. An argument name is required.
`alias`	Alias of the argument. An argument alias is optional.
`default-value`	Default value of the argument to pass to the implementation of the command if one is not provided. A default value is optional.
`datatype`	Data type of the argument. These are the valid values: integer, double, float, string, pointer, object, flag and results. A data type is optional for local command syntax arguments.
`required`	Indicates whether the argument is required. The default value is no.

This table describes each value for the `transaction` element that can exist in a command definition.

Value	Description
`new`	Value forces a new transaction to be created for the scope of the command.
`required`	Value uses the current transaction for the scope of this command. This is the default value.

IMPORTANT: Care should be taken when using the transaction element to create a new nested transaction. It can have adverse effects when used in a "single threaded" MOCA environment and C code is executed in the scope of the nested transaction. Usually this is shown when transaction hooks on the C side are run more than they should.

Trigger Definition Files

Description

Trigger definition files are used to define the implementation of triggers that may reside within a component library, as well as to disable and enable the implementation of triggers that exist at lower levels.

All trigger definition files for a component level definition file must exist in one of the following directories:

- If a <directory> tag is specified in the component level definition file, the specified directory
- If a <directory> tag is not specified, a directory with the following characteristics:
    - Subdirectory of the directory in which the component level definition file exists
    - Named with the same name (without the extension) as the component level definition file

All trigger definitions files have these rules:

- The filename should be named with the name of the command on which the trigger fires, the name of the trigger itself and it must have a .mtrg suffix (for example, `scan_serial_numer-my_trigger.mtrg`)
- Only one trigger definition is allowed per file

Trigger Definition

This code is an example of a trigger named "my trigger" on the command `scan serial number`. After the **scan serial number** command completes execution, the local syntax defined in this trigger is executed by the MOCA server.

```
<trigger>
 <name>my trigger</name>
 <on-command>scan serial number</on-command >
 <description>
  This is a trigger on the scan serial number command
 </description>
 <local-syntax>
  log scanned serial number where @+serial_number
 </local-syntax>
 <fire-sequence>100</fire-sequence>
</trigger>
```

Trigger Disable Definition

This code is an example of a trigger definition that disables the existing trigger "my trigger" on the **scan serial number** command.

```
<trigger>
 <name>my trigger</name>
 <on-command >scan serial number</on-command >
 <disable>yes</disable>
</trigger>
```

Trigger Enable Definition

This code is an example is a trigger definition that enables the existing trigger "my trigger" on the **scan serial number** command.

```
<trigger>
 <name>my trigger</name>
 <on-command >scan serial number</on-command >
 <enable>yes</enable>
</trigger>
```

XML Element Descriptions

This table describes each element that can exist in a trigger definition.

Element	Description
`name`	Name of the component level. A component level name is required.
`on-command`	Command the trigger fires on.
`description`	Description of the component level. A component level description is optional.
`fire-sequence`	Sequence the trigger fires if more than one trigger exists on the same command. The lowest number sequence triggers are fired first.
`local-syntax`	Local syntax that the command executes. Local syntax is required for commands implemented by local syntax but should not be provided for any other type of command.
`enable`	Indicates whether the trigger should be enabled. This element should only be used to enable triggers that have already been disabled. If this element is used, the "name" and "on-command" elements are the only other required elements. These are the valid values: yes and no.
`disable`	Indicates whether the trigger should be disabled. This element should only be used to disable triggers that have already been enabled. If this element is used, the "name" and "on-command" elements are the only other required elements. These are the valid values: yes and no.
`argument`	One element per defined argument. Each argument element contains attributes that describe the argument. Because all triggers are implemented by local syntax, arguments are not required to be defined, and are only used for documentation purposes. See the table that follows for a description of the arguments.
`read-only`	Indicates whether the trigger definition file or any of its commands and triggers can be changed using the cmdlib API. These are the valid values: yes and no (default).
`documentation`	A tree of child XML elements that document the command. For more information, see "Documenting Commands and Triggers" (on page 111).
`version`	Version of the trigger definition file. This is optional and can be used as a container for a revision control system ID or version.

This table describes each attribute of the `argument` element that can exist in a trigger definition.

Argument Attribute	Description
`name`	Name of the argument. An argument name is required.
`alias`	Alias of the argument. An argument alias is optional.
`default-value`	Default value of the argument to pass to the implementation of the command if one is not provided. A default value is optional.
`datatype`	Data type of the argument. These are the valid values: integer, float, string, pointer, object, flag, and results. A data type is optional for local command syntax arguments.
`required`	Indicates whether the argument is required. The default value is no.

Initializing and Versioning Component Libraries

Description

Sometimes a component library may require some initialization to occur before any methods or functions within the library can be called. Initialization may involve setting the values of global variables or loading a cache. MOCA supports the initialization of component libraries by specifying a method signature or function prototype that you can implement when creating a component library. When the MOCA server starts, it calls the initialization method or function for each component library, if one exists. If an initialization method or function is able to successfully initialize itself, it should return a 0 (zero); otherwise, it should return an error code to indicate that it was not able to successfully initialize itself. Component libraries that cannot be successfully initialized are not loaded, and its commands are not available for execution.

The versioning of component libraries is implemented in the same manner. When the MOCA server starts, it calls the version method or function for each component library, if one exists, and stores the version information internally. The version method or function should return the product version associated with the component library. The **list library versions** command returns a list of component libraries and their versions. Component libraries that do not implement a version function have a version of "None".

The sections that follow describe the methods and functions that need to be implemented within a component library to support initialization and versioning.

Java Component Libraries

Java component libraries should expose a class named `ComponentLibrary` in their package with the single method `initialize`. The `initialize` method returns an instance of a `com.redprairie.moca.MocaLibInfo` object, which contains version and product information. For non-product code, the product can be `null`.

This code is an example `ComponentLibrary` class for a custom component library. The component library is not product code, so a `null` is passed to the `MocaLibInfo` constructor for the product.

```
package com.customer.project.components;

import com.redprairie.moca.MocaLibInfo;

public class ComponentLibrary {
    public static final String VERSION = "1.22";
    public static final String PRODUCT = null;

    public MocaLibInfo initialize() {
        return new MocaLibInfo(VERSION, PRODUCT);
    }
}
```

Code that comes from Product Development should take a slightly different approach. There is some convenience code built in to MocaLibInfo that derives the version number from a resource, located in the same package as the concrete implementation class of MocaLibInfo. This allows sub-classes to be developed for specific products (or even specific libraries) and the resource determines the reported version. This code is an example of MocaLibInfo.

```
package com.redprairie.myCode;

import com.redprairie.moca.MocaLibInfo

public class MyCodeLibInfo extends MocaLibInfo {
    public MyCodeLibInfo() {
        super("myCode");
    }
}
```

The `build.properties` file should be in the `resources` subpackage of that class's package, and should contain the property `releaseVersion`. This is an example of the `build.properties` file.

resources/build.properties

```
releaseVersion=1.23
```

C Component Libraries

C component libraries need to implement a `MOCAInitialize`, `MOCALicense` and `MOCAVersion` function to provide initializing and versioning. This code is an example of initializing and versioning for C component libraries.

Even though MOCA no longer licenses component libraries, `MOCALicense` must still return the name of the product to which the component library belongs.

```
int MOCAInitialize(void)
{
    return 0;
}

char *MOCALicense(void)
{
    return "moca";
}

char *MOCAVersion(void)
{
    return "2010.1.0.0";
}
```

COM Component Libraries

COM libraries are somewhat of a hybrid of Java and C component libraries. Three separate methods need to be exposed to support initializing and versioning, but they all need to be implemented from a `MOCA` class. This code is an example of initializing and versioning for COM component libraries.

```
public class MOCA
{
    public int Initialize( ) { return 0; }

    public string License( ) { return "moca"; }

    public string Version( ) { return "2010.1.0.0"; }
}
```

Documenting Commands and Triggers

Description

MOCA provides the ability to document commands and triggers alongside the actual command and trigger definition. The `mbuild` utility can then be used to generate HTML-based documentation, including project-based extensions, from the definition file.

This is an example of documentation defined in the **list library versions** command.

```
<command>
 <name>list library versions</name>
 <description>
  List the versions of component libraries.
 </description>
 <type>C Function</type>
 <function>mocaListLibraryVersions</function>
 <insecure>yes</insecure>
 <argument name="category" datatype="string">
The component level name of the component library to get the version
for.
 </argument>
 <documentation>
  <remarks>
   <![CDATA[
This command lists the versions of the component libraries in the
current commands memory file.
   ]]>
  </remarks>
  <retrows>
The number of rows returned is equal to the number of component
libraries the installed product contains.
```

```
  </retrows>
  <retcol name="category" type="string">Category</retcol>
  <retcol name="library_name" type="string">
Library name
  </retcol>
  <retcol name="version" type="string">Library version</retcol>
  <exception value="eOK">
The command completed successfully.
  </exception>
  <seealso cref="list commands"/>
 </documentation>
</command>
```

XML Element Descriptions

This table describes each element that can exist within the documentation element of command and trigger definitions.

Element	Description
`private`	Document to which the command or trigger is private for the component library and should not be referenced by any other commands.
`remarks`	Documents a detailed explanation of the command or trigger. **Note**: In this section, it is often necessary to include HTML markup to format the text appropriately. Use the CDATA tag for that purpose. If no CDATA region is specified, then the text is treated the same as it is in any other section.
`retrows`	Description of the component level. A component level description is optional.
`retcol`	Sequence in which the trigger fires if more than one trigger exists on the same command.
`exception`	Exception that the implementation of the command is capable of raising. Each exception element only defines one exception condition and requires an attribute named "value", whose value should be the name of the exception. If your command is capable of raising more than exception, you must define an `exception` XML element for each exception.
`policy`	One element per policy that dictates the behavior of the command. Each policy element can contain attributes. See the table that follows for a description of each attribute of the policy element.
`example`	One element per example of code that you want to document.
`called-by`	One element per command by which this command is called. Each called-by element requires an attribute named "cref" whose value should be the name of the command that calls this command.
`seealso`	One element per command being referenced. Each seealso element requires an attribute named "cref" whose value should be the name of the command this command references.

This table describes each attribute of the policy element that can exist in a command or trigger definition.

Policy Attribute	Description
`polcod`	Code for the policy.
`polvar`	Variable for the policy.
`polval`	Value for the policy.
`rtnum1`	Return number 1 for the policy.
`rtnum2`	Return number 1 for the policy.
`rtflt1`	Return float number 1 for the policy.
`rtflt2`	Return float number 1 for the policy.
`rtstr1`	Return string 1 for the policy.
`rtstr2`	Return string 1 for the policy.

MBuild Warnings

Description

To improve overall code quality, the MOCA syntax parser produces warnings when compiling local syntax. Those warnings are intended to enforce best practices and find subtle bugs that may slip through due to the dynamic nature of MOCA.

Unresolved Command Warnings

In the past, any commands that were misspelled in a local syntax command could go unnoticed until that particular command (or even, that part of that command) was executed. To improve overall code quality, local syntax commands and triggers are checked when mbuild is run to ensure that any referenced commands are available in the command repository. If an invalid command is encountered, a warning is issued. This is an example of an issued warning when an invalid command is encountered.

```
moca-dev :>mbuild

Mbuild 2011.1.0a1 - Wed Jan 12 23:13:42 2011

Copyright (c) 2002-2010 RedPrairie Corporation. All rights reserved.

WARNING (/opt/rpdev/trunk/mcs/src/cmdsrc/MCSReporter/print_
report.mcmd): Referenced command not found: print crystal report
WARNING (/opt/rpdev/trunk/mcs/src/cmdsrc/MCSReporter/ping_moca_
report_server.mcmd): Referenced command not found: get crystal report
server information
WARNING (/opt/rpdev/trunk/mcs/src/cmdsrc/mcsbase/change_dashboard_
tab_assignment.mcmd): Referenced command not found: change
description
WARNING (/opt/rpdev/trunk/mcs/src/cmdsrc/MCSReporter/list_moca_
report_printers.mcmd): Referenced command not found: get crystal
report server information
```

```
WARNING (/opt/rpdev/trunk/mcs/src/cmdsrc/mcssecurity/create_access_
group_address_authorization.mcmd): Referenced command not found:
validate stack variable not null
.
.
.
```

Some of the referenced commands are only called conditionally (`print crystal report`), so the warning may be considered normal. In that case, it is OK to ignore the warning or suppress it. (For more information on suppressing, see Suppressing New Warnings). Other cases are simple typos (such as "validate stack variable" and "change description").

Too Many Pipes

It has become common practice to create local syntax with a large number of commands piped together. Since MOCA executes piped commands on a stack, having too many piped commands causes the stack to grow excessively and performance suffers. As a result, a message appears stating that the local syntax command has too many commands piped together. The message appears whenever the number of piped commands is larger than 32. This is an example of the message.

```
moca-dev /opt/rpdev/trunk/moca:>mbuild

Mbuild 2011.1.0a3 - Thu Jan 27 10:54:07 2011

Copyright (c) 2002-2010 RedPrairie Corporation. All rights reserved.


WARNING (/opt/rpdev/trunk/mcs/src/cmdsrc/mcsmedia/change_media.mcmd):
line 352.1: depth - pipes nested too deep (43)
.
.
.
```

Suppressing New Warnings

The warning for unresolved commands can be suppressed in cases where the invalid reference is normal. The syntax for suppressing that warning uses the new annotations syntax. Annotations on command calls are allowed inside of local syntax. This code is an example of how to suppress the unresolved commands warning.

```
if (@crystal_installed) {
@SuppressWarnings("noref") print crystal reports where ...
}
```

The warning for stack depth can be suppressed as well, but to do that, you need to apply the annotation to an entire block of code by putting the annotation in front of the braces that surround the block. This code is an example of how to suppress the stack depth warning.

```
some command |
@SuppressWarnings("depth")
{
other command |
other command |
...
}
```

Note: The stack depth warning is a strong indicator of a bad coding practice and indicates that you should consider rewriting the component in a different way, perhaps using Java.

Command Authorization

To control a user's access to a MOCA command, MOCA provides the `authorization` element for use in the command's definition file. See "Command Definition Files" (on page 102) earlier in this chapter. Typically, you want to restrict access to MOCA commands, such as administrative commands, to only those users who should be able to run the commands.

The `authorization` element has the following characteristics:

- Its child elements are tagged with the `option` tag.
- It contains one of the following items:
  - One or more role option codes as defined in the Role Maintenance application. MOCA uses the `get user privileges` MCS command with the argument option type of `U` to determine the user's role options for the command.
  - Static open option, `optOpen`. See Open Authorization later in this chapter.
  - Static closed option, `optClosed`. See Closed Authorization later in this chapter.
- If the `authorization` element is not specified, open authorization applies to the command.

Format: The following code illustrates the format of the command authorization elements.

```
<command>
    ...
    <authorization>
        <option>[Option1]</option>
        <option>[Option2]</option>
        .
        .
        .
        <option>[OptionN]</option>
    </authorization>
    ...
</command>
```

Example: The following code illustrates a role option defined for the `publish data` command. The user executing the command must have the `optMocaAdmin` role option or the command is not executed.

```
<command>
    <name>publish data</name>
    <description>Publish data as a result set.</description>
    <type>Java Method</type>
    <class>CoreService</class>
    <method>publishData</method>
    <authorization>
        <option>optMocaAdmin</option>
    </authorization>
</command>
```

If a user attempts to run a command to which the user is not authorized, MOCA returns a not authorized exception to the client.

For information on the authorization implementation for web service endpoints, see "Authorization" (on page 218).

Open Authorization

Open authorization skips authorization. By default, all commands are open to mimic legacy behavior. To explicitly specify that a command is open and any user can execute it, you can specify the static role option, `optOpen`. However, since MOCA associates the `optOpen` role option with any command that does not specify a role option, you do not need to explicitly specify `optOpen` for a command.

Note: The `optOpen` role option does not exist in Role Maintenance.

Example: The following code illustrates a command definition that explicitly supports open authorization.

```
<command>
    <name>publish data</name>
    <description>Publish data as a result set.</description>
    <type>Java Method</type>
    <class>CoreService</class>
    <method>publishData</method>
    <authorization>
        <option>optOpen</option>
    </authorization>
</command>
```

Example: The following code illustrates a command definition that supports open authorization by not specifying the `authorization` element.

```
<command>
    <name>publish data</name>
    <description>Publish data as a result set.</description>
    <type>Java Method</type>
    <class>CoreService</class>
    <method>publishData</method>
</command>
```

The `optOpen` role option is similar to the `MocaWebAuthorization.OPEN` static variable that is used to specify open authorization for web service endpoints. See "Open Authorization" (on page 220).

Closed Authorization

Closed authorization prevents any user from executing the command when it is used by itself or as the first command in a command stream. See "Command Streams" (on page 117) later in this chapter.

Closed authorization is specified using the static role option, `optClosed`.

Example: The following code illustrates a command definition that supports closed authorization.

```
<command>
    <name>publish data</name>
    <description>Publish data as a result set.</description>
    <type>Java Method</type>
    <class>CoreService</class>
    <method>publishData</method>
    <authorization>
        <option>optClosed</option>
    </authorization>
</command>
```

Command Streams

MOCA only authorizes commands that are used alone or as the first command in a command stream or local syntax script. Therefore, after the first command, MOCA does not verify subsequent commands that are part of a command stream or piped together in local syntax. This behavior reduces issues with managing users and their collection of role options.

Example: Given that the `publish data` command specifies closed authorization, the following code is not executed because the closed command is executed by itself.

```
publish data where value = @i
```

Example: Given that the `publish data` command specifies closed authorization, the following code is not executed because the closed command is the first command in the command stream. None of the commands in the command stream are executed, regardless of their authorization.

```
publish data where value = @i|<Command 2>|<Command 3>|...
```

Example: Given that `command 1` is authorized for the user and the `publish data` command specifies closed authorization, the following code is executed (including the `publish data` and all subsequent commands regardless of authorization), since the first command passes authorization.

```
<command 1>|publish data where value = @i|<Command 3>|...
```

For more information on command streams and local syntax piping, see "Local Syntax" (on page 73).

Option Type

MOCA command authorization uses the same option type as MOCA web services authorization: `U`.

Chapter 9. Developing Components and Applications in Java

Description

This section describes MOCA's Java support and how to develop MOCA components and applications in the Java programming language. The content is not meant to be an all-inclusive resource of all the details of writing components and applications in Java, but it should provide a good base on which to build. For in-depth examples of how to develop components and applications in Java, you can review the existing code base.

Earlier versions of MOCA had primitive support for Java components. There were inherent limitations on what could be done in Java; therefore, the capability essentially went unused. Recently, MOCA support for Java components has been added.

IMPORTANT: Unless otherwise stated, all of the classes and interfaces mentioned in this section are located in the Java package `com.redprairie.moca`. For example, `MocaResults` has the fully-qualified class name of `com.redprairie.moca.MocaResults`.

Javadoc

The documentation for all of the interfaces, classes and methods discussed in this section are available in the MOCA Javadoc, and you are encouraged to learn and understand the Java API exposed by MOCA. If the Javadoc has already been generated, it exists in this directory: `%MOCADIR%\docs\api`. If necessary, it can be generated by running `ant javadoc` in the `%MOCADIR%` directory.

Writing a Java Component

MOCA components in Java are implemented as simple Java classes. The MOCA server uses reflection to find classes and methods to invoke based on a command's definition. A MOCA Java command definition includes the class name, method name and a list of arguments. When a command implemented in Java is invoked, the MOCA server finds the class and method in the `CLASSPATH` using reflection, instantiates the class if necessary, and calls the method, passing the defined arguments' values from the stack. For more information on how to create component library and command definition files for Java components, see "Command Repository" (on page 99).

The first thing you need to do is create a component library by writing a component level definition file with a package name that contains all the classes that implement the Java components. This code is an example of the contents of a component level definition file named `dlxsamples.mlvl`.

```
<component-level>
 <name>DLXSamples</name>
 <description>Sample Library</description>
 <package>com.redprairie.dlx.samples</package>
 <sort-sequence>9999</sort-sequence>
</component-level>
```

Each component also requires a command definition file. This code is an example of a command definition file that defines the sample `split string` command that is implemented by the Java method `splitString` in the `StringUtils` class and takes two arguments.

```
<command>
 <name>split string</name>
 <description>Split a string on a regex.</description>
 <type>Java Method</type>
 <class>StringUtils</class>
 <method>splitString</method>
 <argument name="source" datatype ="string" required="yes">
 </argument>
 <argument name="delimiter" default-value="," datatype="string">
 </argument>
</command>
```

This code is the implementation of the previous command definition. The package, class and method names match what was defined in the component level and command definition files; however, the argument names do not need to match.

```
package com.redprairie.dlx.samples;

/**
 * This is a sample MOCA server-side component written in Java.
 */

public class StringUtils
{
    /**
     * Return an array of strings consisting of the first argument,
     * split on the regular expression given by the second argument.
     */
    public String[] splitString(String arg, String regex)
    {
        return arg.split(regex);
    }
}
```

After rebuilding the commands memory file using `mbuild` and building a JAR file for the new component library using `ant`, you can test this new command using MSQL with a couple of simple arguments. This code is an example of how to test the new command using MSQL.

```
MSQL> split string where source='A,B,1,2'
   1  /

Executing... Success!

result
------
A
B
1
2
```

```
(4 Rows Affected)
```

The arguments are passed to the method in the same order as they are defined in the command definition file, so only the order is important in the method signature. However, arguments are checked at runtime to make sure the types of each argument match. This table describes how the data types specified in a command definition file translate to their Java counterparts.

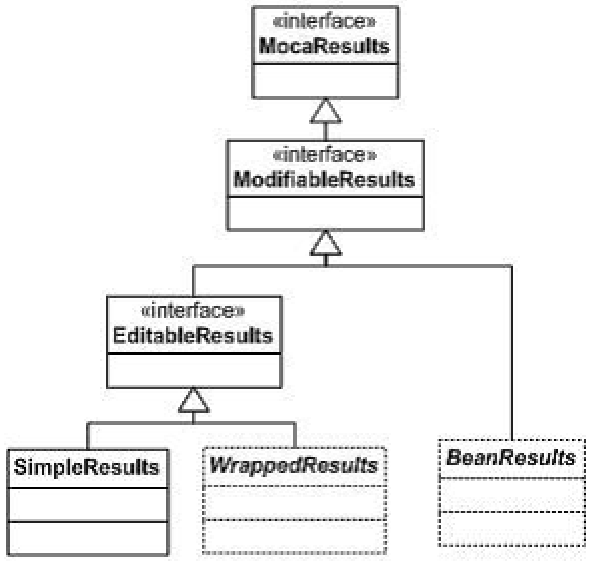
MOCA Type	Java Type
`string`	`string`
`integer`	`int` or `Integer`
`float`	`double` or `Double`
`flag`	`boolean` or `Boolean`
`binary`	`byte[]`
`pointer`	`com.redprairie.moca.server.legacy.GenericPointer`
`results`	`com.redprairie.moca.MocaResults`
`object`	`Object` or specific Type

MocaResults Interface

Description

The code for the `splitString` example method returns an array of Java Strings. (For more information on the `splitString` example, see "Writing a Java Component" (on page 118).) If a component returns a primitive data type (such as a numeric, Boolean or string) or an array of primitive data types, the MOCA server creates a result set consisting of a single column named "result" with a data type corresponding to the data type that was returned from the component.

However, result sets are usually tabular consisting of many columns of different data types. This requires returning a `MocaResults` object rather than a primitive data type. A `MocaResults` object represents a MOCA result set and can be used to read through and process results of commands as well as represent the result set published by a component. There are a number of implementations of the `MocaResults` interface as shown in this class hierarchy:



The MocaResults, ModifiableResults and EditableResults interfaces represent the read-only and read-write versions of a MOCA result set. The MocaResults interface defines the methods to query a result set for its column names and data types as well as to read the values from a result set. The ModifiableResults interface adds the ability to add rows to a result set. Finally, the EditableResults interface adds the ability to change the metadata associated with a result by adding columns.

The SimpleResults, WrappedResults and BeanResults classes implement these interfaces. The SimpleResults class is commonly used and its details are described in this guide. The WrappedResults and BeanResults classes are used internally by the MOCA engine, and the details of these two implementations are outside the scope of this guide.

This code is an example that introduces a change from the previous example in that it declares a MocaContext as its first argument. Components are free to add this optional first argument. The best approach to "future-proof" a component is to have the MocaContext determine the best implementation to return for building a MocaResults. The MocaContext.newResults method returns an instance of an empty EditableResults with no rows or columns defined, which is preferable to writing code that is bound to a concrete implementation of a class.

```
package com.redprairie.dlx.samples;

import com.redprairie.moca.MocaResults;
import com.redprairie.moca.EditableResults;
import com.redprairie.moca.MocaType;

/**
 * This is a sample MOCA server-side component written in Java.
 */

public class StringUtils
{
    /**
     * Return results containing key/value pairs.
     * @param arg
     */
    public MocaResults splitStringMapping(MocaContext moca, String
arg)
    {
        EditableResults res = moca.newResults();
        res.addColumn("key", MocaType.STRING);
        res.addColumn("value", MocaType.STRING);

        String[] elements = arg.split(",");
        for (int i = 0; i < elements.length; i++) {
            String[] keyValue = elements[i].split("\\|");
            res.addRow();
            res.setStringValue("key", keyValue[0]);
            res.setStringValue("value", keyValue[1]);
        }

        return res;
    }
```

This example is the result of testing this new component using MSQL. It results in a tabular result set.

```
MSQL> split string mapping where source='k1|v1,k2|v2,k3|v3'
   1  /

Executing... Success!

key  value
---  -----
k1   v1
k2   v2
k3   v3

(3 Rows Affected)
```

An alternative approach is to define your method to return a non-primitive Java object (or an array of them), which the MOCA server queries for its properties in a JavaBeans style and build a result set similar to how it built a result set in the previous example where a primitive data type was returned.

If you want to return the same two columns as in the previous example, you could define a class that has two properties with each one corresponding to the columns of the result set.

```
public static class MappingResults {

    public MappingResults(String key, String value) {
        _key = key;
        _value = value;
    }

    public String getKey() {
        return _key;
    }

    public String getValue() {
        return _value;
    }

    private final String _key;
    private final String _value;
}
```

Then you can change your method to return instances of this class instead of an instance of `MocaResults`.

```
public MappingResults[] splitStringMapping(String arg)
{
    List list = new ArrayList();

    String[] elements = arg.split(",");
    for (int i = 0; i < elements.length; i++) {
        String[] keyValue = elements[i].split("\\|");
        list.add(new MappingResults(keyValue[0], keyValue[1]));
    }

    return (MappingResults[]) list.toArray(new MappingResults[0]);
}
```

The value of using the "bean results" approach is that, unlike when using `MocaResults`, the data types of columns added to a result set is determined at compile time. Also, if you have a number of methods that return similar results, they can be coded to use the same results class and avoid the duplication of result set initialization code.

MocaColumn Annotation

The behavior of `BeanResults` can be controlled by adding annotations to the getters defined on the class that is being returned to the MOCA server if the default behavior is not desired.

The `MocaColumn` annotation defines these two properties:

- **Name** – Name controls the column name returned in the result.
- **Order** – Order controls the position of each column within the result set.

This code is an example that builds on the previous example by adding annotations to define the name and position of each column.

```
public static class MappingResults {

    public MappingResults(String key, String value) {
        _key = key;
        _value = value;
    }
    @MocaColumn(name="map_key", order = 2)
    public String getKey() {
        return _key;
    }

    @MocaColumn(name="map_value", order = 1)
    public String getValue() {
        return _value;
    }

    private final String _key;
    private final String _value;
}
```

MocaContext Interface

The `MocaContext` interface provides a set of methods for components to execute other commands, enumerate through or lookup arguments from the stack, and write log or trace messages. A `MocaContext` can be obtained from inside a component's code in any of these ways:

- The class that provides the implementation of one or more commands can define a constructor that takes a `MocaContext` as its single argument. The `MocaContext` can then be referenced by the methods within the class.
- A method implementing a command can optionally declare its first argument to be a `MocaContext`. The MOCA server then passes an object representing the `MocaContext` to the method during invocation.
- A method can get the current context from the MOCA server by explicitly calling `MocaUtils.currentContext`.

The preferred approach is to declare each method to accept a `MocaContext` as its first argument.

MOCA Exception Handling

Description

Normal execution of a Java component results in a `MocaResults` object being created and returned to the MOCA server. If a component wants to signal an error condition to the server, it raises an exception. MOCA provides a base exception class named `MocaException` to support raising exceptions from components. To send an error code to the system, it is necessary to throw an instance of `MocaException` out of your method. It is common practice to extend `MocaException` for any application-specific exceptions.

This code is an example of an application-specific exception `StringFormatExecption` that extends the base `MocaException` class.

```
public class StringFormatException extends MocaException {
    public static final int CODE = 1234;
    public StringFormatException(String message) {
        super(CODE, message);
    }
}
```

You could throw the exception when parsing the key/value pairs in the implementation of the previous **split string mapping** command.

```
public MocaResults splitStringMapping(MocaContext moca, String arg)
        throws StringFormatException
{
    EditableResults res = moca.newResults();
    res.addColumn("key", MocaType.STRING);
    res.addColumn("value", MocaType.STRING);

    String[] elements = arg.split(",");
    for (int i = 0; i < elements.length; i++) {
        String[] keyValue = elements[i].split("\\|");
        if (keyValue.length < 2) {
            throw new StringFormatException(
                    "missing delimiter: " + elements[i]);
        }
        if (keyValue.length > 2) {
            throw new StringFormatException(
                    "too many delimiters: " + elements[i]);
        }
        res.addRow();
        res.setStringValue("key", keyValue[0]);
        res.setStringValue("value", keyValue[1]);
    }

    return res;
}
```

In the previous example, an exception is thrown if the current token being parsed does not consist of exactly one key and one value. If this component is executed from a C component using `srvInitiateCommand` or `srvInitiateInline` an error code is returned that corresponds to the error code defined in the exception.

This code is an MSQL session that shows an example of the `StringFormatException` exception defined above being thrown from the component.

```
MSQL> split string mapping where source='k1|v1,k2|v2,k3|notV3,k4'
   1  /

Executing... Error!

ERROR: 223 - missing delimiter: k4

MSQL> split string mapping where source='k1|v1,k2|v2,k3|notV3,k4||b'
   1  /
```

```
Executing... Error!

ERROR: 223 - too many delimiters: k4||b

MSQL>
```

The MOCA server recognizes any exception being thrown from a component as an error condition. If the exception thrown from the component does not extend `MocaException` or `MocaRuntimeException`, the server considers it an unexpected error and writes an exception stack trace. For more information on `MocaRuntimeException`, see "Runtime versus Checked Exceptions" (on page 127).

MOCA defines a few well-known exceptions that are safe to catch explicitly. A `NotFoundException` corresponds to one of these MOCA "not found" error codes: `eSRV_NO_ROWS_AFFECTED` (510) and `eDB_NO_ROWS_AFFECTED` (-1403). The behavior of components when dealing with each of these error codes is essentially interchangeable, so the same exception class is used for both error codes. In addition, `UniqueConstraintException`, located in the package `com.redprairie.moca.db`, is used to indicate a database constraint violation.

This code provides a common usage scenario for a component that calls a "list" command that may raise a `NotFoundException` but wants to continue execution if that occurs rather than raise the exception to the MOCA server.

```
public int countOrderLines(MocaContext moca, String orderid)
        throws MocaException {
    try {
        MocaResults res = moca.executeInline("list order lines where
order_id = '" + orderid + "'");
        int i = 0;
        while (res.next()) {
            i++;
        }
        return i;
    }
    catch (NotFoundException e) {
        return 0;
    }
}
```

This method declares that it throws `MocaException`, yet it catches `NotFoundException` to handle a special case. Any other exceptions thrown from the **list order lines** command are still raised to the MOCA server because they are not being caught by the code.

Exception Message Translation

When an exception is thrown from a component, its message is translated to the appropriate language. This translation is performed using the `MessageResolver` hook that is configured in the `hooks.xml` file. For more information, see "Configuration" (on page 11).

The error code associated with the exception dictates how it is translated. After the message has been translated, arguments replace any variables that are surrounded by the caret character (^). An argument can be added by calling the `addArg` method on the exception object.

This code is an example of where an exception uses an error code of 2963 and defines a default message associated with that error code, which contains these arguments: "colnam" and "codval". The values for those arguments are added using the `super.addArg` calls that are made.

```
public class CodeInvalidException extends MocaException {
    private static final long serialVersionUID =
1449921785524402935L;

    /**
     * @param errorCode
     * @param message
     */
    public CodeInvalidException(String codeName, String codeValue) {
        super(2963, "Invalid code value (^codval^) for ^colnam^");
        super.addArg("colnam", codeName);
        super.addArg("codval", codeValue);
    }

}
```

Runtime versus Checked Exceptions

Checked exceptions are declared in the throws clause of a method and must be dealt with either using a try/catch block or by declaring that your calling method also throws the same exception. Because this can become tiresome, especially for some fairly simple aspects of the Java language, the Java designers developed the concept of runtime exceptions. Runtime exceptions do not have to be caught or declared. They are allowed to "bubble up" out of any method of any class. Most of the built-in Java runtime exceptions indicate that some sort of programming error has occurred.

Checked exceptions in Java indicate that some unforeseen error situation has occurred, such as a file not being where it was expected to be or a database error occurring. Because of their use in handling MOCA invocation errors, MOCA uses checked exceptions for most operations. It is possible, however, to wrap a `MOCAException` in a runtime exception to allow a low-level exception to bubble up out of a method without having to declare it everywhere.

MOCA has chosen the checked exception approach but allows you to wrap those checked exceptions in `MocaRuntimeException` to avoid the burden of checked exceptions if desired.

JDBC Connections

Many tools exist to help you with handling object persistence, and creating and executing database queries. Universally, all of those tools need to have a JDBC database connection to interact with the database. MOCA employs a JDBC-based database adapter to allow Java code to obtain a `Connection` object to interact directly with the same database that the MOCA server is using for its interactions.

A reference to a MOCA server's database connection can be obtained by calling the `MocaContext.getConnection` method. That `Connection` can then be used in code that calls JDBC directly or to enable third-party persistence tools such as Hibernate or Solarmetric KODO. MOCA has preliminary built-in support for Hibernate. For more information on using Hibernate within MOCA components, see "MocaConnection Interface" (on page 128).

MocaConnection Interface

The `MocaConnection` interface provides methods required for Java applications to connect to a MOCA server as a client in order to execute commands. A `MocaResults` object is returned after successfully executing a command; a `MocaExecption` is raised when an exception is thrown during execution of a command.

The static method `ConnectionUtils.createConnection` found in `com.redprairie.moca.client` can be called to get a `MocaConnection`. The method takes an argument specifying a URL of the MOCA server as well as a `Map` of environment variables to set for the connection. The URL is usually of the form `[http://`*hostname*`:`*port*`/service]`. URLs of the form *hostname*`:`*port* assume a connection to a classic MOCA server using the older socket-based protocol rather than the newer HTTP-based protocol. Once a `MocaConnection` has been created, only insecure commands can be called until the connection is authenticated using a call to the static method `ConnectionUtils.login`.

A connection should be closed when you are done working with it to ensure the session related to your connection is properly disposed of by the MOCA server. This is especially important given that HTTP is a stateless protocol that does not provide a mechanism for the server to recognize when a connection is closed.

This code establishes a connection to the MOCA server running on port 4500 of the local host, executes the **list order lines** command, and counts the number of rows in the result set returned from the command. The connection is then closed in the finally block, which ensures that the connection is closed even if the command raises an exception.

```
MocaConnection conn = ConnectionUtils.createConnection("http://localhost:4500/service",
null);

try {
    ConnectionUtils.login(conn , "user", "password");

    MocaResults res = conn.executeCommandWithArgs("list order lines", new MocaArgument
("order_id", orderid));

    int count = 0;
    while (res.next()) {
        count++;
    }
}
finally {
    // We want to close out the connection in a finally to make sure we close even on an
error.
    conn .close();
}
```

IMPORTANT: MocaConnection implementations are not thread safe, therefore instances should not be used concurrently across multiple threads. Instead, you should have only one instance per thread.

Hibernate

Description

Hibernate is a third-party Object Relational Mapping tool for reading Java objects from the database. For more information, see the Hibernate website.

Getting a Hibernate Session

A Hibernate session can be established through a call to `HibernateTools.getSession`. The `server.prod-dirs` registry key is used to find the `hibernate.cfg.xml` and `hibernate.properties` configuration files.

Configuring Hibernate

The configuration for Hibernate within MOCA is done through these Hibernate files:

- **`hibernate.cfg.xml`** – The `hibernate.cfg.xml` file contains information about the classes, JARs, resources and files to load, and configuration options for the session factory.
- **`hibernate.properties`** – The `hibernate.properties` file is used to configure global settings for Hibernate.

IMPORTANT: The Hibernate **`hibernate.transaction.factory_class`** setting, which can generally be set using the **`hibernate.cfg.xml`** files, is manually set to org.hibernate.transaction.CMTTransactionFactory by MOCA to ensure the proper configuration for use of the Hibernate auto-flush feature.

Hibernate Mapping Files

Create `hbm.xml` files as you usually do when using Hibernate and place the files in the same directory as their Java source code. The MOCA server can then find the XML files in a JAR file using the `CLASSPATH`. There is only one option for loading an XML file; you use a "mapping" element with a "resource" attribute in the `hibernate.cfg.xml` file. This method requires a separate "mapping" element for each `hbm.xml` file.

This code is an example of loading an XML file.

```
<!DOCTYPE hibernate-configuration PUBLIC
    "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
    "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">

        <hibernate-configuration>
          <session-factory>
            <mapping
resource="com/redprairie/moca/task/TaskDefinition.hbm.xml"/>
          </session-factory>
        </hibernate-configuration>
```

Using Annotations

Alternatively, Hibernate can be used by supplying the appropriate annotations to your class files. To facilitate the MOCA server knowing which classes are annotated, you should add a `hibernate.cfg.xml` file in one of the directories specified by the `server.data-dirs` registry key. This file can then contain various values that are used by the Hibernate session factory. You can either use the:

- **Class element** – The class element would require the fully qualified name of the class.
- **Package element** – The package element requires the package name, which would then include all annotated classes within that package.

This code is an example of the `hibernate.cfg.xml` file.

```
<!DOCTYPE hibernate-configuration PUBLIC
    "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
    "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">

        <hibernate-configuration>
          <session-factory>
            <mapping
class="com.redprairie.moca.task.TaskDefinition"/>
            <mapping class="com.redprairie.moca.job.JobDefinition"/>
            <mapping class="com.redprairie.moca.task.Example"/>
            <mapping package="com.redprairie.moca.core"/>
          </session-factory>
        </hibernate-configuration>
```

Global Interceptors

A MOCA global interceptor is an entity that you can use to monitor and react programmatically to lifecycle events of a Hibernate session. MOCA Hibernate interceptors must implement the following interface:

- **`MocaContextAwareInterceptor`**: You can implement this interface to provide custom hook functionality when entities are loaded, saved, updated, or deleted in a Hibernate session.

MOCA additionally provides a no operation (no-op) implementation of a `MocaContextAwareInterceptor` called an `EmptyMocaContextAwareInterceptor`. This class can be used to implement an interceptor where you only need to override the methods that you want to use.

An interceptor implementation must have the following characteristics:

- **Thread-safe**: Because an interceptor's methods can potentially be called at the same time across multiple threads, the interceptor implementation must be thread-safe. This is typically achieved by a stateless implementation, which is inherently thread-safe.
- **Acceptable performance**: Because it can potentially be called frequently, the implementation must provide an acceptable level of performance. Since the interceptors are registered globally, determining whether your interceptor should take any action should also provide acceptable performance. This determination can be as simple as checking if an entity implements a particular interface (as is the case in the example that follows later in this section).

Task list

To implement and use an interceptor, complete the following tasks:

1. Take one or more of the following actions:
   - Implement the `MocaContextAwareInterceptor` interface.
   - Extend the `EmptyMocaContextAwareInterceptor` class and override the methods that you want to use.
2. Register your custom interceptors using the `registerInterceptor` method of the `HibernateTools` class. For example:

   ```
   HibernateTools.registerInterceptor(new CustomInterceptor());
   ```

   Where `CustomInterceptor` is the class name of the interceptor implementation. After registration, the interceptor is used for all subsequent sessions that are generated using `HibernateTools.getSession()`. The interceptor should only be registered once, typically on start up of the application and using the application's start up process, such as Spring bootstrapping.

Example: logging information about an entity when it changes

Logging information about an entity when it changes is an example use case for which you can use an interceptor. Instead of adding code to an entity to log changes, you can create an interceptor to capture and log entity changes.

Example: The following code illustrates a marker interface that is implemented with entities that are tracked using logging.

```
/**
 * Marks entities as tracked via logging
 */
public interface Loggable {}
```

Example: The following code illustrates the interceptor, `LoggableInterceptor` class, that is used to intercept and log changes to an entity. It extends the `EmptyMocaContextAwareInterceptor` class since it only needs to override the `onFlushDirty` (database update) method. It intercepts a loggable entity that is being updated in the database and writes out information about the entity's properties and state changes.

```
public class LoggableInterceptor extends EmptyMocaContextAwareInterceptor {

    @Override
    public boolean onFlushDirty(MocaContext context, Object entity,
                                Serializable id, Object[] currentState,
                                Object[] previousState, String[] propertyNames,
                                Type[] types) {
        if (entity instanceof Loggable) {
            // Writes out the old state vs new state on changes for Loggable entities
            Logger log = getLogger(entity);
            log.info("Flush dirty\n Properties: {}\n Previous state: {}\n Current state: 
{}",
                propertyNames, previousState, currentState);
        }
```

```
        return false;
    }

    private Logger getLogger(Object entity) {
        return LoggerFactory.getLogger(entity.getClass());
    }

}
```

Example: The following code illustrates registering the interceptor.

```
HibernateTools.registerInterceptor(new LoggableInterceptor());
```

Chapter 10. Developing Components and Applications in C

Description

This section explains writing MOCA components and applications in C. It provides a high-level overview and good starting point for developers that need to maintain and support existing C code. For in-depth examples of how to develop components and applications in C, you can review the existing code base. However, because the MOCA engine is written in Java, it is suggested that you favor Java-based components and applications over C-based ones.

Note: For C API documentation, see previous versions of the *MOCA Developer Guide* within the `%MOCADIR%\docs` directory.

Writing a C Component

Description

C-based MOCA components are implemented as C functions that declare zero or more arguments and return a pointer to a `RETURN_STRUCT` structure defined by MOCA. The `RETURN_STRUCT` represents the result set that is returned to the MOCA server. Component libraries containing C functions require a library name to be defined in their component level definition file that maps a component library to a C DLL or shared library. The command definition file for a C component then contains the function name and any arguments. The names of any arguments are unimportant. However, the order of those arguments must match the order as they are defined in the function prototype. When a command is invoked that is implemented in C, the MOCA server obtains a function pointer to the associated function within the DLL or shared library and uses it to call the function including passing any arguments. For more information on how to create component library and command definition files for C components, see "Command Repository" (on page 99).

Component Level Definition File

This code is the component level definition file for the "VARint" component library. On Windows platforms, a library named `VARint.dll` needs to exist in the search path. On Linux and UNIX platforms, the shared library named `VARint.sl` or `VARint.so` needs to exist in the appropriate library search path.

```
<component-level>
<name>MOCAbase</name>
<description>MOCA Base Components</description>
<library>MOCAbase</library>
<sort-sequence>0</sort-sequence>
</component-level>
```

Command Definition File

This code is the command definition file for the MOCA **get file info** command. The command defines a component type of "C Function", a function name of "mocaGetFileInfo" and a single argument of "pathname".

```
<command>
 <name>get file info</name>
 <description>Get file information.</description>
 <type>C Function</type>
 <function>mocaGetFileInfo</function>
 <argument name="pathname" required="yes" datatype="string">
 </argument>
</command>
```

Source Code

Using the example **get file info** command definition, this code is an example of the implementation of the command. The function is declared to return a pointer to a `RETURN_STRUCT` and takes a single argument representing the path name. Because the function must be callable by the MOCA server from outside the library, the `LIBEXPORT` macro should be used in the function declaration for all C-based components to ensure the function is "exported" from the library.

```
LIBEXPORT
RETURN_STRUCT *mocaGetFileInfo(char *pathname)
{
    long size;
    char type, *created, *accessed, *modified;

    RETURN_STRUCT *ret = NULL;

    .
    .
    .

    /* Get all the file information. */
    osFileInfo(ptr, &type);
    osFileSize(ptr, &size);
    osFileCreated(ptr, &created);
    osFileAccessed(ptr, &accessed);
    osFileModified(ptr, &modified);

    /* Create the return structure. */
    ret = srvResults(eOK,
                     "pathname", COMTYP_CHAR,         1024,  pathname,
                     "type",     COMTYP_CHAR,            1,  &type,
                     "size",     COMTYP_INT,  sizeof(long),  size,
                     "created",  COMTYP_CHAR,          100,  created,
                     "accessed", COMTYP_CHAR,          100,  accessed,
                     "modified", COMTYP_CHAR,          100,  modified,
                     NULL);

    return ret;
}
```

Argument Data Type Mapping

Arguments are passed to the command implementation function in the same order as they are defined in the command definition file, so only the order is important in the function declaration. However, the type of the argument as it is defined in the command definition file should also match the type of the argument in the function declaration.

This table lists the valid argument data types that can be defined in a command definition file along with how the corresponding arguments should be defined in the C function. All arguments are passed to C functions by reference regardless of the data type.

Argument Type	C Type
`integer`	`long *`
`float`	`double *`
`string`	`char *`
`pointer`	`void *`
`object`	`void *`
`flag`	`long *`
`results`	`mocaDataRes **`

RETURN_STRUCT Structure

Description

The `RETURN_STRUCT` structure defined by MOCA encapsulates the tabular result set that is returned to the MOCA server from the C function along with a status code and associated message. While the result set and message are optional and do not need to be set within a `RETURN_STRUCT`, the status code must always be set in order to communicate to the MOCA server if the execution of the component was successful or if a failure occurred.

A number of `srvlib` functions are exposed from the `MOCA.dll` Windows DLL or `libMOCA.sl` Linux/UNIX shared library to support creating and manipulating a `RETURN_STRUCT` from C code.

Result Set Data Type Mapping

Previously the example source code called `srvResults` was used in the `RETURN_STRUCT` structure. Anytime a `RETURN_STRUCT` is built in this manner it is important to provide the correct metadata for the result set, including the name of each column along with its data type and the maximum length of any value that it can store. The data types for each column are represented by a set of MOCA "communication type" macros defined in `common.h`. This table provides the mapping of these macros to their corresponding C data type.

MOCA Type	C Type
`COMTYP_CHAR` or `COMTYP_STRING`	`char *`
`COMTYP_DATTIM`	`char *`
`COMTYP_INT` or `COMTYP_LONG`	`long`
`COMTYP_FLOAT`	`double`
`COMTYP_BOOLEAN`	`long`
`COMTYP_BINARY`	`void *`
`COMTYP_RESULTS`	`mocaDataRes *`
`COMTYP_POINTER` or `COMTYP_GENERIC`	`void *`

mocaDataRes Structure

The `mocaDataRes` structure defined by MOCA is the representation of the tabular result set and is also included within a `RETURN_STRUCT`. Behind the scenes the `srvResults` call in the previous example source code created a `mocaDataRes` using the meta data and actual values for the result set.

The functions to create and manipulate a `mocaDataRes` from C code are provided by the `sqllib` library and are exposed from the `MOCA.dll` Windows DLL or `libMOCA.sl` Linux/UNIX shared library. In addition, many of the `sqllib` functions populate a `mocaDataRes` for the caller. For example, a C component that executes an SQL statement against a database can call `sqlExecStr`, which returns a status code and populate a `mocaDataRes` with the results of the given SQL statement.

A set of functions exist in `srvlib` that support extracting a `mocaDataRes` from a `RETURN_STRUCT` as well as creating a `RETURN_STRUCT` from a `mocaDataRes`.

Writing Server Applications

Description

MOCA-based server applications can be thought of as a simple MOCA server where commands are initiated directly from the application rather than clients, jobs or tasks.

Prerequisites

Prior to trying to execute any commands or SQL statements, server applications are required to make a call to `srvInitialize`. This function performs the necessary bootstrapping necessary for server applications including starting a JVM. The `srvInitialize` function takes these arguments:

- A process name, which is used as the session ID in trace messages and on the Sessions page of the MOCA Console
- A flag indicating if the application should run as a single-threaded (`1`) or multi-threaded (`0`) MOCA-based server application

While the C code for server applications must be single-threaded because the C API provided by MOCA is not thread-safe, a server application could execute components that perform some multi-threading during execution. This is OK and supported but requires telling MOCA that your application may be calling these components so that the bootstrapping that is performed by the `srvInitialize` call can take that into account. MOCA does not assume multi-threading, because it requires additional intialization and overhead that a single-threaded component or application does not need. To help you decide whether to use single-threading or multi-threading, these are the behaviors and architectures of server applications running in single-threaded versus multi-threaded mode:

- Single-Threaded Applications:
    - The execution of only single-threaded Java-based components is supported. Java-based components that attempt to perform any multi-threaded operations result in undefined behavior.

    - The overall footprint consists of only the application itself.
    - C-based components execute directly inside the application.
    - C-based components that crash during execution cause the entire application to crash.
- Multi-Threaded Applications:
    - The execution of multi-threaded Java-based components is supported.
    - The overall footprint consists of the application as well as one or more native processes. These are started to host the execution of C-based components just as they are by the MOCA server. The implication is that a potentially much larger footprint exists if a number of native processes need to be started to run C-based components in parallel.
    - C-based components execute outside of the application in a "native process".
    - C-based components that crash during execution return a 537 error code rather than causing the entire application to crash.

Transactions

Perhaps the most important characteristic of server applications is that they control their own commit context, whereas a MOCA server automatically issues a commit or rollback after a command completes. This implies that it is absolutely critical that if you write server applications that you are aware of the implications of executing SQL statements that may request locks of database resources. You must keep their transactions as short-lived as possible and perform a commit or rollback using a call to `srvCommit` or `srvRollback` to complete their transactions as soon as possible.

Writing Client Applications

MOCA exposes a number of `mcclib` functions from the `MOCA.dll` Windows DLL or `libMOCA.sl` Linux/UNIX shared library. These function can be called from C-based client applications to establish a connection to a MOCA server, authenticate users, execute commands and log out. When commands are executed from client applications, a status code and `mocaDataRes` are returned. The `mocaDataRes` can be interrogated using `sqllib` functions just as it would be from a component.

The majority of the `mcclib` functions that can be called from C-based client applications are thin wrappers around the same methods that Java-based applications would call. Therefore, it is important to understand that client applications start a Java VM in order to support communicating with a MOCA server. Aside from that, establishing a connection to a MOCA server and executing commands is straightforward and is well documented in older versions of the *MOCA Developer Guide*. For more information, see previous versions of the *MOCA Developer Guide* within the `%MOCADIR%\docs` directory.

Chapter 11. Jobs

Description

This section explains the concept of jobs. More specifically, it provides a description of jobs and explains:

- How job tracing works.
- How jobs work in a clustered environment.
- Where you can monitor jobs.
- Where you can stop and start job scheduling.
- Where and how you configure jobs.
- The database tables that contain the job configuration details.

Job Description

Description

A job is a command that is configured to run in the background while the MOCA server is operating. Jobs are executed within a thread of the MOCA server, and they can be:

- **Timer-based** – Timer-based jobs are started by the MOCA server at set intervals that you define.
- **Schedule-based** – Schedule-based jobs are started by the MOCA server at predefined times that you configure using a Quartz-based timer similar to the UNIX cron utility.

Jobs are configured on the Jobs page of the MOCA Console, stored in the database, and managed by the MOCA server's Job Manager.

Job Tracing

Logging for jobs can be configured on the Jobs page of the MOCA Console. If logging is enabled, messages from the job at all logging levels are logged. If a log file is defined for the job, logging messages are written to the job's log file instead of the main server logging output. The MOCA Console should be used for all job maintenance activities instead of the Job Maintenance application in the JDA SCE client.

Job tracing is performed with Apache Log4j 2. With Log4j 2, you can configure an appender to support rolling log files or writing to locations other than a simple log file. For more information on configuring an appender for a job, see "Tracing and Logging" (on page 53).

Monitoring, Stopping and Starting Jobs

You can monitor (view the runtime configuration), and stop and start the scheduling of jobs on the Jobs page in the MOCA Console. Because the Console connects to a single MOCA server, when you are working in a clustered environment, you must ensure that the server to which you connect is running the role which contains the job that you want to monitor, stop or start.

You can also monitor the session associated with a job on the Sessions page in the Console or with a Java tool such as jconsole.

For more information on the Console, see the *MOCA Console User Guide*.

Jobs Page

Description

You can use the Jobs page in the MOCA Console to complete all maintenance and management of the jobs used by your JDA SCE applications. For example, when creating a job you can:

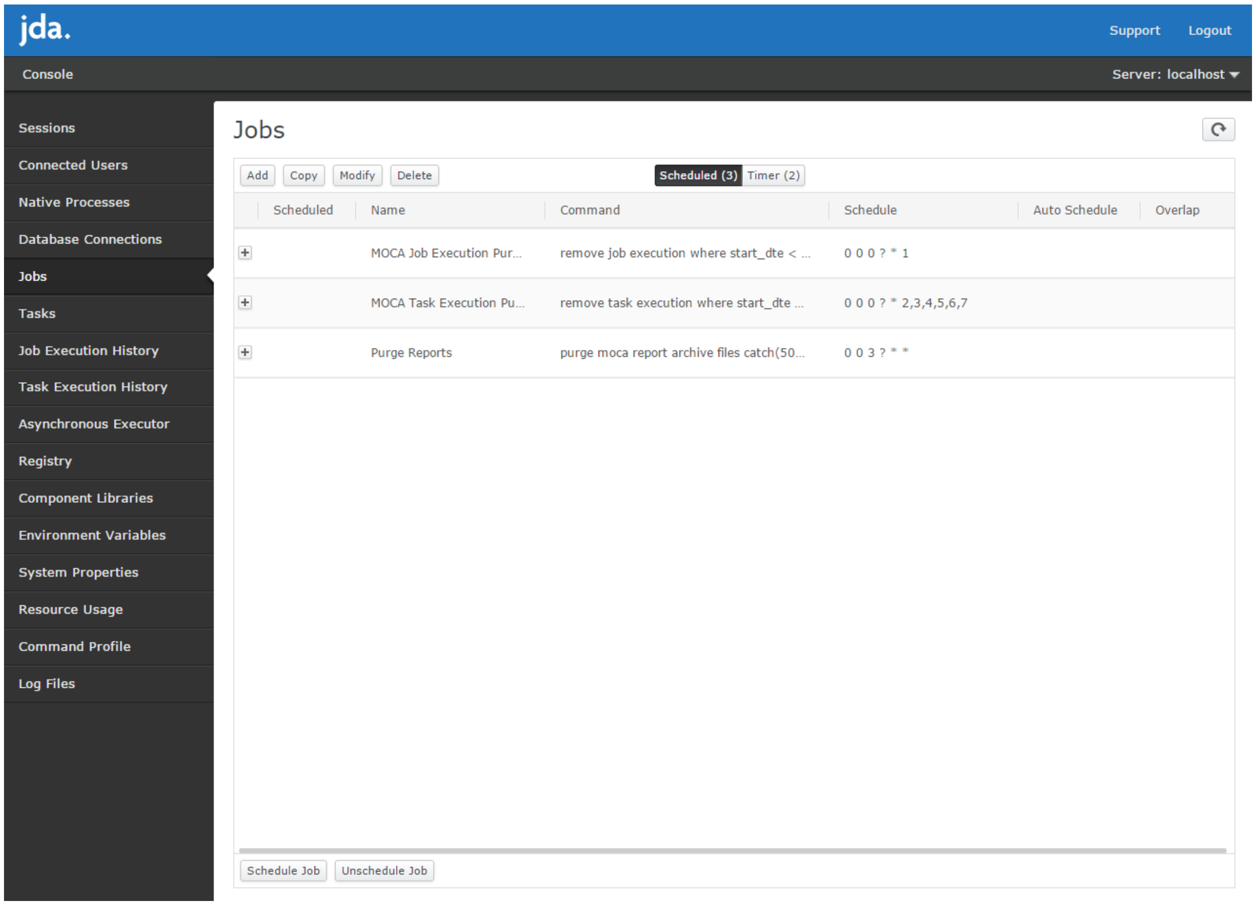
- Specify a unique ID for the job.
- Provide a descriptive name for the job.
- Enter the command that executes the job.
- Optionally, define log file details.
- Optionally, specify a role ID if working in a clustered environment. Tasks and jobs with the same role always run on the same node.
- Indicate whether the job is:
  - Timer- or schedule-based, and then define the timing or schedule for the job.
  - Enabled to start.
  - Able to start even though another instance of that job is currently running.
- Optionally, define environment variables that are specific to the job.

After initially configuring a job, you typically do not need to update the configuration, except for a few options that are used on an as-needed basis, such as tracing to research an issue with a job.

IMPORTANT: You should maintain jobs on the Jobs page of the MOCA Console, not the Job Maintenance application that is accessed from the JDA SCE client.

Example: Jobs

This image is an example of the Jobs page in the MOCA Console.



Scheduled	Name	Command	Schedule	Auto Schedule	Overlap
	MOCA Job Execution Pur...	remove job execution where start_dte < ...	0 0 0 ? * 1		
	MOCA Task Execution Pu...	remove task execution where start_dte ...	0 0 0 ? * 2,3,4,5,6,7		
	Purge Reports	purge moca report archive files catch(50...	0 0 3 ? * *		

Job Execution History Page

Description

You can use the Job Execution History page in the MOCA Console to view information about the MOCA jobs that have run. In a clustered environment, the Job Execution History page displays all of the jobs that have run on all of the nodes in the cluster. Job execution information is useful when troubleshooting jobs and helps answer the following questions:

- When did the job last run?
- On which node did the job last run?
- How long did the job take to run?
- Did the job complete successfully?

By default, the execution history is sorted by **Start Date**, with the most recent execution on the first page in the first row of the grid, and the oldest execution on the last page in the last row of the grid.

For each job execution, you can view the following information:

- **Job ID** – Unique identifier that is specified when creating the job.
- **Node URL** – URL of the node on which the job ran.

- **Status** – Value that the job returned after it finished running:
  - **0** – The job completed successfully.
  - **Anything else** – The job did not complete successfully.
- **Message** – Additional information, if any, returned by the job. For example, if the job involved retrieving MOCA results that returned an error, a message is also returned that provides more information on the error.
- **Start Date** – Date and time that the job started running.
- **End Date** – Date and time that the job stopped running.

You can complete the following tasks:

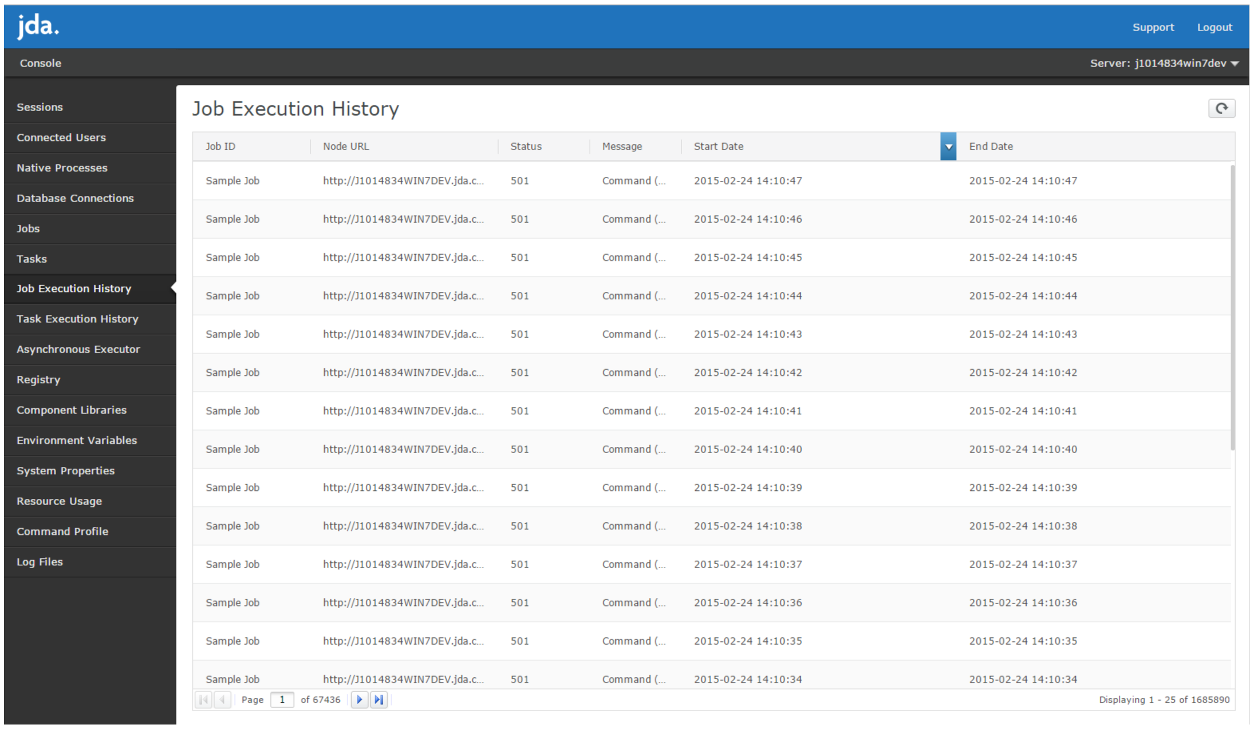
- View the job execution history on a different page:

  Note: Each page displays 25 rows except the last page, which can display fewer than 25 rows.

  - To view the first page, click ⏮.
  - To view the last page, click ⏭.
  - To view the next page, click ▶.
  - To view the previous page, click ◀.
  - To view a specific page, in the **Page** box, enter the page number.
- Sort the job execution history:
  - To sort the rows on the current page, click the column heading.
  - To sort the rows on all pages, click the column heading, and then click ⟳.

Example: Job Execution History

This image is an example of the Job Execution History page in the MOCA Console.



Job Execution History

Job ID	Node URL	Status	Message	Start Date	End Date
Sample Job	http://J1014834WIN7DEV.jda.c...	501	Command (...	2015-02-24 14:10:47	2015-02-24 14:10:47
Sample Job	http://J1014834WIN7DEV.jda.c...	501	Command (...	2015-02-24 14:10:46	2015-02-24 14:10:46
Sample Job	http://J1014834WIN7DEV.jda.c...	501	Command (...	2015-02-24 14:10:45	2015-02-24 14:10:45
Sample Job	http://J1014834WIN7DEV.jda.c...	501	Command (...	2015-02-24 14:10:44	2015-02-24 14:10:44
Sample Job	http://J1014834WIN7DEV.jda.c...	501	Command (...	2015-02-24 14:10:43	2015-02-24 14:10:43
Sample Job	http://J1014834WIN7DEV.jda.c...	501	Command (...	2015-02-24 14:10:42	2015-02-24 14:10:42
Sample Job	http://J1014834WIN7DEV.jda.c...	501	Command (...	2015-02-24 14:10:41	2015-02-24 14:10:41
Sample Job	http://J1014834WIN7DEV.jda.c...	501	Command (...	2015-02-24 14:10:40	2015-02-24 14:10:40
Sample Job	http://J1014834WIN7DEV.jda.c...	501	Command (...	2015-02-24 14:10:39	2015-02-24 14:10:39
Sample Job	http://J1014834WIN7DEV.jda.c...	501	Command (...	2015-02-24 14:10:38	2015-02-24 14:10:38
Sample Job	http://J1014834WIN7DEV.jda.c...	501	Command (...	2015-02-24 14:10:37	2015-02-24 14:10:37
Sample Job	http://J1014834WIN7DEV.jda.c...	501	Command (...	2015-02-24 14:10:36	2015-02-24 14:10:36
Sample Job	http://J1014834WIN7DEV.jda.c...	501	Command (...	2015-02-24 14:10:35	2015-02-24 14:10:35
Sample Job	http://J1014834WIN7DEV.jda.c...	501	Command (...	2015-02-24 14:10:34	2015-02-24 14:10:34

Page 1 of 67436 Displaying 1 - 25 of 1685890

Job Database Tables

Description

Job configuration details are stored in the `job_definition` and `job_env_definition` database tables.

- The `job_definition` table contains the information that describes the job, the commands that run when the job is started, and the timer or schedule that tell the MOCA server when to start the job.
- The `job_env_definition` table contains the unique environment variable values that you can define for a job.

Job Definition Table

This table describes the columns in the `job_definition` table.

Column	Type	Required?	Description
`job_id`	string	✓	Unique identifier for the job in all caps (for example, SYNC-ADRMST). **IMPORTANT**: The job_id must be stored in the database in all caps; otherwise, appenders do not work correctly.
`role_id`	string		For clustered environments, the role that is associated with the job. Roles are used to determine the node(s) on which a job runs. For more information on role managers, see "Clustering" (on page 181).

Column	Type	Required?	Description
name	string	✓	Name of the job.
enabled	integer		Indicates whether the job is enabled. These are the valid values: 0 and 1.
type	string	✓	Type of job. These are the valid values: cron and timer.
command	string	✓	Command to run when the job is started.
log_file	string		Path to and name of the log file to which trace messages for the job are to be written.
trace_level	string		Level of information to generate in a trace file each time a job is run: • *: All trace levels run, with the exception of command profiling.
overlap	integer	✓	Indicates whether another instance of a job can start when a previous instance is still running. These are the valid values: 0 and 1.
schedule	string		Quartz-style schedule for the job. Only used for schedule-based jobs. The job_definition table's schedule column must contain a valid Quartz "CronTrigger" schedule expression to work correctly. Also, information about the schedule expression format can be found on the Quartz documentation website.
start_delay	integer		Number of seconds that the MOCA server waits before starting a job for the first time. Only used for timer-based jobs. If a start delay is not defined, the job is started immediately.
timer	integer		Interval (in seconds) between executions of the job. Only used for timer-based jobs.
grp_nam	string		Name of the group that owns the job. This is display only, and it assists in administration and debugging.

Job Environment Definition Table

This table describes the columns in the job_env_definition table.

Column	Type	Required?	Description
job_id	string	✓	Unique identifier for the job.
name	string	✓	Environment variable name.
value	string		Environment variable value. Environment variable values are expanded at runtime; consequently, a value can include a reference to other environment variables. If a value is not defined, the environment variable is cleared for the job.

Job Commands

You can use the **add job**, **remove job** and **list job** commands to query and manage the `job_definition` and `job_env_definition` database tables. You can view more information about these commands by running the **mgendoc** command. After the command runs, you can find the documentation in the `%LESDIR%\docs\commands` folder.

Note: For more information on configuration of jobs in a clustered environment, see "Clustering" (on page 181).

Chapter 12. Tasks

Description

This section explains the concept of tasks. More specifically, it provides a description of tasks and explains:

- How task tracing works.
- How tasks work in a clustered environment.
- Task exceptions.
- Where you can monitor, stop and start tasks.
- Where and how you configure tasks.
- The database tables that contain the task configuration details.

Task Description

Description

A task is either a program or a Java thread that starts when the MOCA server starts, and it continues to run in the background, as configured, while the MOCA server is operating. A task can be run either as a:

- Completely separate program that runs in parallel with the MOCA server (process-based and daemon-based tasks). Process-based and daemon-based tasks are executables (programs) that can be implemented in any programming language. Process-based and daemon-based tasks are different only in that process-based tasks are restarted when the MOCA server is restarted using the console. Daemon-based tasks continue running across restarts of the MOCA server and are only shut down when the MOCA server is shut down.
- Separate Java thread that runs within the application server itself (thread-based task). Thread-based tasks are fully-qualified Java class names. The task manager instantiates thread-based tasks using the Java class name that implements the Java runnable interface.

  Note: A Java thread-based task uses fewer system resources than a process-based or daemon-based task.

Tasks are configured on the Tasks page of the MOCA Console, stored in the database, and managed by the MOCA server's Task Manager.

Task Tracing

Tracing works differently for thread-based, process-based, and daemon-based tasks.

- Thread-based tasks run as a thread within the MOCA server; consequently, trace messages are written to the server's log file using the same configuration associated with the server. If you define a log file for a task, trace messages are also written to the task's log file.
- Process-based and daemon-based tasks run as a separate program; consequently, trace messages are not written to the server's log file. Trace messages are only written to the task's log file using the configuration set by the task itself.

All tasks use Apache Log4j 2 for tracing. With Log4j 2, you can configure an appender to support rolling log files or writing to locations other than a simple log file. For more information on configuring an appender for a task, see "Tracing and Logging" (on page 53).

Special Task Environment Variables

In addition to the standard environment variables that are available from the environment section of the MOCA registry, MOCA provides special task environment variables to help with task definitions and environment configurations. MOCA replaces environment variables with their actual values at run time.

Note: For more information on the environment section of the MOCA registry, see "Environment Section" (on page 30).

The following table describes the special task environment variables.

Environment Variable Name	Value Description
JAVA	Replaced by the value of the `java.vm` MOCA registry key.
PATH_SEPARATOR	Replaced by the platform-specific path separator: • **Windows**: semicolon (;) • **Unix-based systems**: colon (:)

Example: The following configuration illustrates a Java-based process task that redefines the `CLASSPATH` variable.

```
$JAVA -cp /path/to/folder/with/other/dependencies$PATH_SEPARATOR$CLASSPATH
com.company.your.main.class
```

Continuing the example, at run time, MOCA performs the following tasks:

1. Replaces the `$JAVA` environment variable with the value of the MOCA registry entry for `java.vm`. For example, on Windows if the `java.vm` registry key is set to, **C:\Program Files\Java\jre7\bin\java.exe** (a typical value), MOCA replaces `$JAVA` with **C:\Program Files\Java\jre7\bin\java.exe**.

2. Resolves the `-cp` argument (classpath) and replaces the `$PATH_SEPARATOR` variable with the platform-appropriate path separator:

   - For example, on Windows, MOCA resolves the argument to the following class path with a semicolon for the path separator:

     ```
     /path/to/folder/with/other/dependencies;/original/classpath/here
     ```

   - For example, on a Unix-based platform, MOCA resolves the argument to the following class path with a colon for the path separator:

     ```
     /path/to/folder/with/other/dependencies:/original/classpath/here
     ```

Note: You should use forward slashes (/) in file paths for compatibility with both Windows and Unix-based platforms. The type of slash used in file paths depends on the platform. Windows accepts backslashes (\) and forward slashes, but a Unix-based platform only accepts forward slashes.

Task Exceptions

Description

The information that follows explains the exceptions that exist for both thread-based and C-based tasks.

Environment Variables for Thread-Based Tasks

To properly support environment variables for thread-based tasks, the Task Manager creates a session variable for each defined environment variable. Session variables are not reflected in the JVM's system properties as they would be in process-based and daemon-based tasks. Session variables can be retrieved by calling `MocaContext.getSystemVariable(`*`name`*`)`.

Thread-Based Task Testing

While testing process-based and daemon-based tasks can be accomplished directly from a command line, testing thread-based tasks requires the use of the `runtask` tool. The `runtask` tool provides a way for you to run a thread-based task directly from the command line using the task's definition from the `task_definition` database table.

The `runtask` tool runs a single task within its own process using the same logic that the Task Manager uses. Consequently, it provides a runtime environment for testing that is consistent with a task running within the MOCA server.

Note: The `runtask` tool can only run a single task at a time; therefore, it does not reproduce issues when multiple tasks are running concurrently. In addition, the `runtask` tool is only applicable to thread-based tasks.

This code is the command line used for the `runtask` tool.

```
moca-dev :>runtask -h

Usage: runtask [ -h ] TaskId [task arguments]
```

Java Remote Method Invocation (RMI)

Environment variables are not be available to a thread-based task that exports an object in RMI. The environment variables are available to the code that exports the object, but any remote calls to the object in RMI do not have access to these same environment variables.

Tracing configuration is also not affected by the task settings or environment. RMI trace messages are only written to the MOCA server's log file. This is an inherent issue with how RMI itself works.

AIX Servers and C-Based Tasks

The runtime loader behaves differently on AIX servers than it does on other types of Linux and UNIX servers. As a result, C-based tasks that call `srvInitialize` currently require an extra configuration step. More specifically, the `LIBPATH` environment variable must be defined for the task, and the value must be set to `$JREDIR/lib/ppc:$JREDIR/lib/ppc/j9vm:$LIBPATH`. You can add this environment variable and value to the task on the Tasks page in the MOCA Console. If you do not define the environment variable, then C-based tasks that call `srvInitialize` will not start, and this message is written to the MOCA server's log file.

```
Can't find JVM shared library: Could not load module /usr/java6_
64/jre/lib/ppc64/j9vm/libjvm.so.
```

```
The module has an invalid magic number.
Unable to initialize JVM
srvInitialize() failed with error code: 2
```

Monitoring, Stopping and Starting Tasks

Description

You can monitor, and stop and start tasks on the Tasks page in the MOCA Console. In addition, you can monitor the session associated with a thread-based task on the Sessions page in the Console or with a Java tool such as JConsole.

Note: Currently, process-based and daemon-based tasks can only be monitored through a tool such as JConsole.

Note: Because the Console connects to a single MOCA server, when you are working in a clustered environment, you must ensure that the server to which you connect is running the role which contains the task that you want to monitor, stop or start.

For more information on the Console, see the *MOCA Console User Guide*.

Monitoring Process-Based and Daemon-Based Java Tasks with JConsole

Monitoring process-based and daemon-based Java tasks in the MOCA Console is currently not supported. The Tasks page in the MOCA Console is displayed if the task is running, but detailed information is not available. However, you can view more detailed information by connecting to the task using one of the tools that is provided with the JDK, such as `jconsole` or `jvisualvm`.

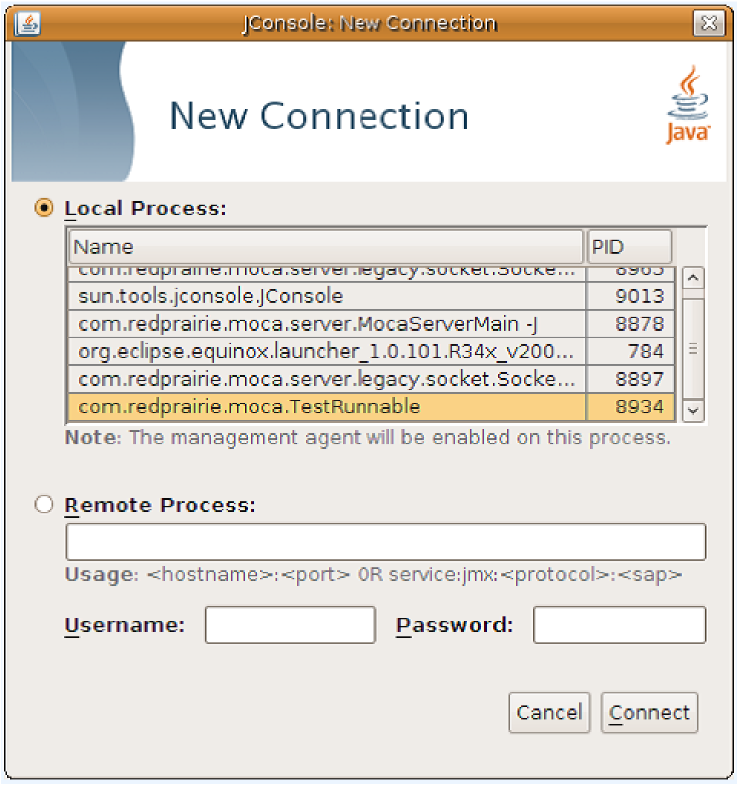
Note: Connecting to a process-based or daemon-based task when the MOCA server is running as a Windows service is not supported by JConsole or JVisualVM.

Example: Monitoring Tasks with JConsole

This example illustrates how you can use JConsole to locally connect to a Java process-based or daemon-based task for monitoring.

To monitor tasks with JConsole:

1. Start JConsole.



JConsole: New Connection

New Connection

Local Process:

Name	PID
com.redprairie.moca.server.legacy.socket.Socke...	8965
sun.tools.jconsole.JConsole	9013
com.redprairie.moca.server.MocaServerMain -J	8878
org.eclipse.equinox.launcher_1.0.101.R34x_v200...	784
com.redprairie.moca.server.legacy.socket.Socke...	8897
com.redprairie.moca.TestRunnable	8934

Note: The management agent will be enabled on this process.

Remote Process:

Usage: <hostname>:<port> OR service:jmx:<protocol>:<sap>

Username: Password:

2. On the New Connection window, double-click the class name or process ID for the task that you want to monitor.

   Note: If you want to monitor a C-based process task, then you must first locate the process ID. For a Windows server, you can use the Windows Task Manager to locate the process ID. For a UNIX server, you can use the **ps** command to locate the process ID.

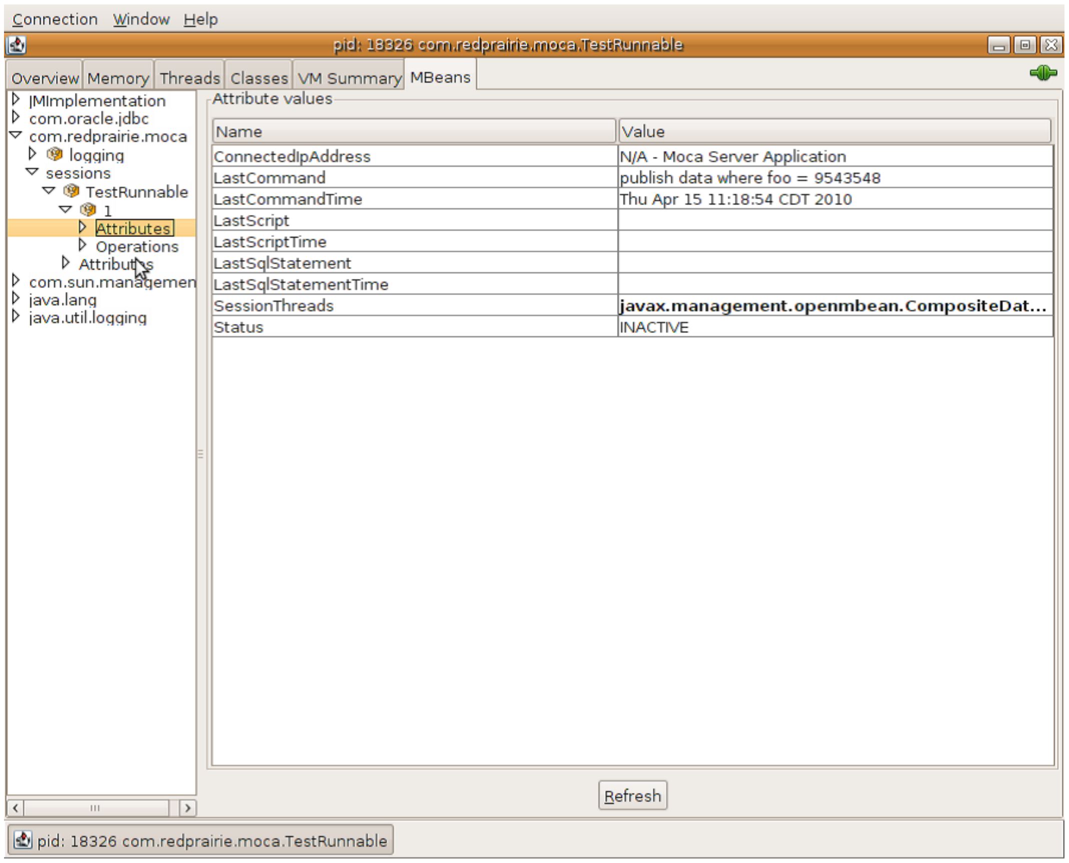
3. Select the **MBeans** tab.
4. Expand **com.redprairie.moca**.
5. Expand **sessions**.
6. Expand the name of the session for your task.

   Note: The session name is determined by what was passed to `ServerUtils.setupDaemonContext` Java method or `srvInitialize` C function.

7. Select and expand the appropriate thread ID for the session.

   Note: Typically, you have a single thread; consequently, the thread ID is **1**.

8. Under the thread ID, select **Attributes**.



Attribute values

Name	Value
ConnectedIpAddress	N/A - Moca Server Application
LastCommand	publish data where foo = 9543548
LastCommandTime	Thu Apr 15 11:18:54 CDT 2010
LastScript	
LastScriptTime	
LastSqlStatement	
LastSqlStatementTime	
SessionThreads	**javax.management.openmbean.CompositeDat...**
Status	INACTIVE

This table describes the task attribute values that appear.

Name	Description
ConnectedIpAddress	IP address that is connected for the session thread. For a task this is always N/A.
LastCommand	Last command to be executed for the task.
LastCommandTime	Date and time at which the last command was executed for the task.
LastScript	Last script that was executed for the task.
LastScriptTime	Date and time at which the last script was executed for the task.
LastSqlStatement	Last SQL statement that was executed for the task.
LastSqlStatementTime	Date and time at which the last command was executed for the task.
SessionThreads	Status of the execution relative to the session thread. These are the valid values: • `JAVA_EXECUTION` • `INACTIVE` • `IN_ENGINE`

Name	Description
	• `JAVA_EXECUTION` • `C_EXECUTION` • `COM_EXECUTION` • `SQL_EXECUTION` • `SCRIPT_EXECUTION` • `LOCAL_SYNTAX_EXECUTION`

9. Under the thread ID, select **Operations**. This table describes the task operation values that appear.

Name	Description
interrupt	Attempts to interrupt the current thread for the session. This is not the same as stopping a process-based or daemon-based task on the `Tasks` page of the MOCA Console. Stopping a process-based or daemon-based task actually kills the process.
queryDataStack	Retrieves the current MOCA data stack from the thread session.

Tasks Page

Description

You can use the Tasks page in the MOCA Console to complete all maintenance and management of the tasks used by your JDA SCE applications. For example, when creating a task you can:

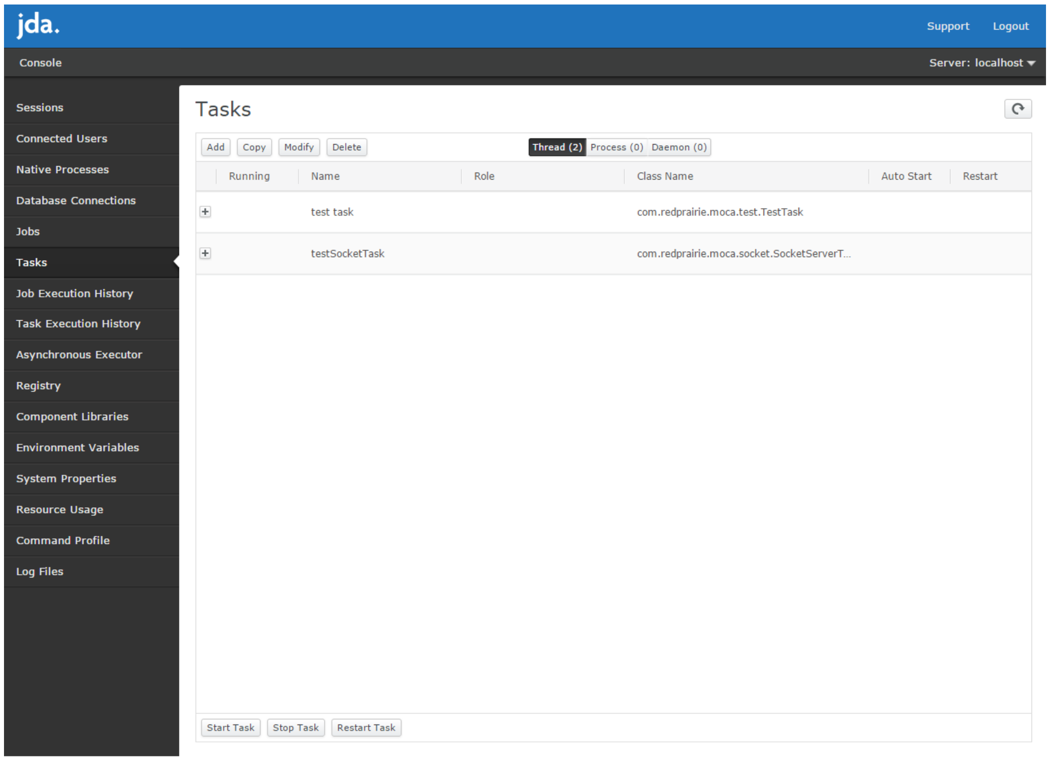
- Specify a unique ID for the task.
- Provide a descriptive name for the task.
- Indicate whether the task is:
  - Enabled to start.
  - Able to restart if terminated.
- Enter the program that executes the task (for process-based and daemon-based tasks).
- Enter the class name that executes the task (for thread-based tasks).
- Identify the directory from which the program to run process-based and daemon-based tasks executes.
- Optionally, specify a log file name and location, and whether logging is enabled.
- Optionally, specify a role ID if working in a clustered environment. Tasks and jobs with the same role always run on the same node.
- Optionally, define environment variables that are specific to the task.

After initially configuring a task, you typically do not need to update the configuration, except for a few options that are used on an as-needed basis, such as tracing to research an issue with a task.

IMPORTANT: You should maintain tasks on the Tasks page of the MOCA Console, not the Task Maintenance application that is accessed from the JDA SCE client.

Example: Tasks

This image is an example of the Tasks page in the MOCA Console.



Running	Name	Role	Class Name	Auto Start	Restart
	test task		com.redprairie.moca.test.TestTask		
	testSocketTask		com.redprairie.moca.socket.SocketServerT...		

Task Execution History Page

Description

You can use the Task Execution History page in the MOCA Console to view information about the MOCA tasks that have run. In a clustered environment, the Task Execution History page displays all of the tasks that have run on all of the nodes in the cluster. Task execution information is useful when troubleshooting tasks and helps answer the following questions:

- When did the task last run?
- On which node did the task last run?
- How long did the task take to run?
- Did the task complete successfully?

By default, the execution history is sorted by **Start Date**, with the most recent execution on the first page in the first row of the grid, and the oldest execution on the last page in the last row of the grid.

For each task execution, you can view the following information:

- **Task ID** – Unique identifier that is specified when creating the task.
- **Node URL** – URL of the node on which the task ran.
- **Status** – Value that the task returned after it finished running:
    - **Task finished executing normally** – The thread-based task completed successfully.
    - **Process Task finished with <*Return Status*> return status** – The process-based task completed and returned the status specified by <*Return Status*>.
    - **Anything else** – The task did not complete successfully.
- **Start Cause** – Entity and action that caused the task to run:
    - **USERSTART** – A user manually started the task by clicking **Start Task** on the Tasks page.
    - **RESTART** – A user manually restarted the task by clicking **Restart Task** on the Tasks page.
    - **AUTOSTART** – The MOCA server automatically started the task at initial startup or during a server restart because the task is configured to automatically start.
- **Start Date** – Date and time that the task started running.
- **End Date** – Date and time that the task stopped running.

You can complete the following tasks:

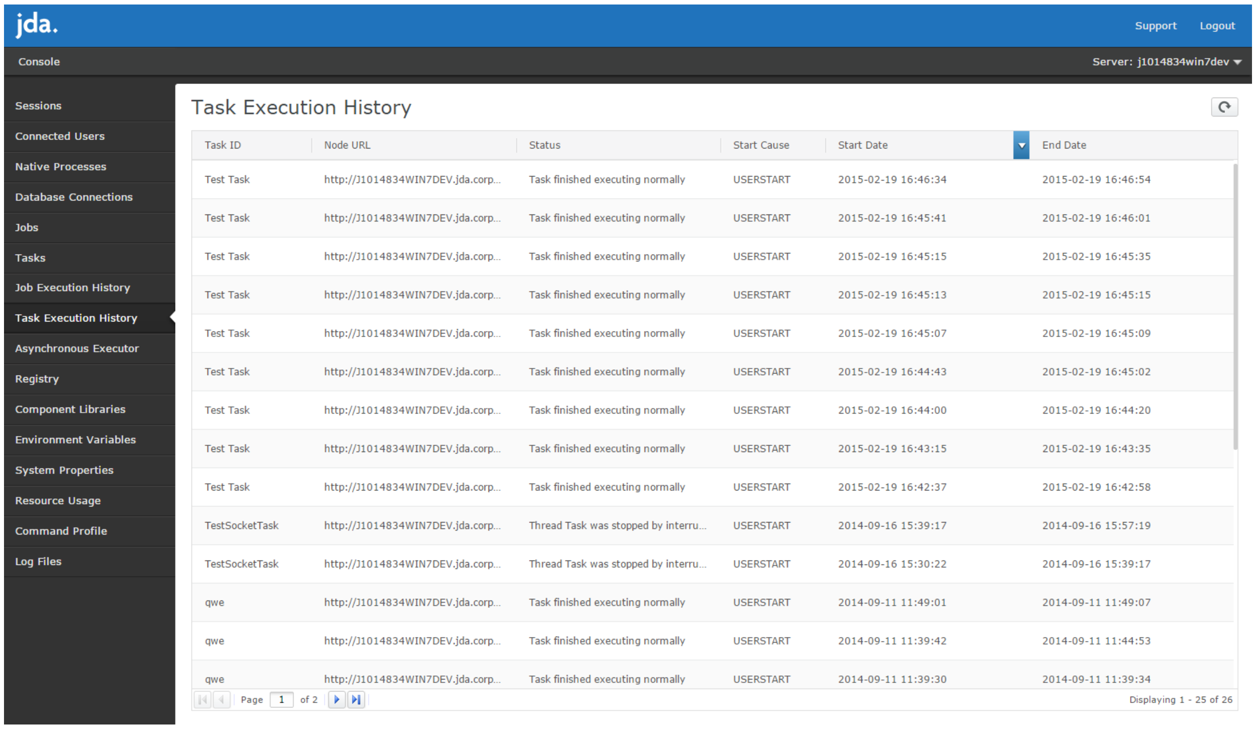
- View the task execution history on a different page:

  Note: Each page displays 25 rows except the last page, which can display fewer than 25 rows.

    - To view the first page, click ⏮.
    - To view the last page, click ⏭.
    - To view the next page, click ▶.
    - To view the previous page, click ◀.
    - To view a specific page, in the **Page** box, enter the page number.
- Sort the task execution history:
    - To sort the rows on the current page, click the column heading.
    - To sort the rows on all pages, click the column heading, and then click ⟳.

Example: Task Execution History

This image is an example of the Task Execution History page in the MOCA Console.



Task Execution History

Task ID	Node URL	Status	Start Cause	Start Date	End Date
Test Task	http://J1014834WIN7DEV.jda.corp...	Task finished executing normally	USERSTART	2015-02-19 16:46:34	2015-02-19 16:46:54
Test Task	http://J1014834WIN7DEV.jda.corp...	Task finished executing normally	USERSTART	2015-02-19 16:45:41	2015-02-19 16:46:01
Test Task	http://J1014834WIN7DEV.jda.corp...	Task finished executing normally	USERSTART	2015-02-19 16:45:15	2015-02-19 16:45:35
Test Task	http://J1014834WIN7DEV.jda.corp...	Task finished executing normally	USERSTART	2015-02-19 16:45:13	2015-02-19 16:45:15
Test Task	http://J1014834WIN7DEV.jda.corp...	Task finished executing normally	USERSTART	2015-02-19 16:45:07	2015-02-19 16:45:09
Test Task	http://J1014834WIN7DEV.jda.corp...	Task finished executing normally	USERSTART	2015-02-19 16:44:43	2015-02-19 16:45:02
Test Task	http://J1014834WIN7DEV.jda.corp...	Task finished executing normally	USERSTART	2015-02-19 16:44:00	2015-02-19 16:44:20
Test Task	http://J1014834WIN7DEV.jda.corp...	Task finished executing normally	USERSTART	2015-02-19 16:43:15	2015-02-19 16:43:35
Test Task	http://J1014834WIN7DEV.jda.corp...	Task finished executing normally	USERSTART	2015-02-19 16:42:37	2015-02-19 16:42:58
TestSocketTask	http://J1014834WIN7DEV.jda.corp...	Thread Task was stopped by interru...	USERSTART	2014-09-16 15:39:17	2014-09-16 15:57:19
TestSocketTask	http://J1014834WIN7DEV.jda.corp...	Thread Task was stopped by interru...	USERSTART	2014-09-16 15:30:22	2014-09-16 15:39:17
qwe	http://J1014834WIN7DEV.jda.corp...	Task finished executing normally	USERSTART	2014-09-11 11:49:01	2014-09-11 11:49:07
qwe	http://J1014834WIN7DEV.jda.corp...	Task finished executing normally	USERSTART	2014-09-11 11:39:42	2014-09-11 11:44:53
qwe	http://J1014834WIN7DEV.jda.corp...	Task finished executing normally	USERSTART	2014-09-11 11:39:30	2014-09-11 11:39:34

Page 1 of 2 Displaying 1 - 25 of 26

Task Database Tables

Description

Task configuration details are stored in the `task_definition` and `task_env_definition` tables.

- The `task_definition` table contains the information that describes the task, and the program or class name that executes the task.
- The `task_env_definition` table contains the unique environment variable values that you can define for a task.

Task Definition Table

This table describes the columns in the `task_definition` table.

Column	Type	Required?	Description
`task_id`	string	✓	Unique identifier for the task in all caps (for example, MTF_ SERVER). **IMPORTANT**: The task_id must be stored in the database in all caps; otherwise, appenders do not work correctly.
`role_id`	string		For clustered environments, the role that is associated with the task. Roles are used to determine the node(s) on which a task runs. For more information on role managers, see "Clustering" (on page 181).
`name`	string	✓	Name of the task.

Column	Type	Required?	Description
`task_typ`	string	✓	Type of task. These are the valid values: `T`, `P` or `D`.
`cmd_line`	string	✓	Program and arguments to execute for process-based tasks, or fully qualified Java class name and arguments to invoke for thread-based tasks.
`run_dir`	string	✓	Directory in which to start the task. For process-based tasks only.
`log_file`	string		Path to and name of the log file to which trace messages for the task are to be written.
`trace-level`	string		Level of information to be generated in a trace file when a task is running: • *****: All trace levels run, with the exception of command profiling.
`auto_start`	integer		Indicates whether the task should be started by the MOCA server. These are the valid values: `0` and `1`.
`restart`	integer		Indicates whether the task should be restarted by the MOCA server if the previous instance of the task is stopped before it is complete. These are the valid values: `0` and `1`.
`start_delay`	integer		Number of seconds that the MOCA server waits before starting the task. If a start delay is not defined, the task starts immediately.
`grp_nam`	string		Name of the group that owns the task. This is for display purposes only, and it assists in administration and debugging.

Command Line Column

The Task Manager uses the `cmd_line` column in the `task_definition` table to determine the program name for process-based tasks or the fully-qualified Java class name for thread-based tasks. In addition, the arguments passed to either the program name or class name can be provided through the `cmd_line` value.

The Task Manager instantiates thread-based tasks using the fully-qualified Java class name from the `cmd_line` that must implement the Java `Runnable` interface. If the value of `cmd_line` also includes arguments, the arguments are parsed by the Task Manager that tries to instantiate the Java class using a constructor with a `String[]` argument, if one is available.

Task Environment Definition Table

This table describes the columns in the `task_env_definition` table.

Column	Type	Required?	Description
`task_id`	string	✓	Unique identifier for the task.
`name`	string	✓	Environment variable name.

Column	Type	Required?	Description
value	string		Environment variable value. Environment variable values are expanded at runtime; consequently, a value can include a reference to other environment variables. If a value is not defined, the environment variable is cleared for the task.

This table describes the special environment variable, MOCA_CLEAN_SHUTDOWN.

Variable Name	Description
MOCA_CLEAN_ SHUTDOWN	If this environment variable is set, the Task Manager attempts to notify process-based tasks when a shutdown is requested to give the task a chance to shutdown normally. Process-based and daemon-based tasks that want to be notified by the Task Manager should continually read from their standard input and shut down when an EOF is read. This is automatically done for process-based and daemon-based tasks that call the ServerUtils.setupDaemonContext Java method or srvInitialize C function. If this environment variable is not set, the process-based or daemon-based task is forcibly terminated. This environment variable can only be used for process-based and daemon-based tasks.

IMPORTANT: If you are creating a process-based or daemon-based task in the C programming language on an AIX server, and the task calls srvInitialize(), then you must define a unique LIBPATH environment variable for the task. If you do not set this environment variable, then the task that calls srvInitialize() does not start, and you will see an error message in the log file indicating that the JVM shared library cannot be found.

Task Commands

You can use the **add task**, **remove task** and **list task** commands to query and manage the task_definition and task_env_definition database tables. You can view more information about these commands by running the **mgendoc** command. After the command runs, you can find the documentation in the %LESDIR%\docs\commands folder.

Note: For more information on configuration of tasks in a clustered environment, see "Clustering" (on page 181).

Chapter 13. Socket Server Tasks

Description

This section explains socket server tasks. To simplify the handling of socket-based interfaces in MOCA, a new mechanism was put in place that allows components to control the reading of sockets using Java objects placed on the stack by a Java thread-based task. This section describes the mechanism, shows how to configure a simple socket protocol task and how to develop more advanced socket-based tasks.

SocketServerTask Class

To set up a socket server task, create a new thread-based task in MOCA with the command line set to `com.redprairie.moca.socket.SocketServerTask`. This table describes the available command line options.

Option	Required?	Description
-p *port*	✓	Port number on which to listen.
-c "*command*"	✓	Command to run when socket activity is discovered
-P *pool-size*		Size of the thread pool to use when executing commands. If more than pool-size sockets have activity at one time, some will block while waiting for a thread to become available. Default pool size if not specified is 10.
-s		Generates a unique session for each request. Omitting this option configures requests to share a session.

Processing Command

When activity is detected by the socket server task, the processing command (given on the command line using the `-c` option) is initiated. In order for that command to read the actual data from the socket, a Java object is placed on the stack by the socket server and made available to the processing command.

This code is an example. The object is placed on the stack with the name `socket` and is of type `SocketEndpoint`.

com.redprairie.moca.socket.SocketEndpoint

```
/**
 * A socket communication endpoint.  An instance of this class is sent to
 * SocketProcessor instances when handling an incoming socket request.
 *
 * Copyright (c) 2010 RedPrairie Corporation
 * All Rights Reserved
 *
 * @author programmerA
 */
public final class SocketEndpoint {
    /**
     * Returns a textual representation of this socket.
     */
    public String getRemoteInfo() {
        return _socket.getInetAddress().getHostAddress();
    }
    /**
     * An <code>InputStream</code> for the incoming socket request.  The stream
     * returned from this method must not be closed by the caller.  To close the
     * socket, use the <code>close</code> method.
     *
     * @return an open <code>InputStream</code> for the socket.
     */
    public InputStream getInputStream() {
        return _in;
    }

    /**
     * An <code>OutputStream</code> for the socket request.  The stream returned
     * from this method must not be closed by the caller.  To close the socket,
     * use the <code>close</code> method.
     *
     * @return an open <code>OutputStream</code> for the socket.
     */
    public OutputStream getOutputStream() {
        return _out;
    }

    /**
     * Closes the socket.  After calling this method, any additional I/O on the
     * input or output streams will throw an <code>IOException</code>.
     */
    public void close() {
        try {
            _in.close();
            _out.close();
            _socket.close();
        }
        catch (IOException e) {
            // Ignore close errors
        }
    }

    . . .
}
```

A simple processing command can be written in Groovy using the dynamic nature of Groovy to use the `SocketEndpoint` object on the stack. This code is an example of a simple "echo" socket service, reading a line at a time, and echoing that line to the socket.

```
[[
  if (!socket) {
    throw new MocaArgumentException('socket')
  }
  def reader = socket.inputStream.newReader()
  def text = reader.readLine()
  socket.outputStream << text
]]
```

Note: A more complex processing command is possible, but the previous example illustrates that a simple socket service can be written in a few lines of Groovy code. However, when working in Java, you must write much more code, especially to handle IO Exceptions.

When the processing command completes, the server uses the result status to either commit or roll back any transaction created as part of the processing. In that way, it works much like the MOCA command processing engine.

Chapter 14. Service Manager

Description

This section describes how the Service Manager works, and how services can be configured and managed.

The Service Manager provides an interface to the Windows Service Control Manager and supports consistency in installing, uninstalling, starting and stopping applications running as Windows services in a JDA supply chain execution (SCE) environment.

Command Line Usage

Description

You must have your environment bootstrapped by calling `env.bat` prior to calling `servicemgr.exe` for any of the command line usages.

The following text is Service Manager command line usage:

```
servicemgr [-a <application name>]
      [-e <environment name>]
      [-s auto|manual]
      [-u <username> -p <password>]
      [-t <timeout(seconds)>]
      <action>
    where "action" must be one of the following values:
    install
    update
    uninstall
    start
    stop
    dump
```

The following table describes the available parameters for the Service Manager command line.

Parameter	Description
-a <application name>	The application name used to build the service name. If no name is specified, the name defaults to MOCA.
[-e <environment name>]	The environment name for the current environment that is configured when the application is installed. The environment name is used to build the service name.
[-s <start up type>]	How the service is configured to start with Windows. These are the value values: • **auto**: Service starts up automatically after a server restart. • **manual**: You must start the service manually after a server restart.
[-u <username> -p <password>]	Specific user name and password to run the service. If no user name or password is specificed, a system account runs the service.
[-t <timeout>]	Defines the timeout in seconds that procrun waits for the service to exit gracefully.

Parameter	Description
`<action>`	The configuration action the command line executes. These are the valid values: • **install**: Installs a service. • **update**: Updates the installed service with changes from registry. • **uninstall**: Uninstalls a service. • **start**: Starts the service using the method specified in the command line. • **stop**: Stops the service. • **dump**: Creates an env.bat file of all environment settings. **Note**: These actions are described in the sections below.

Installing a Service

This command line searches the path names defined by the `server.prod-dirs` registry key for a `services.xml` file with a matching application name. Once found, the service manager uses the `services.xml` file to install the Windows service. If an application name is not provided, "moca" is used. If <start mode> is "auto", the service is configured to automatically start. If <start mode> is "manual", the service must be started manually. If a username and password are provided, the service is configured to run under that user. If one is not provided, it runs under the Local System account.

Example: `servicemgr -a[-e] [-s] [-u-p] install`

Uninstalling a Service

This command line uninstalls the Windows service named "<application name>.%MOCA_ENVNAME%". If an application name is not provided, "moca" is used.

Example: `servicemgr -a[-e] uninstall`

Starting a Service

This command line starts the Windows service named "<application name>.%MOCA_ENVNAME%". Services can also be started using the Windows Services tool or `net start <service name>` without first bootstrapping your environment. If an application name is not provided, "moca" is used.

Example: `servicemgr -a[-e] start`

Stopping a Service

This command line stops the Windows service named "<application name>.%MOCA_ENVNAME%". Services can also be stopped using the Windows Services tool or `net stop <service name>` without first bootstrapping your environment. If an application name is not provided, "moca" is used.

Example: `servicemgr -a[-e] stop`

Dump an env.bat File

This command line creates an `env.bat` file that can be used to bootstrap an environment.

Example: `servicemgr [-e] dump`

How Service Manager Works

Description

Services are installed, uninstalled, started and stopped using the Service Manager `servicemgr.exe`. Services can also be started and stopped using the Windows Services tool from the Microsoft Management Console, or using the **net start <service name>** and **net stop <service name>** commands. The Service Manager is built on top of the Apache Commons Daemon.

Installing a Service

When the Service Manager is invoked to install a Windows service, the list of path names defined by the `server.prod-dirs` registry key is searched for files named `services.xml`. Information regarding a service, including its name, description and how it is started and stopped, is defined within one or more `services.xml` files. Higher-level `services.xml` files that define a service (or portion of a service) override lower-level files allowing application teams to supplement the definition of a service (for example, add a service dependency). The Service Manager builds a single service definition from the list of `services.xml` files and calls the Apache Common Daemon's `prunsrv.exe` tool to create the Windows service itself as well as populate some registry keys that the Apache Common Daemon uses when starting or stopping the service.

Uninstalling a Service

When the Service Manager is invoked to uninstall a Windows service, the Apache Commons Daemon's `prunsrv.exe` is called with the appropriate arguments and the service name. Uninstalling a service includes removing the Windows service itself as well as the additional registry keys that were populated for the Apache Common Daemon's use when the service was originally installed. The `services.xml` files are not used to uninstall a service.

Starting a Service

A service start request can be made using the Server Manager, Windows Services tool or the command line with the **net start** command. All three methods trigger the same process.

When a service start request is made, the Windows Service Control Manager calls Apache Commons Daemon's `prunsrv.exe` with a set of arguments that provide the context for the service to start. `prunsrv.exe` uses those arguments to look up its own configuration in the additional Windows registry keys that were populated when the service was installed and, in turn, calls a class provided by MOCA. That class bootstraps the environment for the service, looks up the definition of the service in the `services.xml` files and uses the definition to determine how to start the application associated with the service itself.

Services can be defined to start as a simple executable or a 64-bit Java process. The command line for simple executables is provided in the definition of the service in the `services.xml` file(s). Similarly, the Java class name and arguments are provided in the definition of the service for services running as 64-bit Java processes. Services that are defined to start using a 64-bit Java process must implement a method with this signature: `public void main(String args[]).`

Stopping a Service

A service stop request can be made using the Server Manager, Windows Services tool or the command line with the **net stop** command. As with a service start request, all three methods trigger the same process, which is nearly identical to how a service start request is made.

When a service stop request is made, the Windows Service Control Manager calls Apache Commons Daemon's `prunsrv.exe` with a set of arguments that provide the context for the service to stop. The `prunsrv.exe` program uses those arguments to look up its own configuration in the additional Windows registry keys that were populated when the service was installed and, in turn, calls a class provided by MOCA. That class bootstraps the environment for the service, looks up the definition of the service in the `services.xml` files and uses the definition to determine how to stop the application associated with the service itself.

Services can be defined to stop using a simple executable or a 64-bit Java process. Each of these options needs to have a method of communicating with the application that is currently running as a service. The command line for simple executables is provided in the definition of the service in the `services.xml` files. Similarly, the Java class name and arguments are provided in the definition of the service for services running as 64-bit Java processes. Services that are defined to stop using a 64-bit Java process must implement a method with this signature: `public void main(String args[]).`

Services Configuration Files

To support a level of consistency, applications running as a Windows service must provide a service definition in a `services.xml` file under the product's `%PRODDIR%`\data directory. This `services.xml` file defines information for the service, including its application name, display name, startup dependencies (if any), and rules for how to start and stop the service. The environment and application names are concatenated together to form the Windows service name. For example, an application named "moca" for the environment named "prod" is installed as the Windows service named "moca.prod". The `services.xml` files are referenced when a service is installed, started and stopped; they are not referenced when uninstalling a service. Higher-level `services.xml` files (files that exist earlier in the list of path names defined by the `server.prod-dirs` registry key) take precedence over lower-level files and can either replace or supplement lower-level definitions.

The Service Manager's application name argument is used to find a matching application name element in the `services.xml` files. The contents of the matching service from the `services.xml` files are then used to build the full set of information to pass to the Apache Commons Daemon for managing the Windows service.

This is an example service configuration file from `%MOCADIR%\data\services.xml` that defines the application "moca" for the MOCA Server.

%MOCADIR%dataservices.xml

```
<services>
  <service>
    <application>moca</application>
    <displayName>MOCA Server (%MOCA_ENVNAME%)</displayName>
    <description>The MOCA Server for %MOCA_ENVNAME%.</description>

    <start>
      <mode>exe</mode>
      <command>mocaserver</command>
    </start>

    <stop>
      <mode>java</mode>
      <class>com.redprairie.moca.server.MocaServiceFunctions</class>
      <argument>stop</argument>
    </stop>

    <dependsOn>
      <service>mssqlserver</service>
    </dependsOn>

  </service>
</services>
```

Calling the Service Manager with an application name of "moca" references the `services.xml` file because its "service" element's "application" element text value is "moca".

Because the information in the `services.xml` file is generic and does not support installing more than a single Windows service, the environment's name (taken from the `%MOCA_ENVNAME%%` environment variable) is used in conjunction with the information from the `services.xml` file to generate unique values for the Windows service itself. The `%MOCA_ENVNAME%%` is also passed as an argument to each image or class defined in the `services.xml` file when starting or stopping the service.

Using the previous example for an environment named "prod", these Windows service configurations would be installed:

- **Service Name** – moca.prod
- **Display Name** – MOCA Server (prod)
- **Description** – MOCA Server for prod

Environment variable references in `services.xml` files are expanded.

Start and Stop Modes

Description

Services can be defined to start or stop using these two separate modes: exe and java.

EXE Mode

Services defined to start or stop using the exe mode provide these elements:

- Mode element that contains a child text value of exe.
- Command element that contains a child text value that includes the full command line (including arguments) to be executed to start or stop the service.

This code defines a service that is started by executing the command "mocaserver".

%MOCADIR%dataservices.xml

```
<services>
  <service>
    .
    .
    .

    <start>
```

```
      <mode>exe</mode>
      <command>mocaserver</command>
    </start>

    .
    .
    .
  </service>
</services>
```

Additional command line arguments, which could include environment variable references, could also be provided as part of the command.

Environment variable references are expanded prior to executing the command.

JAVA Mode

Services defined to start or stop using the java mode provide a mode element with a child text value of `java` as well as a class element with a child text value that includes the fully qualified class name of a Java class that exposes a `public void main(String args[])` method. In addition, one or more argument elements can be defined, which are passed to the method. Java-mode methods are invoked by executing the command "<java> <vm arguments> <class name> <argument> ...", where the path name to Java is defined by the `java.vm` registry key value and the vm arguments are defined by the `java.vmargs` registry key value. The `java.vmargs` registry key value can be overridden by defining a `java.<application name>.vmargs` registry key value that supersedes the `java.vmargs` registry key value.

This code defines a service that is started up by executing the command "java.exe -Xmx4096m com.redprairie.moca.server.MocaServiceFunctions".

%MOCADIR%dataservices.xml

```
<services>
  <service>
    .
    .
    .

    <stop>
      <mode>java</mode>
      <class>com.redprairie.moca.server.MocaServiceFunctions</class>
      <argument>stop</argument>
    </stop>

    .
    .
    .
  </service>
</services>
```

Additional command line arguments, which could include environment variable references, could also be provided as part of the command.

JAVA32 Mode

As of release version 9.1, 32-bit support is deprecated. Configuring a Service to run in JAVA32 mode will result in an exception.

Start Dependencies

Dependencies can be defined for services using the `<dependsOn>` element. There are two potential child elements for the `<dependsOn>` element. If a `<service>` element is defined, that element's text value is taken literally as the service name, and a start dependency is configured for it. If an `<application>` element is defined, that element's text value is used and concatenated with the `%MOCA_ENVNAME%%` environment variable value to determine the service name (for example, terracotta.prod), and a start dependency is configured for it.

This code is an example that defines the Transportation Management Terracotta server as a service and places a start dependency on the MOCA Server.

%TMDIR%dataservices.xml

```
<services>
  <service>
    .
    .
    .

    <dependsOn>
      <application>moca</application>
    </dependsOn>

  </service>
</services>
```

Logging

These log files are written to by the Apache Commons Daemon when a service is installed, uninstalled, started or stopped:

- `%LESDIR%/log/<application name>.<environment name>.YYYY-MM-DD.log`
- `%LESDIR%/log/<application name>.<environment name>-stdout.YYYY-MM-DD.log`
- `%LESDIR%/log/<application name>.<environment name>-stderr.YYYY-MM-DD.log`

A short set of messages is written to `%LESDIR%/log/<application name>.<environment name>.YYYY-MM-DD.log` anytime a service is installed, uninstalled, started or stopped. This log file can be helpful to determine when an action was performed on a service but provides little additional value.

The "stdout" and "stderr" log files are written to by the Apache Commons Daemon and Service Manager. They are not written to by the application running as a service unless debugging is enabled (for more information, see "Troubleshooting" (on page 167)). Services that are correctly configured usually write nothing to either of these files except for an "initialized" log message, which can be helpful when troubleshooting a service that fails to start or stop successfully.

Troubleshooting

If a service cannot be started or does not appear to be running, use these steps to help you understand the issue.

To troubleshoot a service that cannot be started or does not appear to be running:

1. Check each of the service's log files for error messages and other clues.
2. Run the command that the Service Manager would have run but from the command line with full tracing enabled.

Chapter 15. Security

Description

This section describes security measures that should be taken since MOCA enables the user and developer to execute commands that have a significant amount of development power and some measure of risk. Due to features such as unrestricted access to SQL and Groovy scripts (effectively, arbitrary Java code submitted from the client), some security measures are important.

Sessions

Description

The MOCA server operates in these two security levels:

- **Level 0** – Level 0, also known as insecure, allows any unauthenticated request to come through, but highly restricts what those sessions can execute. SQL, Groovy or remote commands cannot be submitted by clients, and only certain commands (those with a matching security level) can be called.
- **Level 1** – Level 1, also known as secure, allows execution of any command, including SQL or Groovy. In order to execute secure commands, a session must be authenticated.

Session Authentication

To authenticate a session, the command **login user** is called. The insecure **login user** command is defined by MOCA but is overridden by higher-level command libraries that provide more capabilities than the core MOCA version. These features include:

- Database username/password matching with secure hash
- LDAP authentication
- Single sign-on
- Interactive user auditing

It is important that the authentication, which occurs after `login user` executes cleanly. Once a session has been authenticated, it is given a *session key* that associates that session with a user ID. From that point on the session can execute any command.

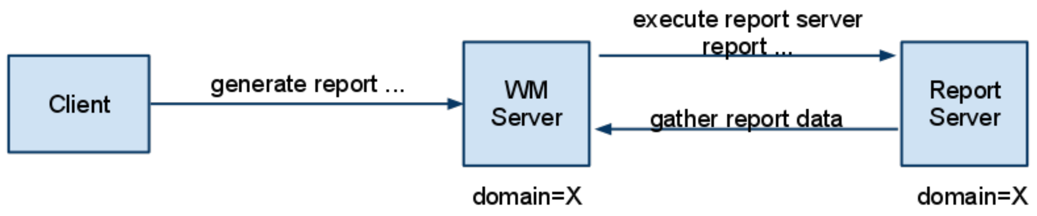
Session Keys and Domains

Session keys are always valid. There is no way to revoke a session key, so they can be very powerful. In particular, security for remote calls should be considered very carefully when designing the security model of a system. *Domains* let you restrict on the session key to certain MOCA servers. While not required (a session key with no domain is still valid), it is a good idea to have unique domains on all systems to avoid giving a secure session on one server (for example, a development server) access to another server (for example, a production server). The `security.domain` registry key can be used to configure a domain.

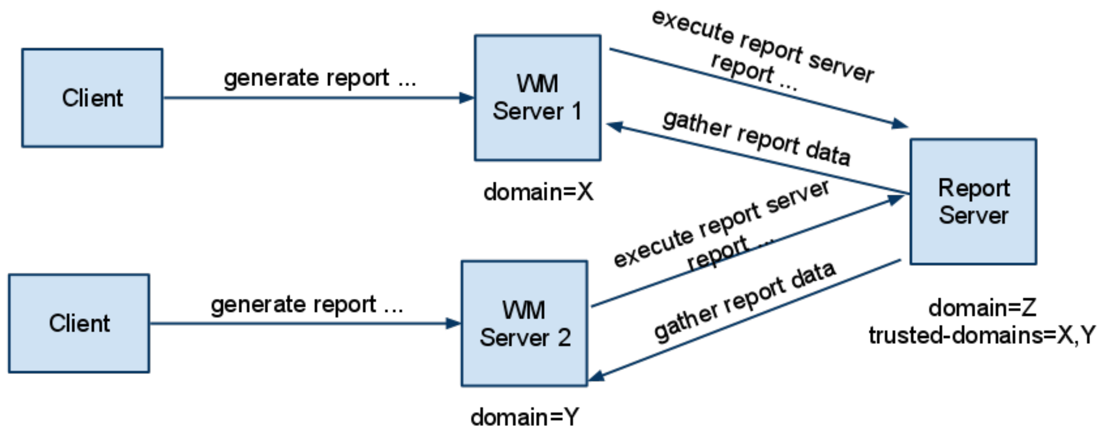
```
[SECURITY]
domain=production
```

Sometimes multi-server access is desired, such as with remote command execution. When a remote command is initiated from one MOCA server to another, the session key associated with the original client interaction is associated with the remote call. The remote server needs to accept the session key of the caller. The easiest way to do this is to configure both servers with the same domain.

To illustrate this, consider a multi-server scenario. A Warehouse Management system with a separate report server is a fairly common configuration. When running reports, the interaction is initiated with the Warehouse Management server, and a remote call is executed to the report server that calls back to the Warehouse Management server to gather the report data.



In this scenario, all MOCA servers can share the same security domain (in this case, "X") and all sessions are valid on both servers. However, in a larger configuration, a single report server often serves multiple servers. In that situation, it is desirable to separate the two Warehouse Management instances so that sessions on one server cannot execute commands on the other.



To allow them both to make remote calls to a third MOCA server, the idea of trusted domains can be used.

```
[SECURITY]
trusted-domains=X Y
```

Note: Every interaction initiated by a user is associated, end-to-end, with that user's session, so the deeply nested remote call (WM -> Report -> WM) is still tied to the original session key.

Hiding the Domain

To prevent spoofing attacks on the session key, it is possible to hide a portion of the security domain from the session key. This option reduces the possibility that a caller can generate their own session key that matches the MOCA server's trusted domain. Anything after the # character in the domain name is hidden from the caller but still used to validate the session.

```
[SECURITY]
domain=SQUID_WMPROD#xuq1zB8
trusted-domains=SQUID_LENSPROD#289fdsjkl20
```

Passwords

Hashed password generation

To generate a new hashed password, run the `mpasswd` tool. It generates a new password hash that can be copied into a MOCA registry key.

```
moca-dev :>mpasswd
Usage: mpasswd [-cdawlzhv] [-n password]
-c              Print the admin console password
-d              Print the database user's password
-a              Print the database administrator's password
-l              Print the ldap bind password
-w              Print the web container's password
-z              Print the zabbix api password
-n <password>   New password
-h              Show help
-v              Show version information
```

Administrative password

In version 2013.2, the MOCA Console was enhanced from using a single administrative password to user-based logins following the same authentication process used in other JDA supply chain execution applications. To access the MOCA Console, you can assign users and roles to the MOCA Console in Authorization Maintenance by using the distributed roles (Console Administrator and Console Read Only User) or creating your own roles. However, standard authentication does not work in the following situations:

- **Applications that do not require a database**: For example, a stand-alone Reporting instance can run without a database. When a database is not present, there is no place to store user credentials.
- **Database access issues**: When a database is configured but there are either issues accessing the database or obtaining a connection to a database, you cannot use standard authentication to access the MOCA Console. For example, if you are troubleshooting leaked database connections, it may not be possible to access a new database connection due to the connection pool being entirely consumed.

To address the situations where you cannot use standard authentication, an administrative user and password is available as an option. To enable the administrative user and password, specify values for the `admin-user` and `admin-password` keys in the SECURITY section of the MOCA registry. For example:

```
[SECURITY]
admin-user=console-admin
admin-password=|H|BNGND45FJ4CS7Q1CIGO572EE6QOCF7
```

For security purposes, the password must be hashed using mpasswd. For example:

```
mpasswd -c
```

The level of access given to the administrative user depends on the MOCA configuration:

- **Administrative user is enabled and the MOCA server has a database configured**: Logging in as the administrative user provides read-only MOCA Console access. The administrative user is used for temporary access to the MOCA Console for debugging purposes in rare cases when database access is not available; for example during a connection pool issue or network outage. This read-only access is enough to generate a support ZIP for troubleshooting the server. For full access in this configuration, the standard user-based authentication should be used rather than the administrative user.
- **Administrative user is enabled, but the MOCA server does not have a database configured**: Logging in as the administrative user provides full access to the MOCA Console since there is no other way to authenticate access.

Database Password

The username and password required for the MOCA server to establish a database connection is also maintained in the registry file. SQL Server requires a second database administrator's username and password as well, which is also maintained in the registry file as a second set of registry keys. These two passwords can be stored either in clear text or in encrypted text.

```
[DATABASE]
username=wm
password=|B|576WYsTOyUvS2oHMM/uu1yZC20IclKhk
dba-username=sa
dba-password=|B|McOSDTtfMOOBiETs/7cMj2Q7pcKuEmrE
```

The `mpasswd` tool can be used to generate a 128-bit Blowfish-encoded database user password and, if necessary, a database administrator password. The encrypted text passwords can then be copied into the `database.password` and `database.dba-password` registry keys.

```
moca-dev :>mpasswd -d
New Password: ******
Confirm New Password: ******
New database user password: |B|576WYsTOyUvS2oHMM/uu1yZC20IclKhk

moca-dev :>mpasswd -a
New Password: ******
Confirm New Password: ******
New database admin password: |B|McOSDTtfMOOBiETs/7cMj2Q7pcKuEmrE
```

LDAP Bind Password

Similar to the database password, the `mpasswd` tool can be used to hash the LDAP bind password. It can be placed in the security.ldap-bind-password registry key.

```
moca-dev :>mpasswd -l
New Password: ******
Confirm New Password: ******
New ldap bind password : |B|xHzSx4CTTKd4H41dYgNJaW+6O5wohmvU
```

Authorization

You can use a user's role options (specified in Authorization Maintenance) to control a user's access to a MOCA command or web service endpoint. See "Command Authorization" (on page 115) and "Authorization" (on page 218).

Traces and Sensitive Data

Argument Blacklist

When the Server Arguments trace level is enabled for the MOCA server or any other MOCA-based server-side application, argument names and values are written as a trace message similar to this code example.

```
26  00e7 D 15:29:58,326 (DefaultSer) [1] -------------------------------------
26  00e7 D 15:29:58,326 (DefaultSer) [1] Looking up Command: login user
26  00e7 D 15:29:58,326 (DefaultSer) [1] Executing Command: MOCAbase/login user
26  00e7 D 15:29:58,329 (DefaultSer) [2] -------------------------------------
26  00e7 D 15:29:58,329 (Argument  ) [2] Argument  usr_id=super  (STRING)
26  00e7 D 15:29:58,329 (Argument  ) [2] Argument  usr_pswd=super  (STRING)
26  00e7 D 15:29:58,329 (DefaultSer) [2] -------------------------------------
```

Displaying a user's password in clear text is not desirable. Many customers have other data that may be passed as an argument to a command that they consider sensitive and also want to hide from a trace. The `server.arg-blacklist` registry key can be used to avoid displaying an argument's value in clear text. The registry key value should be set to a comma-separated list of argument names whose values should not be displayed.

After setting `server.arg-blacklist=usr_pswd`, the sample trace above would instead look like this.

```
26  25ab D 15:39:18,133 (DefaultSer) [1] -------------------------------------
26  25ab D 15:39:18,133 (DefaultSer) [1] Looking up Command: login user
26  25ab D 15:39:18,133 (DefaultSer) [1] Executing Command: MOCAbase/login user
26  25ab D 15:39:18,135 (DefaultSer) [2] -------------------------------------
26  25ab D 15:39:18,135 (Argument  ) [2] Argument  usr_id=super  (STRING)
26  25ab D 15:39:18,135 (Argument  ) [2] Argument  usr_pswd=**** (STRING)
26  25ab D 15:39:18,138 (DefaultSer) [2]-------------------------------------
```

Executing a Command with Sensitive Data

Although an argument blacklist addresses a major security concern when Server Argument tracing is enabled, there are other trace messages that could include sensitive data. When a command is executed from either a client or server application, it is common to send the entire command and its arguments in clear text. That text is written exactly as it was provided as a trace message that looks like this.

```
1   main D 16:21:47,097 (DefaultSer) [0] Command initiated: [login
user where usr_id = 'super' and usr_pswd = 'super']
```

As with the actual argument trace messages previously discussed, this is not the type of information that you want to capture in a trace. However, because the command and its arguments are all one continuous stream of characters, MOCA cannot recognize the sensitive text within the larger stream of characters.

The solution to this problem is to pass the arguments as "bind variables" and only include references to those arguments in the command itself. Trace messages for commands executed using this method instead look like this.

```
1   main D 16:26:25,129 (DefaultSer) [0] Command initiated: [login
user where usr_id = @usr_id and usr_pswd = @usr_pswd]
```

Note: Providing an in-depth example of how to do this for Java and C from a client or server application is outside the scope of this guide; however, a description of the methods and functions for each language follows.

Java Methods

The `MocaConnection` interface provides a set of methods for client applications to pass sensitive data as arguments to a command and the `MocaContext` interface exposes a similar set of methods for server applications. For more information on these classes and methods, see the MOCA Javadoc.

```
MocaResults MocaConnection.executeCommandWithArgs(String command,
MocaArgument... args)
MocaResults MocaConnection.executeCommandWithContext(String command,
MocaArgument[] args, MocaArgument[] commandArgs)

MocaResults MocaContext.executeCommand(String command, Map<String,?>
args)
MocaResults MocaContext.executeCommand(String command,
MocaArgument... args)

MocaResults MocaContext.executeInline(String command, Map<String,?>
args)
MocaResults MocaContext.executeInline(String command, MocaArgument...
args)

MocaResults MocaContext.executeSQL(String command, Map<String,?>
args)
MocaResults MocaContext.executeSQL(String command, MocaArgument...
args)
```

C Functions

MOCA does not provide a method for C-based clients to pass sensitive data as arguments to a command. However, a set of functions are provided for server applications. Server applications that require passing sensitive data should call `srvCompileCommand` with variable references to its arguments. Those arguments can then be placed into a "bind list" by calling `sqlBuildBindList`. Finally, the compiled command and bind list can be passed to `srvInitiateCompiled` to execute the command.

```
int sqlBuildBindListFromArgs(mocaBindList **head, va_list args)
void sqlFreeBindList(mocaBindList *head)

long srvCompileCommand(char *command, SRV_COMPILED_COMMAND **exec)
long srvInitiateCompiled(SRV_COMPILED_COMMAND *compiled, mocaBindList
*bindList, RETURN_STRUCT **ret, short useContext)
```

Client Key Security

Description

MOCA includes an additional layer of security to validate whether a given client can have access to certain commands. Some commands (and whole classes of commands, such as SQL and Groovy) are highly dangerous from a security perspective. Most IT organizations want to have strict controls over the operations that specific users are able to perform, and the data that users are allowed to change. You may not want users or third-party client programs with valid logins to directly invoke commands, such as SQL, and thereby gain full access to the system. MOCA provides security for commands using a client key approach.

When invoking the `login user` command, an additional parameter can be passed. The `client_key` parameter is used to indicate to the server the type of client that is logging in. The client key must be well-formed, and must adhere to certain rules before the server trusts the client key and allows the client to log in.

Client Key Format

The `client_key` parameter to the `login user` command contains these two parts:

- **Client ID** – Name that specifies the type of client (for example, `msql` or `mcs_framework`). In addition, to prevent spoofing, the client ID can contain a private segment that is not encoded in the transmitted key. The ID takes the form `ID#private_segment`. If security of the client ID is not a concern, you do not need to include the private segment.
- **Server ID** – Each server (or cluster of servers) can be configured with a unique server ID. The server ID is used to ensure that client keys are calculated separately for each server. The server ID is returned in the result set for the core MOCA command, `get encryption information`.

Client Key Calculation

The client key is calculated using the client ID, server ID and this process:

1. Calculate the SHA-1 hash of the full client ID concatenated with the server ID, encoded as UTF-8 strings.
2. Encode the hash (128 bits) using base64. This produces a simple alphanumeric string.
3. Prepend the public part of the client ID and the forward slash (`/`) onto the string.

APIs

MOCA provides client APIs that you can use to calculate a client key.

Java

These are the Java-based APIs:

ConnectionUtils.java

```
public static String generateClientKey(String clientKey, String
serverKey);
...
public static void login(MocaConnection conn, String user, String
password, String clientKey);
...
```

Note: The clientKey is the client Id; the serverKey is the server ID.

In addition, the `MocaConnection` interface has a method (`getServerKey`) that returns the server ID. Most clients simply replace calls that used to look like this:

```
MocaConnection conn = ConnectionUtils.createConnection(url, env);
ConnectionUtils.login(user, password);
```

with calls that look like this:

```
MocaConnection conn = ConnectionUtils.createConnection(url, env);
ConnectionUtils.login(user, password, clientId);
```

C/C++

The C API (mcclib) uses the same mechanism and object structure as the Java API. This API function is available: `mccLoginWithClientKey`.

```
long MOCAEXPORT mccLoginWithClientKey(mccClientInfo *client, char
*userid, char *password, char *clientKey);
```

Note: The clientKey is the client Id.

Security Levels

To support client key security, MOCA defines several security levels to which you can assign commands. These security levels define security rings, where the outermost rings are less dangerous, more public commands, and the inner rings are more dangerous commands (for example, as needed SQL, as needed Groovy and remote calls). The list of levels is fixed and some levels have predefined meanings to the MOCA server. Other levels can be assigned to a command using the `<security-level>` tag in the command definition .xml file.

Level	Associated Command Type
OPEN	Command that can be executed by any client—even non-authenticated clients. These commands can be called from a command definition.
PUBLIC	Command that can be executed by any authenticated client. Most commands belong to this level. These commands can be called from a command definition.
PRIVATE	Command that most callers do not need to call. These commands can be called from a command definition.
ADMIN	Command that needs special privileges to run. Use this level for commands that could have an adverse impact on the server or its security (for example, accessing or modifying server configuration files). These commands can be called from a command definition.
SQL	SQL type of command. This level is automatically assigned to bracketed SQL statements. These commands are not called from a command definition.
SCRIPT	Script type of command. This level is automatically assigned to bracketed Groovy scripts and expressions. These commands are not called from a command definition.
REMOTE	Remote type of command. This level is automatically assigned to remote (and parallel and

Level	Associated Command Type
	in parallel) commands. These commands are not called from a command definition.
ALL	Command that is rarely or never called. This level is the highest most secure level. These commands are not called from a command definition.

Client Configuration

While it would be preferable to limit all clients to `PUBLIC` commands, existing practice (and code) dictates that certain clients need to be given higher level access. Eventually, we would like to have all "modern" clients connect and execute only public commands. To configure the security level for a client, edit the registry and add the client ID (including the private part) in the MOCA registry under the `[CLIENTS]` key, along with the security level to be granted. The special key `*` is a wildcard that matches all client IDs not already given other levels. The special value NONE means that a given client ID is granted no access at all.

```
[clients]
*=PUBLIC
msql=NONE
mycustomclient#SECRET000=ADMIN
```

IMPORTANT: Starting with version 2011.2.0, to ensure backward compatibility, the default is to grant `ALL` access to all incoming clients.

Known Keys

There are several known keys for internal JDA supply chain execution (SCE) products and frameworks. Configuring these keys is not necessary, as access is already coded into the MOCA server. It is possible to restrict access to these clients, but it is necessary to know the entire client ID. For most installations, it won't be necessary to restrict access to any clients that have a private part of their key.

Client Type	Key	Default Level	Notes
MCS Framework	mcsframework#...	ALL	Requires ALL to function correctly
MTF Applications	mtfframework#...	ALL	Requires ALL to function properly
WAFFLE	waffle#...	ALL	Requires ALL to function properly
RPWEB	rpweb#...	ALL	—
Web Services	webservices#...	PUBLIC	—
MLoad	mload#...	ALL	Requires ALL to function properly.
MSQL	msql	ALL	The msql key does not have a private part, so it's possible for other applications to "act like" MSQL, and be treated the same. Since msql doesn't do anything to prevent execution of commands (it executes as coded to execute), securing MSQL is typically a good first step in limiting client access. This key is used by

Client Type	Key	Default Level	Notes
			the command line msql application, as well as the Java test client.

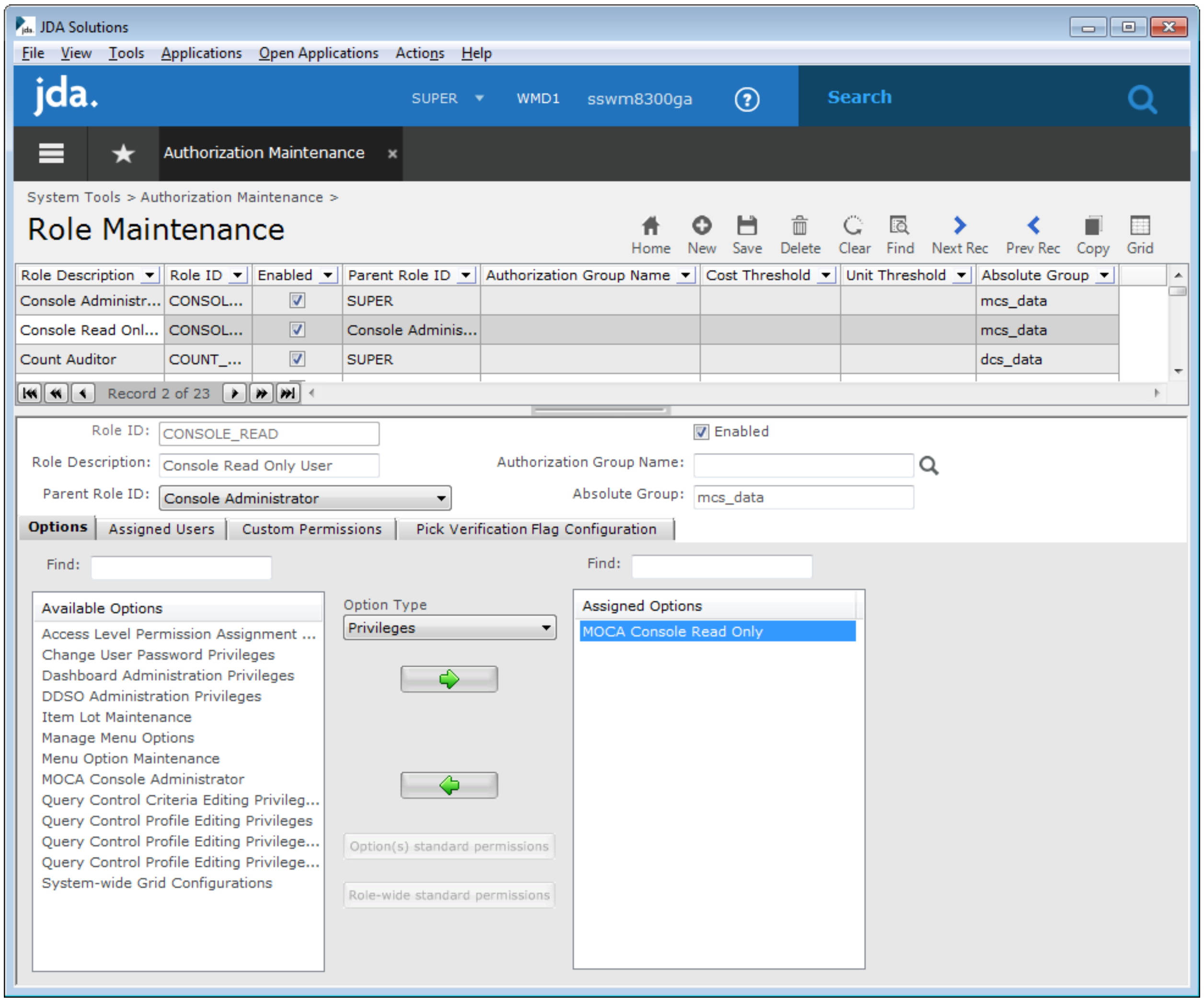
The private portion of known keys, especially for ones that must be given blanket access, does not appear in the table. To obtain those known keys for the purpose of limiting access to certain clients (for example, sites without MTF), contact the MOCA development team.

MOCA Console Privileges

Full access to the MOCA Console gives a user great control over the running system. You can limit a user's actions in the MOCA Console by specifying the user's privileges. A user with no MOCA Console privileges is not allowed to log in to the MOCA Console. For users that are allowed to log in to the MOCA Console, you can assign one of the following privilege levels:

- **MOCA Console Administrator**: Allows all actions including starting and stopping jobs, revoking another user's authentication, and restarting the server or cluster.
- **MOCA Console Read Only**: Allows the user to view information but not take actions that could affect the system. A read-only user is also allowed to download support files, logs, and command profiles.

The MOCA Console privileges can be assigned to any role in Role Maintenance. These privileges are also used to let a user connect using JMX.


JDA Solutions
File View Tools Applications Open Applications Actions Help
jda.
SUPER
WMD1
sswm8300ga
Search
Authorization Maintenance
System Tools > Authorization Maintenance >
Role Maintenance
Home
New
Save
Delete
Clear
Find
Next Rec
Prev Rec
Copy
Grid

Role Description	Role ID	Enabled	Parent Role ID	Authorization Group Name	Cost Threshold	Unit Threshold	Absolute Group
Console Administr...	CONSOL...	✓	SUPER				mcs_data
Console Read Onl...	CONSOL...	✓	Console Adminis...				mcs_data
Count Auditor	COUNT_...	✓	SUPER				dcs_data

Record 2 of 23

Role ID: CONSOLE_READ
✓ Enabled
Role Description: Console Read Only User
Authorization Group Name:
Parent Role ID: Console Administrator
Absolute Group: mcs_data

Available Options
- Access Level Permission Assignment ...
- Change User Password Privileges
- Dashboard Administration Privileges
- DDSO Administration Privileges
- Item Lot Maintenance
- Manage Menu Options
- Menu Option Maintenance
- MOCA Console Administrator
- Query Control Criteria Editing Privileg...
- Query Control Profile Editing Privileges
- Query Control Profile Editing Privilege...
- Query Control Profile Editing Privilege...
- System-wide Grid Configurations

Option Type: Privileges

Option(s) standard permissions
Role-wide standard permissions

Assigned Options
- MOCA Console Read Only

Chapter 16. Transaction Management Support

Description

This section provides information about transaction management support, including information about how a third-party tool joins a transaction and a list of supported third-party tools.

Transaction Manager

MOCA controls the commit or rollback of transactions through the use of a JTA TransactionManager implementation. The manager controls the committing of these transactions through the use of a two-phase commit, which is very similar to the MOCA remote two-phase commit process. This guarantees that these resources are not committed unless all of the resources are in a state that can be committed, called the "prepare phase".

This can be especially beneficial when used in conjunction with third-party products that then can be registered to the same transaction that is currently being controlled by MOCA. This enables the use of transactional caches, transactional JMS queues and other features.

Generally you would not need to know about the transaction manager; however, in the few cases when you want to integrate with a third-party tool to tie their transactional behavior to the MOCA commit context, you need to be able to provide a reference to the `TransactionManager` to the third-party tool (or to manually enlist a resource). It is preferred to just provide the third-party tool with the transaction manager and have them handle the registering if possible.

Accessing the Transaction Manager

You can access the transaction manager directly, or you can enlist the resources manually.

Direct Access to the Transaction Manager

Normally you would not have to interact with the `TransactionManager`, but when using third-party tools you need a way to get access to the `TransactionManager` that MOCA is using to pass it off to the third-party tool. This can be accomplished by using the class and method: `public static TransactionManager com.redprairie.moca.server.exec.TransactionManagerUtils.getManager();`.

Enlist Resources Manually

For code that provides access to the `XAResource` and has no way of providing automatic resource enlistment, you can enlist it yourself by calling it on the `MocaContext` using `public void MocaContext.enlistResource(XAResource resource) throws MocaException;`.

When you enlist that resource, it is then committed at the same relative time with the database transaction and other registered resources.

Third-Party Tools

This table describes some of the third-party tools that can be tied into the transaction manager; other third-party tools are available.

Tool	Description	Link
Infinispan	Default cache provider used by MOCA. For more information on configuring a cache to be transactional, see "Caching" (on page 197).	http://infinispan.org/documentation/
Ehcache	A commonly used cache.	http://ehcache.org/documentation
HornetQ	An open source JMS provider.	http://www.jboss.org/hornetq/docs
ActiveMQ	An open source JMS provider.	http://activemq.apache.org/index.html
SwiftMQ	An open source JMS provider.	http://www.swiftmq.com/
OpenMQ	An open source JMS provider.	http://mq.java.net/

Chapter 17. Clustering

Introduction

Description

Standard installations are not configured to run in a cluster. This section provides the information necessary to configure two or more installed environments to coexist in a cluster and troubleshoot cluster-related issues.

The 2011.1 and later MOCA-based product versions provide improved support for clustering of environments. Previous versions provided limited support for clusters. The current clustering support lets separate cluster nodes communicate with each other. This provides:

- The ability to restart an entire cluster from the console.
- Dynamic redistribution of jobs and tasks as cluster nodes are started and shut down.
- Submission of tasks to the cluster

Note: The 9.1.4.0 release features a new clustering mode that provides faster role distribution, more consistent behavior, and improved cluster stability. See the "Cluster Mode" (on page 189) section for more information.

Nomenclature

These are the key terms used in clustering:

- **Node** – A node is a single installed environment that, together with other nodes, makes up a cluster.
- **Cluster** – A cluster is a group of two or more installed environments working together that combined can provide improved performance and availability compared to a single installed environment.
- **Role Manager** – A role manager negotiates with other node's role managers for the roles that a node owns within the cluster. Role managers enable the cluster to continue to operate and provide the full set of functionality that is provided by the system as a whole, when a node joins or leaves a cluster. For example, if a node that owned role "X" leaves the cluster, another node needs to assume ownership of that role and begin managing the jobs and tasks configured with role "X".
- **Role** – When a node is started, it assumes the ownership of one or more roles within a cluster. The role(s) the node owns is used to determine what jobs and tasks are managed by the node. For example, nodes that own role "X" manages jobs and tasks configured with role "X".

Clustering Configuration

To configure clustering, you must configure the MOCA registry clustering keys. See "Registry Keys" (on page 182).

Optionally, you can also configure the following items:

- JGroups. For information, see "JGroups" (on page 183).
- Role managers. For information, see "Role Managers" (on page 188).

You can use one of the following techniques to configure JGroups:

- **Basic**: Specify the applicable JGroups-related keys in the MOCA Registry and let MOCA generate the JGroups XML configuration file. This is the recommended technique.
- **Advanced**: Create your own JGroups XML file. You can use this technique to fine tune JGroups, but at a greater risk than the basic technique. If you choose to use the advanced technique, you should start with the distributed samples that are stored in the instance's MOCADIR/samples/data subdirectory.

Registry Keys

These are some of the main registry keys in the `CLUSTER` section of the MOCA registry file are used to configure a cluster:

- **General related:**
  - `name`
  - `cookie-domain`
  - `remote-retry-limit`
  - `exclude-process-tasks`
  - `remote-retry-limit`
- **JGroups related:**
  - `jgroups-xml`
  - `jgroups-protocol`
  - `jgroups-bind-addr`
  - `jgroups-bind-interface`
  - `jgroups-bind-port`
- **Role Manager related:**
  - `role-manager`
  - `roles`
  - `exclude-roles`

Basic Configuration

If you are using the basic configuration technique and configuring the cluster with the MOCA registry, the following table lists the required registry keys.

Setting	Description
cluster.name	The name of the cluster. This must be identical for all nodes that you want to join into a cluster.
cluster.jgroups-bind-addr	The network IP address that you bind to JGroups. If the node has multiple IP addresses, an address should be chosen which is usable by other nodes on the network.

For a full list of configuration options, see the cluster section information in "Configuration" (on page 11).

Example: This example illustrates a basic MOCA registery configuration within the cluster section.

```
[CLUSTER]
name=production
jgroups-bind-addr=10.47.5.93roles=mtf
cookie-domain=acme.com
```

JGroups

Description

JGroups is an open-source toolkit that provides reliable, group-membership-based multicast communication between nodes in a cluster with the flexibility of choosing one or more communication transport protocols. MOCA uses JGroups to facilitate communication between the nodes of a cluster. By providing configurable tools for reliable multicast communication deployment, JGroups lets MOCA developers focus on application development.

Transport Protocols

JGroups can be configured to use a number of transport protocols. However, in the context of a MOCA cluster, these are the most appropriate transport protocols:

- **User Datagram Protocol (UDP)** – UDP is a unicast communication transport protocol that provides high-speed data transfer of large amounts of data, but lower reliability. It is a simpler protocol that does not provide some features like congestion control and acknowledgment of packet reception. JGroups accounts for reliability issues with application level protocols. MOCA uses this protocol by default, if the `cluster.protocol` registry key is not set.
- **Transmission Control Protocol (TCP)** – TCP is a unicast communication protocol that provides higher reliability, but a slower transfer rate. It is a more complex protocol and is primarily used in Internet communication (TCP/IP). When using TCP, initial group membership in the cluster can be discovered:
    - Dynamically using UDP multicast.
    - Dynamically using a shared database.
    - Using a static pre-defined list of hosts.

Choosing a transport protocol and discovery mechanism

For clusters where every node is on the same subnet and where the network allows multicast using UDP across subnets, it is recommended that JGroups be configured to use UDP. These are the reasons for the UDP preference:

- Using IP multicasting with UDP lets MOCA send a message to all nodes in the cluster with a single message, as opposed to TCP which requires that a single unicast message is sent to all nodes in the cluster. Because of the single message behavior, UDP provides significantly better scalability of network traffic as the cluster grows. However, this benefit of UDP decreases when you use point-to-point messages.
- It is easier to find nodes in a cluster using UDP. This is because each node listens on the same port for messages and is discovered dynamically using multicast.
- JGroups for both UDP and TCP uses application-level protocols to guarantee the order and transmission of messages, which overcomes the reliability issues of UDP versus TCP.

If multicast is not allowed on the network, then TCP should be used. When using TCP, there are several choices for the type of discovery to be used to find the nodes in the cluster. This decision typically involves choosing between dynamic (tcp-db) and static (tcp-hosts) discovery. If there is a fixed number of nodes in the cluster and the nodes are known at configuration time, then it is recommended that you define them using tcp-hosts, which defines a list of the nodes that are expected to exist in the cluster. If there is a requirement that nodes are added dynamically at run time, then tcp-db should be used, which provides dynamic discovery through the shared MOCA database. Even though this discovery protocol provides dynamic discovery, which is easier to configure, it has a slightly higher overhead than tcp-hosts since it must occasionally write to the database to determine the nodes in the cluster. Because this configuration is more expensive than basic TCP checks, it should only be used if dynamic discovery is a requirement.

Furthermore, tcp-db works best when the nodes of the cluster are launched with a staggered start. Starting all nodes at the same time may partition the cluster, resulting in the duplication of roles. The cluster is equipped with a mechanism to heal itself that runs periodically (configurable "merge3" element of jgroups.xml) but until it finishes, the cluster will be inconsistent.

The sections that follow provide the details of setting up the discovery protocols.

General Configuration

JGroups must be appropriately configured so that the nodes in a cluster have a way to discover and pass information between each other. JGroups configuration is a required step in a clustered environment. Once JGroups is configured and working, you have a minimum working MOCA cluster and can continue with other configuration steps such as setting up jobs and tasks. For information, see "Jobs" (on page 138) and "Tasks" (on page 145).

Configuring JGroups typically only requires changing the port number that JGroups use for communication between nodes. This basic configuration can be completed in the MOCA registry file, regardless of whether you are using UDP or TCP. If additional configuration is needed beyond changing the port number, this more advanced JGroups configuration must be completed in a JGroups xml-based file. Sample JGroups configuration files (such as jgroups-udp.xml and jgroups-tcp.xml) are available in %MOCADIR%\samples\data\. The MOCA registry file and/or JGroups .xml configuration file must be configured on each node in the cluster. Typically, you are able to configure the file(s) once, copy the file(s) to the other node(s) in the cluster, and then make minimal or no changes to the file(s) on the other node (s).

For more information on the JGroups MOCA registry keys, see "Configuration" (on page 11).

For more information on configuring JGroups for UDP or TCP, see the documentation on the JGroups website.

UDP-Specific Configuration

When using UDP, the multicast port number can be overridden by the Java `jgroups.udp.mcast_port` property as an alternative to modifying the configuration file. You might want to do this, for example, when you have multiple unrelated instances running on the same intranet and you do not want the unrelated instances to communicate with each other.

Configure Basic UDP Transport Protocol

If you are using UDP and only need basic configuration, complete this procedure for every node in the cluster to configure JGroups to use UDP.

Example: This example illustrates the basic MOCA registry configuration for UDP.

```
[CLUSTER]
name=cluster1
jgroups-protocol=udp
```

```
jgroups-bind-addr=10.47.5.93
jgroups-mcast-port=45588
```

To configure basic UDP transport protocol:

1. If you have not already done so, enter the required registry keys in MOCA registry file. See "Basic Configuration" (on page 182) earlier in this chapter.
2. For the **jgroups-protocol** key value, type **udp**.

   Note: By default, MOCA assumes UDP. However, it is best practice to specifically list jgroups-protocol=udp to avoid confusion and possible upgrade issues.

3. For the **jgroups-bind-port** key value, type a port number that is not in use on the current node and can be used by every node in the cluster.

   IMPORTANT: This port number must not be in use in any other cluster.

4. Repeat this procedure for any additional nodes in the cluster.

Configure Advanced UDP Transport Protocol (Optional)

If you are using UDP and need advanced configuration, complete this procedure for every node in the cluster to configure JGroups to use UDP. You only need to perform this procedure if you want to use features of JGroups that are not completely configurable in the MOCA registry file.

To configure advanced UDP transport protocol:

1. If you have not already done so, enter the required registry keys in MOCA registry file. See "Basic Configuration" (on page 182) earlier in this chapter.
2. In the cluster section, for the **jgroups-xml** key value, type **jgroups-udp.xml**.
3. Copy the sample **jgroups-udp.xml** file from **%MOCADIR%\samples\data** to **%LESDIR%\data**.
4. In the %LESDIR%\data directory, edit **jgroups-udp.xml** file.
5. Navigate to the **UDP** element and **mcast_port** attribute.
6. Type the port number that you used in the basic configuration (MOCA registry file, cluster section, jgroups-bind-port key value).
7. Make other configuration changes as desired.
8. Save and close the **jgroups-udp.xml** file.
9. Repeat this procedure for any additional nodes in the cluster.

TCP-Specific Configuration

When using TCP, if your hardware contains more than one network interface controller (NIC), you must define the `bind_addr` attribute on several JGroups protocols. Later versions of MOCA require that you specify a `cluster.bind_addr` registry key or the complementary `jgroups.bind_addr` system property. For more information on these configuration elements, see the JGroups documentation.

Configure Basic TCP Transport Protocol

If you are using TCP and only need basic configuration, complete this procedure for every node in the cluster to configure JGroups to use TCP. This procedure requires you to decide on the method of discovery:

- **tcp-mcast** - UDP multicast-based dynamic host discovery
- **tcp-hosts** - TCP socket static host address discovery
- **tcp-db** - Dynamic host discovery using a shared database

Example: This example illustrates the basic MOCA registry configuration for TCP with dynamic initial group discovery using multicast.

```
[CLUSTER]
name=cluster1
jgroups-protocol=tcp-mcast
jgroups-mcast-port=45588
```

Example: This example illustrates the basic MOCA registry configuration for TCP with static initial group discovery.

```
[CLUSTER]
name=cluster1
jgroups-protocol=tcp-hosts
jgroups-bind-port=7800
jgroups-bind-addr=10.47.5.93
jgroups-tcp-hosts=host1[7800],host2[7800]
```

Example: This example illustrates the basic MOCA registry configuration for TCP with dynamic initial group discovery using a shared database.

```
[CLUSTER]
name=cluster1
jgroups-protocol=tcp-db
jgroups-bind-addr=10.47.5.93
```

To configure basic TCP transport protocol:

1. If you have not already done so, enter the required registry keys in MOCA registry file. See "Basic Configuration" (on page 182) earlier in this chapter.
2. For the **jgroups-protocol** key value, type the corresponding value for the protocol.

Protocol	Value
TCP with dynamic initial group discovery using multicast	**tcp-mcast**
TCP with static initial group discovery	**tcp-hosts**
TCP with dynamic initial group discovery using a shared database	**tcp-db**

3. For the **jgroups-bind-port** key value, type a port number that is not in use on the current node and can be used by every node in the cluster.

IMPORTANT: This port number must not be in use in any other cluster.

4. If you set jgroups-protocol to `tcp-hosts` (static initial group discovery), then for the **jgroups-tcp-hosts** key value, enter a comma-separated list of all the addresses and JGroups bind port numbers of the nodes in the cluster in the form of `address[port]`, not including the current node being configured. For example, if you have nodes `host1`, `host2` and `host3` in your cluster, communicating on port 7800 and are currently configuring `host3`, **initial_hosts** would be `host1[7800],host2[7800]`.
5. Repeat this procedure for any additional nodes in the cluster.

Configure Advanced TCP Transport Protocol (Optional)

If you are using TCP and need advanced configuration, complete this procedure for every node in the cluster to configure JGroups to use TCP. You only need to perform this procedure if you want to use features of JGroups that are not completely configurable in the MOCA registry file.

To configure advanced TCP transport protocol:

1. If you have not already done so, enter the required registry keys in MOCA registry file. See "Basic Configuration" (on page 182) earlier in this chapter.
2. In the cluster section, for the **jgroups-xml** key value, type **jgroups-tcp.xml**.
3. Copy the sample **jgroups-tcp.xml** file from **%MOCADIR%\samples\data** to **%LESDIR%\data**.
4. In the %LESDIR%\data directory, edit **jgroups-tcp.xml** file.
5. Navigate to the **TCP** element and **bind_port** attribute.
6. Type the port number that you used in the basic configuration (MOCA registry file, cluster section, jgroups-bind-port key value).
7. For the **bind_addr** attribute, type the IP address of the node that you are configuring.
8. Navigate to the **MPING** element.
9. To use UDP multicast for dynamic group discovery, for the **mcast_port** attribute, type a port number that is not in use on the current node and can be used by every node in the cluster.

   IMPORTANT: This port number must not be in use in any other cluster.
10. If UDP multicast group discovery is not an option for any reason, then take one of the following actions:
    - Configure static hosts discovery (TCPPING):
      a. Comment out the **MPING** and **MocaJdbcPing** elements.
      b. Navigate to and uncomment the **TCPPING** element.
      c. For the **initial_hosts** attribute, enter a comma-separated list of all the addresses and JGroups bind port numbers of the nodes in the cluster in the form of `address [port]`, not including the current node being configured. For example, if you have nodes `host1`, `host2` and `host3` in your cluster, communicating on port 7800 and are currently configuring `host3`, **initial_hosts** would be `host1[7800],host2 [7800]`.

- Configure dynamic discovery using a shared database (MocaJdbcPing):
  a. Comment out the **MPING** and **TCPPING** elements.
  b. Uncomment the **MocaJdbcPing** element.

11. If the node's physical hardware has more than one NIC, configure the appropriate settings. For more information, see the JGroups documentation.
12. Make other configuration changes as desired.
13. Save and close the **jgroups-tcp.xml** file.
14. Repeat this procedure for any additional nodes in the cluster.

Role Managers

Description

Three separate role manager implementations (preferred, fixed, and dynamic) are provided to support different requirements for distributing the management of jobs and tasks across the nodes of a cluster. When a node starts and joins a cluster, the role manager on that node negotiates with the role managers running on the other nodes to determine the roles that the new node owns.

Note: Some example configurations that illustrate how jobs and tasks can be distributed across the nodes of a cluster are provided in "Cluster Mode" (on page 189).

Preferred Role Manager

A preferred role manager tries to assume ownership of roles defined in its `cluster.roles` MOCA registry key that are not already owned by another role manager. In addition, a preferred role manager assumes ownership of other roles (that are not defined in its `cluster.roles` MOCA registry key) if necessary to balance the roles across every node in the cluster. A preferred role manager is the default type of role manager and, if a `cluster.role-manager` MOCA registry key is not defined, MOCA implements a node's role manager as a preferred role manager.

A preferred role manager is similar to an active-passive manager. If multiple preferred role managers exist in a cluster, the first role manager that starts up runs the specified roles. Role managers that start up after the first role manager do not run those roles unless the first role manager node stops working.

A node that is using a preferred role manager runs the following jobs and tasks:

- Jobs and tasks configured with a role that matches a role listed in this node's `cluster.roles` MOCA registry key, if another node does not already own the role.
- Tasks configured without a role that the role manager negotiates with other nodes' role managers to own.
- Jobs configured without a role. These are considered “clustered jobs” and execute on any node each time in the cluster.
- Jobs and tasks configured with the * role. These jobs and tasks run on all nodes.

Fixed Role Manager

A fixed role manager forces its node to assume ownership of the roles defined by its `cluster.roles` MOCA registry key and immediately starts running the roles at start up. This type of role manager lets more than one node assume ownership of the same role, which lets jobs and tasks run on more than one node at a time. A fixed role manager does not take ownership of other roles to help balance jobs and tasks across the cluster.

If a fixed role manager and a dynamic role manger define the same role in the `cluster.roles` MOCA registry key, both role managers assume the role.

A node that is using a fixed role manager runs the following jobs and tasks:

- Jobs and tasks configured with a role that matches a role listed in this node’s `cluster.roles` MOCA registry key, regardless of whether other nodes are running those jobs and tasks as well.
- Jobs configured without a role. These are considered “clustered jobs” and execute on any node each time in the cluster.
- Jobs and tasks configured with the * role. These jobs and tasks run on all nodes.

Dynamic Role Manager

A dynamic role manager forcefully takes ownership of roles defined in its `cluster.roles` MOCA registry key even if another node explicitly defines the same roles in the its `cluster.roles` MOCA registry key. In addition, a dynamic role manager assumes ownership of other roles that are not defined in this node's `cluster.roles` MOCA registry key if necessary to balance the roles across every node in the cluster.

If a dynamic role manager and a preferred role manger define the same role in the `cluster.roles` MOCA registry key, the dynamic role manager always ends up "stealing" this role away from the preferred role manager.

A node that is using a dynamic role manager runs the following jobs and tasks:

- Jobs and tasks configured with a role that matches a role listed in this node's `cluster.roles` MOCA registry key. Regardless of whether other nodes are running those jobs and tasks as well.
- Tasks configured without a role that the role manager negotiates with other nodes' role managers to own.
- Jobs configured without a role. These are considered “clustered jobs” and execute on any node each time in the cluster.
- Jobs and tasks configured with the * role. These jobs and tasks run on all nodes.

Cluster Mode

In the 9.1.4.0 release, MOCA has a new clustering mode. This mode maintains the same guarantees as the old clustering mode, while adding more features and making more consistency guarantees. This mode is enabled by default and is the go-forward standard.

If any issues are observed while using this mode, it is possible to temporarily switch back to the old clustering mode by changing the `cluster.mode` MOCA registry key value to "infinispan".

Node Identification

In the new clustering mode, the following node identifiers are available:

- **Node name**: Unique identifier for the node. You can configure a node name for each node using the `cluster.node-name` MOCA registry key. The node name is visible in the MOCA Console, in support ZIP files, in MOCA logs, and in JGroups addresses.
- **Node ID**: Unique number for the node. The node in the cluster with the lowest node ID is assigned as the cluster coordinator. The node ID is usually included with the node name where a node name such as "MAIN_NODE-15" would signify that its ID is 15.

Role Distribution

Role distribution has been completely rewritten in the new clustering mode. Roles are assumed by each node's role manager much quicker than before. Role distribution is also now consistent, which means that roles are handed out to the same node each time as long as clustering configuration is not changed, no matter which node starts up first.

Process Based Tasks

The new clustering mode allows users to prevent process based tasks from joining the cluster. To prevent process based tasks from joining the cluster, set the `exclude-process-tasks` MOCA registry setting to "true".

Enabling this setting could improve cluster stability. Note that for certain installs it is not possible to enable this setting because the product may depend on clustered caches to work correctly.

Configuration Scenarios

Run a Job or Task on Any Node Scenario

In this scenario, it does not matter what node runs a job or task. The job or task needs to run somewhere in the cluster. This scenario is typical of a job or task that truncates a data store (for example, a database table) that can be accessed from any node in the cluster.

In this scenario, do not define a role for the job or task that can run on any node.

Example: This example illustrates a job_definition.csv file that, when loaded, would run the PurgeAuditLog job on any one node in the cluster. Note that the role_id entry is empty.

```
job_id,role_id,name,enabled,type,command,log_file,trace_
level,overlap,schedule,start_delay,timer,grp_nam
PurgeAuditLog,,Purge old audit logs,1,timer,"purge table where name =
'audit log'",,,0,,0,60,
```

Run a Job or Task on All Nodes Scenario

In this scenario, the job or task must run on every node in the cluster. This scenario is typical of a job or task that truncates log files from a directory.

In this scenario, configure the job or task with a role of "*" to represent any role.

Example: This example illustrates a job_definition.csv file that, when loaded, would run the PurgeLogFiles on every node in the cluster. Note that the role_id entry is *.

```
job_id,role_id,name,enabled,type,command,log_file,trace_
level,overlap,schedule,start_delay,timer,grp_nam
```

```
PurgeLogFiles,*,Purge old log files,1,timer,"purge old log files
where pathname = '$LESDIR/log'",,,0,,0,60,
```

Run a Job or Task on a Specific Node Scenario

In this scenario, the job or task must run on a specific node in the cluster. This scenario is typical of a job or task that requires access to a resource (for example, a file drop or JMS queue) that only exists on a specific node or where a client expects a specific node to provide a service (for example, an MTF or Voice server).

In this scenario, configure the job or task with a unique role. Then, on the specific node, add the role to the `cluster.roles` MOCA registry key and configure either a dynamic or fixed role manager.

Example: This example illustrates part of a MOCA registry file that configures the role manager and role MTF that is only configured for this node. Note that the roles value is MTF.

```
[CLUSTER]
name=production
cookie-domain=.acme.com
jgroups-xml=jgroups-udp.xml
role-manager=dynamic
roles=MTF
```

Example: This example illustrates a task_definition.csv file that, when loaded, would run the MTFServer task on the specific node configured for role "MTF". Note that the role_id entry is MTF.

```
task_id,role_id,name,task_typ,cmd_line,run_dir,log_file,restart,auto_start,start_
delay,grp_nam
MTFServer,MTF,MTF Server,P,java MTFServerMain,$LESDIR/log,mtfserver.log,1,1,30,
```

Run a Job or Task on Two or More Nodes Scenario

In this scenario, the job or task can run on more than one node in the cluster, but not every node. This scenario is nearly identical to the "Run a Job or Task on a Specific Node Scenario" (on page 191).

In this scenario, configure the job or task with a unique role. Then, on each specific node where you want to run the job or task, add the role to the `cluster.roles` MOCA registry key and configure either a dynamic or fixed role manager.

Frequently Asked Questions

Should I Use UDP or TCP? Which Is Better?

See "Choosing a transport protocol and discovery mechanism" (on page 183) earlier in this chapter.

How Can I Tell If I Am Connected to a Cluster?

Start the instance using the `-t` option when running `mocaserver`. This starts tracing.

Example: This example illustrates a trace message that indicates that you are connected to the cluster.

```
-------------------------------------------------------------------
GMS: address=nodename-38534, cluster=moca-cluster, physical
address=192.168.56.1:51611
-------------------------------------------------------------------
1        D 11:23:08,017 (PreferredC) Node view: [InfinispanNode
[address=nodename-38534]
```

The GMS message is very important, especially the physical address that indicates the interface address in use. The second message is stating what other nodes are known to be available in the cluster. In the example, there is only one node: nodename-38534 and it is the same address as the GMS message lists. The second message only appears when full tracing is enabled.

When a new node starts up, or if your node is joining a new cluster, you typically see multiple nodes in the node view message.

Example: This example illustrates a message that has two nodes in the cluster that are aware of each other.

```
-------------------------------------------------------------------
GMS: address=nodename1-39238, cluster=moca-cluster, physical
address=192.168.56.1:50473
-------------------------------------------------------------------
1         D 15:21:04,276 (PreferredC) Node view: [JGroupsNode
[address=nodename2-43629], JGroupsNode [address=nodename1-39238]]
```

Also, the MOCA Console displays the nodes that are in the cluster in the drop-down list. If there is no drop-down list, then you are not in a cluster.

My Nodes Are Not Finding Each Other in the Cluster. What Am I Doing Wrong?

First, get the physical address information (see "How Can I Tell If I Am Connected to a Cluster?" (on page 191)). With this information you can determine:

- From what network interface you are trying to join the cluster.
- If IPv4 or IPv6 is being used. You want to force the use of IPv4. Most network adapters try to use IPv6 first by default. However, IPv6 may not work across all nodes or even all networks. Problems have been experienced especially when trying to use IPv6 addresses when connecting nodes with different operating systems. The easiest way to work around this is to force the network adapter to use IPv4. You can do this by adding `-Djava.net.preferIPv4Stack=true` to the vmargs value in the JAVA section of the MOCA registry file.

```
[JAVA]
vmargs=-Djava.net.preferIPv4Stack=true
```

For more information, see the JGroups installation and configuration documentation website.

The Physical Address Printed in the GMS Is Not the Correct Adapter. How Do I Change It?

This can occur if a network adapter is not specifically chosen in the configuration and JGroups picked the incorrect one. To specify a network interface for JGroups to use, you must set the `bind_addr` using one of the following methods:

- Specify the `bind_addr` as a system property or MOCA registry entry. As a result, all the protocols use it, and you do not have to specify the `bind_addr` on each protocol. This method is the recommended approach.
- If you are using XML configuration, for each protocol in the stack that supports the configuration of a bind address, specify the `bind_addr` for the protocol. To determine the protocols that support the configuration of a bind address, see the JGroups documentation for your specific version.

Example: The following example illustrates the MOCA registry configuration for a specific node that is using a system property to specify the JGroups `bind_addr` (bind_addr="10.3.2.85"):

```
[JAVA]
vmargs=-Djava.net.preferIPv4Stack=true -Djgroups.bind_addr=10.3.2.85
```

Example: The following example illustrates the MOCA registry configuration for a specific node that is using a registry entry to specify the JGroups `bind_addr` (bind_addr="10.3.2.85"):

```
[CLUSTER]
jgroups-bind-addr=10.3.2.85
```

If you are using XML instead of MOCA registry configuration, perform the following tasks:

1. Make sure that the appropriate JGroups .xml configuration file (such as `jgroups-tcp.xml` or `jgroups-udp.xml`) exists in the node's `%LESDIR%\data` directory.
2. Make sure that the jgroups-xml key in the CLUSTER section of the MOCA registry file has a value that is the same as the name of the .xml file that you are using.
3. Edit the JGroups .xml configuration file and change the value of one of the following elements:
   - `bind_interface_str` – Allows for the name of the adapter itself. If the node is using a dynamically-assigned IP address, the `bind_interface_str` is preferred so that you do not have to change this setting if the IP address were to change.
   - `bind_addr` – Must be the physical IP address of the interface.

This change can be required for either TCP or UDP. For more information as to what the various arguments do within the protocols, see the JGroups list of protocols documentation website.

With the change, the physical address has changed to be 10.3.2.85 instead of the incorrect IP address previously reported by GMS.

```
-------------------------------------------------------------------
GMS: address=nodename-11030, cluster=moca-cluster, physical
address=10.3.2.85:58476
-------------------------------------------------------------------
```

Something Is Not Working as Expected. How Can I Enable JGroups Tracing?

In MOCA, only the third-party messages that are at the warning or higher level are displayed. If you want to view all the JGroups messages, you can add and/or modify the `logging.xml` that is in the `%LESDIR%\data` directory. For more information on tracing, see "Tracing and Logging" (on page 53).

Example: The following example illustrates the changes that are needed in the `logging.xml` file to enable and direct JGroups tracing to a file. The output in this example is directed to the file, `JGroupsTracing.log`.

```
<File name="JGroupsTracing"
fileName="C:\dev\trunk\env\log\JGroupsTracing.log">
    <PatternLayout>
        <Pattern>%d{ISO8601} %-5p [%c{1}] %m %n</Pattern>
    </PatternLayout>
</File>
```

```
<logger name="org.jgroups" level="ALL" additivity="false">
    <appender-ref ref="JGroupsTracing"/>
</logger>
```

Example: The following example illustrates the changes that are needed in the `logging.xml` file to enable and direct JGroups tracing to the standard output.

```
<logger name="org.jgroups" level="ALL"/>
```

UDP-Specific Issues

A common problem with UDP is that the physical machines may be on different subnets. If this is true, then UDP packets may not be receivable between the two nodes. You need to check with your network administrator to see if IP multicast across routers is allowed. For more information as well as a test application that you can run, see the JGroups advanced concepts documentation website.

TCP-Specific Issues

When UDP is unavailable, particularly due to its reliance on multicast, then TCP must be used. (There are other protocols in JGroups that are available for finding hosts.) The preferred method is to use the TCPPING protocol with a list of initial hosts to determine which nodes are in the cluster. This method discovers members with direct TCP connections that are listed in the `initial_hosts fields`. However, dynamic discovery is not available when using this method.

Example: This example illustrates the code that is needed to enable TCPPING to discover members for a cluster of size 3 (including the node on which this setting is being configured).

```
<TCPPING timeout="3000"
             initial_hosts="${jgroups.tcpping.initial_hosts:localhost
[7800],localhost[7801]}"
             port_range="1"
             num_initial_members="3"/>
```

One of the nodes listed in the initial_hosts value must be the first node to start up in a cluster and must always be available when an additional node starts up. Therefore, it is best to have as many known nodes listed in the initial_hosts attribute to prevent a cluster from becoming invalid when none of the initial hosts are started.

By default the `jgroups-tcp.xml` has a defined port of 7800 on which it listens. This is configured in the TCP protocol element in the `jgroups-tcp.xml` file. When listing nodes, you must ensure that the port for a provided node is the same port to which TCP is binding. If not, then the node cannot be asked for members in the cluster. If you change the port number, you must update the port number in the `jgroups-tcp.xml` file of all other nodes that are referencing the initial node.

Example: This example illustrates the code for the TCP bind port number.

```
<TCP bind_port="7800"
... rest omitted ...
/>
```

Thus, a configured initial nodes list that is stored on all of the nodes in the cluster could be configured with all nodes contacting on the default configured port 7800. In this case, all of the known nodes that exist in the cluster are listed.

Example: This example illustrates the code for the list of nodes in a cluster.

```
<TCPPING timeout="3000"
             initial_hosts="${jgroups.tcpping.initial_hosts:host1
[7800],host2[7800],host3[7800],
                             host4[7800],host5.domainname.com
[7800]}"
             port_range="1"
             num_initial_members="6"/>
```

Split Cluster

It is possible for a configured cluster to split into two individual clusters (also known as the split brain scenario). This situation may cause duplicate jobs and tasks to run because each node incorrectly determines that the other node is not functioning. The best way to avoid this situation is to prevent the JGroups cluster from splitting. JGroups runs a configured failure detection protocol to check for nodes that are potentially not functioning. A negative response to these checks causes the cluster to determine that a node has probably stopped functioning. It is important to configure failure detection so that it is correct for the network environment. By default, MOCA uses the following approaches to failure detection:

- **For UDP**:
    - `FD_SOCK` passive protocol: The cluster forms a ring of TCP connections with each node having a connection to its neighbor. Each node listens for an abnormal closing of this socket.
    - `FD_ALL` active protocol: Each node multicasts a heartbeat message to the cluster while keeping a table of cluster nodes and recent heartbeats.
- **For TCP**:
    - `FD_SOCK` passive protocol: The cluster forms a ring of TCP connections with each node having a connection to its neighbor. Each node listens for an abnormal closing of this socket.
    - `FD` active protocol: Each node polls its neighbor for functionality (liveness) and determines that a neighbor has stopped functioning (node death) using multicast.

There are many more protocols available from JGroups and these can be configured with the XML-based configuration.

Increasing the number of timeouts and retries on the failure detection approaches leads to a more stable cluster. You should use two failure detection approaches to increase the chance of detecting that a node has stopped functioning. Using more than one failure detection approach may cause more false positives, but those false positives should be corrected by higher level protocols; for example, `VERIFY_SUSPECT`.

If heartbeat messages are inexplicably not reaching their destination, you can enable the `msg_counts_as_heartbeat` setting on several protocols such as `FD` and `FD_ALL`. This setting also treats normal communication between nodes as heartbeat messages and resets the counter each time a message is received. This setting is more CPU intensive, especially if there is a lot of communication between nodes.

Example: The following example illustrates the XML configuration that treats normal messages as heartbeat messages.

```
<FD_ALL interval="3000" timeout="15000" msg_counts_as_
heartbeat="true"/>
```

Depending on the configured failure detection protocols, there may be an incorrect determination that a node has stopped functioning. To prevent this, if you are using XML-based configuration, make sure that you have a verification of death protocol configured. Registry-based configuration handles the verification of death protocol automatically.

Example: The following example illustrates the XML configuration that verifies node death before removing it from the cluster.

```
<!-- Double-checks whether a suspected member is really dead,
otherwise the suspicion generated from protocol below is discarded --
>
<VERIFY_SUSPECT num_msgs="3" timeout="2000"/>
```

In the list of JGroups protocols in the XML configuration file, the `VERIFY_SUSPECT` protocol should be the next protocol higher in the stack after the failure detection protocols (lower in the page in XML).

Chapter 18. Caching

Introduction

Description

This section provides information on caching provided by MOCA and Infinispan (a third-party caching provider).

MOCA uses the `ConcurrentMap` interface that is supported by an open source cache provider named Infinispan. Infinispan allows for many optional features to occur on the cache such as:

- Timing out entries after a set time.
- Timing out after inactivity of that key.
- Loading the cache.
- Persisting the cache.
- Having clustered caches.

The `ConcurrentMap` interface provides better concurrent operations (for example, `replace` and `putIfAbsent`) with caches.

For more information on Infinispan, see the JBoss Infinispan website.

Additional resources

See the following javadocs for operations available on caches:

- http://docs.oracle.com/javase/6/docs/api/java/util/concurrent/ConcurrentMap.html
- http://docs.oracle.com/javase/6/docs/api/java/util/Map.html

Configuring the Registry File

To enable a `ConcurrentMap` cache to use an Infinispan-based cache as its provider with access to Infinispan cache features, perform one of the following tasks:

- Configure the named `ConcurrentMap` cache in the registry file to use Infinispan. This is done by adding `com.redprairie.moca.cache.infinispan.InfinispanCacheProvider` as a factory for a named cache. In the following example, the named `ConcurrentMap` *test* is set up to use Infinispan.

  ```
  [CACHE]

  test.factory=com.redprairie.moca.cache.infinispan.InfinispanCacheProvider
  ```

- Programmatically pass in `InfinispanCacheProvider` as your cache provider when retrieving a cache from `CacheManager`. In the following example, the `InfinispanCacheProvider` is passed in.

  ```
  public void someMethod() {

      // Note this cache can be cast to anything generics wise, but every user of the
  named cache should be the same generics
  ```

```
    ConcurrentMap<String, String> cache = CacheManager.getCache("test", new
InfinispanCacheProvider(), null, null);
}
```

Note: When using clustered caches, to enable any type of Infinispan clustering you must also ensure that the CLUSTER section of your registry is properly configured. For more information, see "Clustering" (on page 181).

Configuring Infinispan

Description

MOCA allows you all of the flexibility that Infinispan provides with the ability to override caches as part of the layered product layout. When starting Infinispan, MOCA reads all of the `infinispan.xml` files located in all the products' `data` directories as defined by the `server.prod-dirsregistry` key value. Each product can possibly override what the lower-level product has defined if needed, although that is not recommended. MOCA only allows for default and named cache elements to be overridden.

Named Caches

Infinispan, like `ConcurrentMap`, supports having named caches. That is, you can specify settings for a named cache that do not affect other caches; for example, you can specify that one cache is clustered and another is local. Infinispan also lets you specify default cache values that are used by every other cache, unless explicitly overridden by a cache.

Note: MOCA defaults all caches to be local transactional caches. No additional items are provided, such as eviction/removal policies.

Cache Types

Infinispan includes these four types of caches:

- **Local** – Cache that is local to the node on which it was created. This could be considered an enhanced `ConcurrentMap`.
- **Distributed** – Clustered cache that distributes elements put into the cache into various nodes of the cluster. This type of cache is able to scale linearly as more nodes are brought into the cluster. Normally this type of caching is done where there are two copies of every element spread across the cluster so if one goes down there is always a copy.
- **Replicated** – Clustered cache where every node has an equivalent copy of what is in the cache. This is good for situations that require very high throughput and do not have a persistent store, such as a database.
- **Invalidating** – Cache that each node has a local independent copy of the cache. However, if an element changes, then every other node is told to invalidate or clear that element from their cache so that they can later get the updated value. This cache is the least "chatty" of clustered caches and is recommended when you have some kind of backing store such as a database to repopulate such values. This is the standard cache used when using a hibernate second-level cache. This type of cache is intended for heavy read access and does not perform as well if there is an increased rate of writes to the cache.

For more information on the four cache types, see the Infinispan clustering modes documentation website.

IMPORTANT: If a cache is configured as a distributed or replicated cache, all keys and values must all implement `Serializable`. If a cache is invalidated, then all keys must implement `Serializable`. Failure to do so causes runtime exceptions to occur when attempting to put elements into the cache.

Configure Transaction Support

You can optionally define specific caches to be transactional. The changes that are involved in the current transaction are not updated to other nodes or even other threads. Therefore, the other nodes and threads do not see the cache changes.

This can be done by adding this code to your cache definition in the `infinispan.xml` file. By default MOCA already does this for all caches unless overridden.

```
<transaction transaction-manager-
lookup="com.redprairie.moca.cache.infinispan.MocaTransactionManagerLookup" />
```

If you wish to disable transaction support, you need to define a transaction element in the `infinispan.xml` file that points to the DummyTransactionManager.

```
<transaction transaction-manager-
lookup="org.infinispan.transaction.lookup.DummyTransactionManagerLookup"/>
```

Hibernate Second-Level Cache

MOCA also supports using Infinispan as a Hibernate second-level cache provider. Everything for Hibernate needs to be configured through the `hibernate.cfg.xml` file and a specific Infinispan xml file. MOCA distributes a `hibernate.cfg.xml` file that contains the elements required to enable a second-level cache tied to Infinispan, but the elements are commented out.

This code is an example of the `hibernate.cfg.xml`.

```
<session-factory>
    ...
    <property name="hibernate.cache.use_second_level_
cache">true</property>
    <property name="hibernate.cache.use_query_cache">false</property>
    <property name="hibernate.cache.region.factory_class">

com.redprairie.moca.cache.infinispan.MocaInfinispanRegionFactory
    </property>
    ...
</session-factory>
```

All the Hibernate entity and collection caches are also stored in the infinispan.xml file located in each product's data directory as defined by the prod-dirs registry setting. With this, each layered product can override or add new Hibernate second-level cache configuration settings.

MOCA distributes `infinispan-hibernate.xml` and `infinispan-hibernate-cluster.xml` files in the `$MOCADIR/samples/data` directory. Those files also contain default values for your entity/collection/query/timestamp caches. Infinispan also lets you define Hibernate end level caches on an entity or collection basis. (These named caches are described in the examples in "Example Cache Configurations" (on page 200).)

IMPORTANT: Having a transactional cache implementation for your Hibernate second-level cache is required when your cache allows writing to entries in the cache. This is irrespective of clustering and is problematic even when using a local cache.

Example Cache Configurations

Local Transactional Infinispan ConcurrentMap

The following code is an example of the `infinispan.xml` configuration.

```
<?xml version="1.0" encoding="UTF-8"?>
<infinispan

      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

      xsi:schemaLocation="urn:infinispan:config:7.0
http://www.infinispan.org/schemas/infinispan-config-7.0.xsd"

      xmlns="urn:infinispan:config:7.0">

   <local-cache name="date_format">

   <!-- READ_COMMITTED is default so don't need to change it really if don't want -->

   </local-cache>

</infinispan>
```

Local Non-Transactional Infinispan ConcurrentMap

By default, all caches are transactional unless you set the **transaction mode** to **"NONE"**. By making a cache non-transactional, the changes performed on the cache are immediately persisted and are not undone if a rollback occurs.

The following code is an example of the `infinispan.xml` configuration.

```
<?xml version="1.0" encoding="UTF-8"?>
<infinispan

      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

      xsi:schemaLocation="urn:infinispan:config:7.0
http://www.infinispan.org/schemas/infinispan-config-7.0.xsd"

      xmlns="urn:infinispan:config:7.0">

   <local-cache name="date_format">

      <transaction mode="NONE"/>

   </local-cache>

</infinispan>
```

Distributed Transactional Infinispan ConcurrentMap

The following code is an example of the `infinispan.xml` configuration.

```
<?xml version="1.0" encoding="UTF-8"?>
<infinispan

      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="urn:infinispan:config:7.0
http://www.infinispan.org/schemas/infinispan-config-7.0.xsd"

      xmlns="urn:infinispan:config:7.0">

   <distributed-cache name="distributed-cache">

      <!-- <eviction max-entries="3000" strategy="LRU"></eviction> -->

      <!-- <expiration lifespan="43200000" max-idle="43200000" interval="600000"/> -->

      <transaction mode="NONE"/>

      <state-transfer timeout="20000"/>

   </distributed-cache>

</infinispan>
```

Replicated Infinispan ConcurrentMap with a Persistent Store with Eviction and Expiration of Elements

A replicated cache is useful when you want to have the fastest throughput at the cost of server memory. However, a replicated cache is not typically implemented unless you have a component that is called multiple times in a row as separate requests that occur on multiple nodes, and the requests do not have a backing store, such as the database. Typically, the previous scenario is associated with data that is only stored in memory, and you want other requests to have access to the data regardless of the node submitting the request. This can also be accomplished with DIST since it scales much better but may not be as fast as another implementation methodology.

This configuration provides a method (called a loader) to load and store these values to the hard disk. If too many elements are collected, they can be evicted and stored off to the file to be read back in as requested (called passivation). Passivation can be compared to how swapping works for an operating system, except passivation overflows when the maximum number of eviction elements is reached.

The following code is an example of the `infinispan.xml` configuration.

```
<?xml version="1.0" encoding="UTF-8"?>
<infinispan
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="urn:infinispan:config:7.0
http://www.infinispan.org/schemas/infinispan-config-7.0.xsd"
      xmlns="urn:infinispan:config:7.0">
   <replicated-cache name="replication" mode="ASYNC">
       <!-- You will want to match the database isolation if possible - we set up the
database as READ_COMMITTED -->
       <!-- concurrency-level="1000" - The concurrency level is an optimization for
concurrent access.  This can
                be better described in the ConcurrentHashMap java class javadoc -->
       <!-- acquire-timeout="15000" - This is how long the cache will wait to attempt to
acquire a write lock -->
       <!-- striping="false" - Lock striping is done to have a lock be more broad then a
single key.  This
            allows the map to take a smaller footprint memory wise vs throughput.  For
            best throughput then you would want to disable lock striping -->
      <locking isolation="READ_COMMITTED" concurrency-level="1000" acquire-
timeout="15000" striping="false" />
      <eviction max-entries="1000" strategy="LRU"/>
      <expiration max-idle="100000" lifespan="-1"/>
      <!-- passivation="true" - Whether to flow over to disk when an element is evicted
-->
      <!-- shared="false" - if the loaders are shared, needed when using a JDBC based
loader so only 1
              node will update the DB -->
      <!-- preload="true" - Whether the elements should be loaded at startup or only as
requested. -->
      <persistence passivation="true">
         <file-store shared="false" preload="true" fetch-state="true" purge="true"
path="${java.io.tmpdir}/replication" singleton="true"/>
      </persistence>
   </replicated-cache>
</infinispan>
```

Invalidating Infinispan ConcurrentMap

An example of a `ConcurrentMap` that uses invalidation is policies. Policies are stored in the database and are used as an invalidation cache to store only recently retrieved policies from the database. When a policy is updated, all the node representations of that value are discarded. This then requires a node to access the database again to reload the cache value. By using a `ConcurrentMap` that has a controller that accesses the database, the cache can internally perform simple gets and never has to load the cache.

The following code is an example of the `infinispan.xml` configuration.

```
<?xml version="1.0" encoding="UTF-8"?>
<infinispan
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="urn:infinispan:config:7.0
http://www.infinispan.org/schemas/infinispan-config-7.0.xsd"
      xmlns="urn:infinispan:config:7.0">
   ...
   <!-- By adding sync we are telling that when an invalidation occurs wait until all
nodes are updated.
   This could be done asynchronously if really needed by not providing this element -->
   <invalidation-cache name="Policy" mode="SYNC">
      <!-- You will want to match the database isolation if possible - we set up the
database as READ_COMMITTED -->
      <!-- concurrency-level="1000" - The concurrency level is an optimization for
concurrent access.  This can
           be better described in the ConcurrentHashMap java class javadoc -->
      <!-- acquire-timeout="15000" - This is how long the cache will wait to attempt to
acquire a write lock -->
      <!-- striping="false" - Lock striping is done to have a lock be more broad then a
single key.  This
           allows the map to take a smaller footprint memory wise vs throughput.  For
           best throughput then you would want to disable lock striping -->
      <locking isolation="READ_COMMITTED" concurrency-level="1000" acquire-
timeout="15000" striping="false" />
      <eviction max-entries="1000" strategy="LRU"/>
      <!-- Optional expiration configuration.  This defines the max idle time an entity
can be in this cache in milliseconds.
           There is also a lifespan that defines how long this entity will be in this
cache before being removed.
           With expiration the element is removed from all nodes in the cluster as
applicable.  This is not required. -->
      <expiration max-idle="100000" lifespan="-1"/>
   </invalidation-cache>
</infinispan>
```

Local Hibernate Second-Level Cache

The following code is an example of the `infinispan.xml` configuration.

```
<?xml version="1.0" encoding="UTF-8"?>
<infinispan
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="urn:infinispan:config:7.0
http://www.infinispan.org/schemas/infinispan-config-7.0.xsd"
      xmlns="urn:infinispan:config:7.0">
   <!-- Infinispan uses entity as a special named cache for entities -->
   <!-- Infinispan uses entity as a special named cache for entities -->
   <local-cache name="entity">
      <!-- You will want to match the database isolation if possible - we set up the
database as READ_COMMITTED -->
      <!-- concurrency-level="1000" - The concurrency level is an optimization for
concurrent access.  This can
         be better described in the ConcurrentHashMap java class javadoc -->
      <!-- acquire-timeout="15000" - This is how long the cache will wait to attempt to
acquire a write lock -->
      <!-- striping="false" - Lock striping is done to have a lock be more broad then a
single key.  This
         allows the map to take a smaller footprint memory wise vs throughput.  For
         best throughput then you would want to disable lock striping -->
      <locking isolation="READ_COMMITTED" concurrency-level="1000" acquire-
timeout="15000" striping="false" />
      <eviction max-entries="1000" strategy="LRU"/>
      <!-- Optional expiration configuration.  This defines the max idle time an entity
can be in this cache in milliseconds.
         There is also a lifespan that defines how long this entity will be in this cache
before being removed.
         With expiration the element is removed from all nodes in the cluster as
applicable.  This is not required. -->
      <expiration max-idle="100000" lifespan="-1"/>
   </local-cache>
</infinispan>
```

Clustered Hibernate Second-Level Cache

A clustered Hibernate second-level cache is very similar to the local cache, except you have one additional element configured for your cache.

Normally an invalidating cache is recommended for a Hibernate second-level cache. That way you limit the amount of remote calls invoked, since a remote call is only ever performed when changing an existing element, and those messages are very small. Also an invalidating cache keeps everything local in memory as requested, so you have the performance of local lookups and never take a hit of a remote lookup miss.

A replicated Hibernate second-level cache provides the greatest amount of hits but requires a much larger memory footprint as each cache contains every element. Also remote calls are much "chattier".

A distributed cache is an interesting choice as it is the most scalable second-level cache, but it is unknown if this provides much performance benefit, since depending on how many nodes there are, the cache has to do remote lookups and fail over to the database when not found. With distributed caches a properly tuned first-level cache for Infinispan is strongly suggested.

Therefore, an invalidated cache is typically chosen and possibly distributed. As described earlier, an invalidating cache is usually used when you already have a backing store (such as a database) and is used just as a local lookup to avoid hitting the database.

This code is an example of the `infinispan.xml` configuration.

```
<?xml version="1.0" encoding="UTF-8"?>
<infinispan
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="urn:infinispan:config:6.0
http://www.infinispan.org/schemas/infinispan-config-6.0.xsd"
      xmlns="urn:infinispan:config:6.0">
   <!-- Infinispan uses entity as a special named cache for entities -->
   <namedCache name="entity">
      <clustering mode="invalidation">
        <!-- State retrieval can be used to warm a cache upon startup.  This is only used
with invalidating
             and replicated.  In this case our invalidated cache doesn't need to be
warmed up -->
         <stateRetrieval fetchInMemoryState="false" timeout="20000"/>
        <!-- By providing sync you are making sure that updates are done synchronously
with the rest of
             the cluster nodes.  This guarantees that when you invalidate something that
the other nodes
             will be in the same state when your call returns.  The timeout is in
milliseconds and
             will throw an exception if that timeout exceeds -->
         <sync replTimeout="20000"/>
      </clustering>
       <locking isolationLevel="READ_COMMITTED" concurrencyLevel="1000"
lockAcquisitionTimeout="15000" useLockStriping="false" />
      <eviction wakeUpInterval="5000" maxEntries="1000" strategy="LRU"/>
      <expiration maxIdle="100000" lifespan="-1"/>
   </namedCache>
</infinispan>
```

Local Hibernate Second-Level Cache on Entity Basis

Infinispan lets you configure a cache to have special settings on an entity- or collection-basis when used as a Hibernate second-level cache provider.

You first need to set a property in your `hibernate.cfg.xml` to tell what named cache in Infinispan to use for a specific entity or collection.

This code is an example of the `hibernate.cfg.xml` configuration.

```
<hibernate-configuration>
    ...
    <session-factory>
        ...
        <property name="hibernate.cache.infinispan.com.acme.Person.cfg">
           person-entity
        </property>
        <property name="hibernate.cache.infinispan.com.acme.Person.addresses.cfg">
           addresses-collection
        </property>
        ...
    </hibernate-configuration>
    ...
</session-factory>
```

In the previous code example, the `com.acme.Person` entity is mapped to the Infinispan named cache `person-entity`, and the `com.acme.Person.addresses` collection is mapped to the Infinispan named cache `addresses-collection`.

Then you can configure your caches for those entity/collections however you want. You can have this cache be just local or also have it be a clustered cache as shown in previous examples.

This code is an example of the `infinispan.xml` configuration.

```
<?xml version="1.0" encoding="UTF-8"?>
<infinispan
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="urn:infinispan:config:7.0
http://www.infinispan.org/schemas/infinispan-config-7.0.xsd"
      xmlns="urn:infinispan:config:7.0">
   <distributed-cache name="person-entity">
      .. configuration here ..
   </distributed-cache>
   <distributed-cache name="addresses-collection">
      .. configuration here ..
   </distributed-cache>
</infinispan>
```

Infinispan Least Recently Used and Collection Based Caches

Starting with version 2013.2, MOCA no longer supports its own implementation of a Least Recently Used (LRU) or collection based cache. Instead, MOCA uses the Infinispan caching configuration options to create the same type of caches. See the infinispan documentation mentioned earlier in this chapter for additional configurations not covered in this chapter.

LRU Cache

To have a LRU Infinispan cache, the configuration must contain an eviction policy. The following XML code is an example of the configuration for eviction using LRU.

```
<eviction max-entries="1000" strategy="LRU"/>
```

The configuration allows only 100 cache elements and evicts them based on the LRU algorithm.

The following XML code is an example of a complete configuration.

```
<local-cache name="userCache">
    <eviction max-entries="1000" strategy="LRU"/>
    <transaction mode="NONE"/>
</local-cache>
```

Collection Based Cache

The legacy collection based cache supported the expiration of elements by a timeout. This can be done with the Infinispan expiration policy.

```
<expiration lifespan="1000"/>
```

The configuration has no upper limit as to how many items can be stored, but removes entries after 1000 milliseconds from creation or last retrieval.

The following XML code is an example of a complete configuration:

```
<local-cache name="userCache">
    <expiration lifespan="1000"/>
    <transaction mode="NONE"/>
</namedCache>
```

Infinispan CacheStores (CacheControllers)

Starting with version 2013.2, MOCA uses the Infinispan `CacheStore` instead of the legacy MOCA `CacheController`.

If you are working with a version of MOCA that is older than 2013.2, you have one of the following options:

- Convert your `CacheController` to a `CacheStore`.
- Utilize `MocaCacheStore` which wraps a `CacheController`.

CacheStore

It is suggested that the long term solution is to convert each `CacheController` to an appropriate `CacheStore`:

1. Extend `AbstractMocaCacheStore` instead of `CacheController`. For more information, see the Infinispan AbstractCacheStore web page.
2. Implement the appropriate methods, with the following methods being the most important:
    - `load(Object key)`
    - `loadAll()`
    - `onStart()`
3. Specify the loader in the infinispan.xml.

`OnStart` is called during the startup of the `CacheStore`. If any MOCA-related actions are performed in `onStart()`, MOCA handles the cleanup of the `MocaContext` for you (unlike the `start()` method from `AbstractCacheStore`). JDA supply chain execution applications are responsible for any necessary commit or rollback actions, although it is suggested that you do not perform commits or rollbacks during the startup of the `CacheStore`.

The following example specifies the `loader` class which must have a no arg constructor to work properly.

```
<infinispan>
    <local-cache name="date_format">
        <persistence passivation="false">
            <store
class="com.redprairie.moca.server.expression.function.DateFormatCacheStore"
                    fetch-state="false" read-only="false" purge="false" shared="false"
preload="false">
                <async enabled="true"/>
            </store>
        </persistence>
        <transaction transaction-manager-
lookup="com.redprairie.moca.cache.infinispan.MocaTransactionManagerLookup"/>
    </local-cache>
</infinispan>
```

The following code snippet is an example of a `CacheStore`.

```
public class SampleStringCacheStore extends AbstractMocaCacheStore<Object, String> {
    private final ConcurrentMap<Object, String> _map;
    public SampleStringCacheStore() {
        _map = new ConcurrentHashMap<Object, String>();
        // Load some default values here
    }
    // @see org.infinispan.persistence.spi.CacheWriter#write
(org.infinispan.marshall.core.MarshalledEntry)
    @Override
    public void write(MarshalledEntry<?, ? extends String> entry) {
        _map.put(entry.getKey(), entry.getValue());
    }
    // @see org.infinispan.persistence.spi.CacheWriter#delete(java.lang.Object)
    @Override
    public boolean delete(Object key) {
        String old = _map.remove(key);
        return (old != null);
    }
    // @see org.infinispan.persistence.spi.CacheLoader#load(java.lang.Object)
    @Override
    public MarshalledEntry<Object, String> load(Object key) {
        String value = _map.get(key);
        MarshalledEntry<Object, String> ice = new MarshalledEntryImpl<Object, String>(
                key, value, null, null);
        return ice;
    }
    // @see org.infinispan.persistence.spi.CacheLoader#contains(java.lang.Object)
    @Override
    public boolean contains(Object key) {
        return _map.containsKey(key);
    }
    // @see com.redprairie.moca.cache.infinispan.loaders.AbstractMocaCacheStore#onStart()
    @Override
    protected void onStart() {
    }
    // @see org.infinispan.persistence.spi.CacheWriter#init
(org.infinispan.persistence.spi.InitializationContext)
    @Override
    public void init(InitializationContext ctx) {
    }
    // @see org.infinispan.lifecycle.Lifecycle#stop()
    @Override
    public void stop() {
    }
}
```

Wrapping CacheControllers

The other available option is to use the MOCA base implementation that wraps the current `CacheController`. This also requires that the `CacheController` has a no arg constructor. To implement, define the class loader in the `infinispan.xml` file.

The following code is an example of setting the loader to the `MocaCacheStore` and specifying the legacy `CacheController` as a property.

```
<infinispan>
    <local-cache name="moca-test-cache">
```

```
        <persistence passivation="false" >
            <store class="com.redprairie.moca.cache.infinispan.loaders.MocaCacheStore"
fetch-state="false"
                    read-only="false" purge="false" shared="false" preload="false">
                <async enabled="true"/>
                <properties>
                    <property name="loaderClass"
value="com.redprairie.moca.cache.MocaTestController"/>
                </properties>
            </store>
        </persistence>
        <transaction transaction-manager-
lookup="com.redprairie.moca.cache.infinispan.MocaTransactionManagerLookup"/>
    </local-cache>
</infinispan>
```

Infinispan Resources

For access to all of the Infinispan documentation, see the Infinispan documentation home page. For more information on all of the available elements that can be used when configuring a cache, see the Infinispan configuration options website.

Chapter 19. Asynchronous Execution

Introduction

Description

This section describes asynchronous execution and provides information on how to:

- Obtain the asynchronous executor.
- Submit callable tasks for pull notification.
- Submit callable tasks for push notification.
- Perform asynchronous execution in a clustered environment.

Terminology

These are terms that are used in this section:

- **Callable task** – Java object that implements the Callable interface.
- **Submitter** – Code that submits a callable task.
- **Runner** – Code that actually executes the callable task.

Asynchronous Description

MOCA supports executing any code asynchronously so that the code is executed outside of the scope of the submitter. The submitter can either continue processing without waiting for its submitted callable task to complete, or wait for the callable task to complete and return its result. This lets submitters execute multiple callable tasks in parallel from a single request, and then retrieve all the results from the request and collate the data before returning.

Even though the code submitted to the asynchronous executor is executed in a different scope, tracing settings of the submitter are inherited. That means that if a callable is submitted to the executor while the caller has tracing enabled, such as with session tracing, then the asynchronous code is traced in the same way as the caller until the completion of the callable. This is true even if the submitter disabled his tracing after submitting the callable.

MOCA automatically commits or rolls back the callable task's transaction after the callable task is invoked. Therefore, asynchronous executor must not be used when the callable task's transaction behavior depends on the submitter. Only callable tasks that are either non-transactional or are ready to be committed should be implemented as an `AsynchronousExecutor` task. A commit occurs if a value is returned from the method, even if it is null, whereas a rollback occurs if any exception is thrown from the callable task.

Asynchronous Executor Usage

How to Retrieve the AsynchronousExecutor Instance

This code provides an example of how the `AsynchronousExecutor` can be retrieved by invoking the `asyncExecutor` method on the `MocaUtils` class.

```
AsynchronousExecutor async = MocaUtils.asyncExecutor();
```

IMPORTANT: The `AsynchronousExecutor` interface is only available in the mocaserver process. Therefore, process mode tasks do not have access and the `asyncExecutor` method returns a null.

Single Submission Future Approach

The `Future` approach gives the submitter the largest amount of control over the execution of a callable task. This approach lets the submitter cancel the callable, if needed. The Future class is a standard Java class that allows for timed and full blocking on an execution to complete and returns when the callable task has completed with either returning or throwing an exception.

This code is an example of how to use the `Future` approach.

```
AsynchronousExecutor async = MocaUtils.asyncExecutor();

Callable<MocaResult> callable = MocaUtils.mocaCommandCallable("fullfill order",
    new MocaArgument("ordnum", "TEST-ORD"));

Future<MocaResults> futureRes = async.submitAsynchrounously(callable);

try {
    MocaResults res = futureRes.get();
}
catch (InterruptedException e) {
    futureRes.cancel(true);
    throw new MocaInterruptedException(e);
}
catch (ExecutionException e) {
    e.getCause().printStackTrace();
}
```

Multiple Submission CompletionService Approach

The `CompletionService` approach lets multiple callable tasks be executed in a single method call. The completion service then permits the submitter to wait for any of the callable tasks to finish on a single thread. The submitter can retrieve the `Future` objects as they finish. This allows for immediate response when a callable task is finished by only using one thread, whereas the normal `Future` approach would require a thread for each submitted callable task. This can reduce resources required and potentially improve throughput since results can be processed as they are completed instead of in submission order. This method does not let a submitter cancel a callable task.

This code is an example of how to use the `CompletionService` approach.

```
AsynchronousExecutor async = MocaUtils.asyncExecutor();

Callable<MocaResult> callable = MocaUtils.mocaCommandCallable("fullfill order",
    new MocaArgument("ordnum", "TEST-ORD"));

Callable<MocaResult> callable2 = MocaUtils.mocaCommandCallable("fullfill order",
    new MocaArgument("ordnum", "TEST-ORD2"));

CompletionService<MocaResult> completion = async.executeGroupAsynchrounously(callable,
callable2);


try {
    MocaResults res = completion.take().get();
    MocaResults res2 = completion.take().get();
}
catch (InterruptedException e) {
    throw new MocaInterruptedException(e);
}
catch (ExecutionException e) {
    e.getCause().printStackTrace();
}
```

Single Submission AsynchronousExecutorCallback Approach

The `AsynchronousExecutorCallback` approach is the most submitter-passive approach for execution; it requires the least amount of interaction with the submitter. This approach is similar to a push notification in that the framework calls a method automatically when the submitter is finished. This uses the `executeAsynchronously` method that takes a second argument that implements `AsynchronousExecutorCallback`. This interface describes a single method that is called when the execution is complete. Then the method is called with the first argument set to the callback that was provided, and the second argument set to the Future object that contains the now finished values (a successfully returned value or thrown exception). The callback occurs in a separate transaction from the callable task. If no exception is thrown, the new transaction is committed; otherwise, it is rolled back.

This code is an example of how to use the `AsynchronousExecutorCallback` approach.

```
private static class ResultCallback implements AsynchronousExecutorCallback<MocaResults>
{
    public void done(Callable<MocaResults> callable, Future<MocaResults> future) throws
Exception {
        try {
            // This is guaranteed to return immediately
            MocaResults res = future.get();
            // Do something with result
        }
        catch (InterruptedException e) {
            // This cannot happen, but be safe anyways
            throw new MocaInterruptedException(e);
        }
        catch (ExecutionException e) {
            e.getCause().printStackTrace();
        }
    }
}
AsynchronousExecutor async = MocaUtils.asyncExecutor();

Callable<MocaResult> callable = MocaUtils.mocaCommandCallable("fullfill order",
    new MocaArgument("ordnum", "TEST-ORD"));

// The callback handles the results
async.submitAsynchrounously(callable, new ResultCallback());
```

Example of Asynchronous Execution for Each Row Returned in a List

This is an example of a simple component that calls a list command and then submits an asynchronous execution for each of the returned rows. The code stores the results and exceptions in the result set, but each component can handle errors on an independent basis. Using the `Future` method lets the submitter decide when to check for results and interrupt one or more of the executing callable tasks, if needed.

```
public MocaResults fullfillOrders(MocaContext moca) throws MocaException {

        MocaResults res = moca.executeCommand("list orders");

        AsynchronousExecutor async = MocaUtils.asyncExecutor();
        List<Future<MocaResults>> results = new ArrayList<Future<MocaResults>>(
                res.getRowCount());

        while (res.next()) {
            String order = res.getString("ordnum");
            results.add(async.executeAsynchronously(new FullfillOrder(order)));
        }

        EditableResults retRes = moca.newResults();
        retRes.addColumn("returned", MocaType.STRING);

        boolean errored = false;

        for (Future<MocaResults> future : results) {
            retRes.addRow();
            try {
                MocaResults futureRes = future.get();
                String returned = futureRes.next() ? futureRes.getString(
                    "returned") : "nothing";
                retRes.setStringValue("returned", returned);
            }
            catch (InterruptedException e) {
                // The least we should do is rethrow
                throw new MocaInterruptedException(e);
            }
            catch (ExecutionException e) {
                errored = true;
                retRes.setStringValue("returned", e.getCause().toString());
            }
        }

        if (errored) {
            MocaException excp = new MocaException(12345);
            excp.setResults(retRes);
            throw excp;
        }
        else {
            return retRes;
        }
    }
    private static class FullfillOrder implements Callable<MocaResults> {

        public FullfillOrder(String orderNumber) {
            _orderNumber = orderNumber;
        }

        // @see java.util.concurrent.Callable#call()
        @Override
        public MocaResults call() throws Exception {
            return MocaUtils.currentContext().executeCommand("fullfill order",
                new MocaArgument("ordnum", _orderNumber));
        }

        private final String _orderNumber;
    }
```

Clustered Asynchronous Executors

Description

MOCA supports executing asynchronous callable tasks across the cluster by using the clustered asynchronous executor.

```
AsynchronousExecutor async = MocaUtils.clusterAsyncExecutor();
```

Prerequisites

To use the clustered asynchronous executor, the MOCA registry must have a cluster section with a cluster name specified. For more information, see "Configuration" (on page 11).

Where to Use

The clustered asynchronous executor can be used in server mode and process mode tasks.

How to Use

The clustered asynchronous executor requires that all submitted callable tasks implement the `Serializable` or `Streamable` interface. This requirement enables the `Callable` object to be transmitted to and executed on another node. MOCA provides the convenience method, `mocaCommandCallable`, that submits a serializable MOCA command callable task to the clustered asynchronous executor so the request can be called on any node in the cluster. The benefit of using the method on clustered asynchronous executors is greater scalability.

The following code is an example of using the clustered asynchronous executor and is similar to the code used for the regular asynchronous executor:

```
AsynchronousExecutor async = MocaUtils.clusterAsyncExecutor();
async.executeAsynchronously(MocaUtils.mocaCommandCallable("list users
where usr_id = @usr_id", new MocaArgument("usr_id", "SUPER")));
```

When calling commands on the cluster, the command cannot be dependent on any attribute that is unique to a specific node in the cluster. For example, if you write a command (`list files for directory`) that lists the files for a specific directory, the command is dependent on the file system which can be different for each node in the cluster (different directories may exist on each node). Therefore, `list files for directory` is not eligible for being called on the cluster.

MOCA Registry

When using the clustered asynchronous executor, there are two MOCA registry values that can be tuned.

```
[CLUSTER]
async-runners = 10
async-submit-cap = 10000
```

The async-runners value specifies the number of thread runners that the server starts to handle the incoming asynchronous execution requests.

The async-submit-cap value specifies the maximum number of asynchronous execution requests that a node can handle at one time. Any callable task submitted beyond this limit blocks until a node can handle another callable task.

Since each environment handles a load differently, monitor these values using the Cluster Async Executor page in the MOCA Console and then tune the values accordingly.

Monitoring

The Cluster Async Executor page in the MOCA Console lets you monitor the activity of the clustered asynchronous executors.

The Cluster Async Executor page lets you:

- View the clustered asynchronous executors that are running on each node by selecting a node from the Server drop-down list.
- View the current runner threads that are running with the callable task that is running (if applicable) in the Current Runner Tasks pane. If the runner isn't running a callable task, it displays Idle for the Task Name.
- View the callable tasks that are queued (running and waiting) in the Current Requests pane. If the callable task is running, you can also view the node that is running the callable task which could be different from the node that submitted the callable task.
- Click the node link in the Current Requests pane to change the Cluster Async Executor page to view the node.
- Click the Task Thread ID link to view to open the Sessions page and view additional details about the runner.
- Add or remove a runner thread by selecting it and then selecting the desired action from the Actions drop-down list. This lets you adjust the node's asynchronous load; for example, when the current requests list is too large (too few threads in the current runner tasks list) or too light (too many threads in the current runner tasks list).

Chapter 20. Web Services

Description

MOCA provides support for hosting web services using Spring MVC. For more information, see the Spring Web MVC framework documentation website. This section provides information on creating web services using the functionality that MOCA provides through Spring MVC and Jetty. By using Jetty, an external web hosting solution (such as Apache Tomcat) is not needed to host the web services. Instead, web archives (WARs) can be deployed or hot swapped with Jetty. For more information, see the Eclipse Jetty website.

Note: For more information on the Configurable Web Services framework, see "Configurable Web Services" (on page 229).

Web Service Documentation

This section describes how developers and users can view the current web service documentation, which is generated using the Swagger library. See the Swagger web page.

Navigating

MOCA provides a URL to easily access the current documentation. The documentation is shown by WAR.

Example: The following example illustrates a sample URL to view the MOCA admin web service API documentation. If you navigate to the default path, you must enter the path to the WAR API documentation. You may optionally set the default path using the URL parameters in the example.

```
//Format
http://{hostname}:{port}/api

//Specific war
http://{hostname}:{port}/api?url=http://{hostname}:{port}/ws/{war}/api-docs

//Example using localhost:4500
http://localhost:4500/api?url=http://localhost:4500/ws/admin/api-docs
```

Adding documentation to endpoints

The following code is a full example of adding documentation details to endpoints.

```
@RequestMapping(value="login", method=RequestMethod.POST)
    @ApiOperation(value="A POST endpoint used for logging in.")
    public void login(HttpServletRequest request, HttpServletResponse response,
                      @ApiIgnore MocaContext moca,@ApiParam(name="argMap", value="A
                      json map of the user's login credentials", required=true)
                      @RequestBody Map<String, String> argMap) throws MocaException,
                      UnsupportedEncodingException {
```

The following java annotations can be used to create more thorough documentation:

- @Api
- @ApiOperation
- @ApiIgnore
- @ApiParam

For more information on annotations, see the Swagger developers' annotation web page.

Deployment

Description

With MOCA, the deployment mechanism is web archive (WAR) based. This lets you deploy any servlet type. Also, this lets you dynamically redeploy your web services since WARs can be added, removed or changed while the server is running. The server scans the `$LESDIR/webdeploy` directory for objects to deploy.

MOCA provides assistance in deploying Spring MVC-based WARs by supplying a default web.xml to use Spring and the necessary Spring-based configuration files.

Layout

Starting with version 2013.2, MOCA provides a package layout that mirrors how WARs are generated. This provides greater flexibility.

```
<prod-dir>
    <ws>
        <war-name>
        <src>/<java>
        <src>/<resources>
        <spring>
        <lib>
        <any-other-dir>
        ...
```

The layout lets MOCA generate a proper WAR. Specifically:

- **<war-name>** – Name to use for the generated WAR.
- **<src>/<java>** – Directory that contains all of the files that need to be compiled. The compiled WAR-specific classes are then saved in the <classes> subdirectory of the generated WAR.
- **<src>/<resources>** – One or more files and/or directories that contain all of the resources for the WAR. These files and/or directories are copied to the <classes> subdirectory of the generated WAR.
- **<spring>, <lib>, <any-other-dir>** – Subdirectories of the <war-name> directory that need to be included in the generated WAR. These directories and their contents are copied to the WEB-INF subdirectory of the generated WAR. You can list as many directories under the <war-name> directory that you need to include in your generated WAR.

Default WAR Files

By default, MOCA provides these files in the WAR:

- **<spring>/webservices-config.xml** – XML-based file that contains MOCA-specific configuration that applies all filters and interceptors for Spring. You typically do not want to override this configuration at the application level.
- **<spring>/webservices.xml** – XML-based file that contains application-specific configuration that exposes endpoints and other beans. By default, MOCA provides one bean that performs classpath scanning to find all classes with the Component annotation in the com.redprairie package. You may override this, for example, to provide more specific bean name lookups or to add additional beans to the context for dependency injection and so on.

- **jetty-web.xml** – XML-based file that contains configuration for any additional values that Jetty controls. By default, MOCA sets the WAR's context path to /ws.
- **web.xml** – XML-based configuration file that defines the interaction with Spring. Spring then references the two .xml files in the <spring> directory (webservices-config.xml and webservices.xml).

Ant Support

MOCA provides targets in the build.xml that can be copied or referenced in other product build.xml files. If you have set up your WAR directory structure as outlined earlier, the targets build a product's WAR file.

The `targetwar` target takes a parameter of target. The target value should be the <war-name> directory name under the ws directory. The WAR is placed in the <prod-dir>/<webdeploy> directory.

This is an example of the XML code that you need to provide in the build.xml file for building the WAR using Ant.

```
<antcall target="targetwar">
  <param name="target" value="moca"/>
</antcall>
```

Deployment Tools

WSDeploy

MOCA has an executable named **wsdeploy** that automatically deploys all WARs from their product directories to the `%LESDIR%\webdeploy` directory.

The tool is located in the `%MOCADIR%\bin` directory. If the MOCA bin directory is in the operating system path, you can run `wsdeploy` from the command line.

In addition to deploying WARs wsdeploy can be configured to cleanup old or obsolete WAR files that have already been deployed to the `%LESDIR%\webdeploy` directory. This can be achieved by creating a wsdeploy.properties configuration file in the data directory of any product directory configured in the MOCA registry. Once this file is created the wsdeploy.remove-wars property can be used to specify any old or obsolete WAR files to be removed from the `%LESDIR%\webdeploy` directory (if any the specified WAR files does not exist wsdeploy continues as usual).

The following sample is what the file could look like.

`%APPDIR%\data\wsdeploy.properties`

```
wsdeploy.remove-wars=old-app.war,old-app.2.war
```

Authentication and Web Services

Description

MOCA Spring web services provide session key authentication similar to MOCA clients before the introduction of Spring web services.

Authentication is obtained by calling the `/ws/auth/login` web service and passing credentials. These are the methods for passing credentials:

- **URL Arguments** – Pass the user ID and password as URL arguments. This method is more for testing, since not all characters are supported. For example:

  ```
  ?usr_id=UserA&password=APswd
  ```

- **JSON Body** – Pass the user ID and password in a JSON body. When using this method, it is preferred that you use UTF-8. Also, make sure that content-type is set to application/json. For example:

  ```
  { "usr_id" : "UserA", "password" : "APswd" }
  ```

If the user ID and password are successfully authenticated, you see a 200 return status and a response cookie. The cookie is named MOCA-WS-SESSIONKEY and contains the cluster-generated session key. This key supports authentication across all nodes in the same cluster. In addition, this key times out (similar to non-Spring web services). Therefore, there is no guarantee that the authenticated session key is always valid.

The web services key is the same as the JDA SCE Client key. Therefore, if you want to use single sign on, you can use the same authenticated session key for either the web client or JDA SCE Client.

All endpoints by default require authentication except for the `/ws/auth` endpoint.

Adding MOCA Authentication to a Third-Party WAR

You can configure a third-party web archive (WAR) to enable authentication through the MOCA server. To enable MOCA authentication, you need to add the authentication filter to the WARs web.xml.

```
<filter>
    <filter-name>Authentication</filter-name>
    <filter-
class>com.redprairie.moca.servlet.AuthenticationFilter</filter-class>
    <description>
       This filter does standard authentication
    </description>
  </filter>

  <filter-mapping>
    <filter-name>Authentication</filter-name>
    <url-pattern>/*</url-pattern>
  </filter-mapping>
```

Authorization

To control a user's access to a web service endpoint, MOCA provides the `@Authorization` Java annotation. Typically, you want to restrict web service access to MOCA commands, such as administrative commands, to only those users who should be able to run the commands.

The `@Authorization` annotation has the following characteristics:

- It can be applied to classes and methods.
- It has one element, `options`, that is an array of `Strings` where each `String` is a role option as defined in the Role Maintenance application or a static variable. See "Open Authorization" (on page 220) following in this section.

IMPORTANT: Distributed role options can be used in custom code, but they cannot be overridden in standard code.

- It uses the `get user privileges` MCS command with the argument option type of `U` (for URL) to determine the user's role options for MOCA web service endpoints.

Definition: The following code defines the `@Authorization` annotation.

Authorization.java

```
@Target(value={ElementType.METHOD, ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Inherited
@Documented
public @interface Authorization {
    String[] options();
}
```

Example: The following code illustrates the application of the `@Authorization` annotation to the `getTaskInformation` method in the `TaskService` class. To call the `/ws/<war>/tasks` web service endpoint with a `GET`, the user must have a role that has the `optMocaAdmin` role option.

TaskService.java

```
@Controller
public class TaskService {

...
    @Authorization(options={"optMocaAdmin"})
    @RequestMapping(value="tasks", method=RequestMethod.GET)
    public List<TaskStatusWrapper> getTaskInformation() {
        List<TaskDefinition> tasks = _manager.getCurrentTasks();
        List<TaskStatusWrapper> wrappers = new
ArrayList<TaskService.TaskStatusWrapper>(
                tasks.size());
        for (TaskDefinition task : tasks) {
            wrappers.add(new TaskStatusWrapper(task, _
manager.isRunning(
                task.getTaskId())));
        }

        return wrappers;
    }
...
```

Example: The following code illustrates the application of the `@Authorization` annotation to the `TaskService` class.

TaskService.java

```
@Controller
@Authorization(options={"optMocaAdmin"})
public class TaskService {
...
}
```

For information on the authorization implementation for MOCA commands, see "Command Authorization" (on page 115).

Open Authorization

While not recommended and should rarely be used, you can skip authorization by setting the `@Authorization` `options` element to the `OPEN` static variable in the `MocaWebAuthorization` class, which results in skipping `get user privileges`.

Example: The following code illustrates the use of open authorization in the MOCA login service (any user should be able to log in):

LoginService.java

```
@Controller
@Authorization(options={MocaWebAuthorization.OPEN})
public class LoginService {
...
}
```

The `OPEN` static variable is similar to the `optOpen` role option that is used to specify open authorization for MOCA commands. See "Open Authorization" (on page 116).

Application of class and method authorizations

MOCA applies the class and method `@Authorization` annotations based on the following rules.

Method authorization only

If a method annotation exists but the method's class does not have an annotation, if one of the user's role options matches one of the method `@Authorization` role options, authorization succeeds and the method can run.

IMPORTANT: If you are using method authorization only, you should annotate all of the methods in the class.

Example: The following code illustrates method authorization without class authorization. If one of the user's role options matches `optAuth1`, the `getMethod1` method can run.

```
public class Class1 {
...
    @Authorization(options={"optAuth1"})
    public getMethod1() {
    ...
    }
}
```

Class authorization only

If a class annotation exists but the method in the class does not have an annotation, if one of the user's role options matches one of the class `@Authorization` role options, authorization succeeds and the method can run.

Example: The following code illustrates class authorization without method authorization. If one of the user's role options matches `optAuth1`, the `getMethod1` and `getMethod2` methods can run.

```
@Authorization(options={"optAuth1"})
public class Class1 {
...
    public getMethod1() {
    ...
    }
    public getMethod2() {
    ...
    }
}
```

Method and class authorization

If both class and method annotations exist, the class annotation is ignored and only the method annotation is used, which provides a way to override the class annotation. Specifically, if one of the user's role options matches one of the method `@Authorization` role options, authorization succeeds and the method can run. If you do not want the method annotation to override the class annotation, you must add the class authorizations to the method authorizations.

Example: The following code illustrates class and method authorization:

- If one of the user's role options matches `optAuth1`, the `getMethod2` and `getMethod3` methods can run.
- If one of the user's role options matches `optAuth2`, the `getMethod1` and `getMethod3` methods can run.

```
@Authorization(options={"optAuth1"})
public class Class1 {
...
    @Authorization(options={"optAuth2"})
    public getMethod1() {
    ...
    }
    public getMethod2() {
    ...
    }
    @Authorization(options={"optAuth1","optAuth2"})
    public getMethod3() {
    ...
    }
}
```

No authorization

If no annotations exist on a method or its class, authorization currently behaves like open authorization. See "Open Authorization" (on page 220) earlier in this section. However, once the JDA SCE applications implement `@Authorization` annotations, no annotations cause authorization to fail. See "Authorization failure" (on page 222) following in this section.

Authorization failure

If a user does not have at least one role option that matches a class or method role option, the following results occur:

- Authorization fails
- MOCA raises an `AuthorizationException` error and returns the `FORBIDDEN (403)` error status to the client
- The web service endpoint does not run

Definition: The following code defines the MOCA response to an authorization failure.

```
@ResponseStatus(value=HttpStatus.FORBIDDEN, reason="Unauthorized
endpoint")
public class AuthorizationException extends MocaException{

    public AuthorizationException() {
        super(544, "Could not authorize.");
    }
}
```

Verifying @Authorization annotations

You can use the `get user privileges` MCS command to ensure that users have the role options that you are using in your annotations.

Example of Creating Your Own Web Service

Description

This is an example of creating an admin web archive (WAR) with functionality to query task information.

WAR Subdirectory

Under the %PRODDIR%/ws directory, create a subdirectory that is named the same as your WAR. Use a name that clearly defines the type of functionality the web service endpoints in the WAR provides.

```
trunk moca/ws/admin:>ls
spring  src
```

Webservices.xml File, TaskService Class and @Controller Annotation

Inside the spring directory, MOCA expects to find a `webservices.xml` file that tells Spring what classes (controllers) should be included in the WAR. For example:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"

xsi:schemaLocation="http://www.springframework.org/schema/beans
```

```
http://www.springframework.org/schema/beans/spring-beans.xsd
           http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context.xsd">
    <bean class="com.redprairie.moca.task.TaskService"/>
</beans>
```

In this case, the webservices.xml file lists a TaskService class that is a web service controller. This java file is in the src directory of the WAR folder.

```
trunk moca/ws/admin/src/java/com/redprairie/moca/task:>ls
TaskService.java
```

Every class that is going to be a Controller must be annotated with @Controller. All endpoints from the WAR name are specified by @RequestMapping where the value is the endpoint name and a request method.

```
@RequestMapping(value="tasks", method=RequestMethod.GET)
```

Example: This example illustrates a portion of the code for the TaskService class.

```
@Controller
public class TaskService {
    .
    .
    .
    @RequestMapping(value="tasks", method=RequestMethod.GET)
    public List<TaskStatusWrapper> getTaskInformation() {
        List<TaskDefinition> tasks = _manager.getCurrentTasks();
        List<TaskStatusWrapper> wrappers = new
ArrayList<TaskService.TaskStatusWrapper>(
                tasks.size());
        for (TaskDefinition task : tasks) {
            wrappers.add(new TaskStatusWrapper(task, _
manager.isRunning(
                task.getTaskId())));
        }

        return wrappers;
    }

    @RequestMapping(value="tasks/{taskId}", method=RequestMethod.GET)
    public TaskStatusWrapper getTaskInformation(
        @PathVariable String taskId) throws NotFoundException {
        TaskDefinition task = _manager.getCurrentTask(taskId);
        if (task != null) {
            return new TaskStatusWrapper(task, _manager.isRunning(
                    task.getTaskId()));
        }
        throw new NotFoundException();
    }
```

```
      .
      .
      .
  }
```

All returns from methods are written to a Spring View and sent back to the client.

Calling Your Web Service Endpoint

Description

You can use a plugin supported by modern browsers (for example, Poster) to send requests to your endpoints.

This example illustrates querying for the task, MTF_SERVER.

Web Service Request

Send the web service request to:

`http://localhost:4500/ws/admin/tasks/MTF_SERVER`

Results as XML

When specifying Accept as `application/xml`, these are the results:

```
<map>
   <entry>
      <string>taskStatusWrapper</string>
      <TaskStatus>
         <task>
            <autoStart>false</autoStart>
            <cmdLine>java -Xrs com.redprairie.mtf.terminal.Terminal -
v DEFAULT -W wh_id -a http://localhost:4500/service -j $MTFDIR/data
/mtf_logging.xml</cmdLine>
            <environment class="map"/>
            <name>MTF SERVER</name>
            <restart>false</restart>
            <runDirectory>$MTFDIR</runDirectory>
            <startDelay>0</startDelay>
            <taskId>MTF_SERVER</taskId>
            <type>D</type>
         </task>
         <running>false</running>
      </TaskStatus>
   </entry>
</map>
```

Results as JSON

When specifying Accept as `application/json`, these are the results:

```
{"taskStatusWrapper":{"running":false,"name":"MTF
SERVER","type":"D","traceLevel":null,"environment":
{},"role":null,"taskId":"MTF_
SERVER","autoStart":false,"startDelay":0,"cmdLine":"java -Xrs
com.redprairie.mtf.terminal.Terminal -v DEFAULT -W wh_id -a
http://localhost:4500/service -j $MTFDIR/data/mtf_
logging.xml","runDirectory":"$MTFDIR","logFile":null,"restart":fals
e}}
```

Tracing

Starting a trace

MOCA enables tracing on a per HTTP request basis. This means that you can enable tracing for a particular HTTP request. To do this, set the value of HTTP request header field `moca-tracefile` to the name of the log file. This creates the log file in `%LESDIR%\log`. Web service tracing is session based; the tracing information that is associated with each part of the request, including one or more MOCA commands, is appended to the log file.

In response to any HTTP request, MOCA specifies the node that processed the HTTP request in the value of the HTTP response header field `node`. This information can be used to determine the node on which the log file for the corresponding HTTP request is located. For more information, see "Clustering and Web Services" (on page 226).

Requesting a trace file

In addition to starting a trace, MOCA also provides an endpoint that can be used for retrieving a trace file from the MOCA server or cluster. This ability is especially useful when you have a MOCA cluster sitting behind a load balancer. This endpoint allows any log file with a specified name to be retrieved from the cluster regardless of the node from which the endpoint is called.

Endpoint	Request Method	Permissions (role option)	Input	Known Responses
/ws/admin/logs/<*File Name*>	GET	optMocaAdmin	<*File Name*> is the name of a file that exists in the **$LESDIR/log** directory	• **404** – No file was found with the specified name • **400** – The specified file name is not valid • **200** – The specified file name is returned with the response body as a .zip file (octet stream)

Example: The following endpoint illustrates requesting a trace file named **mocaserver.log** from the **$LESDIR/log** directory of the instance installed on the local server and port number 4500. If the local server is part of a cluster, then the following endpoint produces a .zip file that contains every log file named **mocaserver.log** in each node's **$LESDIR\log** directory.

```
http://localhost:4500/ws/admin/logs/mocaserver.log
```

Absolute paths and symbols for leaving the subdirectory, such as "..", are not allowed and, if used, a BAD REQUEST response is generated. If the server cannot find any file with the specified name, it generates a 404 NOT FOUND response. The .zip file is named the same as the original file name with .zip appended to it; for example, **mocaserver.log.zip**.

If more than one file is found in the cluster with the same name, each file is listed with its host name and port number prefixed to the file; for example, *<Host Name>-<Port Number>-<File Name>*.

Example: Calling endpoint **http://localhost:4500/ws/admin/logs/testlogfile.log** on a cluster where **testlogfile.log** exists on two nodes, the endpoint returns a file named **testlogfile.log.zip** that contains the following log files:

- **J1014808WIN7LT.jda.corp.local-4500-testlogfile.log**
- **KLPACKAG1WIN7DT.jda.corp.local-4500-testlogfile.log**

Monitoring and Performance

Performance data is collected in the form of probes for all web services. See Web Service Probes. Additionally, the probe information can be viewed using the MOCA Console. See the "Web Services Usage page" section in the *MOCA Console User Guide*.

Clustering and Web Services

When MOCA servers exist in a clustered configuration, a request could go to any node because of load balancing. This could be a problem in certain situations (such as finding a log file). Because of these situations, MOCA sets the value of an HTTP response header field named `Node`. This is set on every HTTP response with the MOCA instance URL. The `node` information lets you determine which node in the cluster processed the corresponding HTTP request.

Tasks and Jobs

MOCA provides endpoints that pertain to task and job definitions. These endpoints let you list tasks and jobs, and acquire their status including whether they are running. All task and job web services can be found in the admin web archive (WAR).

These are the URLs for the task-related endpoints:

- **http://<host>:<port>/ws/admin/tasks** – Returns a list of all task definitions.
- **http://<host>:<port>/ws/admin/task/<Task-ID>** – Returns the task definition for the specified <Task-ID>.

These are the URLs for the job-related endpoints:

- **http://<host>:<port>/ws/admin/jobs** – Returns a list of all job definitions.
- **http://<host>:<port>/ws/admin/job/<Job-ID>** – Returns the job definition for the specified <Job-ID>.

MOCA Context

When developing a web service endpoint, MOCA supports using a `MocaContext` object. On any `@RequestMapping` method inside your controller that has a `MocaContext` object as a parameter, MOCA injects the `MocaContext` object automatically. For example:

```
@RequestMapping(value="listOrders", method=RequestMethod.GET)
    public List<Order> listOrders(HttpServletRequest request, HttpServletResponse
response,
                      MocaContext moca) {
        MocaResults rs = moca.executeCommand("list orders");
        .
        .
        .
```

Request Variables

Description

Sometimes a client needs to set a request variable that overrides a session variable for one or more requests. For these situations, MOCA has implemented support for request variables. A MOCA request variable that is named the same as a session variable overrides that session variable.

MOCA-Request Header

To set a request variable, use an HTTP request header field named `moca-request` to access the variables through the MOCA context. However, headers require a special character encoding; otherwise, the request variables do not behave as expected. Therefore, MOCA provides a helper class named `ResponseUtils` (in `com.redprairie.moca.util.ResponseUtils`) to properly encode the request variables. All request variable entries are displayed in a trace file.

Request Variable Definition

Request variables are represented as a key-value pair, similar to configuration files. Separate more than one key-value pair with a comma.

Example: This example illustrates the code for setting one request variable in a request.

```
connection.addRequestProperty("moca-request",
ResponseUtils.encodeHeader("wh_id=newId"));
```

Example: This example illustrates the code for setting more than one request variable.

```
connection.addRequestProperty("moca-request",
ResponseUtils.encodeHeader("wh_id=newId,client_id=newClientID"));
```

IMPORTANT: Since request variables are delimited by commas, a request variable value cannot contain commas.

Request Variable Access

Example: This example illustrates the code for accessing a request variable.

```
MocaContext ctx = MocaUtils.currentContext();
String whId = ctx.getSystemVariable("wh_id");
```

IMPORTANT: Session variable wh_id can only be overridden per request. A request variable for wh_id does not apply across requests.

Web Services and Dates

MOCA web service endpoints only accept date strings in specific ISO8601 date formats. The following table lists the ISO8601 date formats and whether MOCA web service endpoints accept the format.

Date Format	Accepted
2014-07-15T01:01:01-05:00	Yes
2014-07-15T01:01:01-0500	No
2014-07-15T01:01:01.000-05:00	Yes
2014-07-15T01:01:01.000-0500	Yes

MOCA Console Support

All client sessions for web services are viewable from the MOCA Console. In addition, all operations that you can normally perform with the MOCA Console (such as interruption) are available.

Monitoring

The MOCA Console displays performance and monitoring-related data for web services. See the "Web Services Usage page" section in the *MOCA Console User Guide*.

MOCA Support and Restrictions

Description

MOCA supports or restricts Spring web services in several ways.

Stateless Requests

All requests are stateless. Specifically:

- There is no mode for auto commit being disabled; a commit or rollback occurs after every request. MOCA blocks any commit, rollback or save point work that is performed in the main thread of the request.
- Tracing is only performed on a per request basis (moca-tracefile header). For more information, see "Tracing" (on page 225).
- Keepalive is completely ignored.

Request Context

MOCA automatically sets up a context for each request. Then, if the endpoint code:

- Returns normally, MOCA performs a commit.
- Throws an exception, MOCA performs a rollback

Environment Variables

MOCA automatically attaches the USR_ID and WEB_CLIENT_ADDR to the environment of the request each time the request is made.

For information on using other variables, see "Request Variables" (on page 227).

Chapter 21. Configurable Web Services

Description

Configurable Web Services is a framework that enables developers to expose existing MOCA commands as RESTful web services without writing and compiling new code. The Configurable Web Services framework is used as the foundation for web user interface (UI) extensibility but also can be used for exposing an endpoint for external clients to call a RESTful web service.

How are Configurable Web Services different?

In addition to Configurable Web Services, MOCA supports the following techniques for exposing web services:

- SOAP web services
- Spring MVC

In contrast with SOAP web services, the web services exposed using the Configurable Web Services framework do not support the SOAP protocol. This is beneficial since SOAP web services require the existence of clients and servers that can send and process SOAP calls. Typically, developers must maintain the mappings between SOAP calls and application components. However, modern RESTful clients can call the endpoints exposed by Configurable Web Services and bypass the overhead required by SOAP.

In contrast with Spring MVC, Configurable Web Services are entirely configuration driven; no coding is required. RESTful web services are exposed if there is an existing MOCA command and a correct configuration.

Definitions

Configurable Web Services commonly uses the following terms:

- **Action**: Unit of work that can be performed by a user. Actions are synonymous with web services. Actions can be tied to a resource.
- **Resource**: Entity that models a real-world object that can be represented with a primary key or, optionally, a compound primary key. A resource usually maintains a list of its properties and actions, which enables a user to work with that type of resource.

Configuration

Registry

The MOCA Registry file controls the root URL that exposes Configurable Web Services in the ws-cws-context key under the SERVER section. The default value for server.ws-cws-context is "cws". This means that Configurable Web Services have the URL prefix of <MOCA_URL>/ws/cws. The value can be changed by updating the MOCA registry.

Example: The following example illustrates the MOCA registry updated with the URL prefix to be <MOCA_URL>/ws/app.

```
[SERVER]

ws-cws-context=app
```

Actions

Actions are configured to associate URLs (href) and methods to MOCA commands. The configurations are specified in XML format and saved in an **.action** file.

Example: The following example illustrates a sample action in the XML file format.

```
<action>
    <id>addJob</id>
    <name>Add MOCA Job</name>
    <description>Add a new MOCA job.</description>

    <type>ADD</type>
    <ws>
        <method>POST</method>
        <href>/jobs</href>
    </ws>
    <impl>
        <command>add job</command>
        <arguments>
            <argument name="job_id" type="string" required="true"/>
        </arguments>
    </impl>
    <auth>
        <opt>optOpen</opt>
    </auth>
    <primary>true</primary>
</action>
```

To create an action, identify and configure the following XML elements:

- action
- ws
- auth
- impl
- argument
- listConfig

Each XML element is comprised of one or more elements or attributes.

action element

The following table describes the action elements.

Element	Required	Default	Description
id	yes		Unique identifier for the action. This must be unique among all action IDs.
name			Name for the action. In most places, MOCA defaults to

Element	Required	Default	Description
			using the ID if name is not provided.
description			Description of the action. If not provided, description is filled in with the description of the implementing MOCA command.
type		ACTION	Representation of what the MOCA command does. Must be one of the following: `ADD`, `LIST`, `GET`, `EDIT`, `DELETE`, or `ACTION`. As a best practice, using `ADD` calls a MOCA command that creates, for example, add job; using `EDIT` calls a MOCA command that updates, for example, change job; and so on. These types are loosely coupled with commands meaning we do not validate that the type matches the effect. However, the types are more tightly coupled with the functionality of web UI extensibility.
ws	yes		Configuration for the webservice definition. See below for more details.
impl	yes		Configuration for the functionality implementation. See below for more details.
auth		`optMocaAdmin`	Configuration for authorization. See below for more details.
listConfig			Configuration for List Configuration settings. See below for more details.
primary		false	Indicates whether this is a primary CRUD action. There can only be one primary action of that type per resource. Primary actions are guaranteed to work when using them to manipulate those particular resources. Conversely, some resources may have unrelated CRUD actions on them which are actually working with different resource types and thus are not CRUD actions for the parent resource.

ws element

The following table describes the ws elements.

Element	Required	Default	Description
method	yes		HTTP method with which this web service can be called.
href	yes		URL that HTTP clients can call to execute the MOCA command.

The ws element specifies the endpoint that is created for this CWS action.

Example: The following example illustrates a sample action that uses a path variable.

```
<action>
    ...
    <ws>
        <method>GET</method>
        <href>/jobs/{job_id}</href>
    </ws>
    <impl>
        <command>list job</command>
        <arguments>
            <argument name="job_
id" type="string" required="true" location="path" description="ID of
the job."/>
        </arguments>
    </impl>
</action>
```

The curly braces "{ }" specify a path variable placeholder in the href. The name of the placeholder inside the braces is not important, and does not have to match the argument name in the <arguments> element; in the previous example it is {job_aid}, but {id} or any other name would work as well. It is good practice to make the placeholder variable name match the defined path argument for clarity purposes. The name of the placeholder may be changed when looked at through the metadata API, to preserve specific API contracts with REFS.

Multiple path variables, for example /jobs/{job_id}/env/{env_name} are allowed. Any action which requires multiple points of information in the path should use a compound path variable. See the argument element description for more information.

The contents of the the method and `href` elements must be unique when combined. For example, action addJob has `href` jobs and method POST. We cannot define the action startJob with `href` jobs and method POST. Instead, a different `href` is required: `/jobs/start/{id}` and method POST. However, the same `href` can be used with a different type. For example, action addJob has `href` jobs and method POST and action listJobs has `href` jobs and method GET, which works fine. This uniqueness must be maintained for all web services in the application, so consider that Spring MVC web services could be defined in a manner that conflicts with an href and method configuration pair.

auth element

The following table describes the auth element.

Element	Required	Default	Description
opt	yes	OPEN	Role permission required to call the web service.

If more than one opt is configured for the action, the elements are applied in an OR fashion, meaning that the user calling the action must have at least one of the specified permissions. See Authorization in the "Authorization" (on page 218) chapter.

impl element

The following table describes impl elements.

Element	Required	Default	Description
command	yes		The MOCA command that is called. The command must be a concrete command that is implemented with an mcmd file. It cannot be defined with free-form local syntax directly in the action definition.
arguments			Configuration for the inputs to the MOCA command. This element is a collection of argument attributes. See below for more details.

Arguments should be declared if and only if they are not already declared on the command mcmd. Declaring arguments on the action overrides the arguments on the mcmd if they are declared on the mcmd.

argument element

The following table describes the argument attributes.

Attribute	Required	Default	Description
name	yes		The name of the argument.
type	yes		The datatype of the argument. It should be one of: `DATE`, `INTEGER`, `DOUBLE`, `FLOAT`, `STRING`, `BOOL`, `ARRAY`, `OBJECT`.
required		`false`	Indicates whether this argument is required when calling the action webservice. For PATH arguments, this field is used to denote whether the argument is a primary key for the action. Unless specified as required, arguments are considered optional by default.
description			The description for the argument.
location		`QUERY`	The location of the argument. This can be one of `PATH`, `QUERY`, `HEADER`, `BODY`.

Any resource that has more than one primary key is referred to as having a compound primary key. An action that works with that resource only needs one set of braces "{ }" in the href configuration.

Example: The following example illustrates an action which has two required arguments corresponding to the primary keys: `job_id` and `name`.

```
<action>
    <id>deleteJobEnv</id>

    <type>DELETE</type>
    <ws>
        <method>DELETE</method>
        <href>/jobs/env/{id}</href>
    </ws>
    <impl>
        <command>delete job env</command>
        <arguments>
            <argument name="job_
id" type="string" required="true" location="path"/>
            <argument
name="name"   type="string" required="true" location="path"/>
        </arguments>
    </impl>
    <auth>
        <opt>optOpen</opt>
    </auth>
</action>
```

Notice that there is only one path argument placeholder in the href but two required path arguments. When calling this web service, to specify values for the compound primary key, include an asterisk and exclamation point (*!) between each of the arguments in the URL. The order of the actual values must match the lexicographic order of the path argument names.

Names of the arguments don't need to match the placeholder variable between the curly braces. Even if there is one non-compound path variable, it does not need to match what is written in the curly braces. The CWS engine wires the arguments according to their configured location and name, so the `{id}` name in the above example is thrown away.

listConfig elements

The following table describes the listConfig elements.

Element	Required	Default	Description
pageable		false	A flag indicating whether or not the Action is pageable.
filterable		false	A flag indicating whether or not the Action is filterable.
sortable		false	A flag indicating whether or not the Action is sortable.

The listConfig elements are involved in configuring how list actions perform. The settings that can be configured using the listConfig elements include pageable, filterable, and sortable. It should be noted that these configurations are only used for Actions that are of type **LIST**. For any other Action type these configuration types are ignored.

Also the listConfig elements on their own do not make an Action pageable, filterable, or sortable. They are merely indicators to the API that the Action should be used as if it is page, filterable, or sortable. The underlying command implementing the Action needs to actually implement the pageable, filterable, and sortable functionality for the Action to work properly for each setting.

Example: The following example illustrates an action using listConfig elements to indicate that the action is sortable.

```
<action>
    <id>listJobs</id>

     <type>LIST</type>
    ...
    <listConfig>
       <sortable>true</sortable>
    </listConfig>
</action>
```

The listConfig element can also be configured using the MOCA Command directly, see "Command Repository" (on page 99). This means that the Configurable Web Service framework uses any of the listConfig elements if they are enabled by either the MOCA Command definition or the Action definition. Any MOCA Command can have these configurations but the Configurable Web Service framework only uses them if the Action is of type LIST as noted above.

Resource

A resource is configured to handle primary keys in web service calls and allow for the associating the resource to one or more actions. The configurations are specified in XML format and saved in a **.resource** file.

Example: The following example illustrates a sample job resource in the XML file format.

```
<resource>
    <id>moca_job</id>
    <name>MOCA Job</name>
    <description>A job is a command that is configured to run in the
background while the MOCA server is operating. Jobs are executed
within a thread of the MOCA server</description>
         <properties>
        <property name="job_
id"     alias="jobId"     pk="true" description="The primary key.
This is the primary identifier for the given job."/>
        <property name="role_
id"        type="string"    description="This job will run as part of
this role in a clustered environment.  All tasks/jobs that have the
same role are guaranteed to run on the same physical machine."/>
        <property
name="name"           type="string"    description="The name of the
job.  This is not unique and is used to easier find a job."/>
        <property
name="command"        type="string"  description="The MOCA command
line this job should execute when running."/>
        <property
name="enabled"        type="boolean"     description="Controls
whether this job is enabled."/>
        <property
name="overlap"        type="boolean"  description="Controls whether
multiple jobs can run at the same time."/>
        <property name="log_
file"       type="string"   description="The name of the log file the
standard output will go to."/>
        <property name="trace_
level"    type="string"   description="The trace level for the
execution of this job."/>
        <property
name="timer"          type="int"      description="If a timer job,
the time delay between runs."/>
        <property name="start_
delay"    type="string"  description="If a timer job, the start delay
before the first run."/>
        <property
name="schedule"       type="string"   description="If a schedule-type
job, the cron-style schedule of this job."/>
        <property name="grp_
nam"        type="string"   description="Group name."/>
        <property name="type"        type="string"   description="The
type of job -- cron or timer."/>
    </properties>
    <actions>
```

```
        <action type="add"      ref="addJob"/>
        <action type="list"     ref="listJobs"/>
        <action type="get"      ref="getJob"/>
        <action type="edit"     ref="editJob"/>
        <action type="delete"   ref="deleteJob"/>
        <action type="action"   ref="startJob"/>
    </actions>
</resource>
```

To create a resource, identify and configure the following XML elements:

- resource
- property
- action

resource element

The following table describes the resource elements.

Element	Required	Default	Description
id	yes		Unique identifier for the resource. This must be unique among all resource IDs.
name			Name for the resource.
description			Description of the resource.
properties			Specifies the fields or properties that define this resource. This element is a collection of property elements. See below for more details.
actions			Specifies the actions which can be taken with this resource. This element is a collection of argument elements. See below for more details.

There can be only one `ref` for each type of action (`ADD`, `LIST`, `GET`, `EDIT`, `DELETE`). However, you can provide more than one ACTION type.

Example: The following example illustrates a job resource that has two ACTION type actions configured: a startJob action and a job statistics action that provides information such as how many times the job has run and when it runs next.

```
<actions>
    <action type="add"      ref="addJob"/>
    <action type="list"     ref="listJobs"/>
    <action type="get"      ref="getJob"/>
    <action type="edit"     ref="editJob"/>
    <action type="delete"   ref="deleteJob"/>
    <action type="action"   ref="startJob"/>
    <action type="action"   ref="displayJobStats"/>
</actions>
```

property element

The following table describes the property attributes.

Attribute	Required	Default	Description
name	yes		Unique identifier for the property.
alias			Alternative name for the property.
pk		`false`	Indicates whether this is (or is part of) the primary key of the resource.
type		`STRING`	The data type of the property. Must be one of the following values: `STRING, INTEGER, BOOL, DOUBLE, DATE, OBJECT, ARRAY` The default value is `STRING`.
description			Additional information about the property.

action element

Note: These attributes should not be confused with top-level `<action>` elements which are defined in `.action` files. While they share the same name, these attributes are really *foreign-keys* to the top-level actions.

The following table describes the action XML attributes.

Attributes	Required	Default	Description
type	yes		Action type of the action in the ref attribute as specified in the action's XML configuration file. Must be one of the following values: `ADD`, `LIST`, `GET`, `EDIT`, `DELETE`, or `ACTION`.
ref	yes		Reference to an action to associate with the given resource. This should match an ID of an existing action.

Action link

Action links are configured to associate existing XML or Java actions with existing Java resources. This is needed because a Java resource cannot be edited like an XML resource can to add a new action. Extending an existing resource allows a new action to be used in REFS Page Builder for that resource. The configurations are specified in XML format and saved in an **.actionLink** file.

Example: The following example illustrates an action link in the XML file format.

```
<actionLink>
    <actionId>test-order-controller.get[id]</actionId>
    <newActionId>copy-test-order-controller.get[id]</newActionId>

    <javaResourceId>com.redprairie.moca.web.rest.services.resources.T
estOrderLine</javaResourceId>
</actionLink>
```

To create an action link, identify and configure actionLink element.

actionLink elements

The following table describes the actionLink elements.

Element	Required	Default	Description
actionId	yes		Action ID which is linked to the Java resource. This ID can belong to a Java or XML action.
javaResourceId	yes		Java resource ID to which to add the new action.
newActionId	yes		Action ID for the new action that is created. This is required because all actions must have a unique ID and action links create a new action.

File Location

On startup, MOCA scans all directories listed in the `server.prod-dirs` registry key and searches the the **data/ws** subdirectory for .action and .resource files. For example, given the following MOCA Registry, the LESDIR/data/ws/vcListUsers.action file is loaded on startup.

MOCA_REGISTRY

```
[SERVER]
prod-dirs=%LESDIR%;%HUBDIR%;%SLDIR%;%MCSDIR%;%MOCADIR%;%DEVTOOLS%
```

IMPORTANT: The application needs to be restarted for changes to take effect. Also, for clustering, make sure that actions and resources are copied to each node on the cluster.

Limitations

Configurable Web Services have the following limitations:

- They do not support compound resource definitions. For example, if you have an href for orders, `/orders/{id}`, you cannot add an endpoint for order lines.
- They cannot append to an href, for example, `/orders/{id}/lines/{id}`. Instead, the href should handle this situation with a compound key and single identifier, for example, `/orders/lines/{id}` or `/orderlines/{id}`.

Other limitations that exist are due to the enforcement of the following best practices by HTTP method:

- **LIST**: Path variables and body are ignored.
- **GET**: Request body query parameters are ignored. The only parameters allowed are path parameters, for example, `/ws/cws/jobs/{id}`.
- **ADD**: Supports request body.
- **EDIT**: Supports request body.
- **DELETE**: Request body is ignored.

Authorization

A Configurable Web Service action requires the caller to be authenticated and have the correct permissions according to the action configuration. The needed permissions can belong to a role or be assigned to the user directly. The permissions must be of type '**U**' (MOCA Web Services).

Example: The following example illustrates an auth permission within a Configurable Web Service action.

```
<action>
    <id>sampleAction</id>
    ...
    <auth>
        <opt>optAppAdmin</opt>
        <opt>optGeneralAdmin</opt>
    </auth>
</action>
```

If more than one opt is configured for the action, they are applied in an OR fashion, meaning that the user calling the action must have at least one of the specified permissions. If an auth permission is not specified in the action XML, the action defaults to requiring the **optMocaAdmin** permission. See "Authorization" (on page 218) in the Web Services chapter for more information on the MOCA web service auth system.

Example Configuration And Execution

Login

All of the examples in the Example Configuration And Execution section call `ws/auth/login` because like regular MOCA web services, configurable web services require the caller to be authenticated.

Example: The following example illustrates the login call.

```
GET
http://localhost:4500/ws/auth/login?usr_id=super&password=super
```

Simple Configurable Web Service action

To configure a simple action, create an **.action** XML file in the **%LESDIR%/data/ws/** directory.

Example: The following example illustrates a listJobs action.

```
<action>
    <id>listJobs</id>
    <name>List MOCA Jobs</name>
    <description>List all MOCA jobs.</description>

    <type>LIST</type>

    <ws>
        <method>GET</method>
        <href>/jobs</href>
    </ws>
    <impl>
        <command>list jobs</command>
    </impl>
    <auth>
        <opt>optOpen</opt>
    </auth>
</action>
```

Example: The following example illustrates a call, of the simple action. It does not require any arguments. The Configuable Web Service framework constructs a local syntax statement from the defined command and executes it.

```
GET
http://localhost:4500/ws/cws/jobs
```

Creating a resource with actions

To configure a resource, create a **.resource** XML file in the **%LESDIR%/data/ws/** directory.

Example: The following example illustrates a sample resource.

```
<resource>
    <id>moca_job</id>
    <name>MOCA Job</name>
    <description>A job is a command that is configured to run in the
background while the MOCA server is operating.</description>

    <properties>
        <property name="job_
id"     alias="jobId"     pk="true" description="The primary key.
This is the primary identifier for the given job."/>
        <property name="role_
id"        type="string"    description="This job will run as part of
this role in a clustered environment.  All tasks/jobs that have the
same role are guaranteed to run on the same physical machine."/>
        <property
name="name"           type="string"    description="The name of the
job.  This is not unique and is used to easier find a job."/>
        <property
name="command"        type="string"  description="The MOCA command
line this job should execute when running."/>
        <property
name="enabled"        type="boolean"     description="Controls
whether this job is enabled."/>
        <property
name="overlap"        type="boolean"  description="Controls whether
multiple jobs can run at the same time."/>
        <property name="log_
file"       type="string"   description="The name of the log file the
standard output will go to."/>
        <property name="trace_
level"    type="string"   description="The trace level for the
execution of this job."/>
        <property
name="timer"          type="int"      description="If a timer job,
the time delay between runs."/>
        <property name="start_
delay"    type="string"  description="If a timer job, the start delay
```

```
before the first run."/>
        <property
name="schedule"       type="string"   description="If a schedule-type
job, the cron-style schedule of this job."/>
        <property name="grp_
nam"        type="string"   description="Group name."/>
        <property name="type"        type="string"   description="The
type of job -- cron or timer."/>
    </properties>

    <actions>
        <action type="add"      ref="addJob"/>
        <action type="list"     ref="listJobs"/>
        <action type="get"      ref="getJob"/>
        <action type="edit"     ref="editJob"/>
        <action type="delete"   ref="deleteJob"/>
        <action type="action"   ref="startJob"/>
    </actions>
</resource>
```

Then create and link actions for the resource. To link the action to the resource, specify the action ID in the ref attribute in the resource actions.

Example:The following example illustrates the `addJob` action.

```
<action>
    <id>addJob</id>
    <name>Add MOCA Job</name>
    <description>Add a new MOCA job.</description>

    <type>ADD</type>

    <ws>
        <method>POST</method>
        <href>/jobs</href>
    </ws>
    <impl>
        <command>add job</command>
    </impl>
    <auth>
        <opt>optOpen</opt>
    </auth>
</action>
```

Notice that the action does not need to define any arguments. This is because the add job component already defines arguments for itself in the **.mcmd** file. The Configurable Web Service framework references RESTful standards to determine where arguments need to be located in the web service request. In this case, since the action is of type ADD, arguments should be posted as a JSON block in the body. When calling an action web service, if the request does not include all the required argument of the action, a 400 HTTP status is returned by the server.

Example: The following example illustrates execution of the action.

```
POST
http://localhost:4500/ws/cws/jobs
{"job_id":"test_job", "name":"test_
job", "command":"noop", "timer":60}
```

Using path variables and PKs

Actions also allow you to use path variables to identify a specific resource. When defining an action, you can specify a PATH variable which corresponds to the curly braces in the `HREF`.

Example: The following example illustrates an action that uses a path variable for the job ID.

```
<action>
    <id>editJob</id>
    <name>Edit MOCA Job</name>
    <description>Edit MOCA Job.</description>

    <type>EDIT</type>

    <ws>
        <method>PUT</method>
        <href>/jobs/{job_id}</href>
    </ws>
    <impl>
        <command>change job</command>
        <arguments>
            <argument name="job_
id" type="string" required="true" location="path" description="ID of
the job."/>
        </arguments>
    </impl>
    <auth>
        <opt>optOpen</opt>
    </auth>
    <primary>true</primary>
</action>
```

Example: The following example illustrates a call to an endpoint with a single primary key and a body.

```
PUT
http://localhost:4500/ws/cws/jobs/test_job
{"timer":30}
```

An action can also have multiple path variables, which corresponds to having a compound primary key.

Example: The following example illustrates an action that uses more than one path variable.

```
<action>
    <id>changeJobEnv</id>
    <name>Change MOCA Job Environment Variable</name>
    <description>This action allows you to change an environment
```

```
variable for a specific job.</description>

    <type>EDIT</type>

    <ws>
        <method>PUT</method>
        <href>/jobs/env/{pk}</href>
    </ws>
    <impl>
        <command>change job env</command>
        <arguments>
            <argument name="job_
id"  type="string"  required="true" location="path" description="ID
of the job."/>
            <argument
name="name"    type="string"  required="true" location="path" descrip
tion="Name of the variable."/>
            <argument
name="value"   type="string"  required="true" location="body" descrip
tion="Value of the variable."/>
        </arguments>
    </impl>
    <auth>
        <opt>optOpen</opt>
    </auth>
</action>
```

This action takes the primary key of the resource (job_id, name) from the path variable and takes the value from the body. Notice that the contents inside the curly braces do not have to match any argument name since the value corresponds to multiple arguments.

When calling the action, the path arguments should be specified together in a lexicographic order of their names, separated by the separator string.

Example: The following example illustrates a call to an endpoint with a compound primary key.

```
PUT
http://localhost:4500/ws/cws/jobs/env/ems_spooler*!path
{"value":"%PATH%"}
```

Extending an existing Java resource

Actions can belong to a resource, or they can be defined by themselves. Sometimes you may need to tie a floating action to a resource, or to tie a Java action to a different Java resource, for example for the purposes of Page Builder. To do this, the Configurable Web Service framework uses action links. An action link specifies the ID of an XML or Java action along with a resource ID. This link finds the action and copies it to the given resource - giving it a new ID in the process. It does not modify the original action.

Example: The following example illustrates an action link used to copy an action that returns information about an Order to the OrderLine resource.

```
<actionLink>
    <actionId>test-order-controller.get[id]</actionId>
    <newActionId>copy-test-order-controller.get[id]</newActionId>

    <javaResourceId>com.redprairie.moca.web.rest.services.resources.T
estOrderLine</javaResourceId>
</actionLink>
```

This action link takes the action **test-order-controller.get[id]**, which is implemented as a Java web service as part of the TestOrder resource, and makes a copy under the TestOrderLine resource. Anyone looking at the machine readable API is now able to see this action, now with the **copy-test-order-controller.get[id]** ID, under the TestOrderLine resource.

Note: Action IDs for Spring web services are generated and are in the form of `simpified-class-name.method[parameters]`. Look at the machine-readable API to find the exact ID.

Example: The following example illustrates the machine readable API output with the action link in place.

```
{
    "version" : "1.0",
    "compositeKeySeparator" : "*!",
    "resources" : [{
    "id" :
"com.redprairie.moca.web.rest.services.resources.TestOrderLine",
    "name" : "TestOrderLine",
     "properties" : ...,
    "actions" : [{
    "id" : "copy-test-order-controller.get[id]",
    ...,
    }, {
    "id" : "test-order-line-controller.list[]",
    ...
    },
    ...
    ]
    }
    ],
    "actions" : [...]
}
```

Action links work for both XML and Java actions. They do not work for XML resources. Add an action to an XML resource directly in the resource XML file.

Utilities

Machine readable API

MOCA offers a web service API to list resources and actions on the server.

Example: The following example illustrates the call using the MOCA web service.

```
GET
ws/admin/services
```

To use the API, use a GET HTTP method with the corresponding endpoint. The request must also have the Accept header set to `application/json`. The returned response is a JSON object listing resources and actions.

Resources returned by the API are objects that relate specifically to web services, namely those defined by **.resource** files or those used as generic types in controllers that extend AbstractIdentifiableWebResourceController or AbstractWebResourceController. The API does not list other types of domain objects like Hibernate Data Access Objects (DAOs), or any other Plain Old Java Objects (POJOs) or beans that exist on the server. With current web service standards, most resources correspond to interfaces instead of implementation classes, for example, TestAddress instead of TestAddressImpl in MOCA. In these cases, the resources are modeled by inspection of the interface methods. The primary keys (PKs) are determined by inspection of the PK fields defined on the PK generic type, for example, TestAddressPK in MOCA. Java resources are uniquely identified by their fully qualified classname.

Actions returned by the API are either existing Java webservices or Configurable Web Service actions defined by **.action** files. They are organized according to their potential relationship to a resource. Actions that are tied to a resource are displayed under the respective resource's actions JSON array. On the other hand, actions that are not associated with a specific resource are listed in the top-level actions array of the response. Actions that are associated with a resource are not listed again in the top-level array. Spring-based actions are uniquely identified by an ID that is based on the implementing controller class name, the respective controller method, and its parameter names. XML-based actions are identified by the specified id in the XML definition.

Note: Some metadata presented in the API may be slightly modified by MOCA to preserve API contacts with REFS. For example, the hrefs for XML actions could have different placeholder names.

Configuration

MOCA uses the best available information to guess the values for existing web services when listing actions and resources. In general, information returned by the machine-readable API should be consistent with Swagger documentation. Developers have a few options to tune to make sure that information for their web services in the API is correct.

Action Type

Action type is derived from the HTTP method of the action as well as the return type. By default, MOCA would guess that an action with HTTP method POST for a web service that logs a user into the system is of type *CREATE*, but you may want to mark it as type ACTION. To do so, add the `@ActionType` annotation on the controller method implementing the webservice.

Action Description

Action description is derived the controller method name. Otherwise it is overridden with the `@ApiOperation` Swagger annotation if present on that controller method.

Action Parameters

Action parameters are derived from the controller method parameter names, types, and their Swagger annotations. Parameters can be adjusted with the `@ApiParam` annotation and parameters which are not listed in the method themselves can be defined with `@ApiImplicitParam` annotation.

Example: The following example illustrates a sample controller method with annotations to adjust API values.

```
@ActionType(type = Action.Type.ACTION)
@RequestMapping(value="login", method=RequestMethod.POST)
@ApiOperation(value="A POST endpoint used for logging in.")
public void login(HttpServletRequest request, HttpServletResponse
response,
                  @ApiIgnore MocaContext moca,
                  @ApiParam(name="argMap", value="A json map of the
user's login credentials", required=true)
                  @RequestBody
                  Map<String, String> argMap) throws MocaException {
    // perform login
}
```

Chapter 22. Command Profiling

Description

This section provides information on how to enable and disable command profiling and then analyze command profile information.

Every time the MOCA server executes a command, it keeps track of the amount of time that the command takes to execute. This information can be made available in a number of ways in order to support debugging and performance analysis efforts. Additionally, performance statistics for SQL and trigger execution is made available in the same way.

Enabling Profiling

By default, command profiling is enabled within a process only. Each process that contains a MOCA engine (the MOCA server and server-side tasks) internally tracks command usage. For information about enabling persistent profiling that tracks overall usage across server restarts, see "Capturing Profile Output from Long-Running Processes" (on page 250).

Disabling All Profiling

To disable all profiling, including the in-memory profiling (to avoid reducing memory performance), add this code to the SERVER section of the registry file.

```
[SERVER]
command-profile=off
```

Viewing Profiling Statistics

MOCA Console

The easiest way to view command profiling statistics is to view the Command Profile page in the MOCA Console. This page presents the data in an easy to view format as a grid with expandable rows. You can also use the page to clear the statistics or download a CSV file that contains the command profile information.

List Command Usage Command

Description

Another way to view command profiling statistics is to use the **list command usage** command. The **list command usage** command returns information about each command that has been executed in the MOCA server.

Command Path

The MOCA server tracks command information based on a concept called the command path. The command path is a way of capturing the commands that, through their execution, caused a particular command to be executed.

For example, if a local syntax command `close orders` was implemented as two commands, "list orders where @* | process order close", then calling the `close orders` command would produce these three distinct command paths:

- close orders
- close orders->list orders

- close orders->process order close

Output of the List Command Usage Command

This table describes the published fields of the **list command usage** command.

Field	Description
`component_level`	For commands, the component level of the command being executed. For SQL or triggers, this field is blank.
`command`	Command being executed. For SQL, this field displays "SQL". For triggers, this field displays the trigger name
`type`	Type of the command being executed. These are the valid values: Java Method, C Function, COM Method or Local Syntax. For SQL statements, this field displays "SQL".
`command_path`	Command path. This is the path of execution that preceded this particular command/SQL/trigger execution.
`execution_count`	Number of times this command path has been executed.
`min_ms`	Minimum execution time in milliseconds.
`max_ms`	Maximum execution time in milliseconds.
`avg_ms`	Average time per call in milliseconds. This field can have a fractional value.
`total_ms`	Total time spent executing the command path over all executions.
`self_ms`	Time spent executing the command minus the time spent executing commands further down the command path.
`avg_self_ms`	Average time per call in milliseconds. This field can have a fractional value.

Review the `avg_self_ms` and `execution_count` fields. Commands that have a large value for those two fields are good candidates for review for potential performance improvement.

Example Profiling Session

This is an example of a command in MOCA, where the **get time string from seconds** command is implemented as a local syntax command.

```
publish data
    where hours=int(@seconds/3600)
      and minutes=int(@seconds/60) % 60
      and secs=@seconds % 60
|
[select ltrim(to_char(@hours,   '999')) || ':' ||
        ltrim(to_char(@minutes, '09'))  || ':' ||
        ltrim(to_char(@secs,    '09')) time_string
   from dual]
```

In general, selecting constant values from the `dual` table is considered a misuse of the database. To better understand the impact, it is useful to perform some profiling.

This example executes the **get time string from seconds** command in a loop using different values for the `seconds` argument.

```
MSQL> do loop where count = 86400 | get time string from seconds where seconds = @i;
   1  noop
   2  /

Executing... Success!

(0 Rows Affected)

MSQL> list command usage
   1  /

Executing... Success!

component_level  command                       type          command_path
 execution_count  min_ms  max_ms  avg_ms    total_ms  self_ms  avg_self_ms
---------------  ----------------------------  ------------  ----------------------------
-------------------------------  ---------------  ------  ------  --------  --------  --
-----  -----------
MOCAbase         publish data                  JAVA_METHOD   MOCAbase/get time string
from seconds->MOCAbase/publish data  86400            0       0       0.00685   591
591      0.00685
MOCAbase         do loop                       JAVA_METHOD   MOCAbase/do loop
 1                100     100     100.457   100       100      100.457
MOCAbase         get time string from seconds  LOCAL_SYNTAX  MOCAbase/get time string
from seconds                        86400            0       1501    1.072869  92695
1536     0.01778
                 SQL                           SQL           MOCAbase/get time string
from seconds->SQL                   86400            0       1501    1.048237  90567
90567    1.048237

(4 Rows Affected)
```

Reviewing the published fields, you can see that the command is taking about a millisecond per execution, but the "self" time is much lower. Just about all the time is being spent in the database. Each SQL statement run from the **get time string from seconds** command is taking more than a millisecond by itself (seemingly not much on a small scale). In this case, it is fairly easy to eliminate the database round trip and create a more efficient component.

Groovy was designed to facilitate some of these simple data manipulations. This example demonstrates the results when the SQL call is replaced with Groovy code.

```
publish data
    where hours=int(@seconds/3600)
      and minutes=int(@seconds/60) % 60
      and secs=@seconds % 60
|
[[
    time_string = hours + ':' +
                  minutes.toString().padLeft(2, '0') + ':' +
                  secs.toString().padLeft(2, '0')
]]
```

After an `mbuild`, the same tests are run again.

```
MSQL> do loop where count = 86400 | get time string from seconds where seconds = @i;
   1  noop
   2  /

Executing... Success!

(0 Rows Affected)

MSQL> list command usage
   1  /

Executing... Success!

component_level  command                        type          command_path
 execution_count  min_ms  max_ms  avg_ms    total_ms  self_ms  avg_self_ms
---------------  ----------------------------  ------------  ----------------------------
------------------------------  ---------------  ------  ------  --------  --------  --
-----  -----------
MOCAbase         publish data                   JAVA_METHOD   MOCAbase/get time string
from seconds->MOCAbase/publish data  86400           0       23      0.006602  570
570      0.006602
MOCAbase         do loop                        JAVA_METHOD   MOCAbase/do loop
 1                42      42      42.038    42        42       42.038
MOCAbase         get time string from seconds  LOCAL_SYNTAX  MOCAbase/get time string
from seconds                         86400           0       203     0.044507  3845
3274     0.037905

(3 Rows Affected)
```

Reviewing the published fields, you can see that the Groovy command takes just 0.04 ms instead of the SQL-based command taking an average of more than 1ms. Therefore, the Groovy command is more than 20 times faster, which means that the total time goes from 96 seconds down to under 5 seconds.

Capturing Profile Output from Long-Running Processes

To enable persistent profiling that tracks command usage across invocations of the application server, add this code to the "SERVER" section of the registry file.

```
[SERVER]
command-profile=%LESDIR%\log\mocaprofile
```

For simplicity and convenience of analysis, the profile data is stored in simple .csv files that can be loaded into Microsoft Excel or another spreadsheet program. The name of the server-side process is appended to the given name, along with the `.csv` suffix. The MOCA server places its output into a file with `-server.csv` appended. Using the previous example, the command profile is saved as `%LESDIR%\log\mocaprofile-server.csv`.

Chapter 23. Pooling Framework

Description

MOCA provides a pooling framework that lets you manage a group of shared resources, such as database connections. The pooling framework lets you easily implement your own pooling behavior and semantics. This allows for the reuse of the pooling logic while using different types of objects and even different validation schemes to determine when an object is automatically removed from the pool.

The pool is driven internally by a set of implementable interfaces: `Builder`, `Pool`, `BlockingPool`, and `Validator`.

Builders

Description

A builder is an interface that defines a single build method that returns a typed object. You typically provide the implementation for the `Builder` interface as appropriate for the pool that you want to create. For example, MOCA currently provides two standard builder implementations: one for creating the standard database connection pool and one for creating the native process thread pool.

Builder Interface

This is the code for the `Builder` interface.

```
public interface Builder<T> {
    /**
     * Returns a new instance of the type T
     * @return T a new instance of the type T
     */
    public T build();
}
```

The `Builder` interface lets you implement all of the code required to construct an object in one module.

Builder Return Objects

Your builder implementation can return any object to a pool with these constraints:

- The object must match compile type checking.
- The object must satisfy the validators that you implement. For more information, see "Validators" (on page 253).

Exception Handling

When implementing your builder, it is common practice to provide code to handle exceptions. Because pooling behavior can be asynchronous, the code must at least log an exception or warning to the trace and return a null object. This is important for troubleshooting your builder implementation, since it is possible that the exception will be thrown in a separate thread that is not tied to a request.

Pools

Description

MOCA provides two interfaces for Pools:

- **Pool** – This interface is used for normal operations such as polling objects and returning them in a non-blocking manner.
- **BlockingPool** – This interface is an extension to the non-blocking pool interface that adds blocking methods. This lets you more easily interact asynchronously with the pool objects.

Currently, MOCA has only implemented pools that are blocking asynchronous pools. You typically do not need to implement your own pool, but the interfaces are provided to implement pools, if desired.

Pool Implementation Types

These are the available pool implementation types:

- **Fixed** – This is the simplest pool implementation type. A fixed pool is defined with a fixed size and operates so that the total number of idle and busy objects in the pool is always equal to the fixed size. When the pool starts, it automatically builds the number of objects specified by the fixed size. If an object is invalidated, the pool automatically calls the builder to create a new object.

  You can create a fixed pool by using the `fixedSize(int size)` method on the `BlockingPoolBuilder` class.

- **Demand** – A demand type of pool implementation is similar to how MOCA handled the pooling of resources prior to the introduction of the pooling framework in version 2012.2.0. A demand pool is defined with a maximum pool size. The pool only creates an object when a request is received, but there are no idle objects available to fill the request and the pool has not yet reached its maximum pool size. With this type of pool implementation, the `poll` method always returns `null` unless there is an idle object. The `poll` method actually forces an object to be built, but there is no guarantee that the object build will complete before the poll method returns.

  A demand pool has the least scalability in regards to client requests since it requires locking to ensure that the state of the pool is valid. The pool state always needs to be valid so that demand-based object creation is handled properly.

  You can create a demand pool by using the `minMaxSize(int min, int max)` method on the `BlockingPoolBuilder` class by providing 0 as the value for min.

- **MinIdle** – A minimum idle type of pool implementation is the default pool used by the current standard MOCA pools (database connection and native process threads). A minimum idle pool is defined with a minimum number of idle objects and a maximum pool size. This type of pool maintains the specified minimum number of idle objects in the pool at all times, unless the maximum size is reached. When the pool starts, it automatically builds the specified minimum number of idle objects. Having available idle objects within the pool lets requests immediately retrieve a pool object instead of waiting for a new one to be built. If the actual number of idle objects drops below the specified minimum number of idle objects, the pool starts building objects asynchronously in the background until the number of idle objects in the pool returns to the specified minimum number. However, there is one scenario in which the number of idle objects in the pool may be less than the specified minimum number—if the pool reaches the specified maximum size.

You can create a minimum idle pool by using the `minMaxSize(int min, int max)` method on the `BlockingPoolBuilder` class and passing a value for min that is greater than 0 but not equal to max. If min and max are equal the builder instead uses a fixed pool.

All three blocking pool implementation types are built from a single abstract pool that holds all of the pooled objects in a blocking queue but uses non-locking updates to store the pool's internal state (specifying the number of idle, busy and created pool objects that exist in the pool). If you plan on extending this class, great care must be taken as non-blocking state updates are difficult to do correctly. You may want to use the TU_AbstractBlockingPool class to test lots of use cases.

Validators

Description

A validator is used to implement:

- Initializing a pool object when it is first created.
- Performing additional initialization of a pool object when it is retrieved from the pool for use by a requester.
- Determining whether a returning pool object is still valid for reuse and can be returned to the pool.
- Cleaning up a pool object that is no longer valid before discarding it.

You typically spend most of your development time on the code for validator implementation.

Validator Interface

The validator interface describes four methods for you to implement.

```
public interface Validator<T> {
    /**
     * This is called on an object after it has been first been
created.
     * @param t The object that was just created
     * @throws PoolException thrown if any error occurs during
initialization code
     */
    public void initialize(T t) throws PoolException;

    /**
     * Calls any setup methods required by this object when it is
retrieved
     * from the pool.
     * @param t the object to setup
     * @throws PoolException thrown if any error occurs while
reseting the
     *         pooled object
     */
    public void reset(T t) throws PoolException;
    /**
     * Checks whether the object is valid when returned back to the
pool.  If
     * this method returns true the object will be placed in the
pool.  If the
     * object is no longer valid it will instead be removed resulting
in
     * the {@link #invalidate(Object)} method being called.  Any
exceptions
     * are not propagated and
     * are treated as the object being invalid.
     *
     * @param t the object to check.
     * @return <code>true</code> if the object is valid else
<code>false</code>.
     */
    public boolean isValid(T t);

    /**
     * Performs any cleanup activities before discarding the object.
For example
     * before discarding database connection objects, the pool will
want to
     * close the connections.  Any exceptions from the object are not
propagated
     * and depending on implementation will most likely be logged.
     *
     * @param t the object to cleanup
     */
    public void invalidate(T t);
}
```

Typical Pool Object Life Cycle

A pool object life cycle typically flows through these steps.

1. The object is created using the `Builder.build()` method call.
2. The created object is immediately passed to the `Validator.initialize()` method.
3. The initialized object is added to the pool.
4. A request is received for a pool object.
5. The initialized object is passed to the `Validator.reset()` method.
6. The reset object is passed to the requester.

7. The requester returns the used object by passing it to the `Pool.release()` method that immediately passes the used object to the `Validator.isValid()` method.
   a. If the used object is determined to be valid, it is returned to the pool.
   b. If the used object is determined to be invalid, it is passed to the `Validator.invalidate()` method and is no longer referenced.

Circumventing the Validator

The Pool class has a method `Pool.removePooledObject()` that lets a caller circumvent the validator. However, this method should very rarely be used since it does not guarantee that the object is in a consistent state. (The validator helps ensure that a pool object maintains a consistent state.)

Default Implemented MOCA Validators

MOCA provides a few implemented validators that cover the more common use cases. Typically, you only need to extend one of the provided classes and/or override one of the initialize, reset and/or invalidate methods, instead of implementing an entire validator. These are the standard MOCA validators:

- **com.redprairie.moca.pool.validators.BaseValidator** – The base validator does nothing to a pool object in any method calls and always returns the object as valid. Use this validator for pools whose objects are always valid or as a base starting class to extend if you only want to implement a single method.
- **com.redprairie.moca.pool.validators.SimpleValidator** – The simple validator only implements the isValid method to set a pool object as invalid. Internally, this validator keeps a WeakHashMap of each object that has been marked as invalid. When the validator is used to determine whether the object is invalid, the validator checks the WeakHashMap. If the object is present, the validator reports that the object is invalid.
- **com.redprairie.moca.pool.validators.PoolUsageValidator** – The pool usage validator implements pool usage counting. Every time an object is retrieved from the pool, the validator updates the count in a WeakHashMap. In addition, the validator takes an argument that specifies the maximum retrieval count. When the number of times that an object has been retrieved from the pool (object reset count) equals the maximum retrieval count, the validator invalidates the object using the invalidate method.
- **com.redprairie.moca.pool.validators.MultiValidator** – The multiple validator lets you use more than one validator for the same pool by passing the validators to the `MultiValidator` class. When a validator method is invoked, the `MultiValidator` class calls each of its associated validators in validator argument order. If the initialize and/or reset methods throw a PoolException, it is caught and the associated validators can complete their processing. However, the first exception is re-thrown after completion. The `isValid()` method does not work for a multiple validator; the method does not call the associated validators if `isValid()` returns false.

Default MOCA Pool Builder

MOCA provides a builder that you can use to create a pool: the `com.redprairie.moca.pool.BlockingPoolBuilder` class. The pool builder requires a pool object builder to force generic compile type checks and to create a non-empty, usable pool that can be populated with pool objects. For more information, see "Builders" (on page 251).

```
_pool = new BlockingPoolBuilder<PooledObject>(builder)
            .name(poolName)
```

```
              .validators(validators)
              .minMaxSize(poolMinIdleSize, _poolMaxSize).build();
```

The `BlockingPoolBuilder` class lets you pass a single validator or multiple validators. If you pass in multiple validators, the class internally uses the `MultiValidator` class. For more information, see "Validators" (on page 253).

You can also set the name value for the pool. Naming a pool can help with debugging. If you have more than one pool, the error messages may be similar except for the pool name.

Dynamic Proxies (Optional)

MOCA provides an optional dynamic proxy utility class: `com.redprairie.moca.pool.validators.BasePoolHandler`. This class is the base class used for the dynamic proxy. You can use a dynamic proxy to call a validator method to control whether an object is returned to the pool instead of using the original class method to close or discard an object. For example, you might typically call the `close()` method of the `java.util.Connection` class on a connection to free resources. However, by using a dynamic proxy when you call the `close()` method, instead of actually closing the connection, the dynamic proxy calls the validator's `isValid()` method and attempts to return the connection to the connection pool.

In addition, a dynamic proxy supports being passed a set of exceptions. If an exception is ever received on an invocation, the dynamic proxy also sets the pool object to be invalid. This forces the pool object to be invalidated when it is returned to the pool and the invalidate method to be invoked.

Chapter 24. Localization

Introduction

Description

MOCA supports localization of the MOCA Console according to a locale. The locale determines how language text is displayed in the user interface. For more information on locales, see "Locale IDs" in the *Supply Chain Execution OnLine Expert*.

How it works

MOCA Console localization works according to the following rules:

- **MOCA Console**: These pages are automatically displayed in the user's locale which is specified in the **LOCALE_ID** in the **les_usr_ath** database table. However, if the user's locale cannot be determined from the database, the default locale is used.
- **MOCA Console login page**: Since the login page is not associated with a specific user and therefore cannot be automatically displayed in the user's locale, the login page is displayed based on the default locale. If the default locale is not suitable for the user who is logging in, the user can select a different locale from a list of installed locales to automatically update the login page to the suitable locale.

If a default locale is not specified in the MOCA registry, then U.S. English is used.

Configuration

Configure the locale for a user

To configure a locale for a user, select a **Locale ID** for the user in Authorization Maintenance (this action sets the user's **LOCALE_ID** to a valid locale). This is the same task that you complete to change a user's locale for other parts of the system, such as JDA SCE applications. For more information on specifying a locale for a user, see the help for Authorization Maintenance in the *Supply Chain Execution OnLine Expert*.

Configure the default locale

Typically, you only need to specify the default locale if the most common locale for the MOCA Console is not U.S. English. The default locale is used for the login page of the MOCA Console. It is also used as the locale for the MOCA administrative user. For more information, see the "Administrative password" section in the "Security" (on page 168) chapter.

Example: The following text is an example of setting the default locale to Great Britain English.

```
[SERVER]
console-default-locale=en-gb
```

1. Edit the MOCA registry.
2. For the **server.console-default-locale** key, specify a value that matches the appropriate **name** attribute that is defined in the locale mapping file, **%MOCADIR%/web/resources/locale-mapping.xml**.

Add a new locale to an existing instance

Use the following procedure to add a locale that did not exist when the instance was installed.

Note: Locales that exist during an installation are installed and configured automatically.

1. **Verify the mapping exists**. Verify that **%MOCADIR%/web/resources/locale-mapping.xml** contains a mapping for the language.

   Example: The following code illustrates how the Great Britain English locale XML specification should appear in the mapping.

   ```
   <root>
       <locale name="en-us"
               mapping="US_ENGLISH"
               description="English (US)"/>
       <locale name="en-gb"
               mapping="UK_ENGLISH"
               description="English (UK)"/>
   </root>
   ```

2. **Add the messages**. Add the translated XML message files to **%MOCADIR%/web/resources/messages/<LOCALE_NAME>/**. For example, the Great Britain English XML message files should be stored in **%MOCADIR%/web/resources/messages/en-gb/**.

   Note: Messages are translated from the original U.S. English files which are stored in **%MOCADIR%/web/resources/messages/en-us/*.xml**.

3. **Verify the messages**. Verify that the message XML files have the correct locale **name** attribute specified on the **locale** root element. For example, the following XML message file lists the root locale as **en-gb** for Great Britain English:

   ```
   <locale name="en-gb" version="..." xmlns="..." xmlns:xsi="..."
   xsi:schemaLocation="...">
     <msgs msg-ns="rp.moca.common">
       <msg id="fries">chips</msg>
       <msg id="chips">crisps</msg>
       <msg id="trunk">boot</msg>
       ...
   ```

4. **Build messages**. The procedure to build messages varies depending on the operating system and type of environment.

   a. **To build messages in a development environment**, use the following command to build MOCA and the locales:

      ```
      cd %MOCADIR% && ant web
      ```

      Note: Messages are also built automatically as part of the standard MOCA compilation process; for example, `ant default` or `ant`.

b. **To build messages in a UNIX-based production environment**, build the messages manually using the message build script:

   i. Ensure privileges by using `sudo` to run the UNIX commands.

   ii. Bootstrap the environment (**MOCADIR** must resolve to the MOCA main directory). For more information, see "Starting MOCA" (on page 47).

   iii. Run the message build script:

```
cd $MOCADIR
scripts/locales
```

c. **To build messages in a Windows-based production environment**, build the messages manually using the message build script:

   i. Ensure privileges by running `cmd` as an administrator.

   ii. Bootstrap the environment (**MOCADIR** must resolve to the MOCA main directory).

   iii. Run the message build script:

```
cd %MOCADIR%
scripts/locales.bat
```

Chapter 25. Monitoring and Diagnostics in MOCA

Introduction

Starting with the 2012.2 release, the Monitoring and Diagnostics library has been incorporated into MOCA to use probes to monitor and report information regarding internal MOCA objects. This section describes how Monitoring and Diagnostics works, how Monitoring and Diagnostics is used inside of MOCA, and how to access the information measured and reported by Monitoring and Diagnostics.

What is Monitoring and Diagnostics?

Monitoring and Diagnostics is a Java monitoring library that has a primary focus of providing information about the overall state, performance, and health of Java-based applications. It originally was developed for the 2012.2 release of MOCA at the same time as the development of probes in MOCA, but has since changed into a standalone library that can be used by other applications that are not MOCA based.

At a high level, the Monitoring and Diagnostics library support the following main functions:

- Implementing probes.
- Reporting and viewing probe data.
- Integrating with third-party monitoring systems.

Using the Monitoring and Diagnostics library in MOCA provides the following major benefits:

- Ability to collect information about the server that can be used to determine the health of the application and its sub-components.
- Quicker resolution times when issues are encountered.
- Ability to view potential areas of improvement with the server.

The Monitoring and Diagnostics Probe

A probe is the most basic tool of the Monitoring and Diagnostics library. A probe is a Java class that has one or more attributes that reflect the state of the probe. For example, one type of probe is a timer. A timer has various aggregation attributes, such as minimum, maximum, and mean. These attributes provide an overall summary of the state of a certain aspect of the application about which the probe is measuring timing information. Monitoring and Diagnostics uses Yammer Metrics to implement probes by wrapping Metrics and taking advantage of the JMX registration functionality.

Reporting and Viewing Probe Data

Monitoring and Diagnostics provides the following main methods for accessing probe information:

- programatically
- Java Management Extensions (JMX)
- Monitoring and Diagnostics reporting API

Probe data can be accessed using JMX by using Java monitoring and management tools (such as JVisualVM), using the Jolokia web-service, or by using the third-party monitoring system, Zabbix. The ability to use Java monitoring and management tools are available automatically since probes are always registered with JMX. Also, use of the Jolokia web-service is automatically available since the Jolokia servlet is deployed in MOCA. However using Zabbix and the reporting API require setup. More information about how to use these options to view the probe data is covered in the "How to View the Probe Data" (on page 261) section later in this chapter.

Integrating with Third-Party Monitoring Systems

In addition to the ability to view and report probe data, the Monitoring and Diagnostics library provides a way to integrate with third-party monitoring systems. Currently, there is only one third-party monitoring system that Monitoring and Diagnostics has implemented integration with, Zabbix. Zabbix is an open source, third-party monitoring and data persistence tool. It can be used to view current probe values and also save probe values over time for data aggregation, trend analysis, and alerting.

How to View the Probe Data

Probe data can be viewed in the following way: programatically, using JMX, and using the reporting API. This section provides additional details about how to access the probe and MBean information.

Viewing Probe Data Programatically

The suggested method for accessing probe data programatically is to use the Monitoring and Diagnostics core framework. When accessing probe data both programatically and using JMX, the concept of a `MadName` is used heavily. In general a `MadName` is an object that is used to specify a single probe. Each `MadName` has the attributes that are listed in the following table.

Attribute	Attribute Value
Group	String that corresponds to each specific product of the application. Typically, this is a package name. By default, MOCA uses the value `com.redprairie.moca`.
Type	String used to group related items in a product. MOCA uses component groups for this field such as Jobs, Tasks, or SQL Executions.
Name	String naming the individual probe meant to describe a single attribute of the application, such as executions-successful or executions-errored.
Scope	Optional string that provides the ability to have multiple instances in a type. For example, the Jobs type is further separated into individual job scopes.

Reading probes programatically requires the use of the core Monitoring and Diagnostics objects `MadFactory` and `ProbeReader`.

Using JMX to View Probe Data

As mentioned earlier, accessing probe data using JMX can be accomplished in several different ways (by using Java monitoring and management tools, by web service (Jolokia), or by using Zabbix). At their core, all of these options use JMX.

Similar to how Monitoring and Diagnostics uses `MadName` objects to identify probes, JMX uses `ObjectName` objects to identify individual MBeans. As mentioned earlier, Monitoring and Diagnostics uses Yammer Metrics at its core, which automatically registers internal objects to JMX. Monitoring and Diagnostics uses this functionality to map its `MadName` objects to `ObjectName` objects, which typically looks like the following example:

```
domain;<key-porperty-list>

// Example
com.redprairie.myapp;key1=value1,key2=value2
```

In our case the `MadName` instances match up to `ObjectName` instances in the following way:

```
group;type=<MadName-type>,name=<MadName-name>[,scope=<MadName-scope>]

// Example
com.redprairie.moca;type=Jobs,scope=MyNewJob,name=is-scheduled
```

In the following sections we discuss different ways that JMX is used to retrieve probe data.

Using Java Monitoring and Management Tools to View Probe Data

There are several Java management tools that are available to view JMX MBeans information (and by extension, probe data). The major tools include: JConsole, JVisualVM, and (in JDK versions 7 update 40 and later) Java Mission Control. Despite the asthetic differences between the tools, the process of using the tools is similar. All these tools include the following steps:

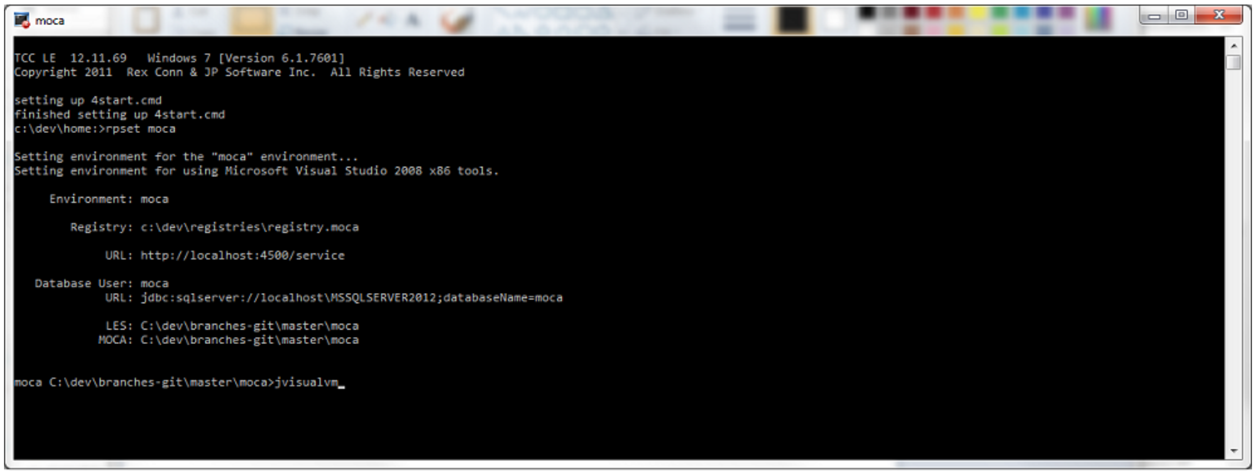
1. Run an executable (jconsole, jvisualvm, or jmc).
2. Specify the Java process to monitor (typically, the MOCA server process as `com.redprairie.moca.server.MocaServerMain`).
3. Navigate to the MBeans tab in the tool's user interface (UI).
4. Navigate through the list of MBeans (typically listed in a hierarchy in order of group, type, scope, and name).

Here's an example using JVisualVM using the general steps.

Start up JVisualVM Using the `jvisualvm` Command

We first start by running the command, `JVisualVM`. This task can be accomplished by using a command prompt or by creating a Windows shortcut.

IMPORTANT: When running this command, you must use **Administrator Mode**. There have been issues in the past connecting to the MBeans server when not using **Administrator Mode**. This means that, if you are using the command prompt you must use the command prompt in**Administrator Mode** or use a shortcut using **Administrator Mode**.

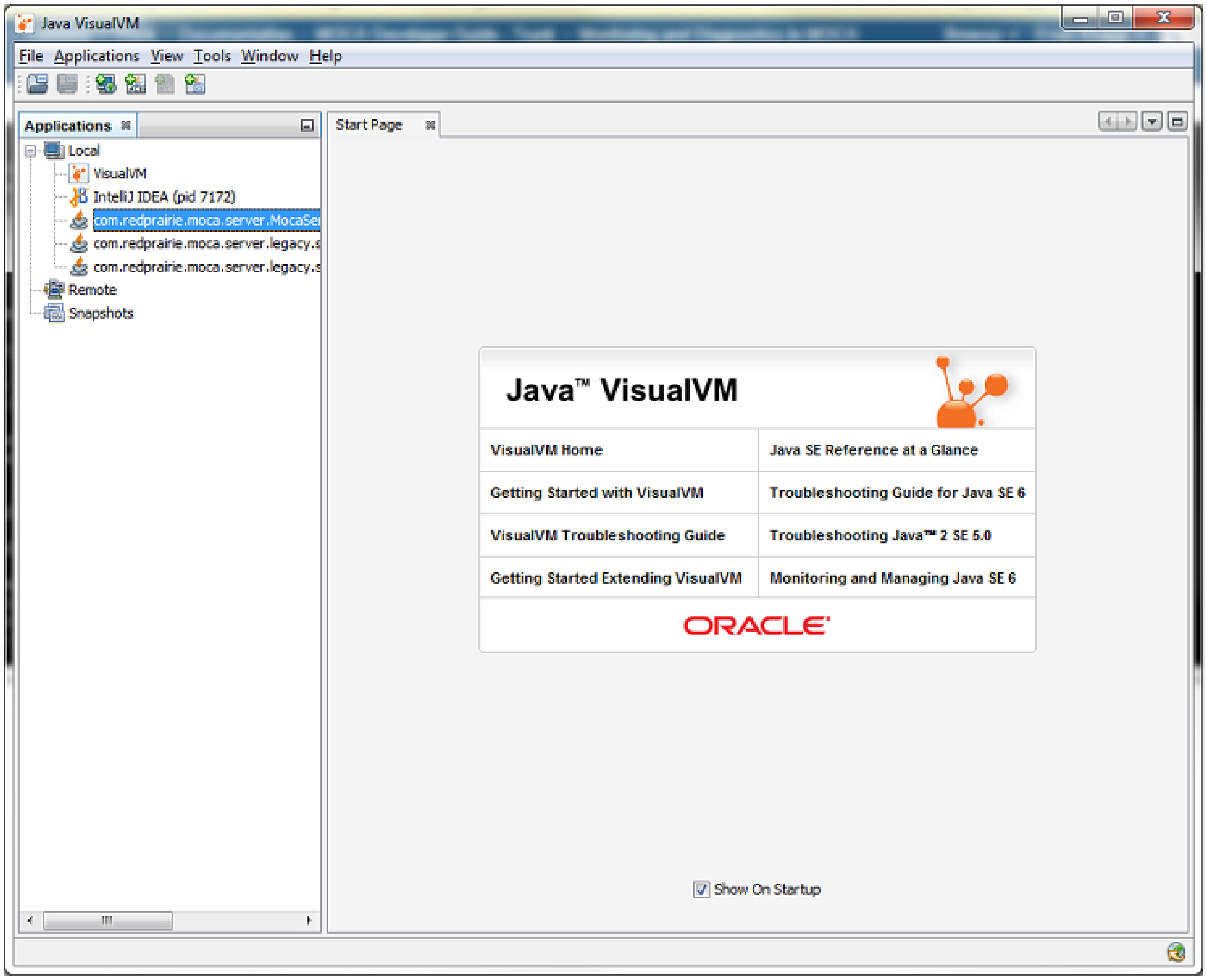


```
TCC LE  12.11.69   Windows 7 [Version 6.1.7601]
Copyright 2011  Rex Conn & JP Software Inc.  All Rights Reserved

setting up 4start.cmd
finished setting up 4start.cmd
c:\dev\home:>rpset moca

Setting environment for the "moca" environment...
Setting environment for using Microsoft Visual Studio 2008 x86 tools.

     Environment: moca

        Registry: c:\dev\registries\registry.moca

             URL: http://localhost:4500/service

   Database User: moca
             URL: jdbc:sqlserver://localhost\MSSQLSERVER2012;databaseName=moca

             LES: C:\dev\branches-git\master\moca
            MOCA: C:\dev\branches-git\master\moca


moca C:\dev\branches-git\master\moca>jvisualvm_
```

Select the MOCA Server Main Process

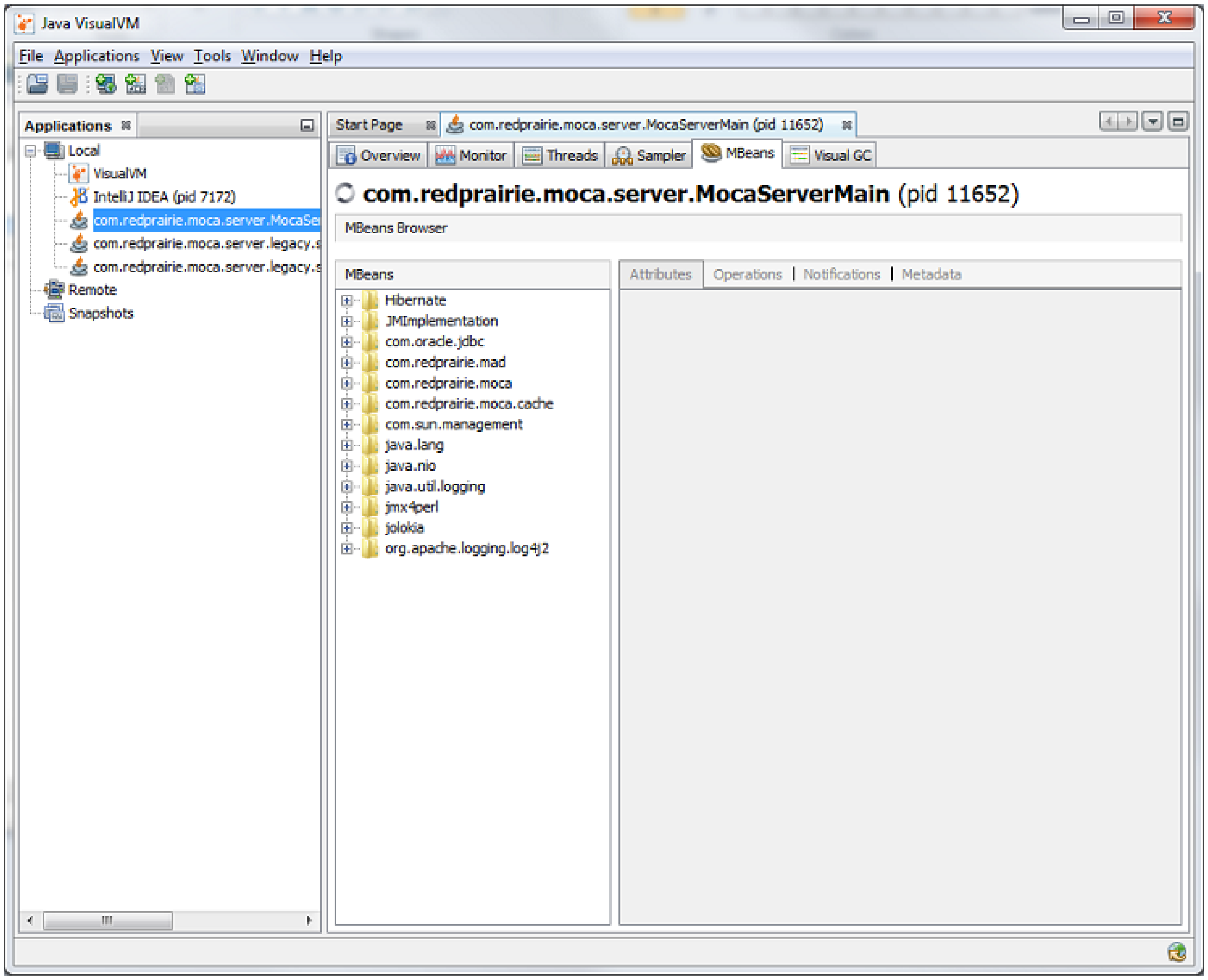
At this point, the window displays all of the processes that are running within the JVM. There are also a number of MOCA processes listed, which list legacy connections (which are MOCA native processes) and then the MOCA Server Main process. Choose the MOCA Server Main Process since it contains the MBean server with which Monitoring and Diagnostics registers its probes.

IMPORTANT: The MOCA Server must be running!


Java VisualVM
File Applications View Tools Window Help
Applications
Local
VisualVM
IntelliJ IDEA (pid 7172)
com.redprairie.moca.server.MocaSe
com.redprairie.moca.server.legacy.s
com.redprairie.moca.server.legacy.s
Remote
Snapshots
Start Page
Java™ VisualVM
VisualVM Home
Java SE Reference at a Glance
Getting Started with VisualVM
Troubleshooting Guide for Java SE 6
VisualVM Troubleshooting Guide
Troubleshooting Java™ 2 SE 5.0
Getting Started Extending VisualVM
Monitoring and Managing Java SE 6
ORACLE
Show On Startup

Navigate to the MBean Tab

At this point, the Overview tab for the MOCA Server Main process is displayed. To view the MBeans for this process, select the MBean tab.


Java VisualVM
File Applications View Tools Window Help
Applications
Local
VisualVM
IntelliJ IDEA (pid 7172)
com.redprairie.moca.server.MocaSe
com.redprairie.moca.server.legacy.s
com.redprairie.moca.server.legacy.s
Remote
Snapshots
Start Page
com.redprairie.moca.server.MocaServerMain (pid 11652)
Overview
Monitor
Threads
Sampler
MBeans
Visual GC

com.redprairie.moca.server.MocaServerMain (pid 11652)

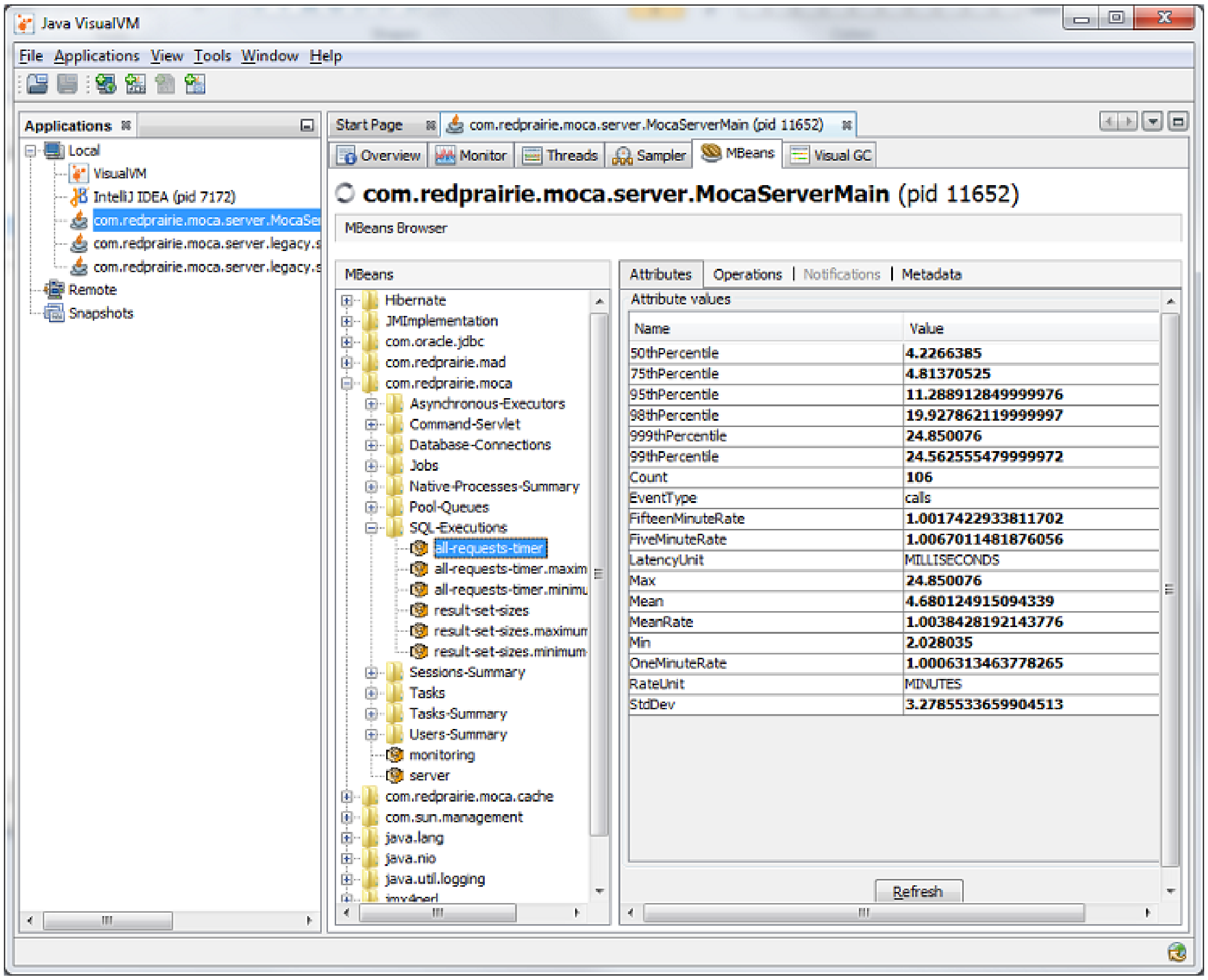
MBeans Browser

MBeans
- Hibernate
- JMImplementation
- com.oracle.jdbc
- com.redprairie.mad
- com.redprairie.moca
- com.redprairie.moca.cache
- com.sun.management
- java.lang
- java.nio
- java.util.logging
- jmx4perl
- jolokia
- org.apache.logging.log4j2

Attributes
Operations
Notifications
Metadata

Navigate the MBeans

At this point, you can navigate through the registered MBeans. The MOCA probes are registered under the `com.redprairie.moca` group name. To view the probe data, you must select an individual MBean. The probe data is displayed in the Attribute tab of the MBean.



Name	Value
50thPercentile	4.2266385
75thPercentile	4.81370525
95thPercentile	11.288912849999976
98thPercentile	19.927862119999997
999thPercentile	24.850076
99thPercentile	24.562555479999972
Count	106
EventType	calls
FifteenMinuteRate	1.0017422933811702
FiveMinuteRate	1.0067011481876056
LatencyUnit	MILLISECONDS
Max	24.850076
Mean	4.680124915094339
MeanRate	1.0038428192143776
Min	2.028035
OneMinuteRate	1.0006313463778265
RateUnit	MINUTES
StdDev	3.2785533659904513

More Information

For more information about Java Monitoring and Management Tools see:

- Monitoring and Managements for the Java Platform using JConsole
- VisualVM
- Java Mission Control

Using Jolokia to View Probe Data

Jolokia is web service based solution for viewing JMX data. Jolokia was chosen as a solution because of the speed of executing batch commands as opposed to simply using Remote Method Invocation (RMI) to update and read MBean information. Jolokia is deployed by MOCA in the JMX web archive (WAR). Because of this fact, whenever the MOCA server is running, any web service calls to the Jolokia endpoint, (`jmx/read`), can be used to view probe data. Jolokia requires the use of the JMX `ObjectName` in its endpoints to specify or filter the probes that it returns in an HTTP Response in the following way:

```
http://<moca-hostname>:<moca-port>/ws/jmx/read/<object-name>
```

Using Zabbix to View Probe Data

Zabbix is an open source third-party monitoring system that includes a server that can consume MBean (probe) data. One benefit of Zabbix is that as a monitoring solution Zabbix provides data persistence for probe information for historical monitoring. This means that Zabbix can be configured to create graphs based on probe information. Zabbix also provides the ability to create notifications based on checks against the collected probe data. For example, Zabbix can send an email notification when a registry change is detected.

While use of Java Monitoring and Managements tools and Jolokia is configured automatically for MOCA, Zabbix requires configuration. This configuration includes setting up Operating System Agents and Java Agents to report back to the Zabbix Server. To make these configuration steps easier Monitoring and Diagnostics provides integration with Zabbix.

Using the Reporting API to View Probe Data

Probe information can also be accessed using the extendable reporting API in Monitoring and Diagnostics. The reporting API is designed to periodically report probe data to various back ends. For example, the reporting API provides a simple reporter that periodically reports probe data to the file system as a CSV file using the Monitoring and Diagnostics CSV Reporter. However, other reporters can be configured such as an XmlReporter that, instead of reporting probe information in CSV format, reports information in XML format.

Monitoring and Diagnostics in MOCA

Staring with the 2012.2 release, MOCA uses the Monitoring and Diagnostics framework to implement MOCA probes to monitor the application. Like Monitoring and Diagnostics, MOCA also uses Java Management Extensions (JMX) to publish custom MBeans that are designed to provide information about the server through custom reports. Additionally, Monitoring and Diagnostics also provides the ability to send custom Notifications to JMX.

This section includes a summary of how to configure Monitoring and Diagnostics in MOCA, a summary of each of the probes and custom MBeans that are implemented in MOCA, and a summary of each of the Monitoring and Diagnostics notifications that MOCA has implemented.

Configuring Monitoring and Diagnostics in MOCA

While Monitoring and Diagnostics is enabled by default for MOCA, the MOCA registry has keys that can be used to configure Monitoring and Diagnostics.

Monitoring and Diagnostics MOCA Registry Settings

```
[SERVER]
mad-probing-enabled=<value>

[MONITORING]
csv-reporter-enabled=<value>
csv-reporter-directory=<value>
csv-reporter-keep-hours=<value>
csv-reporter-archive-directory=<value>
csv-reporter-archive-keep-hours=<value>
csv-reporter-support-default-hours=<value>
```

The following table lists a summary of each key including the valid values and the default values that are used if no registry setting is provided.

Registry Key	Key Summary	Valid Values	Default
`mad-probing-enabled`	Enables or disables Monitoring and Diagnostics in MOCA.	`1`, `true`, or `yes` *(case insensitive)* to enable, any other value to disable	`true`
`csv-reporter-enabled`	Enables or disables the CSV Reporter for use in MOCA.	`true` *(case insensitive)* to enable, any other value to disable	`true`
`csv-reporter-directory`	Specifies the directory to which the CSV Reporter writes data.	A directory name *(environment variables are allowed)*	`$LESDIR/data/csv_probe_data`
`csv-reporter-keep-hours`	Specifies the number of hours to keep CSV probe data for before purging.	An integer value	`72`
`csv-reporter-archive-directory`	Specifies the directory to which the CSV archives probe data.	A directory name *(environment variables are allowed)*	*none*
`csv-reporter-archive-keep-hours`	Specifies the number of hours to keep archived CSV probe data.	An integer value	`168`
`csv-reporter-support-default-hours`	Specifies the maximum age in hours that CSV data can be used in the Support Zip.	An integer value	`24`

Note: The MOCA registry does also include settings for configuring Zabbix.

Probes

The following sections list all of the MOCA probes that currently exist in MOCA grouped into component-specific major sections. Information about each probe includes a short summary of the data that the probe provides, how the probe's data can be used to determine either the health of the application or potential performance issues. The following table is a template that illustrates how the probe information is provided.

Attribute	Attribute Summary
MAD Name	The MadName as used in the application code. It specifies the group, type, name, and scope (if there is one).
Probe Type	The MAD Probe Type. Probe type options include:

Attribute	Attribute Summary
	• Gauge • Counter • Meter • Histogram • Timer • Custom: Typically a manually created MBean with composite return objects such as task or job definitions.
Data	A brief summary of the information that the probe provides.
Application	A brief explanation of how the probe's information can be used to diagnose issues or determine the health of the application.
Update Type	A description of how the probe's data is updated: • **Polling (interval)**: This indicates that the value is updated by being polled on set interval. • **JMX Request**: This indicates that the value is updated by requesting information in real time from an in-memory object. • **Trigger Event**: This is indicates that the value is updated by a triggering event, such as a histogram taking a sample or a timer finishing a timing instance.
Reporting Interval	The interval at which the probe's values are reported using the Reporting API.

Asynchronous Executor Probes

Overview

This set of probes is meant to describe the present state of the active asynchronous executor in MOCA. This set of probes includes three gauges and a custom MBean. For more information on asynchronous executors and how they are used in MOCA, see "Asynchronous Execution" (on page 209).

Active Executors

MAD Name	Group: com.redprairie.moca Type: Asynchronous-Executors Name: active-executors
Probe Type	Gauge (Integer)
Data	The number of active executor threads in the asynchronous executor.
Application	This value could point out that there are too many active threads being executed at the same time causing a performance bottleneck.
Update Type	JMX Request
Reporting Interval	Never

Executors

MAD Name	Group: com.redprairie.moca Type: Asynchronous-Executors Name: executors
Probe Type	Custom
Data	Returns a list of all the currently executing tasks and information about them including the status, age, callable in string form, and thread ID of the task.
Application	The status and age information could be useful for determining if the task has stopped responding or is in a never-ending loop.
Update Type	JMX Request
Reporting Interval	Every 2 minutes

Maximum Executors

MAD Name	Group: com.redprairie.moca Type: Asynchronous-Executors Name: maximum-executors
Probe Type	Gauge (Integer)
Data	The maximum number of concurrently executing threads that are allowed within the executor.
Application	This value could point out a need to change (increase or decrease) the configuration for the maximum number of concurrent threads.
Update Type	JMX Request
Reporting Interval	Never

Queue Callables

MAD Name	Group: com.redprairie.moca Type: Asynchronouse-Executors Name: queue-callables
Probe Type	Gauge (Integer)
Data	The current size of the executors queue of tasks to be executed.
Application	If this value is large it could indicate that a potential backup exists in tasks that are yet to be run by the executor.
Update Type	JMX Request
Reporting Interval	Never

Clustered Asynchronous Executor Probes

Overview

This set of probes is the same as the asynchronous executor probes except that they describe the present state of the active clustered asynchronous executor in MOCA. Like the regular asynchronous executor probes, this set of probes includes three gauges and a custom MBean.

Active Executors

MAD Name	Group: com.redprairie.moca Type: Clustered-Asynchronous-Executors Name: active-executors
Probe Type	Gauge (Integer)
Data	The number of active threads in the clustered asynchronous executor.
Application	This value could indicate that there are too many active threads being executed at the same time causing a performance bottleneck.
Update Type	JMX Request
Reporting Interval	Never

Executors

MAD Name	Group: com.redprairie.moca Type: Clustered-Asynchronous-Executors Name: executors
Probe Type	Custom
Data	Returns a list of all of the currently executing tasks and information about them including the status, age, callable in string form, and thread ID of the task.
Application	The status and age information could be useful for determining if the task has stopped responding or is in a never-ending loop.
Update Type	JMX Request
Reporting Interval	Every 2 minutes

Maximum Executors

MAD Name	Group: com.redprairie.moca Type: Clustered-Asynchronous-Executors Name: maximum-executors
Probe Type	Gauge (Integer)
Data	The maximum number of concurrently executing threads allowed in the clustered executor.
Application	This value could indicate a need to change (increase or decrease) the configuration for the maximum number of concurrent threads.

Update Type	JMX Request
Reporting Interval	Never

Queue Callables

MAD Name	Group: com.redprairie.moca Type: Clustered-Asynchronouse-Executors Name: queue-callables
Probe Type	Gauge (Integer)
Data	The current size of the clustered executor's queue of tasks to be run.
Application	If this value is large it could indicate that there is a potential backup of tasks that are yet to run by the executor.
Update Type	JMX Request
Reporting Interval	Never

Command Servlet Probes

Overview

This set of probes summarizes the interaction with the MOCA server by HTTP request. There are six probes; four gauges, one histogram, and one timer. The histogram tracks the size of an HTTP request query's returned results set sizes. The timer tracks the average time it takes to complete a request to the MOCA server. The gauges track the servlet request queries that take the maximum and minimum amount of time for both the timer and histogram.

All Servlet Requests Timer

MAD Name	Group: com.redprairie.moca Type: Command-Servlet Name: all-requests-timer
Probe Type	Timer (Duration Unit: Milliseconds, Rate Unit: Minutes)
Data	Tracks the amount of time that the MOCA server takes to complete commands and queries provided by web service requests.
Application	This timer indicates the number of commands per minute that the MOCA server is receiving through the MOCA servlet and, by extension, all MOCA clients. This timer also provides a more specific information about the amount of time that it takes to complete commands to the MOCA server. This information could show recent changes or trends in command completion timing that indicates a recently encountered performance issue.
Update Type	Trigger Event (updated every time a command is sent to the MOCA Server)
Reporting Interval	Every 5 minutes

All Servlet Requests Timer Maximum Context

MAD Name	Group: com.redprairie.moca Type: Command-Servlet Name: all-requests-timer.maximum-context
Probe Type	Gauge (String)
Data	The MOCA servlet request query that takes the longest amount of time to complete.
Application	This probe could indicate that a servlet request query might need refactoring or reworking to reduce the time it takes to complete execution.
Update Type	Trigger Event (updated every time the all-requests-timer timer probe encounters a new maximum time value)
Reporting Interval	Never

All Servlet Requests Timer Minimum Context

MAD Name	Group: com.redprairie.moca Type: Command-Servlet Name: all-requests-timer.minimum-context
Probe Type	Gauge (String)
Data	The servlet request query that takes the shortest amount of time to complete.
Application	This probe indicates the servlet request query that takes the least amount of time.
Update Type	Trigger Event (updated every time the all-requests-timer timer probe encounters a new minimum time value)
Reporting Interval	Never

Results Set Size

MAD Name	Group: com.redprairie.moca Type: Command-Servlet Name: result-set-sizes
Probe Type	Histogram
Data	Tracks the number of rows returned in result sets by servlet responses.
Application	This histogram could indicate performance issues with returning very large results sets.
Update Type	Trigger Event (updated with the size of the results set in the response every time a request is made to the MOCA Server)
Reporting Interval	Every 5 minutes

Results Set Size Maximum Context

MAD Name	Group: com.redprairie.moca Type: Command-Servlet Name: result-set-sizes.maximum-context
Probe Type	Gauge (String)
Data	The MOCA servlet request query that returns the largest number of rows in its results set upon completion.
Application	This probe could indicate commands that return large results sets that need to be re-factored.
Update Type	Trigger Event (updated every time the result-set-sizes histogram probe encounters a new maximum size value)
Reporting Interval	Never

Results Set Size Minimum Context

MAD Name	Group: com.redprairie.moca Type: Command-Servlet Name: result-set-sizes.minimum-context
Probe Type	Gauge (String)
Data	The MOCA servlet request query that returns the smallest number of rows in its results set upon completion.
Application	This probe indicates the servlet requests that return small results sets.
Update Type	Trigger Event (updated every time the result-set-sizes histogram probe encounters a new minimum size value)
Reporting Interval	Never

Database Connection Summary Probes

Overview

This set of probes provides summary information about the database connection pool in MOCA. This set of probes Includes three gauges that summarize the current number of connections, the maximum number of possible connections, and the peak number of concurrent connections recorded by the connection pool.

Current Connections

MAD Name	Group: com.redprairie.moca Type: Database-Connections Name: current-connections
Probe Type	Gauge (Integer)
Data	The current number of active database connections within the MOCA application's database connection pool.

Application	If this number is very high it may indicate that the **Maximum Number of Database Connections** needs to be increased.
Update Type	JMX Request
Reporting Interval	Never

Max Connections

MAD Name	Group: com.redprairie.moca Type: Database-Connections Name: max-connections
Probe Type	Gauge (Integer)
Data	The maximum number of database connections possible in the MOCA application's database connection pool. This value is configured by the MOCA registry.
Application	In addition to the current number of database connections, this probe may indicate a need to increase or decrease the maximum number of database connections that are allowed.
Update Type	JMX Request
Reporting Interval	Never

Peak Connections

MAD Name	Group: com.redprairie.moca Type: Database-Connections Name: peak-connections
Probe Type	Gauge (Integer)
Data	The largest number of database connections held concurrently by the MOCA application's database connection pool.
Application	This value may point to the need to increase or decrease the maximum number of database connections that allowed in MOCA.
Update Type	JMX Request
Reporting Interval	Never

Database Connection Pool Probes

Introduction

This set of probes provides more detailed information regarding the database connection pool. It contains nine probes including six gauges, two timers, and one meter. The gauges track the counts of busy, idle, and created connections, and the queue size, the percentage of connections that are busy, the implementation type of the connection pool, and the current size of the pool's queue. The timers track the build time for the connections and the time is takes to perform a request. The meter tracks the frequency of request timeouts.

Connection Creation Timer

MAD Name	Group: com.redprairie.moca Type: Pool-Queues Scope: Connection Name: build-time
Probe Type	Timer (Duration Unit: Milliseconds, Rate Unit: Minutes)
Data	Tracks how long it takes the MOCA application to create a database connection in the application's database connection pool.
Application	If the average time it takes to create a database connection is very high it could indicate that communication issues exist with the database, or there is a database connection shortage.
Update Type	Trigger Event (updated every time a database connection is created)
Reporting Interval	Every 5 minutes

Busy Connection Count

MAD Name	Group: com.redprairie.moca Type: Pool-Queues Scope: Connection Name: busy-count
Probe Type	Gauge (Integer)
Data	Tracks the number of database connections that are running in the application's database connection pool.
Application	If the connection pool busy count is very high or constantly equal to the created connection count it may indicate that there is a database connection shortage.
Update Type	JMX Request
Reporting Interval	Never

Created Connection Count

MAD Name	Group: com.redprairie.moca Type: Pool-Queues Scope: Connection Name: created count
Probe Type	Gauge (Integer)
Data	Keeps track of how many database connections have been created for the application's database connection pool.
Application	Used with the busy count, idle count, and percentage busy probes and can indicate that there are database connection shortages in the pool.
Update Type	JMX Request

Reporting Interval	Never

Idle Connection Count

MAD Name	Group: com.redprairie.moca Type: Pool-Queues Scope: Connection Name: idle-count
Probe Type	Gauge (Integer
Data	Tracks the number of database connections in the application's database connection pool that are idle.
Application	Used with the busy count, idle count, and percentage busy probes and can indicated that there are database connection shortages in the pool.
Update Type	JMX Request
Reporting Interval	Never

Connection Pool Implementation Type

MAD Name	Group: com.redprairie.moca Type: Pool-Queues Scope: Connection Name: implementation-type
Probe Type	Gauge (String)
Data	Lists the pool implementation type that is being used for the database connection pool.
Application	The implementation type of the pool might indicate customization or implementation details to help with diagnosing performance issues.
Update Type	JMX Request
Reporting Interval	Never

Percent of Connections Busy

MAD Name	Group: com.redprairie.moca Type: Pool-Queues Scope: Connection Name: percent-busy
Probe Type	Gauge (Ratio/Long)
Data	Tracks the percentage of database connections in the application's database connection pool that are running.
Application	Like the other integer gauges, this probe can indicate that there is shortage of database connections in the pool and more should be created.

Update Type	JMX Request
Reporting Interval	Never

Connection Pool Queue Size

MAD Name	Group: com.redprairie.moca Type: Pool-Queues Scope: Connection Name: queue-size
Probe Type	Gauge (Integer)
Data	Tracks the number of tasks that are waiting to be executed in the queue of the application's database connection pool.
Application	If this number is large and keeps growing, it may indicate the need for a larger database connection pool.
Update Type	JMX Request
Reporting Interval	Never

Connection Pool Request Timer

MAD Name	Group: com.redprairie.moca Type: Pool-Queues Scope: Connection Name: request-time
Probe Type	Timer (Duration Unit: Milliseconds, Rate Unit: Minutes)
Data	Tracks the amount of time it takes the MOCA application to retrieve a database connection from the application's database connection pool.
Application	If the average time it takes to retrieve a database connection is very high it could indicate that there are communication issues with the database or there is a database connection shortage.
Update Type	Trigger Event (updated every time a Database Connection is retrieved)
Reporting Interval	Every 5 minutes

Connection Pool Timeout Meter

MAD Name	Group: com.redprairie.moca Type: Pool-Queues Scope: Connection Name: request-timeouts
Probe Type	Meter (Rate Unit: Minutes)
Data	Tracks the frequency of timeouts when retrieving a database connection from the application's database connection pool.

Application	If the rate of recent connection request timeouts is very high it could indicate that there has been a recent drop in communication with the database.
Update Type	Trigger Event (updated every time a database connection request times out)
Reporting Interval	Every 5 minutes

Individual Job Probes

Overview

This set of probes summarizes information about individual Jobs. For every job that exists in the MOCA application, there are at least two probes. There is a Boolean gauge stating whether the job is scheduled and a custom MBean providing more in-depth information about the job's configuration. In addition, if the job is scheduled there are an additional seven probes. There are two timers, which keep track of the time it takes to successfully and unsuccessfully execute the job. The remaining five probes are gauges, which track the last execution date, the amount of time that has elapsed since the last execution date, the last execution return status, the next scheduled execution date, and a countdown until the next execution date.

Is Scheduled

MAD Name	Group: com.redprairie.moca Type: Jobs Scope: *<JOB_ID>* Name: is-scheduled
Probe Type	Gauge (Boolean)
Data	Boolean value that indicates whether the job with the **job-id** *JOB_ID* is scheduled.
Application	Enables the user to verify that a job is scheduled or not scheduled.
Update Type	JMX Request
Reporting Interval	Never

Job Configuration

MAD Name	Group: com.redprairie.moca Type: Jobs Scope: *<JOB_ID>* Name: job-configuration
Probe Type	Custom
Data	A custom MBean containing job configuration information including the associated command, whether the command is enabled, the job schedule, and the environment mapping.
Application	This information could indicate potential job configuration issues that are causing performance problems while running the job.
Update Type	JMX Request

Reporting Interval	Never

Errored Executions

MAD Name	Group: com.redprairie.moca Type: Jobs Scope: *<JOB_ID>* Name: executions-errored
Probe Type	Timer (Duration Unit: Milliseconds, Rate Unit: Hours)
Data	Tracks the amount of time it takes for all errored job executions to complete in MOCA.
Application	This timer could indicate a recent trend of an increasing number of errors while executing jobs, or a recent spike in the amount of time it takes to complete jobs that are experiencing errors.
Update Type	Event Trigger (updated every time a job execution returns with an error)
Reporting Interval	Every 30 minutes

Successful Executions

MAD Name	Group: com.redprairie.moca Type: Jobs Scope: *<JOB_ID>* Name: executions-successful
Probe Type	Timer (Duration Unit: Milliseconds, Rate Unit: Hours)
Data	Tracks the amount of time it takes for all successful job executions to complete in MOCA.
Application	This information could indicate the need for jobs to be re-factored if this timer has a very high average time value.
Update Type	Event Trigger (updated every time a job execution returns successfully without an error)
Reporting Interval	Every 30 minutes

Last Execution Date

MAD Name	Group: com.redprairie.moca Type: Jobs Scope: *<JOB_ID>* Name: last-execution-date
Probe Type	Gauge (Date)
Data	The date of the last time this job was run.
Application	This value indicates to the user the last time this job was run, which may not be

	often.
Update Type	JMX Request
Reporting Interval	Never

Last Execution Date Elapsed

MAD Name	Group: com.redprairie.moca Type: Jobs Scope: *<JOB_ID>* Name: last-execution-date-elapsed
Probe Type	Gauge (String)
Data	The amount of time since the job's last execution reported in String form.
Application	This information gives the user an indication of how long it has been since the job's last execution.
Update Type	JMX Request
Reporting Interval	Never

Last Execution Status

MAD Name	Group: com.redprairie.moca Type: Jobs Scope: *<JOB_ID>* Name: last-execution-status
Probe Type	Gauge (Integer)
Data	The status of the last execution of this job.
Application	This information indicates whether this job was successful or errored the last time it was run, and, if it errored, the return status.
Update Type	JMX Request
Reporting Interval	Never

Next Execution Date

MAD Name	Group: com.redprairie.moca Type: Jobs Scope: *<JOB_ID>* Name: next-execution-date
Probe Type	Gauge (Date)
Data	The date of the next scheduled execution of the job.
Application	This probe indicates the date and time that this job is next scheduled to run.

Update Type	JMX Request
Reporting Interval	Never

Next Execution Date Countdown

MAD Name	Group: com.redprairie.moca Type: Jobs Scope: *<JOB_ID>* Name: next-execution-date-countdown
Probe Type	Gauge (String)
Data	The time until the next scheduled execution of the job reported in String form.
Application	This information indicates the date and time, in String form, that this job is next scheduled to run.
Update Type	JMX Request
Reporting Interval	Never

Native Process Summary Probes

Overview

This set of probes summarizes information about the application's native process pool. There are three Integer probes, which provide information about the current number of native processes, the maximum number of allowed native processes at the same time, and the peak number of concurrently running native processes.

Current Processes

MAD Name	Group: com.redprairie.moca Type: Native-Processes-Summary Name: current-processes
Probe Type	Gauge (Integer)
Data	The current number of active processes in the native process pool.
Application	Shows the demand of how many C based components are being requested that need to run using a native process.
Update Type	JMX Request
Reporting Interval	Never

Maximum Processes

MAD Name	Group: com.redprairie.moca Type: Native-Processes-Summary Name: maximum-processes
Probe Type	Gauge (Integer)

Data	The maximum number of processes allowed in the native process pool. This value is configured using the MOCA registry.
Application	If this number is too often measured as the same as the current number of native processes, this could indicate a configuration change needs to be made.
Update Type	JMX Request
Reporting Interval	Never

Minimum Processes

MAD Name	Group: com.redprairie.moca Type: Native-Processes-Summary Name: peak-processes
Probe Type	Gauge (Integer)
Data	The largest number of active processes at the same time in the native process pool.
Application	This number could indicate that the maximum number of processes in the native process pool needs to be increased or decreased.
Update Type	JMX Request
Reporting Interval	Never

Native Process Pool Probes

Introduction

This set of probes provides more detailed information regarding the application's native process pool. It contains nine probes including six gauges, two timers, and one meter. The gauges track the counts of busy, idle, and created processes, and the queue size, the percentage of processes that are busy, the implementation type of the process pool, and the current size of the pool's queue. The timers track the build time for processes and the time it takes to perform a request. The meter tracks the frequency of request timeouts.

Process Creation Timer

MAD Name	Group: com.redprairie.moca Type: Pool-Queues Scope: Native-Process Name: build-time
Probe Type	Timer (Duration Unit: Milliseconds, Rate Unit: Minutes)
Data	Tracks the amount of time it takes the MOCA application to create a native process in the application's native process pool.
Application	If the average time it takes to create a native process is very high it could indicate that there is a native process shortage in the pool.
Update Type	Trigger Event (updated every time a native process is created)
Reporting Interval	Every 5 minutes

Busy Process Count

MAD Name	Group: com.redprairie.moca Type: Pool-Queues Scope: Native-Process Name: busy-count
Probe Type	Gauge (Integer)
Data	Tracks the number of native processes in the application's native process pool that are running.
Application	If the process pool busy count is very high or constantly equal to the created process count, it may indicate a native process shortage.
Update Type	JMX Request
Reporting Interval	Never

Created Process Count

MAD Name	Group: com.redprairie.moca Type: Pool-Queues Scope: Native-Process Name: created-count
Probe Type	Gauge (Integer)
Data	Tracks the number of native processes that have been created for the application's native process pool.
Application	Used with the busy count, idle count, and percentage busy probes, and can indicate native process shortages in the pool.
Update Type	JMX Request
Reporting Interval	Never

Idle Process Count

MAD Name	Group: com.redprairie.moca Type: Pool-Queues Scope: Native-Process Name: idle-count
Probe Type	Gauge (Integer
Data	Tracks the number of native processes in the application's native process pool that are idle.
Application	Used with the busy count, idle count, and percentage busy probes, and can indicate native process shortages in the pool.
Update Type	JMX Request
Reporting Interval	Never

Process Pool Implementation Type

MAD Name	Group: com.redprairie.moca Type: Pool-Queues Scope: Native-Process Name: implementation-type
Probe Type	Gauge (String)
Data	Lists the pool implementation type that is being used for the native process pool.
Application	The implementation type of the pool indicates customization or implementation details to help with diagnosing performance issues.
Update Type	JMX Request
Reporting Interval	Never

Percent of Processes Busy

MAD Name	Group: com.redprairie.moca Type: Pool-Queues Scope: Native-Process Name: percent-busy
Probe Type	Gauge (Ratio/Long)
Data	Tracks the percentage of native processes in the application's native process pool that are running.
Application	Like the other integer gauges, this probe can indicate that there is a shortage of native processes in the pool and more should be created.
Update Type	JMX Request
Reporting Interval	Never

Process Pool Queue Size

MAD Name	Group: com.redprairie.moca Type: Pool-Queues Scope: Native-Process Name: queue-size
Probe Type	Gauge (Integer)
Data	Tracks the number of tasks that are yet to be executed in the queue of the application's native process pool.
Application	If this number is large and keeps growing, it may indicate that the native process pool needs to be larger.
Update Type	JMX Request
Reporting Interval	Never

Process Pool Request Timer

MAD Name	Group: com.redprairie.moca Type: Pool-Queues Scope: Native-Process Name: request-time
Probe Type	Timer (Duration Unit: Milliseconds, Rate Unit: Minutes)
Data	Tracks the amount of time it takes the MOCA application to retrieve a native process from the application's native process pool.
Application	If the average time it takes to retrieve a native process is very high, it could indicate a native process shortage.
Update Type	Trigger Event (updated every time a native process is retrieved)
Reporting Interval	Every 5 minutes

Process Pool Timeout Meter

MAD Name	Group: com.redprairie.moca Type: Pool-Queues Scope: Native-Process Name: request-timeouts
Probe Type	Meter (Rate Unit: Minutes)
Data	Tracks the frequency of timeouts when retrieving a native process from the application's native process pool.
Application	If the rate of recent connection request timeouts is very high, it could indicate a shortage of native processes in the pool or that there is processes in the pool that have stopped responding.
Update Type	Trigger Event (updated every time a native process request times out)
Reporting Interval	Every 5 minutes

SQL Execution Probes

Overview

This set of probes summarizes the performance of queries performed on the database. There are two base probes (a histogram and a timer) and four additional probes, which track the maximum and minimum values of each base probe. The timer probe tracks the amount of time it takes to complete the execution of each SQL query. The histogram keeps track of the result set sizes of each SQL query.

All Requests Timer

MAD Name	Group: com.redprairie.moca Type: SQL-Executions Name: all-requests-timer

Probe Type	Timer (Duration Unit: Milliseconds, Rate Unit: Minutes)
Data	Tracks the amount of time it takes to execute each SQL query given to the MOCA server.
Application	This probe could indicate that there are performance issues caused by SQL queries taking long periods of time to execute. This timer also shows the rate at which SQL queries are being executed per minute on this instance.
Update Type	Event Trigger (updated every time an SQL query is executed by MOCA)
Reporting Interval	Every 5 minutes

All Requests Timer Maximum Context

MAD Name	Group: com.redprairie.moca Type: SQL-Executions Name: all-requests-timer.maximum-context
Probe Type	Gauge (String)
Data	Holds the SQL query that has taken the longest time to execute since the all-requests timer was last reset.
Application	This value could indicate the need to refactor a SQL query because of performance issues.
Update Type	Event Trigger (updated every time there is a new maximum executed SQL query time measured)
Reporting Interval	Never

All Requests Timer Minimum Context

MAD Name	Group: com.redprairie.moca Type: SQL-Executions Name: all-requests-timer.minimum-context
Probe Type	Gauge (String)
Data	Holds the SQL query that has taken the shortest amount of time to execute since the all-requests timer was last reset.
Application	This value shows generally faster running queries.
Update Type	Trigger Event (updated every time there is a new minimum executed SQL query time measured)
Reporting Interval	Never

Results Set Size

MAD Name	Group: com.redprairie.moca Type: SQL-Executions Name: result-set-sizes

Probe Type	Histogram
Data	Holds a collection of SQL query result set sizes sorted by the number of rows returned.
Application	If this histogram has an average result set size that is very high, it could indicate that fact being the cause of performance issues.
Update Type	Trigger Event (updated every time a SQL query is executed and returns a result set with rows in MOCA)
Reporting Interval	Every 5 minutes

Results Set Size Maximum Context

MAD Name	Group: com.redprairie.moca Type: SQL-Executions Name: result-set-sizes
Probe Type	Gauge (String)
Data	Holds the SQL query that returned the largest number of rows in the result set.
Application	This SQL query may need to be refactored or the user should use this query sparingly due to its potential performance issues.
Update Type	Trigger Event (updated every time a returned SQL result set size is greater than the last maximum results set size)
Reporting Interval	Never

Results Set Size Minimum Context

MAD Name	Group: com.redprairie.moca Type: SQL-Executions Name: result-set-sizes.minimum-context
Probe Type	Gauge (String)
Data	Holds the SQL query that returned the smallest number of rows in the result set.
Application	This SQL query generally is the query that takes the shortest amount of processing time when returning results sets.
Update Type	Trigger Event (updated every time a returned SQL query result set size is smaller than the last minimum result set)
Reporting Interval	Never

Session Manager Summary Probes

Overview

This set of probes summarizes information regarding the current state of the MOCA session manager. There are three Integer gauges, which provide the current number of active Sessions in the application, the maximum number of sessions allowed concurrently, and the peak number of concurrent active sessions.

Current Sessions

MAD Name	Group: com.redprairie.moca Type: Sessions-Summary Name: current-sessions
Probe Type	Gauge (Integer)
Data	The current number of active sessions in the MOCA application.
Application	This value could indicate that there are a large number of stale sessions or there is a configuration issue.
Update Type	JMX Request
Reporting Interval	Never

Maximum Sessions

MAD Name	Group: com.redprairie.moca Type: Sessions-Summary Name: maximum-sessions
Probe Type	Gauge (Integer)
Data	The maximum number of active sessions allowed in MOCA concurrently.
Application	This could indicate that there is a configuration issue if this number is equal to the current-sessions gauge on a regular basis.
Update Type	JMX Request
Reporting Interval	Never

Peak Sessions

MAD Name	Group: com.redprairie.moca Type: Sessions-Summary Name: peak-sessions
Probe Type	Gauge (Integer)
Data	The largest number of active sessions running concurrently in MOCA.
Application	This value could indicate that there is a configuration issue if this number is equal to the maximum-sessions value on a regular basis.

Update Type	JMX Request
Reporting Interval	Never

Individual Task Probes

Overview

This set of probes summarizes information about each individual task that exists in the MOCA application. There are three probes provided for each task. There is a gauge, which tracks whether the task is running. There is a counter, which tracks the number of times the associated task has been restarted. There is also a custom MBean, which provides in-depth configuration information about the associated task.

Number of Restarts

MAD Name	Group: com.redprairie.moca Type: Tasks Scope: <TASK_ID> Name: number-restarts
Probe Type	Counter
Data	The number of times the task with the task id <TASK_ID> has been restarted.
Application	A high number of task restarts could indicate that there is a run-time issue.
Update Type	Trigger Event (incremented every time the task with the task id <TASK_ID> is restarted)
Reporting Interval	Every 30 minutes

Running

MAD Name	Group: com.redprairie.moca Type: Tasks Scope: <TASK_ID> Name: running
Probe Type	Gauge (Boolean)
Data	Boolean value indicating whether the task with the task id <TASK_ID> is currently running.
Application	This probe enables the user to determine whether the task with the task id <TASK_ID> is currently running.
Update Type	JMX Request
Reporting Interval	Never

Task Configuration

MAD Name	Group: com.redprairie.moca Type: Tasks

	Scope: <TASK_ID> Name: task-configuration
Probe Type	Custom
Data	This MBean contains summary information regarding the task with the task id <TASK_ID>, such as the command line for the task, and the name, task ID, and type.
Application	If there seems to be a performance issue with the task, the task configuration might help in diagnosing configuration issues.
Update Type	JMX Request
Reporting Interval	Never

Task Manager Probes

Overview

This probe is a custom MBean providing information that summarizes the current state of the task manager, which includes the number and name of the running and stopped tasks.

Summary

MAD Name	Group: com.redprairie.moca Type: Tasks-Summary Name: summary
Probe Type	Custom
Data	This custom MBean includes task summary information such as a list of stopped and running tasks, and the count of total tasks, running tasks, and stopped tasks.
Application	This MBean provides high-level information about the number of tasks that are running or stopped at any specific point in time.
Update Type	JMX Request
Reporting Interval	Never

User Summary Probes

Overview

This set of probes provides summary information about the users that are currently connected to MOCA. There are two probes. One probe tracks the number of unique users that are connected to MOCA, and the other probe tracks the user IDs that are connected to MOCA.

Connected Users

MAD Name	Group: com.redprairie.moca Type: Users-Summary Name: connected-users

Probe Type	Gauge (List of String)
Data	A list of the names of the currently connected users.
Application	Allows the user to see the list of all users that are currently connected to MOCA.
Update Type	JMX Request
Reporting Interval	Never

Unique Connected Users

MAD Name	Group: com.redprairie.moca Type: Users-Summary Name: unique-connected-users
Probe Type	Gauge (Integer)
Data	The number of unique users that are connected to MOCA.
Application	This probe specifies the number of unique users that are currently connected to MOCA.
Update Type	JMX Request
Reporting Interval	Never

Web Services Probes

Overview

This set of probes provides performance and monitoring information about the web services that are being called on the server. Additionally, you can view this information using the MOCA Console. See the "Web Services Usage page" section in the *MOCA Console User Guide*.

Overall web service request performance

MAD Name	Group: com.redprairie.moca Type: Web-Services Name: requests
Probe Type	Timer
Data	Timing performance data that is updated every time a web service request is made.
Application	Allows the user to see the overall performance of web service requests and how frequently they are occurring (rates).
Reporting Interval	5 minutes

Overall web service error count

MAD Name	Group: com.redprairie.moca Type: Web-Services Name: total-error-count

Probe Type	Gauge (Long)
Data	The number of web service requests that have resulted in an error (the HTTP status response is greater than or equal to 400).
Application	Allows the user to see the overall number of errors that occur for web services.
Reporting Interval	Never

Overall web service errors by HTTP status code

This probe is dynamically named, error-responses-<*HTTP Status Code*>, where <*HTTP Status Code*> is the specific HTTP status code; for example, error-responses-404.

MAD Name	Group: com.redprairie.moca Type: Web-Services Name: error-responses-<*HTTP Status Code*>
Probe Type	Meter
Data	Tracks the number and rate of occurrence for a specific HTTP error status code (greater than or equal to 400) for all web services.
Application	Allows the user to see the overall rate of occurrence for a specific HTTP status code.
Reporting Interval	Never

Endpoint-specific web service probes

Endpoint-specific web service probes use a dynamic name for the Scope attribute of the MAD Name to identify the web service endpoint. The dynamic name is <*Endpoint*>-<*HTTP Verb*>, where <*Endpoint*> is the web service endpoint with the slashes (/) replaced by two hyphens (--) because slash is an invalid character for probes, and <*HTTP Verb*> is the standard HTTP method used on the endpoint. For example, an HTTP GET on endpoint admin/tasks/{taskId} results in a probe scope of admin--tasks--{taskId}-GET. The following probes use this notation for the probe scope.

Endpoint-specific web service request performance

MAD Name	Group: com.redprairie.moca Type: Web-Services Scope: <*Endpoint*>-<*HTTP Verb*> Name: requests
Probe Type	Timer
Data	Timing performance data that is updated every time a web service request is made for this specific endpoint.
Application	Allows the user to see the performance of this specific web service endpoint and how frequently requests to it are made.
Reporting Interval	15 minutes

Endpoint-specific web service error count

MAD Name	Group: com.redprairie.moca Type: Web-Services Scope: *<Endpoint>-<HTTP Verb>* Name: total-error-count
Probe Type	Gauge (Long)
Data	The number of web service requests for this specific web service endpoint that have resulted in an error (HTTP status response greater than or equal to 400).
Application	Allows the user to see the overall number of errors that occur for a specific web service.
Reporting Interval	Never

Endpoint-specific web service errors by HTTP status code

This probe is dynamically named, error-responses-*<HTTP Status Code>*, where *<HTTP Status Code>* is the specific HTTP status code; for example, error-responses-404.

MAD Name	Group: com.redprairie.moca Type: Web-Services Scope: *<Endpoint>-<HTTP Verb>* Name: error-responses-*<HTTP Status Code>*
Probe Type	Meter
Data	Tracks the number and rate of occurrence for a specific HTTP error status code (greater than or equal to 400) for a specific web service endpoint.
Application	Allows the user to see the overall rate of occurrence for a specific HTTP status code in a specific web service.
Reporting Interval	Never

The MOCA Monitoring MBean

Overview

The monitoring MBean is a custom MBean designed to provide information through text reports to the user. This MBean does not have any attributes but instead has four MBean operations, which generate text reports that provide information about the state of several components of the MOCA application.

Dump Session Information

This MBean operation generates a summary report of all of the current sessions that exist in MOCA. This report includes a combination of the Session Summary Probes and information about each individual session. The following sample illustrates the output that might result from this operation.

```
Session Count: 0

Max Session Count: 10000

Peak Session Count: 0
```

```
Session-ID: task-SL_IFDCREATOR_0

Thread-ID: 71

Session-Type: TASK

Session-Status: INACTIVE

Start-Time: Mon Dec 17 08:01:32 CST 2012

Last-SQL: select ed.evt_data_seq evt_data_seq, ed.evt_id evt_id,
ed.sys_id sys_id from sl_evt_data ed where ed.evt_stat_cd = :evt_
stat_cd and ed.work_user_id is null and rownum <= 10 order by evt_
data_seq

Last-SQL-Time: Mon Dec 17 08:38:04 CST 2012

Last-Command:

Last-Command-Time:

Last-Script:

Last-Script-Time:

Trace-File: $LESDIR/log/sl_ifdcreator_0.log
```

Dump Database Connection Information

This MBean operation generates a report that summarizes database connection information. The report contains information about each database connection and the number of current database connections to the pool. The following sample report illustrates the output that might result from this operation.

```
Total number of database connections: 2

id: wmd-000005

thread_id: 71

last_sql_dt: Mon Dec 17 08:43:14 CST 2012

last_sql: select TOP 10 ed.evt_data_seq evt_data_seq, ed.evt_id evt_
id, ed.sys_id sys_id from sl_evt_data ed where ed.evt_stat_cd = ? and
ed.work_user_id is null and 1=1 order by evt_data_seq

executions: 253command_path: SQL

id: wmd-000006

thread_id:

last_sql_dt: Mon Dec 17 08:43:13 CST 2012

last_sql: select count wkoinstalled

from poldat_view

where polcod = ?

and polval = ?

and polvar = ?

and rtnum1 = ?
and wh_id = ?

executions: 6202

command_path: DCSjob/job pick release manager->DCSpck/process pick
release->TRIGGER(prepare pick for list process on process pick
release)>DCSlist/prepare picks for list process>SQL
```

Dump Connected User Information

This MBean operation generates a report that summarizes the users that are currently connected to MOCA. The information provided includes in-depth information about each connected user and the number of currently connected users. The following sample report illustrates the output that might result from this operation.

```
Connected-Users:     1

User-ID:             SUPER

Session-ID:          773ff8ee-59e5-47f1-a33c-57754842dca0

Created-Date:        Mon Dec 17 08:47:49 CST 2012

IP:                  0:0:0:0:0:0:0:1

Environment:         {DEVCOD=null}

Last-Accessed-Date:  Mon Dec 17 08:47:49 CST 2012
```

Dump Native Process Information

This MBean operation generates a report that summarizes information about the native process pool and each native process. The information provided includes all of the information gathered by the Native Process Pool Summary Probes and information about each native process. The following sample report illustrates the output that might result from this operation.

```
Native-Processes:        5

Peak-Processes:          5

Maximum-Processes        20

Process-ID:              moca-process-3

Date-Created:            Mon Dec 17 08:02:33 CST 2012

Thread:                  null

Last-Request:            DCSpck.pckProcessPickRelease(null, null,
null, null, null, WMD4, 1)

Last-Request-Date:       Mon Dec 17 08:50:03 CST 2012

Alive:                   true

Keep-Alive:              false

Requests:                384
```

Monitoring and Diagnostics Notifications in MOCA

Starting with the 2012.2 release, MOCA can send asynchronous notifications to Zabbix. Depending on the configuration, these messages could then trigger an alert on a certain condition resulting in an email or Short Message Service (SMS). In the following summary, each of the notifications is uniquely identified by its key. Each notification is summarized with the following information:

- **Key** - The unique key for this notification.
- **Data** - A summary of notification contents.
- **Application** - A summary of how the data contained in the notification could be used.
- **Triggers** - Conditions that must be met for this notification to be sent. If there are multiple triggers listed, they have an `OR` relationship. That is, each of the triggers that are listed results in the respective notification.

MOCA Query Row Limit Met

Key	moca.query-row-limit
Data	Text indicating that the SQL query return row limit is equal to the number that is specified by a user executing a specified SQL statement.
Application	This notification may inform the user that certain SQL statements are causing potential performance issues and need to be refactored and used sparingly.
Trigger 1	MOCA executing a SQL statement that returns a result set size that exceeds the configured limit.

MOCA Native Process Crash

Key	moca.native-process-crash
Data	Text indicating that a C function in a specific library and function name has crashed. The notification also includes the associated task ID, if one exists.
Application	Notifies the user that a native process has crashed.
Trigger 1	MOCA executing a C Function using a Native Process that then results in a remote exception.

MOCA Registry

Key	moca.registry
Data	The MOCA context as a string.
Application	Allows a user to view the current MOCA configuration.
Trigger 1	MOCA initializing the server.

Chapter 26. Extensibility

Overview

Starting with the 9.1.2.0 release, MOCA provides a consistent way to extend database tables in the web-based user interface. The Column Editor, located in the EXTENSIONS menu, is the preferred way to add new user-defined columns to existing database tables. The Column Editor lets you maintain column definitions, which can then be applied to one or more database tables. The editor also allows changes to be exported, so that changes can be tested on one system first and then exported to another system.

Adding Columns

When using the Column Editor to add a column to a database table, the column name is enforced to start with either `uc_` or `vc_` to signify that the column is an extension defined by a user and not part of standard product. When selecting the data type, you may have to enter a length and precision where applicable. Column definitions must specify a compatibility version when defining the column. The compatibility version dictates how these columns are handled on the server side.

IMPORTANT: A server restart is required for all features related to the new user-defined column to be enabled.

A column definition can be created by itself, without applying the column to any tables. A column definition can later be modified to apply it or remove it from any table.

Note: The Column Editor restricts modifying certain core tables. When applying a column to a table, the restricted tables are not listed in the Available Tables column.

New user-defined columns do not contain data. The implementer must make sure that the existing records are populated with data for the columns.

Removing Columns

You can use the Column Editor to remove columns previously added with Column Editor. Removing a column deletes the column definition and removes the column from any tables to which it was applied.

Exporting Changes

You use the Generate Creation Script action from the Column Editor Actions drop-down list to extract all the user-defined column definitions from one application instance for the purpose of migrating the changes to another instance of the same application. A ZIP file is created that contains separate individual SQL scripts for each column definition. These scripts include SQL to create the column definition and apply the column to all of the tables to which the column was applied in the instance.

When extracted, the SQL script files can be included as an input to a rollout to update an instance. If you are producing a rollout or performing a migration and do not need to include all of the columns, you can choose the scripts that you want to include.

IMPORTANT: The creation script only copies new user-defined column definitions into an instance. It does not modify or remove user-defined columns that already exist. Modifying a user-defined column that exists in an instance is not possible without manual intervention. You must manually remove the existing column from the target instance before you can use the script to copy the modified column to the target instance.

Column Data Behavior

Data that is added with this method is automatically integrated into many parts of the system, given that some basic requirements are met. In general, commands should be written in such a way so as to support working with new user-defined columns.

Web Services

Column data is automatically enabled for RESTful web services created with the MOCA web service framework, as well as for Configurable Web Services. These web services can read the data when using GET endpoints, as well as create and modify the data when using POST and PUT endpoints. Note that this only applies to web services created with MOCAWebResource objects. Regular Spring Controllers outside of the WebResource framework cannot make use of these features.

The following is an example call to `/ws/mcs/policyHistories` after applying the `uc_count` column to the `poldat_hst` table:

```
{
    "@type": "ResponseBodyWrapper",
    "data": [
        {
            "actionType": "I",
            "comment": null,
            "modifiedDate": "2016-04-29T19:16:01.000+0000",
            "modifiedUserId": null,
            "newReturnFloat1": null,
            "newReturnFloat2": null,
            "newReturnNum1": 0,
            "newReturnNum2": null,
            "newReturnString1": "raise ems event for login failure",
            "newReturnString2": null,
            "oldReturnFloat1": 0,
            "oldReturnFloat2": 0,
            "oldReturnNum1": 0,
            "oldReturnNum2": 0,
            "oldReturnString1": null,
            "oldReturnString2": null,
            "policyCode": "EMS",
            "policyValue": "MCS-LOGIN-FAILURE",
            "policyVariable": "EVENTS",
            "sortSequence": 0,
            "temporaryDate": null,
            "uc_count": null,
            "warehouseId": null
        }
    ]
}
```

Since this is a RESTful web service created with the MOCA web service framework, it is able to automatically use the new columns with no additional work because the underlying commands for this web service support new columns.

Similarly, Configurable Web Services are able to work with new columns with no changes to the configured action files, given that the registered commands support new columns.

Compatibility

The way that some user-defined column data types are handled depends on the configured compatibility version.

- **Date** - Compatibility version 2 column dates are handled slightly differently than version 1 dates. The input is always expected in an ISO8601 format, and the output is always given in an ISO8601 format. Compatibility version 1 dates are expected as ISO8601, but returned as ISO8601 only if there is some metadata available to identify it as a date, otherwise it is returned in MOCA date format.

  For example, posting a date to the server is always the same. When returning the data with a GET web service, the data may look differently.

  Example of a POST body:

  ```
  {"vc_custom_prop":"2010-06-15T01:22:01.000+0000"}
  ```

  Without any metadata, the dates are returned in the MOCA date format.

  Example response with compatibility version 1 metadata or no metadata:

  ```
  {
      "dateValue" : "2010-06-15T01:22:01.000+0000",
      "vc_custom_prop" : "20100614202201",
      "stringValue" : "20100614202201",
  }
  ```

  Whereas having metadata available, such as when creating a compatibility version 2 column definition, the web service is able to convert the date to ISO8601 format.

  Example response with compatibility version 2 metadata:

  ```
  {
      "dateValue" : "2010-06-15T01:22:01.000+0000",
      "vc_custom_prop" : "2010-06-15T01:22:01.000+0000",
      "stringValue" : "20100614202201",
  }
  ```

- **Boolean** - Compatibility version 1 Boolean columns exhibit legacy MOCA behavior of using the integer values of `1` and `0` to signify `true` and `false` respectively. Compatibility version 2 Boolean columns natively support the Boolean data type. For example, a compatibility version 1 column `vc_test_boolean_1` is returned as an integer, whereas a compatibility version 2 column `vc_test_boolean_2` is returned as a native Boolean for a RESTful web service.

  ```
  {
      ...
      "vc_test_boolean_1": 1,
      "vc_test_boolean_2": true
  ```

```
}
```

This also applies to Configurable Web Services. Configurable Web Service actions return a compatibility version 1 column `vc_test_boolean_1` as an integer, and a compatibility version 2 column `vc_test_boolean_2` is returned as a native Boolean.

Limitations

1. The commands used for the implementation of the web service WebResource Manager must be able to accept the new columns. For list commands this means that the underlying SQL statement must use `select *` syntax instead of selecting individual columns. For other commands it means that they must generally use `@*` in the where clause. If the underlying command is not able to list the new columns or accept them as inputs, then the web service cannot use them. If the underlying commands accept additional columns as inputs and outputs, then the web services built around these commands are able to work with them as well.
2. REFS-based client filtering is currently not supported for additional columns.

Hibernate Entities

Hibernate entities that model tables which have been extended automatically work with the new columns. This is accomplished by automatically instrumenting the entities that have been created with the MOCA Hibernate tools to track the new columns.

The instrumented classes are dropped in the directory $LESDIR/extensibility. After server restart, these classes are automatically added to the classpath so that they are usable by all code that works with MOCA based Hibernate entities. These classes should not be manually added to the MOCA classpath.

Sample Hibernate Entity

```
@Entity
@Table(name = "test_car")
public class TestCar implements Serializable {

    private static final long serialVersionUID =
6176014469248329408L;

    @Id
    @Column(name = "car_id")
    public String getCarId() {
        return car_id;
    }

    public void setCarId(String car_id) {
        this.car_id = car_id;
    }

    @Column(name = "make")
    public String getMake() {
        return make;
    }

    public void setMake(String make) {
        this.make = make;
    }

    @Column(name = "model")
    public String getModel() {
        return model;
    }

    public void setModel(String model) {
        this.model = model;
    }

    @Column(name = "mfr_year")
    public Integer getMfrYear() {
        return mfr_year;
    }

    public void setMfrYear(Integer mfr_year) {
        this.mfr_year = mfr_year;
    }

    /**
     * This formula is a derived column for the model year.
     */
    @Formula(value = "mfr_year + 1")
    public Integer getModelYear() {
        return model_year;
    }

    public void setModelYear(Integer model_year) {
        this.model_year = model_year;
    }

    private String car_id;
    private String make;
    private String model;
    private Integer mfr_year;
    private Integer model_year;
}
```

After extending the test_car table, the entity is automatically extended on server restart with getters and setters for the new columns. The generated methods have a generated signature based on the configured column types and column name. The method names are generated in a way where the column name is stripped of all underscores and then camel cased.

Examples:

- **vc_test_boolean** -> getVcTestString(), setVcTestString(String s)
- **vc_count** -> getVcCount(), setVcCount(Integer i)

Sample customization code using the new columns:

```
TestCar car;
```

```
car.getVcTestString();
```

Limitations

1. Version 9.1.2.0 does not fully support instrumented Boolean columns on Hibernate entities. The API on these entities still expect to use Integer types, matching the legacy MOCA behavior.
2. The instrumented new columns have access type PROPERTY, so they are accessed through the new getter and setter methods.

Limiting Table Extensibility

MOCA provides a mechanism for products to identify tables that are not eligible for extension.

The purpose is twofold:

- To prevent users from adding user-defined columns to core system tables that may be adversely affected by modifications, such as sequence key tables and the component version table.
- To improve the UI presentation in the Column Editor, by displaying the list of tables that are routinely extended.

By default, all product tables are extendable. After it's determined that a table should not be extended, users are prevented from adding new columns to the table on the Column Editor, the back-end web service, and MOCA command calls. Extending a database table using SQL scripts to modify DDLs is highly discouraged but is not prevented.

Approach

MOCA distributes an **ExtendableTableExclusionFilter** singleton bean in the ROOT application context.

```
@Bean

public ExtendableTableExclusionFilter extendableTableExclusionFilter()
{

    return new ExtendableTableExclusionFilter();

}
```

Products can access this bean in their own ROOT Spring configuration classes and add their own tables via the filter's **addExclusions** method. This method is case-insensitive.

```
@Autowired

public void loadExtendableTableExclusionFilter
(ExtendableTableExclusionFilter filter) {

    // Prevent users from customizing base system tables

    filter.addExclusions("COMP_VER", "CUST_COL_DEFINITION", "INDEX_
HISTORY", "JGROUPSPING", "JOB_DEFINITION",

             "JOB_ENV_DEFINITION", "MOCA_DATASET", "MOCA_
DBVERSION", "TASK_DEFINITION", "TASK_ENV_DEFINITION");

}
```

Viewing Extendable Tables

Use this MOCA command to view the system's extendable tables: `list extendable tables`