

Jolokia - Reference Documentation

Version 1.0.3

Copyright © 2011 Roland Huß

1. Introduction	1
2. Architecture	2
2.1. Agent mode	2
2.2. Proxy Mode	2
3. Agents	4
3.1. JEE Agent (WAR)	5
3.1.1. Installation and Configuration	5
3.1.2. Security Setup	6
3.1.3. Programmatic usage of the Jolokia agent servlet	7
3.2. OSGi Agents	8
3.2.1. jolokia-osgi.jar	8
3.2.2. Running on Glassfish v3 upwards	10
3.2.3. jolokia-osgi-bundle.jar	11
3.2.4. Programmatic servlet registration	11
3.2.5. Restrictor service	12
3.3. Mule Agent	12
3.4. JVM Agent	14
3.4.1. Jolokia as JVM Agent	14
3.4.2. Attaching a Jolokia agent on the fly	15
4. Security	19
4.1. Policy based security	19
4.1.1. IP based restrictions	19
4.1.2. Commands	19
4.1.3. Allow and deny access to certain MBeans	20
4.1.4. HTTP method restrictions	20
4.1.5. Cross-Origin Resource Sharing (CORS) restrictions	21
4.1.6. Example for a security policy	21
4.1.7. Policy Location	22
4.2. Jolokia Restrictors	22
5. Proxy Mode	24
5.1. Limitations of proxy mode	24
6. Jolokia Protocol	25
6.1. Requests and Responses	25
6.1.1. GET requests	25
6.1.2. POST requests	27
6.1.3. Responses	28
6.1.4. Pathes	29
6.2. Jolokia operations	30
6.2.1. Reading attributes (read)	30
6.2.2. Writing attributes (write)	33
6.2.3. Executing JMX operations (exec)	35
6.2.4. Searching MBeans (search)	37
6.2.5. Listing MBeans (list)	38
6.2.6. Getting the agent version (version)	41
6.3. Processing parameters	42
6.4. Object serialization	43
6.4.1. Response value serialization	43
6.4.2. Request parameter serialization	44
6.4.3. Jolokia and MXBeans	46
6.5. Tracking historical values	47
6.6. Proxy requests	47

6.7. Jolokia protocol versions	47
7. Jolokia MBeans	49
7.1. Configuration MBean	49
7.1.1. Debugging	49
7.1.2. History store	49
7.2. Server Handler	50
8. Clients	51
8.1. Javascript Client Library	51
8.1.1. Installation	51
8.1.2. Usage	52
8.1.3. Simple API	55
8.1.4. Maven integration	57
8.2. Java Client Library	59
8.2.1. Tutorial	59
8.2.2. J4pClient	60
8.2.3. Request types	63
8.2.4. Exceptions	65
9. Tools	66
9.1. Jmx4Perl	66
9.2. Jolokia Roo Addon	66

Chapter 1. Introduction

JMX (Java Management Extensions) is *the* standard management solution in the Java world. Since JDK 1.5 it is available in every Java Virtual Machine and especially JEE application servers use JMX for their management business.

I love JMX. It is a well crafted specification, created in times where other concepts like EJBs failed spectacularly. Even more than ten years after its incubation it is still the one-and-only when it comes to management in the Java world. Especially the various levels of sophistications for implementing MBeans, starting with dead simple *Standard MBeans* and ending in very flexible Open MBeans and MXBeans, are impressive.

However, some of the advanced JMX concepts didn't really appeal to the public and are now effectively obsolete. Add-on standards like [JSR-77](#) didn't received the adoption level they deserved. And then there is [JSR-160](#), JMX remoting. This specification is designed for ease of usage and has the ambition to transparently hide the technical details behind the remote communication so that it makes (nearly) no difference, whether MBeans are invoked locally or remotely. Unfortunately, the underlying transport protocol (RMI) and programming model is very Java centric and is not usable outside the Java world.

This is where Jolokia steps in. It is an agent based approach, living side by side with JSR-160, but uses the much more open HTTP for its transport business where the data payload is serialized in JSON. This opens a whole new world for different, non-Java clients. Beside this protocol switch, Jolokia provides new features for JMX remoting, which are not available in JSR-160 connectors: Bulk requests allow for multiple JMX operations with a single remote server roundtrip. A fine grained security mechanism can restrict the JMX access on specific JMX operations. Other features like the JSR-160 proxy mode or history tracking are specific to Jolokia, too.

This reference manual explains the details of Jolokia. After an overview of Jolokia's architecture in Chapter 2, *Architecture*, installation and configuration of the various Jolokia agents are described in Chapter 3, *Agents*. Jolokia's security policy mechanism (Chapter 4, *Security*) and proxy mode (Chapter 5, *Proxy Mode*) are covered in the following chapters. For implementors of Jolokia client bindings the protocol definition is probably the most interesting part (Chapter 6, *Jolokia Protocol*). Jolokia itself comes with the preregistered MBeans listed in Chapter 7, *Jolokia MBeans*. The available client bindings are described in Chapter 8, *Clients*.

Chapter 2. Architecture

The architecture of Jolokia is quite different to that of JSR-160 connectors. One of the most striking difference is Jolokia's typeless approach.

JSR-160, released in 2003, has a different design goal than Jolokia. It is a specification with which a client can transparently invoke MBean calls, regardless whether the MBean resides within a local or remote MBeanServer. This provides a good deal of comfort for Java clients of this API, but it is also dangerous *because* it hides the remoteness of JMX calls. There are several subtle issues, performance being one of them. It *does* matter whether a call is invoked locally or remotely. A caller should at least be aware what happens and what the consequences are. On the other side, there are message passing models which include remoting explicitly, so that the caller *knows* from the programming model that she is calling a potentially expensive remote call. This is probably the main reason why RMI (the default protocol stack of JSR-160 connectors) lost market share to more explicit remote protocols.

One problem with JSR-160 is its implicit reliance on RMI and its requirement for a complete (Java) object serialization mechanism for passing management information over the wire. This closes the door for all environments which are not Java (or more precisely, JVM) aware. Jolokia uses a typeless approach, where some sort of lightweight serialization to JSON is used (in both directions, but a bit *asymmetrically* in its capabilities). Of course this approach has some drawbacks, too, but also quite some advantages. At least it is unique in the JMX world ;-).

2.1. Agent mode

Figure 2.1, “Jolokia architecture” illustrates the environment in which Jolokia operates. The agent exports on the frontside a JSON based protocol over HTTP that gets bridged to invocation of local JMX MBeans. It lives outside the JSR-160 space and hence requires a different setup. Various techniques are available for exporting its protocol via HTTP. The most prominent being to put the agent into a servlet container. This can be a lightweight one like Tomcat or Jetty or a full-blown JEE Server. Since it acts like a usual web application the deployment of the agent is well understood and should pose no entry barrier for any developer who has ever dealt with Java web applications.

Figure 2.1. Jolokia architecture

But there are more options. Specialized agents are able to use an OSGi HttpService or come with an embedded Jetty-Server in case of the Mule agent. The JVM agent uses the HTTP-Server included with every Oracle JVM 6 and can be attached dynamically to any running Java process. Agents are described in detail in Chapter 3, *Agents*.

Jolokia can be also integrated into one's own applications very easily. The `joloki-core` library (which comes bundled as a jar), includes a servlet which can be easily added to a custom application. Section 3.1.3, “Programmatic usage of the Jolokia agent servlet” contains more information about this.

2.2. Proxy Mode

Proxy mode is a solution for when when it is impossible to deploy the Jolokia agent on the target platform. For this mode, the only prerequisite for accessing the target server is a JSR-160 connection. Most of the time this happens for political reasons, where it is simply not allowed to deploy an extra piece of software or where doing so requires a lengthy approval process. Another reason could be that the target server already exports JMX via JSR-160 and you want to avoid the extra step of deploying the agent.

A dedicated proxy servlet server is needed for hosting `jolokia.war`, which by default supports both the *agent mode* and the *proxy mode*. A lightweight container like Tomcat or Jetty is a perfect choice for this kind of setup.

Figure Figure 2.2, “Jolokia as JMX Proxy” describes a typical setup for the proxy mode. A client sends a usual Jolokia request containing an extra section for specifying the target which should be queried. All routing information is contained in the request itself so that the proxy can act universally without the need of a specific configuration.

Figure 2.2. Jolokia as JMX Proxy

Having said all that, the proxy mode has some limitations which are listed in Chapter 5, *Proxy Mode* .

To summarize, the proxy mode should be used only when required. The agent servlet on its own is more powerful than the proxy mode since it eliminates an additional layer adding to the overall complexity and performance. Also, some features like merging of MBeanServers are not available in the proxy mode.

Chapter 3. Agents

Jolokia is an agent based approach to JMX, which requires that clients install an extra piece of software, the so-called *agent*. This software either needs to be deployed on the target server which should be accessed via remote JMX (Section 2.1, “Agent mode”), or it can be installed on a dedicated proxy server (Section 2.2, “Proxy Mode”). For both operational modes, there are four different kind of agents¹.

Webarchive (War) agent

This agent is packaged as a JEE Webarchive (War). It is the standard installation artifact for Java webapplications and probably one of the best known deployment formats. Jolokia ships with a war-agent which can be deployed like any other web application. This agent has been tested on many JEE servers, from well-known market leaders to rarer species.

OSGi agent

[OSGi](#) is a middleware specification focusing on modularity and a well defined dynamic lifecycle². The Jolokia OSGi agent bundles comes in two flavors: a minimal one with a dependency on a running [OSGi HttpService](#), and a all-in-one bundle including an embedded *HttpService* implementation (which is exported, too). The former is the recommended, puristic solution, the later is provided for a quick startup for initial testing the OSGi agent (but should be replaced with the minimal bundle for production setups).

Mule agent

[Mule](#) is one of the leading Open Source Enterprise Service Busses³ (ESB). It provides a management API into which a dedicated Jolokia agent plugs in nicely. This agent includes an embedded Jetty for providing JMX HTTP access.

JVM agent

Starting with Java 6 the JDK provided by Oracle contains a lightweight HTTP-Server which is used e.g. for the reference WebService stack implementation included in Java 6. Using the Java-agent API (normally used by profilers and other development tools requiring the instrumentation during the class loading phase), the JVM 6 Jolokia agent is the most generic one. It is able to instrument *any* Java application running on a Oracle JDK 6⁴. This Jolokia agent variant is fully featured, however tends to be a bit slow since the provided HTTP-Server is not optimized for performance. However it is useful for servers like Hadoop or Teracotta, which do not provide convenient hooks for an HTTP-exporting agent on their own.

¹ Although the proxy mode is available for all four agents, you are normally free to setup the proxy environment. The recommendation here is the war-agent for which very lightweight servlet container exists. Tomcat or Jetty are both a perfect choice for a Jolokia proxy server.

² Of course, there is much more to OSGi, a platform and programming model which I *really* like. This is my personal pet agent, so to speak ;-).

³ [What is the proper plural form of "bus"?](#)

⁴ You could even instrument a JEE application server this way, however this is not recommended.

3.1. JEE Agent (WAR)

3.1.1. Installation and Configuration

The WAR agent is the most popular variant, and can be deployed in a servlet container just like any other JEE web application.

Tomcat example

A simple example for deploying the agent on Tomcat can be found in the Jolokia [quickstart](#).

Often, installation is simply a matter of copying the agent WAR to a deployment directory. On other platforms an administrative Web GUI or a command line tool need to be used for deployment. Providing detailed installation instructions for every servlet container is out of scope for this document.

The servlet itself can be configured with `init-param` declarations within the servlet definition in `WEB-INF/web.xml`. The known parameters are described in Table 3.1, “Servlet init parameters”. The stock agent needs to be repackaged, though, in order to modify the internal `web.xml`.

Table 3.1. Servlet init parameters

Parameter	Description	Example
<code>dispatcherClasses</code>	Classnames (comma separated) of <code>RequestDispatcher</code> used in addition to the <code>LocalRequestDispatcher</code> . Dispatchers are a technique used by the JSR-160 proxy to dispatch (or 'route') a request to a different destination.	<code>org.jolokia.jsr160.Jsr160RequestDispatcher</code> of (this is the dispatcher for the JSR-160 proxy)
<code>policyLocation</code>	Location of the policy file to use. This is either a URL which can read from (like a <code>file:</code> or <code>http:</code> URL) or with the special protocol <code>classpath:</code> which is used for looking up the policy file in the web application's classpath. See Section 4.1.7, “Policy Location” for details about this parameter.	<code>file:///home/jolokia/jolokia-access.xml</code> for a file based access to the policy file. Default is <code>classpath:/jolokia-access.xml</code>
<code>debug</code>	Debugging state after startup. Can be changed via the config MBean during runtime.	Default: <code>false</code>
<code>historyMaxEntries</code>	Entries to keep in the history. Can be changed at runtime via the config MBean.	Default: <code>10</code>

Parameter	Description	Example
<code>debugMaxEntries</code>	Maximum number of entries to keep in the local debug history (if enabled). Can be changed via the config MBean at runtime.	Default: 100
<code>maxDepth</code>	Maximum depth when traversing bean properties. If set to 0, depth checking is disabled	Default: 15
<code>maxCollectionSize</code>	Maximum size of collections returned when serializing to JSON. When set to 0, collections are never truncated.	Default: 1000
<code>maxObjects</code>	Maximum number of objects which are traversed when serializing a single response. Use this as an airbag to avoid boosting your memory and network traffic. Nevertheless, when set to 0 no limit is imposed.	Default: 0
<code>mbeanQualifier</code>	Qualifier to add to the <code>ObjectName</code> of Jolokia's own MBeans. This can become necessary if more than one agent is active within a servlet container. This qualifier is added to the <code>ObjectName</code> of this agent with a comma. For example a <code>mbeanQualifier</code> with the value <code>qualifier=own</code> will result in Jolokia server handler MBean with the name <code>jolokia:type=ServerHandler,qualifier=own</code>	
<code>mimeType</code>	MIME to use for the JSON responses	Default: <code>text/plain</code>

3.1.2. Security Setup

In order use JEE security within the war, some extrat configuration steps are required within `web.xml`.

Using jmx4perl's `jolokia` tool

[jmx4perl](#) comes with a nice command line utility called [jolokia](#) which allows for an easy setup of security within a given `jolokia.war`. See Section 9.1, "Jmx4Perl" for more details.

There is a commented section which can serve as an example. All current client libraries are able to use BASIC HTTP authentication with user and password. The `<login-config>` should be set accordingly. The `<security-constraint>` specifies the URL pattern (which is in the default setup specify all resources provided by the Jolokia servlet) and a role name which is used to find the proper authentication credentials. This role must be referenced outside the agent WAR within the servlet container, e.g. for Tomcat the role definition can be found in `$TOMCAT/config/tomcat-users.xml`.

3.1.3. Programmatic usage of the Jolokia agent servlet

The Jolokia agent servlet can be integrated into one's own web-applications as well. Simply add a servlet with the servlet class `org.jolokia.http.AgentServlet` to your own `web.xml`. The following example maps the agent to the context `/jolokia`:

```
<servlet>
  <servlet-name>jolokia-agent</servlet-name>
  <servlet-class>org.jolokia.http.AgentServlet</servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
  <servlet-name>jolokia-agent</servlet-name>
  <url-pattern>/jolokia/*</url-pattern>
</servlet-mapping>
```

Of course, any init parameter as described in Table 3.1, “Servlet init parameters” can be used here as well.

In order for this servlet definition to find the referenced Java class, the JAR `jolokia-core.jar` must be included. This jar can be found in [Jolokia's maven respository](#). Maven users will can declare a dependency on this jar artifact:

```
<project>
  <!-- .... -->
  <dependencies>
    <dependency>
      <groupId>org.jolokia</groupId>
      <artifactId>jolokia-core</artifactId>
      <version>0.90</version>
    </dependency>
  </dependencies>

  <!-- Maven repository hosting Jolokia -->
  <repositories>
    <repository>
      <id>labs-consol-release</id>
      <name>ConSol* Labs Repository (Releases)</name>
      <url>http://labs.consol.de/maven/repository</url>
    </repository>
  </repositories>
</project>
```

The `org.jolokia.http.Agent` can be subclassed, too in order to provide a custom restrictor or a custom log handler. See Section 4.2, “Jolokia Restrictors” for details.⁵

Also, multiple Jolokia agents can be deployed in the same JVM without problem. However, since the agent deploys some Jolokia-specific MBeans on the single `PlatformMBeansServer`, for multi-agent deployments it is important to use the `mbeanQualifier` init parameter to distinguish multiple Jolokia MBeans by adding an extra property to those MBeans' names. This also needs to be done if multiple webapps containing Jolokia agents are deployed on the same JEE server.

3.2. OSGi Agents

There are several free implementations available of OSGi `HttpService`. This bundle has been tested with the [Pax Web](#) and [Apache Felix](#) `HttpService`, both of which come with an embedded Jetty as servlet container by default.

Jolokia agents are also available as [OSGi](#) bundles. There are two flavors of this agent: A nearly bare agent `jolokia-osgi.jar` declaring all its package dependencies as imports in its Manifest and an all-in-one bundle `jolokia-osgi-bundle.jar` with minimal dependencies. The pure bundle fits best with the OSGi philosophy and is hence the recommended bundle. The all-in-one monster is good for a quick start since normally no additional bundles are required.

3.2.1. jolokia-osgi.jar

This bundle depends mostly on a running [OSGi HttpService](#) which it uses for registering the agent servlet.

All package imports of this bundle are listed in Table 3.2, “Package Imports of `jolokia-osgi.jar` (SB: exported by system bundle)”. Note that the `org.osgi.framework.*` and `javax.*` packages are typically exported by the system bundle, so no extra installation effort is required here. Whether the `org.osgi.service.*` interfaces are available depends on your OSGi container. If they are not provided, they can be easily fetched and installed from e.g. [maven central](#). Often the `LogService` interface is exported out of the box, but not the `HttpService`. You will notice any missing package dependency during the resolve phase while installing `jolokia-osgi.jar`.

Table 3.2. Package Imports of `jolokia-osgi.jar` (SB: exported by system bundle)

Package	SB	Package	SB	Package	SB	Package	SB
<code>org.osgi.framework</code>	X	<code>javax.servlet</code>		<code>org.w3c.dom</code>	X	<code>javax.management</code>	X
<code>org.osgi.service.http</code>		<code>javax.servlet.http</code>		<code>org.xml.sax</code>	X	<code>javax.management.openmbean</code>	X
<code>org.osgi.service.log</code>		<code>javax.naming</code>	X	<code>javax.xml.parsers</code>	X	<code>javax.management.remote</code>	X
<code>org.osgi.util.tracker</code>	X						

This agent bundle consumes two services by default: As stated above, an `org.osgi.service.http.HttpService` which is used to register (deregister) the Jolokia agent as a servlet under the context `/jolokia` by default as soon as the `HttpService` becomes available

⁵ Replace `org.jolokia.osgi.http.AgentServlet` with `org.jolokia.http.AgentServlet` to use the servlet in a non-OSGi environment.

(unavailable). Secondly, an `org.osgi.service.log.LogService` is used for logging, if available. If such a service is not registered, the Jolokia bundle uses the standard `HttpServlet.log()` method for its logging needs.

The Jolokia OSGi bundle can be configured via the properties which typically can be configured in a global configuration file of the OSGi container. All properties start with the prefix `org.jolokia` and are listed in Table 3.3, “Jolokia Bundle Properties”. They are mostly the same as the `init-param` options for a Jolokia servlet when used in a JEE WAR artifact.

Table 3.3. Jolokia Bundle Properties

Property	Default	Description
<code>org.jolokia.user</code>		User used for authentication with HTTP Basic Authentication. If not given, no authentication is used.
<code>org.jolokia.password</code>		Password used for authentication with HTTP Basic Authentication.
<code>org.jolokia.agentContext</code>	<code>/jolokia</code>	Context path of the agent servlet
<code>org.jolokia.dispatcherClasses</code>		Class names (comma separated) of request dispatchers used in addition to the <code>LocalRequestDispatcher</code> . E.g using a value of <code>org.jolokia.jsr160.Jsr160RequestDispatcher</code> allows the agent to play the role of a JSR-160 proxy.
<code>org.jolokia.debug</code>	<code>false</code>	Debugging state after startup. This can be changed via the <code>Config MBean (jolokia:type=Config)</code> at runtime
<code>org.jolokia.debugMaxEntries</code>	<code>100</code>	Maximum number of entries to keep in the local debug history if switched on. This can be changed via the <code>config MBean</code> at runtime.
<code>org.jolokia.maxDepth</code>	<code>0</code>	Maximum depth when traversing bean properties. If set to 0, depth checking is disabled
<code>org.jolokia.maxCollectionSize</code>	<code>0</code>	Maximum size of collections returned when serializing to JSON. When set to 0, collections are not truncated.

Property	Default	Description
<code>org.jolokia.maxObjects</code>	0	Maximum number of objects which are traversed when serializing a single response. Use this as an airbag to avoid boosting your memory and network traffic. Nevertheless, when set to 0 no limit is imposed.
<code>org.jolokia.historyMaxEntries</code>	10	Number of entries to keep in the history. This can be changed at runtime via the Jolokia config MBean.
<code>org.jolokia.listenForHttpService</code>	true	If <code>true</code> the bundle listens for an OSGi <code>HttpService</code> and if available registers an agent servlet to it.
<code>org.jolokia.httpServiceFilter</code>		Can be any valid OSGi filter for locating a <code>HttpService</code> which is used to expose the Jolokia servlet. The syntax is that used by the <code>org.osgi.framework.Filter</code> which is in turn a RFC 1960 based filter . The use of this property is described in Section 3.2.2, "Running on Glassfish v3 upwards"
<code>org.jolokia.useRestrictorService</code>	false	If <code>true</code> the Jolokia agent will use any <code>org.jolokia.restrictor.RestrictorService</code> for applying access restrictions. If this option is <code>false</code> the standard method of looking up a security policy file is used, as described in Section 4.1, "Policy based security".

This bundle also exports the service `org.jolokia.osgi.servlet.JolokiaContext` which can be used to obtain context information of the registered agent like the context path under which this servlet can be reached. Additionally, it exports `org.osgi.service.http.HttpContext`, which is used for authentication. Note that this service is only available when the agent servlet is active (i.e. when an `HttpService` is registered).

3.2.2. Running on Glassfish v3 upwards

You have a couple of choices when running jolokia on Glassfish v3 and up, since Glassfish is a both a fully fledged JEE container and an OSGi container. If you choose to run the Section 3.1, “JEE Agent (WAR)” then it is completely straight forward just deploy the war in the normal way. If you choose to deploy the Section 3.2, “OSGi Agents” then you will need to configure the `org.jolokia.httpServiceFilter` option with a filter to select either the Admin `HttpService` (4848 by default) or the Default `HttpService` which is where WAR files are deployed to.

In Glassfish 3.1.2 the OSGi bundle configuration is done in `glassfish/conf/osgi.properties` in version's prior to this the configuration is by default in `glassfish/osgi/felix/conf/config.properties` or if you are using Equinox `glassfish/osgi/equinox/configuration/config.ini`

```
# Restrict the jolokia http service selection to the admin host
org.jolokia.httpServiceFilter=(VirtualServer=__asadmin)
# Or alternatively to the normal http service use : (VirtualServer=server)
```

Deploying the bundle can be either be done by copying the `jolokia-osgi.jar` into the domain `glassfish/domains/<domain>/autodeploy/bundles` directory or it can be added to all instances by copying the jar to `glassfish/modules/autostart`

By default the agent will be available on `http://localhost:<port>/osgi/jolokia` rather than `http://localhost:<port>/jolokia` as with WAR deployment.

3.2.3. jolokia-osgi-bundle.jar

The all-in-one bundle includes an implementation of `org.osgi.service.http.HttpService`, i.e. the [Felix implementation](#). The `HttpService` will be registered as OSGi service during startup, so it is available for other bundles as well. The only package import requirement for this bundle is `org.osgi.service.LogService`, since the Felix Webservice requires this during startup. As mentioned above, normally the `LogService` interface gets exported by default in the standard containers, but if not, you need to install it e.g. from the OSGi [compendium](#) definitions.

This bundle can be configured the same way as the pure bundle as described in Section 3.2.1, “jolokia-osgi.jar”. Additionally, the embedded Felix `HttpService` can be configured as described in its [documentation](#). e.g. setting the port to 9090 instead of the default port 8080, a property `org.osgi.service.http.port=9090` needs to be set. This might be useful, if this bundle is used within containers which already occupy the default port (Glassfish, Eclipse Virgo) but don't expose an OSGi `HttpService`.

3.2.4. Programmatic servlet registration

It is also possible to register the Jolokia agent servlet manually instead of relying of the OSGi bundle activator which comes with the agents. For this use case `jolokia-osgi.jar` should be used. This bundle exports the package `org.jolokia.osgi.servlet` which includes the servlet class `JolokiaServlet`. This class has three constructors: A default constructor without arguments, one with a single `BundleContext` argument and finally one with an additional `Restrictor` (see Section 4.2, “Jolokia Restrictors” for details how access restrictions can be applied). The constructor with a `BundleContext` as its argument has the advantage that it will use an OSGi `LogService` if available and adds various OSGi server detectors which adds server information like product name and version to

the `version` command. Refer to Section 6.2.6, “Getting the agent version (version)” for details about the server infos provided.

Please note that for this use case the bundle `org.jolokia.osgi` should not be *started* but left in the state *resolved*. Otherwise, as soon as an OSGi `HttpService` registers, this bundle will try to add yet another agent servlet to this service, which is probably not what you want. Alternatively, the bundle property `org.jolokia.listenForHttpService` can be set to `false` in which case there will be never an automatic servlet registration to an `HttpService`.

3.2.5. Restrictor service

As described in Section 4.2, “Jolokia Restrictors”, the Jolokia agent can use custom restrictors implementing the interface `org.jolokia.restrictor.Restrictor`. If the bundle property `org.jolokia.useRestrictorService` is set to `true` and no restrictor is configured by other means, the agent will use one or more OSGi service which register under the name `org.jolokia.restrictor.Restrictor`. If no such service is available, access to the agent is always denied. If one such restrictor service is available, the access decision is delegated to this service. When more than one restrictor service is available, access is only granted if all of them individually grant access. A sample restrictor service as a maven project can be found in the Jolokia source at `agent/osgi/restrictor-sample`.

3.3. Mule Agent

Jolokia's [Mule](#) agent uses Mule's own agent interface for plugging into the ESB running in standalone mode.

The agent needs to be included into the Mule configuration as shown in the following example, which is the way how to configure the agent for Mule 3:

```
<mule xmlns="http://www.mulesoft.org/schema/mule/core"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:management="http://www.mulesoft.org/schema/mule/management"
  xmlns:spring="http://www.springframework.org/schema/beans"
  xsi:schemaLocation="
    http://www.mulesoft.org/schema/mule/core
      http://www.mulesoft.org/schema/mule/core/3.1/mule.xsd
    http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
    http://www.mulesoft.org/schema/mule/management
      http://www.mulesoft.org/schema/mule/management/3.1/mule-management.xsd">

  <!-- .... -->
  <custom-agent name="jolokia-agent" class="org.jolokia.mule.JolokiaMuleAgent">
    <spring:property name="port" value="8899"/>
  </custom-agent>
  <management:jmx-server/>
</mule>
```

For Mule 2, the configuration is slightly different since the `<custom-agent>` is contained in the management namespace for Mule 2 (`<management:custom-agent>`)

This agent knows about the following configuration parameters

Table 3.4. Mule agent configuration options

Parameter	Description	Example
host	Hostaddress to which the HTTP server should bind to.	<code>InetAddress.getLocalHost()</code>
port	Port the HTTP server should listen to.	8888
user	Use to authenticate against. This switches on security and requires a client to provide a user and password.	
password	Password to check against when security is switched on.	
debug	Debugging state after startup. Can be changed via the Section 7.1, "Configuration MBean" during runtime.	<code>false</code>
historyMaxEntries	Entries to keep in the history. Can be changed at runtime via the Section 7.1, "Configuration MBean".	10
debugMaxEntries	Maximum number of entries to keep in the local debug history (if enabled). Can be changed via the Section 7.1, "Configuration MBean" at runtime.	100
maxDepth	Maximum depth when traversing bean properties. If set to 0, depth checking is disabled	5
maxCollectionSize	Maximum size of collections returned when serializing to JSON. When set to 0, collections are never truncated.	0
maxObjects	Maximum number of objects which are traversed when serializing a single response. Use this as an airbag to avoid boosting your memory and network traffic. Nevertheless, when set to 0 no limit is imposed.	10000

The context under which the agent is reachable is fixed to `/jolokia`. As an alternative to this Mule agent, the Section 3.4, “JVM Agent” can be used for Mule, too. This agent also knows about SSL encryption and authentication.

3.4. JVM Agent

The JVM agent is right agent when it comes to instrument an arbitrary Java application which is not covered by the other agents. This agent can be started by any Java program by providing certain startup options to the JVM. Or it can be dynamically attached (and detached) to an already running Java process. This universal agent uses the [JVM agent API](#) and is available for every Sun/Oracle JVM 1.6 and later.

3.4.1. Jolokia as JVM Agent

The JVM agent uses the [JVM Agent interface](#) for linking into any JVM. Under the hood it uses an HTTP-Server, which is available on every Oracle/Sun JVM from version 1.6 upwards.

The JDK embedded HTTP-Server is not the fastest one (it is used e.g. for the JAXWS reference implementation), but for our monitoring needs the performance is sufficient. It is sufficient for our needs. There are several configuration options for tuning the HTTP server's performance. See below for details.

3.4.1.1. Installation

This agent gets installed by providing a single startup option `-javaagent` when starting the Java process.

```
java -javaagent:agent.jar=port=7777,host=localhost
```

`agent.jar` is the filename of the Jolokia JVM agent. Options can be appended as a comma separated list. The available options are the same as described in Table 3.1, “Servlet init parameters” plus the one described in table Table 3.5, “JVM agent configuration options”. If an options contains a comma, an equal sign or a backslash, it must be escaped with a backslash.

Table 3.5. JVM agent configuration options

Parameter	Description	Example
<code>agentContext</code>	Context under which the agent is deployed. The full URL will be <code>protocol://host:port/agentContext</code>	<code>/jolokia</code>
<code>host</code>	Host address to which the HTTP server should bind to.	<code>InetAddress.getLocalHost()</code>
<code>port</code>	Port the HTTP server should listen to.	<code>8778</code>

Parameter	Description	Example
<code>protocol</code>	HTTP protocol to use. Should be either <code>http</code> or <code>https</code> . For the SSL stack there are various additional configuration options.	<code>http</code>
<code>backlog</code>	Size of request backlog before requests get discarded.	<code>10</code>
<code>executor</code>	Threading model of the HTTP server:	<code>single</code>
	<code>fixed</code> Thread pool with a fixed number of threads (see also <code>threadNr</code>)	
	<code>cached</code> Cached thread pool which creates threads on demand	
	<code>single</code> A single thread only	
<code>threadNr</code>	Number of threads to be used when the <code>fixed</code> execution model is chosen.	<code>5</code>
<code>keystore</code>	Path to the SSL keystore to use (https only)	
<code>keystorePassword</code>	Keystore password (https only)	
<code>useSslClientAuthentication</code>	Whether client certificates should be used for authentication (https only). (true OR false).	<code>false</code>
<code>config</code>	Path to a properties file from where the configuration options should be read. Such a property file can contain the configuration options as described here as key value pairs (except for the <code>config</code> property of course :)	

Upon successful startup the agent will print out a success message with the full URL which can be used by clients for contacting the agent.

3.4.2. Attaching a Jolokia agent on the fly

A Jolokia agent can be attached to any running Java process as long as the user has sufficient

access privileges for accessing the process. This agent uses the [Java attach API](#) for dynamically attaching and detaching to and from the process. It works similar to JConsole connecting to a local process. The Jolokia advantage is, that after the start of the agent, it can be reached over the network.

The JAR containing the JVM agent also contains a client application which can be reached via the `-jar` option. Call it with `--help` to get a short usage information:

```
$ java -jar jolokia-jvm-1.0.1-agent.jar --help

Jolokia Agent Launcher
=====

Usage: java -jar jolokia-jvm-1.0.1-SNAPSHOT-agent.jar [options] <command> <pid/regexp>

where <command> is one of
  start      -- Start a Jolokia agent for the process specified
  stop       -- Stop a Jolokia agent for the process specified
  status     -- Show status of an (potentially) attached agent
  toggle     -- Toggle between start/stop (default when no command is given)
  list       -- List all attachable Java processes (default when no argument is given at

[options] are used for providing runtime information for attaching the agent:

  --host <host>                Hostname or IP address to which to bind on
                                (default: InetAddress.getLocalHost())
  --port <port>                 Port to listen on (default: 8778)
  --agentContext <context>     HTTP Context under which the agent is reachable (default
  --user <user>                 User used for Basic-Authentication
  --password <password>        Password used for Basic-Authentication
  --quiet                       No output. "status" will exit with code 0 if the agent i
  --verbose                     Verbose output
  --executor <executor>         Executor policy for HTTP Threads to use (default: single
                                "fixed" -- Thread pool with a fixed number of threads (
                                "cached" -- Cached Thread Pool, creates threads on deman
                                "single" -- Single Thread
  --threadNr <nr threads>       Number of fixed threads if "fixed" is used as executor
  --backlog <backlog>           How many request to keep in the backlog (default: 10)
  --protocol <http|https>       Protocol which must be either "http" or "https" (default
  --keystore <keystore>         Path to keystore (https only)
  --keystorePassword <pwd>      Password to the keystore (https only)
  --useSslClientAuthentication Use client certificate authentication (https only)
  --debug                       Switch on agent debugging
  --debugMaxEntries <nr>        Number of debug entries to keep in memory which can be f
  --maxDepth <depth>            Maximum number of levels for serialization of beans (def
  --maxCollectionSize <size>     Maximum number of element in collections to keep when se
  --maxObjects <nr>             Maximum number of objects to consider for serialization
  --policyLocation <url>         Location of a Jolokia policy file
  --mbeanQualifier <qualifier>  Qualifier to use when registering Jolokia internal MBean
  --config <configfile>         Path to a property file from where to read the configura
  --help                       This help documentation

<pid/regexp> can be either a numeric process id or a regular expression. A regular express
against the processes' names (ignoring case) and must be specific enough to select exactly

If no <command> is given but only a <pid> the state of the Agent will be toggled
between "start" and "stop"

If neither <command> nor <pid> is given, a list of Java processes along with their IDs
is printed
```

For more documentation please visit www.jolokia.org

Every option described in Table 3.5, “JVM agent configuration options” is reflected by a command line option for the launcher. Additionally, the option `--quiet` can be used to keep the launcher silent and `--verbose` for adding some extra logging.

The launcher knows various operational modes, which needs to be provided as a non-option argument and possibly require an extra argument.

start

Use this to attach an agent to an already running, local Java process. The additional argument is either the *process id* of the Java process to attach to or a *regular expression* which is matched against the Java processes names. In the later case, exactly one process must match, otherwise an exception is raised. The command will return with an return code of 0 if an agent has been started. If the agent is already running, nothing happens and the launcher returns with 1. The URL of the Agent will be printed to standard out on an extra line except when the `--quiet` option is used.

stop

Command for stopping an running and dynamically attached agent. The required argument is the Java process id or an regular expression as described for the `start` command. If the agent could be stopped, the launcher exits with 0, it exits with 1 if there was no agent running.

toggle

Starts or stops an dynamically attached agent, dependending on its current state. The Java process ID is required as an additional argument. If an agent is running, `toggle` will stop it (and vice versa). The launcer returns with an exit code of 0 except when the operation fails. When the agent is started, the full agent's URL is printed to standard out. `toggle` is the default command when only a numeric process id is given as argument or a regular expression which *not* the same as a known command.

status

Command for showing the current agent status for a given process. The process id or a regular expresssion is required. The launcer will return with 0 when the agent is running, otherwise with 1.

list

List all local Java processes in a table with the process id and the description as columns. This is the default command if no non-option argument is given at all. `list` returns with 0 upon normal operation and with 1 otherwise.

The launcher is especially suitable for *one-shot*, *local* queries. For example, a simple shell script for printing out the memory usage of a local Java process, including (temporarily) attaching an Jolokia agent looks simply like in the following example. With a complete client library like [Jmx4Perl](#) even more one shot scripts are possible⁶.

```
#!/bin/sh
url=`java -jar agent.jar start $1 | tail -1`
```

⁶And in fact, some support for launching this dynamic agent is planned for a forthcoming release of `jmx4perl`.

```
memory_url="${url}read/java.lang:type=Memory/HeapMemoryUsage"
used=`wget -q -O - "${memory_url}/used" | sed 's/^.*"value":\([0-9]*\).*$/\1/'`
max=`wget -q -O - "${memory_url}/max" | sed 's/^.*"value":\([0-9]*\).*$/\1/'`
usage=$((($used*100/$max))
echo "Memory Usage: $usage %"

java -jar agent.jar --quiet stop $1
```

Chapter 4. Security

Security in JSR-160 remoting is an all-or-nothing option. Either all or none of your MBeans are accessible (except when your application server uses a `SecurityManager`, but that is not often the case). Jolokia, on the other hand, allows for fine grained security defined in an XML security policy file. It allows for access restrictions on MBean names (or patterns), attributes, operations, source IP address (or a subnet) and type of Jolokia operation.

4.1. Policy based security

Access to MBean and to the Jolokia agents in general can be restricted with an XML policy file. This policy can be configured for various parameters and is divided into several sections.

4.1.1. IP based restrictions

Overall access can be granted based on the IP address of an HTTP client. These restrictions are specified within a `<remote>` section, which contains one or more `<host>` elements. The source can be given either as an IP address, a host name, or a netmask given in [CIDR format](#) (e.g. "10.0.0.0/16" for all clients coming from the 10.0 network). The following allows access from localhost and all clients whose IP addresses start with "10.0". For all other IP addresses access is denied.

```
<remote>
  <host>localhost</host>
  <host>10.0.0.0/16</host>
</remote>
```

4.1.2. Commands

This section specifies the Jolokia commands for which access is generally granted. For each command in the list, access can be further restricted within the `<deny>` part and each command missing in the list, which is forbidden globally, can be selectively enabled for certain MBeans in the `<allow>` section. If the `<commands>` section is missing completely, access to all commands is allowed.

All Jolokia commands described in Chapter 6, *Jolokia Protocol* can be used in this section:

read

Reading of MBean attributes

write

Setting of MBean attributes

exec

Execution of JMX operations

list

List the available MBeans along with their supported attributes and operations.

search

Searching for MBeans

version

Getting version and server information

In the following example, access is granted to the `read`, `list`, `search` and `version` command, but not to `write` and `exec` operations.

```
<commands>
  <command>read</command>
  <command>list</command>
  <command>version</command>
  <command>search</command>
</commands>
```

4.1.3. Allow and deny access to certain MBeans

Within an `<allow>` section, access to MBeans can be granted regardless of the operations specified in the `<commands>` section. The reverse is true for the `<deny>` section: It rejects access to the MBeans specified here. Both sections contain one or more `<mbean>` elements which have a format like:

```
<mbean>
  <name>java.lang:type=Memory</name>
  <attribute>*Memory*</attribute>
  <attribute mode="read">Verbose</attribute>
  <operation>gc</operation>
</mbean>
```

Within the `<name>` section the name of the MBean is specified. This can be either a complete `ObjectName` or a MBean pattern containing wildcards. The value given here must conform to the JMX specification for a valid `ObjectName`. On this MBean (or *MBeans* if `name` is a pattern), attributes are specified within one or more `<attribute>` elements and operations within one or more `<operation>` elements. The content can also be a pattern, which uses a wildcard `*`. e.g. `<attribute>*</attribute>` specifies all attributes on the given MBean. If for an `<attribute>` element the XML attribute `mode="read"` is given, then this attribute can be accessed only read-only.

4.1.4. HTTP method restrictions

Finally, access can be restricted based on the HTTP method with which an Jolokia request was received with the `<http>` element. Method allowed (`post` or `get`) are specified with an `<method>` inner element. The following example restricts the access to POST requests only:

```
<http>
  <method>post</method>
</http>
```

If the `<http>` section is missing completely, any HTTP method can be used.

4.1.5. Cross-Origin Resource Sharing (CORS) restrictions

Jolokia (since version 1.0.3) supports the W3C specification for [Cross-Origin Resource Sharing](#) (also known as "CORS") which allows browser to access resources which are located on a different server than the calling script is loaded from. This specification provides a controlled way to come around the *same origin policy*. Most [contemporary browsers](#) support CORS.

By default Jolokia allows cross origin access from any host. This can be limited to certain hosts by using `<allow-origin>` sections within a `<cors>` sections. This tags can contain the origin URL provided by browsers with the `Origin:` header literally or a wildcard specification with `*`.

```
<cors>
  <!-- Allow cross origin access from www.jolokia.org ... -->
  <allow-origin>http://www.jolokia.org</allow-origin>

  <!-- ... and all servers from jmx4perl.org with any protocol ->
  <allow-origin>*://*.jmx4perl.org</allow-origin>
</cors>
```

4.1.6. Example for a security policy

The following complete example applies various access restrictions:

- Access is only allowed for clients coming from localhost
- Only HTTP Post requests are allowed
- By default, only `read` and `list` requests are allowed.
- A single `exec` request is allowed for triggering garbage collection.
- Read access to the C3P0 connection pool is restricted to forbid fetching the pool's properties, which in fact contains the DB password as clear text.

```
<?xml version="1.0" encoding="utf-8"?>
<restrict>
  <remote>
    <host>127.0.0.1</host>
  </remote>

  <http>
    <method>post</method>
  </http>

  <commands>
    <command>read</command>
    <command>list</command>
  </commands>

  <allow>
    <mbean>
      <name>java.lang:type=Memory</name>
```

```

    <operation>gc</operation>
  </mbean>
</allow>

<deny>
  <mbean>
    <name>com.mchange.v2.c3p0:type=PooledDataSource,*</name>
    <attribute>properties</attribute>
  </mbean>
</deny>

</restrict>

```

4.1.7. Policy Location

A great tool which helps in repackaging an agent for inclusion of a `jolokia-access.xml` policy file is the command line tool [jolokia](#), which comes with the [jmx4perl](#) distribution. See Chapter 9, *Tools* for an introduction.

But how do the agents lookup the policy file ? By default, the agents will lookup for a policy file top-level in the classpath under the name `jolokia-access.html`. Hence for the war agent, the policy file must be packaged within the war at `WEB-INF/classes/jolokia-access.xml`, for all other agents at `/jolokia-access.xml`. The location can be overwritten with the configuration parameter `policyLocation`, which has to be set differently depending on the agent type. Please refer to Chapter 3, *Agents* for more details. The value of this init parameter can be any URL which can be loaded by the JVM. A special case is an URL with the scheme `classpath:` which results in a lookup of the policy file within the classpath. As stated above, the default value of this parameter is `classpath:/jolokia-access.xml`. If a non-classpath URL is provided with this parameter, and the target policy file could not be found then access is completely denied. If a classpath lookup fails then access is globally granted and a warning is given on standard output.

4.2. Jolokia Restrictors

In order to provide fine grained security, Jolokia uses the abstract concept of an *Restrictor*. It is represented by the Java interface `org.jolokia.restrictor.Restrictor` and comes with several implementations. The most prominent one is the `PolicyRestrictor` which is described in Section 4.1, “Policy based security”. This is also the restrictor which is active by default. For special needs, it is possible to provide a custom implementation of this interface for the WAR and OSGi agents. It is recommended to subclass either `org.jolokia.restrictor.AllowAllRestrictor` or `org.jolokia.restrictor.DenyAllRestrictor`.

For the WAR agent (Section 3.1, “JEE Agent (WAR)”), a subclass of `org.jolokia.http.AgentServlet` should be created which overrides the `createRestrictor()`

```

public class RestrictedAgentServlet extends AgentServlet {

    @Override
    protected Restrictor createRestrictor(String policyLocation) {
        return new MyOwnRestrictor();
    }
}

```

```
}  
}
```

`policyLocation` is a URL pointing to the policy file, which is either the default value `classpath:/jolokia-access.xml` or the value specified with the init parameter `policyLocation`. This servlet can then be easily configured in a custom `web.xml` the same way as the Jolokia agent.

For programmatic usage there is an even simpler way: `AgentServlet` provides an constructor which takes an restrictor as argument, so no subclassing is required in this case.

For an OSGi agent (Section 3.2, “OSGi Agents”), `org.jolokia.osgi.servlet.JolokiaServlet` is the proper extension point. It can be subclassed the same way as shown above and allows a restrictor implementation as constructor parameter, too. In contrast to `AgentServlet` this class is also OSGi exported and can be referenced from other bundles. Additionally, the OSGi agent can also pick up a restrictor as an OSGi service. See Section 3.2, “OSGi Agents” for details.

Chapter 5. Proxy Mode

Using Jolokia in proxy mode enables for agentless operation on the target server. A dedicated agent deployment proxies by accepting Jolokia requests as input, translating them to JSR-160 requests for the target. This setup is described in Chapter 2, *Architecture*. As noted there, the real target is given within the original request, which must be sent as a POST request.

Agents of all types support the proxy mode. However, since one has usually the free choice of platform for a dedicated Jolokia proxy, an environment optimized for HTTP communication should be used. These are either servlet container or JEE server hosting the WAR agent or an OSGi runtime with an OSGi `HttpService` (which in turn is typically based on an embedded servlet container like Tomcat or Jetty). The two other agents, the Mule and JVM agents are not that well suited for this job.

All client libraries (jmx4perl, Java and Javascript) support the usage of proxy mode in its API.

5.1. Limitations of proxy mode

The proxy mode has some limitations compared to the direct agent mode, so it is recommended to use a direct agent deployment if possible. The limitations are:

- There is no automatic merging of JMX MBeanServers as in the case of the direct mode. Most application servers use their own MBeanServer in addition to the `PlatformMBeanServer` (which is always present). Each MBean is registered only in one MBeanServer. The choice of which `MBeanServer` to use has to be given up front, usually as a part of the JMX Service URL. But even then (as it is the case for JBoss 5.1) you might run into problems when selecting the proper MBeanServer.
- Proxying adds an additional remote layer which causes additional problems. I.e. the complex operations like `list` might fail in the proxy mode because of serialization issues. E.g. for JBoss it happens that certain MBeanInfo objects requested for the list operation are not serializable. This is a bug of JBoss, but I expect similar limitations for other application servers as well.
- Certain workarounds (like the JBoss "*can not find MXBeans before MBeanInfo has been fetched*" bug) works only in agent mode.
- It is astonishingly hard to set up an application server for JSR-160 export. And there are even cases (combinations of JDK and AppServer Version) which don't work at all properly (e.g. JDK 1.5 and JBoss 5). For certain application servers, detailed setup instructions are published at our [blog](#).

Chapter 6. Jolokia Protocol

Jolokia uses a JSON-over-HTTP protocol which is described in this chapter. The communication is based on a request-response paradigm, where each request results in a single response.

GET URLs are chatty

Keep in mind that many web servers log the requested path of every request, including parameters passed to a GET request, so sending messages over GET often bloats server logs.

Jolokia requests can be sent in two ways: Either as a HTTP GET request, in which case the request parameters are encoded completely in the URL. Or as a POST request where the request is put into a JSON payload in the HTTP request's body. GET based requests are mostly suitable for simple use cases and for testing the agent via a browser. The focus here is on simplicity. POST based requests uses a JSON representation of the request within the HTTP body. They are more appropriate for complex requests and provide some additional features (e.g. bulk requests are only possible with POST).

The response returned by the agent uses always JSON for its data representation. It has the same format regardless whether GET or POST requests are used.

The rest of this chapter is divided into two parts: First, the general structure of requests and responses are explained after which the representation of Jolokia supported operations defined.

Note

Unfortunately the term *operation* is used in different contexts which should be distinguished from one another. *Jolokia operations* denote the various kind of Jolokia requests, whereas *JMX operations* are methods which can be invoked on an JMX MBean. Wherever the context requires it, this documents uses *Jolokia* or *JMX* as prefix.

6.1. Requests and Responses

Jolokia knows about two different styles of handling requests, which are distinguished by the HTTP method used: GET or POST. Regardless of what method is used, the agent doesn't keep any state on the server side (except of course that MBeans are mostly obviously stateful). So in this aspect, the communication can be considered [REST](#) like¹.

6.1.1. GET requests

The simplest way to access the Jolokia is by sending HTTP GET requests. These requests encode all its parameters within the access URL. Typically, Jolokia uses the path-info part of an URL to extract the parameters. Within the path-info, each part is separated by a slash (/). In general, the request URL looks like

```
<base-url>/<type>/<arg1>/<arg2>/..../
```

¹This document will avoid the term REST as much as possible in order to avoid provoking any dogmatic resentments.

The `<base-url>` specifies the URL under which the agent is accessible. It typically looks like `http://localhost:8080/jolokia`, but depends obviously on your deployment setup. Normally, the last part of this URL is the name of the deployed agent, which by default is based on the agent's filename (e.g. `jolokia.war`). `<type>` specifies one of the supported Jolokia operations (described in the next section), followed by one or more operation-specific parameters separated by slashes.

For example, the following URL executes a `read` Jolokia operation on the MBean `java.lang:type=Memory` for reading the attribute `HeapMemoryUsage` (see Section 6.2.1, “Reading attributes (read)”). It is assumed, that the agent is reachable under the base URL `http://localhost:8080/jolokia`:

```
http://localhost:8080/jolokia/read/java.lang:type=Memory/HeapMemoryUsage
```

Why escaping ?

You might wonder why simple URI encoding isn't enough for escaping slashes. The reason is that JBoss/Tomcat has a strange behaviour when returning an HTTP response `HTTP/1.x 400 Invalid URI: noSlash` for any URL which contains an escaped slash in the path info (i.e. `%2F`). The reason behind this behaviour is security related, slashes get decoded on the agent side before the agent-servlet gets the request. Other appservers might exhibit a similar behaviour, so Jolokia uses an own escaping mechanism.

If one of the request parts contain a slash (`/`) (e.g. as part of you bean's name) it needs to be escaped. An exclamation mark (`!`) is used as escape character². A exclamation mark needs to be doubled for escaping. Any other characted preceded by an exclamation mark is taken literally. Table Table 6.1, “Escaping rules” illustrates the escape rules as used in GET requests.

Table 6.1. Escaping rules

Escaped	Unescaped
<code>! /</code>	<code>/</code>
<code>!!</code>	<code>!</code>
<code>!(anything else)</code>	<code>(anything else)</code>

For example, to read the attribute `State` on the MBean named `jboss.jmx:alias=jmx/rmi/RMIAdaptor`, an access URL like this has to be constructed:

```
.../read/jboss.jmx:alias=jmx!/rmi!/RMIAdaptor/State
```

Client libraries like [JMX::Jmx4Perl](#) do this sort of escaping transparently.

Escaping can be avoided altogether if a slightly different variant for a request is used (which doesn't

² A backslash (`\`) can not be used, since most servlet container translate a backslash into a forward slash on the fly when given in an URL.

look that REST-stylish, though). Instead of providing the information as path-info, a query parameter `p` can be used instead. This should be URL encoded, though. For the example above, the alternative is

```
http://localhost:8080/jolokia?p=/read/test:strangeName=%2Fbla%2F%2F/attribute
```

6.1.2. POST requests

POST requests are the most powerful way to communicate with the Jolokia agent. There are fewer escaping issues and it allows for features which are not available with GET requests. POST requests uses a fixed URL and put their payload within the HTTP request's body. This payload is represented in [JSON](#), a data serialization format originating from JavaScript world.

The JSON format for a single request is a JSON object, which is essentially a map with keys (or *attributes*) and values. All requests have a common mandatory attribute, `type`, which specifies the kind of JMX operation to perform. The other attributes are either operation specific as described in Section 6.2, "Jolokia operations" or are *processing parameters* which influence the overall behaviour and can be mixed in to any request. See Section 6.3, "Processing parameters" for details.

A sample read request in JSON format looks like the following example. It has a `type` "read" (case doesn't matter) and the three attributes `mbean`, `attribute` and `path` which are specific to a read request.

Example 6.1. JSON Request

```
{
  "type" : "read",
  "mbean" : "java.lang:type=Memory",
  "attribute" : "HeapMemoryUsage",
  "path" : "used",
}
```

Each request JSON object results in a single JSON response object contained in the HTTP answer's body. A *bulk request* contains multiple Jolokia requests within a single HTTP request. This is done by putting individual Jolokia requests into a JSON array:

```
[
  {
    "type" : "read",
    "attribute" : "HeapMemoryUsage",
    "mbean" : "java.lang:type=Memory",
    "path" : "used",
  },
  {
    "type" : "search"
    "mbean" : "*:type=Memory,*",
  }
]
```

This request will result in a JSON array containing multiple JSON responses within the HTTP response. They are returned in same order as the requests in the initial bulk request.

6.1.3. Responses

Responses are always encoded in UTF-8 JSON, regardless whether the request was a GET or POST request. In general, two kinds of responses can be classified: In the normal case, a HTTP Response with response code 200 is returned, containing the result of the operation as a JSON payload. In case of an error, a 4xx or 5xx code will be returned and the JSON payload contains details about the error occurred (e.g. 404 means "not found"). (See [this page](#) for more information about HTTP error codes..)

In the non-error case a JSON response looks mostly the same for each request type except for the `value` attribute which is request type specific.

The format of a single Jolokia response is

Example 6.2. JSON Response

```
{
  "value": .... ,
  "status" : 200,
  "timestamp" : 1244839118,
  "request": {
    "type": ... ,
    ....
  },
  "history": [
    { "value": ... ,
      "timestamp" : 1244839045
    }, ....
  ]
}
```

For successful requests, the `status` is always 200 (the HTTP success code). The `timestamp` contains the epoch time³ when the request has been handled. The request leading to this response can be found under the attribute `request`. Finally and optionally, if history tracking is switched on (see Section 6.5, "Tracking historical values"), an entry with key `history` contains a list of historical values along with their timestamps. History tracking is only available for certain type of requests (`read`, `write` and `exec`). The `value` is specific for the type of request, it can be a single scalar value or a monster JSON structure.

If an error occurs, the `status` will be a number different from 200. An error response looks like

```
{
  "status": 400,
  "error_type": "java.lang.IllegalArgumentException",
  "error": "java.lang.IllegalArgumentException: Invalid request type 'java.lang:type=Memo",
  "stacktrace": "java.lang.IllegalArgumentException: Invalid request type 'java.lang:type="
```

³ Seconds since 1.1.1970

```

\tat org.cpan.jmx4perl.JmxRequest.extractType(Unknown Source)\n
\tat org.cpan.jmx4perl.JmxRequest.<init>(Unknown Source) ...."
}

```

For status codes it is important to distinguish status codes as they appear in Jolokia JSON response objects and the HTTP status code of the (outer) HTTP response. There can be many Jolokia status codes, one for each Jolokia request contained in the single HTTP request. The HTTP status code merely reflect the status of agent itself (i.e. whether it could perform the operation at all), whereas the Jolokia response status reflects the result of the operation (e.g. whether the performed operation throws an exception). So it is not uncommon to have an HTTP status code of 200, but the contained JSON response(s) indicate some errors.

I.e. the `status` has a code in the range 400 .. 499 or 500 .. 599 [as it is specified for HTTP return codes](#). The `error` member contains an error description. This is typically the message of an exception occurred on the agent side⁴. Finally, `error_type` contains the Java class name of the exception occurred. The `stacktrace` contains a Java stacktrace occurred on the server side (if any stacktrace is available).

For each type of operation, the format of the `value` entry is explained in Section 6.2, “Jolokia operations”

6.1.4. Pathes

Before we come to the Jolokia operation, the general concept of *inner pathes* requires some explanation.

An *inner path* points to a certain substructure (plain value, array, hash) within a a complex JSON value. Think of it as something like "XPath lite". This is best explained by an example:

The attribute `HeapMemoryUsage` of the MBean `java.lang:type=Memory` can be requested with the URL `http://localhost:8080/j4p/read/java.lang:type=Memory/HeapMemoryUsage` which returns a complex JSON structure like

```

{
  "committed" : 18292736,
  "used" : 15348352,
  "max" : 532742144,
  "init" : 0
}

```

In order to get to the value for used heap memory you should specify an inner path `used`, so that the request `http://localhost:8080/j4p/read/java.lang:type=Memory/HeapMemoryUsage/used` results in a response of `15348352`. If the attribute contains arrays at some level, use a numeric index (0 based) as part of the inner path if you want to traverse into this array.

For both, GET and POST requests, pathes must be escaped as described in Table 6.1, “Escaping rules” when they contain slashes (/) or exclamation marks (!).

⁴ If the server exception is a subtype of `MBeanException`, the wrapped exception's message is used.

6.2. Jolokia operations

6.2.1. Reading attributes (read)

Reading MBean attributes is probably the most used JMX method, especially when it comes to monitoring. Concerning Jolokia, it is also the most powerful one with the richest semantics. Obviously the value of a single attribute can be fetched, but Jolokia supports also fetching of a list of given attributes on a single MBean or even on multiple MBeans matching a certain pattern.

Reading attributes are supported by both kinds of requests, `GET` and `POST`.

Note

Don't confuse fetching multiple attributes on possibly multiple MBeans with bulk requests. A single read request will always result in a single read response, even when multiple attribute values are fetched. Only the single response's structure of the `value` will differ depending on what kind of read request was performed.

A read request for multiple attributes on the same MBean is initiated by giving a list of attributes to the request. For a `POST` request this is an JSON array, for a `GET` request it is a comma separated list of attribute names (where slashes and exclamation marks must be escaped as described in Table 6.1, "Escaping rules"). If no attribute is provided, then all attributes are fetched. The MBean name can be given as a pattern in which case the attributes are read on all matching MBeans. If a MBean pattern and multiple attributes are requested, then only the value of attributes which matches both are returned, the others are ignored. Pathes cannot be used with multi value reads, though.

6.2.1.1. GET read request

The `GET` URL for a read request has the following format:

```
<base-url>/read/<mbean name>/<attribute name>/<inner path>
```

Table 6.2. GET Read Request

Part	Description	Example
<code><mbean name></code>	The ObjectName of the MBean for which the attribute should be fetched. It contains two parts: A domain part and a list of properties which are separated by <code>:</code> . Properties themselves are combined in a comma separated list of key-value pairs. This name can be a pattern in which case multiple MBeans are queried for the attribute value.	<code>java.lang:type=Memory</code>
<code><attribute name></code>	Name of attribute to read. This	<code>HeapMemoryUsage</code>

Part	Description	Example
	can be a list of Attribute names separated by comma. Slashes and exclamations marks need to be escaped as described in Table 6.1, "Escaping rules". If no attribute is given, all attributes are read.	
<inner path>	This optional part describes an <i>inner path</i> as described in Section 6.1.4, "Pathes"	used

With this URL the used heap memory can be obtained:

```
http://localhost:8080/jolokia/java.lang:type=Memory/HeapMemoryUsage/used
```

6.2.1.2. POST read request

A the keys available for read POST requests are shown in the following table.

Table 6.3. POST Read Request

Key	Description	Example
type	read	
mbean	MBean's ObjectName which can be a pattern	java.lang:type=Memory
attribute	Attribute name to read or a JSON array containing a list attributes to read. No attribute is given, then all attributes are read.	HeapMemoryUsage, ["HeapMemoryUsage", "NonHeapMemoryUsage"]
path	Inner path for accessing the value of a complex value (Section 6.1.4, "Pathes")	used

The following request fetches the number of active threads:

```
{
  "type": "read",
  "mbean": "java.lang:type=Threading",
  "attribute": "ThreadCount"
}
```

6.2.1.3. Read response

The general format of the JSON response is described in Section 6.1.3, “Responses” in detail. A typical response for an attribute read operation for an URL like

```
http://localhost:8080/jolokia/read/java.lang:type=Memory/HeapMemoryUsage/
```

looks like

```
{
  "value":{
    "init":134217728,
    "max":532742144,
    "committed":133365760,
    "used":19046472
  },
  "status":200,
  "timestamp":1244839118,
  "request":{
    "mbean":"java.lang:type=Memory",
    "type":"read",
    "attribute":"HeapMemoryUsage"
  },
  "history":[{"value":{
    "init":134217728,
    "max":532742144,
    "committed":133365760,
    "used":18958208
  },
    "timestamp":1244839045
  }, ....
]
```

The `value` contains the response's value. For simple data types it is a scalar value, more complex types are serialized into a JSON object. See Section 6.4, “Object serialization” for detail on object serialization.

For read request to a single MBean with multiple attributes, the returned value is a JSON object with the attribute names as keys and their values as values. For example a request to `http://localhost:8080/jolokia/read/java.lang:type=Memory` leads to

```
{
  "timestamp": 1317151518,
  "status": 200,
  "request": {"mbean":"java.lang:type=Memory", "type":"read"},
  "value":{
    "Verbose": false,
    "ObjectPendingFinalizationCount": 0,
    "NonHeapMemoryUsage": {"max":136314880,"committed":26771456,"init":24317952,"used":1521...},
    "HeapMemoryUsage": {"max":129957888,"committed":129957888,"init":0,"used":2880008}
  }
}
```

A request to a MBean pattern returns as value a JSON object, with the MBean names as keys and as value another JSON object with the attribute name as keys and the attribute values as values. For example a request `http://localhost:8080/jolokia/read/java.lang:type=*/HeapMemoryUsage` returns something like

```
{
  "timestamp": 1317151980,
  "status": 200,
  "request": {"mbean": "java.lang:type=*", "attribute": "HeapMemoryUsage", "type": "read"},
  "value": {
    "java.lang:type=Memory": {
      "HeapMemoryUsage": {"max": 129957888, "committed": 129957888, "init": 0, "used": 3080912}
    }
  }
}
```

6.2.2. Writing attributes (write)

Writing an attribute is quite similar to reading one, except that the request takes an additional `value` element.

6.2.2.1. GET write request

Writing an attribute with an GET request, an URL with the following format has to be used:

```
<base url>/write/<mbean name>/<attribute name>/<value>/<inner path>
```

Table 6.4. GET Write Request

Part	Description	Example
<mbean name>	MBean's ObjectName	<code>java.lang:type=ClassLoading</code>
<attribute name>	Name of attribute to set	<code>Verbose</code>
<value>	The attribute name to value. The value must be serializable as described in Section 6.4.2, "Request parameter serialization".	<code>true</code>
<path>	Inner path for accessing the parent object on which to set the value. (See also Section 6.1.4, "Pathes"). Note, that this is <i>not</i> the path to the attribute itself, but to the object carrying this attribute. With a given path it is possible to deeply set an value on a	

Part	Description	Example
	complex object.	

For example, you can set the garbage collector to verbose mode by using something like

```
http://localhost:8080/j4p/write/java.lang:type=Memory/Verbose/true
```

6.2.2.2. POST write request

The keys which are evaluated for a POST write request are:

Table 6.5. POST Write Request

Key	Description	Example
type	write	
mbean	MBean's ObjectName	java.lang:type=ClassLoading
attribute	Name of attribute to set	Verbose
value	The attribute name to value. The value must be serializable as described in Section 6.4.2, "Request parameter serialization".	true
path	An optional inner path for specifying an inner object on which to set the value. See Section 6.1.4, "Pathes" for more on inner pathes.	

6.2.2.3. Write response

As response for a write operation the old attribute's value is returned. For a request

```
http://localhost:8080/j4p/write/java.lang:type=ClassLoading/Verbose/true
```

you get the answer (supposed that verbose mode was switched off for class loading at the time this request was sent)

```
{
  "value": "false",
  "status": 200,
  "request": {
    "mbean": "java.lang:type=ClassLoading",
    "type": "write",
    "attribute": "Verbose",
    "value": true
  }
}
```

```
}
}
```

The response is quite similar to the read operation except for the additional `value` element in the request (and of course, the different `type`).

6.2.3. Executing JMX operations (exec)

Beside attribute provides a way for the execution of exposed JMX operations with optional arguments. The same as for writing attributes, Jolokia must be able to serialize the arguments. See Section 6.4, “Object serialization” for details. Execution of overloaded methods is supported. The JMX specifications recommends to avoid overloaded methods when exposing them via JMX, though.

6.2.3.1. GET exec request

The format of an GET exec request is

```
<base url>/exec/<mbean name>/<operation name>/<arg1>/<arg2>/....
```

Table 6.6. GET Exec Request

Part	Description	Example
<code><mbean name></code>	MBean's ObjectName	<code>java.lang:type=Threading</code>
<code><operation name></code>	Name of the operation to execute. If an overloaded method, it is mandatory to provide a method signature as well. A signature consist the fully qualified argument class names or native types, separated by columns and enclosed with parentheses.	<code>loadUsers(java.lang.String,int)</code>
<code><arg1>, <arg2>, ...</code>	String representation for the arguments required to execute this operation. Only certain data types can be used here as desribed in Section 6.4.2, “Request parameter serialization”.	<code>"true", "true"</code>

The following request will trigger a garbage collection:

```
http://localhost:8080/j4p/exec/java.lang:type=Memory/gc
```

6.2.3.2. POST exec request

Table 6.7. POST Exec Request

Key	Description	Example
type	exec	
mbean	MBean's ObjectName	java.lang:type=Threading
operation	The operation to execute, optionally with a signature as described above.	dumpAllThreads
arguments	An array of arguments for invoking this operation. The value must be serializable as described in Section 6.4.2, "Request parameter serialization".	[true,true]

The following request dumps all threads (along with locked monitors and locked synchronizers, that's what the boolean arguments are for):

```
{
  "type": "EXEC",
  "mbean": "java.lang:type=Threading",
  "operation": "dumpAllThreads",
  "arguments": [true,true]
}
```

6.2.3.3. Exec response

For an `exec` operation, the response contains the return value of the operation. `null` is returned if either the operation returns a null value or the operation is declared as void. A typical response for an URL like

```
http://localhost:8080/j4p/exec/java.util.logging:type=Logging/setLoggerLevel/global/INFO
```

looks like

```
{
  "value": null,
  "status": 200,
  "request": {
    "type": "exec",
    "mbean": "java.util.logging:type=Logging",
    "operation": "setLoggerLevel",
    "arguments": ["global", "INFO"]
  }
}
```

The return value get serialized as described in Section 6.4, “Object serialization”.

6.2.4. Searching MBeans (search)

With the Jolokia search operation the agent can be queried for MBeans with a given pattern. Searching will be performed on every `MBeanServer` found by the agent.

6.2.4.1. GET search request

The format of the search GET URL is:

```
<base-url>/search/<pattern>
```

This mode is used to query for certain MBean. It takes a single argument `pattern` for specifying the search parameter like in

```
http://localhost:8080/jolokia/search/*:j2eeType=J2EEServer,*
```

You can use patterns as described [here](#), i.e. it may contain wildcards like `*` and `?`. The Mbean names matching the query are returned as a list within the response.

6.2.4.2. POST search request

A search POST request knows the following keys:

Table 6.8. POST Search Request

Key	Description	Example
<code>type</code>	search	
<code>mbean</code>	The MBean pattern to search for	<code>java.lang:*</code>

The following request searches for all MBeans registered in the domain `java.lang`

```
{
  "type": "SEARCH",
  "mbean": "java.lang:*" ,
}
```

6.2.4.3. Search response

The answer is a list of MBean names which matches the pattern or an empty list if there was no match.

For example, the request

```
http://localhost:8888/j4p/search/*:j2eeType=J2EEServer,*
```

results in

```
{
  "value": [
    "jboss.management.local:j2eeType=J2EEServer,name=Local"
  ],
  "status":200,
  "timestamp":1245305648,
  "request": {
    "mbean": "*:j2eeType=J2EEServer,*", "type": "search"
  }
}
```

The returned MBean names are properly [quoted](#) so that they can be directly used as input for other requests.

6.2.5. Listing MBeans (list)

The list operation collects information about accessible MBeans. This information includes the MBean names, their attributes, operations and notifications along with type information and description (as far as they are provided by the MBean author which doesn't seem to be often the case).

6.2.5.1. GET list request

The GET request format for a Jolokia list request is

```
<base-url>/list/<inner path>
```

The `<inner path>`, as described in Section 6.1.4, “Paths” specifies a subset of the complete response. You can use this to select a specific domain, MBean or attribute/operation. See the next section for the format of the complete response.

6.2.5.2. POST list request

A list POST request has the following keys:

Table 6.9. POST list Request

Key	Description	Example
type	list	
path	Inner path for accessing the value of a subset of the complete list (Section 6.1.4,	<code>java.lang/type=Memory/attr</code>

Key	Description	Example
	"Pathes").	

The following request fetches the information about the MBean `java.lang:type=Memory`

```
{
  "type": "LIST",
  "path": "java.lang/type=Memory"
}
```

6.2.5.3. List response

The `value` has the following format:

```
{
  <domain> :
  {
    <prop list> :
    {
      "attr" :
      {
        <attr name> :
        {
          "type" : <attribute type>,
          "desc" : <textual description of attribute>,
          "rw" : true/false
        },
        ....
      },
      "op" :
      {
        <operation name> :
        {
          "args" : [
            {
              "type" : <argument type>
              "name" : <argument name>
              "desc" : <textual description of argument>
            },
            .....
          ],
          "ret" : <return type>,
          "desc" : <textual description of operation>
        },
        .....
      },
      "not" :
      {
        "name" : <name>,
        "desc" : <desc>,
        "types" : [ <type1>, <type2> ]
      }
    },
    ....
  },
  ....
}
```

The domain name and the property list together uniquely identify a single MBean. The property list is in the so called *canonical order*, i.e. in the form "`<key1>=<val1>,<key2>=<val2>,...`" where the keys are ordered alphabetically. Each MBean has zero or more attributes and operations which can be reached in an MBeans JSON object with the keys `attr` and `op` respectively. Within these groups the contained information is explained above in the schema and consist of Java types for attributes, arguments and return values, descriptive information and whether an attribute is writable (`rw == true`) or read-only.

As for reading attributes you can fetch a subset of this information using an path. E.g a path of `domain/prop-list` would return the value for a single bean only. For example, a request

```
http://localhost:8080/jolokia/list/java.lang/type=Memory
```

results in an answer

```
{
  "value":
  {
    "op":
    {
      "gc":
      {
        "args": [],
        "ret": "void",
        "desc": "gc"
      }
    },
    "attr":
    {
      "NonHeapMemoryUsage":
      {
        "type": "javax.management.openmbean.CompositeData",
        "rw": false,
        "desc": "NonHeapMemoryUsage"
      },
      "Verbose":
      {
        "type": "boolean",
        "rw": true,
        "desc": "Verbose"
      },
      "HeapMemoryUsage":
      {
        "type": "javax.management.openmbean.CompositeData",
        "rw": false,
        "desc": "HeapMemoryUsage"
      },
      "ObjectPendingFinalizationCount":
      {
        "type": "int",
        "rw": false,
        "desc": "ObjectPendingFinalizationCount"
      }
    }
  },
  "status": 200,
  "request":
```

```
{
  "type": "list",
  "path": "java.lang/type=Memory"
}
```

6.2.5.4. Restrict depth of depth

The optional parameter `maxDepth` can be used to restrict the depth of the return tree. Two values are possible: A `maxDepth` of 1 restricts the return value to a map with the JMX domains as keys, a `maxDepth` of 2 truncates the map returned to the domain names (first level) and the MBean's properties (second level). The final values of the maps don't have any meaning and are dummy values.

6.2.6. Getting the agent version (version)

The Jolokia command `version` returns the version of the Jolokia agent along with the protocol version.

6.2.6.1. GET version request

The GET URL for a version request has the following format:

```
<base-url>/version
```

For GET request the `version` part can be omitted since this is the default command if no command is provided as path info.

6.2.6.2. POST version request

A version POST request has only a single key `type` which has to be set to **version**.

6.2.6.3. Version response

The response value for a version request looks like:

```
{
  "timestamp": 1287143106,
  "status": 200,
  "request": { "type": "version" },
  "value": {
    "protocol": "4.0",
    "agent": "0.80",
    "info": {
      product: "glassfish",
      vendor: "Sun",
      extraInfo: {
        amxBooted: false
      }
    }
  }
}
```

`protocol` in the response value contains the protocol version used, `agent` is the version of the Jolokia agent. See Section 6.7, "Jolokia protocol versions" for the various protocol versions and the interoperability. If the agent is able to detect the server, additional meta information about this server is returned (i.e. the product name, the vendor and optionally some extra information added by the server detector).

6.3. Processing parameters

Jolokia operations can be influenced by so-called *processing parameters*. These parameters are provided differently for POST and GET requests.

For GET request, the processing parameters are given as normal query parameters:

```
<GET request URL>?param1=value1&param2=value2&...
```

For example the request

```
http://localhost:8080/jolokia/list?maxObjects=100
```

will limit the response to at max 100 values.

POST requests take the processing instructions within the JSON request below the key `config`:

```
{
  "type" : "list"
  "config" : {
    "maxObjects" : 100
  }
}
```

The list of known processing parameters is:

`maxDepth`

Maximum depth of the tree traversal into a bean's properties. The maximum value as configured in the agent's configuration is a hard limit and cannot be exceeded by a query parameter.

`maxCollectionSize`

For collections (lists, maps) this is the maximum size.

`maxObjects`

Number of objects to visit in total. A hard limit can be configured in the agent's configuration.

`ignoreErrors`

If set to "true", a Jolokia operation will not return an error if an JMX operation fails, but includes the exception message as value. This is useful for e.g. the read operation when requesting multiple attributes' values. Default: false

`mimeType`

The MIME type to return for the response. By default, this is `text/plain`, but it can be useful for some tools to change it to `application/json`. Init parameters can be used to change the default mime type.

6.4. Object serialization

Jolokia has some object serialization facilities in order to convert complex Java data types to JSON and vice versa. Serialization works in both ways in requests and responses, but the capabilities differ.

Complex data types returned from the agent can be serialized completely into a JSON value object. It can detect cycles in the object graph and provides a way to limit the depth of serialization. For certain types (like `File` or `ObjectName`) it uses simplifier to not expose internal and redundant information.

Object values used for values in *write* operations and arguments in *exec*, type support is limited to a handful of data types.

6.4.1. Response value serialization

Jolokia can serialize any object into a JSON representation when generating the response. It uses some specific converters for certain well known data type with a generic bean converter as fallback.

The following types are directly supported:

- Arrays and `java.util.List` are converted to JSON arrays
- `java.util.Map` gets converted into a JSON object. Note, however, that JSON Object keys are *always strings*.
- `javax.management.openmbean.CompositeData` is converted in a JSON object, with the keys taken from the `CompositeData`'s key set and the value are its values.
- `javax.management.openmbean.TabularData` is serialized into one or multiple nested JSON objects where the keys are derived from its `TabularType.indexName()`. If there is a single valued index, the index's value is the key and a `TabularData`'s row (which in turn is a `CompositeData`) is a map. With multi valued keys, the map is nested (first level: first index's value, second level: second index's value and so on). For the serialization of `TabularData` resulting from a [MXBean](#) translation for maps, see Section 6.4.3, "Jolokia and MXBeans"
- `java.lang.Class` gets converted to a JSON object with keys `name` (the class name) and `interfaces` (the implemented interfaces, if any)
- `java.io.File` becomes a JSON object with keys `name` (file name), `modified` (date of last modification), `length` (file size in bytes), `directory` (whether the file is a directory), `canonicalPath` (the canonical path) and `exists`.
- `javax.management.ObjectName` is converted into a JSON object with the single key `objectName`.
- `java.net.URL` becomes a JSON object with the key `url` containing the URL as String.
- `java.util.Date` is represented in an ISO-8601 format. When used with a path `time` the milliseconds since 1.1.1970 00:00 UTC are returned.
- `org.w3c.dom.Element` is translated into a JSON object with the properties `name`, `value` and `hasChildNodes`.

Primitive and simple types (like String) are directly converted into their string presentation. All objects

not covered by the list above are serialized in JSON objects, where the keys are the public bean properties of the object and the values are serialized (recursively) as described.

Serialization can be influenced by certain processing parameters given with the request (see Section 6.3, “Processing parameters”). I.e. the recursive process of JSON serialization can be stopped when the data set gets too large. Self and other circular references are detected, too. If this happen, special values indicate the truncation of the generated JSON object.

[this]

This label is used when a property contains a self reference

[Depth limit]

When a depth limit is used or the hard depth limit is exceeded, this label contains a string representation of the next object one level deeper. (see Section 6.3, “Processing parameters”, parameter `maxDepth`)

[Reference]

If during the traversal an object is visited a second time, this label is used in order to break the cycle.

[Object limit exceeded]

The total limit of object has been exceeded and hence the object are not deserialized further. (see Section 6.3, “Processing parameters”, parameters `maxCollectionSize` and `maxObjects`)

6.4.2. Request parameter serialization

Serialization in the upstream direction (i.e. when sending values for `write` operations or arguments for `exec` operations) differs from from the object serializaton as used as response values which is described in Section 6.4.1, “Response value serialization”. Not all types are supported for upstream serialization⁵ and the capabilities differ also for POST and GET requests. GET upstream serialization is limited to basic types and simple arrays. POST requests on the other support a much large set of types, including the serialization of `MapS`, `ListS` and all [Open Types](#).

6.4.2.1. GET request values

Since parameters get encoded in the URL for GET request, only the following types can used for values and arguments in `write` and `exec` requests:

- String
- Integer / int
- Long / long
- Byte / byte
- Short / short

⁵ Conversion from a typed system to an untyped representation is obviously much easier than vice versa. Please note, that Jolokia does not replace a full blown JSON object serialization framework like Jackson. Nor does it use one in order to keep the agent small and simple with a low dependency count.

- Float / float
- Double / double
- char
- Boolean / boolean
- Date

The serialized value is simply the string representation of those types. Dates can be set either by an long value (epoch milliseconds) or with a string value (ISO-8601 format). Arrays of the given types are serialized as a comma separated list.

Note

The array support is somewhat limited since it makes a native split on commas. It does not yet take into account any quoting or escaping. For a much safer way to transport arrays to the agent, please consider using POST requests.

Certain *tag values* are used to mark special values. A `null` value has to be serialized as `[null]`, an empty String as `"`. Tag values are not required for POST requests.

6.4.2.2. POST request values

POST request take advantage of the JSON type of the value transferred. These are basic types for numbers (42 or 23.5), booleans (`true` or `false`) and strings ("habanero"). Also, JSON knows about `null` values so no special 'tags' like for GET requests are not required. Since JSON supports intrinsically key-value maps and array types, these can be used directly, too. I.e. if the JMX operation to execute takes a `Map` argument, the argument can be given as a JSON object. Be aware, however, that JSON maps (objects) only support strings as keys.

The agent knows how to convert an JSON array to Java Arrays (of a basic type) or Lists, depending on the requirement as dictated by the MBeans operation or attribute signature. Numbers in JSON are always transferred as long or double values and are as well tried to fit to the MBean's signature. In case of an overflow (e.g. when trying to treat a long with a too large value as int), an exception is raised.

Upstream serialization also supports [OpenTypes](#). If the signature of JMX exec operation or the value type of a JMX attribute is a `OpenType`, they are serialized as follows:

- `SimpleTypes` are extracted from their corresponding JSON type.
- `ArrayType` is extracted from a `JSONArray` where the elements are serialized recursively with this algorithms. Only `ArrayTypes` with element type `CompositeType` or `SimpleType` are supported.
- `CompositeType` is extracted recursively from a `JSONObject` where there the string keys must fit to the `CompositeType`'s item names and the values must be serializable as open types.
- `TabularType` is converted from `JSONObject`. If it is single index (i.e. has only one single index name), the `JSONObject` must have the index values as string keys and the map values are other `JSONObject`s representing the row data. For `TabularTypes` with more than one index name, the incoming `JSONObject` must be a nested object with each index as an additional layer. E.g. the

following JSON object works for a `TabularType` with the two index names `lastname` and `firstname`:

```
{
  "Mann": {
    "Thomas": {
      lastname: "Mann",
      firstname: "Thomas",
      birth: 1875
    },
    "Heinrich": {
      lastname: "Mann",
      firstname: "Heinrich",
      birth: 1871
    }
  }
}
```

`TabularType` used by the MXBean framework for serialization of Maps are translated directly from maps. More details are explained in the next section Section 6.4.3, “Jolokia and MXBeans”.

6.4.3. Jolokia and MXBeans

The [MXBean Framework](#) is available in the JRE since version 6 and allows for easy creation and registration of own MBeans. MXBeans are some what the successor for standard MBeans and support an annotation driven as well as a naming convention driven programming model. The most important difference to standard MBeans is the restriction of MXBean to reference only open types.

Although to the outside only open types are exposed by the MXBean framework, MXBean themselves can use more complex data types. The framework will translate forth and back between the custom and open types according to certain rules as declared in the MXBean [specification](#). Most of the translations to open types fits naturally to Jolokia's serialization, except for the translation of Map.

When an MXBean references a map, the MXBean framework translates this map into a `TabularData` with a fixed internal structure, i.e. with an index `key` and rows with keys `key` and `value`. This leads directly to a JSON representation which is quite artificial. E.g a map with two keys `kind` and `hotness` will be converted by the MXBean framework to a `TabularData` object which in turn would be translated by Jolokia to the following JSON structure

```
{
  "kind" : {
    "key": "kind",
    "value": "Habanero"
  },
  "hotness" : {
    "key": "hotness",
    "value": 10
  }
}
```

Since this representation of a simple map is unnecessarily complicated, Jolokia treats `TabularData` of this kind (i.e. one index `key` and rows with properties `key` and `value`) specially in order to translate it back (and forth) to

```
{
  "kind" : "Habanero",
  "hotness" : 10
}
```

6.5. Tracking historical values

The Jolokia agents are able to keep requested values in memory along with a timestamp. If history tracking is switched on, then the agent will put the list of historical values specific for this request into the response. History tracking is toggled by an MBean operation on a Jolokia-owned MBean (see Chapter 7, *Jolokia MBeans*). This has to be done individually for each attribute or JMX operation to be tracked.

A `history` entry is contained in every response for which history tracking was switched on. A certain JMX operation on an Jolokia specific MBean has to be executed to turn history tracking on for a specific attribute or operation. See Chapter 7, *Jolokia MBeans* for details. The `history` property of the JSON response contains an array of json objects which have two attributes: `value` containing the historical value (which can be as complex as any other value) and `timestamp` indicating the time when this value was current (as measured by the server). Example 6.2, “JSON Response” has an example of a response containing historical values.

For multi attribute read requests, the history entry in the response is a JSON object instead of an array, where this object's attributes are the request's attribute names and the values are the history arrays as described above.

6.6. Proxy requests

For proxy requests, POST must be used as HTTP method so that the given JSON request can contain an extra section for the target which should be finally reached via this proxy request. A typical proxy request looks like

```
{
  "type" : "read",
  "mbean" : "java.lang:type=Memory",
  "attribute" : "HeapMemoryUsage",
  "target" : {
    "url" : "service:jmx:rmi:///jndi/rmi://targethost:9999/jmxrmi",
    "user" : "jolokia",
    "password" : "s!cr!t"
  }
}
```

`url` within the `target` section is a JSR-160 service URL for the target server reachable from within the proxy agent. `user` and `password` are optional credentials used for the JSR-160 communication.

6.7. Jolokia protocol versions

The protocol definition is versioned. It contains of a major and minor version. Changes in the minor

version are backward compatible to other protocol with the same major version. Major version changes incorporate possibly backwards incompatible changes. This document describes the Jolokia protocol version **5.0**

6.1 (since 1.0.2)

Error responses contain now the original request as well, for single and bulk requests.

6.0 (since 1.0.0)

Escaping has been changed from `/-/` to `!/`. This affects GET Urls and *inner paths*.

5.0 (since 0.95)

`javax.management.openmbean.TabularData` is serialized differently when generating the response. In fact, the serialization as an array in the former versions of this protocol is not correct, since `TabularData` in fact is a hash and not a list. It is now generated as map (or multiple maps), depending on the declared *index*. Also, access via path is now an access via key, not a list index. For the special case of MXBean map serialization, where the returned `TabularData` has a fixed format (i.e. with `key` and `value` columns), the `TabularData` is transformed to an appropriate map..

Removed JSON property `modified` from the serialized JSON representation of a File return value since it duplicated the `lastModified` property on the same object.

4.3 (since 0.91)

The `list` operation supports a `maxDepth` option for truncating the answer.

4.2 (since 0.90)

Response values are returned in the native JSON datatype, not always as strings as in previous versions of this protocol. Parameter serialization for writing attribute values or for arguments in `exec` operations has been enhanced for POST requests, which are now represented as native JSON types and not in a string representation as before. GET requests still use a simplified string representation.

4.0 (17.10.2010)

This is the initial version for Jolokia. Versions below 4 are implemented by `jmx4perl`

Chapter 7. Jolokia MBeans

Besides bridging JMX to the HTTP/JSON world, the Jolokia agents also install their own MBeans which provide the extra services described in this chapter.

7.1. Configuration MBean

This MBean, which is registered under the name **jolokia:type=Config**, allows changing configuration parameters. Changes are non-persistent and get lost after a restart of the hosting application server. Debugging mode and the history store can be configured with this MBean.

7.1.1. Debugging

Debugging can be switched on by setting the attribute `Debug`. When debugging is switched on, the Jolokia agent will store debug information in a ring buffer in memory, whose size can be tuned with the attribute `MaxDebugEntries`. The debug information can be fetched by the operation `debugInfo`. This debugging output will contain the JSON responses (which in turn contain their requests) sent to the client. Finally, the operation `resetDebugInfo` clears the debug history.

7.1.2. History store

The *history store* can be used to remember attribute and return values within the agent's memory. The Nagios check **check_jmx4perl**, for instance, uses this feature for its delta check, which measures changes in attribute values. In order to switch on history tracking, two operations are provided:

setHistoryLimitForOperation

JMX operation for switching on tracking of the execution of JMX operations. It takes five arguments: The MBean and operation name, an optional target URL when the agent is used in proxy mode and as limit the number of maximal entries to track and a duration in seconds. If the target URL is given, then request for this specific target are tracked, otherwise, if the URL is null, requests to this operation on the local agent are tracked. The return value of calling this operations is stored in a buffer with the specified length, where the oldest elements will be shifted out in case of an overflow.

setHistoryLimitForAttribute

JMX operation for switching on tracking of an JMX attribute's value. It takes six arguments: The MBean and attribute name, an optional path and target URL and as limit the maximal number of entries to remember and/or an maximum duration for the elements to keep in the history. As above, the target URL is only used for proxy requests. The path can be used to store only read requests with the given path.

There are two kinds of limits which can be applied: Either by a maximum number of historical values to remember or a maximum duration for the values to keep. If both limits are given in a configuration call on the MBean above, both limits are applied. In any case, there are never more values remembered than the global limit which can be set and retrieved with attribute `HistoryMaxEntries`.

The History store can be emptied with a call to the operation `resetHistoryEntries`. This also switches off all history tracking.

If for a request history tracking is switched on, the JSON response will contain an extra field `history` which contains a list with historical values along with the timestamp when it was recorded. This format is described in detail in Section 6.5, “Tracking historical values”.

7.2. Server Handler

The MBean `jolokia:type=ServerHandler` has a single operation `mBeanServersInfo` with no arguments. This operation can be used to dump out the name of all registered MBeans on all found MBeanServers. It is helpful to get a quick and condensed overview of the available JMX information.

Chapter 8. Clients

Three client implementations exist for Jolokia: Jmx4Perl, the Perl binding (the grandmother of all clients ;-), a Java library and a Javascript library. This reference describes the client bindings bundled with Jolokia. More JVM based client libraries are planned for inclusion in Jolokia (e.g. Groovy, Scala or JRuby). Information about Jmx4Perl can be found ???.

8.1. Javascript Client Library

The Jolokia Javascript library provides a Javascript API to the Jolokia agent. It comes with two layers, a basic one which allows for sending Jolokia requests to the agent synchronously or asynchronously and one with a simplified API which is less powerful but easier to use. This library supports bulk requests, HTTP GET and POST requests and JSONP for querying agents which are located on a different server.

All methods of this library are available via the `Jolokia` client object, which needs to be instantiated up-front. In the following example a client object is created and the used heap memory is requested synchronously via the simple API. The agent is deployed within the same webarchive which also serves this script.

```
var j4p = new Jolokia("/jolokia");
var value = j4p.getAttribute("java.lang:type=Memory", "HeapMemoryUsage", "used");
console.log("Heap Memory used: " + value);
```

8.1.1. Installation

The Jolokia Javascript library is distributed in two parts, in compressed and uncompressed forms:

`jolokia.js` and `jolokia-min.js`

Base library containing the Jolokia object definition which carries the `request()`

`jolokia-simple.js` and `jolokia-simple-min.js`

Library containing the Jolokia simple API and which builds up on `jolokia.js`. It must be included after `jolokia.js` since it adds methods to the `Jolokia` object definition.

All four files can be obtained from the [download page](#). For production environments the compressed version is highly recommended since the extensive API documentation included in the original version is stripped off here. For Maven users there is an even better way to integrate them, described in Section 8.1.4, "Maven integration".

`jolokia.js` uses [jQuery](#), which must be included as well. If the target platform doesn't support native JSON serialization, [json2.js](#) needs to be included as well. As sample HTML head for including all necessary parts looks like:

```
<head>
  <script src="jquery-1.4.4.js"></script>
  <script src="json2.js"></script>
  <script src="jolokia-min.js"></script>
```

```
<script src="jolokia-simple-min.js"></script>
</head>
```

A Jolokia client is always created as an instance of `Jolokia`. Requests to the agent are sent by calling methods on this object. The constructing function takes a plain object, which provides default parameters which are used in the `request()` if no overriding are given there.

8.1.2. Usage

All function of this library are available as methods of the `Jolokia` object. The options needs to be instantiated as usual and takes a set of default options, which can be overwritten by subsequent requests. On the most basic layer is a single `request()` method, which takes two arguments: A request object and an optional option object. For example, a synchronous request for obtaining the agent's version for a agent running on the same server which delivered the Javascript looks like:

```
var j4p = new Jolokia({url: "/jolokia"});
var response = j4p.request({type: "version"}, {method: "post"});
console.log("Agent Version: " + response.value.agent);
```

If the constructor is used with a single string argument, this value is considered to be the agent's access URL. I.e. in the example above the construction of the `Jolokia` could have been performed with a single string argument (`new Jolokia("/jolokia")`).

8.1.2.1. Requests

Jolokia requests and responses are represented as JSON objects. They have exactly the same format, which is expected and returned by the agent as defined in Chapter 6, *Jolokia Protocol* for POST requests. All request types are supported.

The `request()` expects as its first argument either a single request object or, for bulk requests, an array of request objects. Depending on this for synchronous operations either a single response JSON object is returned or an array of responses (in the order of the initial request array). For asynchronous request one or more callbacks are called for each response separately. See Section 8.1.2.3, "Operational modes" for details.

The following example shows a single and bulk request call to the Jolokia agent:

```
var j4p = new Jolokia({url: "/jolokia"});
var req1 = { type: "read", mbean: "java.lang:type=Memory", attribute: "HeapMemoryUsage" };
var req2 = { type: "list" };
var response = j4p.request(req1);
var responses = j4p.request([ req1, req2 ]);
```

8.1.2.2. Request options

Each request can be influenced by a set of optional options provided either as default during construction of the `Jolokia` object or as optional last parameter for the request object. The known options are summarized in Table 8.1, "Request options"

Table 8.1. Request options

Key	Description
<code>url</code>	Agent URL (mandatory)
<code>method</code>	Either "post" or "get" depending on the desired HTTP method (case does not matter). Please note, that bulk requests are not possible with "get". On the other hand, JSONP requests are not possible with "post" (which obviously implies that bulk request cannot be used with JSONP requests). Also, when using a <code>read</code> type request for multiple attributes, this also can only be sent as "post" requests. If not given, a HTTP method is determined dynamically. If a method is selected which doesn't fit to the request, an error is raised.
<code>jsonp</code>	Whether the request should be sent via JSONP (a technique for allowing cross domain request circumventing the infamous "same-origin-policy"). This can be used only with HTTP "get" requests.
<code>success</code>	Callback function which is called for a successful request. The callback receives the response as single argument. If no <code>success</code> callback is given, then the request is performed synchronously and gives back the response as return value. The value can be an array of functions which is used for bulk requests to dispatch multiple responses to multiple callbacks. See Section 8.1.2.3, "Operational modes" for details.
<code>error</code>	Callback in case a Jolokia error occurs. A Jolokia error is one, in which the HTTP request succeeded with a status code of 200, but the response object contains a status other than OK (200) which happens if the request JMX operation fails. This callback receives the full Jolokia response object (with a key <code>error</code> set). If no error callback is given, but an asynchronous operation is performed, the error response is printed to the Javascript console by default.
<code>ajaxError</code>	Global error callback called when the Ajax request itself failed. It obtains the same arguments as the error callback given for <code>jQuery.ajax()</code> , i.e. the <code>XmlHttpRequest</code> , a text status and an error thrown. Refer to the jQuery documentation for more information about this error handler.

Key	Description
username	A username used for HTTP authentication
password	A password used for HTTP authentication
timeout	Timeout for the HTTP request
maxDepth	Maximum traversal depth for serialization of complex return values
maxCollectionSize	Maximum size of collections returned during serialization. If larger, the collection is returned truncated.
maxObjects	Maximum number of objects contained in the response.
ignoreErrors	If set to true, errors during JMX operations and JSON serialization are ignored. Otherwise if a single deserialization fails, the whole request returns with an error. This works only for certain operations like pattern reads.

8.1.2.3. Operational modes

Requests can be send either synchronously or asynchronously via Ajax. If a `success` callback is given in the request options, the request is performed asynchronously via an Ajax HTTP request. The callback gets these arguments: a Jolokia JSON response object (see Section 6.1, "Requests and Responses") and an integer index indicating for which response this callback is being called. For bulk requests, this index corresponds to the array index of the request which lead to this response. The value of this option can be an array of callback functions which are called in a round robin fashion when multiple responses are received in case of bulk requests. These callbacks are called only when the returned Jolokia response has a status code of 200, otherwise the callback(s) given with the `error` option are consulted. If no error callback is given, the error is printed on the console by default. As for success callbacks, error callbacks receive the Jolokia error response as a JSON object.

The following example shows asynchronous requests for a single Jolokia request as well as for bulk request with multiple callbacks.

```
var j4p = new Jolokia("/jolokia");

// Single request with a single success callback
j4p.request(
  { type: "read", mbean: "java.lang:type=Memory", attribute: "HeapMemoryUsage" },
  { success: function(response) {
    if (response.value.used / response.value.max > 0.9) {
      alert("90% of heap memory exceeded");
    }
  } },
  { error: function(response) {
    alert("Jolokia request failed: " + response.error);
  } }
);
```

```
// Bulk request with multiple callbacks
j4p.request(
  [
    { type: "read", mbean: "java.lang:type=Threading", attribute: "ThreadCount"},
    { type: "read", mbean: "java.lang:type=Runtime", attribute: ["VmName", "VmVendor"]}
  ],
  { success: [
    function(response) {
      console.log("Number of threads: " + response.value);
    },
    function(response) {
      console.log("JVM: " + response.value.VmName + " -- "
        + response.value.VmVendor);
    }
  ],
  error: function(response) {
    alert("Jolokia request failed: " + response.error);
  }
}
);
```

Both callbacks, `success` and `error`, are only called when the Ajax request succeeds. In case of an error on the HTTP level, the callback `ajaxError` is called with the `XMLHttpRequest`, a `textStatus` and an optional exception object. It has the same signature as the underlying `error` callback of the `jQuery.ajax()` call. (See the [jQuery documentation](#) for details).

The Jolokia agent also supports [JSONP](#) requests for cases where the Jolokia agent is served on a different server or port than the Javascript client. By default, such access is forbidden by the so called *same-origin-policy*. To switch on JSONP, the option `jsonp` should be set to `"true"`.

As explained in Section 6.1, “Requests and Responses” the Jolokia agent supports two HTTP methods, `GET` and `POST`. `POST` is more powerful since it supports more features. e.g. bulk requests and JMX proxy requests are only possible with `POST`. By default, the Jolokia Javascript library selects an HTTP method automatically, which is `GET` for simple cases and `POST` for more sophisticated requests. The HTTP method can be overridden by setting the option `method` to `"get"` or `"post"`.

There are some limitations in choosing the HTTP method depending on the request and other options given:

- Bulk requests (i.e. an array of multiple requests) can only be used with `POST`.
- `READ` requests for multiple attributes (i.e. the `attribute` request parameter is an array of string values) can only be used with `POST`.
- The JMX proxy mode (see Chapter 5, *Proxy Mode*) can only be used with `POST`.
- JSONP can only be used with `GET` and only in asynchronous mode (i.e. a `success` callback must be given). This is a limitation of the JSONP technique itself.

The restrictions above imply, that JSONP can only be used for single, simple requests and not for JMX proxy calls.

8.1.3. Simple API

Building upon the basic `Jolokia.request()` method, a simplified access API is available. It is contained in `jolokia-simple.js` which must be included after `jolokia.js`. This API provides dedicated method for the various request types and supports all options as described in Table 8.1, “Request options”. There is one notable difference for asynchronous callbacks and synchronous return values though: In case of a successful call, the callback is fed with the response's `value` object, not the full response (i.e. `response.value`). Similar, for synchronous operations the value itself is returned. In case of an error, either an `error` callback is called with the full response object or an `Error` is thrown for synchronous operations.

`getAttribute(mbean,attribute,path,opts)`

This method returns the value of an JMX attribute `attribute` of an MBean `mbean`. A path can be optionally given, and the optional request options are given as last argument(s). The return value for synchronous operations are the attribute's value, for asynchronous operations (i.e. `opts.success != null`) it is `null`. See Section 6.2.1, “Reading attributes (read)” for details.

For example, the following method call can be used to synchronously fetch the current heap memory usage:

```
var memoryUsed = j4p.getAttribute("java.lang:type=Memory", "HeapMemoryUsage", "used");
```

`setAttribute(mbean,attribute,value,path,opts)`

For setting an JMX attribute, this method takes the MBean's name `mbean`, the attribute `attribute` and the value to set as `value`. The optional `path` is the *inner path* of the attribute on which to set the value (see Section 6.2.2, “Writing attributes (write)” for details). The old value of the attribute is returned or given to a `success` callback.

To enable verbose mode in the memory-handling beans, use

```
var gsLoggingWasOn = j4p.setAttribute("java.lang:type=Memory", "Verbose", true);
```

`execute(mbean,operation,arg1,arg2,...,opts)`

With this method, a JMX operation can be executed on the MBean `mbean`. Beside the operation's name `operation`, one or more arguments can be given depending on the signature of the JMX operation. The return value is the return value of the operation. See Section 6.2.3, “Executing JMX operations (exec)” for details.

The following example asynchronously fetches a thread dump as a JSON object and logs it into the console:

```
j4p.execute("java.lang:type=Threading", "dumpAllThreads", true, true,
  {
    success: function(value) {
      console.log(JSON.stringify(value));
    }
  });
```

`search(mBeanPattern,opts)`

Searches for one or more MBeans whose object names fit the pattern `mBeanPattern`. The return

value is a list of strings with the matching MBean names or `null` if none is found. See Section 6.2.4, “Searching MBeans (search)” for details.

The following example looks up all application servers available in all domains:

```
var appServerNames = j4p.search("*:j2eeType=J2EEServer,*");
```

list(path,opts)

For getting meta information about registered MBeans, the `list` command can be used. The optional `path` points into this meta information for retrieving partial information. The format of the return value is described in detail in Section 6.2.5, “Listing MBeans (list)”.

This example fetches only the meta information for the attributes of the `java.lang:type=OperatingSystem` MBean:

```
var attributesMeta = j4p.list("java.lang/type=OperatingSystem/attr");
```

version(opts)

The `version` method returns the agent's version, the protocol version, and possibly some additional server-specific information. See Section 6.2.6, “Getting the agent version (version)” for more information about this method.

A sample return value for a Glassfish server looks like:

```
{
  protocol: "4.0",
  agent: "0.82",
  info: {
    product: "glassfish",
    vendor: "Sun",
    extraInfo: {
      amxBooted: false
    }
  }
}
```

8.1.4. Maven integration

For maven users' convenience, the Jolokia Javascript package is also available as a JavaScript artifact. It can be easily included with help of the `javascript-maven-plugin`.

Recommended plugin

Unfortunately, the "official" version of this plugin hosted on Codehaus has been stuck at version `1.0-alpha-1-SNAPSHOT`. Although it is quite usable, in order to avoid a snapshot dependency, it is recommended to use a fork of this plugin hosted on [GitHub](#) and deployed at the [Sonatype Maven repository](#).

The following example shows a sample configuration which could be used within a `pom.xml`:

```

<project>
...
  <dependencies>
    <dependency>
      <groupId>org.jolokia</groupId>
      <artifactId>jolokia-client-javascript</artifactId>
      <type>javascript</type>
      <version>0.82</version>
    </dependency>
    ....
  </dependencies>

  <build>
    <plugins>
      <plugin>
        <groupId>com.devspan.mojo.javascript</groupId>
        <artifactId>javascript-maven-plugin</artifactId>
        <version>0.9.3</version>
        <extensions>true</extensions>
        <configuration>
          <useArtifactId>>false</useArtifactId>
        </configuration>
        <executions>
          <execution>
            <goals>
              <goal>war-package</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
      .....
    </plugins>
    ....
  </build>
  ...
  <pluginRepositories>
    <pluginRepository>
      <id>sonatype-oss</id>
      <url>https://oss.sonatype.org/content/groups/public</url>
    </pluginRepository>
  </pluginRepositories>
</project>

```

Then, in your webapp project, `jolokia.js`, `jolokia-simple.js` and `json2.js` can be found in the `scripts/lib` directory (relative to the top level of you WAR). In order to include it in your HTML files use something like this:

```

<head>
  <script src="jquery-1.4.4.js"></script>
  <script src="scripts/lib/json2.js"></script>
  <script src="scripts/lib/jolokia.js"></script>
  <script src="scripts/lib/jolokia-simple.js"></script>
</head>

```

`jQuery.js` has to be included on its own, though and is not included within the dependency. If the compressed version of `jolokia.js` should be used, add a `classifier="compressed"` to the

`jolokia-client-javascript` dependency, and include `scripts/lib/jolokia-min.js`

A full working example can be found in the Jolokia src at `client/javascript/test-app/pom.xml`.

8.2. Java Client Library

The Java client library provides an easy access to the Jolokia agent from within Java. Since JSR-160 connectors themselves provide Java based remote access to MBeans one might wonder about the benefits of a Jolokia Java binding. There are several, though:

- It provides a typeless access to remote MBeans. The big advantage is that for any non-OpenMBean access to custom typed objects is still possible without having the type information locally in the classpath.
- Jolokia can be used in setups where JSR-160 connectors can not be used. I.e. in firewall secured environments it is much easier to get through to a Jolokia Agent than to an JSR-160 connector using RMI as transport protocol.
- *Remoteness* is explicit in this API instead of JSR-160 connector's seeked *transparent remoteness*. RMI has some arguable conceptual advantages, but hiding all remote aspects proved to have quite some disadvantages when it comes to the programming model. Explicit awareness of a 'heavy-weight' remote call is better than false transparency in order to know the price tag.

The Java client library follows a strict request-response paradigm, much like the underlying HTTP. It uses generics heavily and can be centered around three classes: `J4pClient` is the client side agent, which has various variants of a `execute()` for sending requests. This method takes one or more `J4pRequest` objects as arguments and returns one or more `J4pResponse` objects as result.

What the heck is this 'J4p' ?

That's a reminiscence to Jolokia's roots which lies in [Jmx4Perl](#). It is always good to remember where one comes from ;-)

But before we got into the details, the next section gives a first tutorial to get a feeling how the API can be used.

8.2.1. Tutorial

As seen in the following example, the usage is quite easy. First a client object `client` is created pointing to a Jolokia agent at `http://localhost:8080/jolokia`. A read request for querying the heap memory usage from the `MemoryMXBean` is created and then send via the `execute()` to the agent. The response returned is of type `J4pReadResponse` and holds the result which finally is printed out to standard output.

```
import org.jolokia.client.J4pClient;
import org.jolokia.client.request.*;

public class MemoryDemo {
    public static void main(String[] args) {
        J4pClient client = new J4pClient("http://localhost:8080/jolokia");
        J4pReadRequest request =
```

```

        new J4pReadRequest("java.lang:type=Memory", "HeapMemoryUsage", "used");
        J4pReadResponse response = client.execute(request);
        System.out.println("Memory used: " + response.getValue());
    }
}

```

In order to compile and run this sample, two support libraries are required in addition to `jolokia-client-java.jar` ([Download](#)):

- Apache [HttpClient](#), 4.1 ([Download](#))
- [json-simple](#), 1.1 ([Download](#))

For maven users, the following dependency is sufficient (it will include the other two as transitive dependencies):

```

<dependency>
  <groupId>org.jolokia</groupId>
  <artifactId>jolokia-client-java</artifactId>
  <version>1.0.0</version>
</dependency>

```

8.2.2. J4pClient

`J4pClient` is the entry point for sending requests to a remote Jolokia agent. It can be created in multiple ways. For simple cases, public constructors are provided taking the mandatory Jolokia agent URL and optionally a `org.apache.http.client.HttpClient` instance which is used for the HTTP business. The recommended style is to use the `J4pClientBuilder`, though. This way, all parameters for the HTTP communication can easily be set:

```

J4pClient j4p = J4pClient.url("http://localhost:8080/jolokia")
    .user("roland")
    .password("s!cr!t")
    .connectionTimeout(3000)
    .build();

```

The builder supports the following parameters with the given defaults:

Table 8.2. J4pClient parameters

Parameter	Description	Default
<code>url</code>	The URL to the Jolokia agent. This is the only mandatory parameter.	
<code>user</code>	User name when authentication is used. If not set, no authentication is used. If set, <code>password</code> must be set, too	

Parameter	Description	Default
password	Password used for authentication. Only used when <code>user</code> is set.	
target	A JMX JSR-160 ServiceURL which should be used by the agent as the <i>real</i> target. This parameter should be set if the client is used for accessing the agent in Chapter 5, <i>Proxy Mode</i> .	
targetUser	The JSR-160 user to use when using the proxy mode. If not given (and <code>target</code> is set), then no authentication is used for JSR-160 communication.	
targetPassword	JSR-160 Password to use for the proxy mode.	
connectionTimeout	The timeout in milliseconds until a connection is established. A timeout value of zero is interpreted as an infinite timeout.	20000
pooledConnection	Specifies, that the underlying HttpClient should use pooled connection manager, which is thread safe and can service connection requests from multiples threads simultaneously. This is important if the <code>J4pClient</code> is to be used in a multi threaded context. The size of the pool is restricted by the parameter <code>maxTotalConnection</code> . <code>ThreadSafeClientConnManager</code> is the underlying connection manager. Pooled connections are the default.	
singleConnection	Specifies that single connection should be used which maintains only one active connection at a time. Even though <code>J4pClient</code> is still thread-safe it ought to be used by one execution thread only. The underlying connection manager is <code>SingleClientConnManager</code> .	

Parameter	Description	Default
	Pooled connections are the default.	
<code>maxTotalConnections</code>	Defines the number of total connections to be pooled. It is only used when <code>pooledConnection</code> is used.	20
<code>maxConnectionPoolTimeout</code>	Defines the timeout for waiting to obtain a connection from the pool. This parameter is only used when <code>pooledConnections</code> are used.	500
<code>socketTimeout</code>	Defines the socket timeout (<code>SO_TIMEOUT</code>) in milliseconds, which is the timeout for waiting for data or, put differently, a maximum period inactivity between two consecutive data packets. A timeout value of zero is interpreted as an infinite timeout.	0
<code>contentCharset</code>	Defines the charset to be used per default for encoding content body.	ISO-8859-1
<code>expectContinue</code>	Activates <code>Expect: 100-Continue</code> handshake for the entity enclosing methods. The purpose of the <code>Expect: 100-Continue</code> handshake to allow a client that is sending a request message with a request body to determine if the origin server is willing to accept the request (based on the request headers) before the client sends the request body. The use of the <code>Expect: 100-continue</code> handshake can result in noticeable performance improvement for entity enclosing requests that require the target server's authentication.	true
<code>tcpNoDelay</code>	Determines whether Nagle's algorithm is to be used. The Nagle's algorithm tries to conserve bandwidth by minimizing the number of	true

Parameter	Description	Default
	segments that are sent. When applications wish to decrease network latency and increase performance, they can disable Nagle's algorithm (that is enable <code>TCP_NODELAY</code>). Data will be sent earlier, at the cost of an increase in bandwidth consumption.	
<code>socketBufferSize</code>	Determines the size of the internal socket buffer in bytes used to buffer data while receiving and transmitting HTTP messages.	8192

The `J4pClient` provides various variants of a `execute()` method, which takes either one single request or a list of requests. For a single request, the preferred HTTP method (GET or POST) can be specified optionally. The `List<R>` argument can be used type only for a homogenous bulk request, i.e. for multiple requests of the same time. Otherwise an untyped list must be used.

Each request can be tuned by giving a map of processing options along with their values to the `execute` method. The possible options are shown in table Table 8.3, "J4pClient query parameters".

Table 8.3. J4pClient query parameters

J4pQueryParameter enum	Description
<code>MAX_DEPTH</code>	Maximum traversal depth for serialization of complex objects. Use this with a "list" request to restrict the depth of the returned meta data tree.
<code>MAX_COLLECTION_SIZE</code>	Maximum size of collections returned during serialization. If larger, a collection is truncated to this size.
<code>MAX_OBJECTS</code>	Maximum number of objects returned in the response's value.
<code>IGNORE_ERRORS</code>	Option for ignoring errors during JMX operations and JSON serialization. This works only for certain operations like pattern reads and should be either <code>true</code> or <code>false</code> .

8.2.3. Request types

For each request type a dedicated request object is provided which all are subclasses from `J4pRequest`. For all requests it can be specified which HTTP method is to be used by setting the property `preferredHttpMethod` to either `GET` or `POST`. Each request type has a corresponding response type which used for the return values of the `J4pClient.execute()`.

The constructor of each kind of request can take a `J4pTargetConfig` as argument for using a request in Chapter 5, *Proxy Mode*. This configurational object holds the JMX service url and (optionally) credentials for JSR-160 authentication. When given, this proxy target specification overrides any default proxy configuration set during the initialization of the `J4pClient`.

`J4pReadRequest` and `J4pReadResponse`

`J4pReadRequest` is a read request to get one or more attributes from one or more MBeans within a single request. Various constructor variants can be used to specify one or more attributes along with the `ObjectName` (which can be a pattern). A `path` can be set as property for specifying an *inner path*, too.

`J4pReadResponse` is the corresponding response type and allows typed access to the fetched value for a single attribute fetch or to multiple values for a multi attribute read. In the latter case, the found object and attributes names can be retrieved as well.

For more information on fetching the value of multiple attributes and multiple MBeans at once, please refer to Section 6.2.1, “Reading attributes (read)” or the Javadoc of `J4pReadResponse`.

`J4pWriteRequest` and `J4pWriteResponse`

A `J4pWriteRequest` is used to set the value of an MBean attribute. Beside the mandatory object and attribute name the value must be given in the constructor as well. Optionally a `path` can be provided, too. Only certain types for the given value can be serialized properly for calling the Jolokia agent as described in Section 6.4.2, “Request parameter serialization”.

The old value is returned as `J4pWriteResponse`'s value.

`J4pExecRequest` and `J4pExecResponse`

`J4pExecRequest`s are used for executing operation on MBeans. The constructor takes as mandatory arguments the MBean's object name, the operation name and any arguments required by the operation. Only certain types for the given arguments can be serialized properly for calling the Jolokia agent as described in Section 6.4.2, “Request parameter serialization”.

The returned `J4pExecResponse` contains the return value of the operation called.

`J4pSearchRequest` and `J4pSearchResponse`

A `J4pSearchRequest` contains a valid single MBean object name pattern which is used for searching MBeans.

The `J4pSearchResponse` holds a list of found object names.

`J4pListRequest` and `J4pListResponse`

For obtaining meta data on MBeans a `J4pListRequest` should be used. It can be used with a *inner path* to obtain only a subtree of the response, otherwise the whole tree as described in Section 6.2.5.3, “List response” is returned. With the query parameter `maxDepth` can be used to restrict the depth of returned tree.

The single value of a `J4pListResponse` is a tree (or subtree) as a JSON object, which has the format described in Section 6.2.5.3, “List response”.

`J4pVersionRequest`

A `J4pVersionRequest` request the Jolokia agent's version information and takes no argument.

The `J4pVersionResponse` returns the agent's version (`agentVersion`), the protocol version

`protocolVersion`), the application server product name (`product`), the vendor name (`vendor`) and any extra info (`extraInfo`) specific to the platform the Jolokia is running on.

8.2.4. Exceptions

In case of an error when executing a request a `J4pException` or one its subclass is thrown.

`J4pConnectException`

Exception thrown when the connection to the server fails. It contains the original `ConnectException` as nested value.

`J4pTimeoutException`

Exception thrown in case of an timeout. The nested exception is of type `ConnectTimeoutException`.

`J4pRemoteException`

Generic exception thrown when an exception occurred on the remote side. This is the case when the JSON response obtained is an error response as described in Section 6.1.3, "Responses". The error type, the status, the request leading to this error and the remote stacktrace (as string) can be obtained from this exception.

`J4pBulkRemoteException`

Exception thrown when a bulk request fails on the remote side. This contains a mixed list which contains the `J4pRemoteException` occurred as well as the `J4pResponse` objects for the requests, which succeeded. The list obtained by `getResults()` contains these objects in the same order as the list of requests given to `execute`. All responses und remote exceptions can also be obtained seperately in homogenous lists.

`J4pException`

Base exception thrown, when no other exception fits, i.e. when the exception happened on the client side. The original exception is contained as nested exception.

Chapter 9. Tools

Various tools complete the Jolokia portfolio. Some of the are available under the Jolokia umbrella, some of them are hosted elsewhere. This chapter gives an overview of this tool landscape.

9.1. Jmx4Perl

9.2. Jolokia Roo Addon

The Jolokia [Roo](#) addon allows for easy integration of an agent servlet in an existing Roo web project.

Note

This addon has been submitted to the Roobot, a central Roo addon registry. Until it is publicly available you can directly install the addon from our repository.

```
roo> osgi obr url add --url http://labs.consol.de/maven/repository/roo-repository.xml
roo> osgi obr start --bundleSymbolicName org.jolokia.roo
```

Alternatively, if there are problems with the approach above (which is currently the case because the hard coded public PGP keyserver which is used by Roo 1.1.1 is down) and you don't need PGP verification, you can install the addon bundle directly from our repository:

```
roo> osgi start --url http://labs.consol.de/maven/repository/org/jolokia/jolokia-roo/0.83/
```

As soon as `src/main/webapp/WEB-INF/web.xml` is available in the roo project, Jolokia can be setup with the command **jolokia setup**. This will add the proper dependency in the `pom.xml` and adapt `web.xml` so that an agent servlet gets registered under the subcontext `jolokia` (so, when you web application is deployed under the context `/mywebapp`, the agent is reachable under `/mywebapp/jolokia`). This command knows about the options described in Table 9.1, “jolokia setup Options”, all of which are optional.

Table 9.1. jolokia setup Options

Option	Description
<code>--addPolicy</code>	This adds an additional <code>jolokia-access.xml</code> below <code>src/main/resources</code> to allow putting access restrictions into place. The installed template, however, doesn't come with any restriction but contains sample configurations commented out.
<code>--addJsrl60Proxy</code>	Adapts the agent servlet's <code>init-param</code> to add an

Option	Description
--addDefaultInitParams	<p>additional <code>org.jolokia.jsr160.Jsr160RequestDispatcher</code> request dispatcher which allows the installed servlet to act as an JSR-160 proxy. See Chapter 5, <i>Proxy Mode</i> for details about the JSR-160 proxy</p> <p>By default, the servlet gets registered without any init parametes. With this option, all available <code>init-param</code> are added to the servlet definition with their default values.</p>
