SimpleJMX Package

Version 1.18
May 2021

Gray Watson

Table of Contents

SimpleJMX

Version 1.18 – May 2021

This package provides classes that simplify the publishing of objects using Java's Management Extensions (JMX). These published objects can be investigated with jconsole or another JMX client. Included is also a programmatic JMX client which you can use to connect to and interrogate remote JMX servers as well as a web/HTTP interface so you can publish JMX beans to a browser or other web client.

To get started quickly using SimpleJMX, see Chapter 1 [Quick Start], page 2. You can also take a look at the examples section of the document which has various working code packages. See Chapter 3 [Examples], page 14. There is also a HTML version of this documentation.

Gray Watson http://256stuff.com/gray/

1 Start Using Quickly

To use SimpleJMX you need to do the following steps. For more information, see Chapter 2 [Using], page 3.

1. Download SimpleJMX from the SimpleJMX release page. See Section 2.1 [Downloading], page 3.
2. Add `@JmxResource` annotation to the top of each class you want to publish via JMX. See Section 2.2 [Naming Objects], page 3.

   ```
   @JmxResource(domainName = "your.domain", description = "Runtime counter")
   public class RuntimeCounter {
   ```

3. Add `@JmxAttributeField` annotation to each of the attribute fields that you want to expose via reflection over JMX. See Section 2.3 [Expose Items], page 5.

   ```
   @JmxAttributeField(description = "Start time in millis",
      isWritable = true)
   private long startTimeMillis;
   ```

   If you want to annotate the get/set/is attribute methods instead then use the `@JmxAttributeMethod` annotation on those methods. See Section 2.3.2 [JmxAttributeMethod Annotation], page 5.

   ```
   @JmxAttributeMethod(description = "Run time in seconds or milliseconds")
   public long getRunTime() {
   ```

4. Add `@JmxOperation` annotation to each of operation methods that you want to make accessable over JMX. See Section 2.3.3 [JmxOperation Annotation], page 6.

   ```
   @JmxOperation(description = "clear the cache")
   public void clearCache() {
   ```

5. Start your JMX server. See Section 2.4 [Start Server], page 7.

   ```
   // create a new JMX server listening on a port
   JmxServer jmxServer = new JmxServer(8000);
   jmxServer.start();
   ```

6. Register the objects you want to publish via JMX. See Section 2.5 [Register Objects], page 7.

   ```
   jmxServer.register(someObject);
   jmxServer.register(someOtherObject);
   ```

For more extensive instructions, see Chapter 2 [Using], page 3.

2 Using SimpleJMX

2.1 Downloading Jar

To get started with SimpleJMX, you will need to download the jar file. The SimpleJMX release page is the default repository but the jars are also available from the central maven repository.

The code works with Java 6 or later.

2.2 Naming Objects

When you publish an object you need to tell JMX what the object's *unique* name is and where it should be located in the various folders shown by jconsole. There are a couple of different ways to do this with SimpleJMX:

2.2.1 @JmxResource Annotation

The `@JmxResource` annotation is used to define that the resource is to be published via JMX and how it is to be named.

```
@JmxResource(domainName = "your.domain", description = "Runtime counter")
public class RuntimeCounter {
    ...
```

The above example shows that the `RunetimeCounter` object is to be published via JMX inside of the `your.domain` folder with a text description.

The fields in `@JmxResource` are:

`domainName`
: Domain name of the object which turns into the top-level folder inside of jconsole. This must be specified unless the object is self-naming. See Section 2.2.2 [Self Naming], page 4.

`beanName`
: Name of the JMX object in the jconsole folder. The default is to use the class name.

`folderNames`
: Optional array of strings which translate into sub-folders below the domain-name folder. Default is for the object to show up in the domain-name folder. The folder names can either be in `name=value` format in which case they should be in alphabetic order by name. They can also just be in `value` format.

  Folders are used when you have a large number of JMX objects being published and you want to group the objects so that you can find them faster than scrolling through a large list. For example, all of your database objects could go in the folder "database" while the database connections could go into the sub-folder "database/connections".

```
@JmxResource(domainName = "your.domain",
    folderNames = { "database", "connections" })
public class MySqlDatabaseConnection {
```

`description`
: Textual description of the class for jconsole or other JMX clients. Default is something like: "Information about class-name".

2.2.2 Self Naming Objects

Instead of using the `@JmxResource` annotation to define the name/folders for your JMX object, you can have the object implement the `JmxSelfNaming` interface. This allows the object to name itself and will override any settings from the `@JmxResource` annotation, if it is specified.

It is particularly necessary to make your object `JmxSelfNaming` if there are to be multiple of them published via JMX. For example, if you have multiple database connections that you want to publish then to ensure that they have a *unique* name, each of the objects should be self-naming and should provide a unique name that identifies itself:

```
// we only use this to set the domain name and the folders
@JmxResource(domainName = "your.domain",
   folderNames = { "database", "connections" })
public class DatabaseConnection extends BaseJmxSelfNaming
      implements JmxSelfNaming {
   @Override
   public String getJmxBeanName() {
      // return our toString as our name
      return toString();
   }
}
```

In the above example, we extend the `BaseJmxSelfNaming` abstract class which has default implementations for all of the `JmxSelfNaming` methods, so all we need to do is override what we want to change.

The methods in the `JmxSelfNaming` interface are:

`String getJmxDomainName();`
: Return the domain name of the object. Return null to use the one from the `@JmxResource` annotation instead.

`String getJmxBeanName();`
: Return the name of the object. Return null to use the one from the `@JmxResource` annotation instead.

`JmxFolderName[] getJmxFolderNames();`
: Return the appropriate array of folder names used to built the associated object name. Return null for no folders in which case the bean will be at the top of the hierarchy in jconsole without any sub-folders.

2.3 Exposing Fields and Methods

Once we have named our object, we need to tell the JMX server which fields and methods should be exposed to the outside world. JMX can expose what it calls attributes, operations, and notifications. At this time, only attributes and operations are supported by SimpleJmx.

Attributes can be primitives or simple types such as `String` or `java.util.Date`. With SimpleJMX you can expose them by using reflection on the object's fields directly using the `@JmxAttributeField` annotation or instead via the get/set/is methods using the `@JmxAttributeMethod` annotation.

Operations are methods that do *not* start with get/set/is but which perform some function (ex: `resetTimer()`, `clearCache()`, etc.). They can be exposed with the `@JmxOperation` annotation.

2.3.1 @JmxAttributeField Annotation

SimpleJMX allows you to publish your primitive or simple types by annotating your fields with the `@JmxAttributeField` annotation.

```
@JmxAttributeField(description = "Start time in millis",
   isWritable = true)
private long startTimeMillis;
```

In the above example, the `startTimeMillis` long field will be visible via JMX. It will show its value which can be changed because `isWriable` is set to true. `isReadable` is set to true by default. The description is available in jconsole when you hover over the attribute.

The fields in the `@JmxAttributeField` annotation are:

`String description`
: Description of the attribute for jconsole. Default is something like: "someField attribute".

`boolean isReadible`
: Set to false if the field should not be read through JMX. Default is true.

`boolean isWritable`
: Set to true if the field can be written by JMX. Default is false.

2.3.2 @JmxAttributeMethod Annotation

Instead of publishing the fields directly, SimpleJMX also allows you to publish your attributes by decorating the get/set/is methods using the `@JmxAttributeMethod` annotation. This is *only* for methods that start with `getXxx()`, `setXxx(...)`, or `isXxx()`.

The `Xxx` name should match precisely to line up the get and set JMX features. For example, if you are getting and setting the `fooBar` field then it should be `getFooBar()` and `setFooBar(...)`. `isFooBar()` is also allowed if `foobar` is a `boolean` or `Boolean` field.

Notice that although the field-name is `fooBar` with a lowercase 'f', the method name camel-cases it and turns it into `getFooBar()` with a capital 'F'. In addition, the `getXxx()` method must not return void and must have no arguments. The `setXxx(...)` method must

return `void` and must take a single argument. The `isXxx()` method is allowed if it returns boolean or Boolean and the method has no arguments.

Exposing a get method allows you to do some data conversion when the value is published. Exposing a set method allows you to do data validation.

```
@JmxAttributeMethod(description = "Run time in seconds or milliseconds")
public long getRunTime() {
```

The only field in the `@JmxAttributeMethod` annotation is the description. The annotation on the `get...` method shows that it is readable and the annotation on the `set...` method shows that it is writable.

2.3.3 @JmxOperation Annotation

Operations are methods that do *not* start with get/set/is but which perform some function. They can be exposed with the `@JmxOperation` annotation. The method can either return `void` or an object. It is recommended that the method return a primitive or a simple object that is in Jconsole's classpath already. Otherwise Jconsole will be unable to display it. It also should not throw an unknown `Exception` class.

```
@JmxOperation(description = "clear the cache")
public void clearCache() {
   ...
```

A pattern that is common is to return a `String` from the method to provide some feedback to the remote user and to catch and return any exceptions as a `String`.

```
@JmxOperation(description = "clear the cache")
public void clearCache() {
   try {
      // do the cache clearing here
      return "Cache cleared";
   } catch (Exception e) {
      return "Threw exception: " + e;
   }
}
```

The fields in the `@JmxOperation` annotation are:

`String description`
: Description of the attribute for jconsole. Default is something like "someMethod operation".

`String[] parameterNames`
: Optional array of strings which gives the name of each of the method parameters. The array should be the same length as the `parameterDescriptions` array. Default is something like "p0".

```
@JmxOperation(parameterNames = { "minValue", "maxValue" },
   parameterDescriptions = { "low water mark",
      "high water mark" })
public void resetMaxMin(int minValue, int maxValue) {
   ...
```

`String[] parameterDescriptions`
: Optional array of strings which describes each of the method parameters. The array should be the same length as the `parameterNames` array.

`OperationAction operationAction`
: This optional field is used by the JMX system to describe what sort of work is being done in this operation.

2.4 Starting a JMX Server

The `JmxServer` class proves a server that jconsole and other JMX clients can connect to. The easiest way to do this is to choose a port to use, define the server, and then start it:

```
// create a new JMX server listening on a port
JmxServer jmxServer = new JmxServer(8000);
// start our server
jmxServer.start();
```

Instead of starting your own server, you can have the `JmxServer` make use of the `MBeanServer` that was started by the JVM platform. It gets the server from a call to `ManagementFactory.getPlatformMBeanServer()`.

```
// define a server that makes use of the platform MBeanServer
JmxServer jmxServer = new JmxServer(true);
```

Before your program exits, it is best to stop the server, so the following try/finally block is a good pattern to use:

```
// create a new JMX server listening on a port
JmxServer jmxServer = new JmxServer(8000);
try {
   // start our server
   jmxServer.start();
   ...
   // register objects with the server and do other stuff here
   ...
} finally {
   // un-register objects
   // stop our server
   jmxServer.stop();
}
```

2.5 Register Objects

To published objects via the server via JMX you must register them with the `JmxServer`:

```
jmxServer.register(someObject);
```

There also is an `unregister(...)` method which will un-publish from the server:

```
jmxServer.unregister(someObject);
```

The objects that are registered must be named and the fields and methods that are to be exposed must be specified.

The `register(...)` and `unregister(...)` methods allow you to publish JMX information about dynamic objects that get created and destroyed at runtime. For example, if you want to see details about an attached client connection, you could do something like the following:

```
// accept a connection
Socket socket = serverSocet.accept();
ClientHandler handler = null;
try {
   // create our handler
   handler = new ClientHandler(socket);
   // register it via jmx
   jmxServer.register(handler);
   // handle the connection
   handler.handle();
} finally {
   socket.close();
   if (handler != null) {
      // unregister it from JMX
      jmxServer.unregister(handler);
   }
}
```

2.6 Publishing Using Code Definitions

Sometimes, you want to expose a class using JMX but you don't control the source code or maybe you don't want to put the SimpleJMX annotations everywhere. If this is the case then you also have the option to expose just about any object programmatically.

The `JmxServer` has a register function which takes just an `Object`, an `ObjectName` which can be generated with the use of the `ObjectNameUtil` class, and an array of attribute-fields, attribute-methods, and operations.

The object-name can also be defined using the `JmxResourceInfo` object that defines the fields in the `@JmxResource` programmatically.

```
JmxResourceInfo resourceInfo =
   new JmxResourceInfo("your.domain", "beanName",
      new String[] { "database", "connections" },
      "your resource desription"),
```

The attribute-fields are specified as an array of `JmxAttributeFieldInfo` objects that are associated with fields that are exposed through reflection:

```
JmxAttributeFieldInfo[] attributeFieldInfos =
   new JmxAttributeFieldInfo[] {
     new JmxAttributeFieldInfo("startMillis", true /* readable */,
```

```
            false /* not writable */, "When our timer started"),
      };
```

The attribute-methods are specified as an array of `JmxAttributeMethodInfo` objects that are associated with fields that are exposed through get/set/is methods:

```
JmxAttributeMethodInfo[] attributeMethodInfos =
   new JmxAttributeMethodInfo[] {
      new JmxAttributeMethodInfo("getRunTime",
         "Run time in seconds or milliseconds"),
   };
```

The operations are specified as an array of `JmxOperationInfo` objects that are associated with operation methods:

```
JmxOperationInfo[] operationInfos =
   new JmxOperationInfo[] {
      new JmxOperationInfo("restartTimer", null /* no params */,
         null  /* no params */, OperationAction.UNKNOWN,
         "Restart the timer"),
   };
```

To register the object you would then do:

```
jmxServer.register(someObject,
   ObjectNameUtil.makeObjectName("your.domain", "SomeObject"),
   attributeFieldInfos, attributeMethodInfos, operationInfos);
```

For more information, take a look at the random-object example program. See [random object example], page 14.

You can also use the `PublishAllBeanWrapper` which exposes all public fields and methods as attributes or operations. All public fields will be exposed as an attribute and if not-final will be exposed as writable. All public methods that start with `is` or `get` will be exposed as attributes and if they have a `set` method will be writable. All other public methods will be exposed as operations.

Take a look at the publish-all example program to see working code that uses this class. See [publish all example], page 14.

2.7 Using the JMX Client

SimpleJMX also includes a programmatic, simple JMX client which you can use to interrogate JMX servers. You connect to the server by specifying the host and port.

```
JmxClient client = new JmxClient("server1", 8000);
```

To get a list of the available beans use the

```
Set<ObjectName> beanNames = jmxClient.getBeanNames();
```

Then you can get the attributes and operations associated with an `ObjectName`:

```
MBeanAttributeInfo[] attributeInfos =
   jmxClient.getAttributesInfo(objectName);
MBeanOperationInfo[] operationInfos =
   jmxClient.getOperationsInfo(objectName);
```

You can then get an attribute from the info or invoke an operation:

```
boolean showSeconds =
    jmxClient.getAttribute(objectName, attributeInfo.getName());
client.invokeOperation(objectName, operationInfo.getName());
```

You can also get attributes by strings instead of `ObjectNames`:

```
boolean showSeconds =
   jmxClient.getAttribute("your.domain", "RuntimeCounter",
      "showSeconds");
```

If you need to construct the object name directly then you can use `ObjectName` static methods. This is particularly useful if you use Jconsole to find a bean that you want to operate on via the `JmxClient`. Just copy the `ObjectName` field associaed to a bean from Jconsole and then do something like the following:

```
int availableProcessors =
   client.getAttribute(
      ObjectName.getInstance("java.lang:type=OperatingSystem"),
      "AvailableProcessors");
```

You can also call operations with string arguments:

```
client.invokeOperation(ObjectName.getInstance("java.lang:type=Memory"),
   "gc");
```

See the `JmxClient` javadocs for more information

2.8 Using With the Spring Framework

SimpleJMX has an optional dependency on the Spring Framework jar(s). This means that if you have the Spring framework included in your project, you can make use of the `BeanPublisher` class which will automagically register the beans from your spring XML configuration files to the `JmxServer` class. The `BeanPublisher` class will register classes that have the `@JmxResource` annotation, that implement the `JmxSelfNaming` interface, that are of type that are of type `JmxBean`, and are of type `PublishAllBeanWrapper`.

Here's a sample bean configuration:

```
<!-- registers @JmxResource and JmxSelfNaming classes with the server -->
<bean id="beanPublisher" class="com.j256.simplejmx.spring.BeanPublisher">
   <property name="jmxServer" ref="jmxServer" />
</bean>

<!-- our JMX server which publishes our JMX beans -->
<bean id="jmxServer" class="com.j256.simplejmx.server.JmxServer"
     init-method="start" destroy-method="stop">
   <property name="registryPort" value="8000" />
</bean>
```

Notice that the `JmxServer` is configured with an `init-method` and `destroy-method` which cause the JMX server to be started and stopped when the application context is loaded and closed.

You can also wire a JMX bean for another bean that is not using the `@JmxResource` annotation and does not implement the `JmxSelfNaming` interface with the help of the `JmxBean` class:

```
<!-- some random bean defined in your spring files -->
<bean id="someBean" class="your.domain.SomeBean">
   ...
</bean>

<!-- publish information about that bean via JMX -->
<bean id="jmxServerJmx" class="com.j256.simplejmx.spring.JmxBean">
   <!-- helps build the ObjectName -->
   <property name="jmxResourceInfo">
      <bean class="com.j256.simplejmx.common.JmxResourceInfo">
         <property name="jmxDomainName" value="your.domain" />
         <property name="jmxBeanName" value="SomeBean" />
      </bean>
   </property>
   <!-- defines the fields that are exposed for JMX -->
   <property name="attributeFieldInfos">
      <array>
         <bean class="com.j256.simplejmx.common.JmxAttributeFieldInfo">
            <property name="name" value="someCounter" />
         </bean>
      </array>
   </property>
   <!-- defines the get/is/set methods exposed -->
   <property name="attributeMethodInfos">
      <array>
         <bean class="com.j256.simplejmx.common.JmxAttributeMethodInfo">
            <property name="methodName" value="getSomeValue" />
         </bean>
      </array>
   </property>
   <!-- defines the operations (i.e. non get/is/set) methods exposed -->
   <property name="operationInfos">
      <array>
         <bean class="com.j256.simplejmx.common.JmxOperationInfo">
            <property name="methodName" value="someMethod" />
         </bean>
      </array>
   </property>
   <property name="target" ref="jmxServer" />
</bean>
```

Take a look at the spring example program to see working code that uses this class. See [spring framework example], page 14.

2.9 Exposing Beans Over HTTP

SimpleJMX contains a simple web-server implementation that uses Jetty so that you can access JMX information from a web browser or other web client using the `JmxWebServer` class. To use this class you need to provide a Jetty version in your dependency list or classpath.

```
// first start the jmx server
JmxServer jmxServer = new JmxServer(JMX_PORT);
jmxServer.start();
// register your beans as normal
jmxServer.register(counter);

// create a web server listening on a specific port
JmxWebServer jmxWebServer = new JmxWebServer(WEB_PORT);
jmxWebServer.start();
```

We recommend that you run the JMX server even if you are planning to use the web server most of the time. You still register beans via the `JmxServer` and any beans added to the platform `MBeanServer` will be visible via your browser.

Take a look at the web-server example program to see working code that uses this class. See [web server example], page 14.

2.10 Using the JVM Platform JMX Server

SimpleJMX allows you to start your own JMX server on a particular port. One of the benefits of using the `JmxServer` server instead of the JVM platform server is that you can specify both the registry and server ports. This allows you to better control what ports need to be exposed through a firewall.

Sometimes, however, you will instead want to use the JMX `MBeanServer` that is built into JVM platform and that is enabled by command line parameters to the `java` command. To enable JMX on the command-line you need to use the following system properties:

`com.sun.management.jmxremote` – turns on JMX

`com.sun.management.jmxremote.port` – sets the registry port

`com.sun.management.jmxremote.authenticate` – enables or disables authentication

`com.sun.management.jmxremote.ssl` – enables or disables SSL connections

For example:

```
java -Dcom.sun.management.jmxremote \
-Dcom.sun.management.jmxremote.port=9999 \
-Dcom.sun.management.jmxremote.authenticate=false \
-Dcom.sun.management.jmxremote.ssl=false \
-jar yourApplication.jar
```

WARNING: You should disable authentication and SSL only if necessary. If you are trying to use JMX over an insecure network then this *will* create a security hole.

JMX is actually a two connection protocol. Your JMX client connects first to the RMI registry port which returns what IP/port to call to connect to the application for

JMX. Depending on the complexity of your network configuration, it may be that the JVM attaches the JMX server to an IP that is not one that the client can contact. In this case, you may need to use the `java.rmi.server.hostname` property to define what name/IP the RMI registry should report back.

```
java -Dcom.sun.management.jmxremote \
-Dcom.sun.management.jmxremote.port=9999 \
...
-Djava.rmi.server.hostname=10.1.2.3 \
...
```

To run the `JmxServer` but still use the `MBeanServer` that was started by the JVM platform, you use the `boolean` constructor and pass `true` to it:

```
// define a server that makes use of the platform MBeanServer
JmxServer jmxServer = new JmxServer(true);
```

There is also a `jmxServer.setUsePlatformMBeanServer(true)` method to use with Spring injection.

2.11 Using With Maven

To use SimpleJMX with maven, include the following dependency in your '`pom.xml`' file:

```
<dependency>
<groupId>com.j256.simplejmx</groupId>
<artifactId>simplejmx</artifactId>
<version>1.18</version>
</dependency>
```

3 Example Code

Here is some example code to help you get going with SimpleJMX. I often find that code is the best documentation of how to get something working. Please feel free to suggest additional example packages for inclusion here. Source code submissions are welcome as long as you don't get piqued if we don't chose your's.

Simple, basic
: This is a simple application which publishes a single object. See the source code on github.

Random object example
: This is an example showing how to programmatically expose using JMX a random object without SimpleJMX annotations: See the source code on github.

Spring Framework example
: This is an example showing how you can use SimpleJMX with the Spring Framework. See the source code on github. The example spring config file is also available.

Jmx Web Server example
: This is an example showing how you can use SimpleJMX to expose JMX beans over HTTP to web browsers. See the source code on github.

Publish all example
: This is an example that shows how you can use the `PublishAllBeanWrapper` to publish information about a random object. See the source code on github.

4 Open Source License

Index of Concepts

Q

R

S

T

U

W