# Developer Guide

**jWebSocket**

**JMXPlugIn**

**Lisdey Pérez Hernández**

1. **Overview**

The main objective of this project is to provide a management solution using JMX (Java Management Extensions) technology for jWebSocket framework. To this end, is developed the JMXPlugIn module which guarantees remote management of the applications developed using this framework through RMI and HTTP protocols. To achieve it, this module uses the JMX integration provided by Spring framework, specifically modifying the class export mechanism for coupling it to the characteristics of jWebSocket.

1. **Infrastructure, Model, Approach**

The JMXPlugIn module is based on the native technology in Java, JMX. Therefore, this module implements the architecture specified for this technology, which is designed for building highly scalable management solutions. The JMX architecture is divided into three main layers: Instrumentation, Agent and Distributed Services.

<https://jwsdev.org:9443/svn/jWebSocket/thesis/JMXPlugIn_Lisdey/Images/components_packages.png>

In keeping with the JMX architecture, JMXPlugIn has an Instrumentation layer which contains the components responsible for encapsulating the objects to be managed, i.e. contains the called MBeans or manageable objects. Moreover, the module presents an Agent layer that contains all the components for controlling the MBeans belonging to the Instrumentation layer and makes them available for remote access. This is the main layer of the module, which allows creating the JMX infrastructure for the jWebSocket server. Finally, the Distributed Services layer contains the elements that enable remote access to the functionality of the module. In this specific case, the access is via the RMI connector or HTTP adapter. It is also the layer that is responsible for the security in the module. Importantly, this layer also contains the JMX-compatible client applications, which are used to access the module. However, these are not part of the implementation, which are not reflected in JMXPlugIn infrastructure.

**Used Design Patterns**

Among the design patterns used in the JMXPlugIn module can mention the Observer pattern, which allows you to configure and manage the event notification mechanism of the JMX technology. Moreover the Facade pattern was used in order to establish a common interface to allow communication between plugins thus enabling remotely invokes its functionality.

**TokenPlugIn or EventPlugIn**

The JMXPlugIn module was developed using TokenPlugIn since it is a low-level structure, which provides improved performance for applications with few features and whose message exchange infrastructure is easier.

**Communication Between Plugins**

One of the features provided by this module is invoked remotely functionality of the applications that are running on a certain jWebSocket server. To achieve this was necessary to create a common interface to allow internal communication between plugins. Thus JMXPlugIn is able to interact with those plugins that implement this communication mechanism.

1. **Requirement and Prerequisites**

The requirements for the continued development of JMXPlugIn module are:

* Network connection
* Using Spring framework specifically JMX integration present therein.

1. **Modules, Structure**

The main aspects that describe the organizational structure of the module JMXPlugIn is reflected in the following table:

|  |  |
| --- | --- |
| Project Name: | JMXPlugIn |
| Location of the sources in the SVN server: | <https://jwsdev.org:9443/svn/jWebSocket/branches/jWebSocket-1.0/jWebSocketPlugIns/jWebSocketJMXPlugIn> |
| SVN branch: | jWebSocket-1.0 |
| Maven dependencies: | <dependency>  <groupId>org.jwebsocket</groupId>  <artifactId>jWebSocketCommon</artifactId>  <version>1.0</version>  </dependency>  <dependency>  <groupId>org.jwebsocket</groupId>  <artifactId>jWebSocketServerAPI</artifactId>  <version>1.0</version>  </dependency>  <dependency>  <groupId>org.jwebsocket</groupId>  <artifactId>jWebSocketServer</artifactId>  <version>1.0</version>  </dependency>    <dependency>  <groupId>mx4j</groupId>  <artifactId>mx4j-tools</artifactId>  <version>2.1.1</version>  </dependency> |
| JAR module: | jWebSocketJMXPlugIn-1.0.jar |
| Package structure: | NameSpace: org.jwebsocket.plugins.jmx  <https://jwsdev.org:9443/svn/jWebSocket/thesis/JMXPlugIn_Lisdey/Images/source_code_packege.png> |
| org.jwebsocket.plugins.jmx:  Contains the core classes of the module that allows the creation of the JMX infrastructure for jWebSocket framework. | |
| org.jwebsocket.plugins.jmx.configdefinition:  Contains the definition of the configuration file used to integrate the plugins and classes that will be exported via JMX. | |
| org.jwebsocket.plugins.jmx.mbeanspring:  Contains the classes that allow engaging the class export mechanism used in Spring to jWebSocket. | |
| org.jwebsocket. plugins.jmx.util:  It contains the generic libraries with utility functions used in the module. | |

**4.1 Code Structure**

The structure of the developed module source code matches the picture shown below:

<https://jwsdev.org:9443/svn/jWebSocket/thesis/JMXPlugIn_Lisdey/Images/folders.png>

**Elements Description:**

src: This directory contains all the source code of classes and libraries of the solution.

target: This directory temporarily stores the compiled source code, its content is not included in the version control.

nbactions.xml: Project configuration file created by the NetBeans IDE. The content of this file is not included in the version control.

pom.xml: Configuration file that contains information about the project and details of the settings used by Maven to build the project.

**4.2 Package Description**

The JMXPlugIn module is structured in 4 packages. Among these is the org.jwebsokcket.plugins.jmx package which contains the following classes:

|  |  |
| --- | --- |
| **Class** | **Description** |
| JMXPlugIn | Main class of the module which is responsible for creating the JMX infrastructure to use and initialize all other components within the module. |
| JMXPlugInFunctions | Class that allows invokes certain features of the plugins that are running on a given server. It also provides information about the plugins that are loaded and functionality that can be invoked. |
| JMXServerFunctions | Class that allows manages certain functions of jWebSocket server. |
| JMXPlugInsExporter | Main class of the mechanism for integrating plugins and classes to the JMX infrastructure of the module. Is responsible for reading all the configuration files created for this purpose and register the object so they can be remotely exported. |

The org.jwebsokcket.plugins.jmx.configdefinition package contains the following classes:

|  |  |
| --- | --- |
| **Class** | **Description** |
| AttributeDefinition | Class that allows defines the attributes of the plugins or classes to export and their metadata. |
| ConstructorDefinition | Class that allows defines the constructors of the classes to export and their metadata |
| ConstuctorParameterDefinition | Class that allows defines the input parameters of the constructors of the classes to export and their metadata |
| FeatureDefinition | Generic class that allows defines the name and description of all elements of the plugins or classes to export. |
| JMXDefinition | Main class that contains all the elements necessary to form the class or plugin object to export and its metadata. |
| JMXDefinitionException | Class that allows defines an exception associated with a plugin or class that will be exported. Thus if an exception is thrown when trying to create a plugin or class will be show an object of this type. |
| JMXPluginDefinition | Class that contains the specific elements to create the plugin object to export. |
| NotificationDefinition | Class that allows defines the event notifications of the plugins and classes to export and their metadata. For the module have been defined notifications for the following events: before and after performing an operation and when you change the value of an attribute. |
| OperationDefinition | Class that allows defines the operations of the plugins and classes to export and their metadata. |
| ParameterDefinition | Class that allows defines the input parameters of the operations of the plugins and classes to export and their metadata. |

The org.jwebsokcket.plugins.jmx.mbeanspring package contains the following clases:

|  |  |
| --- | --- |
| **Class** | **Description** |
| MBeanEnabledExporter | Class that redefine certain functionalities of the Spring MBeanExporter class, taken into account the characteristics of the JMXPlugIn module. |
| ModelMBeanExtension | Class that implements a ModelMBean class specific to the module to develop. |
| ModelMBeanUtil | Class that contains the auxiliary methods for the ModelMBean creation. |
| NotificationInfoMap | Class that define an event notification map that will be created for a certain ModelMBean. |

The org.jwebsokcket.plugins.jmx.util package contains the following classes:

|  |  |
| --- | --- |
| **Class** | **Description** |
| JMXHandler | Class that dynamically converts the data type Map, largely driven by jWebSocket framework, to the CompositeData data type, which is used by the JMX technology for working with complex data types like objects, since for default this technology is only able to handle simple data types. |
| JMXPlugInAuthenticator | Class that implements the security mechanism for remote access to the module via the RMI protocol. |

1. **Source Code**

**5.1 Common Code Standards**

The common code standards used to develop the JMXPlugIn module are the same defined to jWebSocket framework.

**5.2 Reusable Components**

The module architecture is divided into separate components which ensure its reuse in applications with similar characteristics. The components belonging to this module which can be reused are:

* **JMXHandler:** Class that dynamically converts the data type Map, largely driven by jWebSocket framework, to the CompositeData data type, which is used by the JMX technology for working with complex data types like objects, since for default this technology is only able to handle simple data types.
* **JMXPlugInExporter:** allow you to modify the default behavior of the JMX technology support provided by Spring. This component allows the jWebSocket framework has a greater control and personalization to display information about the classes or plugins to be exported using JMX technology. This avoids the use of interfaces or Java annotations, Spring default mechanisms that limit the work with exported class.

1. **Interfaces (if applicable)**

To access the JMXPlugIn module is possible to use JMX-compatible client applications. However, so far the implementations of these applications are not part of the scope of this project.

1. **Frameworks, Libraries and Tools**

To develop the module was used JMXPlugIn Spring framework which has the Apache License 2.0[[1]](#footnote-1) license. This framework has integration with JMX technology which is used mostly for developing JMXPlugIn. Given that currently jWebSocket integrates the IoC Container of Spring as well as Authentication and Validation module, is not needed any additional configuration. For more information on the Spring framework visit the following link: <http://www.springsource.org/>.

Moreover, JMXPlugIn module use MX4J library which holds the Apache-style[[2]](#footnote-2) license. This library is used for the remote access via HTTP protocol. For more information about the MX4J library visit the following link: <http://mx4j.sourceforge.net/>.

1. **Database and Persistence (if applicable)**

JMXPlugIn module does not require the use of a database for its operation.

1. **Hardware (if applicable)**

JMXPlugIn module does not require the use of additional hardware for its operation.

1. **Security**

In order to ensure the safety of JMXPlugIn module was implemented an authentication component which uses MD5 encryption algorithm and in turn establishes communication over SSL for each of the protocols used for remote access, i.e. HTTP and RMI protocols.

1. **Tests, Quality Assurance and Continuous Integration**

*This section describes the execution of automated and manual tests for quality insurance. Right now we do not yet have continuous integration, so this is still pending.*

* *Test cases, functional (JUnit, Jasmine) and UI tests (iMacros, pending).  
  Which automated tests are available, how are they invoked, how are the test cases integrated into our deployment test suite? Where are the reports?*
* *Are there known issues (traps, also refer to bug-list), what are the reasons, is this approved and what can be done in case of these issues (work-arounds)?*
* *What about the error and potential issue handling (e.g. database not available, network connection broken other resources do not exist).*
* *What about the general error and exception handling?*
* *If required (because not (yet) covered or not coverable), which manual tests can/have to be run before deployment?*

1. **Profiling**

*This section describes the methods and strategies for continous improvements.*

* *"Improvements" means provisions for higher speed, less memory consumption, simpler maintainability or similar, which do not affect the functionality in general (e.g. "the solutions uses a quicksort instead of a bubblesort", or "in future the solution should use MongoDB instead of MySQL because..."). It does not mean bug-fixing.*

1. **Reference**

*This is a structured lookup document; here it is about a complete reference of details, just with keywords or short sentences rather than much text,*

* *Complete JavaDocs or JavaScriptDocs (inline documentation) are mandatory for publishing (classes, constants, variables, methods, arguments, result, examples, properties, flags), all classes and methods in all languages should be fully documented for efficient knowledge management and maintainability.*
* *The JavaDocs and JavaScript Docs as well as other automatically generated documention which is targetted for online-distribution (on our website), does not need to be printed in the Developer Guide, here a reference to the online area is sufficient.*
* *A "Token Reference" is mandatory for the Developer Guide (see e.g. the Channel Documentation on our WebSite)*
* *List of files, as far as not automatically generated by any inline documentation tool, like JavaDocs. (e.g. a certain set of configuration files and their purpose), for configuration files, list of settings and possible options. This is also important for the packaging and deployment of the solution.*

1. <http://www.apache.org/licenses/LICENSE-2.0.html> [↑](#footnote-ref-1)
2. <http://mx4j.sourceforge.net/docs/ch01s06.html> [↑](#footnote-ref-2)