ABEILLE Gwenaëlle



Guide for developers of Tango devices

Tango Device in Java

Table of contents

[1. Introduction 3](#_Toc411952102)

[2. A first device 4](#_Toc411952103)

[3. Device 6](#_Toc411952104)

[4. Command 7](#_Toc411952105)

[5. Attribute 8](#_Toc411952106)

[6. Pipe 11](#_Toc411952107)

[7. Init 11](#_Toc411952108)

[8. Delete 12](#_Toc411952109)

[9. State 13](#_Toc411952110)

[10. Status 13](#_Toc411952111)

[11. Device property 14](#_Toc411952112)

[12. Device properties 14](#_Toc411952113)

[13. Class property 14](#_Toc411952114)

[14. Around Invoke 15](#_Toc411952115)

[15. State machine 16](#_Toc411952116)

[16. Device Manager 16](#_Toc411952117)

[17. Dynamic API 17](#_Toc411952118)

[a. Dynamic Command 17](#_Toc411952119)

[i. Configuration 17](#_Toc411952120)

[ii. StateMachine 18](#_Toc411952121)

[iii. Execution 18](#_Toc411952122)

[b. Dynamic Attribute 18](#_Toc411952123)

[i. Configuration 18](#_Toc411952124)

[ii. StateMachine 19](#_Toc411952125)

[iii. Read attribute 19](#_Toc411952126)

[iv. Write attribute 19](#_Toc411952127)

[v. Update write part 20](#_Toc411952128)

[vi. Forwarded Attribute 20](#_Toc411952129)

[18. Default dynamic attributes and commands 20](#_Toc411952130)

[19. Events 20](#_Toc411952131)

[a. Polled events 22](#_Toc411952132)

[b. Pushed events 22](#_Toc411952133)

[20. Error management 24](#_Toc411952134)

[21. Logging 24](#_Toc411952135)

[22. Start up: DB/NO DB 25](#_Toc411952136)

[a. Server with Tango Database 25](#_Toc411952137)

[b. Device without Tango database 26](#_Toc411952138)

[23. Annexes 27](#_Toc411952139)

[a. Full sample device code 27](#_Toc411952140)

[b. Command with ICommandBehavior 28](#_Toc411952141)

[c. Attribute with IAttributeBehavior 28](#_Toc411952142)

[d. Extended example 30](#_Toc411952143)

[e. Logging configuration with logback 33](#_Toc411952144)

[f. Properties file for a device without Tango Database 33](#_Toc411952145)

# Introduction

This paper is a documentation intended for developers. It described how to build a Java Tango Device. A background in the Java language is strongly recommended. The pre-requisites are:

* The Tango concepts: attribute, command, device property… Please read the Tango reference manual : <http://www.tango-controls.org/>, <http://www.tango-controls.org/Documents/tango-kernel/>
* The Java language Standard Edition : <http://www.oracle.com/technetwork/java/javase/documentation/index.html>
* The concept of annotations introduced in Java version 5: <http://docs.oracle.com/javase/tutorial/java/javaOO/annotations.html>
* Java beans : <http://en.wikipedia.org/wiki/JavaBeans>

# A first device

Here is the code of a simple device class with one Tango command and one attribute (see annexes for full code):

@Device

**public** **class** TestDevice {

**private** **final** Logger logger = LoggerFactory.*getLogger*(TestDevice.**class**);

/\*\*

\* Attribute myAttribute READ WRITE, type DevDouble.

\*/

@Attribute

**public** **double** myAttribute;

/\*\*

\* Starts the server.

\*/

**public** **static** **void** main(**final** String[] args) {

ServerManager.*getInstance*().start(args, TestDevice.**class**);

}

/\*\*

\* init device

\*/

@Init

**public** **void** init() {

logger.debug("init");

}

/\*\*

\* delete device

\*/

@Delete

**public** **void** delete() {

logger.debug("delete");

}

/\*\*

\* Execute command start. Type VOID-VOID

\*/

@Command

**public** **void** start() {

logger.debug("start");

}

/\*\*

\* Read attribute myAttribute.

\*

\* **@return**

\*/

**public** **double** getMyAttribute() {

logger.debug("getMyAttribute {}", myAttribute);

**return** myAttribute;

}

/\*\*

\* Write attribute myAttribute

\*

\* **@param** myAttribute

\*/

**public** **void** setMyAttribute(**final** **double** myAttribute) {

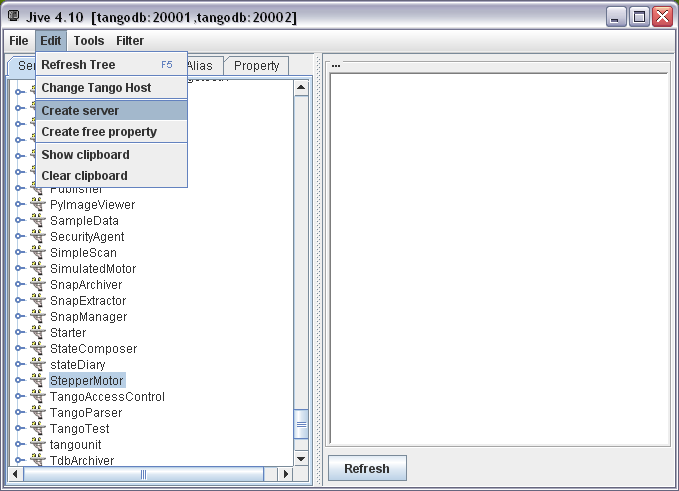
logger.debug("setMyAttribute {}", myAttribute);

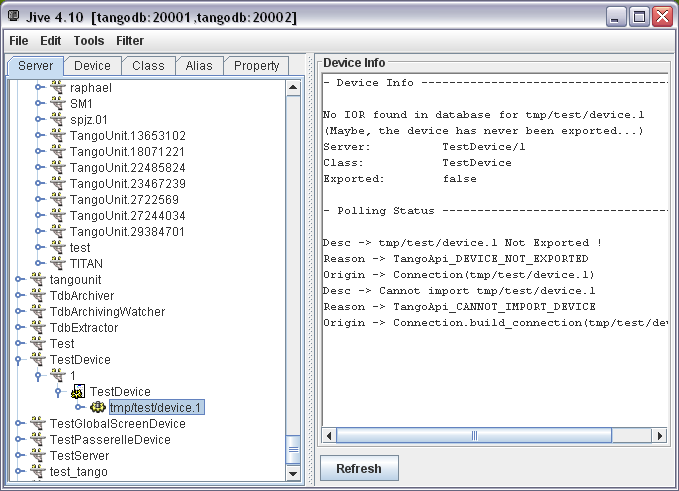
**this**.myAttribute = myAttribute;

}

}

Before starting this device, it has to be declared in the Tango database with Jive menu “Create server”. Hereafter, a server “TestDevice/1” with one device “tmp/test/device.1” is created:

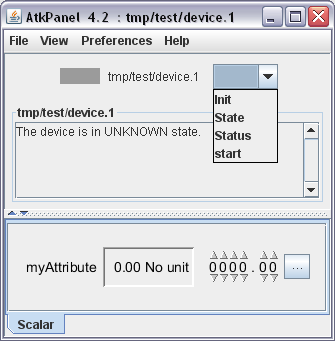




As the TestDevice class of this device has a main method, it can be started as a standard Java program:

1. A Java system property “TANGO\_HOST” must be defined. For instance “tangodb:20001,tangodb:20002” , like in the Jive screenshot above.
2. The mandatory program argument is the instance name (1 in above example).

Once started, the device can be tested. Here is an example of the Tango generic client ATKPanel:



NB: In Tango, the commands *Init*, *State*, *Status* and the attributes *State*, *Status* are created by default for any device.

Here is a first code explanation:

* The “@Device” annotation on a class defines this class as a Tango Device.
* The “@Attribute” annotation defines a field as a Tango attribute:
  + The attribute type is defined by the field type;
  + If this field has a getter, it is a READ attribute;
  + If it has a setter, it is a WRITE attribute;
  + If it has both getter and setter, it is a READ/WRITE attribute.
* The annotation “@Command” defines a method as a Tango command:
  + The parameter type defines the input type
  + The return type defines the output type
* The “@Init” annotation defines a method called:
  + At server startup;
  + When “Init” command is called.
* The “@Delete” annotation defines a method called:
  + At server shutdown;
  + At “Init” command, just before “@Init”.
* The main method starts the server
* The logger field is to log.

The following chapters will describes all this in details.

# Device

A Tango device class must have the following Java annotation:

org.tango.server.annotation.Device

@Device

**public** **class** TestDevice {

}

This class can only have a no-arguments constructor.

This annotation has an option to configure how the server will manage client transactions. Default value is “NONE”. Here is an example for one client request at a time per device:

@Device(transactionType = TransactionType.*DEVICE*)

All transaction values are:

* TransactionType.*DEVICE*: One client request per device.
* TransactionType.*CLASS*: One client request per device class (that may contain several devices).
* TransactionType.*SERVER*: One client request per server (that may contain several classes).
* TransactionType.*ATTRIBUTE*: One client request per attribute.
* TransactionType.*COMMAND*: One client request per command.
* TransactionType.*ATTRIBUTE\_COMMAND*: One client request per attribute or command.
* TransactionType.*NONE*: Default value. All client requests can be done at the same time.

NB: A good choice has to be made between performance and thread-safety of the device depending of the use-cases:

* Using TransactionType.NONE means that several clients can modify values, states in the device at the same time. In this case, the developer has to implement the thread-safety by himself if necessary. A good use case for this configuration is a “stateless” device where each request is an independent transaction that is unrelated to any previous request.
* Using TransactionType.DEVICE means that only one client can do a request on the device at a time. So, if a lot of clients are connected to the device, their performance can be drastically reduced while waiting for other clients. The main use case for this configuration is a “statefull” device that contains a conversation state that is retained across transactions.

# Command

A tango command is created with this Java annotation on a method:

org.tango.server.annotation.Command

Example code of a command with a parameter of type DEVVARDOUBLEARRAY and a returned type of DEVLONG:

@Command

**public** **int** testCmd(**final** **double**[] in) {

**return** 0;

}

The command name is by default the method name. The Command annotation has some parameters to change its name, its description, its polling configuration… See javadoc for details.

The method has to be public.

The input and output types are defined by the method definition. Here are the Tango types for each Java type:

|  |  |
| --- | --- |
| Java type | Tango type |
| void | DEVVOID |
| boolean | DEVBOOLEAN |
| long | DEVLONG64 |
| long[] | DEVVARLONG64ARRAY |
| short | DEVSHORT |
| short[] | DEVVARSHORTARRAY |
| float | DEVFLOAT |
| float[] | DEVVARFLOATARRAY |
| double | DEVDOUBLE |
| double[] | DEVVARDOUBLEARRAY |
| String | DEVSTRING |
| String[] | DEVVARSTRINGARRAY |
| int | DEVLONG |
| Int[] | DEVVARLONGARRAY |
| DevState or DeviceState | DEVSTATE |
| byte | DEVUCHAR |
| byte[] | DEVVARCHARARRAY |
| DevEncoded | DEVENCONDED |
| DevVarLongStringArray | DEVVARLONGSTRINGARRAY |
| DevVarDoubleStringArray | DEVVARDOUBLESTRINGARRAY |

NB: Full class names of Tango commands:

fr.esrf.Tango.DevState

fr.esrf.Tango.DevEncoded

fr.esrf.Tango.DevVarLongStringArray

fr.esrf.Tango.DevVarDoubleStringArray

org.tango.DeviceState

Tango provides also other types that do not have equivalent in Java types: DEVULONG, DEVULONG64, DEVUSHORT, DEVVARULONGARRAY, DEVVARULONG64ARRAY, DEVVARUSHORTARRAY. It is possible to define these types with a dynamic command (Cf chapter dynamic API for details).

NB: The wrappers objects of primitives (Integer, Double…) can also be used, but it could lead to performance issues.

# Attribute

A tango attribute is created with this Java annotation on a method or a field:

org.tango.server.annotation.Attribute

Example code of a DEVDOUBLE scalar read and write attribute:

@Attribute

**private** **double** testAttribute;

**public** **double** getTestAttribute() {

**return** testAttribute;

}

**public** **void** setTestAttribute(**double** testAttribute) {

**this**.testAttribute = testAttribute;

}

Example code for DEVENUM attribute:

**public** **enum** TestType {

*VALUE1*, *VALUE2*

}

@Attribute

**private** TestType enumAttribute = TestType.*VALUE1*;

**public** TestType getEnumAttribute() {

**return** enumAttribute;

}

**public** **void** setEnumAttribute(**final** TestType enumAttribute) {

**this**.enumAttribute = enumAttribute;

}

As defined by the Java bean convention, the setter and getter must contain the name of the field and manage the same type as the field (reminder: a getter for a boolean starts by “is”). The getter and setter have to be public while the field is private.

* If this field has a getter, it is a READ attribute;
* If it has a setter, it is a WRITE attribute;
* If it has both, it is a READ/WRITE attribute.

It is also possible to place the annotation on the getter method.

The attribute name is by default the field name. The annotation has some parameters to change its name, its polling configuration, its memorization configuration… See javadoc for details.

Here are the Tango types for each Java type:

|  |  |  |
| --- | --- | --- |
| Java type | Tango type | Tango format |
| Boolean | DEVBOOLEAN | SCALAR |
| boolean[] | DEVBOOLEAN | SPECTRUM |
| boolean[][] | DEVBOOLEAN | IMAGE |
| Long | DEVLONG64 | SCALAR |
| long[] | DEVLONG64 | SPECTRUM |
| long[][] | DEVLONG64 | IMAGE |
| Short | DEVSHORT | SCALAR |
| short[] | DEVSHORT | SPECTRUM |
| short[][] | DEVSHORT | IMAGE |
| Float | DEVFLOAT | SCALAR |
| float[] | DEVFLOAT | SPECTRUM |
| float[][] | DEVFLOAT | IMAGE |
| Double | DEVDOUBLE | SCALAR |
| double[] | DEVDOUBLE | SPECTRUM |
| double[][] | DEVDOUBLE | IMAGE |
| String | DEVSTRING | SCALAR |
| String[] | DEVSTRING | SPECTRUM |
| String[][] | DEVSTRING | IMAGE |
| Int | DEVLONG | SCALAR |
| int[] | DEVLONG | SPECTRUM |
| int[][] | DEVLONG | IMAGE |
| DevState or DeviceState | DEVSTATE | SCALAR |
| DevState[] or DeviceState[] | DEVSTATE | SPECTRUM |
| DevState[][] or DeviceState[][] | DEVSTATE | IMAGE |
| Byte | DEVUCHAR | SCALAR |
| byte[] | DEVUCHAR | SPECTRUM |
| byte[][] | DEVUCHAR | IMAGE |
| DevEncoded | DEVENCONDED | SCALAR |
| Enum | DEVENUM | SCALAR |

NB: Full class names of tango attributes:

fr.esrf.Tango.DevState

fr.esrf.Tango.DevEncoded

fr.esrf.Tango.DevVarLongStringArray

fr.esrf.Tango.DevVarDoubleStringArray

org.tango.DeviceState

Tango provides also other types that do not have equivalent in Java types: DEVULONG, DEVULONG64, and DEVUSHORT, DEVENUM (with enumerated value configurable from its attribute property). It is possible to define these types with a dynamic attribute (Cf chapter dynamic API for details). Please also refer to this section if the write part of the attribute has to be changed from the device.

NB: The wrappers objects of primitives (Integer, Double…) can also be used, but it could lead to performance issues.

A Tango attribute has also a quality and a timestamp. The default behavior is a valid quality, and the timestamp is the read time. To access these properties, the getter method can return a container for the attribute value, quality and timestamp. The container is: org.tango.server.attribute.AttributeValue. It contains constructors and methods to set the value, quality and timestamp. Please refer to its javadoc for details.

@Attribute

**private** **double** myAttribute;

**public** AttributeValue getMyAttribute() **throws** DevFailed {

AttributeValue value = **new** AttributeValue(myAttribute);

value.setQuality(AttrQuality.*ATTR\_CHANGING*);

value.setTime(System.*currentTimeMillis*());

**return** value;

}

The default attribute properties are configurable with this annotation:

org.tango.server.annotation.AttributeProperties

Please refer to javadoc for details. Example:

@Attribute

@AttributeProperties(format = "%6.4f", description = "a test attribute")

**private** **double** testAttribute;

# Pipe

A tango pipe is created with this Java annotation on a method or a field:

org.tango.server.annotation.Pipe

Example code of a read pipe:

@Pipe

**private** PipeValue myPipeRO;

…

**final** PipeBlob myPipeBlob = **new** PipeBlob("A");

myPipeBlob.add(**new** PipeDataElement("C", "B"));

myPipeRO = **new** PipeValue(myPipeBlob);

…

**public** PipeValue getMyPipeRO() {**return** myPipeRO;}

# Init

org.tango.server.annotation.Init

@Init

**public** **void** init() {

}

This method must be public with no parameters. It is called:

* At server startup
* And when “Init” command is called.

If this method throws an exception, the device will automatically switch to the “FAULT” state and the status will provide the stack trace.

This annotation has a boolean option called “lazyLoading”. Its default value is false. If the init method takes a lot a time, its execution can be detached with this option set to true. The device will automatically switch in state “INIT” during its execution. This option avoids timeouts when executing the “Init” command as well as a rapid device startup and consequently a rapid control system startup.

# Delete

org.tango.server.annotation.Delete

@Delete

**public** **void** delete() {

}

Method must be public with no parameters. It is called:

* When “Init” command is called before @Init method
* At server shutdown.

The delete method is generally used to close resources.

# State

org.tango.server.annotation.State

@State

**private** DeviceState state;

**public** DeviceState getState() {

**return** state;

}

**public** **void** setState(**final** DeviceState state) {

**this**.state = state;

}

The state annotation defines the state of the device, which will appear in the default command and attribute “State”. The field can be fr.esrf.Tango.DevState or org.tango.DeviceState:

* DevState is the Tango standard type defined by the IDL.
* DeviceState is java Enum that provides easiness to manage a State.

Getter and setter are mandatory.

The device property “StateCheckAttrAlarm” is defined for all Java devices. If set to true, each times a client request the state or the status of the device, all attributes are read to check if some attributes are in ALARM or WARNING quality. If alarms are detected, the state and the status will be updated consequently. The default value of this property is false. WARNING: if some attributes requests are slow, it could lead to performance issues.

# Status

org.tango.server.annotation.Status

@Status

**private** String status;

**public** String getStatus() {

**return** status;

}

**public** **void** setStatus(String status) {

**this**.status = status;

}

The status annotation defines the status of the device, which will appear in the default command and attribute “Status”. The status field must be a String, getter and setter are mandatory.

# Device property

org.tango.server.annotation.DeviceProperty

*NB: Tango reminder: loading order of a device property:*

* *Value defined at device level*
* *If does not exists; value defined at class level*
* *If does not exists; default value*

@DeviceProperty (defaultValue = "", description = "an example")

**private** String devicePropTest;

**public** **void** setDevicePropTest(String devicePropTest) {

**this**.devicePropTest = devicePropTest;

}

The field can be of any standard java type (int, double …), as scalar or array.

The property has some parameters, details are in javadoc.

A setter is mandatory, so that the value can be injected at device initialization.

# Device properties

org.tango.server.annotation.DeviceProperties

It is possible to retrieve all device properties at once. It can be useful if some device properties are not known in advance (Example: some dynamic attributes that have their names as a device property name).

@DeviceProperties

**private** Map<String, String[]> devicePropTest;

**public** **void** setDevicePropTest(**final** Map<String, String[]> devicePropTest) {

**this**.devicePropTest = devicePropTest;

}

The field has to be a java.util.Map with a “String” key and a “String[]” value.

A setter is mandatory, so that the value can be injected at device initialization.

# Class property

org.tango.server.annotation.ClassProperty

@ClassProperty

**private** **double**[] classPropTest;

**public** **void** setClassPropTest(**double**[] classPropTest) {

**this**.classPropTest = classPropTest;

}

The field can be of any standard java type (int, double …), as scalar or array.

The property has some parameters, details are in javadoc.

A setter is mandatory, so that the value can be injected at device initialization.

# Around Invoke

org.tango.server.annotation.AroundInvoke

It defines a public void method with a single parameter of class org.tango.server.InvocationContext. It is called before and after every command and attributes execution. This functionality is known as “always executed hook” in C++.

@AroundInvoke

**public** **void** aroundInvoke(**final** InvocationContext ctxt) {

System.*out*.println("called at " + ctxt.getContext());

System.*out*.println("called command or attributes " +

Arrays.*toString*(ctxt.getNames()));

}

# State machine

org.tango.server.annotation.StateMachine

The StateMachine annotation allows to define some denied states, and some state changes:

* For an “@Init”, it is possible to define the state at the end of its execution
* For a command, its execution can be disallowed for some states and the state at the end of its execution can be defined.
* For an attribute, it can be disallowed to write it for some states and the state at the end of its execution.

@Attribute

@StateMachine(endState = DeviceState.*RUNNING*)

**private** **double** value;

@Init

@StateMachine(endState = DeviceState.*OFF*)

**public** **void** init() {

}

@Command

@StateMachine(deniedStates = { DeviceState.*FAULT*, DeviceState.*UNKNOWN* }, endState = DeviceState.*ON*)

**public** **int** on() {

**return** 0;

}

# Device Manager

org.tango.server.annotation.DeviceManagement

DeviceManager contains common utilities for a device. For example, it provides its name, its admin device name, a way to change attribute properties…

@DeviceManagement

**private** DeviceManager deviceManager;

@Init

**public** **void** init() {

System.*out*.println(deviceManager.getName());

}

**public** **void** setDeviceManager(**final** DeviceManager deviceManager) {

**this**.deviceManager = deviceManager;

}

# Dynamic API

Attributes and commands can be created dynamically with the class org.tango.server.dynamic.DynamicManager that will be injected by using the annotation org.tango.server.annotation.DynamicManagement. It provides methods to add or remove attributes and commands. Typically, the add methods will be called in the @Init method and remove will be called in @Delete method:

@DynamicManagement

**private** DynamicManager dynamicManagement;

**public** **void** setDynamicManagement(DynamicManager dynamicManagement) {

**this**.dynamicManagement = dynamicManagement;

}

@Init

**public** **void** init() **throws** DevFailed {

dynamicManager.addAttribute(**new** TestDynamicAttribute());

dynamicManager.addCommand(**new** TestDynamicCommand());

}

@Delete

**public** **void** delete() **throws** DevFailed {

dynamicManager.clearAll();

}

NB: If a server is running with several devices in the same process, the dynamic commands or attributes can be different for each device.

The following paragraphs explain in details how to create attribute and commands.

## Dynamic Command

A dynamic command is a class that must implement the interface:

org.tango.server.command.ICommandBehavior

See annexes for a full sample code.

### Configuration

The method getConfiguration is used to define a command configuration like its name, its type… see javadoc of org.tango.server.command.CommandConfiguration for details). Here is an example a command called testDynCmd with no parameter and a returned value of type DEVDOUBLE:

**public** CommandConfiguration getConfiguration() **throws** DevFailed {

**final** CommandConfiguration config = **new** CommandConfiguration();

config.setName("testDynCmd");

config.setInType(**void**.**class**);

config.setOutType(**double**.**class**);

**return** config;

}

The command types may be declared in two different ways:

* setInType(Class<?> type) or setOutType: as table in chapter “Command”, the java class defines the command type.
* setTangoInType(int tangoType)or setTangoOutType: defines the type with an integer (constants are defined in class fr.esrf.TangoConst). This method is more flexible as some Tango types do not have equivalent in Java classes: DEVULONG, DEVULONG64, DEVUSHORT, DEVVARULONGARRAY, DEVVARULONG64ARRAY, and DEVVARUSHORTARRAY.

### StateMachine

It is optional and can return “null”. It works like the StateMachine annotation. See its chapter for details.

**public** StateMachineBehavior getStateMachine() **throws** DevFailed {

**final** StateMachineBehavior stateMachine = **new** StateMachineBehavior();

stateMachine.setDeniedStates(DeviceState.*FAULT*);

stateMachine.setEndState(DeviceState.*ON*);

**return** stateMachine;

}

### Execution

The input and output types of the execute method is defined by the configuration above. If the type is void, the parameter or returned value may be null.

**public** Object execute(**final** Object arg) **throws** DevFailed {

**return** 10.0;

}

## Dynamic Attribute

A dynamic attribute is a class that must implement:

org.tango.server.attribute.IAttributeBehavior

See annexes for a full sample code.

### Configuration

The method “getConfiguration” returns the full configuration of the attribute (see javadoc of org.tango.server.attribute.AttributeConfiguration for details). Here is an example for a scalar, DevDouble, READ\_WRITE attribute:

**public** AttributeConfiguration getConfiguration() **throws** DevFailed {

**final** AttributeConfiguration config = **new** AttributeConfiguration();

config.setName("testDynAttr");

// attribute testDynAttr is a DevDouble

config.setType(**double**.**class**);

// attribute testDynAttr is READ\_WRITE

config.setWritable(AttrWriteType.*READ\_WRITE*);

**return** config;

}

The attribute type and format may be declared in two different ways:

* setType(Class<?> type): as table in chapter “Attribute”, the java class defines the attribute type and format.
* setTangoType(int tangoType, AttrDataFormat format): defines the type with an integer (constants are defined in class fr.esrf.TangoConst). The format is defined by the class fr.esrf.AttrDataFormat. This method is more flexible as some Tango types do not have equivalent in Java classes: DEVULONG, DEVULONG64, DEVUSHORT, DEVENUM. Example of DEVENUM:

**final** AttributePropertiesImpl props = **new** AttributePropertiesImpl();

props.setLabel("DevEnumDynamic");

props.setEnumLabels(**new** String[] { "label1", "label2" });

configAttr.setTangoType(TangoConst.*Tango\_DEV\_ENUM*, AttrDataFormat.*SCALAR*);

### StateMachine

Not mandatory, can return “null”. It works like the StateMachine annotation. See its chapter for details.

**public** StateMachineBehavior getStateMachine() **throws** DevFailed {

**final** StateMachineBehavior stateMachine = **new** StateMachineBehavior();

stateMachine.setDeniedStates(DeviceState.*FAULT*);

stateMachine.setEndState(DeviceState.*ON*);

**return** stateMachine;

}

### Read attribute

The “getValue” method is used to read the attribute. It must return an org.tango.server.attribute.AttributeValue (see javadoc for details). Of course, the inserted value must be of the same type as the attribute type (defined in “getConfiguration”).

**private** **double** readValue = 0;

**private** **double** writeValue = 0;

**public** AttributeValue getValue() **throws** DevFailed {

readValue = readValue + writeValue;

**return** **new** AttributeValue(readValue);

}

### Write attribute

The method “setValue” will be called only if the attribute has been defined as writable in “getConfiguration”.

**public** **void** setValue(**final** AttributeValue value) **throws** DevFailed {

writeValue = (Double) value.getValue();

}

### Update write part

In some specific cases, the write part has to be updated from the device (i.e. the last set point of an equipment). This is possible by implementing the interface org.tango.server.attribute.ISetValueUpdater which has one method:

**public** AttributeValue getSetValue() **throws** DevFailed {

**return** **new** AttributeValue(writeValue);

}

### Forwarded Attribute

To create a forwarded attribute, just use org.tango.server.attribute.ForwardedAttribute:

@DynamicManagement

**private** DynamicManager dynamicManagement;

@Init

**public** **void** init() **throws** DevFailed {

dynamicManager.addAttribute(**new** ForwardedAttribute(*fullRootAttributeName*, *attributeName*, defaulltLabel);

}

# Default dynamic attributes and commands

Some default dynamic attributes and commands are already in the library JTangoServerLang, i.e.:

* Attribute and command proxies
* Group command
* Log attribute to send logs to an attribute

Example: org.tango.server.dynamic.command.ProxyCommand will create a Command that is connected to another command. The input and output types will be calculated automatically.

# Events

The detailed concepts of events are described in the Tango kernel documentation. This section is just a reminder of the key concepts and how to apply it in Java.

An event is send from a device’s attribute to the clients that have subscribed to it. There are six different types of events:

* *CHANGE\_EVENT*: Sends an event according to the criteria defined in the attribute properties “abs\_change” and/or “rel\_change”. Sends also an event if the attribute‘s quality changes.
* *PERIODIC\_EVENT*: Sends an event at the period specified by the attribute property “event\_period”
* *ARCHIVE\_EVENT*: Archived event. Can either:
  + Sends a periodic event at period configured in the property “archive\_period”.
  + Or/and change event with values from “archive\_rel\_change” and/or “archive\_abs\_change”
* *USER\_EVENT*: The developer of the device can choose when to send this event.
* *ATT\_CONF\_EVENT*: Attribute configuration event. Sends an event if an attribute's properties change.
* *DATA\_READY\_EVENT*: The developer of the device can choose when to send the event. It is used to notify the client that some data is ready.
* *INTERFACE\_CHANGE:* Each time the lists of commands or attributes change, an event is fire.

There are two ways to send events from a server to clients:

* Polled events: the cache mechanism will take care of sending events.
* Pushed events: the events will be sent directly for the device’s code.

### Polled events

To send a polled event, the polling has to be configured. Only *CHANGE\_EVENT, PERIODIC\_EVENT* and *ARCHIVE\_EVENT* can be send by the polling mechanism. Some default values can be set directly in the device’s code. In the following example, the attribute ‘doubleAtt’ is polled at a 100 milliseconds rate and will send a change event if its value varies at least of 1 since the last time it was sent:

@Attribute(isPolled = **true**, pollingPeriod = 100)

@AttributeProperties(changeEventAbsolute = "1")

**private** **double** doubleAtt = **0**;

### Pushed events

The event types that can be sent from the device’s code are *CHANGE\_EVENT, ARCHIVING\_EVENT, DATA\_READY\_EVENT and USER\_EVENT*. For the *CHANGE* and *ARCHIVING* events types, it is possible to activate the check of the attribute properties criteria before firing it. In this case, it is done by the API before sending the event.

In the following example, a change event is pushed on the attribute ‘doubleAttr’. The API will check if the event must be send according to the criteria ‘changeEventAbsolute’ and ‘changeEventRelative’:

@DeviceManagement

DeviceManager deviceManager;

**public** **void** setDeviceManager(**final** DeviceManager deviceManager) {

**this**.deviceManager = deviceManager;

}

@Attribute(pushChangeEvent = **true**, checkChangeEvent = **true**)

@AttributeProperties(changeEventAbsolute = "1", changeEventRelative = "0.3")

**private** **double** doubleAttr;

…

doubleAttr++;

deviceManager.pushEvent("doubleAttr", **new** AttributeValue(doubleAttr), EventType.*CHANGE\_EVENT*);

…

Here is an example for pushing data ready events:

**private** **int** counter;

@Attribute(pushDataReady = **true**)

**private** **double** doubleAttr;

**…**

counter++;

…

deviceManager.pushDataReadyEvent("doubleAttr", counter);

…

Here is an example that sends a user event:

@Attribute

**public** String getUserEvent() **throws** DevFailed {

**return** "Hello";

}

**…**

deviceManager.pushEvent("userEvent",**new** AttributeValue("test"), EventType.*USER\_EVENT*);

# Error management

The standard exception in Tango is fr.esrf.DevFailed. The class org.tango.DevFailedUtils is useful to throw it. It will, for instance, fill the origin field. See javadoc for details.

@Command

**public** **int** off() **throws** DevFailed {

**throw** DevFailedUtils.*newDevFailed*("DEVICE\_ERROR", "an example error");

}

# **Loggin**g

The Java Tango server API uses SLF4J (<http://www.slf4j.org/>). The underlying libraries use also SLF4J (i.e. jacorb, ehcache…). Here is a declaration example of a logger class:

**import** org.slf4j.Logger;

**import** org.slf4j.LoggerFactory;

…

**private** **final** Logger logger = LoggerFactory.*getLogger*(TestDevice.**class**);

For details about SLF4J, please refer to its documentation: <http://www.slf4j.org/docs.html>

SLF4J is an abstraction layer for various logging frameworks (ie. logback, log4j, java.util.logging…). It allows the end user to choose the logging framework at deployment time. Nevertheless, the logging configuration is framework dependent.

A configuration file allows configuring the logging output to be directed to the console, files, e-mails… It also configures the logging level. This file has to be in the class path of the device. See annexes for an example of a logback configuration file and <http://logback.qos.ch/> for details about configuration.

LIMITATION: JTangoServer depends directly on logback, because it has to implement some particularities to configure it:

* Configuration of the logging level
* Configuration of logging into file or into another device (for logviewer application).

So logback may be used to benefit from the above configuration topics (accessible through the administration device).

# Start up: DB/NO DB

## Server with Tango Database

A device class may contain a main method to start its server. It should call “start” of org.tango.server.ServerManager.

**public** **static** **void** main(**final** String[] args) {

ServerManager.*getInstance*().start(args, TestDevice.**class**);

}

When using the Tango database, the java system property or environment variable TANGO\_HOST must be defined to indicate the host and port of the database. The string array passed in the start method must contain at least the instance name as it has been previously defined in the Tango database. The other options are:

* The "-h" option displays the list of instances declared in the tango database for the given server.
* The “–v x“ option allows to override the default logging level (also called root level) of the logging configuration file where x is a integer value (possible values are OFF=0, FATAL = 1, ERROR = 2, WARN = 3, INFO = 4, DEBUG = 5, TRACE = 6)

It is possible to have several classes in a single server. Here is an example of a server started with two classes (org.tango.Motor and org.tango.PowerSupply):

// add class org.tango.Motor to the server (to be declared as “Motor” in the tango db)

ServerManager.getInstance().addClass(org.tango.Motor.**class**.getSimpleName(), org.tango.Motor.**class**);

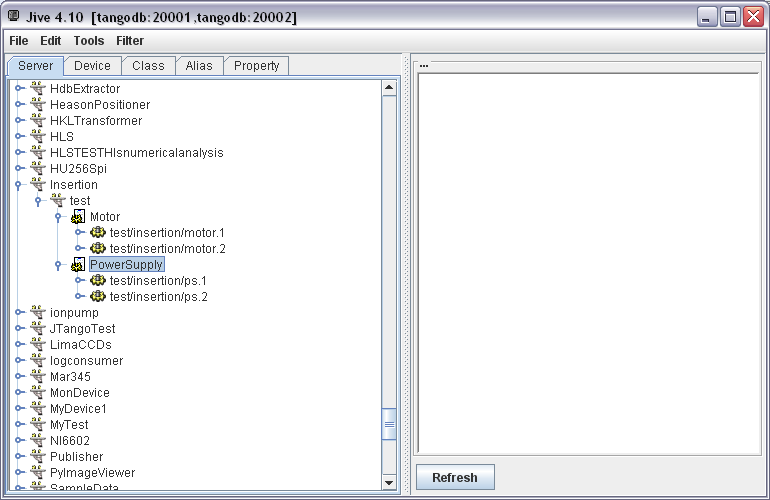
// add class org.tango.PowerSupply to the server (to be declared as “PowerSupply” in the tango db)

ServerManager.getInstance().addClass(org.tango.PowerSupply.**class**.getSimpleName(),org.tango. PowerSupply.**class**);

// start the server “Insertion/test”

ServerManager.getInstance().start(**new** String[] {"test"}, "Insertion");

The following screenshot shows an example declaration of the server “Insertion/test” in the tango db; it contains 4 devices, 2 of class Motor and 2 of class PowerSupply:



## Device without Tango database

A device may also be started without a Tango database, for example to perform unit tests. The system property OAPort (used by JacORB) must specify the port on which the server is started. The following code starts a device “1/1/1” on the port 12354 (NB: a client will connect to it with an address like "tango://localhost:12354/1/1/1#dbase=no")

**public** **static** **final** String *NO\_DB\_DEVICE\_NAME* = "1/1/1";

**public** **static** **final** String *NO\_DB\_GIOP\_PORT* = "12354";

**public** **static** **final** String *NO\_DB\_INSTANCE\_NAME* = "1";

**…**

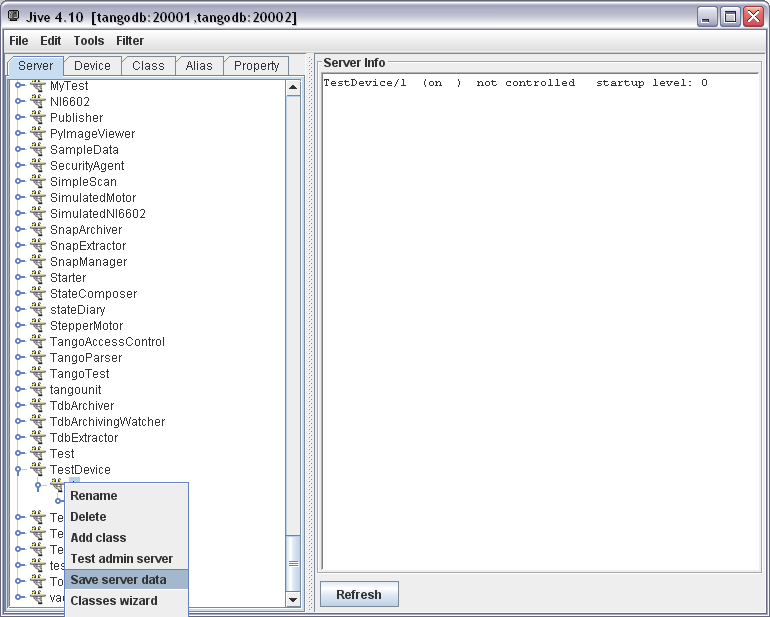
System.*setProperty*("OAPort", *NO\_DB\_GIOP\_PORT*);

ServerManager.*getInstance*().start(**new** String[] { *NO\_DB\_INSTANCE\_NAME*, "-nodb", "-dlist", *NO\_DB\_DEVICE\_NAME* },

TestDevice.**class**);

The start options are for a no db server:

* -nodb to indicate a server without database
* -dlist the list of devices in the server
* -file= the properties file. As the device and class properties are normally defined in the Tango DB, a file can be specified to replace it. (Refer to annexes for an example). If the device started without database is also defined in tango db, it is possible to generate its file with Jive. The “Save server data” menu is accessible by right-clicking on the instance name:



Example:

System.*setProperty*("OAPort", *NO\_DB\_GIOP\_PORT*);

ServerManager.*getInstance*().start(

**new** String[] { *NO\_DB\_INSTANCE\_NAME*, "-nodb", "-dlist", *NO\_DB\_DEVICE\_NAME*,

"-file=" + TestDevice.**class**.getResource("/noDbproperties.txt").getPath() }, TestDevice.**class**);

# Annexes

## Full sample device code

**package** org.tango.test;

**import** org.slf4j.Logger;

**import** org.slf4j.LoggerFactory;

**import** org.tango.server.ServerManager;

**import** org.tango.server.annotation.Attribute;

**import** org.tango.server.annotation.Command;

**import** org.tango.server.annotation.Delete;

**import** org.tango.server.annotation.Device;

**import** org.tango.server.annotation.Init;

@Device

**public** **class** TestDevice {

**private** **final** Logger logger = LoggerFactory.*getLogger*(TestDevice.**class**);

/\*\*

\* Attribute myAttribute READ WRITE, type DevDouble.

\*/

@Attribute

**public** **double** myAttribute;

/\*\*

\* Starts the server.

\*/

**public** **static** **void** main(**final** String[] args) {

ServerManager.*getInstance*().start(args, TestDevice.**class**);

}

/\*\*

\* init device

\*/

@Init

**public** **void** init() {

logger.debug("init");

}

/\*\*

\* delete device

\*/

@Delete

**public** **void** delete() {

logger.debug("delete");

}

/\*\*

\* Execute command start. Type VOID-VOID

\*/

@Command

**public** **void** start() {

logger.debug("start");

}

/\*\*

\* Read attribute myAttribute.

\*

\* **@return**

\*/

**public** **double** getMyAttribute() {

logger.debug("getMyAttribute {}", myAttribute);

**return** myAttribute;

}

/\*\*

\* Write attribute myAttribute

\*

\* **@param** myAttribute

\*/

**public** **void** setMyAttribute(**final** **double** myAttribute) {

logger.debug("setMyAttribute {}", myAttribute);

**this**.myAttribute = myAttribute;

}

}

## Command with ICommandBehavior

**package** org.tango.test;

**import** org.tango.server.StateMachineBehavior;

**import** org.tango.server.command.CommandConfiguration;

**import** org.tango.server.command.ICommandBehavior;

**import** fr.esrf.Tango.DevFailed;

**public** **class** TestDynamicCommand **implements** ICommandBehavior {

@Override

**public** CommandConfiguration getConfiguration() **throws** DevFailed {

**final** CommandConfiguration config = **new** CommandConfiguration();

config.setName("testDynCmd");

config.setInType(**void**.**class**);

config.setOutType(**double**.**class**);

**return** config;

}

@Override

**public** Object execute(**final** Object arg) **throws** DevFailed {

**return** 10.0;

}

@Override

**public** StateMachineBehavior getStateMachine() **throws** DevFailed {

**return** **null**;

}

}

## Attribute with IAttributeBehavior

**package** org.tango.test;

**import** org.tango.server.StateMachineBehavior;

**import** org.tango.server.attribute.AttributeConfiguration;

**import** org.tango.server.attribute.AttributeValue;

**import** org.tango.server.attribute.IAttributeBehavior;

**import** fr.esrf.Tango.AttrWriteType;

**import** fr.esrf.Tango.DevFailed;

/\*\*

\* A sample attribute

\*

\*/

**public** **class** TestDynamicAttribute **implements** IAttributeBehavior {

**private** **double** readValue = 0;

**private** **double** writeValue = 0;

/\*\*

\* Configure the attribute

\*/

@Override

**public** AttributeConfiguration getConfiguration() **throws** DevFailed {

**final** AttributeConfiguration config = **new** AttributeConfiguration();

config.setName("testDynAttr");

// attribute testDynAttr is a DevDouble

config.setType(**double**.**class**);

// attribute testDynAttr is READ\_WRITE

config.setWritable(AttrWriteType.*READ\_WRITE*);

**return** config;

}

/\*\*

\* Read the attribute

\*/

@Override

**public** AttributeValue getValue() **throws** DevFailed {

readValue = readValue + writeValue;

**return** **new** AttributeValue(readValue);

}

/\*\*

\* Write the attribute

\*/

@Override

**public** **void** setValue(**final** AttributeValue value) **throws** DevFailed {

writeValue = (Double) value.getValue();

}

/\*\*

\* Configure state machine if needed

\*/

@Override

**public** StateMachineBehavior getStateMachine() **throws** DevFailed {

**final** StateMachineBehavior stateMachine = **new** StateMachineBehavior();

stateMachine.setDeniedStates(DeviceState.*FAULT*);

stateMachine.setEndState(DeviceState.*ON*);

**return** stateMachine;

}

@Override

**public** AttributeValue getSetValue() **throws** DevFailed {

**return** **new** AttributeValue(writeValue);

}

}

## Extended example

**package** org.tango.test;

**import** java.util.Map;

**import** org.slf4j.Logger;

**import** org.slf4j.LoggerFactory;

**import** org.tango.DeviceState;

**import** org.tango.server.ServerManager;

**import** org.tango.server.annotation.Attribute;

**import** org.tango.server.annotation.ClassProperty;

**import** org.tango.server.annotation.Command;

**import** org.tango.server.annotation.Delete;

**import** org.tango.server.annotation.Device;

**import** org.tango.server.annotation.DeviceProperties;

**import** org.tango.server.annotation.DeviceProperty;

**import** org.tango.server.annotation.DynamicManagement;

**import** org.tango.server.annotation.Init;

**import** org.tango.server.annotation.State;

**import** org.tango.server.annotation.StateMachine;

**import** org.tango.server.dynamic.DynamicManager;

**import** org.tango.server.testserver.JTangoTest;

**import** fr.esrf.Tango.DevFailed;

@Device

**public** **class** TestDevice {

**private** **final** Logger logger = LoggerFactory.*getLogger*(TestDevice.**class**);

/\*\*

\* A device property

\*/

@DeviceProperty(defaultValue = "", description = "an example device property")

**private** String myProp;

@ClassProperty(defaultValue = "0", description = "an example class property")

**private** **int** myClassProp;

@DeviceProperties

**private** Map<String, String[]> deviceProperties;

/\*\*

\* Attribute myAttribute READ WRITE, type DevDouble.

\*/

@Attribute

**public** **double** myAttribute;

/\*\*

\* Manage dynamic attributes and commands

\*/

@DynamicManagement

**public** DynamicManager dynamicManager;

/\*\*

\* Manage state of the device

\*/

@State

**private** DeviceState state = DeviceState.*OFF*;

/\*\*

\* Starts the server.

\*/

**public** **static** **void** main(**final** String[] args) {

ServerManager.*getInstance*().start(args, TestDevice.**class**);

}

**public** **static** **final** String *NO\_DB\_DEVICE\_NAME* = "1/1/1";

**public** **static** **final** String *NO\_DB\_GIOP\_PORT* = "12354";

**public** **static** **final** String *NO\_DB\_INSTANCE\_NAME* = "1";

/\*\*

\* Starts the server in nodb mode.

\*

\* **@throws** DevFailed

\*/

**public** **static** **void** startNoDb() {

System.*setProperty*("OAPort", *NO\_DB\_GIOP\_PORT*);

ServerManager.*getInstance*().start(**new** String[] { *NO\_DB\_INSTANCE\_NAME*, "-nodb", "-dlist", *NO\_DB\_DEVICE\_NAME* },

TestDevice.**class**);

}

/\*\*

\* Starts the server in nodb mode with a file for device and class properties

\*

\* **@throws** DevFailed

\*/

**public** **static** **void** startNoDbFile() **throws** DevFailed {

System.*setProperty*("OAPort", *NO\_DB\_GIOP\_PORT*);

ServerManager.*getInstance*().start(

**new** String[] { *NO\_DB\_INSTANCE\_NAME*, "-nodb", "-dlist", *NO\_DB\_DEVICE\_NAME*,

"-file=" + JTangoTest.**class**.getResource("/noDbproperties.txt").getPath() }, TestDevice.**class**);

}

/\*\*

\* init device

\*

\* **@throws** DevFailed

\*/

@Init

@StateMachine(endState = DeviceState.*ON*)

**public** **void** init() **throws** DevFailed {

logger.debug("myProp value = {}", myProp);

logger.debug("myClassProp value = {}", myClassProp);

logger.debug("deviceProperties value = {}", deviceProperties);

// create a new dynamic attribute

dynamicManager.addAttribute(**new** TestDynamicAttribute());

// create a new dynamic command

dynamicManager.addCommand(**new** TestDynamicCommand());

logger.debug("init done");

}

/\*\*

\* delete device

\*

\* **@throws** DevFailed

\*/

@Delete

**public** **void** delete() **throws** DevFailed {

logger.debug("delete");

// remove all dynamic commands and attributes

dynamicManager.clearAll();

}

/\*\*

\* Execute command start.

\*/

@Command

@StateMachine(endState = DeviceState.*RUNNING*, deniedStates = DeviceState.*FAULT*)

**public** **void** start() {

logger.debug("start");

}

/\*\*

\* Read attribute myAttribute.

\*

\* **@return**

\*/

**public** **double** getMyAttribute() {

logger.debug("getMyAttribute {}", myAttribute);

**return** myAttribute;

}

/\*\*

\* Write attribute myAttribute

\*

\* **@param** myAttribute

\*/

**public** **void** setMyAttribute(**final** **double** myAttribute) {

logger.debug("setMyAttribute {}", myAttribute);

**this**.myAttribute = myAttribute;

}

**public** **void** setMyProp(**final** String myProp) {

**this**.myProp = myProp;

}

**public** **void** setMyClassProp(**final** **int** myClassProp) {

**this**.myClassProp = myClassProp;

}

**public** Map<String, String[]> getDeviceProperties() {

**return** deviceProperties;

}

**public** DeviceState getState() {

**return** state;

}

**public** **void** setState(**final** DeviceState state) {

**this**.state = state;

}

}

## Logging configuration with logback

In this example, the logging is output to the console. The underlying APIs Jacorb and ehcache will log only errors while the classes “org.tango.test” will log in debug level. And the rest of classes will log in debug (root level). See <http://logback.qos.ch/manual/configuration.html> for details.

<?xml version=*"1.0"* encoding=*"UTF-8"* ?>

<configuration>

<jmxConfigurator />

<appender name=*"CONSOLE"* class=*"ch.qos.logback.core.ConsoleAppender"*>

<layout class=*"ch.qos.logback.classic.PatternLayout"*>

<pattern>%-5level %d{HH:mm:ss.SSS} [%thread - %X{deviceName}] %logger{36}.%M:%L - %msg%n</pattern>

</layout>

</appender>

<logger name=*"jacorb"* level=*"ERROR"* />

<logger name=*"net.sf.ehcache"* level=*"ERROR"* />

<logger name=*"org.tango"* level=*"ERROR"* />

<logger name=*"org.tango.test"* level=*"DEBUG"* />

<root level=*"DEBUG"*>

<appender-ref ref=*"CONSOLE"* />

</root>

</configuration>

## Properties file for a device without Tango Database

# --- 1/1/1 properties

1/1/1->myProp:titi

CLASS/TestDevice->myClassProp: 10