



Guide for developers of Tango devices

Tango Device in Java

Design guidelines

ABEILLE Gwenaëlle

Table of contents

1.	Introduction.....	3
2.	A first device.....	4
3.	Device	6
4.	Command	7
5.	Attribute	8
6.	Init.....	10
7.	Delete	11
8.	State.....	12
9.	Status.....	12
10.	Device property	13
11.	Device properties	13
12.	Class property.....	14
13.	Around Invoke	14
14.	State machine.....	15
15.	Device Manager.....	15
16.	Dynamic API.....	16
a.	Dynamic Command	16
i.	Configuration.....	16
ii.	StateMachine	17
iii.	Execution	17
b.	Dynamic Attribute	17
i.	Configuration.....	17
i.	StateMachine	18
ii.	Read attribute	18
iii.	Write attribute	18
iv.	Update write part	19
17.	JTangoServerLang library	19
18.	Error management	19
19.	Logging	19
20.	Start up: DB/NO DB	21
a.	Server with Tango Database.....	21
b.	Device without Tango database	22
21.	Annexes	23

a. Full sample device code	23
b. Command with ICommandBehavior	24
c. Attribute with IAttributeBehavior	24
d. Extended example	26
e. Logging configuration with logback	29
f. Properties file for a device without Tango Database	29

1. Introduction

This paper is a documentation intended for developers. It describes 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 at least the first 3 chapters of 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>

2. A first device

Here is the code of simple device 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");
    }

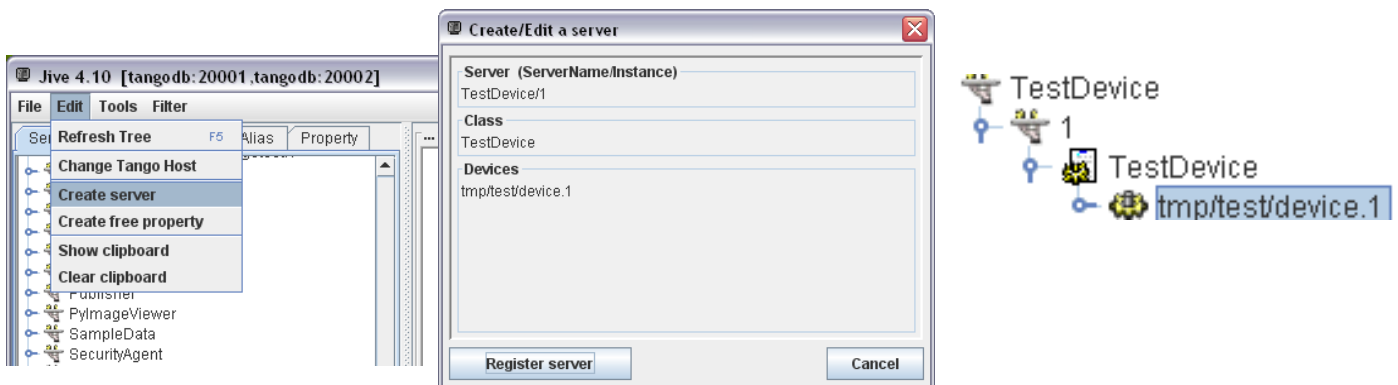
    /**
     * delete device
     */
    @Delete
    public void delete() {
        logger.debug("delete");
    }

    /**
     * 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;
    }
}
```

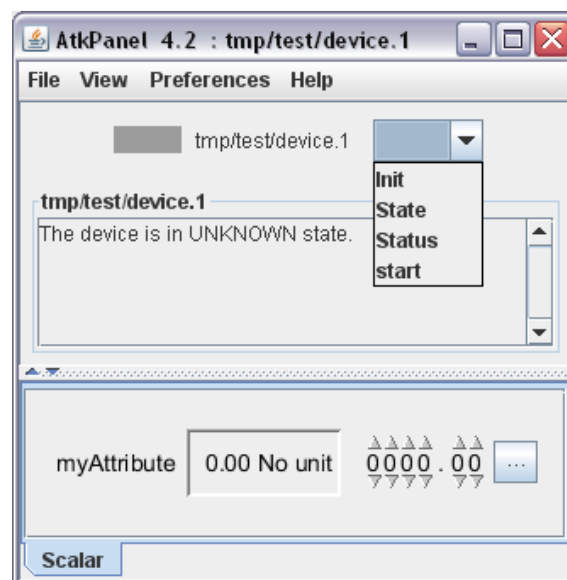
Before starting it, the device has to be declared in the Tango database with Jive with menu “Create server” (cf. chapter Jive for details...). Hereafter, a server “TestDevice/1” with one device “tmp/test/device.1” is created:



As the TestDevice class of this device has a main method, it has to 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 ATKPanel generic client (Tango concept reminder: The commands *Init*, *State*, *Status* and the attributes *State*, *Status* are created by default for any device):



A first code explanation:

- The “@Device” annotation on the 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 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 and when “Init” command is called.
- The “@Delete” annotation defines a method called at server shutdown, and at “Init” command, before “@Init”.
- Main method: to start the server
- The logger field is to log. It is based on slf4j (<http://www.slf4j.org/>). See annexes for example of configuration.

The following chapters will describes all this in details.

3. Device

A Tango device is created with the Java annotation on a class:

```
org.tango.server.annotation.Device
```

```
@Device
public class TestDevice {
}
```

This class must 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 retains across transactions.

4. Command

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

```
org.tango.server.annotation.Command
```

Example code:

```
@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[]	DEVVARCHARRAY
DevEncoded	DEVENCODED
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.

5. Attribute

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

```
org.tango.server.annotation.Attribute
```

Example code:

```
@Attribute
private double testAttribute;

public double getTestAttribute() {
    return testAttribute;
}

public void setTestAttribute(double testAttribute) {
    this.testAttribute = testAttribute;
}
```

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 start by "is"). The getter and setter have to be public while the field is private.

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

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

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	DEVENCODED	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, DEVUSHORT. It is possible to define these types with a dynamic command. 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. By default, it is not accessible. 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 method 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;
}
```

Default attribute properties are configurable with 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;
```

6. Init

`org.tango.server.annotation.Init`

```
@Init
public void init() {
}
```

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.

Annotation option: “lazyLoading”. 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.

7. 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 and at server shutdown. The delete method is generally used to close resources.

8. 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 its 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.

9. 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.

10. Device property

org.tango.server.annotation.DeviceProperty

NB: Tango reminder: order of loading 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.

11. Device properties

org.tango.server.annotation.DeviceProperties

It is possible to retrieve all device properties at once. Can be useful if some device properties are not known in advance (Example: some dynamic attributes that has 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.

12. Class property

org.tango.server.annotation.ClassProperty

```
@ClassProperty
private double[] classPropTest = new double[] {1,2};

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.

13. Around Invoke

org.tango.server.annotation.AroundInvoke

It defines a public void method with a single parameter of class `org.tango.server.InvocationContext` that 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()));
}
```

14. State machine

org.tango.server.annotation.StateMachine

The StateMachine annotation allows to define some denied state, 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
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;
}
```

15. Device Manager

org.tango.server.annotation.DeviceManagement

DeviceManager contains commons to for a device. For example, it provides its name, the admin device name, the 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;
}
```


16. Dynamic API

Attributes and commands can be created dynamically with the class `org.tango.server.dynamic.DynamicManager` that will be injected by 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.

a. Dynamic Command

A dynamic command is a class that must implements `org.tango.server.command.ICommandBehavior`. See annexes for a full sample code.

i. Configuration

```
public CommandConfiguration getConfiguration() throws DevFailed {
    final CommandConfiguration config = new CommandConfiguration();
    config.setName("testDynCmd");
    config.setInType(void.class);
    config.setOutType(double.class);
    return config;
}
```

ii. 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;
}
```

iii. Execution

The input and output types of the execute method is define 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;
}
```

NB: 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 and format.
- `setTangoInType(int tangoType)` or `setTangoOutType`: defines the type with a integer (constant 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, DEVVARUSHORTARRAY.

b. Dynamic Attribute

A dynamic attribute is a class that must implements `org.tango.server.attribute.IAttributeBehavior`. See annexes for a full sample code.

i. Configuration

Firstly, the method “`getConfiguration`” returns the full configuration of the attribute (see javadoc for `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;
}
```

NB: 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 a integer (constant 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.

i. 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;
}
```

ii. 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);
}
```

iii. 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();
}
```

iv. Update write part

For 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);  
}
```

17. JTangoServerLang library

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

- Attribute and command proxies
- Group command
- ...

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.

18. 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");  
}
```

19. Logging

The Java Tango server API uses SLF4J (<http://www.slf4j.org/>). The underlying libraries use also SLF4J (i.e. jacob, 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 logging, please refer to its documentation: <http://www.slf4j.org/docs.html>

As SLF4J is an abstraction layer for various logging framework, a framework must be chosen by the developer (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. See annexes for an example of a logback configuration file.

LIMITATION: JTangoServer depends directly on logback (<http://logback.qos.ch/>), 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).

20. Start up: DB/NO DB

a. Server with Tango Database

A device may also contain a main method where it is started. It should contain a call to start of `org.tango.server.ServerManager`.

```
public static void main(final String[] args) {  
    ServerManager.getInstance().start(args, TestDevice.class);  
}
```

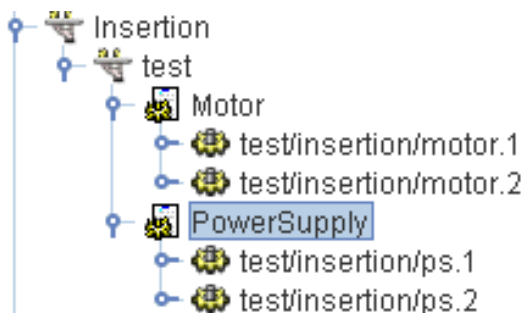
When using the Tango database, the java system property or environment variable `TANGO_HOST` must be defined to indicate the host and port of the database. The string array passed in the start method must contain at least the instance name as it has been previously defined in the Tango database. The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

- The `"-h"` option display the list of instances declared in the tango database for the given server.
- The `"-v x"` option allow 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 several 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.getSimple  
Name(),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:



b. Device without Tango database

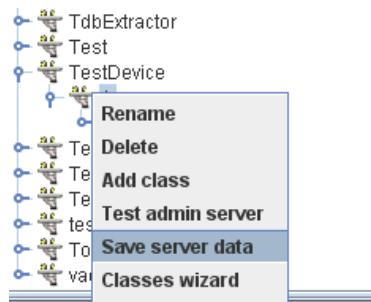
A device may also be started without a Tango database, for example perform do unit tests. The system property `OAPort` (used by JacORB) must specify the port on which the server is started. The following code starts a device "1/1/1" on the port 12354 (NB: a client will connect to it with an address like "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";

public static void main(final String[] args) {
    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 accessible by right-clicking on the instance name:



Example:

```
public static void main(final String[] args) {
    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);
}
```

21. Annexes

a. 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;
    }
}

```

b. 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;
    }
}

```

c. 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);
    }
}

```

```
}
```

d. 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;
    }
}

```

See tango-controls.org/... to download code.

e. Logging configuration with logback

```

<?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="or.tango.test" level="DEBUG" />

    <root level="DEBUG">
        <appender-ref ref="CONSOLE" />
    </root>

</configuration>

```

f. Properties file for a device without Tango Database

```

# --- 1/1/1 properties
1/1/1->myProp:titi

```

```

CLASS/TestDevice->myClassProp: 10

```