



Watchdog

- [1 Introduction](#)
- [2 Install and Configure Watchdog](#)
 - [2.1 Requirements](#)
 - [2.2 Installation](#)
 - [2.3 JAR Files](#)
 - [2.4 E-Mail Server Configuration](#)
 - [2.5 SSH Configuration](#)
 - [2.6 DRMAA Cluster Configuration](#)
- [3 Watchdog Overview](#)
 - [3.1 Modules](#)
 - [3.2 Basic XML structure](#)
- [4 Detailed XML format explanation](#)
 - [4.1 Process blocks](#)
 - [4.2 Dependencies](#)
 - [4.3 Execution environments](#)
 - [4.4 Global constants](#)
 - [4.5 Environment variables](#)
 - [4.6 Mail notification](#)
 - [4.7 Standard streams and working directory](#)
 - [4.8 Task actions](#)
 - [4.9 Simple calculations](#)
 - [4.10 Multiple module search folders](#)
 - [4.11 Custom success and error checker](#)
- [5 Creating custom modules](#)
 - [5.1 Input parameter definition](#)
 - [5.2 Output parameter definition](#)
 - [5.3 Binary call command and other settings](#)
 - [5.4 Assign a name to the new module](#)
 - [5.5 Putting it all together](#)
 - [5.6 Module versions](#)
 - [5.7 Other matters](#)
- [6 Extend Watchdog's functionality](#)
 - [6.1 Virtual file systems for task actions](#)
 - [6.2 XML Plugins](#)
- [7 Docker](#)
 - [7.1 Install the Watchdog Docker image](#)
 - [7.2 Sharing of files](#)
 - [7.3 Port forwarding](#)
 - [7.4 How to use the Docker Watchdog image](#)
 - [7.5 Use Docker in modules](#)

1 Introduction

Here, we present Watchdog, a WMS for the automated and distributed analysis of large-scale experimental data. Watchdog is implemented in Java and is thus platform-independent.

Main features include:

- straightforward processing of replicate data
- support for distributed computer systems
- remote storage support
- customizable error detection
- manual intervention into workflow execution
- a GUI for workflow construction using pre-defined modules
- a helper script for creating new module definitions
- no restriction to specific programming languages
- provides a flexible plugin system for extending without modifying the original sources

New features in Watchdog 2.0:

- *[new]* execution: executed only altered or unfinished tasks
- *[new]* execution: detach from workflow execution and attach later on
- *[new]* execution: custom before and after command script
- *[new]* graphical user interface for module creation
- *[new]* documentation: documentation format for modules
- *[new]* documentation: generation of module reference book
- *[new]* documentation: documentation template generator
- *[new]* reporting: introduced module versions
- *[new]* reporting: retrieval of third-party software versions
- *[new]* reporting: basic workflow execution reporting
- *[new]* sharing: community platform for sharing of modules and workflows

2 Install and Configure Watchdog

2.1 Requirements

Watchdog is platform independent as it is written in Java and requires JRE (Java Runtime Environment) 11 or higher (the GUI is build on JavaFX that is not part of any JRE builds). Modules previously delivered with Watchdog and are now located at [watchdog-wms/watchdog-wms-modules](#) and can be installed using `modules/downloadCommunityModules.sh`.

Most of these modules require a unix based system and were tested on SUSE Linux. They might not run without any changes on other unix systems or Windows but users can define custom modules that are compatible with their installed software and operating system. Each of the modules might require additional software to be installed. These requirements can be checked for the modules that are delivered with Watchdog with the help of `helper_scripts/dependencyTest.sh` as described below.

2.2 Installation

The installation of Watchdog is very easy. Simply extract the provided archive into a folder of your choice using `tar xfvz watchdog.tar.gz`. The folder must be accessible for remote or cluster executors if you plan to use some. Afterwards you should download Watchdog modules from the [community_github_repository](#) and extract them into the `modules/` folder or run `modules/downloadCommunityModules.sh`.

Alternatively Watchdog can be installed automatically via conda using `conda install -c bioconda watchdog-wms`. In that case binaries are named `watchdog-cmd` and `watchdog-gui` while the rest of the files is located in `${PREFIX}/share/watchdog-wms-${VERSION}`. If you want to use Watchdog with Docker, read section [7](#).

In the next few lines the content of each folder is explained:

- `core_lib`: some core functions that can be used in bash module scripts
- `documentation`: contains Watchdog's documentation in HTML- and PDF-format
- `examples`: contains the examples that are also presented in the documentation
- `helper_scripts`: scripts for generating new modules, configure the examples or testing of all modules
- `jars`: runnable JAR-files that are build from Watchdog's source code
- `java_external_lib`: external libraries
- `java_source`: Watchdog's source code
- `modules`: must contain all modules that should be used in workflows
- `test_data`: contains some test data that is used by multiple modules
- `tmp`: is used for Watchdog's temporary files
- `webserver_data`: data which is accessed by the internal webserver
- `xsd`: definition of the module and workflow in xsd format

To test if all software required by installed modules that call bash scripts is available the script `helper_scripts/dependencyTest.sh` can be executed. It checks if all requirements are installed that are also enforced by the module itself. Only dependencies that are defined in the bash script itself stored in the variable named `$USED_TOOLS` are detected by the script. The system must be able to locate the dependencies by using the `PATH` variable. Moreover, all R and perl packages that are used in scripts are checked.

In order to test if all modules that provide tests work as expected on your system you can run `helper_scripts/moduleTest.sh`. If you want to test the examples which are discussed in this manual, you can configure them by running:

```
helper_scripts/configureExamples.sh -i  
/path/to/install/folder/of/watchdog [-m your@mail-adress.com] (mail  
attribute (-m) is optional)
```

Afterwards the configured examples will be located in `/path/to/install/folder/of/watchdog/examples/` and can be executed (from the watchdog installation directory) using the following command: `./watchdog.sh -x examples/filename.xml` or alternatively `java -jar jars/watchdog.jar -x examples/filename.xml`

For instance: `./watchdog.sh -x examples/workflow1_basic_information_extraction.xml`

If you want to use the workflow designer (GUI), you can start it by using (from the watchdog installation directory): `./workflowDesigner.sh` or alternatively `java -jar jars/WatchdogDesigner.jar`

2.3 JAR Files

These runnable JAR-files are shipped together with Watchdog:

- `watchdog.jar`: command-line tool that executes Watchdog workflows
- `watchdogDesigner.jar`: graphical user interface for workflow design and execution
- `moduleCreator.jar`: provides a graphical user interface for module creation
- `libraryGenerator.jar`: creates a module reference book based on a set of modules
- `docuTemplateExtractor.jar`: generates templates for module documentation
- `methodTemplateGenerator.jar`: basic reporting of steps performed during execution of a workflow
- `moduleValidator.jar`: command-line tool that can be used to verify integrity of modules

2.4 E-Mail Server Configuration

As Watchdog will send e-mails it needs a working mail configuration. If you don't want Watchdog to send e-mails, simply don't use the `mail` attribute of the `<tasks>` tag. In that case the content of the mails will be printed to the standard output stream.

By default a server listening on SMTP port 25 is expected that accepts mails without authentication. In order to use another configuration the parameter `-mailConfig` of Watchdog can be used. It expects a tab-separated file that contains information on how to connect to the mail server using the SMTP protocol. If the mail server expects some authentication we strongly suggest to use a mail account that was explicitly created for the use with Watchdog as the password is stored unencrypted.

Example 1: Example mail config for a gmail account

```
1 mail.smtp.auth true
2 mail.smtp.host smtp.googlemail.com
3 mail.smtp.port 587
4 mail.smtp.user johns_watchdog@gmail.com
5 mail.smtp.pw r9x74l(klsab
6 mail.smtp.from johns_watchdog@gmail.com
7 mail.smtp.starttls.enable true
```

Example 1 shows a configuration for a gmail account. More information about the variables that can be used can be found [here](#). An template mail config file that can be edited can be found in `examples/mail_config` once the examples are configured as described above.

2.5 SSH Configuration

Watchdog supports execution of tasks via ssh on remote hosts. In order to use that feature a private ssh key must be provided. It is strongly recommended that the private key is protected by a passphrase. In that case the passphrase must be entered after Watchdog was started and will be hold encrypted in memory until the passphrase is needed.

A key pair that can be used for ssh authentication can be generated using the tool `ssh-keygen` that is part of `openssh`. If you need further information you can find many online tutorials that explain how to use a private key for ssh authentication. E.g. [How To Set Up SSH Keys](#) and [SSH/OpenSSH/Keys](#)

2.6 DRMAA Cluster Configuration

Watchdog supports cluster solutions which provide a `DRMAA` java binding. By default it is bundled with a DRMAA binding for the `sun grid engine` (SGE 6.1).

The following environment variables must be set correctly in order to communicate with the SGE:

- `SGE_ROOT`: path to the installation folder of the SGE

- LD_LIBRARY_PATH: path to the library path of the SGE; in most cases it will be *\$SGE_ROOT/lib/lx24-amd64* or *\$SGE_ROOT/lib/lx24-x86*

Basically there are two ways to change the default cluster extension in order to use another DRMAA solution than the SGE:

- dynamically by adding arguments to the jar invocation:
 1. set class name of DRMAA Sessionfactory via -Dorg.ggf.drmaa.SessionFactory=classname
 2. add DRMAA java binding to class path via -cp */path/to/drmaaImplementation.jar*
- permanently by changing Watchdog's jar file:
 1. jar files can be opened and edited with every tool that supports zip files
 2. replace name of DRMAA Sessionfactory stored in */META-INF/services/org.ggf.drmaa.SessionFactory*
 3. add class files of the DRMAA java binding to Watchdog's jar file

Probably further settings are needed which depend on the used DRMAA library.

Alternatively, binary-based executors can be used, which are currently implemented for SLURM and SGE. The required control binaries must be accessible via the *\$PATH* variable.

3 Watchdog Overview

3.1 Modules

Modules represent re-usable components that perform certain tasks, e.g. compression of files or creating histograms. Watchdog is delivered with a set of predefined modules. Additionally the user has the possibility to define own modules as described in section 5. The modules are stored in the *modules* directory located in the root folder of the Watchdog installation. Each module is stored in its own folder and consists at least of an XSD file with the name of the module. The XSD file contains a definition of the parameters which can be set in the XML format and the tools which are executed in the background when the module is used.

Example 2: XSD definition of the sleep module

```
1  <?xml version="1.0" encoding="UTF-8" ?>
2  <x:schema xmlns:x="http://www.w3.org/2001/XMLSchema"
xmlns:vc="http://www.w3.org/2007/XMLSchema-versioning"
vc:minVersion="1.1" xmlns:xerces="http://xerces.apache.org">
3
4      <!-- definition of the task parameters -->
5      <x:complexType name="sleepTaskParameterType">
6          <x:all>
7              <x:element name="wait" type="paramWait_sleep"
minOccurs="1" maxOccurs="1" />
8          </x:all>
9      </x:complexType>
10
11      ...
12
13      <!-- make task definition available via substitution
group -->
14      <x:element name="sleepTask" type="sleepTaskType"
substitutionGroup="abstractTask" />
15
16      <!-- module specific parameter types -->
17      <x:complexType name="paramWait_sleep">
18          <x:simpleContent>
19              <x:restriction base="paramString">
20                  <x:assertion test="matches($value,
'($ {[A-Za-z_]+})|($(.+))|([({}($[A-Za-z_]+(,s*){0,1}){0,1}([0-
9]+(,s*){0,1}){0,1}[ ]))' ) or matches($value, '^ [0-9]+[smhd]
{0,1}$') " xerces:message="Parameter with name '{ $tag }' must match
[0-9]+[smhd]{0,1}." />
21              </x:restriction>
22          </x:simpleContent>
23      </x:complexType>
```

```
24
25     </x:schema>
```

Example 2 shows parts of the XSD definition of a module which are important to know for the user. At the beginning parameters and flags which are accepted by the module are defined in an element named *sleepTaskParameterType* (5-9). In this case only one parameter named *wait* is defined that must occur exactly once (7). The type of the parameter is specified at the bottom of the example (17-23). In this case the parameter is a string that must match a regex pattern, which first accepts numbers followed by a letter as optional suffix (19-21). Additionally values that are placeholders for constants or variables are allowed by the first *matches()* function whereby the user must take care that the replaced value is valid with regard to the second part of the specification (20). The attribute *name* of the element in line 14 defines how the module can be referenced in the XML file. In this example the module can be called using the name *sleepTask*.

3.2 Basic XML structure

Tasks which should be executed by Watchdog must be defined in an XML file. In the following the structure of the XML file is presented. The expression *<?Task>* is used to refer to a task which is not further specified. In general this syntax is used if some attributes are valid for all classes that inherit from that class type. Within the following examples these variables are user-specific and contain therefore no concrete values:

{%INSTALL%} - path to the root installation directory of Watchdog

{%MAIL%} - email address of the user

{%EXAMPLE_DATA%} - path to the folder in which the example data is located

You already have configured your examples by calling the script *helper_scripts/configureExamples.sh* as described in [2](#).

Example 3: Most basic XML input for Watchdog

```
1     <?xml version="1.0" encoding="UTF-8"?>
2     <watchdog xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xsi:noNamespaceSchemaLocation="watchdog.xsd"
watchdogBase="{%INSTALL%}" isTemplate="true">
3
4         <!-- begin task block and use that mail to inform the
user on success or failure -->
5         <tasks mail="{%MAIL%}">
6
7             <!-- definition a simple sleep task -->
8             <sleepTask id="1" name="sleep">
9                 <parameter>
10                     <wait>30s</wait>
11                 </parameter>
12             </sleepTask>
```



```

13         </tasks>
14     </watchdog>

```

Example 3 shows a most basic XML file that contains only a single task named *sleep*. Every XML file that should be parsed by Watchdog must contain a `<watchdog>` element as root element (2). The attribute `watchdogBase` of it must refer to the folder in which Watchdog was installed. The attribute `isTemplate` prevents Watchdog from executing workflows that contain variables that must be set by the user and is removed automatically by the configure script. Afterwards as childs of `<tasks>` the tasks which should be executed must be defined (5). Each task must contain an `id` and `name` attribute (8). With the `<parameter>` element, values can be assigned to the parameters of the task, which have to be specified in the XSD file of the module. Flags are activated by using `<flagName>true</flagName>` or `<flagName>1</flagName>` while parameters can be set with `<paramName>value</paramName>` (10).

Table 1: Attributes in the context of Watchdog

| element | attribute | type | function |
|-------------------------------|-----------------------------|---------|---|
| <code><watchdog></code> | <code>watchdogBase</code> | string | path to the install path of watchdog |
| <code><watchdog></code> | <code>isTemplate</code> | boolean | prevents Watchdog from executing unconfigured workflow templates; default: <i>false</i> |
| <code><tasks></code> | <code>[mail]</code> | string | mail which is used for notification; if not set, the content of the mails with be printed to the standard output stream; default: not set |
| <code><tasks></code> | <code>[projectName]</code> | string | name of the complete process; default: not set |
| <code><?Task></code> | <code>[id]</code> | integer | numeric id of the task; if not set all id's will be automatically generated; default: not set |
| <code><?Task></code> | <code>name</code> | string | name of the task |
| <code><?Task></code> | <code>[processBlock]</code> | string | processBlock as source of varying parameters (see 4.1) |
| <code><?Task></code> | <code>[executor]</code> | string | execution environment on which the task is executed (see 4.3) |
| <code><?Task></code> | <code>[environment]</code> | string | use globally defined environment variables (see 4.5) |
| <code><?Task></code> | <code>[maxRunning]</code> | integer | maximal number of simultaneously running tasks; default: not restricted |
| <code><?Task></code> | <code>[notify]</code> | enum | notification of the user via mail on success; enabled: release complete task at once when all subtasks are finished; subtask: release every subtask separately; default: <i>disabled</i> (see 4.6) |
| <code><?Task></code> | <code>[checkpoint]</code> | enum | does not schedule tasks which depend on this task until manually released by the user; enabled: release complete task at once when all subtasks are finished; subtask: release every subtask separately; default: <i>disabled</i> |
| <code><?Task></code> | <code>[confirmParam]</code> | enum | allows the user to modify the parameters before the task is scheduled; enabled: task will not be scheduled until the user checks the parameter; default: <i>disabled</i> |
| <code><?Task></code> | <code>[version]</code> | integer | version of the module that should be used for that task; default: <i>1</i> (see 5.6) |
| <code><?Task></code> | <code>[posX]</code> | integer | x coordinate for display in GUI; default: not set |
| <code><?Task></code> | <code>[posY]</code> | integer | y coordinate for display in GUI; default: not set |

In the following sections the structure of the XML format is described in greater detail.

4 Detailed XML format explanation

In the following sections the complete range of functions of Watchdog's XML format is explained. The following elements must be defined as child elements of the `<settings>` element before the `<tasks>` element begins and are valid within the complete XML file:

- `<processBlock>` - process a task with varying parameters (see [4.1](#))
- `<executors>` - define different executor environments (see [4.3](#))
- `<constants>` - defines constants that substitute placeholders (see [4.4](#))
- `<environments>` - define or update environment variables (see [4.5](#))
- `<modules>` - define multiple module include directories (see [4.10](#))

Apart from the `<parameter>` element, the following elements are allowed in `<?Task>` elements:

- `<environment>` - define or update environment variables (see [4.5](#))
- `<dependencies>` - define dependencies between tasks (see [4.2](#))
- `<streams>` - define location of standard streams and set a working directory (see [4.7](#))
- `<checkers>` - usage of custom success or error checkers (see [4.11](#))
- `<actions>` - define task actions that are performed before or after tasks execution (see [4.8](#))

4.1 Process blocks

Watchdog is able to process multiple tasks of the same type, which differ only in some parameter values, without the need to define all of these tasks separately. This function is referred to as **process blocks** while the tasks created by an process block are called **subtasks** of the task. There are four different possibilities to define process blocks as childs of the `<processBlock>` element:

- `<processSequence>` - argument is numeric
- `<processFolder>` - argument is a path to a file
- `<processInput>` - multiple arguments obtained from dependencies
- `<processTable>` - multiple arguments stored in a tab-separated file with names of variables stored in the first line

When the `<processBlock>` attribute of a task is set the argument of the process folder or sequence is substituted at run time within `<parameter>`, `<streams>`, `<checkers>`, `<actions>` and `<environment>` elements in the following manner:

- `<processSequence>` - `[]/{}/()` -> number
- `<processFolder>` - `{}` -> absolute path to the file
- `<processFolder>` - `()` -> absolute path to the parent folder of the file
- `<processFolder>` - `[]` -> name of the file
- `<processFolder>` - `[n]/{n}` -> *n* suffixes of the filename are truncated using *.* as separator

- **<processFolder>** - (*n*) -> *n* suffixes of the parent folder are truncated using / as separator
- **<processFolder>** - ({*n,sep*}) -> suffixes of the value are truncated using *sep* as separator (might also be a regex)
- **<processTable>** - ({*\$COL_NAME*}) -> value stored in the column named *\$COL_NAME*
- **<processTable>** - ({*\$COL_NAME,n,sep*}) -> value stored in the column named *\$COL_NAME* but with suffix truncation as described above
- **<processInput>** - ({*\$RET_NAME*}) -> return value of a dependency with the name *\$RET_NAME*
- **<processInput>** - ({*\$RET_NAME,n,sep*}) -> return value of a dependency with the name *\$RET_NAME* but with suffix truncation as described above

If a task depends on two tasks, which return variables with the same name, the return value of the task with the smaller id will be overwritten. Deviating from this, return values from separate dependencies will overwrite the ones from global dependencies if both use the same name for a variable.

Example 4: Definition of different process blocks

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <watchdog xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xsi:noNamespaceSchemaLocation="watchdog.xsd"
watchdogBase="{%INSTALL%}" isTemplate="true">
3
4      <settings>
5          <!-- definition of different process blocks -->
6          <processBlock>
7              <processSequence name="qualities" start="1"
end="9" step="2" />
8              <processFolder name="specialFiles" folder="
{%EXAMPLE_DATA%}/spec/" pattern="*.spec" />
9              <baseFolder folder="{%EXAMPLE_DATA%}/">
10                 <processFolder name="txtFiles"
folder="txt/" pattern="*.txt" />
11                 <processFolder name="txtFiles"
folder="other_txt/" pattern="*.txt" append="true" maxDepth="1" />
12                 <processFolder name="gzFiles"
folder="txt_zipped/" pattern="*.gz" disableExistenceCheck="true"
/>
13                 <processTable name="sleepTable"
table="processTable.input.txt" />
14                 </baseFolder>
15             </processBlock>
16         </settings>
17
18         <tasks mail="{%MAIL%}">
19             <!-- compress all files with *.txt ending in
/some/base/folder/TXT -->
20             <gzipTask id="1" name="compress files">

```

```

processBlock="txtFiles" checkpoint="enabled">
21         <parameter>
22             <input>{}</input>
23             <output>
24 {%EXAMPLE_DATA%}/txt_zipped/[1].gz</output>
25         </parameter>
26     </gzipTask>
27
28     <!-- test quality values 1,3,5,7 and 9 -->
29     <gzipTask id="2" name="quality test"
processBlock="qualities" checkpoint="subtask">
30         <dependencies>
31             <depends>1</depends>
32         </dependencies>
33         <parameter>
34             <input>
35 {%EXAMPLE_DATA%}/txt/txtFile1.txt</input>
36             <output>
37 {%EXAMPLE_DATA%}/qualityTest/txtFile1_q[].gz</output>
38             <quality>[]</quality>
39         </parameter>
40         <environment>
41             <var name="QUALITY">{}</var>
42         </environment>
43     </gzipTask>
44
45     <!-- sleep tasks which are created based on a
process table -->
46     <sleepTask id="3" name="table sleep"
processBlock="sleepTable">
47         <dependencies>
48             <depends>2</depends>
49         </dependencies>
50         <streams>
51             <stdout>{$OUT, 1}</stdout>
52         </streams>
53         <parameter>
54             <wait>{$DURATION}</wait>
55         </parameter>
56         <environment>
57             <var name="IMPORTANT_ID_RAW">
58 [$IMPORTANT_ID]</var>
59             <var
name="IMPORTANT_ID_CALC">${[$IMPORTANT_ID]*3}</var>
60             </environment>
61         </sleepTask>
62     </tasks>
63 </watchdog>

```

Example 4 shows three different ways how process blocks can be specified. First a `<processSequence>` named `qualities` is defined that creates the numbers 1,3,5,7 and 9 (7). In the next line a `<processFolder>` is defined that will process all files stored in `{%EXAMPLE_DATA%}/spec` that end with `.spec` (8). The syntax which must be used in the `pattern` attribute is the same as in bash. If a `<processFolder>` is a child element of a `<baseFolder>`, the `folder` attribute of the `<processFolder>` will be prefixed with the `folder` attribute of the `<baseFolder>` (9-14). The attribute `disableExistenceCheck` that is enabled for the `<processFolder>` with the name `gzFiles` causes Watchdog not to force the existence of the folder when it is started (12).

The task with `id 1` will compress all `.txt` files in the folders `{%EXAMPLE_DATA%}/txt` and `{%EXAMPLE_DATA%}/other_txt` and store them in `{%EXAMPLE_DATA%}/txt_zipped` (20-25). Whereas, the task with `id 2` will compress a file with different quality values (28-40). The compressed files will be stored with `txtFile1_q` as prefix and the used quality as suffix in `{%EXAMPLE_DATA%}/qualityTest` (34). Additionally, an environment variable with the name `QUALITY` is set which also contains the set quality (38). The sleep task at the end of the example shows how the columns of a `<processTable>` can be used as input (43-57). If the variable is of numeric type it can also be used within simple calculations which are presented in 4.9 (55).

Table 2: Attributes in the context of process blocks

| element | attribute | type | function |
|--------------------------------------|--------------------------------------|---------|--|
| <code><?ProcessBlock></code> | <code>name</code> | string | is used as reference in the <code>processBlock</code> attribute of a task |
| <code><?Task></code> | <code>processBlock</code> | string | name of a <code><?ProcessBlock></code> element |
| <code><?ProcessBlock></code> | <code>[append]</code> | boolean | if set to true, two or more process blocks of the same type can be merged; supported by processSequence and ProcessFolder; default: <code>false</code> |
| <code><processSequence></code> | <code>start</code> | double | inclusive start of the numeric series |
| <code><processSequence></code> | <code>[step]</code> | double | number that is added until the value is greater than end; default: <code>1</code> |
| <code><processSequence></code> | <code>end</code> | double | break condition, might be inclusive |
| <code><processFolder></code> | <code>folder</code> | integer | absolute or relative to a <code><baseFolder></code> path to a folder |
| <code><processFolder></code> | <code>pattern</code> | string | pattern selecting files that should be substituted; syntax as in bash |
| <code><processFolder></code> | <code>[ignore]</code> | string | files matching that pattern will be ignored; syntax as in bash; default: not set |
| <code><processFolder></code> | <code>[disableExistenceCheck]</code> | boolean | folder must not exist when Watchdog is started; default: <code>false</code> |
| <code><processFolder></code> | <code>[maxDepth]</code> | integer | a positive integer will cause that <code>maxDepth</code> levels of subdirectories are traversed while by default only the parent folder is processed; default: <code>0</code> |
| <code><baseFolder></code> | <code>folder</code> | string | absolute path which is used as prefix before the path of the <code><processFolder></code> is added |
| <code><baseFolder></code> | <code>[maxDepth]</code> | integer | see description of <code><processFolder></code> <code>[maxDepth]</code> ; if both are set, the value of the <code><processFolder></code> element is set; default: <code>0</code> |
| <code><processTable></code> | <code>table</code> | string | path to a tab-separated file with header; the column names must consist out of [A-Za-z_] |
| <code><processTable></code> | <code>[disableExistenceCheck]</code> | boolean | table file must not exist when Watchdog is started; default: <code>false</code> |

| element | attribute | type | function |
|----------------|---------------|--------|---|
| <processTable> | [compareName] | column | name that should be used to compare names of separate dependencies; default: complete line |
| <processInput> | sep | string | separator which is used to join multiple values of global dependencies together; default: . |
| <processInput> | [compareName] | string | name of return value that should be used to compare names of separate dependencies; default: name of precursor node |

4.2 Dependencies

By default all tasks specified in the XML document are independent from each other. That implies that all tasks are scheduled at the same time if no other constraints exist. It is possible to define dependencies between tasks using the **<depends>** element that expects as value the id or name of an already defined task. The element must be a child of a **<dependencies>** element. Without any arguments the task will not be scheduled until all (sub)tasks of the dependencies have finished successfully. By setting the **separate** argument to **true** a subtask can depend only on the corresponding subtask the task depends on. This option is only meaningful if both tasks are process block tasks and work on the same input set or a transformed version of it.

Example 5: Definition of dependencies

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <watchdog xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xsi:noNamespaceSchemaLocation="watchdog.xsd"
watchdogBase="{%INSTALL%}" isTemplate="true">
3
4      <settings>
5          <!-- definition of two process folders -->
6          <processBlock>
7              <baseFolder folder="{%EXAMPLE_DATA%}/">
8                  <processFolder name="txtFiles"
folder="txt/" pattern="*.txt" />
9                  <processFolder name="gzFiles"
folder="txt_zipped/" pattern="*.gz" disableExistenceCheck="true"
/>
10                 </baseFolder>
11             </processBlock>
12         </settings>
13
14         <tasks mail="{%MAIL%}">
15             <!-- definition a simple sleep task -->
16             <sleepTask id="1" name="sleep">
17                 <parameter>
18                     <wait>30s</wait>
19                 </parameter>
20             </sleepTask>
21
22             <!-- compress all files with *.txt ending in
/some/base/folder/TXT -->

```



```

23         <gzipTask id="2" name="compress"
processBlock="txtFiles">
24             <parameter>
25                 <input>{}</input>
26                 <output>
27                 {%EXAMPLE_DATA%}/txt_zipped/[1].gz</output>
28             </parameter>
29             <!-- dependency definition -->
30             <dependencies>
31                 <depends>1</depends>
32             </dependencies>
33         </gzipTask>
34
35         <!-- decompress all files with *.gz ending in
/some/base/folder/TXT_ZIPPED -->
36         <gzipTask id="3" name="decompress"
processBlock="gzFiles">
37             <parameter>
38                 <input>{}</input>
39                 <output>
40                 {%EXAMPLE_DATA%}/txt_decompressed/[1].txt</output>
41                 <decompress>true</decompress>
42             </parameter>
43             <!-- dependency definition -->
44             <dependencies>
45                 <depends separate="true" prefixName="
[1]">2</depends>
46             </dependencies>
47         </gzipTask>
48     </tasks>
49 </watchdog>

```

In example 5 a *compress* task with id *2* is defined which depends on the before defined *sleep* task (23-32). Additionally, a task, which will decompress the compressed files immediately after the compression is finished, is defined (35-45). In order to achieve this behavior the *separate* attribute is set to *true* (43). Because the *.txt* ending of the original filename was cropped and a *.gz* ending was added, only the first part of the filename is considered as specified in the *prefixName* attribute (26,43).

Table 3: Attributes in the context of dependencies

| element | attribute | type | function |
|----------------|--------------|---------|---|
| <dependencies> | | | parent of <depends> elements and child of <?Task> |
| <depends> | | integer | already defined task id on which the task should depend on |
| <depends> | [separate] | boolean | if set to <i>true</i> each subtask depends only on its corresponding subtask; default: <i>false</i> |
| <depends> | [keep4Slave] | boolean | if set to <i>true</i> a executor in slave mode will wait until all tasks with that id, which are running on that slave, are finished; only valid for separate dependencies; default: <i>false</i> |

| element | attribute | type | function |
|------------------------------|---------------------------|-----------------------|--|
| <code><depends></code> | <code>[prefixName]</code> | <code>[[0-9]*]</code> | only meaningful if <code>separate</code> is set to <code>true</code> ; defines in which manner the variables of the two process blocks must be equal to each other: <code>[]/[0]</code> : complete variables of the subtasks are compared <code>[n]</code> : it is checked if the variable of a subtask begins with the prefix of the finished subtask this task depends on; the first <code>n</code> parts are taken was prefix whereby <code>'.'</code> is used as separator; default: <code>[]</code> |
| <code><depends></code> | <code>[sep]</code> | string | separator which is used together with <code>prefixName</code> ; default: <code>.</code> |

4.3 Execution environments

By default the tasks are executed one after the other locally on the host which runs Watchdog. It is possible to define different execution environments using the `<executors>` element. Possible environments:

- `<local>` - task is executed on the local host
- `<remote>` - task is executed on a remote host using ssh
- `<cluster>` - task is executed on a computer cluster using DRMAA
- `<sge>` - task is executed on a SGE computer cluster using control binaries
- `<slurm>` - task is executed on a SLURM computer cluster using control binaries

Example 6: Definition of different execution environments

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <watchdog xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xsi:noNamespaceSchemaLocation="watchdog.xsd"
watchdogBase="{%INSTALL%}" isTemplate="true">
3
4      <settings>
5          <!-- examples of different execution environments
-->
6          <executors>
7              <local name="localhost" maxRunning="2" />
8              <sge name="defaultCluster" default="true"
memory="1G" beforeScripts="ulimitMemory.sh" queue="short.q" />
9              <sge name="highPerformanceCluster" slots="4"
memory="3G" maxRunning="4" queue="short.q" />
10             <remote name="superComputer"
user="mustermann" host="superComputer"
privateKey="/path/to/private/auth/key" port="22"
disableStrictHostCheck="false" />
11         </executors>
12     </settings>
13
14     <tasks mail="{%MAIL%}">
15         <!-- execute this task on the localhost -->
16         <sleepTask id="1" name="sleep"
executor="localhost">
17             <parameter>
18                 <wait>30s</wait>

```



```

19             </parameter>
20         </sleepTask>
21     </tasks>
22 </watchdog>

```

In example 6 four different execution environments are defined as childs of the **<executors>** element (6-11). Tasks scheduled on first executor will run on the host on which Watchdog was started (7).

The executor with the name *defaultCluster* is used by default and runs on the *short.q* queue of the computer cluster (8). Before the actual module command is executed on that executor, the commands stored in the before script *ulimitMemory.sh* are executed. If a relative path is given, the script must be located in *core_lib/executor_scripts*. In this example the script will enforce that the requested memory is not exceeded using the *ulimit* command.

The next executor is reserved for high performance tasks because it reserves 4 slots on the cluster with each slot consuming three gigabyte of main memory (9). In order not to occupy the complete computing power the attribute **maxRunning** is set to four which means that a maximum of four tasks will run simultaneously on that execution environment.

In line 10 an example for a remote executor is given which executes tasks via ssh using a host named *superComputer*.

Afterwards the same sleep task is defined as in the first example and will run on the local executor (16). The other executors can be tested once you adapted them to your local infrastructure (see [2.5](#) and [2.6](#)).

Table 4: Attributes in the context of execution environments

| element | attribute | type | function |
|---------------------------|--------------------------|---------|--|
| <? Executor> | name | string | is used as reference in the executor attribute of a task |
| <?Task> | executor | string | name of a <?Executor> element |
| <? Executor> | [environment] | string | environment with that name is used as default environment; default: not set |
| <? Executor> | [default] | boolean | defines which execution environment is taken as default; default: <i>false</i> |
| <? Executor> | [maxRunning] | integer | number of tasks that can run at the same time; default: not restricted |
| <? Executor> | [workingDir] | string | working directory to which the executor switches before task execution; default: <i>/usr/local/storage/</i> |
| <? Executor> | [stickToHost] | boolean | activates slave mode for that executor which means that tasks that depend on each other are executed on the same execution host; default: <i>false</i> |
| <? Executor> | [maxSlaveRunning] | integer | number of tasks that can run at the same time on a slave if stickToHost is enabled; default: 1 |

| element | attribute | type | function |
|---------------------------------|---------------------------------------|---------|--|
| <code><? Executor></code> | <code>[pathToJava]</code> | string | path to java binary which is used for slave mode execution; default: <code>/usr/bin/java</code> |
| <code><? Executor></code> | <code>[shebang]</code> | string | shebang, which is used if a temporary script is build that defines environment variables or before/after script commands; default: <code>#!/bin/bash</code> |
| <code><? Executor></code> | <code>[beforeScripts]</code> | string | path to a script containing commands that are executed before the actual module command; multiple scripts can be provided by using ':' as separator; default: not set |
| <code><? Executor></code> | <code>[afterScripts]</code> | string | path to a script containing commands that are executed after the actual module command; multiple scripts can be provided by using ':' as separator; default: not set |
| <code><remote></code> | <code>user</code> | string | name of the user on the remote host system |
| <code><remote></code> | <code>host</code> | string | name of the host which should be used for execution; multiple hostnames must be separated by ';' - in that case the maxRunning argument is applied on each host separately |
| <code><remote></code> | <code>privateKey</code> | string | path the to private ssh auth key; should be protected by a passphrase! |
| <code><remote></code> | <code>[port]</code> | integer | port which is used for the ssh connection; default: <code>22</code> |
| <code><remote></code> | <code>[disableStrictHostCheck]</code> | boolean | disables the validation of the public key of the host; not recommended!; default: <code>false</code> |
| <code><cluster></code> | <code>[customParameters]</code> | string | additional parameters that are directly passed to the DRMAA system without further processing; default: not set |
| <code><sge></code> | <code>[slots]</code> | integer | number of cores which are reserved on the computer cluster; default: <code>1</code> |
| <code><sge></code> | <code>[memory]</code> | string | memory per slot suffixed with M (megabyte) or G (gigabyte); default: <code>3000M</code> |
| <code><sge></code> | <code>[queue]</code> | string | queue on which the tasks should run on the computer cluster; default: not set |
| <code><sge></code> | <code>[disableDefault]</code> | boolean | default parameters are ignored; default: <code>false</code> |
| <code><sge></code> | <code>[customParameters]</code> | string | additional parameters that are directly passed to the SGE system without further processing; default: not set |
| <code><slurm></code> | <code>cluster</code> | string | cluster to communicate with; default: not set |
| <code><slurm></code> | <code>[cpu]</code> | integer | number of cores which are reserved on the computer cluster; default: <code>1</code> |
| <code><slurm></code> | <code>[memory]</code> | string | memory per slot suffixed with M (megabyte) or G (gigabyte); default: <code>3000M</code> |
| <code><slurm></code> | <code>[partition]</code> | string | partition of the cluster on which the job should be executed; default: not set |
| <code><slurm></code> | <code>[timelimit]</code> | string | maximum time the task will require to complete; default: 0-12:0 |
| <code><slurm></code> | <code>[disableDefault]</code> | boolean | default parameters are ignored; default: <code>false</code> |
| <code><slurm></code> | <code>[customParameters]</code> | string | additional parameters that are directly passed to the SLURM system without further processing; default: not set |

4.4 Global constants

A constant can be defined globally using the `<const>` elements which must be a child of a `<constants>` element. The parent element itself must be a child of the `<settings>` environment. Every `<const>` element must own a unique name which is set with the `name` attribute. The value of the constant is stored between the opening and closing element tag. `_${NAME_OF_CONSTANT}` is substituted with the corresponding constant in every attribute or text content. Only the `watchdogBase` attribute of `<watchdog>`, the `default` attribute of `<?Executor>` and the `<id>` attribute of `<?Task>` and within

<depends> elements can not be substituted.

Currently, there is one pre-defined constant named `${TMP}` which is substituted within **<?Task>** tags with the working directory of the executor that will execute the task.

Example 7: Definition and use of global constants

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <watchdog xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xsi:noNamespaceSchemaLocation="watchdog.xsd"
watchdogBase="{%INSTALL%}" isTemplate="true">
3
4      <settings>
5          <!-- definition of a constant named WAIT_TIME -->
6          <constants>
7              <const name="WAIT_TIME">30s</const>
8              <const name="FILE_NAME">sleep</const>
9              <const name="LOG_BASE">/tmp</const>
10             </constants>
11         </settings>
12
13         <tasks mail="{%MAIL%}">
14
15             <!-- definition a simple sleep task with constant
replacement -->
16             <sleepTask id="1" name="sleep test">
17                 <streams>
18
19 <stdout>${LOG_BASE}/${FILE_NAME}.out</stdout>
19                 </streams>
20                 <parameter>
21                     <wait>${WAIT_TIME}</wait>
22                 </parameter>
23             </sleepTask>
24         </tasks>
25     </watchdog>
```

In example 7 three constants are defined (6-10). The constant named `${WAIT_TIME}` is used as wait time in the sleep task (21). The other two constants are used to construct the standard output file path (18).

Table 5: Attributes in the context of global constants

| element | attribute | type | function |
|--------------------------|-------------------|--------|--|
| <constants> | | | parent of <const> elements and child of <settings> |
| <const> | <code>name</code> | string | name of the variable that is replaced with <code>\${name}</code> in attributes and text content; only chars out of [A-Za-z_] are allowed as first character followed by [A-Za-z_0-9] in the name; apart from a few exceptions it is allowed everywhere |
| <const> | | string | replacement value |

4.5 Environment variables

Some tools expect specific environment variables to be set correctly. For example the **PATH** variable is important because executable programs are located only in directories defined by that variable. The environment variables which are set on the host running Watchdog can be simply inherited. With help of the `<var>` element new variables can be defined or updated. The name of the variable must be defined with the `name` attribute while the value is stored between the opening and closing element tag. The parent element of each `<var>` element must be a `<environment>` element which also owns a `name` attribute. This `name` attribute is used to link the environment with a task using the `environment` attribute all tasks possess. It is also possible to define environment variables locally within task definitions. If local and global variables with the same name are set, the local ones override the global variables.

The following environment variables are set by Watchdog by default:

- **IS_WATCHDOG_JOB**: if module was executed by Watchdog this value is set to `1`
- **WATCHDOG_CORES**: number of reserved cores if task runs on a cluster environment
- **WATCHDOG_MEMORY**: number of total reserved memory in megabyte if task runs on a cluster environment

Example 8: Definition of environment variables

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <watchdog xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xsi:noNamespaceSchemaLocation="watchdog.xsd"
watchdogBase="{%INSTALL%}" isTemplate="true">
3
4      <settings>
5          <!-- definition of a environment -->
6          <environments>
7              <environment name="pathEnv">
8                  <var name="PATH"
update="true">~/software/bin</var>
9                  </environment>
10             </environments>
11         </settings>
12
13         <!-- begin task block and use that mail to inform the
user on success or failure -->
14         <tasks mail="{%MAIL%}">
15
16             <!-- definition of a simple sleep task using
custom environment variables -->
17             <envTask id="1" name="env" environment="pathEnv">
18                 <streams>
19                     <stdout>/tmp/env.test</stdout>
```

```

20         </streams>
21
22         <!-- definition of a local environment with
two variables -->
23         <environment>
24             <var name="SHELL">/bin/sh</var>
25             <var name="TEST" update="true"
sep="@">separator test</var>
26         </environment>
27     </envTask>
28 </tasks>
29 </watchdog>

```

In example 8 an environment named `pathEnv` is defined in which the variable `PATH` is updated (8). The entry `~/software/bin` is added at the beginning of the `PATH` variable and after the default separator character the previous value is kept. The `environment` attribute of the `env` task is set to the name of the previously defined environment (17). Additionally two local environment variables are defined (23-26). The first one replaces the default shell with `/bin/sh` while the second one updates a variable called `TEST` using an alternative separator.

Table 6: Attributes in the context of environment variables

| element | attribute | type | function |
|---------------|---------------------|---------|---|
| <environment> | name | string | is used as reference in the <code>environment</code> attribute of a task |
| <environment> | [copyLocalValue] | boolean | copies all environment variables which are set on the host running Watchdog; set variables are not deleted on the remote system; bash functions which names are ending with () are not copied as this might cause problems; default: <code>false</code> |
| <environment> | [useExternalExport] | boolean | uses a external command to set the variables; is necessary to update variables on remote or cluster executors and might also be necessary to set environment variables on remote hosts because of ssh security policies; default: <code>true</code> |
| <environment> | [exportCommand] | string | custom command to set a environment variable; <code>{ \$NAME }</code> and <code>{ \$VALUE }</code> are substituted and must be part of the command; default: <code>export { \$NAME }="{ \$VALUE }"</code> |
| <?Task> | environment | string | name of a <environment> element |
| <var> | | string | value of the environment variable |
| <var> | name | string | name of the environment variable |
| <var> | [update] | boolean | if <code>true</code> the value is added at the beginning of the variable and the old values comes afterwards separated with the value stored in the <code>sep</code> attribute; default: <code>false</code> |
| <var> | [sep] | string | separator which is used when the value of the variable should be updated; default: <code>:</code> |
| <var> | [copyLocalValue] | boolean | copies the environment variables with the name <code>name</code> which is set on the host running Watchdog; default: <code>false</code> |

4.6 Mail notification

By default Watchdog informs the user only when an error occurs during the execution of a task or if an error was detected afterwards. But these behaviour can be changed using the **notify** attribute of tasks.

Example 9: Different mail notification options

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <watchdog xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xsi:noNamespaceSchemaLocation="watchdog.xsd"
watchdogBase="{%INSTALL%}" isTemplate="true">
3
4      <settings>
5          <!-- definition of different process blocks -->
6          <processBlock>
7              <processSequence name="sleepTime" start="5"
end="15" step="5" />
8          </processBlock>
9      </settings>
10
11      <!-- begin task block and use that mail to inform the
user on success or failure -->
12      <tasks mail="{%MAIL%}">
13          <!-- definition a simple sleep task -->
14          <sleepTask id="1" name="sleep simple"
notify="enabled">
15              <parameter>
16                  <wait>10s</wait>
17              </parameter>
18          </sleepTask>
19
20          <!-- definition of process sequence sleep tasks --
>
21          <sleepTask id="2" name="sleep process sequence"
notify="subtask" processBlock="sleepTime">
22              <parameter>
23                  <wait>[]s</wait>
24              </parameter>
25          </sleepTask>
26      </tasks>
27  </watchdog>
```

In example 9 different notification options are presented. The first defined task is the simple sleep task from the previous examples for which the **notify** attribute is set to *enabled* (14). Once the task is finished a mail will be sent to the address the user specified in the **mail** attribute of the **<tasks>** element (12). The second defined task is based on a process block named *sleepTime* and causes Watchdog to inform the user as soon as a subtask is finished because the **notify** attribute is set to *subtask* (7, 21).

Table 7: Attributes in the context of mail notification

| element | attribute | type | function |
|---------|--------------|--------|---|
| <tasks> | mail | string | mail adress which is used for notification |
| <?Task> | notify | enum | enabled: inform when complete task was executed subtask: inform when a subtask was executed disabled: notification only in case of an error |
| <?Task> | processBlock | string | reference to a process block when <code>notify</code> is set to subtask |

4.7 Standard streams and working directory

By default the stdout and stderr stream of the tool which is executed by watchdog is not saved. This can be changed by setting a path via the <stdout> or <stderr> element. It is also possible to use a file as input via the <stdin> element. Additionally, a working directory can be set by using the <workingDir> element. When this is done also relative path for <stdout>, <stderr> and <stdin> are allowed. Other than usual, the elements must occur in the same order as they are listed in the following: <workingDir>, <stdout>, <stderr> and <stdin> (but each of them is optional)

Example 10: Definition of standard streams and working directory

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <watchdog xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xsi:noNamespaceSchemaLocation="watchdog.xsd"
watchdogBase="{%INSTALL%}" isTemplate="true">
3
4      <!-- begin task block and use that mail to inform the
user on success or failure -->
5          <tasks mail="{%MAIL%}">
6
7              <!-- definition a simple sleep task -->
8              <sleepTask id="1" name="sleep">
9                  <parameter>
10                     <wait>30s</wait>
11                 </parameter>
12                 <!-- definition of a standard output location
and switch of the working directory -->
13                 <streams>
14                     <workingDir>/tmp/</workingDir>
15                     <stdout>
16                     {%EXAMPLE_DATA%}/sleepTest.out</stdout>
17                     <stderr>
18                     append="true">sleepTest.err</stderr>
19                 </streams>
20             </sleepTask>
21         </tasks>
22     </watchdog>

```


In example 10 the working directory is set to `/tmp` (14). Afterwards, the standard output of the `sleep` task is written to the file `{%EXAMPLE_DATA%}/sleepTest.out` (15). The standard error stream is appended at the end of a file named `/tmp/sleepTest.err` (16).

Table 8: Attributes in the context of standard streams_and_working_directory

| element | attribute | type | function |
|--|--------------------------------------|---------|--|
| <code><streams></code> | | boolean | saves used resources to .res if standard output stream is saved to a file; default: <code>false</code> |
| <code><workingDir></code> | | string | sets a custom working directory before the tool is executed; default: <code>/usr/local/storage</code> |
| <code><stdout></code> | | string | writes standard output stream into file; default not saved |
| <code><stderr></code> | | string | writes standard error stream into file; default not saved |
| <code><stdin></code> | | string | file is used as standard input; default: not set |
| <code><stdin></code> | <code>[disableExistenceCheck]</code> | boolean | file must not exist when Watchdog is started; default: <code>false</code> |
| <code><stdout>/<stderr></code> | <code>[append]</code> | boolean | appends the stream at the end of the file; default: <code>false</code> |

4.8 Task actions

Additional operations can be executed before and after the actual task is executed. Currently a set of IO operations are implemented that can be used to roll out data the task depends on or to clean up once the task is finished. This feature is especially useful when Watchdog is running in slave mode (see [4.3](#)) in order to reduce load for the shared file system as some files might be requirements for different tasks. Hence, these files would have to be transferred multiple times from the shared file system to the host executing the task.

Moreover, files stored on remote files systems can be up- or downloaded by Watchdog. By default, virtual file systems based on the protocols File, HTTP, HTTPS, FTP, FTPS and SFTP as well as the main memory (RAM) are supported. These virtual file systems are provided by the Commons Virtual File System project of the Apache Software Foundation. Examples for valid URIs of these file systems can be found [here](#). However, any file system with an implementation of the `FileProvider` can also be included by the user as described in [6.1](#).

Task actions are defined in an `<actions>` tag as child of `<?Task>`. Slave mode is automatically activated if a task action is used. Currently six different IO operations are implemented:

- `<createFile>` - creates an empty file
- `<createFolder>` - create an empty folder
- `<copyFile>` - copies a file
- `<copyFolder>` - copies a folder (with content)
- `<deleteFile>` - deletes a file
- `<deleteFolder>` - deletes a folder (with content)

The event after which the task actions are executed must be defined using the `time` attribute each `<actions>` tag owns. The following arguments are available:

- **beforeTask** - before the task is executed
- **afterTask** - after the task is executed
- **onSuccess** - when the task was successfully executed
- **onFailure** - when task execution failed
- **beforeTerminate** - before Watchdog or a slave terminates itself

Example 11: Definition of task actions

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <watchdog xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xsi:noNamespaceSchemaLocation="watchdog.xsd"
watchdogBase="{%INSTALL%}" isTemplate="true">
3
4      <tasks mail="{%MAIL%}">
5          <gzipTask id="1" name="gzip task">
6              <parameter>
7                  <!-- path to a file that does not exist
yet -->
8
<input>/tmp/watchdog_file_to_compress.tmp</input>
9                  </parameter>
10                 <!-- action that copies a to the input
location -->
11                 <actions time="beforeTask">
12                     <copyFile file="
{%INSTALL%}examples/example_task_actions.xml"
destination="/tmp/watchdog_file_to_compress.tmp" override="true"
/>
13                 </actions>
14             </gzipTask>
15         </tasks>
16     </watchdog>

```

In example 11 a file name `/tmp/watchdog_file_to_compress.tmp` is compressed using gzip (5-14). Before the compress task is executed, the task action defined within the **<actions>** tag is executed because the **time** attribute is set to **beforeTask** (11-13). The task action copies the file stored in `{%INSTALL%}/examples/example_task_actions.xml` to `/tmp/watchdog_file_to_compress.tmp` (12).

Table 9: Attributes in the context of actions

| element | attribute | type | function |
|------------------------|-------------|------|--|
| <actions> | time | enum | defines when the task action block is executed; beforeTask: before the task is executed; afterTask: after the task is executed; onSuccess: when the task was successfully executed; onFailure: when task execution failed; beforeTerminate: before Watchdog or a slave terminates itself |

| element | attribute | type | function |
|----------------|------------------------|---------|---|
| <actions> | [uncoupleFromExecutor] | boolean | if enabled, task actions are executed on the host running Watchdog instead of the execution host; default: <i>false</i> |
| <createFile> | file | string | path to the file that should be created |
| <createFile> | [override] | boolean | defines if an existing file should be overwritten; default: <i>false</i> |
| <createFile> | [createParent] | boolean | defines if the parent directories should be created if nonexistent; default: <i>true</i> |
| <createFolder> | folder | string | path to the folder that should be created and will be empty if action succeeds |
| <createFolder> | [override] | boolean | defines if an existing folder should be deleted; default: <i>false</i> |
| <createFolder> | [createParent] | boolean | defines if the parent directories should be created if nonexistent; default: <i>true</i> |
| <copyFile> | file | string | path to the file that should be copied |
| <copyFile> | destination | string | path to the destination of the new file |
| <copyFile> | [override] | boolean | defines if an existing file should be overwritten; default: <i>false</i> |
| <copyFile> | [deleteSource] | boolean | deletes the source file after the copy operation; default: <i>false</i> |
| <copyFile> | [createParent] | boolean | defines if the parent directories should be created if nonexistent; default: <i>true</i> |
| <copyFolder> | folder | string | path to the folder that should be copied |
| <copyFolder> | destination | string | path to the destination folder |
| <copyFolder> | [pattern] | string | pattern selecting files that should be copied in that folder; syntax as in bash |
| <copyFolder> | [override] | boolean | defines if an existing folder should be deleted; default: <i>false</i> |
| <copyFolder> | [deleteSource] | boolean | deletes the source folder after the copy operation; default: <i>false</i> |
| <copyFolder> | [createParent] | boolean | defines if the parent directories should be created if nonexistent; default: <i>true</i> |
| <deleteFile> | file | string | path to the file that should be deleted |
| <deleteFolder> | folder | string | path to the folder that should be deleted |
| <deleteFolder> | [pattern] | string | pattern selecting files that should be deleted in that folder; syntax as in bash |

4.9 Simple calculations

Within a **<?Task>** element simple calculations can be preformed using the *\$(expr)* construct whereby *expr* must be a numerical equation. The following operators are supported: +, -, *, /, ^, ² and ³. Additionally the brackets *()* are provided. Moreover in case of a **<processSequence>** *i* is replaced by the current value of the process sequence. In the more general case of a **<processBlock>** *x* is substituted by an increasing number starting at *1*. The result of all calculations is rounded to five decimal places or converted to an integer if it is one.

Example 12: Definition of simple calculations

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <watchdog xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xsi:noNamespaceSchemaLocation="watchdog.xsd"
watchdogBase="{%INSTALL%}" isTemplate="true">
3
4      <settings>
```

```

5         <!-- definition of two process blocks -->
6         <processBlock>
7             <processSequence name="sleepTime" start="1"
end="5" step="1.5" />
8             <processFolder name="txtFiles" folder="
{%EXAMPLE_DATA%}/txt/" pattern="*.txt" />
9         </processBlock>
10    </settings>
11
12    <tasks mail="{%MAIL%}">
13        <!-- sleep task with a simple calculation -->
14        <sleepTask id="1" name="sleep"
processBlock="sleepTime">
15            <parameter>
16                <wait>$( (i+1)^2-1)s</wait>
17            </parameter>
18        </sleepTask>
19
20        <!-- compress txt files and write log files to
()/log/* -->
21        <gzipTask id="2" name="quality test"
processBlock="txtFiles">
22            <streams>
23                <stdout>()/log/$(x).out</stdout>
24            </streams>
25            <parameter>
26                <input>{}</input>
27                <output>{}.gz</output>
28                <quality>3</quality>
29            </parameter>
30        </gzipTask>
31    </tasks>
32 </watchdog>

```

In example 12 the usage of the `$(expr)` construct is shown. The wait time for the sleep task is calculated based on the input numbers of the `<processSequence>` (7, 16). In the second gzip task the number of the subtask is used to name the standard output files (23).

4.10 Multiple module search folders

By default Watchdog tries to locate modules in a folder named `modules/` stored in the installation directory of Watchdog. By using the `<modules>` element as child of `<settings>`, additional folders can be added.

Example 13: Definition of multiple module include folders

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <watchdog xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xsi:noNamespaceSchemaLocation="watchdog.xsd"
watchdogBase="{%INSTALL%}" isTemplate="true">
3
4      <settings>
5          <!-- TODO: modify one of these folders
{%INSTALL%}myCustomFolder/ or -->
6          <!-- /home/TODO/additionalModules/ to match a
folder that contains the 'sleep' module -->
7          <modules defaultFolder="myCustomFolder/">
8
9      </modules>
10     </settings>
11
12     <!-- begin task block and use that mail to inform the
user on success or failure -->
13     <tasks mail="{%MAIL%}">
14         <!-- definition a simple sleep task -->
15         <sleepTask id="1" name="sleep">
16             <parameter>
17                 <wait>30s</wait>
18             </parameter>
19         </sleepTask>
20     </tasks>
21 </watchdog>

```

In example 13 the default directory is changed to *myCustomFolder/* with the **defaultFolder** attribute (7). Moreover, in line 8 an additional folder is added to Watchdog's search path. Watchdog will now try to locate modules in the directories *{%INSTALL%}/myCustomFolder/* and */home/additionalModules/*. In order to test that example you must create a new folder, copy the *sleep* module from Watchdog's module folder and adapt the path in line 7 or 8 to match that folder.

Table 10: Attributes in the context of module include_folders

| element | attribute | type | function |
|-----------|-----------------|--------|--|
| <modules> | [defaultFolder] | string | changes the default search folder; an absolute or relative path to Watchdog's install dir is allowed (must end with /); default: <i>modules/</i> |
| <folder> | | string | adds a new directory to that is used for localization of modules |

4.11 Custom success and error checker

In some cases the exit code of a command is not a reliable indicator whether the command was executed successfully or not. For example some tools return as exit code zero regardless of whether the command succeeded or failed. Furthermore, a command could succeed technically but the desired result is not obtained (e.g. wrong index used for

mapping of RNA-seq data results in a very low mapping rate). In order to handle such cases the user has the option to implement custom success and error checkers in Java that are executed by Watchdog once a task has terminated. Two steps must be performed to use custom checkers: implementation in Java and invocation in the XML workflow.

Interfaces for checkers are stored in the package *de.lmu.ifi.bio.watchdog.interfaces*. Basically a function returning a boolean value that indicates whether the task succeeded or failed must be implemented. The constructor must accept as first argument a object of the type *Task* that contains information about the task that was finished. Additional arguments of type *Boolean*, *Integer*, *Double* or *String* can be passed via the XML definition.

Example 14: Load custom checkers

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <watchdog xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xsi:noNamespaceSchemaLocation="watchdog.xsd"
watchdogBase="{%INSTALL%}" isTemplate="true">
3
4      <!-- begin task block and use that mail to inform the
user on success or failure -->
5      <tasks mail="{%MAIL%}">
6
7          <!-- definition a simple sleep task -->
8          <sleepTask id="1" name="sleep">
9              <parameter>
10                 <wait>30s</wait>
11             </parameter>
12             <checkers>
13                 <!-- load a success checker with one
additional constructor argument -->
14                 <!-- it will check, if the file
{%INSTALL%}examples/mail_config exists and is not empty -->
15                 <checker classPath="
{%EXAMPLE_DATA%}/OutputFileExistsSuccessChecker.class"
className="de.lmu.ifi.bio.watchdog.successChecker.OutputFileExistsS
type="success">
16                     <cArg type="string">
{%INSTALL%}examples/mail_config</cArg>
17                 </checker>
18             </checkers>
19         </sleepTask>
20     </tasks>
21 </watchdog>
```

Example 14 shows an example of how an success checker can be added to a task by using the **<checkers>** element as a child of **<?Task>** (12-18). In addition to the location

of the compiled Java class and the full class name arguments can be passed to the constructor of the class. In this example one variable of type *string* is passed to the constructor of the success checker using the `<cArg>` element (16). Once the task is finished, the checkers are evaluated in the same order as they were added in the XML workflow. In cases in which simultaneously success and error were detected, the task will be treated as failed. In this example the success checker will ensure that the file `{%INSTALL%}/examples/mail_config exists` and is not empty (15-16)

Table 11: Attributes in the context of custom checkers

| element | attribute | type | function |
|------------------------------|------------------------|--------|--|
| <code><checker></code> | <code>type</code> | enum | type of the checker; success: checker should be used as success checker; error: checker is used as error checker |
| <code><checker></code> | <code>className</code> | string | complete class name including the package the class is located in |
| <code><checker></code> | <code>classPath</code> | string | absolute path to the compiled java class file |
| <code><cArg></code> | <code>type</code> | enum | type to which the argument should be parsed in java; possible values: boolean, integer, double and string |

5 Creating custom modules

In the following the steps that are needed to create a custom module named `nameOfModule` are explained. Basic XSD skills are needed to understand how things work together. Modules are defined in XSD format and should have the basic structure showed in example 15. To actually create modules the script `helper_scripts/createNewModule.sh` can be used and modified by hand as not all settings can be configured by it.

Example 15: Basic XSD structure

```
1  <?xml version="1.0" encoding="UTF-8" ?>
2  <x:schema xmlns:x="http://www.w3.org/2001/XMLSchema"
xmlns:vc="http://www.w3.org/2007/XMLSchema-versioning"
vc:minVersion="1.1" xmlns:xerces="http://xerces.apache.org">
3
4  ...
5
6  </x:schema>
```

As a first step, a folder with the name `nameOfModule` must be created in the `modules` folder. That folder must contain a file named `nameOfModule.xsd` which will hold the actual module definition.

5.1 Input parameter definition

Example 16 shows how input parameters and flags can be defined.

Example 16: Input parameter

```
1  <!-- definition of the task parameters -->
2  <x:complexType name="nameOfModuleTaskParameterType">
3      <x:all>
4          <x:element name="parameter1"
type="paramAbsolutePath" minOccurs="1" maxOccurs="1" />
5          <x:element name="parameter2" type="paramString"
minOccurs="0" maxOccurs="unbounded" />
6          <x:element name="flag1" type="paramBoolean"
minOccurs="0" maxOccurs="1" />
7      </x:all>
8  </x:complexType>
```

In line 4 and 5 two parameters are defined while line 6 creates a flag. Using the `minOccurs` and `maxOccurs` attributes it can be specified how often a parameter can be

used. Also parameters can have different types which are enforced during the validation of the XML workflows. Some pre-defined types are

- paramBoolean (for flags)
- paramString
- paramInteger
- paramDouble

These types accept by default booleans, strings, integers or doubles but also allow all values that are substituted by Watchdog. Moreover own parameter types can be defined as showed in example 17.

Example 17: Input parameter

```
1  <!-- module specific parameter types -->
2  <x:complexType name="paramWait_sleep">
3      <x:simpleContent>
4          <x:restriction base="paramString">
5              <x:assertion test="matches($value, '(${[A-Za-
z_]+})|($(.+))|([[(]($[A-Za-z_]+(,s*){0,1}){0,1}([0-9]+(,s*)
{0,1}){0,1}[)])])' ) or matches($value, '^([0-9]+[smhd]{0,1}$)' )"
xerces:message="Parameter with name '${tag}' must match [0-9]+
[smhd]{0,1}." />
6          </x:restriction>
7      </x:simpleContent>
8  </x:complexType>
```

5.2 Output parameter definition

Optionally a module can return output parameters that can be used for the following tasks as input. Return parameters must be written to a file in the format {%VAR_NAME%} TAB {%VALUE%}. The name of the file to write into is automatically sent to the module by Watchdog using the *returnFilePathParameter* parameter. If you want to change this default parameter name see [5.3](#).

Example 18: Output parameter

```
1  <!-- define output parameters which must be written to a
file -->
2  <x:complexType name="nameOfModuleTaskReturnType">
3      <x:complexContent>
4          <x:extension base="taskReturnType">
5              <x:all>
6                  <x:element name="outputParam1"
type="x:string" />
7              </x:all>
8          </x:extension>
```



```

9         </x:complexContent>
10    </x:complexType>

```

Line 6 in example 18 specifies that the module must return a parameter named *outputParam1* of type *x:string*. The module itself must ensure that the parameters are written physically before the module exits or otherwise Watchdog will terminate itself. In case of a bash script which is executed, two functions named *writeParam2File* and *blockUntilFileIsWritten* defined in *core_lib/functions.sh* can be used.

5.3 Binary call command and other settings

Now the command that will be executed can be defined. Example 19 specifies that a script named *nameOfModule.sh* that is stored in *modules/nameOfModule* will be called.

Example 19: Binary call command

```

1    <!-- set command and other settings -->
2    <x:complexType name="nameOfModuleTaskOverrideType">
3        <x:complexContent>
4            <x:restriction base="baseAttributeTaskType">
5                <x:attribute name="binName" type="x:string"
fixed="nameOfModule.sh" />
6            </x:restriction>
7        </x:complexContent>
8    </x:complexType>

```

In addition to that, the following settings can be modified:

- *binName*: name of the command which should be called
- *preBinCommand*: command that is added before the binName; e.g. interpreter
- *isWatchdogModule*: by default the command must be located in *modules/binName*; if this parameter is false the command must point to a absolute binary or be reachable via the PATH environment variable
- *returnFilePathParameter*: name of the parameter that is used to store the return values
- *watchdogModuleVersionParameter*: name of the parameter that passes the module version to the called command if a version different to *1* should be used; default: *moduleVersion* (using the default parameter passing options defined with the settings below; use an empty string to disable the passing of the module version)
- *paramFormat*: defines how names of parameters are prefixed; (do not print parameter name, - or --); default: *--*
- *spacingFormat*: defines how names of parameters and values are spaced; default: *blank*
- *quoteFormat*: defines how values are quoted; default: *single quoting*

- *separateFormat*: defines the separator string between multiple occurrences of the same parameter; default: ,
- *versionQueryParameter*: parameter of the *binName* command used to query software versions of third-party software; is called on the executor before task execution and stored together with the used parameters in a log file; default: not set

The last three arguments can also be used for each parameter separately.

5.4 Assign a name to the new module

Finally, a name must be assigned to the module. This can simply be done by creating a element with the attribute **name** of type *nameOfModuleType* and **substitutionGroup** set to *abstractTask*. Example 20 shows the needed line for the example module. Afterwards the type of the task is defined (5-15). If no output parameters are used line 10 can be omitted.

Example 20: Assign a name to the module

```

1  <!-- make task definition available via substitution group -
->
2  <x:element name="nameOfModuleTask" type="nameOfModuleType"
substitutionGroup="abstractTask" />
3
4  <!-- definition of final task -->
5  <x:complexType name="nameOfModuleTaskType">
6      <x:complexContent>
7          <x:extension base="nameOfModuleTaskOverrideType">
8              <x:all>
9                  <x:element name="parameter"
type="nameOfModuleTaskParameterType" minOccurs="1" maxOccurs="1"
/>
10                 <x:element name="return"
type="nameOfModuleTaskReturnType" minOccurs="0" maxOccurs="0" />
11                 <x:group ref="defaultTaskElements" />
12             </x:all>
13         </x:extension>
14     </x:complexContent>
15 </x:complexType>

```

5.5 Putting it all together

Example 21 shows the definition of the complete module.

Example 21: Putting it all together

```

1  <?xml version="1.0" encoding="UTF-8" ?>
2  <x:schema xmlns:x="http://www.w3.org/2001/XMLSchema"

```

```

xmlns:vc="http://www.w3.org/2007/XMLSchema-versioning"
vc:minVersion="1.1" xmlns:xerces="http://xerces.apache.org">
3
4     <!-- definition of the task parameters -->
5     <x:complexType name="nameOfModuleTaskParameterType">
6         <x:all>
7             <x:element name="parameter1"
type="paramAbsolutePath" minOccurs="1" maxOccurs="1" />
8             <x:element name="parameter2"
type="paramString" minOccurs="0" maxOccurs="unbounded" />
9             <x:element name="flag1" type="paramBoolean"
minOccurs="0" maxOccurs="1" />
10        </x:all>
11    </x:complexType>
12
13    <!-- define output parameters which must be written to
a file -->
14    <x:complexType name="nameOfModuleTaskReturnType">
15        <x:complexContent>
16            <x:extension base="taskReturnType">
17                <x:all>
18                    <x:element name="outputParam1"
type="x:string" />
19                </x:all>
20            </x:extension>
21        </x:complexContent>
22    </x:complexType>
23
24    <!-- set command and other settings -->
25    <x:complexType name="nameOfModuleTaskOverrideType">
26        <x:complexContent>
27            <x:restriction base="baseAttributeTaskType">
28                <x:attribute name="binName"
type="x:string" fixed="nameOfModule.sh" />
29            </x:restriction>
30        </x:complexContent>
31    </x:complexType>
32
33    <!-- make task definition availible via substitution
group -->
34    <x:element name="nameOfModuleTask"
type="nameOfModuleTaskType" substitutionGroup="abstractTask" />
35
36    <!-- definition of final task -->
37    <x:complexType name="nameOfModuleTaskType">
38        <x:complexContent>
39            <x:extension
base="nameOfModuleTaskOverrideType">
40                <x:all>
41                    <x:element name="parameter"
type="nameOfModuleTaskParameterType" minOccurs="1" maxOccurs="1"
/>

```

```

42             <x:element name="return"
type="nameOfModuleTaskReturnType" minOccurs="0" maxOccurs="0" />
43             <x:group ref="defaultTaskElements"
/>
44             </x:all>
45         </x:extension>
46     </x:complexContent>
47 </x:complexType>
48
49 </x:schema>

```

5.6 Module versions

As parameters of the command that is called by the module might change over time, it is possible to define different versions of the same module. The version that should be used during XML validation and workflow execution can be set using the **version** attribute each **<?Task>** tag owns. By default version **1** of each module is used. Be aware that only one version of the same module can be used within a workflow.

In order to avoid duplication of the complete XSD file, the minimum and maximum supported module version can be defined for each tag within the XSD file using the **minVersion** and **maxVersion** attributes. If a tag is not part of a particular version, it is removed with all its childs from the XSD file that is generated dynamically at runtime for that module version. By using these two attributes the input parameter, the return parameter, the called command and more can be changed for different module version. Example 22 shows a part of the *featureCounts* module. Among other things a parameter needs to be renamed to support the most recent binary of *featureCounts* (11-12).

Example 22: Rename of parameters

```

1  <?xml version="1.0" encoding="UTF-8" ?>
2  <x:schema xmlns:x="http://www.w3.org/2001/XMLSchema"
xmlns:vc="http://www.w3.org/2007/XMLSchema-versioning"
vc:minVersion="1.1" xmlns:xerces="http://xerces.apache.org">
3
4      <!-- definition of the task parameters -->
5      <x:complexType name="featureCountsTaskParameterType">
6          <x:all>
7              <!-- attributes common to all versions -->
8              ...
9
10             <!-- attributes that differ between versions
-->
11             <x:element name="minOverlap"
type="paramInteger" minOccurs="0" maxOccurs="1" maxVersion="1" />
12             <x:element name="minReadOverlap"
type="paramInteger" minOccurs="0" maxOccurs="1" minVersion="2" />

```

```

13         ...
14         </x:all>
15     </x:complexType>
16     ...
17 </x:schema>

```

Example 23 shows how a module script can be made version dependent using bash functions that are delivered with Watchdog. First, the bash variable `${MODULE_VERSION_PARAMETER_NAME}` is set to the complete name of the parameter that is used to pass the module version (e.g. `--moduleVersion`) (3). After sourcing of `includeBasics.sh` the module version is automatically stored in the `${MODULE_VERSION}` variable. This information can be used to alter the parameters and the behavior of the script as an alternative to duplication if differences between module versions are only minor (15,18).

Example 23: Version dependent bash script

```

1  #!/bin/bash
2  SCRIPT_FOLDER=$( cd "$( dirname "${BASH_SOURCE[0]}" )" &&
pwd )
3  MODULE_VERSION_PARAMETER_NAME="--moduleVersion"
4  source $SCRIPT_FOLDER/../core_lib/includeBasics.sh $@
5  ...
6
7  # define parameters
8  # params used in any version of the module
9  DEFINE_string 'annotation' '' 'feature annotation in GTF or
SAF format' 'a'
10 DEFINE_string 'input' '' 'index bam file which should be
used for counting' 'i'
11 DEFINE_string 'output' '' 'path to output file' 'o'
12 ...
13
14 # params only available in module version 1
15 if [ ${MODULE_VERSION} -eq 1 ]; then
16     DEFINE_string 'minOverlap' '1' '[optional] minimum
number of overlapped bases required to assign a read to a
feature; also negative values are allowed' 'm'
17 # params only available in module version 2
18 elif [ ${MODULE_VERSION} -eq 2 ]; then
19     DEFINE_string 'minReadOverlap' '1' '[optional] minimum
number of overlapped bases required to assign a read to a
feature; also negative values are allowed' 'm'
20     ...
21 fi
22 DEFINE_integer 'moduleVersion' '1' '[optional] version of
the module that should be used' ''
23 DEFINE_boolean 'debug' 'false' '[optional] prints out debug

```

```
messages.' ''
24     ...
```

5.7 Other matters

Exit codes: The module developer must ensure that a command does only exit with exit status 0 if the command was executed successfully. File `core_lib/exitCodes.sh` contains some exit codes which names are also included in mail notifications if they are used. Custom exit codes can be easily added.

Error messages: Watchdog can detect by default error messages in standard out and standard error streams if they begin with `[ERROR]`. The errors are only stored if standard out and error files are saved to disk using the `<streams>` tag. If an error was detected but the exit code was 0 the command will also fail.

Module test: A script named `test_nameOfModule.sh` can also be part of the module. It is automatically called, if the user calls `helper_scripts/moduleTest.sh`. Also the module folder might contain some test data in the folder `test_data`. For simple test cases the bash function `testExitCode` can be used to test, if an input leads to the expected output.

6 Extend Watchdog's functionality

In the following sections two different ways to extend Watchdog's functionality are described.

- *Virtual File Systems* that can be used within task actions (see [6.1](#))
- *XML Plugins* that add new `<?Executor>` and `<?ProcessBlock>` elements (see [6.2](#))

6.1 Virtual file systems for task actions

With the help of task actions, file system operations can be performed before and after tasks (see [4.8](#)). By default, virtual file systems based on the protocols File, HTTP, HTTPS, FTP, FTPS and SFTP as well as the main memory (RAM) are supported. These virtual file systems are provided by the Commons Virtual File System project of the Apache Software Foundation.

In order to add a new virtual file system, a class that implements the *VFSRegister* interface can be added to the jar-file. The class will be automatically loaded by Watchdog and the new virtual file system will be useable without other modifications. The following four methods must be implemented for the interface:

- *getFileProvider* - must return an instance of the *FileProvider* interface as defined in the Commons Virtual File System project
- *getURLSchemes* - returns the url schemes that should be used in combination with that *FileProvider* (e.g. *ftp*)
- *getMimeTypes* - sets schemes that should be used for specific mimetypes
- *getExtensions* - sets schemes that should be used for specific file extension

The class *SimpleVFSRegister* can be extended if an instance of the *FileProvider* class can be created without arguments. Then only the name of the *FileProvider* class and the URL schemes that should be used must be defined. Example 24 shows how the virtual FTP file system is integrated in Watchdog by using the *FtpsFileProvider* class of the Commons Virtual File System project.

Example 24: Simple implementation of the VFSRegister interface

```
1 package de.lmu.ifi.bio.watchdog.task.actions.vfs.impl;
2
3 public class FTPSVFSRegister extends SimpleVFSRegister {
4
5     private static final String CLASS_NAME =
"org.apache.commons.vfs2.provider.ftp.FtpsFileProvider";
6     private static final String[] SCHEME = new String[]
{"ftp"};
7
8     public FTPSVFSRegister() throws Exception {
```



```
9         super(CLASS_NAME, SCHEME);
10     }
11 }
```

6.2 XML Plugins

Watchdog provides a flexible plugin system that allows extending Watchdog by additional types of executors and process blocks without modifying the original Java classes. Essentially, this means creating a new XML element for use in Watchdog workflows as well as implementing additional Java classes that provide the functionality for this element. In brief, you have to do the following to use the plugin system:

- create an XSD file describing the new element and its parent element for use in Watchdog workflows
- Extend a few abstract classes
- Add class files for the new classes to the Watchdog jar-file and copy the new XSD file to a sub-directory of the Watchdog installation directory

In the Watchdog command-line version, all non-abstract classes in the Watchdog jar-file that extend the *XMLParserPlugin* abstract class are loaded dynamically during workflow execution. Currently, this is restricted to XML parsers for the generic type *ProcessBlock* or *ExecutorInfo*. The XML parser for a new XML element provides the functionality to parse this element in a workflow (i.e. a new executor or process block type) and to create a new object representing the corresponding element type. Here, the four most important functions of the *XMLParserPlugin* abstract class that have to be implemented are:

- *getNameOfParseableTag*: returns the name of the element the class can parse
- *getNameOfParentTag*: returns the name of the parent element of this element
- *getXSDDefinition*: returns the path to the XSD file describing this element (relative to the *xsd* sub-directory of the Watchdog directory)
- *parseElement*: implements the actual parsing process.

The last function creates an object of a class representing the new element. This class has to implement the interfaces *XMLDataStore* and *XMLPlugin*, for instance by extending one of the abstract classes *ProcessBlock* or *ExecutorInfo* or one of their subclasses.

For use in the Workflow designer GUI of Watchdog, two additional requirements have to be met:

- An FXML file has to be provided describing how the attributes of the new element type are represented graphically. FXML is an XML-based markup language for describing the layout of a user interface in a JavaFX application.
- Classes extending *PluginView* and *PluginViewController* have to be implemented for testing whether the input is valid and for loading and saving data to and from XML.

All executors and process blocks integrated in Watchdog are using this plugin system. Hence, examples how to implement a new XML element can be found in the package *de.lmu.ifi.bio.watchdog.xmlParser.plugins* of the [Java source code](#).

7 Docker

In order to run a Docker image, Docker must be installed and configured correctly as described [here](#).

7.1 Install the Watchdog Docker image

A Watchdog image for Docker can be obtained from hub.docker.com. The image is rebuilt automatically by the Bioconda project once a new version is released on [Bioconda](#).

You can download the latest version of the image with `docker pull klugem/watchdog-wms`. Within the Docker image the environment variable `WATCHDOG_HOME` is set automatically to the installation directory of Watchdog (required for the `watchdogBase` attribute). The `-useEnvBase` flag of the command line version can be used to override the `watchdogBase` attribute of the XML workflow with the value stored in `WATCHDOG_HOME`. Moreover, the installation directory of Watchdog is mounted under `/watchdog` within the Docker image.

7.2 Sharing of files

In order to exchange files with the host system, the `-v` or `-mount` option of Docker can be used. These option can be used multiple times.

```
docker run -v source_folder_or_file_on_host:destination_folder_or_file[:ro] image command
```

More information can be found in the [documentation](#) of Docker.

7.3 Port forwarding

In order to use the build-in webserver of Watchdog, the port used by the webserver must be forwarded to the host running the Docker container.

The command `docker run -p 8090:8080 image command` maps the container port 8080 (TCP) to the port 8090 (TCP) on the Docker host. More information can be found in the [documentation](#) of Docker.

7.4 How to use the Docker Watchdog image

The examples within the Docker image are automatically configured when `{%Nwatchdog-cmd%N}` is started the first time and are stored in `/watchdog/examples`. The command

```
docker run -h localhost -p 8080:8080 klugem/watchdog-wms watchdog-cmd -useEnvBase -x /watchdog/examples/example_basic_sleep.xml
```

executes the example described in [3.2](#) and forwards the webserver port to the host port 8080.

Alternatively, it is possible to run a workflow that is stored on the host system as described in [7.2](#). Ensure that all files used in the workflow are made accessible within the Docker image.

7.5 Use Docker in modules

A Docker image can also be used in a module. The module *bowtie2Docker* implements an example module that uses the Docker image of [Bowtie 2](#) that is provided by [Bioconda](#) and hosted on [quay.io](#). The Docker image will be automatically downloaded if it is not found locally.

Make sure that the Docker daemon is installed and running before you test this example.

Example 25: Example usage of the Bowtie 2 Docker module

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <watchdog xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xsi:noNamespaceSchemaLocation="watchdog.xsd"
watchdogBase="{%INSTALL%}" isTemplate="true">
3
4      <settings>
5          <constants>
6              <const name="BASE">
{%INSTALL%}/modules/bowtie2Docker/example_data</const>
7              </constants>
8          </settings>
9
10         <tasks mail="{%MAIL%}">
11             <bowtie2DockerTask id="1"
name="bowtie2_in_docker">
12                 <streams>
13
14 <stdout>/tmp/bowtie2.docker.test.out</stdout>
15 <stderr>/tmp/bowtie2.docker.test.err</stderr>
16                 </streams>
17                 <parameter>
18 <genome>${BASE}/index/lambda_virus</genome>
19                 <reads>${BASE}/reads/reads_1.fq</reads>
20                 <reads>${BASE}/reads/reads_1.fq</reads>
21 <outfile>/tmp/bowtie2.docker.test.sam</outfile>
22                 </parameter>
23             </bowtie2DockerTask>
```

```
23         </tasks>
24     </watchdog>
```

Example 25 shows how the *bowtie2Docker* module can be used with the provided example data. The test data that is shipped with Bowtie 2 is stored in the folder *example_data* of the module (6). Log files are written to */tmp/bowtie2.docker.test.[out|err]* (13, 14) while the mapped reads are stored in SAM format in */tmp/bowtie2.docker.test.sam* (20).