Project: Open Microscopy Environment
Authors: Jean-Marie Burel, Andrea Falconi
Version: Final
Date: 2007-02-15

Software Build and Deployment Document

Table Of Contents

1. Introduction

This section provides some preliminary information and an overview of the document.

1.1. Purpose

This document describes how to build and deploy the OMERO.insight client software. OMERO.insight is part of the Open Microscopy Environment (OME) system and is a client that focuses on delivering a software for the visualization and manipulation of both image data and metadata maintained at an OMERO server site.

All the information that is essential to the build and deployment process is unfolded: how to build and verify the various deliverables, library dependencies, how to install and run the software, and the system requirements. In addition to that, we collected some other useful information related to the build process such as a description of the project repository and the design of the build system.

1.2. Readership

The primary audience of this document is system administrators and power-users that wish to build the software (or any other available project artifact, like documentation) from scratch. We assume the reader has some familiarity with the Unix/Windows shells and a minimum knowledge of the Java Runtime Environment.

This document is also very useful as a guide to new developers and is a handy reference for all other OMERO.insight developers. Moreover, the section on the design of the build system contains important information for the development and maintenance of the build system itself. This is more technical information and is only targeted to developers.

1.3. Document Overview

The rest of this document is organized in the following sections:

- Project Repository: Contains useful information on how the repository is organized and how to download it from our Subversion server.
- Build Process: Details how to build the client software from the source code and how to build the other project artifacts, like documentation. Additional information on the various steps of the build process is also provided as well as a list of all required libraries.
- Deployment: How to install and run the software, and the system requirements.
- Build System Design: This section is only relevant to the developers of the build system as it summarizes the design and describes the various components.
- Future Directions: What is coming next in the build system. Again, this section is mainly relevant to developers.
- Legal: Licenses and copyright information.

2. Project Repository

The OMERO.insight source code (code name *Shoola Java Client*) and related artifacts are hosted at our Subversion server and can be found at http://trac.openmicroscopy.org.uk/shoola/browser. The software configuration management for the whole OMERO project relies on t his server; OMERO.insight is a sub-project of the OMERO project For more information on OMERO, see http://trac.openmicroscopy.org.uk/shoola/omero. The OMERO.insight root module is called *Shoola*, everything else is contained within this directory.

The repositories are also browsable online via ViewCVS.

The rest of this section describes the organization of the source code and related artifacts as well as how to retrieve them.

2.1. Repository Organization

The *shoola-omero* Subversion module is the repository of the software artifacts of the OMERO.insight project. Its contents are as follows:

- *build*: This directory contains the tools to compile, run, test, and deliver the application.
- *config*: Various configuration files required by the application to run.
- *docgen*: Documentation artifacts that are used to build actual documents. These are organized in two sub-directories: *javadoc* and *xdocs*. The former only contains resources (like CSS files) to generate programmer's documentation (JavaDoc) -- the actual documentation contents are obtained from the source code. The latter contains both resources (like stylesheets and Java code) to generate all other kinds of documentation (like design and users documents) and the actual documentation contents in the form of XML/HTML files.
- *launch*: Launcher scripts and installation instructions bundled with the default application distribution file. Its sub-directories contain further resources to build platform-specific distributions.
- *legal*: Terms and conditions under which the various artifacts in the shoola-omero SVN module are released. A README file specifies which license applies to which artifact.
- *LIB*: All library files required by the application. Its *test* sub-directory contains additional libraries required by the test code.
- *SRC*: Contains the application source files.
- *TEST*: The test code.
- *CodeTemplate*: This file describes the coding style that all source files should conform to.
- *README*: old trustworthy README file.

2.2. Downloading from Subversion

The whole OMERO.insight source code (code name *Shoola*) and other related artifacts can be freely downloaded (refer to Legal for licensing issues) from our Subversion repository. To do so, you will need a SVN client. If you don't have it, you can get one from http://subversion.tigris.org/. Before you start downloading from Subversion, make a new directory and move there. Thus to checkout the latest source for shoola-omera the module, type:

```
svn co http://cvs.openmicroscopy.org.uk/home/svn/shoola-omero/trunk
```

or (for a specific tagged version)

```
svn co http://cvs.openmicroscopy.org.uk/home/svn/shoola-omero/tags/OMERO.insight_3_Beta1
```

3. Build Process

The build process is designed to be cross-platform and self-contained. What all this means is that you can run the various build tasks (which we'll refer to as *targets* in the remainder of this document) on any machine where a Java Development Kit (JDK version 1.5 or later) is available, without having to worry about resolving library dependencies. In fact, all the libraries required by the application and test code as well as those needed for the build itself come in the SVN module and are automatically linked as needed.

A convenience tool is povided to run any of the available build targets. However, you can run the build with Ant if you prefer -- in this case some configuration is required, as detailed in a later section. The remainder of this section details the various build targets and provides useful information related to the build process.

3.1. Build Tool

The build tool is a convenience tool that we povide to run any of the available build targets. This tool operates from the command line, so you'll need to be a little familiar with your operating system shell in order to use it. Move to the *build* directory and from there type, at the command prompt:

```
java build -p
```

This will output a list of all available targets along with a short description of what each target does. For a more detailed description simply type:

```
java build
```

To run a target, just specify the name of the target after `build` on the command line. For example:

```
java build compile
```

would compile the application sources. If you're an experienced Java developer, you might have already guessed that we're using Ant under the hood. In fact, the build tool simply forwards to Ant whatever follows `build` on the command line. So, in general, you can use the build tool as follows:

```
java build [options] [target [target2 [target3] ...]]
```

where `options` are any of the Ant options. For example:

```
java build -e compile
```

would output logging information on the console without adornments. (You can get the whole list of Ant options by using the `-h` switch.)

Note for developers

The build tool is just a convenience to save you the hassle of configuring Ant manually. If you prefer to use Ant directly, there's some tweaking required first. This is described in a later section.

3.2. Build Targets

The targets available to the build tool can be grouped into the following categories: application, test, documentation, distribution, and shortcuts. All targets output under the *Shoola/OUT* directory. (*Shoola* is the directory where you downloaded the SVN tree.) Unless otherwise specified, all directory pathnames in the following description are given relative to the *Shoola/OUT* directory.

Application Targets

Use the targets in this category to compile and run the application. These targets output under the *app* directory. Follows their description:

- `compile`: Compiles all the application sources and links all the application resources. This target processes the source files that are not up-to-date with respect to the corresponding compiled files. So every subsequent invocation of this target will only compile the source files that have changed since the last invocation. The same policy applies to resource files.
- `run`: Runs the application. This target will automatically execute `compile`, so you won't have to invoke it yourself before running the application.
- `app.clean`: Deletes the *app* directory and all its contents.

Test Targets

Use the targets in this category to test the application and generate test reports. These targets output under the *test* directory. Follows their description:

- `test`: Runs all the tests, outputting a brief report for each test on the console and a detailed report under *test/reports/html*. This target will automatically execute `compile`, so you won't have to invoke it yourself before running the tests.
- `test.clean`: Deletes the *test* directory and all its contents.

Documentation Targets

Use the targets in this category to generate the project documentation. These targets output under the *docs* directory. Follows their description:

- `docs`: Outputs all available project documentation. This includes both technical documents (like this document, architecture/design documents, JavaDoc, etc.) and users documentation.
- `docs.clean`: Deletes the *docs* directory and all its contents.

Distribution Targets

Use the targets in this category to create the deliverables. These targets output under the *dist* directory. Follows their description:

- `dist`: Creates the default distribution bundle. This is a zip file whose name indicates the version number. It contains, besides the application file itself, the configuration files, all the required libraries, various launch scripts for the supported platforms, install instructions, and the license file. This target will first clean the application and test directories and then perform a fresh `compile`-and-`test` before packaging to make sure the application is good for delivery. The application *jar* file (*omeroinsight.jar*) produced during this process is left under the *dist* directory.
- `dist-osx`: Creates the Mac OS X distribution bundle. This is a zip file whose name indicates the version number and ends with `-OSX`. It contains, besides the application file itself, the configuration files, all the required libraries, Mac OS X application stubs, install instructions, and the license file. This target will first clean the application and test directories and then perform a fresh `compile`-and-`test` before packaging to make sure the application is good for delivery. The application *jar* file (*omeroinsight.jar*) produced during this process is left under the *dist* directory.
- `dist-docs`: Creates the documentation bundle. This target first generates all of the project documentation by invoking `docs`. It then zips everything in a single file whose name indicates the version number and ends with `-doc`.
- `dist.clean`: Deletes the *dist* directory and all its contents.

Shortcuts

The targets in this category provide shortcuts to other targets. Follows their description:

- `clean`: Removes all the output generated by the last build.
- `all`: An alias for `clean, dist`.

Note for developers

Some internal targets not mentioned above could be useful during development. For example, you can run only some given tests or generate only a subset of the project documentation. Also, it is possible to easily tweak existing targets by simply overriding their properties from the command line. Refer to the various child build files for details. Finally note that if you need to experiment with new libraries, just drop them into the *LIB* or *LIB/test* directory (depending on whether the application or test code depends on that library) and the build system will pick them up automatically.

3.3. Dependencies

OMERO.insight is entirely written in Java and you will need a Java Runtime Environment (version 1.5 or later) to run it. All the third-party libraries required by OMERO.insight come already in each distribution bundle and are automatically linked at runtime, so you won't have to worry about installing/upgrading libraries.

For the sake of record, here's a brief description of the third-party libraries used by OMERO.insight. First off, all of those libraries are stored in the `LIB` directory under the CVS root. Its `test` sub-directory contains additional libraries required by the test code -- these are not obviously included in the distribution bundles. The libraries required by the OMERO.insight source code are:

- `log4j-1.2.8.jar`: The Log4j logging framework (http://logging.apache.org/log4j/). Version 1.2.8. This library provides logging facilities to OMERO.insight.
- `commons-httpclient-3.0.1.jar`: The Jakarta Commons HttpClient (http://jakarta.apache.org/commons/httpclient/).
- `commons-logging-1.0.4.jar`: The Jakarta Commons Logging (http://jakarta.apache.org/commons/logging/). Version 1.0. This library is not directly required by OMERO.insight, but by the `commons-httpclient-3.0.1.jar`.
- `client-3.0-TRUNK.jar:`: This library is part of the OMERO distributions and provides extensions of JBoss and Spring classes for login and configuration. It provides simple wrappers of some common OMERO types.
- `common-3.0-TRUNK.jar:`: This library is part of the OMERO distributions and contains all model objects, APIs as well as all parameters which pass through API calls.
- JBoss (http://labs.jboss.com/portal/) client libraries, see list below, provide JBoss-specific JNDI and RMI stacks.

    - `jboss-annotations-ejb3-4.0.4.GA.jar`.
    - `jboss-aop-jdk50-client-4.0.4.GA.jar`.
    - `jboss-aspect-library-jdk50-4.0.4.GA.jar`.
    - `jboss-ejb3-4.0.4.GA.jar`.
    - `jboss-ejb3x-4.0.4.GA.jar`.
    - `jbossall-client-4.0.4.GA.jar`.

- `spring-2.0-m3.jar:`: The Spring framework (http://www.springframework.org/). Version 2.0-m3: This library provides configuration for OMERO.insight.
- `TableLayout.jar:`: This library provides a layout manager (https://tablelayout.dev.java.net/).

The additional libraries required by the test code are:

- `junit-3.8.1.jar`: The JUnit Testing Framework (http://www.junit.org/). Version 3.8.1.

3.4. Building with Ant

The build system is completely Ant-based. The build tool that we've been describing has got nothing to do with the actual build files, which are Ant build files. It's just a convenience tool that does all of the Ant's configuration for you. In fact, the various build files invoke optional tasks (please refer to the Ant manual for an explanation of optional tasks) which require additional libraries that don't ship with Ant. Because there might be times when you want to run a build directly with Ant, we now explain how to do it.

First off, you will need Ant 1.6 or later on your machine. (You can download the binary distribution from http://ant.apache.org/bindownload.cgi.) Then all the following libraries:

- The JUnit Testing Framework (http://www.junit.org/). Version 3.8.1.
- The Xalan-J XSLT library from Apache (http://xml.apache.org/xalan-j/). Version 2_6_0.
- The Jar Bundler utility to make Mac OS X application bundles (http://www.loomcom.com/jarbundler/). Version 1.4.
- The Bean Scripting Framework from Jakarta (http://jakarta.apache.org/bsf/). Version 2.3.0-rc1.
- The Rhino JavaScript engine from Mozilla (http://www.mozilla.org/rhino/). Version 1_5R3.
- The Formatting Objects Processor (FOP) from the Apache XML Project (http://xml.apache.org/fop/). Version 0.20.5.
- The Apache Avalon Framework, bundled with the FOP distribution. Version: unknown.
- The Apache Batik SVG Toolkit, bundled with the FOP distribution. Version: unknown.

All these libraries can be found in the *build/tools* directory. It's important that you use exactly the distribution versions listed above. In fact, some later versions may be incompatible with each other -- for example, BSF 2.3.0-rc1 (the latest version at the time of writing) won't work with Rhino versions greater than 1_5 and you might experience similar problems with FOP/Batik.

The next step is to make these libraries available to Ant. There are several ways to link libraries to the Ant's runtime, ranging from the `-lib` switch on the Ant's command line to placing the jars under *{ANT_HOME}/lib* or, even better, under *{YOUR_HOME}/.ant/lib*. Please refer to the Ant manual for details. Finally, you should tweak the runtime properties of each library where required. The documentation released with each library distribution usually contains a section devoted to how to link the library to the Ant's runtime. Please read those documents carefully.

At this point you should be set. Move to the *build* directory and type `ant` at your shell prompt. This will give you the list of available targets, which have already been described in a previous section.

Important

If you bypass the build tool, then you're completely responsible for Ant's configuration. In particular when new tasks will be added (and new libraries brought in), you may have to reconfigure Ant. The *build/lib.xml* header always contains the most up-to-date list of required libraries.

3.5. Versioning

Pending : enter contents.

4. Deployment

OMERO is a client/server system. The server software stores, manages, and analyzes image data and metadata. A server site encompasses data and image servers and is deployed on a powerful server machine. The OMERO servers expose network interfaces so that client software can connect to them and perform tasks on behalf of users. OMERO.insight is one of such clients. Client software is (usually) deployed on desktop machines or workstations.

Installing and running OMERO.insight is very easy: drop the distribution directory somewhere on your filesystem and run! This section details all this more precisely. An obvious pre-requisite is that the OMERO server site you intend to connect to is up and running and you have a valid account.

4.1. System Requirements

In order to run OMERO.insight, your system must have:

- A processor of 1GHz or faster.
- A minimum of 256MB of RAM. (We strongly recommend 512MB.)
- A network link of 100Mb (possibly dedicated) between your machine and the OMERO server site you intend to connect to.
- A Java Runtime Environment, 1.5 or later.

Note

In principle, the software should run on *any* Java Runtime Environment -- 1.5 or later. However, we've only been testing the system extensively on Linux/Blackdown JVM, Mac OS X/Apple JVM, Windows/Sun JVM -- all of which are 1.5 JVMs.

4.2. Installing and Running

Installing and running OMERO.insight is pretty easy. The following paragraphs detail the procedure to follow if you're using the build system or have downloaded/generated the generic or Mac OS X distribution bundle.

Generic Bundle

If you want to install the generic distribution bundle, just extract the archive contents into the directory in which you intend to install. To run, just launch the start-up script for your platform -- the installation directory contains scripts for Mac OS X, Unix, and Windows. A splash screen will pop up, enter a new server address or select a previously entered one, your user name and password to log onto the selected OMERO server site.

Note: Linux/Mac OS X/Unix

We experienced problems with some GUI tools that deflate zip archives on Unix-like platforms. The original file permissions were ignored/overwritten during the archive extraction phase. (Indeed, there's no portable way to store these permissions. We use the Ant zip task which in turn uses the algorithm used by the default versions of zip and unzip for many Unix-like systems.) This resulted in having to set the executable bit for some files manually after the extraction of the archive contents. If you're experiencing similar problems, then proceed as follows after unzipping. Move to your installation directory and from there type, at the command prompt: `chmod +x *.sh`

Mac OS X Bundle

If you want to install the Mac OS X distribution bundle, just extract the archive contents into the directory in which you intend to install. To run, just double-click on the OMERO.insight icon in your installation directory. A splash screen will pop up, enter a new server address or select a previously entered one, your user name and password to log onto the selected OMERO server site.

Note for Linux/Mac OS X/Unix

As already mentioned, we experienced problems with some GUI tools that deflate zip archives. If you double-click on the OMERO.insight icon and the application doesn't start, then probably file permissions haven't been set properly during the archive extraction phase. The fix is easy: open up a terminal and `cd` to your installation directory. From there, again `cd` to *OMERO.insight.app/Contents/MacOS*. Now type: `chmod +x JavaApplicationStub`

5. Build System Design

The build system is Ant-based and is composed of several build files. We use some new Ant features that support complex build systems, so Ant 1.6 or later is required. This section focuses on the breakdown of the build system into build files and the dependencies among targets. Because the various build files already contain detailed information, here we just provide the extra information that you will need to understand how all the parts fit together. We also describe the build tool and finally provide a rationale for the design.

5.1. Build Files

The various build targets have been grouped in several categories -- this has already been discussed in a previous section. All the targets within a category are in the same build file which is a child to a master build file that is used as an entry point to the build. All these build files are kept under the *build* directory:

- *app.xml*: Contains the application targets to compile and run OMERO.insight.
- *test.xml*: Contains the test targets to test the application and generate test reports.
- *docs.xml*: Contains the documentation targets to generate the project documentation.
- *dist.xml*: Contains the distribution targets to create the deliverables.
- *build.xml*: Contains the shortcut targets.

Finally, there's another Ant file under the *build* directory: *lib.xml*. This is an *antlib* file which contains the definitions of custom tasks that are used by the various build files. It also acts as a centralized place where all the dependencies to external libraries are documented -- by external library we mean a library that doesn't ship with Ant and thus has to be made available to the Ant runtime in order to run a build.

The *build.xml* file is the master build file on which Ant is invoked. It is responsible for pulling the child files together at runtime -- this is done through `import` statements. The result is that the Ant runtime treats all the imported targets and properties as if they belonged to a single build file, which avoids the overhead of running sub-builds. The *lib.xml* file is the first one to be imported, followed by the child build files in the same order in which they depend on each other -- because all those files are treated just as one at runtime, it's perfectly legal for a file to reference properties and targets defined in another file as long a this latter file is imported first.

All the child build files are required to follow the same conventions. Specifically, a child file is required to:

- Define a project name after the file name. For example the project name of the child build in *app.xml* is *app*.
- Define a properties namespace. This is done by prefixing all properties defined within the child file with the child project's name. This is important to avoid collisions, as Ant properties are global.
- Check for availability of external properties. If a child depends on properties defined elsewhere (another child file/master file), then it has to explicitly check that those

properties have been defined and error if some is missing. The checks are performed through `checkdef` custom tasks placed at the beginning of the file and outside of any target. This way, a broken dependency graph is likely to show up at import time in the master build file.

- Have a `clean` target. This target has to remove all output generated by the child's targets.
- Have a `usage` target. This target echoes the list of available targets. That is, the list of all public targets within the child build.
- Use the same conventions to denote public targets. The conventions are simple: public targets have a description attribute (private targets do not) and are listed by the usage target.

A final note. The master build file exposes `clean` and `usage` targets that simply forward to each `clean`/`usage` target in the child files. So if a new child file is added, then the `clean` and `usage` targets in the master file have to be updated. (Obviously enough, an `import` statement has to be added as well.)

5.2. Targets

The following diagram depicts the dependencies among targets and the files in which they are contained.

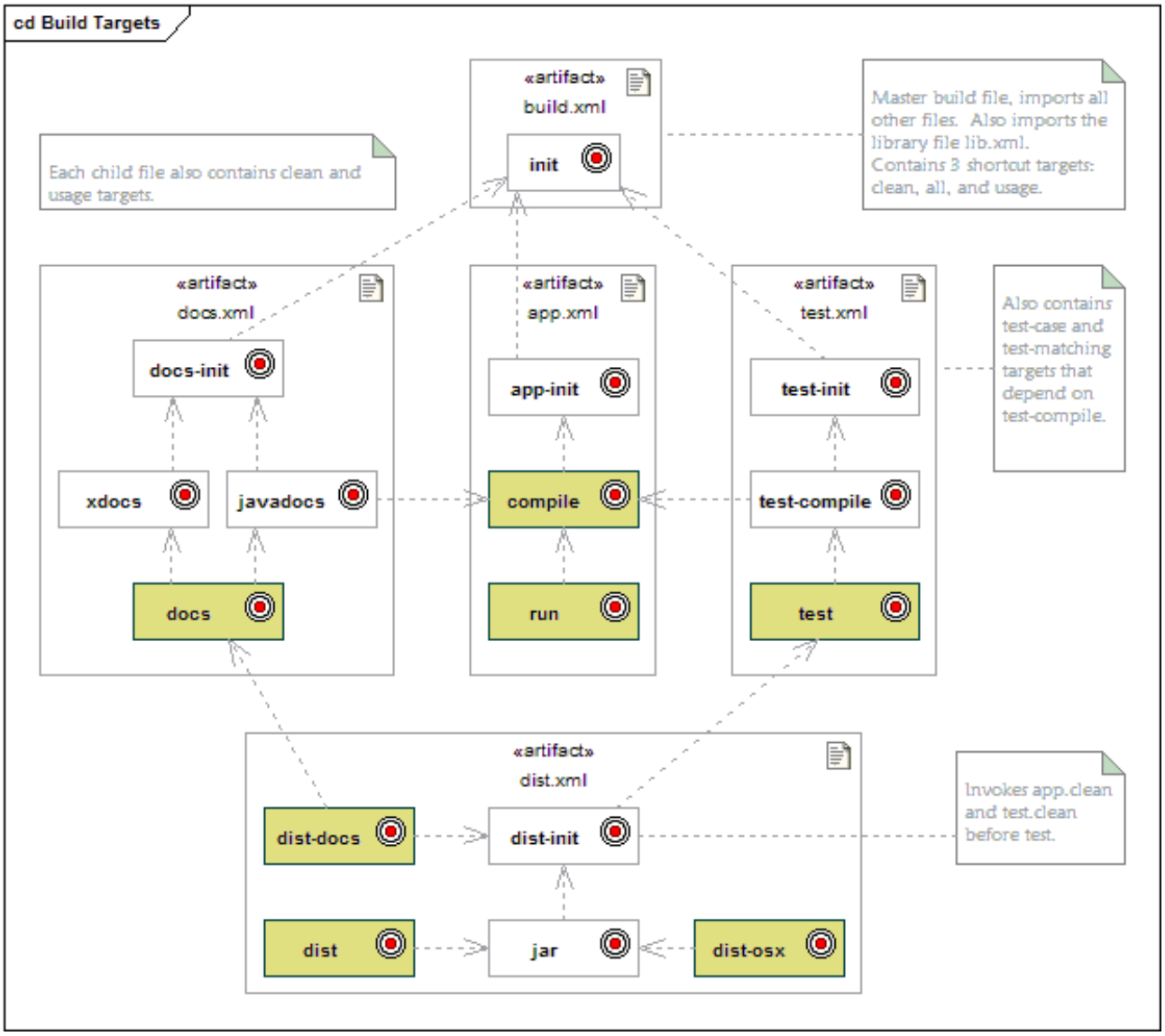

Fig 5.2-1 : Targets, dependencies, and build files.

The boxes labelled with *artifact* represent build files. The boxes with the target icon on the right are build targets. The placement of targets within build file boxes denotes containment -- for example, the `init` target is contained in the *build.xml* file. Dashed arrows denote build dependencies -- for example `app-init` in *app.xml* depends on `init` in *build.xml*. The boxes

with a folded corner are notes.

All the available public targets are highlighted in the diagram. All these targets have already been described in this document. You can find out more about them by reading the inline documentation that accompanies each build file. Also refer to the various build files for a description of what the private targets do.

On a final note, it's worth mentioning that, by design, it's not possible to create a distribution bundle (either generic or OS X) if part of the application doesn't compile or any test fails. In fact, `dist` and `dist-osx` both depend (indirectly) on `dist-init` which, in turn, depends on `test`. This latter target fails the build if any test within the application test suite fails. The `compile` and `test-compile` targets both fail the build in the case of a compilation error. Because of the dependencies layout, `test` can't be executed if `compile` or `test-compile` fails. Which explains why it's not possible to create a distribution bundle in the case of a (even partial) compilation failure.

5.3. Build Tool

The build tool is a Java program that invokes Ant to process the master build file. Its purpose is to run a build without requiring Ant to be installed and configured. It's composed of just one source file -- *build.java*, which is located in the *build/tools* directory. The build tool loads the Ant core and all the other libraries dynamically from the *tools* directory, prepares a suitable environment, and then invokes Ant with whatever arguments were supplied on the command line.

Besides the external libraries discussed in the previous sections, the *tools* directory also contains the Ant core jars and the Ant jars for the optional tasks that we use. So the command `java build` will result in invoking Ant with whatever arguments follow `build` on the command line and with a classpath set to contain all the jar files under *build/tools*. Because the classpath contains all Ant core and optional libraries and all the external libraries, all that is effectively needed to run a build is a JDK -- as opposite to a whole Ant installation with all external libraries.

Note

Default compilation tasks in Ant require the *Sun* JDK tools. Those ship with Sun JDKs and are contained in the *lib/tools.jar*. If you have a JDK from another vendor (or just a plain JRE), the Sun JDK tools might not be available, in which case compilation of OMERO.insight would fail. To fix this, proceed as follows:

- Make sure the Sun JDK tools are not available to the JVM you will use to invoke the Build Tool. Just run `Java build compile`; if it fails the Sun JDK tools are not available.
- Download the Sun JDK tools jar file into *build/tools*/. Make sure the version of the JDK tools is the same as the one of the JVM that you will use to invoke the build tool. (Failure to comply may result in compilation errors due to class file version incompatibilities.)

It's important to verify that the JDK tools are not available before dropping another copy into our *tools* directory. In fact, if the JDK tools are already available, adding a copy under *tools* would result in having two sets of JDK tools classes on the classpath. As you can imagine, some nasty runtime behavior could originate from that if the JDK tools have different versions.

The `build` class file is generated by an Ant build file: the *build.xml* file under the *build/tools* directory. You can refer to the inline documentation in this file for further details on the build tool. Also the *build.java* source file is thoroughly documented if you need to explore the implementation details.

Note

The *tools* directory also contains two XSL files. These are the stylesheets used by the `junitreport` task to make test reports. The reason why we're not using the embedded files that come with the task is that the Xalan `redirect` declared in these files doesn't work with some (nasty) configurations of JRE/Xalan. So we stripped those files out and changed the `redirect` statement.

5.4. Rationale

This section provides a brief discussion and justification of the design choices underpinning the build system.

First off, we chose Ant to implement the build system because of its excellent support for Java projects and its ubiquity among Java developers. We decided for version 1.6 (the latest stable release at the time of writing) because of the new features to support large projects, like *antlibs* and *imports*. In fact, this easily allowed us to split the build process in different categories of tasks and arrange the corresponding targets into separate files according to the category they belong to, thus making for improved understandability and ease of maintenance. Moreover, the definition of custom tasks could be factored out in a library file and shared among all build files. Some of those tasks were defined through scripts. The reason for that as opposed to a custom Ant task in Java is the reduced overhead and time required to write and integrate the code. Of course, this is feasible only in the case of tasks of moderate complexity.

The little extra work required for the build tool was justified by the fact that this tool:

- *Simplifies the build configuration*. In fact, a prerequisite to running the build system under Ant is that all the external libraries are properly configured and linked to the Ant runtime. On top of that, extra work could be needed if different versions of the libraries required by the build system have already been configured on the machine where the build is run. (Recall that some of those libraries depend on specific versions of other libraries and newer versions wouldn't work.)
- *Avoids upgrading to Ant 1.6*. If you have an earlier version of Ant, you can still run a build with the build tool without having to upgrade.
- *Avoids keeping up with build system updates*. If you're using Ant to run the build system, then every time a new custom task and external library is added you'll have to update your local Ant configuration.
- *Allows to replicate a build on different machines*. In fact, when the build tool is used, the only thing that might change across machines is the JDK. In practice, this results in a likelihood close to 100% of reproducing a build.
- *Avoids potential problems in the future*. Even if a future version of Ant would break our build, we can still run it with the build tool.

A final remark. All the libraries required by the build system are kept under the *build/tools* directory. We didn't put them under the *LIB* directory, because they have nothing to do with the application and test code. For the sake of clarity it's better to keep different things in

different places.

6. Legal

Licensing

This software is licensed under the terms of the GNU General Public License (GPL), a copy of which may be found in the *gpl.txt* file under the *legal* directory. The *README* file in that directory states which other licenses apply to third-party artifacts used by our software.

Copyright

```
Copyright (C) 2006-2008 University of Dundee. All Rights reserved.
```