Dev Studio Releng and Release Guide

A guide to the build system and release process for JBoss Developer Studio and JBossTools

1.0

Introduction

The releng system is a build system that we use to build, release, and test JBossTools and JBoss Devleoper Studio. Releng is a PDE-based wrapper that uses ant and custom ant tasks to call PDE-build for each component that needs to be built.

Before you proceed, you'll need to familiarize yourself with the Eclipse PDE-build system. Take a look at the documentation provided by Eclipse for their build system here: *Eclipse build system documentation* [http://dev.eclipse.org/viewcvs/index.cgi/org.eclipse.releng.basebuilder/readme.html?rev=HEAD#configs]

1.1. Check out the source code

Releng lives at *https://svn.jboss.org/repos/devstudio/trunk/releng* and can be checked out by using the following command:

```
svn co https://svn.jboss.org/repos/devstudio/trunk/releng
```

Directory Structure

2.1. Summary

The following is a quick tour of the directory structure of the build system and what you will find there.

```
org.jboss.ide.ecipse.releng
 -- bin
 -- builders
 -- configs
 -- lib
 -- requirements
 -- src
 -- util
```

`bin`	This is where the compiled custom ant tasks for the system live.
`builders`	Contains one folder for each component (including the product). These component folders contain PDE-build-specific property files and build scripts. It also contains the common build.xml file, which is the entry point to the entire build system.
`configs`	Contains the Hudson-configuration and property files
`lib`	Contains third-party dependencies of releng such as dom4j and antcontrib
`requirements`	Contains a folder for each binary required by a component.
`src`	The source code for the custom ant tasks for the system
`util`	Utility scripts for doing various things, specifically in the release process

2.2. Custom Ant Tasks

These tasks are provided as source and compiled by the build customization script. The binaries are then stored in the bin folder. They must be added to the classpath of any script or script

fragment that wishes to make use of them. To add these files to the classpath of an xml file, use the following snippet:

```
<taskdef classpath="../../bin" resource="org/jboss/ide/eclipse/releng/antlib.xml" />
```

`AnalyzeLogTask.java`	Currently unused, analyzes the releng log file to find unit test errors and errors and warnings in the build.
`CalculateFeatureDependenciesTask.java`	This task's purpose is to calculate the set of dependencies for the passed in feature. This task is recursive and will look at each plugin.xml or manifest.mf whether inside a jar or a folder. After calculating the dependencies, the given property will be filled with a comma-delimited list of dependencies. Note that this task will exclude any plugins found to be included in a standard eclipse SDK distribution.
`CreateSubversionFetchScriptTask.java`	deprecated file, was used because PDE did not previously have support for SVN
`DisableExtensionPointTask.java`	Edits an existing plugin.xml file to remove an extension point. The classes that implement that extension point will remain inside the jar, but the extension point itself will be removed. This could have uncertain results if a plugin defines an extension point and assumes some model has been initialized early on by the extension point mechanism before accessing it. Use with care ;) This is currently used to disable the welcome-screen during unit tests.
`EchoProperty.java`	NOT an ant task. Used by Hudson.
`GetFeaturePluginsTask.java`	Retrieves a comma-delimited list of plugins that are included in a specific feature. It is currently unused.
`GetRuntimeJarsTask.java`	Retrieves the runtime jars for a specific plugin by looking at the plugin's build.properties file. Also unused currently.
`RelengUtil.java`	NOT an ant task. Contains utility methods to read resources from features and plugins.

	It is primarily used to inspect a manifest file, plugin.xml, or feature.xml and is used widely.
`ReverseUpdatePlugins.java`	This is a subtask of UpdateVersionsTask. It is current not in use. It was previously used to force a feature.xml to update its version to one that was already in a plugin.
`UpdateVersionsTask.java`	This ant task will update the versions in a set of features and/or plugins.

```
<updateVersions type="[plugin|feature]"
                version="VERSION_TO_USE"
 append="[true|false]">
        <fileset ..>
</updateVersions>
```

Running the build Locally

3.1. Build Customization

Before using the build system, the ant tasks mentioned in the last chapter must be compiled and some properties must be set. The easiest way to do this is by migrating to the `trunk/releng/org.jboss.ide.eclipse.releng` folder and executing the following command:

```
ant -f customizeBuild.xml
```

This will do a few things. First, it will compile all of the ant tasks and put the compiled copies into the bin folder. It will also modify the customize.properties file to set up some custom settings required for the build.

The first two required properties are the location of the releng project root and where the build's output should be stored. It is advisable to choose an external folder for the outputs not within the root of the releng project.

The other options enable publishing to some server via scp once the build is over, or sending an email notification. The option to build from the local filesystem (using the local map file) instead of checking out source code every time (using the regular map file) is also present. If this option is chosen, then you must outline the source trees of the various repositories that encompass the JBDS product.

If local mode is chosen, the roots of these repositories should be either the trunk folder or the folder of a specific branch.

3.2. Common Use Cases

There are three main types of builds. The first is a a standard nightly build. Running it is as simple as typing `ant` while in the `trunk/releng/org.jboss.ide.eclipse.releng/builders` folder. The plugins are given versions similar to `1.1.0.20080815-nightly`

The second is called the integration build. It is no longer used very much, as the version numbers were seen as problematic when having dependencies between plugins resolved. To execute this build, you would type `ant build-integration`

The third build type is called the release build. To execute it, a few parameters are required. The first is the path to a tags file, which matches a variable `{component-name}.svn` to a svn tag location where the tag is stored.

The second variable for a release build is `releaseType`. This type may be either development or stable. The difference between these two values is primarily which update site they are sent to. The development update site receives alphas, betas, and release candidates. It will *not* receive official releases. The stable update site *only* receives official releases.

The third parameter is the release number. A useage example is shown below:

```
ant build-release -Dtags-file=/path/to/tags/file -DreleaseType=(development | stable) -DreleaseNumber=2.0.0.GA
```

To run the build for only one builder, execute the following command:

```
ant -Djbds-builder=birt
```

It is worth noting that as of right now, there is no easy method to run just the build and test of any one plugin. It requires much fiddling, and perhaps even a full build first. It is not easy or intuitive. Officially, there is a single builder for the tests and you can execute it by running `ant -Djbds-builder=tests`, but this one builder runs *all* tests and will fail if all plugins are not already built. It is *impossible* to run just one component's tests.

Debugging

The primary utility for debugging releng builds is reading the log. In it, you will find all of the error messages generated by the ant tasks, or by PDE-build. The most common errors are listed below.

4.1. Compile Errors

These errors can be caused by several things. The first and most obvious is a programming error in a plugin's program code. These are easy to spot, and the error log will point to the line.

Other causes of the typical compile error are a missing dependency. It could be that the plugin depends on another builder, but that builder has not yet been built or it is not listed as a dependency in the build.requires property file for that component.

4.2. Bad Map File

The map file in a builder (jbosside-as.map, for example) lists the location of the source code for a feature and all of its plugins. If either the feature is missing, or one of the feature's plugins, is missing from this file, then a PDE error will cause the build to fail.

4.3. Requirement Failed

If the download of a requirement fails, the build will fail.

4.4. Unable to locate PDE scripts

If the primary eclipse driver has changed, then the build may fail. The remedy to this would be to change the location of the PDE scripts, which is stored in the global.properties file.

4.5. IDs don't match

Other problems that may be more difficult to debug could be mis-matched id's in any of the builder's files. You'll want to check all of the component's plugins, features, manifest files, build.properties files, and builder files (build.properties, build.requires, map files, and customTargets.xml)

4.6. Bad Tags File

If the tags file, which is stored in `builders/product/versionTags/{type}/{version}`, includes incorrect versions, it could cause hard-to-debug problems in the final release.

4.7. Debugging PDE build

Marshall Culpepper has a blog entry about this. It lives at *http://www.arcaner.com/2007/03/22/debugging-eclipses-pde-build/* [http://www.arcaner.com/2007/03/22/debugging-eclipses-pde-build/]

Adding or Updating a requirement

New components will require different libraries, drivers, or other eclipse plugins that they depend on. This section will describe how to add both entirely new requirements, or add new versions of those requirements. In the example below, we'll be adding a requirement named newReq with a version of 1.0.0.

5.1. Mirroring the driver

When a new requirement needs to be created, the first step is to create a folder for it in the eclipse folder of repository.jboss.org. It is important to mirror all requirements on our own svn server to ensure successful builds that don't time out when retrieving a requirement. The section below will show how to add the requirement to our requirements repository.

Before doing this, though, you'll want to make a temporary folder somewhere on your disk to work in. You'll then want to create the new requirement as shown below, and then immediately check it out.

```
[rob@localhost repo]$ mkdir temp
[rob@localhost repo]$ cd temp
[rob@localhost temp]$ svn mkdir https://svn.jboss.org/repos/repository.jboss.org/eclipse/newReq
Committed revision 12160.
[rob@localhost temp]$ svn co https://svn.jboss.org/repos/repository.jboss.org/eclipse/newReq
Checked out revision 12160.
```

Now you'll have to create a version for this requirement.

```
[rob@localhost temp]$ cd newReq/
[rob@localhost newReq]$ mkdir 1.0.0
[rob@localhost newReq]$ svn add 1.0.0/
A         1.0.0
[rob@localhost newReq]$ cd 1.0.0/
[rob@localhost 1.0.0]$ cp ~/Desktop/requirement1.jar .
[rob@localhost 1.0.0]$ svn add requirement1.jar
A         requirement1.jar
[rob@localhost 1.0.0]$ svn commit -m "adding test requirement while writing docs will delete later"
Adding         1.0.0
Adding         1.0.0/requirement1.jar
Transmitting file data .
Committed revision 12161.
```

You can delete this local copy temporary folder right after this, as the requirement is now mirrored properly in version control.

5.2. Adding the requirement to the build system

The requirement must also be added to the build system, under the `trunk/releng/org.jboss.ide.eclipse.releng/requirements` folder. Officially it consists only of one xml file called `buildRequirement.xml` with one target named `build.requirement` which does the work.

However, there is a utility currently stored in the webtools requirement folder which can fetch a requirement if a few properties are set properly.

5.2.1. How to use the buildDriver.xml template

Below is the xml from the dtp requirement's buildRequirement.xml file.

```
<project default="build.requirement">
 <target name="build.requirement">
  <ant antfile="../webtools/buildDriver.xml" target="build.driver" inheritall="true">
   <property name="driver.properties" value="${basedir}/build.properties"/>
          <property name="requirement" value="dtp"/>
  </ant>
 </target>
</project>
```

And from it's build.properties file:

```
build.uri=http://repository.jboss.org/eclipse/dtp/1.6
build.archive=dtp-sdk_1.6.0.zip
```

All that's being done here is the external script, stored in the webtools requirement, is being called with a requirement name (dtp) and a properties file to check for some property names, specifically build.uri and build.archive. The script will check if this file has already been downloaded or not. It will then download and unzip it if it has not.

If a requirement requires multiple jars, as `birt` does, it can repeat the ant call several times specifically designating the build.uri and build.archive properties.

5.3. Updating the requirement

When updating to a new version of a requirement, the same process should be used as in creating a new requirement. A new version folder must be added to the repository (as shown earlier) and

the drivers uploaded. The buildRequirement.xml must also be updated to make use of the new version.

Creating a builder

A builder is both an eclipse PDE construct and a releng construct. We abstract out bits of pieces and create default configurations, add some property files, etc. New builders are allowed to dig down and add custom behavior to some portions of the process.

6.1. Creating a new Builder

The first step to creating a new builder is to physically create the folder for it in the requirements directory.

```
[rob@localhost trunk]$ cd releng/org.jboss.ide.eclipse.releng/builders/
[rob@localhost builders]$ mkdir testBuilder
[rob@localhost builders]$ cd testBuilder
[rob@localhost testBuilder]$
```

The files we'll need to create are: build.properties build.requires customTargets.xml test.local.map test.map

6.1.1. build.properties

The build.properties file is a simple properties file. Most of the properties are duplicates and could probably be abstracted out during a refactor of the system, but for now it is safer to copy another component's file and change only those files which need to be changed.

Assuming the new component lives in the same svn repository as the rest of JBossTools, the only properties which should be changed are: `buildId, zipFile,` and `nodeps-zipFile`. The rest of the properties should be fine as they are.

6.1.2. build.requires

The build.requires file contains only two properties. The first, `builder.requires`, designates which external requirements are needed to compile against. These are things like webtools, dtp, gef, emf, and eclipse itself. The second is which other builders are required and must be built first.

An example of this file is below:

```
builder.requires=jbeap,eclipse,emf,xsd,gef,dtp,webtools,tptp
builder.requiredBuilders=core, common
```

6.1.3. Map Files

The two map files, one for local and one for remote, help eclipse and PDE figure out where to find the source for the components it is about to build. There are two different formats for the file, and in releng we request that all builders have both a local and a remote implementation. Later on, the customTargets.xml file will decide which map file to give to PDE.

6.1.3.1. Remote

An example remote map file (from birt) is shown here:

```
feature@org.jboss.tools.birt.feature=SVN,anonymous,,%svnTag%,%svnURL%,,birt/features/
org.jboss.tools.birt.feature
plugin@org.jboss.tools.birt.core=SVN,anonymous,,%svnTag%,%svnURL%,,birt/plugins/
org.jboss.tools.birt.core
plugin@org.jboss.tools.birt.oda=SVN,anonymous,,%svnTag%,%svnURL%,,birt/plugins/
org.jboss.tools.birt.oda
plugin@org.jboss.tools.birt.oda.ui=SVN,anonymous,,%svnTag%,%svnURL%,,birt/plugins/
org.jboss.tools.birt.oda.ui
```

The `%svnTag%` and `%svnURL%` variables are replaced by releng to meaningful values before being passed to eclipse and PDE.

6.1.3.2. Local

An example local map file (from birt) is shown here:

```
feature@org.jboss.tools.birt.feature=COPY,%root%,birt/features/org.jboss.tools.birt.feature
plugin@org.jboss.tools.birt.core=COPY,%root%,birt/plugins/org.jboss.tools.birt.core
plugin@org.jboss.tools.birt.oda=COPY,%root%,birt/plugins/org.jboss.tools.birt.oda
plugin@org.jboss.tools.birt.oda.ui=COPY,%root%,birt/plugins/org.jboss.tools.birt.oda.ui
```

The `%root%` variable is replaced by releng to a meaningful value before being passed to eclipse and PDE.

6.1.4. customTargets.xml

This is the real meat of the builder, the piece which integrates extensively with PDE. It is highly suggested you copy this file from another builder such as `as` or `birt`. The simplest explanation for how to create this file copy it and then change all id's to match the feature and plugins you're building with this builder.

This script, in each builder, is both the essential piece, and a wrapper for the essential piece. If we look at `allElements`, for example, we see that it calls `${genericTargets}` and passes itself as

a callback file. So while much of the important work is done in customTargets.xml, a large portion is also done in genericTargets, which lives in `trunk/releng/org.jboss.ide.eclipse.releng/builders/common`

6.1.4.1. allElements

The first target is called `allElements`, and an implementation is shown below:

```
<target name="allElements">
	<ant antfile="${genericTargets}" target="${target}" >
		<property name="customTargets" value="${builderDirectory}/customTargets.xml"/>
		<property name="type" value="feature" />
		<property name="id" value="org.jboss.tools.birt.feature" />
	</ant>
 </target>
```

In this example, our customTargets.xml is delegating to our genericTargets.xml file. Eclipse comes with its own genericTargets.xml file, but ours was added long ago when PDE did not provide support for SVN. It is rumored ours can be done away with, but no investigation of this has been done yet.

If your new builder will build a component with two features, or two features and a standalone plugin, this `allElements` target can execute multiple ant calls instead of just the one.

6.1.4.2. getMapFile

This section is short, but you must check to make sure the builder name and map file names are accurate

```
<target name="getMapFiles">
	<!-- we need to export the map file w/ the correct CVS tags -->
	<if>
		<equals arg1="${localMode}" arg2="true" casesensitive="false"/>
		<then>
			<copyMapFile builder="birt" buildDirectory="${buildDirectory}"
				mapFile="birt.local.map" root="${jbosstools.root}"/>
		</then>
		<else>
		   <copyMapFile builder="birt" buildDirectory="${buildDirectory}" svnURL="${svnURL}"
 svnTag="${s
vnTag}"/>
		</else>
	</if>
```

```
</target>
```

6.1.4.3. preSetup

In preSetup, we remove all return.properties files from our requirements folder. This was a properties file that is generated by our build system when we handle a requirement.

6.1.4.4. postFetch

postFetch changes the versions of the created plugins and features to match what we need it to be for nightly or release builds. The plugin and feature patterns here must be changed and accurate.

6.1.4.5. postAssemble

In the postAssemble phase, we possibly bundle our dependencies. We call a macro which, in this case, is declared in `releng/org.jboss.ide.eclipse.releng/builders/common/bundleDependencies.xml`. Remember to change or verify the ids here.

6.2. Other files

The following files need to be updated to include the new builder:

1. util/make_tags.sh
2. builders/product/updateSite/development/site.xml
3. builders/product/updateSite/stable/site.xml
4. builders/tests/build.properties
5. builders/tests/tests.local.map
6. builders/tests/tests.map
7. builders/product/buildResults/buildResults.html

These files are all easy and intuitive to update.

Hudson

7.1. Hudson

Release

8.1. Tagging

The first step to making a release is to tag everything. This can be done by executing the following command

```
BRANCH=trunk TAG=-jbosstools-3.0.0.Alpha1 ./make_tags.sh
```

8.2. Creating a tags file

The tags file is what designates the version for each specific plugin, as well as where they are located in svn. The file lives in `builders/product/versionTags/{product}/{version}.tags`. The general structure is one property for the plugin version and one for where it is located in a repository. It supports both cvs and svn. An example is shown below

```
# cvs tags
jbpm=3.1.4.GA
jbpm.cvs=jbpm_jpdl_gpd_3_1_3_sp1

# svn tags
portlet=1.0.0.Alpha1
portlet.svn=tags/jbosstools-3.0.0.Alpha1
```

8.3. Modify release.properties

This file lives in `configs/{product}/release.properties.`

8.4. QA

No more needs to be said here. Let QA do their work. Once there's a concensus or Max says "go", we go.

8.5. Sourceforge

The first step is to get all output files from download.jboss.org that need to be sent to sourceofrge. You can either download each from the build results page manually or you can sftp into it with the username f-jbosside. This process requires both a passphrase and a key file.

Once the files are grabbed, assuming you have sourceforge access to the project, you would log into sourceforge and upload all of these files. The release notes on sourceforge typically come directly from our JIRA changelog / release notes.

8.6. jboss.org/tools

If you are logged in and have proper permissions, the screen should offer you an "edit" button about halfway down the page. The page here times out, and so should be edited in a text editor rather than in the web window.

8.7. The Update Site

Step one is to log into download.jboss.org and save the site folder for the build which has just been made. Again, sftp-ing into download.jboss.org requires both a key and a passphrase. Locally, rename the site/eclipse folder to `development`. Then, on the download.jboss.org box, change directory to `/htdocs/jbosstools/updates` and rename the current development folder to its proper version name. Then upload your local development to this folder.

8.8. Announcements

Announce on jbosstools-announce, forums, jbosstools.blogspot.com, and Max will probably blog on in.relation.to.

8.9. Developer Studio Erata

For Developer Studio, you must send the product manager, previously Brian Che, links to the installer jar files and the source tarball. All of these files are located under reports.atl.jboss.com. The source tarball is a bit harder to locate. An example url for this file is: `http://reports.qa.atl.jboss.com/binaries/RHDS/release/1.1.0.GA/jbdevstudio-source-1.1.0.GA.tar.gz`