PRIMIX SOLUTIONS

Core Labs

Java Build Environment

CORE LABS

Java Build Environment

One Arsenal Marketplace
Phone (617) 923-6639 • Fax (617) 923-5139

Tapestry contact information:

Howard Ship <hship@primix.com>

http://sourceforge.net/projects/tapestry

Table of Contents

Introduction 2

Installation and Configuration 4

- Installing Cygwin 4
- Environment Variables 5
- Configuration 5

Using JBE 7

- Jar modules 9
- Package Makefiles 11
- War modules 11
- WebLogic modules 13
- jBoss modules 13
- Additonal Variables 15

Future development of the JBE17

Chapter

1

Introduction

What is a build environement?.

As envisioned but most tool makers, the life of a Java developer is a solitary one. Parked at his or her desk, with only his trusty tools, IDEs and the command line, the developer creates the wonderful applets, applications and frameworks possible using Java.

The developer has and requires great freedom; each tool in his or her arsenal may have come from a different company; each tool may have been installed into a directory of his or her liking. This is not a problem because the developer is only accountable to him- or her-self, and these selections of tools and locations will only aftect one person.

Alas, in the real world, developers work on teams and share code using source code repositories. They may even make conflicting changes to code.

What's needed is a system that can adjust for local differences in developer's environments and allow for "clean builds" of projects directly from source. That's what the Java Build Environment (JBE) is for.

JBE is designed to start with basic Java source and utlimately produce a Java Archive (JAR) or Web Archive (WAR) ready for testing or deployment. This may involve many steps, including compiling Java code, creating RMI stubs and skeletons, using application-server specific tools (such as WebLogic ejbc), combining the results into a Jar file ... even creating Javadoc.

Without JBE, there are three options for doing all this:

- The command line. Developers may simply execute the java and jar commands themselves. This leads to problems when steps are missed, or the commands are in some way dependent on a single developer's environment (for example, the setting of the CLASSPATH variable).

- Batch commands / shell scripts. Hard to develop and debug and non-portable between platforms. Often assembled in a hurry that leads to the same kind of environment problems as using the command line.

- IDE. Some IDEs can assemble JARs, or even interface to an application server to build and deploy EJBs and WARs; however, this will require that the user interactively use the IDE's tools. Also, these tools are idiosyncratic; getting them to package up the correct Java classes and resources is often challenging.

JBE includes many hooks to allow custom directories, compiliation options or other configuration to be specified. Other hooks allow for additional processing, such as signing a JAR.

JBE is designed to be portable, meaning the same source files and Makefiles will work across multiple developer's workstations ... even when using different operating systems (such as Windows, Linux and Solaris).

Because the JBE is based on GNU Make it is extermely adaptable and extensible. JBE Makefiles can do more than simply compile Java files, they can also run tests, launch applications, prepare distributions … virtually any operation that can be done using shell scripts or command macros, though Makefile syntax is cleaner, easier and platform independent.

JBE is useful with medium to large scale Java projects. It has no support for compiling anything but Java; projects which use native code are beyond its scope.

Chapter

2

Installation and Configuration

JBE is distributed as a small ZIP or tar-ball containing this document and a collection of Makefiles. These files should be extracted to a permanent directory.

Under Windows, it is necessary to perform a separate installation to provide the necessary GNU tools, including GNU Make. JBE was developed under Windows (Windows NT 4.0 Workstation and Windows 2000 Professional), using Cygwin.

Future plans include porting it to Solaris (in such a way that Makefiles are automatically portable between the two environments); this will support coding under Windows and deployment on Solaris (including compiling on Solaris[1]).

Because my direct experience is to suffer the indignities of Windows development, the examples in this document use Windows pathnames. Developers using Linux or Solaris should be able to translate to their sensibly named file systems.

On any operating system it is necessary to have a JDK installed. JBE was developed using Sun's JDK 1.2.2. The JDK used by the JBE is configurable (details are below).

Installing Cygwin

Cygwin is a set of GNU tools ported so as to run in the Windows NT or Windows 2000 environments (it may also work under Windows 95/98/ME).

Cygwin is available at `http://sources.redhat.com/cygwin/`.

Installing Cygwin is very easy; Cygwin starts by downloading a small installer called **`setup.exe`**. Running this program allows the user to install any of the many individual packages in Cygwin individually.

---

[1] Some would say that this is an uneccesarily conservative step, because of Write Once Run Anywhere, but being a little paranoid never hurt.

Installing all packages is overkill: the JBE requires only a small number of the available tools. Alas, at this time, there is not a definitive list of what *is* required. Installing everything takes a few minutes (with a fast connection) and uses about 125MB of disk space.

Environment Variables

The JBE is packaged with Tapestry; if you are reading this document, then you have probably already unpacked the Tapestry distribtution. The JBE is stored in the `JBE` subdirectory of the distribution.

If you've unpacked the Tapestry distribution to a stable directory, there's no need to go further.

Otherwise, you should copy the JBE directory to some new location (perhaps C:\JBE) for permanent access.

Once the directory containing the JBE has been determined, it is necessary to create an environment variable, SYS_MAKEFILE_DIR, that points to the directory. Use forward slashes as the path seperator, even under Windows. A typical value for this is "`C:/JBE`".

Under Windows, add an additional environment variable `MAKE_MODE` with the value `unix`.

Configuration

Configuration is accomplished by creating additional files used by GNU Make at runtime.

First, create a sub-directory of the JBE directory and name it `config`.

Create a new file, `SiteConfig.mk`, in the directory. This file is primarily used to specify the platform and JDK for the local workstation.

Example:

config / SiteConfig.mk

```
# Defines the local platform.

SITE_PLATFORM := Cygwin

SITE_JDK := Sun_Win32
```

Platforms correspond to the `Platform.`*`name`*`.mk` file in the main JBE directory. JDKs correspond to the `JDK.`*`name`*`.mk` file in the main JBE directory.

This file may also be used to store addtiional, site-wide options (typically, variables that start with the prefix `SITE_`). Such options will apply to all modules built on the local workstation.

In a multiple-developer environment, all developers on the same platform will use identical copies of the `SiteConfig.mk` file.

A second configuration file, `LocalConfig.mk`, is used to establish the directories into which related tools have been installed. For example:

config/LocalConfig.mk

```
TOOLS_DIR := C:/cygwin/bin

JDK_DIR := C:/jdk1.2.2

WEBLOGIC_DIR = C:/WebLogic
```

The first variable, `TOOLS_DIR`, is the directory for the GNU tools. `JDK_DIR` identifies where the Sun JDK was installed. `WEBLOGIC_DIR` identifies where the WebLogic application server was installed ... this is only needed if modules will be built against Weblogic.

The other makefiles, especially `Platform.`*`name`*`.mk` and `JDK.`*`name`*`.mk`, uses this information to locate the GNU and JDK tools needed for builds.

Chapter

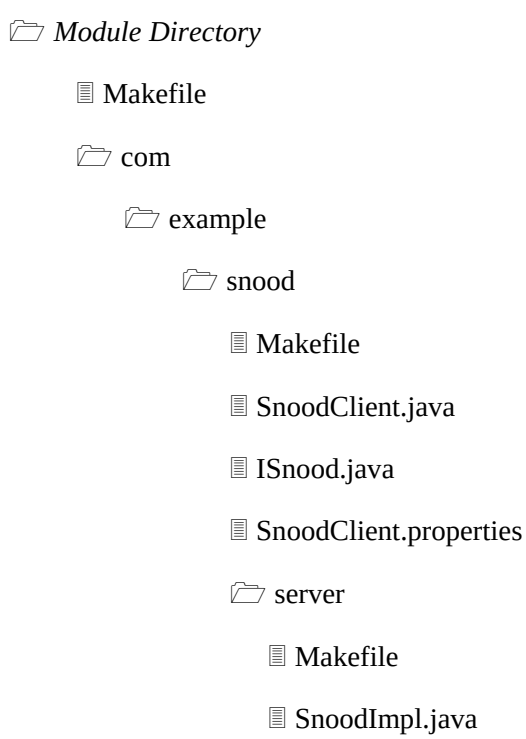
3

Using JBE

JBE performs builds on *modules*. For JBE, a module is a directory which contains a number of Java packages. The source code in all the packages should be compiled and eventually combined into a single JAR file (or, in some cases a WAR file).

JBE requires a master Makefile in the module directory, whose job is to set global options for the entire module, and to identify the list of Java packages. Each package *may* also have a Makefile, which is used to identify the Java source files, resource files and RMI classes for that package.

The standard directory hierarchy for a module has the module directory as the root of the Java package tree for the module:



This lays out a module with two package directories. The first, `com.example.snood` contains `SnoodClient.java` and `ISnood.java` and a resource file `SnoodClient.properties`. The second package, `com.example.snood.server` contains `SnoodImpl.java`..

A module makefile must provide a name for the module (which is used to name the JAR or WAR file) and a list of packages. It may provide additional options used when compiling, generating Javadoc or installing the JAR.

The module directory doesn't *have* to be the source code root, by setting the SOURCE_DIR variable, the source code root directory can be moved to another directory. Many developers prefer to put the Java source code in a directory named "src". In other situations, the Java source code may be in a sibling directory to the module directory.

In our example, the module directory is also the root source code directory, so the Makefile is very simple:

Makefile

```
MODULE_NAME = Snood

PACKAGES = \
      com.example.snood \
      com.example.snood.server

include $(SYS_MAKEFILE_DIR)/Jar.mk
```

The last line identifies this module as a Jar module; one that builds a JAR file.

The other two Makefiles identify that Java source files in the package, any resource files that should be copied into the JAR, and any classes that must be compiled with the RMI compiler.

com/example/snood/Makefile

```
JAVA_FILES = *.java

RESOURCE_FILES = *.properties

include $(SYS_MAKEFILE_DIR)/Package.mk
```

com/example/snood/server/Makefile

```
JAVA_FILES = *.java

RMI_CLASSES = SnoodImpl

include $(SYS_MAKEFILE_DIR)/Package.mk
```

Building this module executes a sequence of commands[2]:

```
make

*** Cataloging package com.example.snood ... ***


*** Cataloging package com.example.snood.server ... ***
```

[2] As the JBE evolves, the exact commands may alter slightly, but the general pattern will be the same.

```
*** Compiling ... ***

C:/jdk1.2.2/bin/javac.exe -d .build/classes -classpath
"D:/Temp/Snood;D:/Temp/Snood/.build/classes" com/example/snood/ISnood.java
com/example/snood/SnoodClient.java com/example/snood/server/SnoodImpl.java

*** Compiling RMI stubs and skeletons ... ***

C:/jdk1.2.2/bin/rmic.exe -d .build/classes -classpath
"D:/Temp/Snood;D:/Temp/Snood/.build/classes" \
         com.example.snood.server.SnoodImpl

*** Copying package resources ...***

Copying: SnoodClient.properties

*** Building Snood.jar ... ***

C:/jdk1.2.2/bin/jar.exe cf Snood.jar -C .build/classes .
```

When a module is first built, JBE catalogs the Java source files, resource files and RMI classes in the packages (this information is kept for subsequent makes[3]). It then uses this information to perform all the remaining work from the module directory.

Here it compiled all the Java files in one pass, built the RMI stubs and skeletons, then copied resource files, and created the final JAR file.On a subsequent build, only files which had changed since the previous build would be recompiled or re-copied.

You can also see that full pathnames are used to access the various GNU and JDK tools. This ensures that the correct JDK is used. It also means the tools are available, even if the user hasn't added the JDK\bin dir to the system PATH.

JBE creates a `.build` directory in the module directory and directs compilation to this directory as well as copying resource files into it. It just becomes a matter of using the JDK jar tool to create a JAR from the directory. WARs are generated the same way (but with a different structure).

Jar modules

The most basic type of JBE module is the Jar module, which builds a JAR file that can be used as a framework or standalone application. The JAR file is created in the module directory (though it can be removed by `make clean`).

A Jar module Makefile should define the following values:

---

[3] If the files change, such as when files are added or removed, the catalog used by make will be out of date. It can be refreshed by building the `catalog` target, though its often simpler to just do a `make clean`.

Variable	Description
INSTALL_DIR	The directory to which the final JAR should be copied after it is built.
LOCAL_CLASSPATH	A space seperated list of the classpath entries (typically, other JAR files) used when compiling. Absolute or relative pathnames may be used. Use the forward slash as the path seperator (even on Windows).
META_RESOURCES	The names of any resources that should be copied from the module directory into the JAR's META-INF directory. This is optional.
MODULE_NAME	The name used when building the JAR, and as the sub-directory when building Javadoc.
PACKAGES	The names of all packages in the module.
SOURCE_DIR	An alternate directory to serve as the root directory for Java source and Java class resources. This is optional; if not specified the module directory is treated as the source directory.

A Jar module has a number of standard Make targets:

Target	Description
catalog	Rebuild the catalog of Java files, resource files and RMI classes. Used after adding or removing such files from a package.
clean	Remove JAR and the .build directory.
compile	Compile changed Java source files, then compile any changed RMI classes.
default	Alias for `compile`.
force	Compile all, not just dirty, then compile all RMI classes
install	`jar`; then copy JAR to INSTALL_DIR
jar	`compile`; then copy resources and build JAR
javadoc	Generate Javadoc for the contents of the JAR

Package Makefiles

Package Makefiles are very simple. They are only used when cataloging; they simply declare what types of files are in the package.

The form of a Package makefile is very simple:

```
JAVA_FILES = list of files

RESOURCE_FILES = list of files

RMI_CLASSES = list of class names

include $(SYS_MAKEFILE_DIR)/Package.mk
```

As shown in the prior examples, the `JAVA_FILES` and `RESOURCE_FILES` may use wild cards (such as "*.java" or "*.properties").

The `RMI_CLASSES` is simply a list of the class names of RMI implementation classes; just the simple name of the class (no extension, and no package name; the package is known from context).

Package makefiles are optional, if not provided, a default makefile is used. The default makefile assumes all files ending in the following extensions are resource files:

- jwc (Tapestry Component Specifications)
- application (Tapestry Application Specifications)
- html (Tapestry HTML templates)
- properties (Java properties files)

War modules

A War module is similar to a Jar module, except that the final file has the extension ".war" instead of ".jar" and the internal layout is different. A Web Application Archive (WAR) is a file that can be deployed into a J2EE compatible application server; it contains servlets and other Java code as well as context resources (images and other assets that are visible to the web server).

In a WAR, classes are stored in the directory `WEB-INF/classes`, rather than at the root. Context resources go in the root of the WAR (these are images and other files that are accessible by the client web browser). There will deployment descriptor files that must also be copied from the module directory into the `WEB-INF` directory as well, and a WAR can include libraries of code in its `WEB-INF/lib` directory.

Variable	Description

CONTEXT_RESOURCES	The names of individual files or directories that should be copied from the module directory into the root of the WAR. Relative pathnames will be maintained when copied. Directories are copied recusively (but directories named 'CVS' are pruned).
INSTALL_DIR	The directory to which the final WAR should be copied after it is built.
INSTALL_LIBRARIES	A space seperated list of libraries that should be installed into the WEB-INF/lib directory. The entries here should be a subset of LOCAL_CLASSPATH.
LOCAL_CLASSPATH	A space seperated list of the classpath entries (typically, other JAR files) used when compiling. Absolute or relative pathnames may be used. Use the forward slash as the path seperator (even on Windows).
META_RESOURCES	The names of any resources that should be copied into the WAR's META-INF directory.
MODULE_NAME	The name used when building the WAR, and as the sub-directory when building Javadoc.
PACKAGES	The names of all packages in the module.
SOURCE_DIR	An alternate directory to serve as the root directory for Java source and Java class resources.
WEB_INF_RESOURCES	The names of files that should be copied into the WEB-INF directory. This should include the application-server specific deployment descriptor. The J2EE deployment descriptor, `web.xml`, is automatically copied.

War modules have similar targets as Jar modules:

Target	**Description**
catalog	Rebuild the catalog of Java files, resource files and RMI classes. Used after adding or removing such files from a package.
clean	Remove WAR, .build directory (in module and in each package)
compile	Compile changed Java source files, then compile any changed RMI classes.
default	Alias for `compile`

force	Compile all, not just dirty, then compile all RMI classes
install	`war`; then copy WAR to INSTALL_DIR
javadoc	Generate Javadoc for the contents of the WAR
war	`compile`; then copy resources and build WAR

WebLogic modules

The WebLogic module type is a specialization of the Jar type used to create deployable EJB JARS for use with the WebLogic application server version 5.1[4]. To use it, the `WEBLOGIC_DIR` variable must be set, usually in `LocalConfig.mk`.

The libraries *`WEBLOGIC_DIR`*`/classes` and *`WEBLOGIC_DIR`*`/lib/weblogicaux.jar` are automatically added to the classpath. These add WebLogic's implementations of the J2EE frameworks (JNDI, EJB, etc.).

For the most part, WebLogic modules work the same as Jar modules. However, the `jar` rule is changed to not only build the normal JAR, but also build the depoyable JAR. It does this by running the WebLogic `ejbc` command, which provides all the WebLogic specific classes needed to deploy (such as stubs and skeletons for EJBs, and a variety of files to support container managed persistence).

The deployable JAR is called *`MODULE_NAME`*`-deploy.jar`. The install rule copies both JARs to the install directory.

JAR files in the LOCAL_CLASSPATH are treated as dependencies of the deployable JAR. If they change, then the deployable JAR is rebuilt (using `ejbc`).

The WebLogic module automatically adds the files `ejb-jar.xml` (the generic EJB deployment descriptor) and `weblogic-ejb-jar.xml` (the WebLogic specific EJB deployment descriptor) to the list of `META_RESOURCES` (files copied to the META-INF directory of the JAR).

jBoss modules

jBoss is an open-source J2EE application server available from http://www.ejboss.org/.

The jBoss module type is a specialization of the Jar type used to create EJB JARS for use with the jBoss. jBoss is easier to work with than WebLogic, since it doesn't require the creation of stubs and skeletons, and deployment is as simple as copying the JAR file to a predetermined directory.

To use it, the `JBOSS_DIR` variable must be set, usually in `LocalConfig.mk`.

---

[4] WebLogic 6.0 is currently in an open beta; it appears that much of the infrastructure provided by the WebLogic makefile is no longer necessary, and deployable EJB JARs can be built using the normal Jar makefile.

The following libraries are automatically included in the classpath:

- *JBOSS_DIR*/lib/ext/ejb.jar
- *JBOSS_DIR*/lib/ext/jndi.jar
- *JBOSS_DIR*/lib/jdbc2_0-stdext.jar

These three libraries provide the basic J2EE functionality. `ejb.jar` provices the `javax.ejb` classes, `jndi.jar` provides the `javax.naming` classes and `jbdc2_0-stdext.jar` provides the `javax.sql` classes.

For the most part, jBoss modules work the same as Jar modules.

The jBoss module automatically adds the file `ejb-jar.xml` (the generic EJB deployment descriptor) to the list of `META_RESOURCES` (files copied to the META-INF directory of the JAR). If you use the auxilliary descriptor files (`jboss.xml` and/or `jaws.xml`) you'll need to manually add those to `META_RESOURCES`.

A jBoss module has a number of standard Make targets. Most of these are the same as the Jar module, with the exception of the deploy target.

Target	Description
catalog	Rebuild the catalog of Java files, resource files and RMI classes. Used after adding or removing such files from a package.
clean	Remove JAR and the .build directory.
compile	Compile changed Java source files, then compile any changed RMI classes.
default	Alias for `compile`.
deploy	`jar`, then copy the JAR file to the JBOSS_DIR/deploy directory to be hot deployed into the running jBoss server.
force	Compile all, not just dirty, then compile all RMI classes
install	`jar`; then copy JAR to INSTALL_DIR
jar	`compile`; then copy resources and build JAR
javadoc	Generate Javadoc for the contents of the JAR

Additonal Variables

The behavior of the JBE is controlled by makefile variables. The prior sections identified the minimum variables needed for the different types of modules, this section provides information on variables that are useful in all modules. Typically these supply additional options to existing commands and rules.

Variable	Declared	Description
EJBC_OPT	Makefile	Additional module-specific options for WebLogic ejbc.
JAVAC_OPT	Makefile	Module specific java compiler options.
JAVADOC_OPT	Makefile	Javadoc options.
LOCAL_JAVAC_OPT	config/LocalConfig.mk or environment variable	Local workstation java compiler options.
LOCAL_RMIC_OPT	config/LocalConfig.mk or environment variable	Local workstation RMI compiler options.
RMIC_OPT	Makefile	Module specific RMI compiler options.
SITE_EJBC_OPT	config/SiteConfig.mk or environment variable	Additional site-wide options for WebLogic ejbc.
SITE_JAVAC_OPT	config/SiteConfig.mk or environment variable	Site-wide java compiler options.
SITE_RMIC_OPT	config/SiteConfig.mk or environment variable	Site-wide RMI compiler options.

For example, there are several options available to enable on Java debugging support when compiling Java code:

- Set `JAVAC_OPT=-g` in the module's Makefile, to affect all future compiles for just that module.
- Execute the command `make compile JAVAC_OPT=-g` to affect just the immediate compiles (this can be combined with the clean or force targets for greater effect).

- Set `LOCAL_JAVAC_OPT=-g` in config/LocalConfig.mk (or an environment variable) to affect compilation in all modules

- Set `SITE_JAVAC_OPT=-g` in config/SiteConfig.mk (or an environment variable) to affect all modules

LOCAL_JAVAC_OPT and SITE_JAVAC_OPT would appear to be redundant, but they have different intents. LOCAL_JAVAC_OPT is for setting options specific to the local developer's workstation, whereas options in SITE_JAVAC_OPT will be shared between developers on a team and should not have any local workstation dependencies.

Chapter

4

Future development of the JBE

Like Tapestry, the JBE is an ongoing effort. As new situations arise, it is extended to meet the need.

Future plans include:

- Support for EARs (Enterprise Application Archives).
- Support for more application servers.
- Support for derived source --- that is, compling Java code generated by tools (such as the IDL compiler)
- JAR signing
- Support for Solaris (in progress) and Linux platforms, and for IBM JDKs.
- Easy way to specify the JDK for a particular Module, with a default JDK when not otherwise specified.