Build System Guide

Contents

Chapter 1

Build System Guide

1.1 Prerequisite

It is recommended that you follow the ***Installation Guide*** steps before you start to use the build system. The installation steps will provide the run-time dependencies that will allow the build system to function. The installation steps will also configure a default directory structure that is required by IVY and other parts of the build system.

1.2 Build System

In order to get up and running quickly, the Spring-Build system has been used as a template and is provides all the functionality that is required from a build system. It is also proven and works well. So we have adopted it and use it to drive our prototype.

Some small modifications have been done to the Spring-Build directories to add additional support to construct template projects and setups for PAR and Bundle projects. This however is minor and all credit goes to the Spring Framework guys for the build system.

An outline of the relevant directory structure is shown below:

```
tinos ..... TINO_HOME - git repository root
└─ build
   ├─ spring-build
   │  └─ scripts ..... Original Template Generator
   ├─ scripts ..... Modified Variant from spring-build
   │  └─ project-gen
   │     ├─ creator.rb ..... modified
   │     ├─ options.rb ..... modified
   │     └─ templates
   │        ├─ bundle ..... added - refer to tinos folder
   │        └─ par ..... added -refer to tinos folder
   └─ tinos ..... Contains Templates
```

Figure 1.1: Build Configuration/Template Layout

1.3 Template Generator

The template generator script is in the *"build/scripts"* folder. This script and the build scripts in general have more functionality than is currently need within the Prototype environment. So only the relevant options and project types needed as described.

1.3.1 PAR Project Template (Obselete)

Note : PAR is now obselete, PLAN is the preferred option

The first order of the day is to create a PAR project template. Once this is created you can add the bundle project templates to it as required. In order to create a PAR template use the following command format:

Format ruby create -n *Name* -t *TargetDirectory* -o *OrganizationString* -a *ProjectTemplate*

Example ruby create -n **sample** -t **/home/user/workspace** -o **org.pouzinsociety.sample** -a **par**

The resulting project structure created looks like the following:

```
/target-directory
└─ prototype
   ├─ dist ..... distribution files
   ├─ par-provided ..... PAR file
   ├─ projects
   │  ├─ spring-build ..... Core Build Files
   │  ├─ build-prototype ..... PAR specific build overides
   │  ├─ org.pouzinsociety.sample ..... PAR Project Area
   │  │  ├─ build.xml ..... Update to reflect bundles to included in PAR
   │  │  └─ META-INF
   │  │     └─ MANIFEST.MF ..... PAR Description
   │  ├─ build.properties
   │  ├─ build.versions ..... Update to reflect required versions of bundles
   │  └─ readme.txt ..... Update to reflect PAR specific information
   └─ readme.txt ..... Update to reflect PAR distribution information
```

Figure 1.2: Template PAR Structure

1.3.2 Bundle Project Template

Note: This is the most commonly used template

A bundle template when invoked will create a bundle directory and an associated osgi integration test bundle. These project structures require that they be placed within a PAR/PLAN project in order to interact correctly with the build system.

The command required to create a bundle template is invoked in the following format:

Format ruby create -n *Name* -t *TargetDirectory* -o *OrganizationString* -a *ProjectTemplate* -p *ParProject-Name*

Example ruby create -n **pouzinsociety-sample-bundle** -t **/home/user/workspace/sample/projects** -o **org.pouzinsociety.sample.bundle** -a **bundle** -p **prototype**

The resulting project structure created looks like the following:

```
/sample
├── dist ..... distribution files
├── par-provided ..... PAR file
└── projects
    ├── spring-build ..... Core Build Files
    ├── build-prototype ..... PAR specific build overides
    ├── org.pouzinsociety.sample ..... PAR Project Area
    ├── org.pouzinsociety.sample.bundle ..... ADDED: Bundle
    ├── org.pouzinsociety.sample.bundle.integration.test . ADDED: Integration
    │   Test Bundle
    ├── build.properties
    ├── build.versions ..... Update to reflect required versions of bundles
    └── readme.txt ..... Update to reflect PAR specific information
└── readme.txt ..... Update to reflect PAR distribution information
```

Bundle Project Layout

The project structure of a bundle will follow the format below. This is generated automatically by the templates and will allow immediate integration with the build system. The is the general layout - it is modified slighty in the integration.test bundle, in that the test bundle has only code within the test folder of the project.

```
/bundle_root
├── src
│   ├── main ..................................................... application code tree
│   │   ├── java ..................................... java code  organised as packages
│   │   │   └── org
│   │   │       └── pouzinsociety
│   │   │           └── sample
│   │   │               └── bundle
│   │   │                   ├── BundleInterface.java ..... Java Interface Definition
│   │   │                   └── impl
│   │   │                       └── BundleInterfaceImpl.java .. Java Interface Implementation
│   │   └── resources
│   │       └── META-INF
│   │           └── MANIFEST.MF ..... PackageBundle Specifications
│   │               └── spring ..... Spring Specific OSGi Configuration
│   │                   ├── module-context.xml
│   │                   └── osgi-context.xml
│   └── test ..................................................... test code tree
│       ├── java ..................................... java test code  organised as packages
│       │   └── org
│       │       └── pouzinsociety
│       │           └── sample
│       │               └── bundle
│       │                   └── impl
│       │                       └── BundleServiceImplTests.java ..... Java JUnit Tests
│       └── resources ..... minimum resource configuration supplied
│           └── log4j.properties ..... Logging configuration
├── ivy.xml ..... build dependencies
└── build.xml ..... build system integration
```

Figure 1.3: Template Bundle Project Layout

```
/bundle_integration_test_root
├── src
│   ├── main ..... application code tree
│   │   ├── java ..... java code organised as packages
│   │   └── resources
│   └── test ..... test code tree
│       ├── java ..... java test code organised as packages
│       │   └── org
│       │       └── pouzinsociety
│       │           └── sample
│       │               └── bundle
│       │                   └── integration
│       │                       └── test ..... Integration Test Package
│       │                           ├── boot-bundles.properties .... Default Bundle Set to load
│       │                           ├── IvyPackagedArtifactFinder.java ..... Integration with
│       │                           │   build dependencies
│       │                           └── LocalFileSystemIvyRepository.java. Integration with
│       │                               build dependencies .11 SimpleBundleTest.java . OSGI
│       │                               Integration Tests
│       └── resources ..... minimum resource configuration supplied
│           └── log4j.properties ..... Logging configuration
├── ivy.xml ..... build dependencies
└── build.xml ..... build system integration
```

Figure 1.4: Template Integration Test Bundle Project Layout

1.4 Building From Code

The following covers the basics of getting up and running with the build system from the command line.

1.4.1 Bundles

The following ant commands are available and applicable to all bundle projects within a PAR project environment. This is quick start guide - further build system configuration is discussed later.

Clean Clean out any build artifacts and bring the project to a clean state.

```
$> ant clean
```

Eclipse Generate a .classpath file to allow eclipse integration. This will link the project dependencies specified in the ivy.xml file to the actual jars (ivy cache) into the eclipse environment.

```
$> ant eclipse
```

Build This will build the bundle. As a side effect the bundle is copied into the ivy-cache so that it can be used by other bundles that are being built. A target folder is also produced in the root of the project and the build artifacts are placed within it.

```
$> ant jar
```

```
target ..... build artifacts
├── artifacts
│   ├── ''Bundle Name''.jar ..... bundle
│   ├── ''Bundle Name''-sources.jar ..... bundle source code
│   └── ivy.xml ..... ivy dependencies for bundle
└── classes ..... class files to generate bundle
    └── xxxx ..... class files in organisation directory structure
```

Figure 1.5: Target folder after "ant jar" command

Test This will build the bundle if required. And it will then execute any test that are present within the project outputting the results in the target/test-results folder at the project root.

```
$> ant test
```

```
target ..... build artifacts
├── artifacts
│   ├── ''Bundle Name''.jar ..... bundle
│   ├── ''Bundle Name''-sources.jar ..... bundle source code
│   └── ivy.xml ..... ivy dependencies for bundle
├── classes ..... class files to generate bunlde
│   └── xxxx ..... class file in organisation directory structure
├── test-classes ..... class files to generate bundle
│   └── xxxx ..... class files in organisation directory structure
└── test-results ..... output from testing
    ├── html ..... in HTML format
    │   └── xxxx
    └── xml ..... in XML format
        └── xxxx
```

Figure 1.6: Target folder after "ant test" command

Publish This will publish the jar (bundle), source code, SHA keys for this project into your local ivy repository. The published bundle can then be stored in your local source/build storage system.

```
$> ant publish-ivy
```

1.4.2 PAR (Obselete)

The following ant commands are available and applicable to the higher level PAR project which wraps your bundle projects. This is quick start guide - further build system configuration is discussed later.

Clean Clean out any build artifacts and bring the project to a clean state. This will also recurse throughout any included bundles for the PAR project. Integration test bundles although present in the project are not included within the PAR build scripts.

```
$> ant clean
```

Build This will build the PAR. This will recurse throughout any included bundles building them and collecting the artifacts for inclusion into the PAR. A target folder is produced and the build artifacts are placed within it. The most noteable is the PAR file itself.

```
$> ant jar
```

```
target ..... build artifacts
├─ artifacts
│  └─ ``Par Name``.par ..... PAR file
└─ par-expanded ..... Bundles within the PAR
   └─ ''Bundle Name''-version.jar ..... Repeat for all bundles
```

Figure 1.7: Target folder after "ant jar" command

Collect This command will iterate through the included bundles picking up any bundle or library that should also be included for distribution with the PAR. See the individual build.xml files for information on how to trigger this collection of dependencies. Also see the target/par-provided folder for any any artifacts collected.

```
$> ant collect-provided
```

```
target ..... build artifacts
├─ artifacts
│  └─ ``Par Name``.par ..... PAR file
├─ par-expanded ..... Bundles within the PAR
│  └─ ''Bundle Name''-version.jar ..... Repeat for all bundles
└─ par-provided ..... Collected dependencies
   ├─ bundles
   │  └─ xxxx ..... Bundles collected
   └─ libraries
      └─ xxxx ..... Libraries collected
```

Figure 1.8: Target folder after "ant collect-provided" command

Package This command will package the par for distribution. A new folder within the target folder is created where the artifacts are placed for distribution.

```
$> ant package
```

```
target ..... build artifacts
├── artifacts
│   └── ``Par Name``.par ..... PAR file
├── par-expanded ..... Bundles within the PAR
│   └── ''Bundle Name''-version.jar ..... Repeat for all bundles
├── par-provided ..... Collected dependencies
│   ├── bundles
│   │   └── xxxx ..... Bundles collected
│   └── libraries
│       └── xxxx ..... Libraries collected
└── package-expanded ..... Distribution Package
    └── ``Par Name''-version ..... Version folder
        ├── dist
        │   └── ``Par Name''-version.par ..... PAR File
        └── par-provided ..... Collected dependencies
            ├── bundles
            │   └── xxxx ..... Bundles collected
            └── libraries
                └── xxxx ..... Libraries collected
```

Figure 1.9: Target folder after "ant package" command

1.5 Build Configuration

1.5.1 projects/build.properties

This file contains some key properties for the configuration of the build system.

```
version=1.0.0
release.type=integration
javadoc.exclude.package.names=**/internal/**,**/internal
natural.name=TINOS-ALL
project.name=TINOS Components
project.key=TINOS
source.version=1.5
target.version=1.5
findbugs.enforce=true
ivy.settings.file=${basedir}/../ivysettings.xml
test.vm.args= -Xmx1024M -XX:MaxPermSize=512M -XX:+HeapDumpOnOutOfMemoryError
```

Figure 1.10: build.properties

1.5.2 projects/build.versions

This file contains the version numbers of all dependencies for all sub-bundle projects within the PLAM/PAR. This is a central location for this information instead of having to update all bundles in the event of bumping a version number. This file is used in conjunction with ivy.xml (dependencies) in every bundle project.

The current provided list contains the minimum set of dependencies to allow the templates as generated work. Developers can change these as required (add, update, remove).

```
# Compile
org.springframework.spring=3.0.0.RELEASE
org.springframework.osgi=1.1.3.RELEASE
org.springframework.integration=1.0.3.RELEASE
org.eclipse.osgi=3.5.0.v20081201-1815
org.apache.commons.logging=1.1.1
org.objectweb.asm=2.2.3
org.aspectj=1.6.3.RELEASE
org.slf4j=1.5.0
org.apache.log4j=1.2.15
org.aopalliance=1.0.0
org.jnode.net=1.0.0
org.jivesoftware.smack=3.1.0
org.jivesoftware.smackx=3.1.0

# Test
org.antlr=2.7.6
org.junit=4.4.0

# Web Interfaces to JNodes (yes - they need the world)
com.sun.facelets=1.1.14
javax.el=1.0.0
net.sourceforge.cglib=2.1.3
org.antlr=2.7.6
org.apache.commons.collections=3.2.0
org.apache.commons.dbcp=1.2.2.osgi
org.apache.commons.logging=1.1.1
org.apache.myfaces=1.2.2
org.dom4j=1.6.1
org.jboss.el=2.0.0.GA
org.jboss.javassist=3.3.0.ga
org.junit=4.4.0
org.springframework.security=2.0.4.A
org.springframework.webflow=2.0.5.RELEASE
javax.el=1.0.0
javax.transaction=1.1.0
```

1.5.3 bundle_project/ivy.xml

This is the dependency file for a bundle. All dependencies for a bundle should be included in this file in order for the build system to resolve them. This file is closely tied with the build.versions file. It is also the file that is used to update the eclipse .classpath file via the "ant eclipse" command.

Any update to this file or the "build.versions" file should be followed by an update of your .classpath file via "ant eclipse".

```
<dependencies>
<!--
 # FIX-ME: Added a few dependencies to get the ball rolling.
 #
 # If you want a bundle to appear in the final "package" add "provided" to the
 # conf e.g conf="compile->runtime" -> conf="provided->runtime"". A conf of
 # "provided->runtime" simply means the server will provide this
 # and do not package it.
 #
 # You can check the <Virgo Web Server>/repositoryr to see what bundles are
 # already installed - this can change as you can download and install bundles
 # into the server via the "MANIFEST.MF" dependencies window in eclipse.
 #
 #   ext => Default Virgo Extension bundles
 #   usr => Downloaded via Eclipse / Added by the User.
 #
-->
<!-- Spring Framework -->
   <dependency org="org.springframework"
name="org.springframework.spring-library"
rev="${org.springframework.spring}" conf="provided->runtime" />
<!-- Logging -->
   <dependency org="org.apache.commons"
name="com.springsource.org.apache.commons.logging"
rev="${org.apache.commons.logging}" conf="provided->runtime" />
<!-- Tests -->
   <dependency org="org.antlr"
name="com.springsource.antlr" rev="${org.antlr}" conf="test->runtime" />
   <dependency org="org.junit"
name="com.springsource.org.junit" rev="${org.junit}" conf="test->runtime" />
</dependencies>
```

1.5.4 spring-build/common/common.properties

A warning to begin with:

DO NOT EDIT THIS FILE UNLESS YOU KNOW WHAT YOU ARE DOING!

The following properties demonstrate how the version strings of the artifacts are generated. During most of the development cycle it will use a timestamp as part of the version string. This

however makes it very hard to target a version of the bundle during the integration testing phase and somewhat in deployment.

So it is easier to uncomment the second line and just use the version number from the MANIFEST.MF file. Specifically the property "Bundle-Version" when building and publishing a bundle into the repository. Just don't forget to bump the "Bundle-Version" as you iterate through official releases. Also remember to comment this line again after you generate the release.

```
build.stamp=BUILD-${timestamp}
# For regular builds
bundle.version=${version}.${build.stamp}
# For Release
#bundle.version=${version}
```

1.5.5 TINOS_HOME/projects/build-all

This is a special build folder within the projects folder. It performs much of the function that a PAR project used to do. Within this folder, you add all the bundle and plan projects you want to be built for the overall TINOS project.

The build.xml file within this folder will need to be updated as you add more bundles to TINOS. The format is as shown below - simply add new pathelement entries as required for the relevant bundle /plan projects. The order of entries is important as this show up inter-project build dependencies.

The template will place a default entry such as the one below for you.

```
<path id="bundles">
<!--
 # FIX-ME : Add the wrapped bundles here, sample provided
 #          Add a new entry for each bundle.
 -->
        <pathelement location="../org.pouzinsociety.prototype.bundle_a" />
</path>
```

1.5.6 General Tips

Search for the string "FIX-ME" within files to check if you need to update files. The core files that require changes are listed above but there are some more changes needed as you change code from the templates. These areas of change are flagged with the "FIX-ME" prefix where possible.

1.6 Ivy

Ivy is a tool for managing (recording, tracking, resolving and reporting) project dependencies. It is characterized by the following:

1 Flexibility and configurability - Ivy is essentially process agnostic and is not tied to any methodology or structure. Instead it provides the necessary flexibility and configurability to be adapted to a broad range of dependency management and build processes.

2 Tight integration with Apache Ant - while available as a standalone tool, Ivy works particularly well with Apache Ant providing a number of powerful Ant tasks ranging from dependency resolution to dependency reporting and publication.

Ivy is open source and released under a very permissive Apache License.

Ivy has a lot of powerful Features, the most popular and useful being its flexibility, integration with ant, and its strong transitive dependencies management engine.

The transitive dependencies management is a feature which let you get dependencies of your dependencies, transitively. In order to address this problematic ivy needs to find metadata about your modules, usually in an ivy file. To find these metadata and your dependencies artifacts (usually jars), Ivy can be configured to use a lot of different repositories.

1.6.1 Why Ivy

Why not. All dependency management options have problems and the main players are *maven* and *ivy*. In other works, I got ivy working quicker and it has not broken so far.

1.6.2 Ivy Cache

An ivy cache is composed of two different parts:

repository cache The repository cache is where Ivy stores data downloaded from module repositories, along with some meta information. This part of the cache can be shared if you use an ad hoc lock strategy.

resolution cache This part of the cache is used to store resolution data, which is used by Ivy to reuse the results of a resolve process. This part of the cache is overwritten each time a new resolve is performed, and should never be used by multiple processes at the same time.

1.6.3 Ivy Repository

A repository in Ivy is a distribution site location where Ivy is able to find your required modules' artifacts and descriptors (i.e. Ivy files in most cases).

1.6.4 Assumptions

The template generator assumes you have an ivy-cache and an ivy-local-repository available for the build process. It assumes these are located in a folder called ivy in the users home directory. You can change these locations using the *build.properties* file.

By default, an empty local repository can be used. As you build bundles and execute the "ant publish-ivy" commands, ivy will add you bundle to the local repository with the correct structure and files. The local repository is primarily a local storage area for your home-grown bundles.

```
tinos ..... TINOS_HOME
 └─ repository ..... Repository Storage Area
    ├─ ivy-cache ..... Ivy Cache - Ivy will create this automatically
    ├─ downloaded ..... Local File System Repository
    │  └─ ''BundleGroupName String''
    │     └─ ''Organisation String''
    │        └─ ''Version String''
    │           ├─ ''BundleName-version''.jar ..... Bundle
    │           ├─ ''BundleName-version''.jar.sha1 ..... Security
    │           ├─ ''BundleName-sources-version''.jar ..... Bundle Source
    │           ├─ ''BundleName-sources-version''.jar.sha1 ..... Security
    │           └─ ivy-''version''.xml ..... Ivy dependencies for this version
```