JBoss Development Process Guide

2004, Ivelin Ivanov, Ryan Campbell, Pushkala Iyer

Table of Contents

Preface

JBoss does not follow to the letter any of the established development methodologies. JBoss borrows ideas, learns from experience and continuously evolves and adapts its process to the dynamics of a largely distributed, highly motivated, and talented team.

This document explains the background and walks through the tools and procedures that are currently used by JBoss for project management and quality assurance.

1

Overview

1.1. Background

The JBoss development process reflects the company *core values*,which incorporate the spirit of open source, individuality, creativity, hard work and dedication. The commitment to technology and innovation comes first, after which decisions can be based on business, then competition.

A typical JBoss project enjoys active support by the open source community. The ongoing collaboration within the community, naturally validates the viability of the project and promotes practical innovation. This process leads to a wide grassroots adoption of the technology in enterprise Java applications.

While community support is the key factor for the widespread adoption of JBoss technology, there are other factors that lead to its successful commercialization, such as return on investment (ROI) and total cost of ownership (TOC). They require JBoss to offer products with strong brand, long term viability, and low maintenance costs. Companies that rely on JBoss products should be able to easily hire expertise on demand or educate existing engineering resources. They should also feel comfortable that the market share and lifespan of these products will protect their investments in the long run.

The dilemma posed to the JBoss development process is how to enable a sound business model around sustainable and supportable products, without disrupting the fast pased technology innovation. The traditional process of gathering requirements from top customers, analysing, architecting, scheduling and building software does not work in the JBoss realm. It ignores the community element and conflicts with the principle that technology comes first.

On the other hand great technology does not necessarily lend itself to commercialization directly. Professional marketing research is needed to effectively determine the best shape and form to position a technology. It is frequently placed as a building block of a broader offering targeted at identified market segments. Ideally it should be possible to "package" technology into products on demand.

To allow harmony between business and technology, JBoss defines a simple and effective interface between the two. The interface is introduced in the form of integration milestones. At certain points of time, pre-announced well in advance, stable versions of JBoss projects are selected, integrated, tested and benchmarked in a coordinated effort. The result is an integrated Middleware stack that is referred to as the JBoss Enterprise Middleware System (JEMS). JEMS is not a single product but a technology stack that can be used for packaging marketable products.

While core JBoss projects evolve and release versions at their own pace, stable versions are regularly merged into JEMS to fuel its continuous growth as a comprehensive platform. Major JEMS versions are spaced out at about 12 months with intermediate milestones on a quarterly basis. This allows sufficient time for the industry to absorb the new features and build a self-supporting ecosystem.

For example the JEMS 5.0 milestones were announced in December of 2004. The first milestone - JEMS 5.0 Alpha is targeted for Q1Y05. It will introduce a standards based POJO Container, which allows a simplified programming

model based on the new EJB 3 standard APIs. JBoss Cache will be one of the projects integrated in JEMS 5 Alpha. JBossCache has three pulblic releases planned in the same timeframe - 1.2.1, 1.2.2 and 1.3. Only one of them will be picked for integration in JEMS 5 Alpha.

The second milestone - JEMS 5.0 Beta is targeted for Q2Y05 and will be the first attempt at a complete integration of core JBoss projects on top of a new JBoss MicroContainer. The JEMS 5.0 Final milestone in Q3Y05 will complete the development cycle by presenting an enterprise grade middleware stack, which is certified and fully supported by JBoss and its authorized partners. Any subset of JEMS 5 could be extracted and deployed in production environment, because its components will have been thoroughly tested to work together and perform well.

1.2. JEMS integration milestones

The JEMS milestones have minimal impact on the progress of the individual JBoss projects. Their purpose is to set expectations for the timing of the integration phases. The process itself is controlled and executed by the QA team in collaboration with each project development team. There are several phases in the development cycle between JEMS milestones.

1. *Feature planning.* This is the first phase in a JEMS integration cycle and normally lasts a few weeks. It is an open planning excersize between QA and project leads about the features that should be available in the next JEMS version (e.g. JEMS 5.0 Alpha). During this phase each project lead proposes the version of their project (e.g. JBoss Remoting 1.0) that should be integrated in JEMS and announces its key features. QA will have minimal input on the feature planning, but will have a say whether or not an implementation has acceptable quality when it is released. Cross project dependencies are identified throughout the discussion and they can result in additional feature requests for a given project version. Ideally the discussion ends with a commonly agreed matrix of projects versions, features and interdependencies. Differences are normally mitigated by the QA team but issues could escalate higher in the management chain. The QA team also sets the acceptance criteria for each project version and the latest date by which the targeted project version should be handed over for integration. If a project version is not released by this date or it does not meet the acceptance criteria, QA has the option to drop the project version and use an older version or find another alternative to minimize the negative impact on JEMS overall.

2. *Scheduling.* Based on the project release dates and interdependencies, the QA team prepares estimates for the amount of work required for testing, benchmarking and documenting the integration between participating projects. Next, the QA team builds out a task schedule that validates whether the planned JEMS release date from phase one is realistic. Individual tasks in the schedule are sized 2-4 days to allow enough level of detail that would reveal ommissions made during the first phase. If adjustments need to be made the QA team opens a brief discussion with the project leads to decide whether some features need to be dropped or the deadlines can be moved out within reason.

3. *Accepting project versions for integration.* At this stage all agreed upon project versions are handed over to QA for verification. Each one is examined to verify if it passes the acceptance criteria set forth early in the iteration. The process can take up to 2 weeks to allow for minor fixes. Acceptance criteria will vary depending on how close the JEMS milestone is to a production release. Earlier milestones will have less stringent requirements on documentation and training material. Projects that cannot pass the verification are removed from the JEMS milestone. In this case the QA team will find a fallback solution, which potentially includes using an older certified version of the project in question. Dependent projects will have to readjust accordingly.

4. *Writing integration test plans.* For the stack of project versions that passed the acceptance criteria, QA develops a more comrehensive suite of integration tests. It covers complex scenarios across multiple projects that closely resemble realistic usage patterns. Tests that fail are addressed either by the corresponding project developers or QA. It is preferable for project teams to be available on a short notice for fixing bugs and quickly releasing minor incremental versions to be merged back in the JEMS stack. Versions contributed to JEMS at this phase should only include fixes to issues raised or confirmed by QA. These versions should NOT be based on the latest development code branch. In cases when bug fixes are not provided in a timely manner or there are risks of missing the JEMS deadlines, QA has the option to find an alternative solution. This includes reverting back to an earlier certified project version.

5. *Benchmarking.* After the functionality of the projects in JEMS is confirmed, QA executes a number of benchmarking plans. They are used to compare the performance of the new version to the previous one and also establish baseline metrics for new features that will be tested again in future versions. Limited code modification and configuration changes can be made to tune the JEMS stack for better performance and reliability.

6. *Documenting.* Basic end user documentation should already be available with each project at the time its handed over to QA. However additional documentation can be added such as integration blueprints, configuration scenarios, tuning tips, performance metrics and others.

7. *Certification.* When all testsuites pass and the best performance numbers are achieved within the time constraints, QA certifies an internal JEMS release for several main platforms (e.g. Linux/Intel, Windows/Intel). This internal release becomes available for a limited time to interested JBoss partners who are interested to certify on their specific platforms (e.g. HP/UX, Solaris/Sparc). Finally QA cuts off and publishes a matrix of platforms where the JEMS versions is certified by JBoss or an authorized partner. Other certified platforms can be added at a later point. This concludes the JEMS iteration and from this point on, various products can be packaged and marketed based on the certified JEMS components.

2

JBoss Issue Tracking

JBoss utilizes JIRA for product lifecycle tracking. It is used during the requirements gathering, task scheduling, QA and maintenance stages of a product lifespan.

JIRA is an overall excellent issue tracking system. However as of version 3.0 Enterprise it does not offer sophisticated project planning and tracking functionality such as calculating critical path, reflecting task dependencies, resolving scheduling conflicts, and resource calendar. These shortcoming can be partially mitigated by splitting development into short iterations (1-2 months) in order to reactively manage diviations from the base line schedule.

2.1. Creating a new Project

To begin the development of a new JBoss project, it needs to be registered in the project management system - JIRA. To do that you need to contact a JIRA Administrator [http://jira.jboss.com/jira/secure/Administrators.jspa].

Once the project is created, you will need to create a version label for the first (or next) production release. Under this release there will be several "blocking" tasks such as requirements gathering, coding, documentation, training material and QA. As a best practice issues should be only closed by their original reporter.

In addition to the production release there you will need to create versions for the intermediate releases at the end of each iteration. See the project named "README 1st - JBoss Project Template" for a starting point.

2.2. Creating Release Notes

The Release Notes for a product version are generated automatically by the Project Management System (JIRA) and additionally edited manually when necessary.

To mazimize the value of the automatically generated Release Notes and minimize the manual work, the following giudelines are in place:

1. Use concise but descriptive issue names
2. Open the right kind of issue.

2.2.1. Adding Issues to Release Notes

In order for an issue to appear in the release notes for a given version it needs to have its field "Fix Version/s" set to the given version. Usually an issue affects only one particular version and it is fixed within that version. Sometimes however an issue affects multiple versions and it is addressed for each one of them. In the latter case the "Fix Version/s" fields comes handy.

2.2.2. Generating Release Notes

1. Go to the project home page. For example the Portal project [http://jira.jboss.com/jira/browse/JBTPL].
2. Click on *Release Notes*
3. Pick the version you are intereted in the *Please Select Version:* drop down menu.
4. Select whether you want HTML or Plain Text format in the *Please Select Style:* menu. The HTML version provides links next to each issue in the release notes report that can be followed for more details. The Text version places the issue ID (e.g. JBTPL-11) next to the release note, which can be also used to obtain issue details.
5. Click Create.
6. You should see something similar to this [http://jira.jboss.com/jira/secure/ReleaseNote.jspa?version=10014&styleName=Html&projectId=10010&Create=Create].

2.3. Issues

2.3.1. Types

1. *Feature Request* - A new feature of the product, which has yet to be developed. Feature requests appear near the top of release notes. Blocker and Critical priorities mark the features that are appropriate to advertise in marketing material such as datasheets and sales presentations.
2. *Patch* - can be used for performance enhancements, code refactoring and other optimization related tasks for existing functionality.
3. *Bug* - a problem which impairs or prevents the functions of the product.
4. *Task* - should be used if none of the other categories seem appropriate.

2.3.2. Priorities

JIRA offers voting mechanism that helps determine the number of people asking for a task as well as who these people are. JBoss Project Leads consult these votes in order to schedule tasks. All other developers in a project coordinate their time and tasks with the project lead. A select number of stakeholders have overriding power for task priorities. The JBoss CTO has the highest authority on development task priorities. When there is ambiguity on task priorities, contact your project lead or development manager.

Possible priorities are:

- *Blocker* - An issue (bug, feature, task) that blocks development and/or testing work, production could not run. An upcoming version that is affected by this issue cannot be released until it's addressed.

- *Critical* - An upcoming version that is affected by this issue cannot be released until it's addressed. A critical bug is one that crashes the application, causes loss of data or severe memory leak.
- *Major* - A request that should be considered seriously but is not a show stopper.
- *Minor* - Minor loss of function, or other problem where easy workaround is present.
- *Optional* - The request should be considered desirable but is not an immediate necessity.
- *Trivial* - Cosmetic problem like misspelt words or misaligned text.

2.3.3. Estimates and Due Dates

Due dates are normally used for scheduling project versions. When entering issues, time estimates should be preferred to due dates. Issue due dates limit the project management software capability to level resources and optimize scheduling.

2.3.4. Affects Checkboxes

To support the updating of release notes and documentation, the Affects field offers several flags when creating or editing an issue.

- *Documentation* - This flag indicates that project documentation (e.g., a reference guide or user guide, etc) requires changes resulting from this issue.
- *Interactive Demo/Tutorial* - Indicates an interactive demo or tutorial requires changes resulting from this issue.
- *Compatibility/Configuration* - Indicates that issue may affect compatibility or configuration with previous releases so they can be highlighted in the release notes overview section.

2.4. Managing Container Projects

Projects such as JBoss Application Server package components from several other projects such as JBoss Cache, Tomcat, JGroups, and Hibernate. To manage the development cycles between these projects the following guidelines apply:

1. A projects that ships as a standalone product has its own entry as a JIRA Project. Examples include JBoss Cache, Hibernate, JBoss jBPM, etc. These projects have independent release cycles.
2. A container project such as JBoss AS that packages other projects has a JIRA component for each one of them. For example the JBoss AS project includes the following components: JTA, JCA, Web Services, Hibernate service, JBoss Cache service, JBoss Web(Tomcat) service. There are two kinds of components:

    a. Components for composing projects that are developed within the container and have release cycles aligned with it (e.g. JTA, JCA)

    b. Components for embedded projects that are integrated within the container, but are also offered stan-

dalone (e.g. Tomcat, Hibernate). These components track the integration tasks for the embedded service (e.g. Tomcat). Typically a release of the container is integrated with a stable version of the standalone project. For example JBoss 4.0.1 embeds Tomcat 5.0.16.

2.5. Project Source Repository and Builds

The source code repository of a container project includes the full source for all composing components. For integrated components, the source repository includes integration source code and stable binaries of the related standalone projects. Building a container from source, compiles the source code for its composing parts as well as integration code, but it does not pull in the source for standalone projects.

2.6. Testsuites

A container testsuite includes the tests for all composing components as well as the integration tests for embedded compoenents. It does not include the tests that are part of the standalone testsuite for an integrated component. For example JBoss AS testsuite covers the HAR deployer, but it does not include tests from the standalone Hibernate project.

2.7. Dependency Tracking with JIRA

Container projects such as JBAS consist of components, some of which are integral to the container (such as CMP, IIOP) and others are based on external projects (MicroContainer, JBossCache).

For each container version and each component based on external project, there should be an integration tasks created in the container project. The task should specify which version of the external project the container component depends on (e.g. JB AS 4.0.1 depends on JBoss Cache 1.2). Both project leads need to be aware and agree on the dependency at the time the integration task is created.

When new issues are created against the dependent project version (JB Cache 1.2) related to the development of the container project version (JB AS 4.0.1), they should be linked to from the integration task. Example: http://jira.jboss.com/jira/browse/JBAS-56

If the dependent project version is released before the container project is (JB Cache released on Dec 10, while JB AS 4.0.1 is not released until Dec 22), there should be a flexible mechanism to accomodate intermediary patches. One option is for the dependent project to maintain a separate branch (JBCache_1_2_JBAS_4_0_1) for the container integration. Another option is for the dependent project to apply patches against its main branch and release minor increments (JB Cache 1.2.0b).

3

Project Structure

As components of the Application Server mature into their own projects, it will become increasingly difficult to manage them as a part of the "jboss" project. They will have their own contributors, releases, components and libraries. Coordinating all of this effort in a single project is unscalable.

All of the existing modules of the AS should be extracted into their own projects. Each project will have its own source repository and will integrate with the AS as an external dependency.

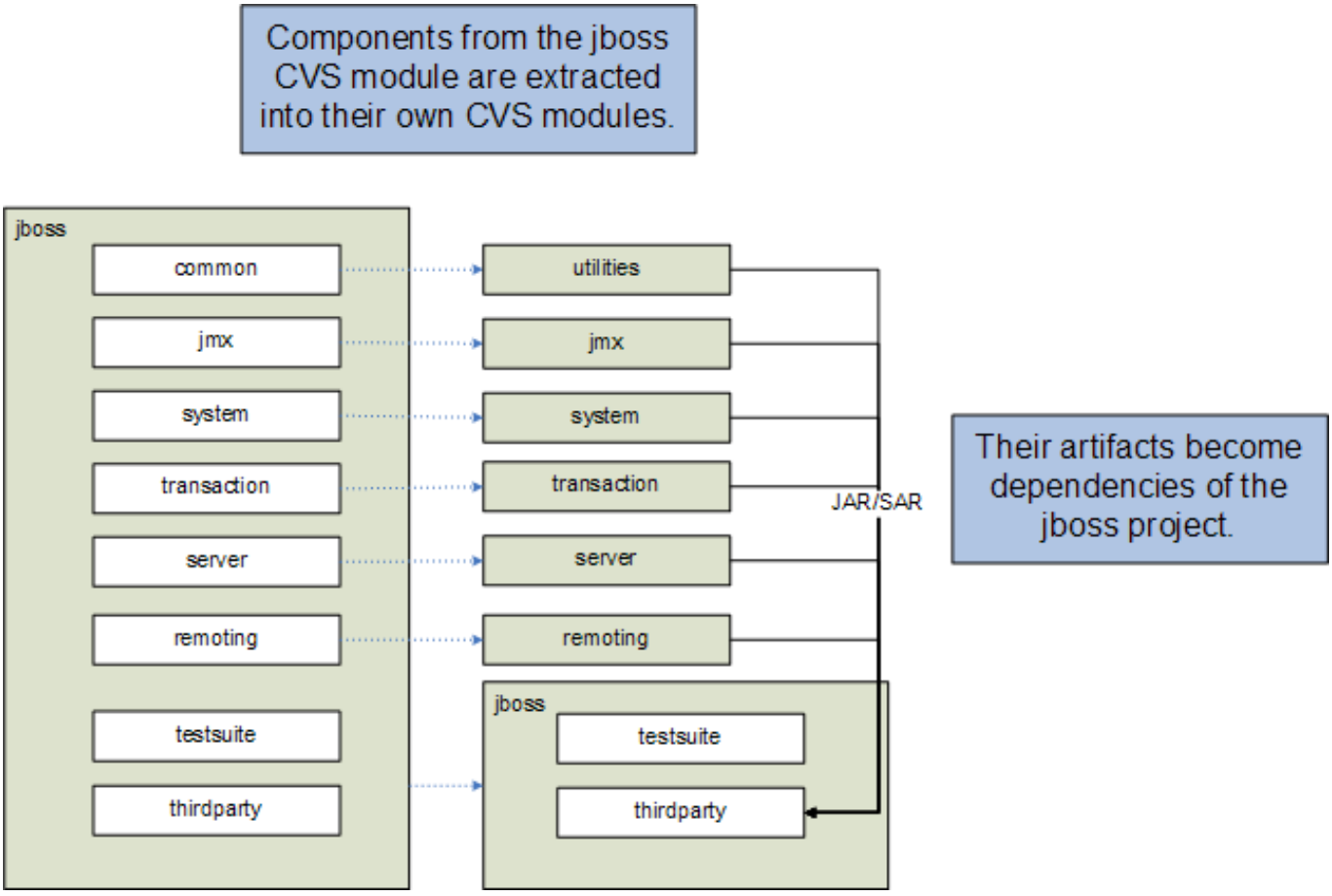


Figure 3.1. Extraction of AS components into their own CVS modules.

Individual projects produce multiple artifacts according to the needs of dependent projects. For instance, JBossCache would produce a standalone artifact for use in lightweight containers. It would also need to produce a SAR with a more extensive dependency set for use in the AS.

The AS project will aggregate its dependencies into multiple distributables (i.e., archived or InstallShield). The AS

project will also include the testsuite for integration tests.

The process for extracting projects from modules can be iterative. However, it needs to start at the top of the dependency tree with the Common module and proceed downwards. For example, in order to extract the Remoting project, all of the dependencies above it must be extracted and made available to the Remoting project as external libraries.

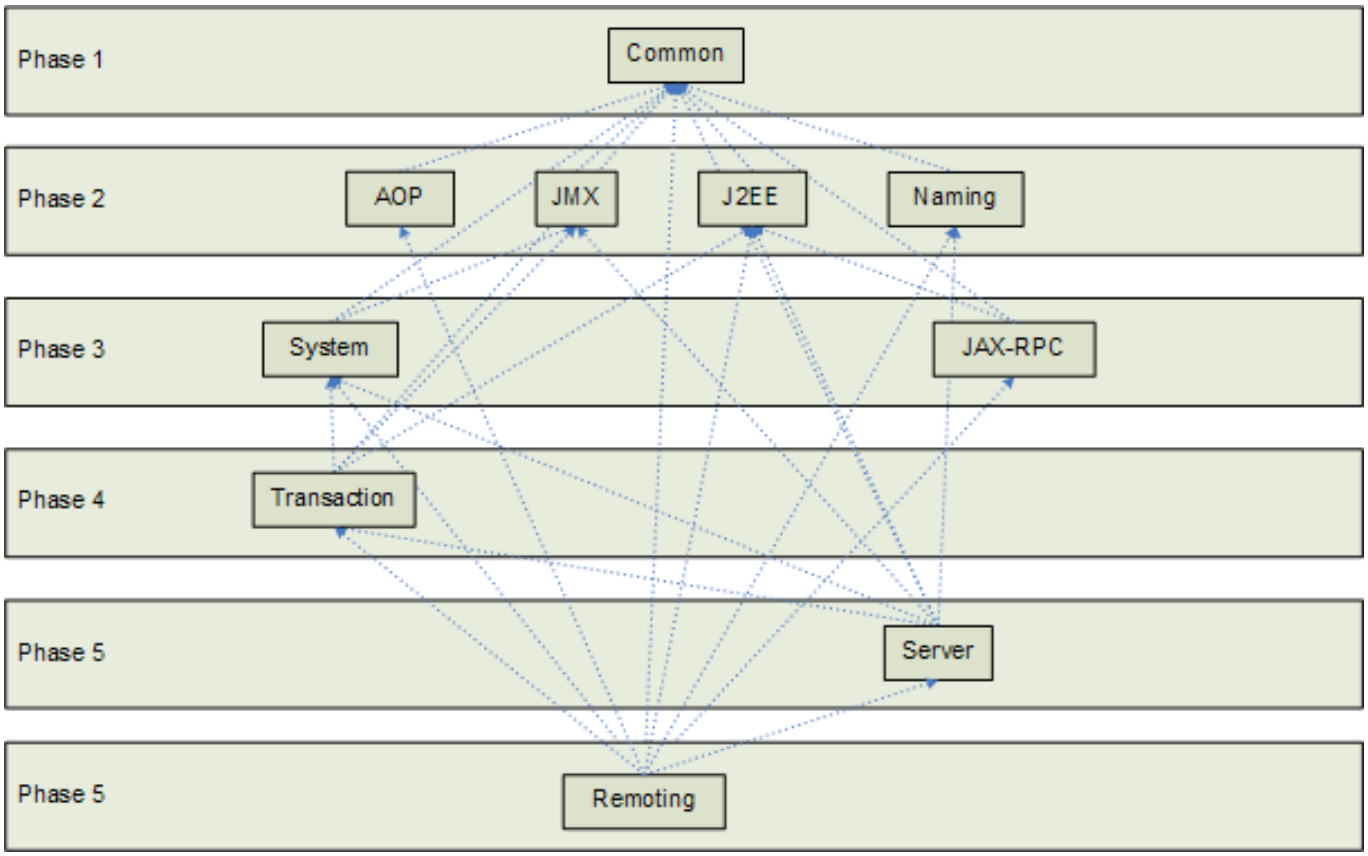


Figure 3.2. Example of Module Extraction Process

This order is necessary to avoid circular dependencies.

4

Build System Requirements

As components of the Application Server mature into their own projects, it will become increasingly difficult to manage them as a part of the "jboss" project. They will have their own contributors, releases, components and libraries. Coordinating all of this effort in a single project is simply not scalable.

The obvious solution is to decouple these "super subprojects" from the main Application Server project. With respect to the application server, they will become external dependencies much like JGroups and Hibernate.

However, splitting the Application Server into multiple projects presents new challenges and requires a fresh approach to the build system. Some of the issues which need addressing:

- Continuous Integration -- Multiple parallel lines of development must be integrated at regular intervals. What was once accomplished by a cvs update and rebuild must now be done by other means.
- Version Alignment -- Projects need the ability to manage their own dependencies. However, the versions of these dependencies must align when projects are aggregated into the Application Server.
- Source vs Binary Tradeoff -- A developer shouldn't have to checkout the source for all dependencies to build a given project. Likewise, it should be possible to checkout a dependency and quickly test modifications to that dependency with the project.

This document will define a Build System which will address the above issues.

4.1. Definitions

Project
: From the perspective of the build system, a project is a directory containing source code which produces one or more artifacts. A project may have dependencies on the artifacts of other projects or third-party libraries.

Component
: A source tree which exists only as a part of another project. For example, JTA is a component of the application server.

Dependency
: A requirement that a library or artifact must be available during compilation of a given project. This term can also refer to the artifact or library itself.

Artifact
: The compiled product of a project. Generally, artifacts are JAR files. The Application Server project will have a dependency on artifacts produced by the MicroCon-

tainer project. An artifact may be a JAR, SAR, or WAR.

Library

A JAR file which is not generated from source. Many projects will have a dependency on the log4j library.

4.2. Goals

4.2.1. Project Decomposition

A primary goal of the new build system is to support the decomposition of the JBoss Application Server source tree into multiple standalone projects. It should be possible to checkout and build any project independently of any container projects. However, the build system should support integrated builds between dependent projects. For instance, it should still be possible to build the application server and all of its subprojects with one invocation.

4.2.2. Dependency Management

The new build system should simplify the way that thirdparty libraries are managed. Specifically, the build system should allow projects to declare which versions of what dependencies they have. The build system should be able to automatically download dependencies from an online repository.

4.2.3. Simplified Build Scripts

Build scripts should be as declarative as possible, while still allowing for customization to an individual project's needs. The inputs and outputs of a component should be clearly documented.

4.2.4. Tool Compatibility

Any new build system needs to be compatible with major IDE's and associated tools used by developers. At the very least, this contstrains project structure and build script implementations. For example, it should be possible to execute a build target from Eclipse.

4.3. Requirements

4.3.1. Declarative Dependencies

Projects should be able to declare dependencies without being concerned with where those dependencies reside, or in what form. Dependency information is also needed for documentation and for use by IDE's, so it should be easy to automatically extract this information for use in other contexts.

A project should only need to declare its direct dependencies. Indirect dependencies should be included automatically.

Dependency information is stored and versioned along with the project source. The dependencies themselves will be resolved by the automatic update feature below.

4.3.2. Automated Updates

Multiple parallel lines of development need to be constantly integrated to tighten the feedback loop. As the number of projects grows, it will be impractical for each developer to update and build each project. The build system should provide the capability for a developer to update dependencies from an online repository.

For example, instead of having to checkout and build the MicroContainer to test if a recent change affects the Application Server, it should be possible to configure the build system to download the most recent build from the online repository.

As illustrated in the figure below, this requires that the continuous build system (cruisecontrol) be enhanced to publish binary artifacts of each component to the online repository. The repository will include both snapshot and released versions of each project's artifacts.

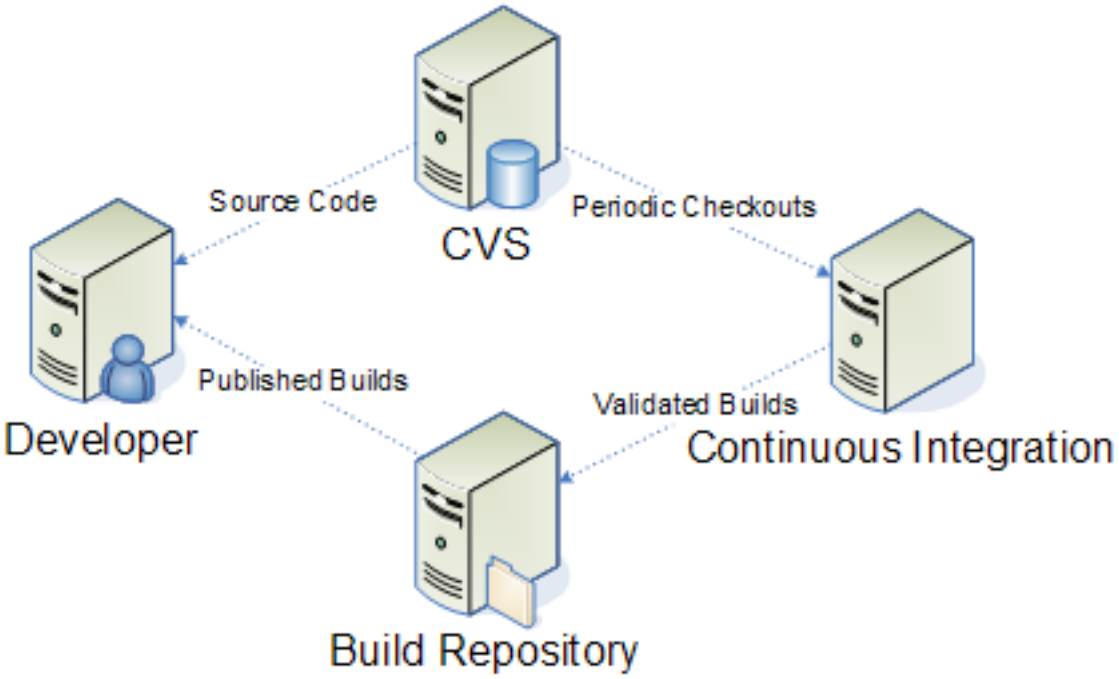


Figure 4.1. Repository Workflow

There will be cases where an updated snapshot dependency will introduce critical bugs which prohibit a downstream developer from moving forward. Therefore, it should also be possible to revert an artifact dependency to a previously working snapshot.

4.3.3. Source Artifact Overrides

It should be possible to work with multiple projects at once. A developer should not be required to copy artifacts by hand when testing the integration between two projects. Therefore, it should be possible to switch between using a binary snapshot of a dependency (from the online repository) and integrating the local build of that project.

Succinctly, it should be possible to use an artifact of a project instead of a the pre-compiled library simply by checking out the project from CVS and configuring the build of the integration project to call the dependent project's build.

The balance between using the source of a dependency and using its pre-built library from the online repository is one of personal choice and circumstance. The build system should be flexible enough to support both modes, and make it painless to switch back and forth.

4.3.4. Dependency Version Reconciliation

Each project depends on a specific version of a dependency. However, when projects are aggregated into an integration project such as the Application Server, they must be compatible with the same version of a library. Since the projects are compiled seperately, there is a risk that when they are combined into a single classpath, there will be runtime LinkageErrors.

Let's take an example of JBossCache and JMS, two projects which both depend on JGroups. Both are built and versioned seperately. A release from each will need to be included in the application server. At this point JBossCache and JMS will need to be compatible with the JGroups library included in the application server.

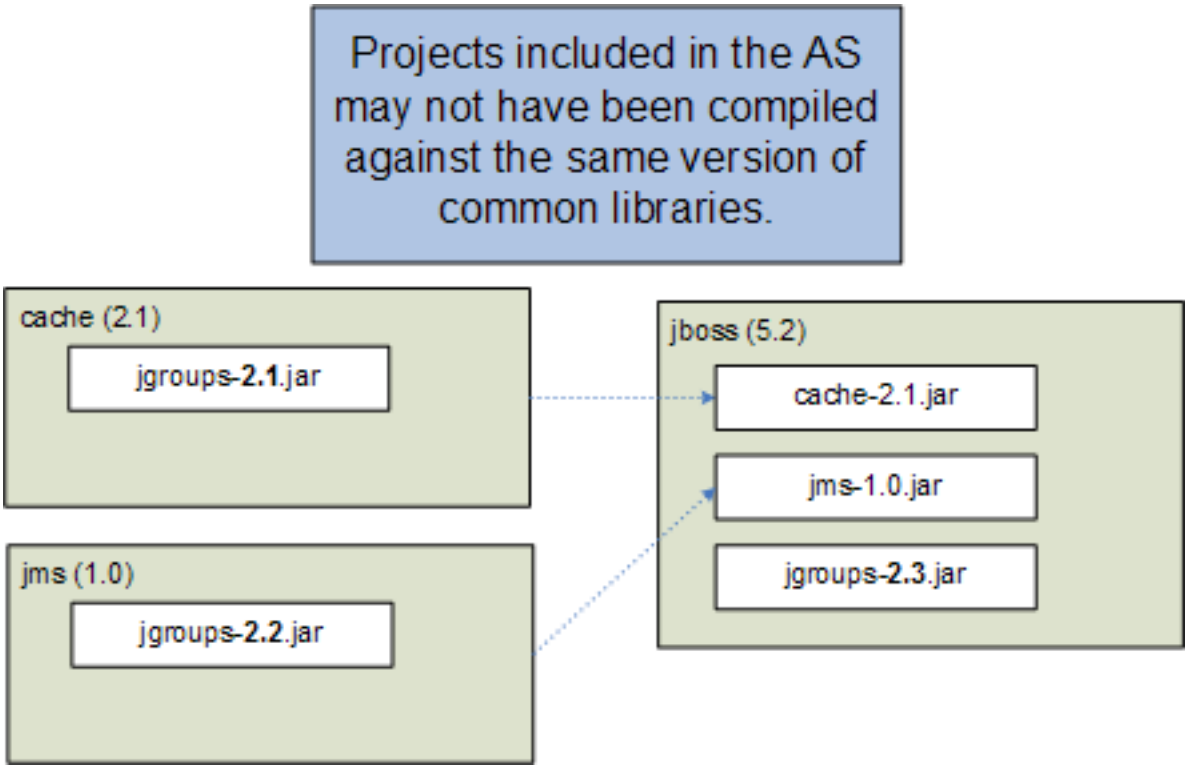


Figure 4.2. Example of Dependency Version Conflict

There is a potential that the JGroups JAR provided by the application server will have binary or functional incompatibilities with JBossCache or JMS because of the version discrepancy. To mitigate this, these projects will need to be continuously integrated with the Application Server to ensure compatibility. This includes compiling and testing these projects *against the library versions dictated by the Application Server*.

Therefore, the build system should support the overriding of dependency versions so that projects can be compiled

and tested with the versions specified by another project. This could be in the form of allowing the library version number to be set as a property. One could then override the properties for all library versions when running a build.

4.3.5. Component Builds

Projects need the ability to modularize source code into components. The build system should provide projects with the ability to integrate component builds and aggregate their artifacts into project releases.

4.3.6. Extensibility

Individual projects need the ability to customize the behavior of the build system to specific needs. The build system should allow for ad-hoc extensions specific to a project to be introduced by clearly defined extension points.

4.3.7. Incremental Builds

The build system should optimize the code, compile and test cycle for the developer. At the very least, this means that dependency checking should be used to ensure that a given invocation of the build performs the least amount of work.

4.3.8. Artifact JDK Version

It should be possible for projects to declare minimum JDK requirements for a given artifact. If the JDK performing the build is at a lower release than an artifact specifies, the artifact will be excluded from the build. For example, a artifact might require JDK 1.5 to build. When the project is built using JDK 1.4, that particular component will not be compiled.

4.3.9. Line-Precise Error Reporting

The build system should, as much as possible, provide line-precise error messages. This should include errors in the build scripts, and misconfigurations in the project structure.

4.4. Use Cases

4.4.1. Intial Project Checkout & Build

A developer wishes to checkout the AS. The initial checkout will not include any libraries or components. Once the build is invoked, the unsatisfied dependencies will be resolved from the online repository.

```
cd $WORK
cvs co jboss-as
:
cd jboss-as
ant build
```

Since this is a clean checkout, none of the libraries will be locally available. The build system will detect the unresolved dependencies and will download them from the repository. Since none of the dependencies (I.E., aop, microcontainer) are available in source form, their artifacts will be downloaded from the repository.

4.4.2. Source Override

After checking out the jboss-as project, the developer wishes to make a change to the jboss-common dependency. To accomplish this, the developer checks out the dependency from CVS into the working directory. When jboss-as is rebuilt, the build for jboss-common will be invoked. The artifact from the jboss-common project will override the jboss-common library which had previously been downloaded from the online repository.

```
cd $WORK
cvs co jboss-common
:
cd jboss-as
ant rebuild
```

If the developer wishes to only build the jboss-common component he can change to that directory and invoke the build. Only the jboss-common project will be built.

```
cd $WORK
cd jboss-comon
ant rebuild
```

4.4.3. JBossCache Example

This use case demonstrates how a developer might test a bug fix in JBossCache with the Application Server. For the purposes of this example, the work directory is empty.

```
cd $WORK
cvs co -r JBoss_Cache_1_1 jboss-cache
cvs co -r JBoss_AS_5_1 jboss-as
cd jboss-as
ant tests
```

In this example, the jboss-as build would see the jboss-cache project in the work directory and would build it before building the AS and then running the testsuite.

4.4.4. Valid Commit

Developer A commits a valid change to jboss-common, CruiseControl detects the modification and performs a clean build of all affected projects. The build succeeds, and the snapshot artifacts are published to the repository. Later, Developer B performs a build of jboss-as without a source copy of jboss-common. The build system downloads the snapshot of the jboss-common jars and successfully builds the project.

4.4.5. Broken Build: Interface Changes

Developer A commits a change to jboss-common which breaks an interface used by jboss-system. Developer A did not build jboss-system against the changed interface and so does not commit any fixes for jboss-system.

Cruisecontrol detects the change in jboss-common and intiates a build of all projects which depend on the given branch of jboss-common. The build fails, and a failure email is issued. No artifacts are published to the repository, so development against the existing snapshot can proceed.

5

Build Reference

This reference guide covers how to use the JBossBuild system.

5.1. Overview and Concepts

JBossBuild is a declarative build system. Instead of having to define each step in the build process, JBossBuild allows a developer to declare the inputs and outputs of a build. JBossBuild then uses these definitions to dynamically generate the Ant targets needed to implement that definition.

JBossBuild is implemented as a set of Ant types and tasks, and target definitions. The types (components, componentdefs, artifacts, etc.) are declared by the developer in the build.xml. These definitions are then combined with the targetdefs in tasks.xml (under tools/etc/jbossbuild) to produce the generated ant targets.

There are two kinds of build definitions: toplevel, and component. The toplevel builds define the components of a release and where the artifacts of each component should be placed in the release. The component builds define how each artifact is built, the sources of those artifacts, and any dependencies of thoses sources.

5.2. Component Build

A component build is made up of two parts: the component info (component-info.xml) and the component definition (build.xml or jbossbuild.xml). The component info is much like a declaration or manifest of the component. It defines what the expected outputs (artifacts) of the components are. The component definition specifies how these artifacts are built from source code.

5.2.1. Component Info Elements Reference

Table 5.1. Component

Name:	component
Purpose:	Declares a project component.
Attributes:	
id	The unique identifier for this component. This should be the same as its directory name in the online repository and in the local directory structure.
module	The CVS module the component source should be checked out from.

version	The version of the component. This version is used when retreiving artifacts from the repository. Artifacts are stored in the repository under the directory [id]/[version].

Table 5.2. Artifact

Name:	artifact
Purpose:	Declares an artifact (jar, war, config file) which is a product of the component build.
Attributes:	
id	The unique identifier for this artifact. The id is the same as the name of the file. This id should be unique across all components in a given build.

Table 5.3. Export

Name:	export
Purpose:	Lists the default artifacts which should be on the classpath when this component is included by another.
Example:	<export> <include input="jnpserver.jar"/> </export>

5.2.2. Component Definition Elements Reference

Table 5.4. Component

Name:	component
Purpose:	Declares a project component.
Attributes:	
id	The unique identifier for this component. This should be the same as its directory name in the online repository and in the local directory structure.
module	The CVS module the component source should be checked out from.
version	The version of the component. This version is used when retreiving artifacts from the repository. Artifacts are stored in the repository under the directory [id]/[version].

5.3. How to Synchronize and Build

You can now partially build jboss-head from the repository with the new build system.

You probably want this in it's own directory:

```
mkdir jboss-dir
cd jboss-dir
```

Then, just check out the toplevel build and the tools module:

```
cvs co jbossas
cvs co tools
```

You will need to set your cvs info in jbossas/local.properties:

```
cvs.prefix=:ext:rcampbell
```

Note, you will need ssh-agent setup to run cvs without entering a password for now. Now you are ready to synchronize and build:

```
ant sychronize
ant build
output/jboss-5.0.0alpha/bin/run.sh -c all
```

The synchronize target will checkout the source components from cvs and download thirdparty components from the repository.

5.4. Tutorial: Anatomy of a Component Build

In this section, we take a component - JBoss Deployment (jboss-head/deployment) and demonstrate how to incorporate it into the JBossAS release. This document assumes you have checked out the AS as outlined here.

5.4.1. Top Level Build

First, we need to add the component to the toplevel build under jbossas/jbossbuild.xml. The ordering of the components is significant; the deployement module must be placed *after* the other source components it depends on (ie, common). The ordering of the components in the file dictates the order the components will be built. So, in this case, we add the component element at the end of the other JBoss components, but before the thirdparty components.

```

```

```
      <!-- ========================================================== -->
      <!-- Deployment                                                 -->
      <!-- ========================================================== -->
      <component id="deployment"
                 module="jboss-deployment"
                 version="5.0-SNAPSHOT">
      </component>
```

At this point, we know that the deployment module will come from the jboss-deployment module in cvs -- represented by the module attribute. We give it the same version as the other components in jboss-head. With this one definition, we have several new targets in our toplevel build:

```
bash-2.05b$ ant -projecthelp | grep deployment
 all.deployment                       Build All for the component deployment
 api.deployment                       Javadoc for the component deployment
 build.deployment                     Build for the component deployment
 clean.deployment                     Clean for the component deployment
 commit.deployment                    Commit for the component deployment
 doc.deployment                       Documentation for the component deployment
 rebuild.deployment                   Synchronize then build for the component deployment
 rebuildall.deployment                Synchronize then build all for the component deployment
 runtest.deployment                   Run tests for the component deployment
 synchronize.after.deployment         After synchronization processing for the component deployment
 synchronize.deployment               Synchronize for the component deployment
 test.deployment                      Build and run the tests for the component deployment
```

These are all dynamically generated by jbossbuild based on the defintion we have provided. At the moment, we are only concerned with the synchronize target since we still don't have the source for this component. So let's see what the synchronize target will do before we try to call it

To see what a target will do before you call it, you can use the "show" target and pass it a property of which target you want to see.

```
bash-2.05b$ ant show -Dshow=synchronize.deployment
Buildfile: build.xml

show:
<!-- Synchronize for the component deployment -->
<target name="synchronize.deployment">
   <mkdir dir="C:\projects\newbuild-jboss\thirdparty\deployment"/>
    <get verbose="true" dest="C:\projects\newbuild-jboss\thirdparty\deployment/component-info.xml"
         usetimestamp="true"
         src="http://cruisecontrol.jboss.com/repository/deployment/5.0-SNAPSHOT/component-info.xml"/>

</target>
```

Whoops! Calling this target will download the component to thirdparty, which is not what we want at this point. In order to get the source for this component, we will want to set a property in the jbossas/synchronize.properties file:

```
checkout.deployment=true
```

```
```

Now, when we show the deployment.synchronize target we see that it intends to pull the source from cvs:

```
bash-2.05b$ ant show -Dshow=synchronize.deployment
Buildfile: build.xml

show:
<!-- Synchronize for the component deployment -->
<target name="synchronize.deployment">
<cvs dest="C:\projects\newbuild-jboss">
  <commandline>
    <argument value="-d"/>
    <argument value=":ext:rcampbell@cvs.forge.jboss.com:/cvsroot/jboss"/>
    <argument value="co"/>
    <argument value="-d"/>
    <argument value="deployment"/>
    <argument value="jboss-deployment"/>
  </commandline>
</cvs>
</target>
```

Ok, so let's go ahead and call this target to checkout the module into our tree (../deployment).

```
bash-2.05b$ ant synchronize.deployment
Buildfile: build.xml

synchronize.deployment:
      [cvs] Using cvs passfile: c:\.cvspass
      [cvs] cvs checkout: Updating deployment
      [cvs] U deployment/.classpath
      [cvs] U deployment/.cvsignore
...
```

We could have also called the toplevel synchronize target if we wanted to update (or checkout) all the other components and thirdparty artifacts.

Ok, now that we have the source, we can get into creating a component-level build. The toplevel build in jbossas/jbossbuild.xml defines all the components, their versions, and the locations of their artifacts. However, the component-level build defines how those artifacts are composed of java classes and other resources.

5.4.2. Component Level Build

Let's start out by just creating a minimal definition and see what happens. First, we want to create our component-info.xml under the deployment module. You can think of this file as the interface for this component. It will be uploaded to the repository along with the artifacts of this component so that other components may reference it.

For now, we can copy the entry from jbossas/jbossbuild.xml.
deployment/component-info.xml

```
```

```
<project name="deployment-component-info">

   <!-- ========================================================== -->
   <!-- Deployment                                                 -->
   <!-- ========================================================== -->
   <component id="deployment"
              module="jboss-deployment"
              version="5.0-SNAPSHOT">
   </component>


</project>
```

Once the component is declared, it needs to be defined. This is the responsibility of the jbossbuild.xml file: deployment/jbossbuild.xml

```


<?xml version="1.0"?>

<!--[snip: license and header comments ]-->

<project name="project"
         default="build"
         basedir="."
>
   <import file="../tools/etc/jbossbuild/tasks.xml"/>
   <import file="component-info.xml"/>

   <componentdef component="deployment" description="JBoss Deployment">
      <source id="main"/>
   </componentdef>

   <generate generate="deployment"/>
</project>


```

At the top, we see the root project element, which is required for all Ant build files. More interestingly, we see that two files are imported. The tasks.xml is from jbossbuild. This file defines the custom Ant tasks (like componentinfo) and ultimately drives the dynamic creation of Ant targets based on our component definition. The other file is the component-info.xml file we created above.

The second thing we see is the source element. This says that we have a source directory named "main". jbossbuild requires that you put all of your source under the "src" directory, so this resolves to "deployment/src/main".

Finally, we see the generate element. This basically a clue to jbossbuild to tell it we are done defining our component and that it should generate the targets.

Let's see what we've got now:

```


bash-2.05b$ ant -f jbossbuild.xml -projecthelp
Buildfile: jbossbuild.xml

Main targets:

 all               Build All
```

```
 api                Javadoc
 build              Build
 build.main         Build for the source src/main
 clean              Clean
 commit             Commit
 doc                Documentation
 rebuild            Synchronize then build
 rebuildall         Synchronize then build all
 runtest            Run tests
 synchronize        Synchronize
 synchronize.after  After synchronization processing
 test               Build and run the tests
Default target: build
```

Again, we see that jbossbuild has automatically generated a basic set of targets for us. Additionally, we see that a specific target has been generated for our main source. As we add artifacts and sources to our component definition, jbossbuild will define specific targets for these as well. Let's take a look at how this target is implemented:

```
bash-2.05b$ ant -f jbossbuild.xml show -Dshow=build.main
Buildfile: jbossbuild.xml

show:
<!-- Build for the source src/main -->
<target name="build.main">

<mkdir dir="C:\projects\newbuild-jboss\deployment\output\classes\main"/>

<depend destdir="C:\projects\newbuild-jboss\deployment\output\classes\main" srcdir="src/main">
  <classpath>
    <pathelement location="C:\projects\newbuild-jboss\deployment\output\classes\main"/>
  </classpath>
</depend>

<javac destdir="C:\projects\newbuild-jboss\deployment\output\classes\main" deprecation="true" srcdir="src
  <classpath>
    <pathelement location="C:\projects\newbuild-jboss\deployment\output\classes\main"/>
  </classpath>
  <src path="src/main"/>
</javac>

</target>
```

Based on this one <source id="main"> element all of the above is generated by jbossbuild. However, if we were to call this target now, it would fail because of unresolved imports. To fix this, we need to define the buildpath for the main source. The easiest way to do this is to find the library.classpath and dependentmodule.classpath in the deployment/build.xml:

```
    <!-- The combined library classpath -->
    <path id="library.classpath">
      <path refid="dom4j.dom4j.classpath"/>
    </path>

    <!-- The combined dependant module classpath -->
    <path id="dependentmodule.classpath">
      <path refid="jboss.common.classpath"/>
```

```
      <path refid="jboss.j2ee.classpath"/>
      <path refid="jboss.j2se.classpath"/>
      <path refid="jboss.system.classpath"/>
    </path>
```

Based on this we can determine the buildpath for the main source:

```
     <source id="main">
        <include component="dom4j-dom4j"/>
        <include component="common"/>
        <include component="j2ee"/>
        <include component="j2se"/>
        <include component="system"/>
     </source>
```

Generally, you should read this as "The main source tree includes these components as input." Concretely, the exported jars from these components are being included in the classpath of the call to javac:

```
$ ant -f jbossbuild.xml show -Dshow=build.main
<javac destdir="C:\projects\newbuild-jboss\deployment\output\classes\main"
       deprecation="true" srcdir="src/main" debug="true" excludes="${javac.excludes}">
  <classpath>
    <pathelement location="C:\projects\newbuild-jboss\j2ee\output\lib\jboss-saaj.jar"/>
    <pathelement location="C:\projects\newbuild-jboss\common\output\lib\namespace.jar"/>
    <pathelement location="C:\projects\newbuild-jboss\system\output\lib\jboss-system.jar"/>
    <pathelement location="C:\projects\newbuild-jboss\common\output\lib\jboss-common.jar"/>
    <pathelement location="C:\projects\newbuild-jboss\deployment\output\classes\main"/>
    <pathelement location="C:\projects\newbuild-jboss\j2se\output\lib\jboss-j2se.jar"/>
    <pathelement location="C:\projects\newbuild-jboss\thirdparty\dom4j-dom4j\lib\dom4j.jar"/>
    <pathelement location="C:\projects\newbuild-jboss\j2ee\output\lib\jboss-jaxrpc.jar"/>
    <pathelement location="C:\projects\newbuild-jboss\j2ee\output\lib\jboss-j2ee.jar"/>
  </classpath>
  <src path="src/main"/>
</javac>
```

How are components resolved to jars? jbossbuild searches for the component-info.xml of the included component. First in the root of the project (..) and second in the thirdparty directory (../thirdparty). The component-info.xml includes an export element which specifies which artifacts should be resolved when the component is included by another component. It's probably not a bad analogy to think of this mechanism as replacing buildmagic's modules.ent and libraries.ent

Now we should compile the source to make sure we got it right. We'll just use the build target because we are lazy and don't want to type build.main (rats!).

```
bash-2.05b$ ant -f jbossbuild.xml build
Buildfile: jbossbuild.xml

build.etc:
    [mkdir] Created dir: C:\projects\newbuild-jboss\deployment\output\etc
     [copy] Copying 1 file to C:\projects\newbuild-jboss\deployment\output\etc
```

```
build.main:
    [mkdir] Created dir: C:\projects\newbuild-jboss\deployment\output\classes\main
    [javac] Compiling 16 source files to C:\projects\newbuild-jboss\deployment\output\classes\main

build:

BUILD SUCCESSFUL
Total time: 7 seconds
```

5.4.2.1. Defining an Artifact

Great! Notice that the output for the source (id=main) is being placed in output/classes/main. Now we are ready to add an artifact definition. Looking at the deployment/build.xml, we see there is one artifact named jboss-deployment.jar. First, let's declare the artifact in our component-info.xml:

```
   <component id="deployment"
              module="jboss-deployment"
              version="5.0-SNAPSHOT">
      <artifact id="jboss-deployment.jar"/>
      <export>
         <include input="jboss-deployment.jar"/>
      </export>
   </component>
```

Notice also that we export this jar. When other components import this one, this is the jar they will want on their classpath.

Now, we need to create an artifactdef for this new artifact. The artifacdef defines how the artifact is composed of other inputs:

```
   ...
   </source>
   <artifactdef artifact="jboss-deployment.jar">
      <include input="main">
        <include pattern="org/jboss/deployment/**"/>
      </include>
   </artifactdef>
</componentdef>
```

This results in the following target being generated:

```
bash-2.05b$ ant -f jbossbuild.xml show -Dshow=build.jboss-deployment.jar
Buildfile: jbossbuild.xml

show:
<!-- Build for the artifact jboss-deployment.jar -->
<target name="build.jboss-deployment.jar">

<mkdir dir="C:\projects\newbuild-jboss\deployment\output\lib"/>

<jar destfile="C:\projects\newbuild-jboss\deployment\output\lib\jboss-deployment.jar">
```

```
  <fileset dir="C:\projects\newbuild-jboss\deployment\output\classes\main">
    <include name="org/jboss/deployment/**"/>
  </fileset>
</jar>

</target>
```

Notice that the <includes input="main"/> is resolved to output/classes/main.

5.4.3. Placing an Artifact in the Release

Now that we have completed the artifact, we need to define where it should be placed in the overall release structure. This information, as you will recall, is stored in the toplevel build (jbossas/jbossbuild.xml). We define the location in the release using the release tag:
jbossas/jbossbuild.xml:

```
     <component id="deployment"
                module="jboss-deployment"
                version="5.0-SNAPSHOT">
        <artifact id="jboss-deployment.jar" release="client"/>
     </component>
```

This will place the artifact in the client directory of the release:

```
bash-2.05b$ ant show -Dshow=release.jboss-deployment.jar
Buildfile: build.xml

show:
<target name="release.jboss-deployment.jar">

<mkdir dir="C:\projects\newbuild-jboss\jbossas\output\jbossas-5.0.0alpha\client"/>
<copy todir="C:\projects\newbuild-jboss\jbossas\output\jbossas-5.0.0alpha\client">
  <fileset file="C:\projects\newbuild-jboss\deployment\output\lib\jboss-deployment.jar"/>
</copy>

</target>
```

Now, you should be able perform a build of the application server:

```
$ ant build
...
```

Congratulations, you've successfully added a new component to jboss AS.

5.5. How to Add a Component to the Repository

This section describes the steps necessary to add a component to the build repository, currently at http://cruisecontrol.jboss.com/repository

1. First, you will want to checkout the repository locally.

```
cvs -d:ext:user@cvs.forge.jboss.com/cvsroot/jboss co repository.jboss.com
```

2. You need to decide on a component name. It is best to use something like organization-component so others can quickly tell what the name refers to. The exception is jboss components which are not prefixed with "jboss".

   Underneath the directory named after the component is the version number, which contains the component-info.xml. The lib directory below this will hold the jars.

```
repository.jboss.com
   + apache-log4j
     + 1.2.8
       + component-info.xml
       + lib
         + log4j.jar
```

3. In addition to adding the jars, you also need to create a component-info.xml. This file allows other components to reference your jars. We want to make sure that the component-info.xml reflects the version we indicated in the directory structure above.

```
<project name="apache-log4j-component-info">
   <!-- ================================================================= -->
   <!-- Apache Log4j                                                      -->
   <!-- ================================================================= -->

   <component id="apache-log4j"
              licenseType="apache-2.0"
              version="1.2.8"
              projectHome="http://logging.apache.org/">
      <artifact id="log4j.jar"/>
      <artifact id="snmpTrapAppender.jar"/>
      <export>
         <include input="log4j.jar"/>
      </export>
   </component>

</project>
```

4. You can commit the new version to the repository using cvs commands. There is (will be) a scheduled process which updates the online repository from cvs every 5 minutes. If this fails, please contact qa@jboss.com

5. Once the component is available in the online build repository, you may configure toplevel (e.g., jbossas/jbossbuild.xml) build to include it:

```
   <component id="apache-log4j"
```

```
            version="1.2.8"
   >
      <artifact id="log4j.jar"/>
      <artifact id="snmpTrapAppender.jar"/>
   </component>
```

6

CVS Access for JBoss Sources

CVS Access for JBoss Sources

Source code is available for every JBoss module and any version of JBoss can be built from source by downloading the appropriate version of the code from the JBoss Forge CVS Repository.

6.1. Obtaining a CVS Client

The command line version of the CVS program is freely available for nearly every platform and is included by default on most Linux and UNIX distributions. A good port of CVS as well as numerous other UNIX programs for Win32 platforms is available from Cygwin [http://sources.redhat.com/cygwin/].

The syntax of the command line version of CVS will be examined because this is common across all platforms.

For complete documentation on CVS, check out The CVS Home Page [http://www.cvshome.org/].

6.2. Anonymous CVS Access

All JBoss projects' CVS repositories can be accessed through anonymous(pserver) CVS with the following instruction set. The module you want to check out must be specified as the modulename. When prompted for a password for anonymous, simply press the Enter key.

The general syntax of the command line version of CVS for anonymous access to the JBoss repositories is:

```
cvs -d:pserver:anonymous@anoncvs.forge.jboss.net:/cvsroot/jboss login
cvs -z3 -d:pserver:anonymous@anoncvs.forge.jboss.net:/cvsroot/jboss co modulename
```

The first command logs into JBoss CVS repository as an anonymous user. This command only needs to be performed once for each machine on which you use CVS because the login information will be saved in your HOME/.cvspass file or equivalent for your system. The second command checks out a copy of the modulename source code into the directory from which you run the cvs command.

To avoid having to type the long cvs command line each time, you can set up a CVSROOT environment variable.

```
set CVSROOT=:pserver:anonymous@anoncvs.forge.jboss.net:/cvsroot/jboss
```

The abbreviated versions of the previous commands can then be used:

```
cvs login
cvs -z3 co modulename
```

The name of the JBoss module alias you use depends on the version of JBoss you want. For the 3.0 branch the module name is jboss-3.0, for the 3.2 branch it is jboss-3.2, and in general, for branch x.y the module name is jboss-x.y.

To checkout the HEAD revision of jboss to obtain the latest code on the main branch you would use jboss-head as the module name.

Releases of JBoss are tagged with the pattern JBoss_X_Y_Z where X is the major version, Y is the minor version and Z is the patch version. Release branches of JBoss are tagged with the pattern Branch_X_Y.

Some checkout examples are:

```
cvs co -r JBoss_3_2_6 jboss-3.2 # Checkout the 3.2.6 release version code
cvs co jboss-head # Checkout the curent HEAD branch code
```

7

Coding Conventions

This section lists some general guidelines followed in JBoss code for coding sources / tests.

All files (including tests) should have a header like the following:

```
/*
 * JBoss, Home of Professional Open Source
 *
 * Distributable under LGPL license.
 * See terms of license at gnu.org.
 */
```

The header asserts the LGPL license, without which the content would be closed source. The assumption under law is copyright the author, all rights reserved or sometimes the opposite - if something is published without asserting the copyright or license it is public domain.

Use the template files on JIRA for consistency. These template files encapsulate settings that are generally followed such as replacing tabs with 3 spaces for portability amongst editors, auto-insertion of headers etc.

7.1. Templates

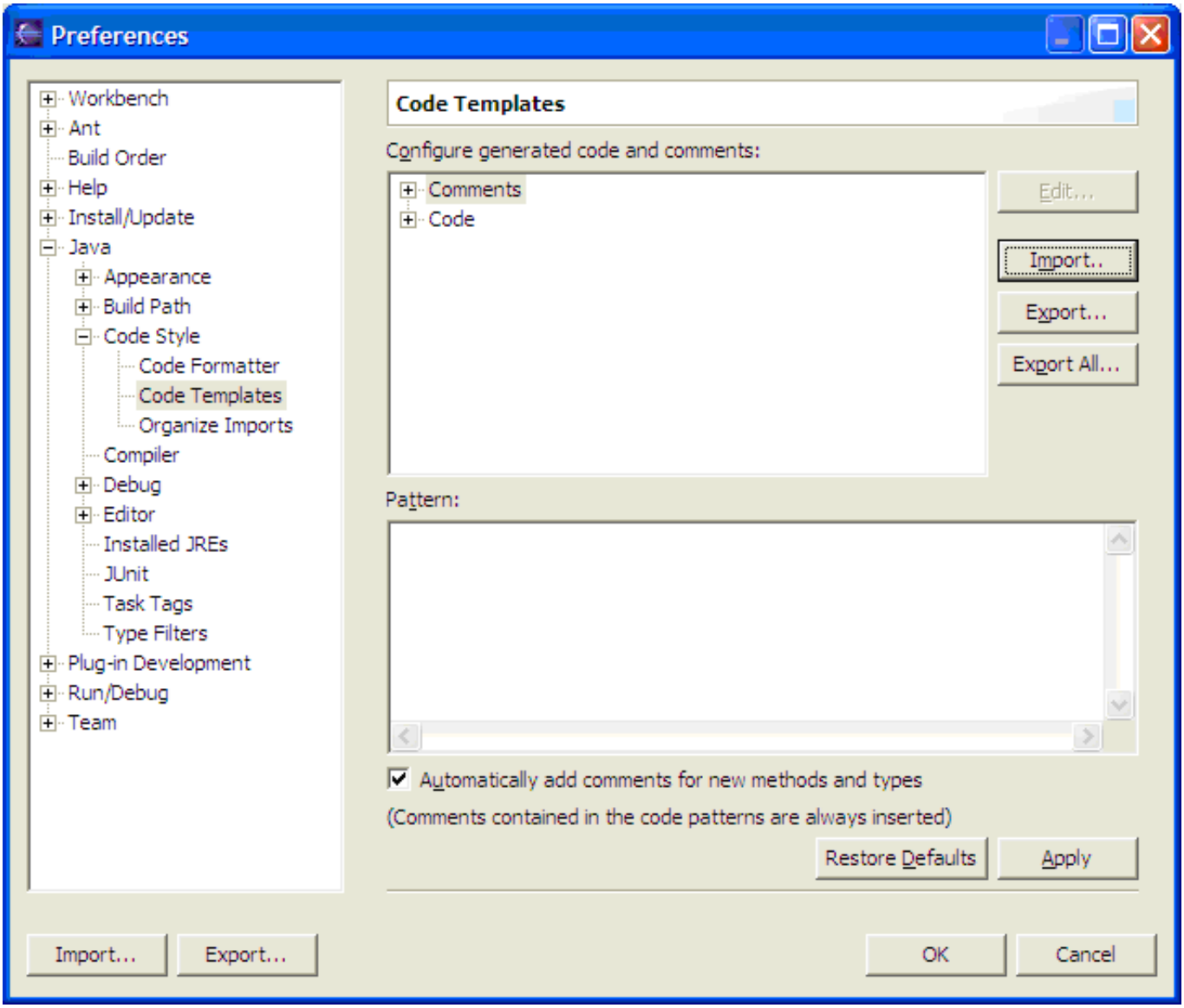
Template files for the Eclipse IDE can be found here: JBoss Eclipse Format [http://jira.jboss.com/jira/secure/attachment/12310313/jboss-format.xml/]. JBoss Eclipse Template [http://jira.jboss.com/jira/secure/attachment/12310312/jboss-template.xml/].

Template files for other IDEs(IntelliJ-IDEA, NetBeans should be available here soon.

7.1.1. Importing Templates into the Eclipse IDE

The process of importing templates into the Eclipse IDE is as follows:

On the IDE, goto Windows Menu => Preferences => Java => Code Style => Code Templates => Import and choose to import the Eclipse template files.



Tools such as Jalopy [http://jalopy.sourceforge.net] help to automate template changes at one shot to numerous files.

7.2. Some more general guidelines

1. Fully qualified imports should be used, rather than importing x.y.*.
2. Use newlines for opening braces, so that the top and bottom braces can be visually matched.
3. Aid visual separation of logical steps by introducing newlines and appropriate comments above them.

7.3. JavaDoc recommendations

1. All public and protected members and methods should be documented.

2. It should be documented if "null" is an acceptable value for parameters.

3. Side effects of method calls, if known, or as they're discovered should be documented.

4. It would also be useful to know from where an overridden method can be invoked.

Example 7.1. A class that conforms to JBoss coding guidelines

```
/*
 * JBoss, the OpenSource J2EE webOS
 *
 * Distributable under LGPL license.
 * See terms of license at gnu.org.
 */

package x;

// EXPLICIT IMPORTS
import a.b.C1; // GOOD
import a.b.C2;
import a.b.C3;

// DO NOT WRITE
import a.b.*;  // BAD

// DO NOT USE "TAB" TO INDENT CODE USE *3* SPACES FOR PORTABILITY AMONG EDITORS

/**
 * A description of this class.
 *
 * @see SomeRelatedClass.
 *
 * @version <tt>$Revision: 1.2 $</tt>
 * @author  <a href="mailto:{email}">{full name}</a>.
```

```
 * @author  <a href="mailto:marc@jboss.org">Marc Fleury</a>
 */

public class X
   extends Y
   implements Z
{
   // Constants -----------------------------------------------------

   // Attributes ----------------------------------------------------

   // Static --------------------------------------------------------

   // Constructors --------------------------------------------------

   // Public --------------------------------------------------------

   public void startService() throws Exception
   {
      // Use the newline for the opening bracket so we can match top
      // and bottom bracket visually

      Class cls = Class.forName(dataSourceClass);
      vendorSource = (XADataSource)cls.newInstance();

      // JUMP A LINE BETWEEN LOGICALLY DISTINCT **STEPS** AND ADD A
      // LINE OF COMMENT TO IT
      cls = vendorSource.getClass();



      if(properties != null)
      {
```

```
        try
        {
        }
        catch (IOException ioe)
        {
        }
        for (Iterator i = props.entrySet().iterator(); i.hasNext();)
        {

           // Get the name and value for the attributes
           Map.Entry entry = (Map.Entry) i.next();
           String attributeName = (String) entry.getKey();
           String attributeValue = (String) entry.getValue();

           // Print the debug message
           log.debug("Setting attribute '" + attributeName + "' to '" + attributeValue + "'");

           // get the attribute
           Method setAttribute =
           cls.getMethod("set" + attributeName, new Class[] { String.class });

           // And set the value
           setAttribute.invoke(vendorSource, new Object[] { attributeValue });
        }
     }

     // Test database
     vendorSource.getXAConnection().close();

     // Bind in JNDI
     bind(new InitialContext(), "java:/"+getPoolName(),
        new Reference(vendorSource.getClass().getName(),
           getClass().getName(), null));
  }
```

```
   // Z implementation ------------------------------------------------

   // Y overrides -----------------------------------------------------

   // Package protected -----------------------------------------------

   // Protected -------------------------------------------------------

   // Private ---------------------------------------------------------

   // Inner classes ---------------------------------------------------
}
```

Example 7.2. An interface that conforms to JBoss coding guidelines

```
/*
 * JBoss, the OpenSource J2EE webOS
 *
 * Distributable under LGPL license.
 * See terms of license at gnu.org.
 */

package x;

// EXPLICIT IMPORTS
import a.b.C1; // GOOD
import a.b.C2;
import a.b.C3;

// DO NOT WRITE
```

```
import a.b.*;  // BAD


// DO NOT USE "TAB" TO INDENT CODE USE *3* SPACES FOR PORTABILITY AMONG // EDITORS


/**
 * A description of this interface.
 *
 * @see SomeRelatedClass
 *
 * @version <tt>$Revision: 1.2 $</tt>
 * @author  <a href="mailto:{email}">{full name}</a>.
 * @author  <a href="mailto:marc@jboss.org">Marc Fleury</a>
 */
public interface X extends Y
{
   int MY_STATIC_FINAL_VALUE = 57;


   ReturnClass doSomething() throws ExceptionA, ExceptionB;
}
```

8

Logging Conventions

Persisted diagnostic logs are often very useful in debugging software issues. This section lists some general guidelines followed in JBoss code for diagnostic logging.

8.1. Obtaining a Logger

The following code snippet illustrates how you can obtain a logger.

```
package org.jboss.X.Y;

import org.jboss.logging.Logger;


public class TestABCWrapper

{

   private static final Logger log = Logger.getLogger(TestABCWrapper.class.getName());


   // Hereafter, the logger may be used with whatever priority level as appropriate.

}
```

After a logger is obtained, it can be used to log messages by specifying appropriate priority levels.

8.2. Logging Levels

1. FATAL - Use the FATAL level priority for events that indicate a critical service failure. If a service issues a FATAL error it is completely unable to service requests of any kind.

2. ERROR - Use the ERROR level priority for events that indicate a disruption in a request or the ability to service a request. A service should have some capacity to continue to service requests in the presence of ERRORs.

3. WARN - Use the WARN level priority for events that may indicate a non-critical service error. Resumable errors, or minor breaches in request expectations fall into this category. The distinction between WARN and ERROR may be hard to discern and so its up to the developer to judge. The simplest criterion is would this

failure result in a user support call. If it would use ERROR. If it would not use WARN.

4. INFO - Use the INFO level priority for service life-cycle events and other crucial related information. Looking at the INFO messages for a given service category should tell you exactly what state the service is in.

5. DEBUG - Use the DEBUG level priority for log messages that convey extra information regarding life-cycle events. Developer or in depth information required for support is the basis for this priority. The important point is that when the DEBUG level priority is enabled, the JBoss server log should not grow proportionally with the number of server requests. Looking at the DEBUG and INFO messages for a given service category should tell you exactly what state the service is in, as well as what server resources it is using: ports, interfaces, log files, etc.

6. TRACE - Use TRACE the level priority for log messages that are directly associated with activity that corresponds requests. Further, such messages should not be submitted to a Logger unless the Logger category priority threshold indicates that the message will be rendered. Use the `Logger.isTraceEnabled()` method to determine if the category priority threshold is enabled. The point of the `TRACE` priority is to allow for deep probing of the JBoss server behavior when necessary. When the `TRACE` level priority is enabled, you can expect the number of messages in the JBoss server log to grow at least a x N, where N is the number of requests received by the server, a some constant. The server log may well grow as power of N depending on the request-handling layer being traced.

8.3. Log4j Configuration

The `log4j` configuration is loaded from the jboss server `conf/log4j.xml` file. You can edit this to add/change the default appenders and logging thresholds.

8.3.1. Separating Application Logs

You can segment logging output by assigning `log4j` categories to specific appenders in the `conf/log4j.xml` configuration.

Example 8.1. Assigning categories to specific appenders

```
<appender name="App1Log" class="org.apache.log4j.FileAppender">
    <errorHandler
           class="org.jboss.logging.util.OnlyOnceErrorHandler"/>
    <param name="Append" value="false"/>
    <param name="File"
           value="${jboss.server.home.dir}/log/app1.log"/>
    <layout class="org.apache.log4j.PatternLayout">
```

```
        <param name="ConversionPattern"
               value="%d{ABSOLUTE} %-5p [%c{1}] %m%n"/>
      </layout>
   </appender>

...

   <category name="com.app1">
     <appender-ref ref="App1Log"/>
   </category>
   <category name="com.util">
     <appender-ref ref="App1Log"/>
   </category>

...

   <root>
      <appender-ref ref="CONSOLE"/>
      <appender-ref ref="FILE"/>
      <appender-ref ref="App1Log"/>
   </root>
```

8.3.2. Specifying appenders and filters

If you have multiple apps with shared classes/categories, and/or want the jboss categories to show up in your app log then this approach will not work. There is a new appender filter called TCLFilter that can help with this. The filter should be added to the appender and it needs to be speciifed what deployment url should logging be restricted to. For example, if your app1 deployment was app1.ear, you would use the following additions to the `conf/log4j.xml`:

Example 8.2. Filtering log messages

```
```

```
   <appender name="App1Log" class="org.apache.log4j.FileAppender">
      <errorHandler
           class="org.jboss.logging.util.OnlyOnceErrorHandler"/>
      <param name="Append" value="false"/>
      <param name="File"
             value="${jboss.server.home.dir}/log/app1.log"/>
      <layout class="org.apache.log4j.PatternLayout">
         <param name="ConversionPattern"
                value="%d{ABSOLUTE} %-5p [%c{1}] %m%n"/>
      </layout>
      <filter class="org.jboss.logging.filter.TCLFilter">
         <param name="AcceptOnMatch" value="true"/>
         <param name="DeployURL" value="app1.ear"/>
      </filter>
   </appender>

...

   <root>
      <appender-ref ref="CONSOLE"/>
      <appender-ref ref="FILE"/>
      <appender-ref ref="App1Log"/>
   </root>
```

8.3.3. Logging to a Seperate Server

The log4j framework has a number of appenders that allow you to send log message to an external server. Common appenders include:

1. `org.apache.log4j.net.JMSAppender`

2. `org.apache.log4j.net.SMTPAppender`
3. `org.apache.log4j.net.SocketAppender`
4. `org.apache.log4j.net.SyslogAppender`
5. `org.apache.log4j.net.TelnetAppender`

Documentation on configuration of these appenders can be found at Apache Logging Services [http://logging.apache.org/].

JBoss has a `Log4jSocketServer` service that allows for easy use of the `SocketAppender`.

Example 8.3. Setting up and using the `Log4jSocketServer` service.

The `org.jboss.logging.Log4jSocketServer` is an mbean service that allows one to collect output from multiple `log4j` clients (including jboss servers) that are using the `org.apache.log4j.net.SocketAppender`.

The `Log4jSocketServer` creates a server socket to accept `SocketAppender` connections, and logs incoming messages based on the local `log4j.xml` configuration.

You can create a minimal jboss configuration that includes a `Log4jSocketServer` to act as your log server.

Example 8.4. An `Log4jSocketServer` mbean configuration

The following MBean Configuration can be added to the `conf/jboss-service.xml`

```
<mbean code="org.jboss.logging.Log4jSocketServer"
   name="jboss.system:type=Log4jService,service=SocketServer">
   <attribute name="Port">12345</attribute>
   <attribute name="BindAddress">${jboss.bind.address}</attribute>
</mbean>
```

The `Log4jSocketServer` adds an MDC entry under the key 'host' which includes the client socket `InetAddress.getHostName` value on every client connection. This allows you to differentiate logging output based on the client hostname using the MDC pattern.

Example 8.5. Augmenting the log server console output with the logging client socket hostname

```

```

```
   <appender name="CONSOLE" class="org.apache.log4j.ConsoleAppender">
   <errorHandler class="org.jboss.logging.util.OnlyOnceErrorHandler"/>
   <param name="Target" value="System.out"/>
   <param name="Threshold" value="INFO"/>

   <layout class="org.apache.log4j.PatternLayout">
      <param name="ConversionPattern" value="%d{ABSOLUTE} %-5p [%c{1},%X{host}] %m%n"/>
   </layout>
</appender>
```

All other jboss servers that should send log messages to the log server would add an appender configuration that uses the SocketAppender.

Example 8.6. `log4j.xml` appender for the Log4jSocketServer

```
<appender name="SOCKET" class="org.apache.log4j.net.SocketAppender">
   <param name="Port" value="12345"/>
   <param name="RemoteHost" value="loghost"/>
   <param name="ReconnectionDelay" value="60000"/>
   <param name="Threshold" value="INFO"/>
</appender>
```

8.3.4. Key JBoss Subsystem Categories

Some of the key subsystem category names are given in the following table. These are just the top level category names. Generally you can specify much more specific category names to enable very targeted logging.

Table 8.1. JBoss SubSystem Categories

SubSystem	Category
Cache	org.jboss.cache
CMP	org.jboss.ejb.plugins.cmp
Core Service	org.jboss.system
Cluster	org.jboss.ha
EJB	org.jboss.ejb
JCA	org.jboss.resource
JMX	org.jboss.mx
JMS	org.jboss.mq
JTA	org.jboss.tm
MDB	org.jboss.ejb.plugins.jms, org.jboss.jms
Security	org.jboss.security
Tomcat	org.jboss.web, org.apache.catalina
Apache Stuff	org.apache
JGroups	org.jgroups

8.3.5. Redirecting Category Output

When you increase the level of logging for one or more categories, it is often useful to redirect the output to a seperate file for easier investigation. To do this you add an appender-ref to the category as shown here:

Example 8.7. Adding an appender-ref to a category

```
   <appender name="JSR77" class="org.apache.log4j.FileAppender">
      <param name="File"
             value="${jboss.server.home.dir}/log/jsr77.log"/>
...
   </appender>
```

```
<!-- Limit the JSR77 categories -->
<category name="org.jboss.management" additivity="false">
   <priority value="DEBUG"/>
   <appender-ref ref="JSR77"/>
</category>
```

This sends all `org.jboss.management` output to the `jsr77.log` file. The additivity attribute controls whether output continues to go to the root category appender. If false, output only goes to the appenders referred to by the category.

8.3.6. Using your own log4j.xml file - class loader scoping

In order to use your own `log4j.xml` file you need to do something to initialize `log4j` in your application. If you use the default singleton initialization method where the first use of `log4j` triggers a search for the `log4j` initialization files, you need to configure a ClassLoader to use scoped class loading, with overrides of the jBoss classes. You also have to include the `log4j.jar` in your application so that new `log4j` singletons are created in your applications scope.

Note

You cannot use a `log4j.properties` file using this approach, at least using `log4j-1.2.8` because it preferentially searches for a `log4j.xml` resource and will find the `conf/log4j.xml` ahead of the application `log4j.properties` file. You could rename the `conf/log4j.xml` to something like `conf/jboss-log4j.xml` and then change the `ConfigurationURL` attribute of the `Log4jService` in the `conf/jboss-service.xml` to get around this.

8.3.7. Using your own log4j.properties file - class loader scoping

To use a `log4j.properties` file, you have to make the change in `conf/jboss-service.xml` as shown below. This is necessary for the reasons mentioned above. Essentially you are changing the `log4j` resource file that jBossAS will look for. After making the change in `jboss-service.xml` make sure you rename the `conf/log4j.xml` to the name that you have give in `jboss-service.xml` (in this case `jboss-log4j.xml`).

```
<!--=====================================================================-->
<!-- Log4j Initialization                                              -->
<!-=====================================================================-->

<mbean code="org.jboss.logging.Log4jService"
   name="jboss.system:type=Log4jService,service=Logging">
```

```
   <attribute name="ConfigurationURL">
               resource:jboss-log4j.xml</attribute>
 <!-- Set the org.apache.log4j.helpers.LogLog.setQuiteMode.
   As of log4j1.2.8 this needs to be set to avoid a possible deadlock
   on exception at the appender level. See bug#696819.
 -->
 <attribute name="Log4jQuietMode">true</attribute>
 <!-- How frequently in seconds the ConfigurationURL is checked for changes -->
 <attribute name="RefreshPeriod">60</attribute>
</mbean>
```

Drop `log4j.jar` in your `myapp.war/WEB-INF`. Make the change in `jboss-web.xml` for class-loading, as shown in the section above. In this case, `myapp.war/WEB-INF/jboss-web.xml` looks like this:

```
<jboss-web>
  <class-loading java2ClassLoadingCompliance="false">
  <loader-repository>
      myapp:loader=myapp.war
      <loader-repository-config>java2ParentDelegation=false
      </loader-repository-config>
   </loader-repository>
 </class-loading>
</jboss-web>
```

Now, in your `deploy/myapp.war/WEB-INF/classes` create a `log4j.properties`.

Example 8.8. Sample `log4j.properties`

```
# Debug log4j
```

```
log4j.debug=true

log4j.rootLogger=debug, myapp



log4j.appender.myapp=org.apache.log4j.FileAppender

log4j.appender.myapp.layout=org.apache.log4j.HTMLLayout

log4j.appender.myapp.layout.LocationInfo=true

log4j.appender.myapp.layout.Title='All' Log

log4j.appender.myapp.File=${jboss.server.home.dir}/deploy/myapp.war/WEB-INF/logs/myapp.html

log4j.appender.myapp.ImmediateFlush=true

log4j.appender.myapp.Append=false
```

The above property file sets the `log4j` debug system to true, which displays `log4j` messages in your jBoss log. You can use this to discover errors, if any in your properties file. It then produces a nice HTML log file and places it in your application's `WEB-INF/logs` directory. In your application, you can call this logger with the syntax:

```
...

private static Logger log = Logger.getLogger("myapp");

...

log.debug("############## A debug message from myapp logger #########");

...
```

If all goes well, you should see this message in `myapp.html`.

After jBossAS has reloaded `conf/jboss-service.xml` (you may have to restart jBossAS), touch `myapp.war/WEB-INF/web.xml` so that JBoss reloads the configuration for your application. As the application loads you should see `log4j` debug messages showing that its reading your `log4j.properties`. This should enable you to have your own logging system independent of the JBoss logging system.

8.3.8. Using your own `log4j.xml` file - `Log4j RepositorySelector`

Another way to achieve this is to write a custom `RepositorySelector` that changes how the `LogManager` gets a logger. Using this technique, `Logger.getLogger()` will return a different logger based on the context class loader. Each context class loader has its own configuration set up with its own `log4j.xml` file.

Example 8.9. A `RepositorySelector`

The following code shows a `RepositorySelector` that looks for a `log4j.xml` file in the `WEB-INF` directory.

```
/*
 * JBoss, the OpenSource J2EE webOS
 *
 * Distributable under LGPL license.
 * See terms of license at gnu.org.
 */

package org.jboss.repositoryselectorexample;

import java.io.InputStream;
import java.util.HashMap;
import java.util.Map;
import javax.servlet.ServletConfig;
import javax.servlet.ServletException;
import javax.xml.parsers.DocumentBuilderFactory;
import org.apache.log4j.Hierarchy;
import org.apache.log4j.Level;
import org.apache.log4j.LogManager;
import org.apache.log4j.spi.LoggerRepository;
import org.apache.log4j.spi.RepositorySelector;
import org.apache.log4j.spi.RootCategory;
import org.apache.log4j.xml.DOMConfigurator;
import org.w3c.dom.Document;

/**
 * This RepositorySelector is for use with web applications.
 * It assumes that your log4j.xml file is in the WEB-INF directory.
 * @author  Stan Silvert
```

```
 */
public class MyRepositorySelector implements RepositorySelector
{
   private static boolean initialized = false;


   // This object is used for the guard because it doesn't get
   // recycled when the application is redeployed.
   private static Object guard = LogManager.getRootLogger();


   private static Map repositories = new HashMap();
   private static LoggerRepository defaultRepository;


   /**
    * Register your web-app with this repository selector.
    */
   public static synchronized void init(ServletConfig config)
        throws ServletException {
      if( !initialized ) // set the global RepositorySelector
      {
         defaultRepository = LogManager.getLoggerRepository();
         RepositorySelector theSelector = new MyRepositorySelector();
         LogManager.setRepositorySelector(theSelector, guard);
         initialized = true;
      }


      Hierarchy hierarchy = new Hierarchy(new
                                RootCategory(Level.DEBUG));
      loadLog4JConfig(config, hierarchy);
      ClassLoader loader =
           Thread.currentThread().getContextClassLoader();
      repositories.put(loader, hierarchy);
   }


   // load log4j.xml from WEB-INF
```

```
private static void loadLog4JConfig(ServletConfig config,

                                    Hierarchy hierarchy)

                                         throws ServletException {

     try {

          String log4jFile = "/WEB-INF/log4j.xml";

          InputStream log4JConfig =



          config.getServletContext().getResourceAsStream(log4jFile);

          Document doc = DocumentBuilderFactory.newInstance()

                                               .newDocumentBuilder()

                                               .parse(log4JConfig);

          DOMConfigurator conf = new DOMConfigurator();

          conf.doConfigure(doc.getDocumentElement(), hierarchy);

     } catch (Exception e) {

          throw new ServletException(e);

     }

 }



private MyRepositorySelector() {

}



public LoggerRepository getLoggerRepository() {

   ClassLoader loader =

          Thread.currentThread().getContextClassLoader();

   LoggerRepository repository =

          (LoggerRepository)repositories.get(loader);



   if (repository == null) {

       return defaultRepository;

   } else {

       return repository;

   }

}
```

```
}
```

8.4. JDK java.util.logging

The choice of the actual logging implementation is determined by the `org.jboss.logging.Logger.pluginClass` system property. This property specifies the class name of an implementation of the `org.jboss.logging.LoggerPlugin` interface. The default value for this is the `org.jboss.logging.Log4jLoggerPlugin` class.

If you want to use the JDK 1.4+ `java.util.logging` framework instead of `log4j`, you can create your own `Log4jLoggerPlugin` to do this. The attached `JDK14LoggerPlugin.java` file shows an example implementation.

To use this, specify the following system properties:

1. To specify the custom JDK1.4 plugin:

```
org.jboss.logging.Logger.pluginClass = logging.JDK14LoggerPlugin
```

2. To specify the JDK1.4 logging configuration file:

```
java.util.logging.config.file = logging.properties
```

This can be done using the `JAVA_OPTS` env variable, for example:

```
JAVA_OPTS="-Dorg.jboss.logging.Logger.pluginClass=logging.JDK14LoggerPlugin

           -Djava.util.logging.config.file=logging.properties"
```

You need to make your custom `Log4jLoggerPlugin` available to JBoss by placing it in a jar in the `JBOSS_DIST/lib` directory, and then telling JBoss to load this as part of the bootstrap libraries by passing in `-L jarname` on the command line as follows:

```
starksm@banshee9100 bin$ run.sh -c minimal -L logger.jar
```

9

JBoss Test Suite

The JBoss Testsuite module is a collection of JUnit tests which require a running JBoss instance for in-container testing. Unit tests not requiring the container reside in the module they are testing.

The setup and initialization of the container is performed in the testsuite's `build.xml` file. The testsuite module also provides utility classes which support the deployment of test artifacts to the container.

9.1. How To Run the JBoss Testsuite

A source distribution of JBoss must be available to run the testsuite. This document applies only to JBoss 3.2.7 and above.

9.1.1. Build JBoss

Before building the testsuite, the rest of the project must be built:

Unix

```
cd build

./build.sh
```

Windows

```
cd build

build.bat
```

9.1.2. Build and Run the Testsuite

To build and run the testsuite, type the following. Note that you no longer are required to seperately start a JBoss server instance before running the testsuite.

Important

You must not have a JBoss instance running before you run the testsuite.

Unix

```
cd ../testsuite

./build.sh tests
```

Windows

```
cd ../testsuite

build.bat tests
```

The build script will start and stop various configurations of JBoss, and then run tests against those configurations.

9.1.3. Running One Test at a Time

To run an individual test, you will need to start the appropriate configuration. For most tests, this will be the `"all"` configuration:

```
build/output/jboss-5.0.0alpha/bin/run.sh -c all
```

And then tell the testsuite which test you want to run:

```
cd testsuite

./build.sh one-test -Dtest=org.jboss.test.package.SomeTestCase
```

9.1.4. Clustering Tests Configuration

Most of the tests are against a single server instance started on localhost. However, the clustering tests require two server instances. By default, the testsuite will bind one of these instances to localhost, and the other will be bound to hostname. You can override this in the `testsuite/local.properties` file.

```
node0=localhost

...

node1=MyHostname
```

The nodes must be bound to different IP addresses, otherwise there will be port conflicts. Also, note these addresses must be local to the box you are running the testsuite on, the testsuite will need to start each server process before running the tests.

9.1.5. Viewing the Results

A browsable HTML document containing the testsuite results is available under `testsuite/output/reports/html`, and a text report (useful for emailing) is available under `testsuite/out-`

put/reports/text.

9.2. Testsuite Changes

The testsuite build.xml has been refactored to allow automated testing of multiple server configurations. The testsuite build scripts include facilities for customizing server configurations and starting and stopping these configurations. Most notably, this improvement allows clustering unit tests to be completely automated.

9.2.1. Targets

Tests are now grouped in to targets according to which server configuration they require. Here is a summary of the targets called by the top-level tests target:

Table 9.1. Build Targets and Descriptions

Target	Description
jboss-minimal-tests	Tests requiring the minimal configuration.
jboss-all-config-tests	Runs the all configuration. Most tests can go here.
tests-security-manager	Runs the default configuration with a security manager.
tests-clustering	Creates two custom configurations based on the all configuration. Tests run in this target should extend JBossClusteredTestCase to access cluster information.
tomcat-ssl-tests	Creates and runs a configuration with Tomcat SSL enabled.
tomcat-sso-tests	Creates and runs a configuration with SSO enabled.
tomcat-sso-clustered-tests	Creates and runs two nodes with SSO enabled.

9.2.2. Files

The testsuite build scripts have been reorganized. The code generation and jar targets have been extracted to their own files in testsuite/imports. These targets are imported for use by the main build.xml file. Also, it is important to note that module and library definitions are in different files.

Table 9.2. Summary of build files

Build File	Description
testsuite/build.xml	Contains test targets. This file imports the macros and targets from the files below.

Build File	Description
testsuite/imports/server-config.xml	Contains macros for creating and starting different server configurations.
tools/etc/buildmagic/modules.xml	Similar to modules.ent, this file contains the Ant classpath definitions for each JBoss module.
tools/etc/buildmagic/thirdparty.xml	Like thirdparty.ent, this contains the Ant classpath definitions for each third party library.
testsuite/imports/code-generation.xml	Xdoclet code generation. This file has the following targets: compile-bean-source, compile-mbean-sources, compile-xmbean-dds, compile-proxycompiler-bean-source.
testsuite/imports/test-jars.xml	All jar tasks. The top-level jars target calls each module's _jar-* target (eg: _jar-aop).

9.3. Functional Tests

Functional tests need to be located in the module which they test. The testsuite needs to be able to include these in the "tests" target.

To contribute functional tests to the testsuite, each module should contain a tests directory with with a `build.xml`. The `build.xml` should contain at least one target, `functional-tests`, which executes JUnit tests. The `functional-tests` target should build the tests, but should assume that the module itself has been built. The `tests/build.xml` should use the Ant `<import/>` task to reuse targets and property definitions from the module's main `build.xml`.

Functional test source code belongs in the `tests/src` directory. The package structure of the tests should mirror the module's package structure, with an additional test package below `org/jboss`.

For example, classes under `org.jboss.messaging.core` should have tests under `org.jboss.test.messaging.core`.

9.3.1. Integration with Testsuite

The `testsuite/build.xml` will include a `functional-tests` target which uses the `<subant>` task to call the funtional-tests target on each module's `tests/build.xml`. The testsuite will only override properties relevant to the junit execution, and the module's `tests/build.xml` must use these properties as values for the corresponding attributes:

1. `junit.printsummary`
2. `junit.haltonerror`
3. `junit.haltonfailure`
4. `junit.fork`
5. `junit.timeout`

6. junit.jvm
7. junit.jvm.options
8. junit.formatter.usefile
9. junit.batchtest.todir
10. junit.batchtest.haltonerror
11. junit.batchtest.haltonfailure
12. junit.batchtest.fork

The following properties are not set by the testsuite:

1. junit.sysproperty.log4j.configuration
2. junit.sysproperty.*

Example 9.1. Example Build Script for Functional Tests

```
<?xml version="1.0" encoding="UTF-8"?>

<!-- ====================================================================== -->
<!--                                                                        -->
<!--  JBoss, the OpenSource J2EE webOS                                      -->
<!--                                                                        -->
<!--  Distributable under LGPL license.                                     -->
<!--  See terms of license at http://www.gnu.org.                           -->
<!--                                                                        -->
<!-- ====================================================================== -->

<!-- $Id: testsuite.xml,v 1.1 2005/04/15 14:42:52 piyer Exp $ -->

<project default="tests" name="JBoss/Messaging">

   <!-- overridden to resolve thirdparty & module deps -->
   <dirname property="remote.root" file="${basedir}"/>
   <dirname property="project.root" file="${remote.root}"/>
```

```
<import file="../../tools/etc/buildmagic/build-common.xml"/>

<import file="../../tools/etc/buildmagic/libraries.xml"/>

<import file="../../tools/etc/buildmagic/modules.xml"/>


<!-- ================================================================== -->

<!-- Configuration                                                      -->

<!-- ================================================================== -->


<!-- Module name(s) & version -->

<property name="module.name" value="jms"/>

<property name="module.Name" value="JBoss Messaging"/>

<property name="module.version" value="5.0.0"/>


<!-- ========= -->

<!-- Libraries -->

<!-- ========= -->

<!-- The combined library classpath -->

<path id="library.classpath">

   <path refid="apache.log4j.classpath"/>

   <path refid="oswego.concurrent.classpath"/>

   <path refid="junit.junit.classpath"/>

   <path refid="jgroups.jgroups.classpath"/>

   <path refid="apache.commons.classpath"/>

</path>


<!-- ======= -->

<!-- Modules -->

<!-- ======= -->

<!-- The combined dependent module classpath -->

<path id="dependentmodule.classpath">

   <path refid="jboss.common.classpath"/>

   <path refid="jboss.jms.classpath"/>

</path>
```

```
<!-- ===== -->
<!-- Tasks -->
<!-- ===== -->

<property name="source.tests.java" value="${module.source}"/>
<property name="build.tests.classes" value="${module.output}/classes"/>
<property name="build.tests.lib" value="${module.output}/lib"/>
<property name="build.tests.output" value="${module.output}/reports"/>
<property name="build.performance.tests.output" value="${module.output}/reports/performance"/>
<property name="build.tests.archive" value="jboss-messaging-tests.jar"/>

<path id="test.classpath">
   <path refid="library.classpath"/>
   <path refid="dependentmodule.classpath"/>
</path>

<!-- Compile all test files -->
<target name="compile-test-classes">
   <mkdir dir="${build.tests.classes}"/>
   <javac destdir="${build.tests.classes}"
      optimize="${javac.optimize}"
      target="1.4"
      source="1.4"
      debug="${javac.debug}"
      depend="${javac.depend}"
      verbose="${javac.verbose}"
      deprecation="${javac.deprecation}"
      includeAntRuntime="${javac.include.ant.runtime}"
      includeJavaRuntime="${javac.include.java.runtime}"
      failonerror="${javac.fail.onerror}">
      <src path="${source.tests.java}"/>
```

```
      <classpath refid="test.classpath"/>
      <include name="**/*.java"/>
   </javac>
</target>



<target name="tests-jar"
        depends="compile-test-classes"
        description="Creates the jar file with all the tests">

   <mkdir dir="${build.tests.lib}"/>

   <!-- Build the tests jar -->
   <jar jarfile="${build.tests.lib}/${build.tests.archive}">
      <fileset dir="${build.tests.classes}">
         <include name="org/jboss/test/messaging/**"/>
      </fileset>
   </jar>
</target>

<!--
   The values from imported files or set by the calling ant tasks will take precedence over
   the values specified below.
-->
<property name="junit.printsummary" value="true"/>
<property name="junit.haltonerror" value="true"/>
<property name="junit.haltonfailure" value="true"/>
<property name="junit.fork" value="true"/>
<property name="junit.includeantruntime" value="true"/>
<property name="junit.timeout" value=""/>
<property name="junit.showoutput" value="true"/>
<property name="junit.jvm" value=""/>
<property name="junit.jvm.options" value=""/>
<property name="junit.formatter.usefile" value="false"/>
```

```
<property name="junit.batchtest.todir" value="${build.tests.output}"/>
<property name="junit.batchtest.haltonerror" value="true"/>
<property name="junit.batchtest.haltonfailure" value="true"/>
<property name="junit.batchtest.fork" value="true"/>
<property name="junit.test.haltonfailure" value="true"/>
<property name="junit.test.haltonerror" value="true"/>

<target name="prepare-testdirs"
        description="Prepares the directory structure required by a test run">
   <mkdir dir="${build.tests.output}"/>
</target>

<target name="tests"
        depends="tests-jar, prepare-testdirs"
        description="Runs all available tests">

   <junit printsummary="${junit.printsummary}"
          fork="${junit.fork}"
          includeantruntime="${junit.includeantruntime}"
          haltonerror="${junit.haltonerror}"
          haltonfailure="${junit.haltonfailure}"
          showoutput="${junit.showoutput}">
      <classpath>
         <path refid="test.classpath"/>
         <pathelement location="${build.tests.lib}/${build.tests.archive}"/>
         <pathelement location="${module.root}/etc"/>
      </classpath>
      <formatter type="plain" usefile="${junit.formatter.usefile}"/>
      <batchtest fork="${junit.batchtest.fork}"
                 todir="${junit.batchtest.todir}"
                 haltonfailure="${junit.batchtest.haltonfailure}"
                 haltonerror="${junit.batchtest.haltonerror}">
         <formatter type="plain" usefile="${junit.formatter.usefile}"/>
```

```
         <fileset dir="${build.tests.classes}">
            <include name="**/messaging/**/*Test.class"/>
            <exclude name="**/messaging/**/performance/**"/>
         </fileset>
      </batchtest>
   </junit>
</target>

<target name="test"
        depends="tests-jar, prepare-testdirs"
        description="Runs a single test, specified by its FQ class name via 'test.classname'">

   <fail unless="test.classname"
         message="To run a single test, use: ./build.sh test -Dtest.clasname=org.package.MyTest"/>

   <junit printsummary="${junit.printsummary}"
          fork="${junit.fork}"
          includeantruntime="${junit.includeantruntime}"
          haltonerror="${junit.haltonerror}"
          haltonfailure="${junit.haltonfailure}"
          showoutput="${junit.showoutput}">
      <classpath>
         <path refid="test.classpath"/>
         <pathelement location="${build.tests.lib}/${build.tests.archive}"/>
         <pathelement location="${module.root}/etc"/>
      </classpath>
      <formatter type="plain" usefile="${junit.formatter.usefile}"/>
      <test name="${test.classname}"
            fork="${junit.batchtest.fork}"
            todir="${junit.batchtest.todir}"
            haltonfailure="${junit.test.haltonfailure}"
            haltonerror="${junit.test.haltonerror}">
      </test>
   </junit>
```

```
   </target>


   <target name="performance-tests"/>


   <target name="functional-tests" depends="tests"/>


   <!-- Clean up all build output -->
   <target name="clean"
      description="Cleans up most generated files.">
      <delete dir="${module.output}"/>
   </target>


   <target name="clobber" depends="clean"/>


</project>
```

9.4. Adding a test requiring a custom JBoss Configuration

Custom JBoss configurations can be added using the `create-config` macro as demonstrated by this `tomcat-sso-tests` target. The `create-config` target has the following attributes/elements:

1. `baseconf` : The existing jboss configuration that will be used as the base configuration to copy
2. `newconf` : The name of the new configuration being created
3. `patternset` : This is the equivalent of the standard patternset element which is used to restrict which content from the baseconf is to be copied into newconf.

In addition, if you need to override configuration settings or add new content, this can be done by creating a directory with the same name as the `newconf` attribute value under the `testsuite/src/resource/tests-configs` directory. In this case, there is a `tomcat-sso` directory which adds some security files to the `conf` directory, removes the jbossweb sar dependencies it does not need, and enables the sso value in the `server.xml`:

```
$ ls -R src/resources/test-configs/tomcat-sso

src/resources/test-configs/tomcat-sso:
```

```
CVS/  conf/  deploy/


src/resources/test-configs/tomcat-sso/conf:

CVS/  login-config.xml*  sso-roles.properties*  sso-users.properties*


src/resources/test-configs/tomcat-sso/deploy:

CVS/  jbossweb-tomcat50.sar/


src/resources/test-configs/tomcat-sso/deploy/jbossweb-tomcat50.sar:

CVS/  META-INF/  server.xml*


src/resources/test-configs/tomcat-sso/deploy/jbossweb-tomcat50.sar/META-INF:

CVS/  jboss-service.xml*
```

The full `tomcat-sso-tests` target is shown here.

```
   <target name="tomcat-sso-tests"
      description="Tomcat tests requiring SSO configured">
      <!-- Create the sso enabled tomcat config starting with the default config -->
      <create-config baseconf="default" newconf="tomcat-sso">
         <patternset>
            <include name="conf/**" />
            <include name="deploy/jbossweb*.sar/**" />
            <include name="deploy/jmx-invoker-adaptor-server.sar/**" />
            <include name="lib/**" />
         </patternset>
      </create-config>
      <start-jboss conf="tomcat-sso" />
      <wait-on-host />
      <junit dir="${module.output}"
         printsummary="${junit.printsummary}"
         haltonerror="${junit.haltonerror}"
         haltonfailure="${junit.haltonfailure}"
```

```
        fork="${junit.fork}"
        timeout="${junit.timeout}"
        jvm="${junit.jvm}">

        <jvmarg value="${junit.jvm.options}"/>
        <sysproperty key="jbosstest.deploy.dir" file="${build.lib}"/>
        <sysproperty key="build.testlog" value="${build.testlog}"/>
        <sysproperty key="log4j.configuration" value="file:${build.resources}/log4j.xml"/>

        <classpath>
           <pathelement location="${build.classes}"/>
           <pathelement location="${build.resources}"/>
           <path refid="tests.classpath"/>
        </classpath>

        <formatter type="xml" usefile="${junit.formatter.usefile}"/>

        <batchtest todir="${build.reports}"
           haltonerror="${junit.batchtest.haltonerror}"
           haltonfailure="${junit.batchtest.haltonfailure}"
           fork="${junit.batchtest.fork}">

           <fileset dir="${build.classes}">
              <patternset refid="tc-sso.includes"/>
           </fileset>
        </batchtest>
     </junit>
     <stop-jboss />
  </target>
```

9.5. Tests requiring Deployment Artifacts

This section describes how to write tests that depend on a deployed artifact such as an EAR.

Deployment of any test deployments is done in the setup of the test. For example, the `HibernateEjbInterceptorUnitTestCase` would add a suite method to deploy/undeploy a `har-test.ear`:

```
public class HibernateEjbInterceptorUnitTestCase extends JBossTestCase {
   /** Setup the test suite.
    */
   public static Test suite() throws Exception
   {
      return getDeploySetup(HibernateEjbInterceptorUnitTestCase.class, "har-test.ear");
   }
...
}
```

If you need to perform additional test setup/tearDown you can do that by extending the test setup class like this code from the `SRPUnitTestCase`:

```
   /** Setup the test suite.
    */
   public static Test suite() throws Exception
   {
      TestSuite suite = new TestSuite();
      suite.addTest(new TestSuite(SRPUnitTestCase.class));

      // Create an initializer for the test suite
      TestSetup wrapper = new JBossTestSetup(suite)
      {
         protected void setUp() throws Exception
         {
            super.setUp();
```

```
            deploy(JAR);
            // Establish the JAAS login config
            String authConfPath = super.getResourceURL("security-srp/auth.conf");
            System.setProperty("java.security.auth.login.config", authConfPath);
         }
         protected void tearDown() throws Exception
         {
            undeploy(JAR);
            super.tearDown();
         }
      };
      return wrapper;
   }
```

9.6. JUnit for different test configurations

We use the ant-task `<junit>` to execute tests. That task uses the concept of formatters. The actual implementation uses the XML formater by specifying type="xml" in the formatter attribute.

If we need to execute the same test more than once, using this default formatter will always overwrite the results. For keeping these results alive, we have created another formatter. So, use these steps to keep JUnit results between different runs:

Define the sysproperty `"jboss-junit-configuration"` during the jUnit calls. Change the formatter and set a different extension for keeping the files between different executions:

Set the class by `classname="org.jboss.ant.taskdefs.XMLJUnitMultipleResultFormatter`

Here is a complete example of the changes:

```
      <junit dir="${module.output}"
         printsummary="${junit.printsummary}"
         haltonerror="${junit.haltonerror}"
         haltonfailure="${junit.haltonfailure}"
         fork="${junit.fork}"
         timeout="${junit.timeout}"
```

```
        jvm="${junit.jvm}"
        failureProperty="tests.failure">
         ....
        <sysproperty key="jboss-junit-configuration" value="${jboss-junit-configuration}"/>
        <formatter classname="org.jboss.ant.taskdefs.XMLJUnitMultipleResultFormatter" usefile="${junit.formatter.usefile}" extensi
        .....
    <junit/>
```

10

Support, Sales Force and Patch Management

10.1. An overview of the Support Process

JBoss has a very active and involved support process.

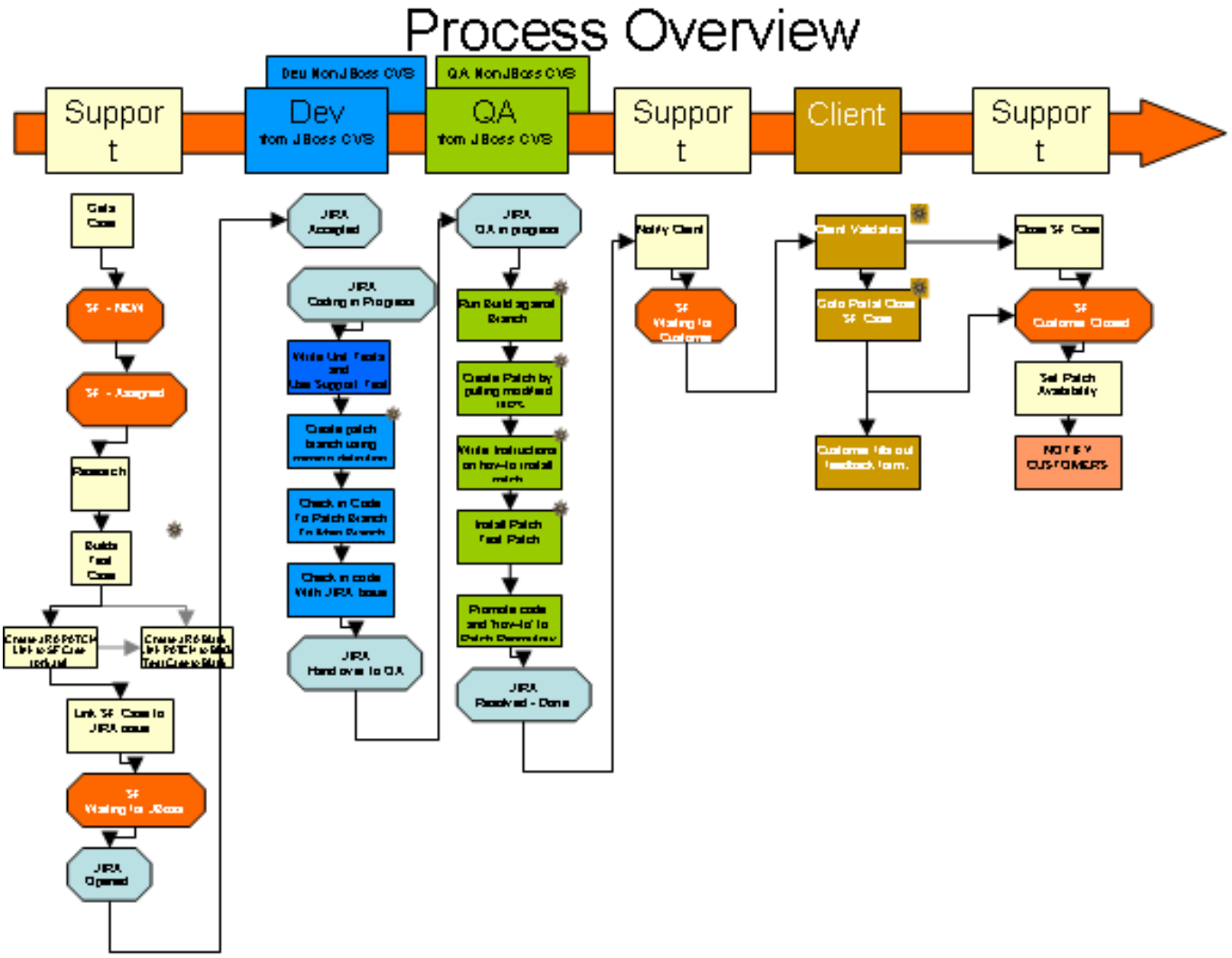
As the support team gets a new case, it researches the issue and builds an appropriate test case. An issue is opened in JIRA, the project management tool used by JBoss.

The onus now shifts onto the Development team, where a developer is tasked with the process of fixing the issue. The developer designs and codes the fix and checks in the fix to CVS, the version control system, on the appropriate Main/Patch branch.

The JBoss QA team now validates the issue. QA first runs the build against the branch, then creates the patch by pulling together the modified jars. Instructions on how to install the patch are written and thereafter the QA team installs and tests the patch. If the behaviour is as expected, the QA team promotes the code and the how-to instructions to the Patch repository. At this point, the issue is resolved in JIRA.

The JBoss Support team then informs the customer that the issue is fixed. Once the client has validated the issue, the Support Force marks the issue closed. Customer feedback on the issue is also noted. At this point, the patch can also be made available to other customers if they are in need of a similar fix.

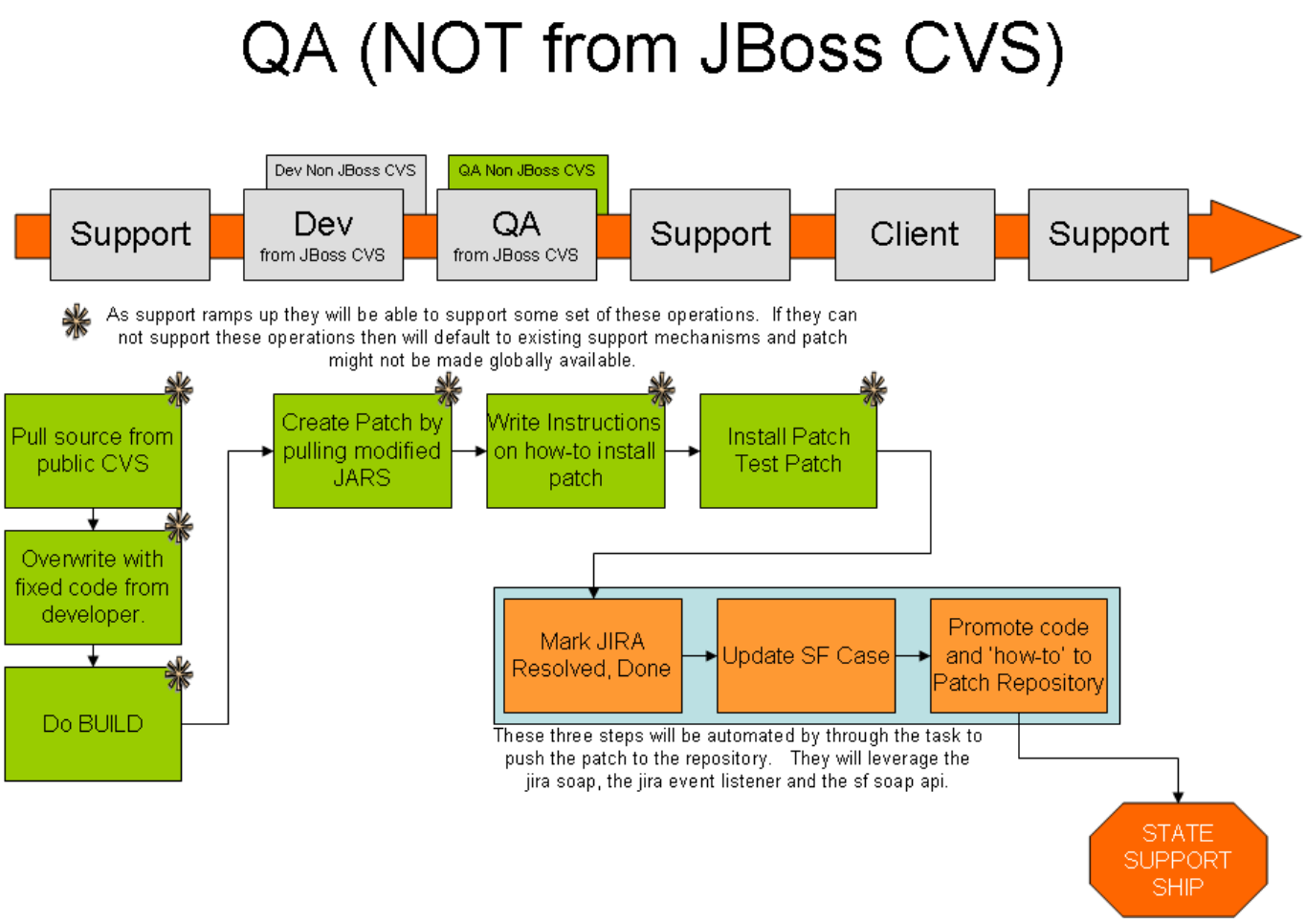
The following image describes the support process:



10.2. Patch Management

Patch Management is mostly the work of the QA team at JBoss. A patch for an issue is created by the QA team after running the build against the branch and updating the build with the modified jars. The QA team writes instructions on how-to install a patch. The created patch is then installed and tested. If the behaviour satisfies the issue requirements, the issue is marked resolved in JIRA. The patch install instructions and the patch itself is committed to the patch repository.

The following image describes the process:

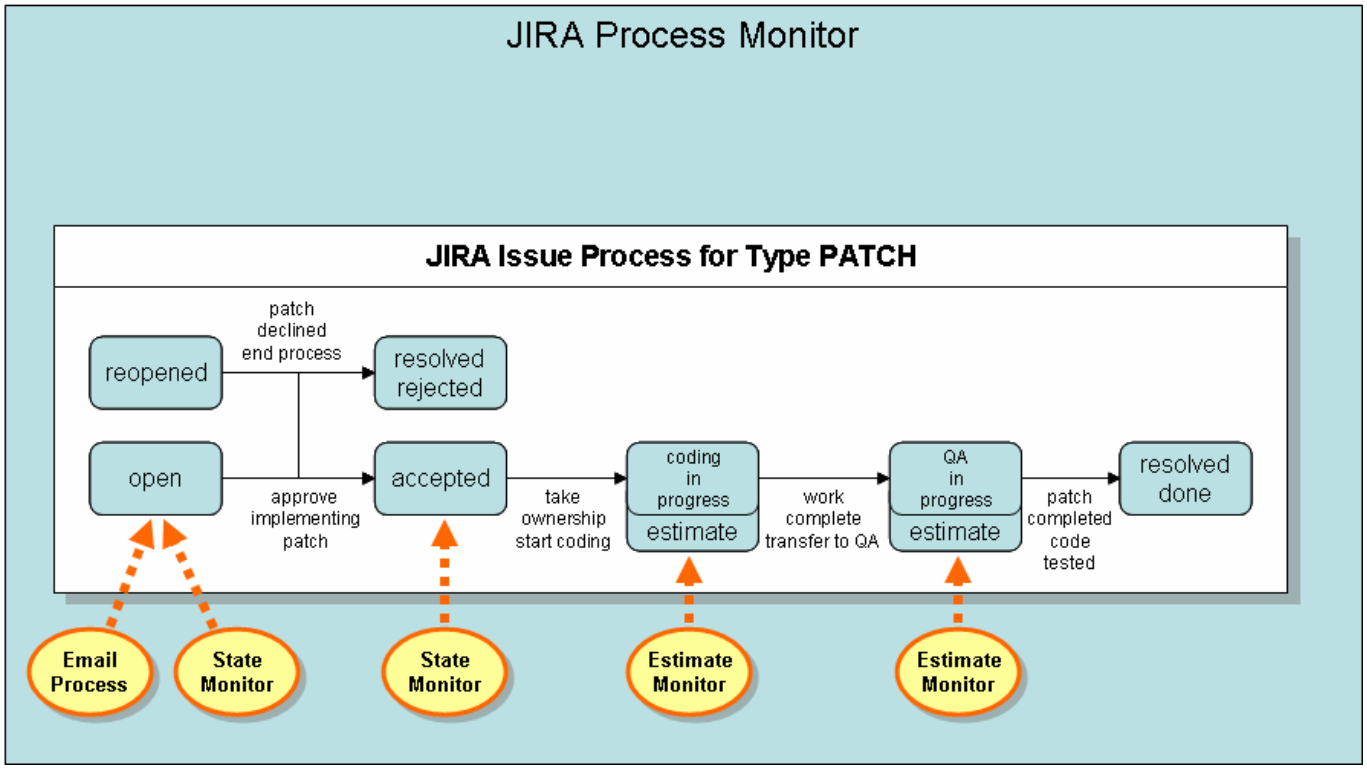


10.3. Monitoring the Support Process

JIRA is used for Project Management as well as proces monitoring. Once an issue is opened, its state is monitored until it is accepted and assigned to a developer. The developer then estimates the amount of time he / she would need to spend working on the issue, and starts working on the issue. Once implementation is complete, it is handed off to QA. QA also makes estimates on the time required to install and test the patch. Monitoring of both Development and QA estimates are done through JIRA.

The following figure illustrates the process:

Support Process Monitor
jira



11

Weekly Status Reports

Every JBoss employee sends a weekly status report to his / her manager on the first working day of every week.

This reporting scheme has been established to monitor work progress on outstanding issues and bottlenecks if any.

The format is as follows:

1. Work done last week:
   This includes:

   a. Development tasks accomplished and the approximate time overall.

   b. Support tasks undertaken and approximate time spent on support.

   c. Remote consulting tasks undertaken and approximate time spent on them.

   d. Any On-site consulting or training and approximate time taken.

   e. Preparing for on-site consulting or training and approximate time taken.

2. Work planned for the current work week.

3. Outstanding issues that require others' help.

4. Any other relevant issues.

12

Documentation and the Documentation Process

12.1. JBoss Documentation

JBoss Inc. provides a wide selection of documentation that provides in-depth coverage across the federation of Professional Open Source projects. All documentation is now free. Several versions of our documentation require registration to the JBoss website, which is also free. If you cannot find the answers that you are looking for, you can get additional support from the following sources:

- Buying Professional Support [http://www.jboss.com/services/profsupport] from JBoss Inc. and getting answers from the experts behind the technology.
- Searching the Wiki [http://www.jboss.com/wiki/wiki.jsp].
- Reviewing the Forums [http://www.jboss.com/index.html?module=bb].
- Watching JBoss Webinars [http://www.jboss.org/services/online_education].

A complete listing of the documentation by project can be found on the Document Index [http://www.jboss.com/docs/index].

12.2. Producing and Maintaining Quality Documentation

For JBoss developers and documentation writers, JIRA and docbook are the two key tools to integrate the documentation process in the development workflow. Now let's clarify documentation responsibilities and adopt a simple process to guarantee our documentation is always accurate and up-to-date.

12.2.1. Responsibilities

12.2.1.1. The product team

The development team is responsible for product-specific documentation. Core developers need to maintain the following documents.

- The product reference guide
- The Javadoc for key APIs and all annotations
- Annotated test cases

- Optional user guides for a specific product
- Optional flash demo for a specific product

Tasks related to producing those documents are managed within the development project's JIRA module. Most of these tasks are assigned to developers within the project but some of them are assigned to documentation team, as we will see in a minute.

12.2.1.2. The documentation team

The documentation team (Michael Yuan and Norman Richards) is responsible for all "cross-cutting" documents that cover several projects, as well as tutorial / technical evangelism materials. Examples of such documents are as follows.

- Overall server guide
- Trail maps (interactive tutorials)
- Sample applications
- Books and articles
- The "what's new" guide
- The "best practice" guide
- etc.

Tasks related to those documents are managed inside the "documentation" JIRA module. Developers are welcome to raise issues there if you see errors and/or coverage gaps in existing documents.

12.2.2. Product documentation review

Before each product release, the documentation team needs to review all the documents maintained by project's core developers (e.g., reference guide and Javadoc). Please create a review task for each document within your project and assign it to a member in the documentation team. The documentation team will read the draft and use that JIRA task to track any issues.

12.2.3. Keep the documentation up-to-date

Since our technology is fast evolving, it is crucial for us to keep the documents up-to-date. If you have any development task that might affect the external interface or observed behavior of the product, please check the appropriate "affects" check box at the bottom of the JIRA task information page.

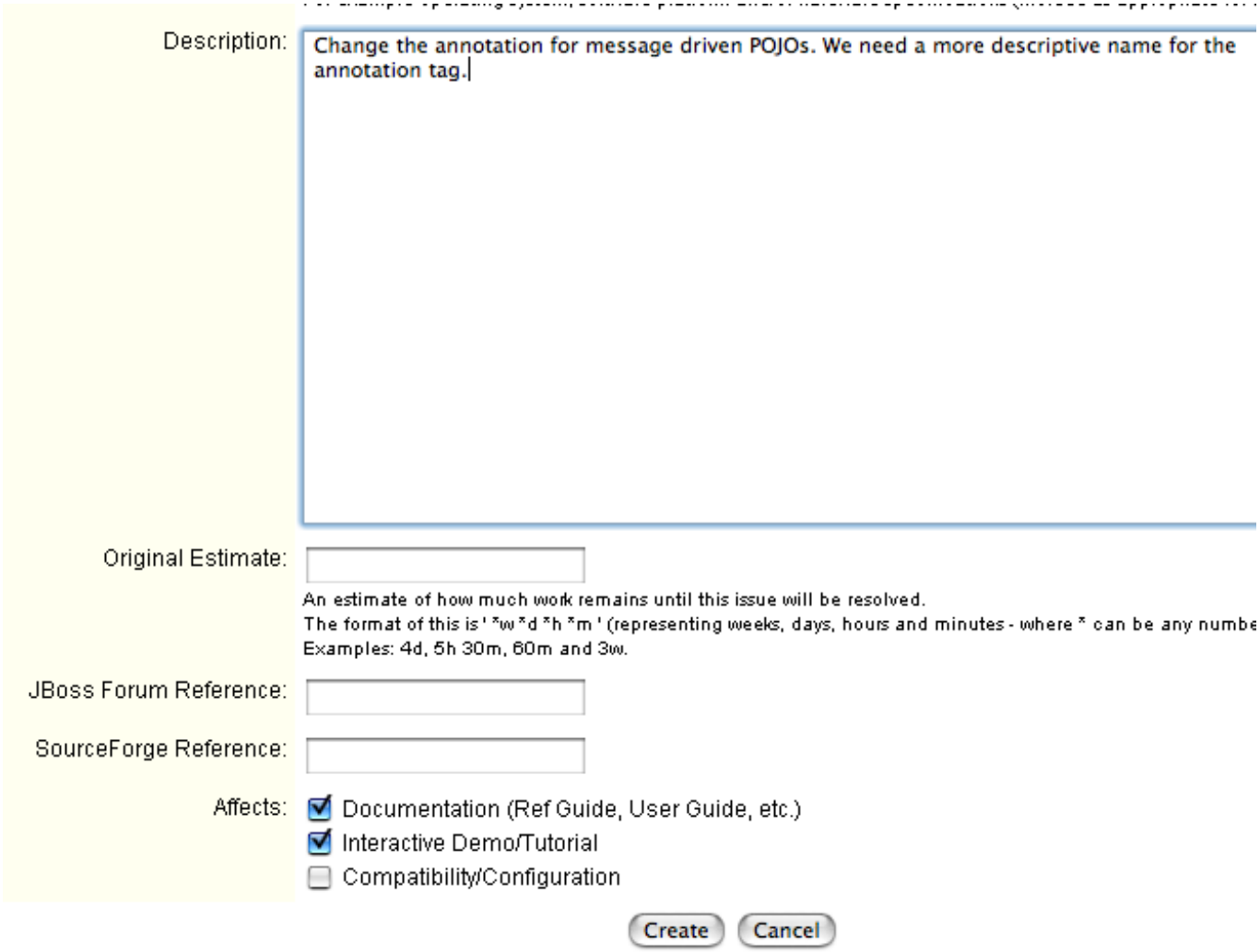


Figure 12.1. Check the "affects" boxes for a task that changes the public API

- The project's documentation maintainer searches those tagged tasks periodically to update the reference guide etc.
- The documentation team searches those tagged tasks periodically to update the cross-product documents.

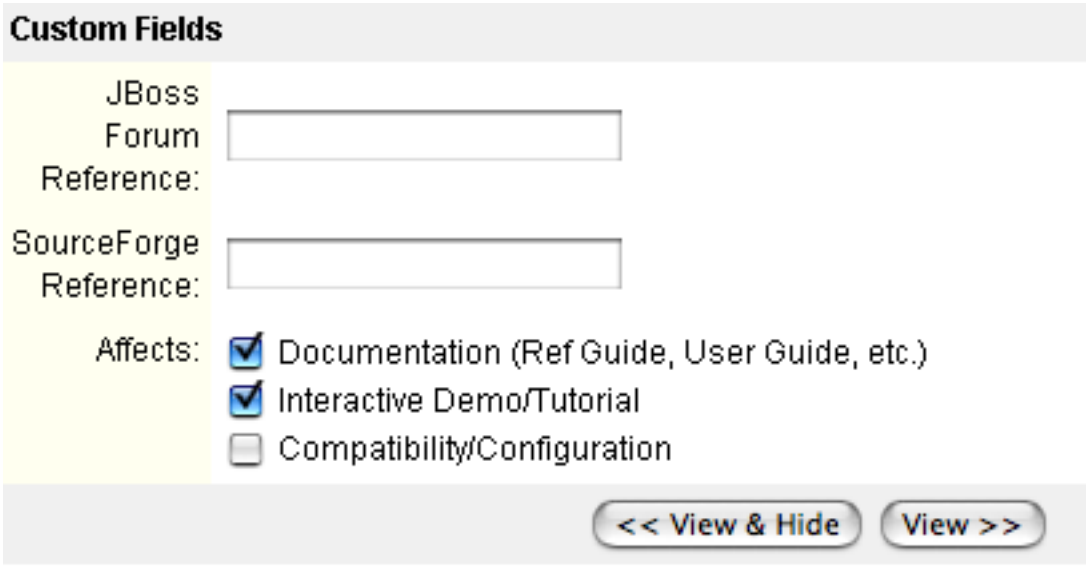


Figure 12.2. Find all tasks that affect docs

12.2.4. Articles and books

The documentation team also serves as our internal editors for technical articles and books in the JBoss book series. If you are interested in writing articles or books, please let us know. Even if you do not have time to write a whole book, we might still find books / articles you can contribute to. So, it is important to keep us informed about your interests in this area.

The documentation team will help develop proposals and manage the relationship with outside editors. If you sign up to write the article / book, a JIRA task in the documentation module would be created and assigned to you to keep track of the progress.

12.2.5. Authoring JBoss Documentation using DocBook

Writing JBoss documentation using the centralized docbook system is really easy. You first need to check out the docbook-support top level module:

```
cvs -d:ext:yourname@cvs.sf.net:/cvsroot/jboss co docbook-support.
```

In the module, you can find the `docs/guide` directory. Copy that directory to the `docs/` directory in your own project and use it as a template for your own docbooks.

For more information about how the directories and build tasks are organized, check out the guide doc in the doc-book-support module:

The PDF version is `docs/guide/build/en/pdf/jboss-docbook.pdf`

The HTML version is `docs/guide/build/en/html/index.html`