JBoss Development Process Guide

2004, Ivelin Ivanov, Ryan Campbell, Pushkala Iyer, Clebert Suconic, Mark Little, Andrig Miller, Alex Pinkin

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

Productizing Steps in the Overall Release Process

Title	Productizing Process for JEMS
Author	Andrig (Andy) Miller
Creation Date	February 9, 2006
Status	Final
Revision	1.0.1
Filename	Productizing Process for JEMS

Date	**Revision**	**Status**	**Author**	**Description**
February 9, 2006	0.1	Draft	Andrig (Andy) Miller	Initial version.
February 20, 2006	0.2	Draft	Andrig (Andy) Miller	Incorporated feedback from Pierre Fricke.
February 21, 2006	0.3	Draft	Andrig (Andy) Miller	Incorporated feedback from Shaun Connolly.
February 22, 2006	0.4	Draft	Andrig (Andy) Miller	Incorporated feedback from Ryan Campbell.
February 28, 2006	0.6	Draft	Andrig (Andy) Miller	Incorporated feedback from Scott Stark.

Date	Revision	Status	Author	Description
February 28, 2006	0.8	Draft	Andrig (Andy) Miller	Incorporated feedback from Sacha Labourey.
March 7, 2006	0.9	Draft	Andrig (Andy) Miller	Incorporated feedback from Ivelin Ivanov.
March 7, 2006	0.9.1	Draft	Andrig (Andy) Miller	Incorporated feedback from Pierre Fricke.
March 20, 2006	0.9.9	Release Candidate	Andrig (Andy) Miller	Incorporated feedback from Adrian Brock, Andrew Oliver, Manik Surtani, Ben Wang, and Ben Sabrin.
March 24,2006	1.0.0	Final	Andrig (Andy) Miller	Incorporated feedback from Andy Oliver, Rich Friedman, Sacha Labourey, and Scott Stark.
April 11, 2006	1.0.1	Final	Andrig (Andy) Miller	Incorporated Pierre's and Shaun's web related checklist for releasing projects, that came from our lessons learned discussion with JBoss Messaging.

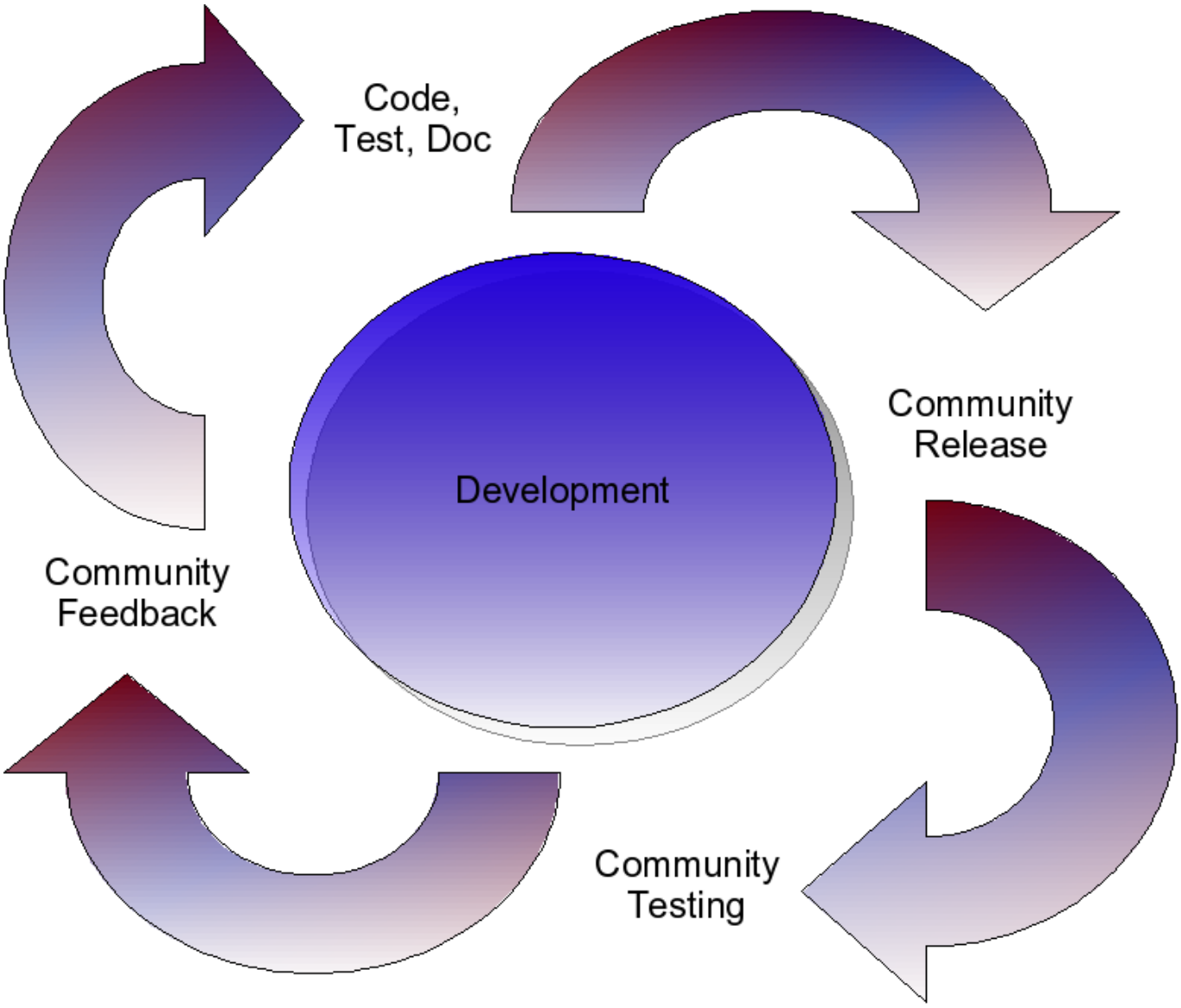
2.1. I. Background

This document will define the process to turn a release of an open source project into a revenue generating product for JBoss, Inc. It is NOT an all encompassing document in that regard though. This document focuses only on those aspects that are led by the development/engineering organization. It does not delve into what product management and services processes are for productizing a JBoss project. However, there are tasks described in this document that do involve product management and services as contributors to the development process in regards to productizing our projects.

This document will not try to dictate process within the development life-cycle of each project, but instead concentrate on the steps that are not directly related to development, but to product.

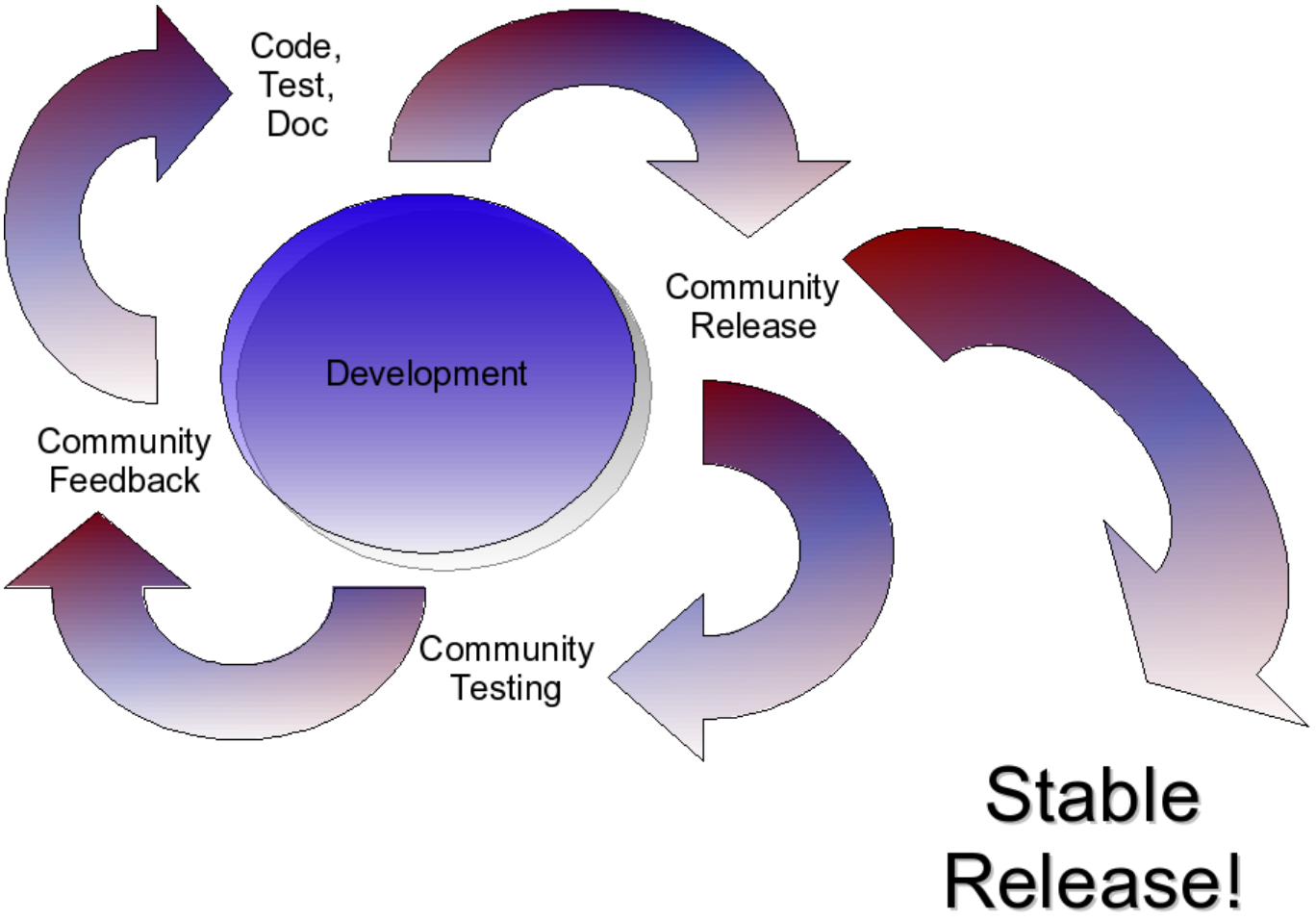
2.2. II. Turning a Project into a Product

Below is an illustration of the development life-cycle of our projects:



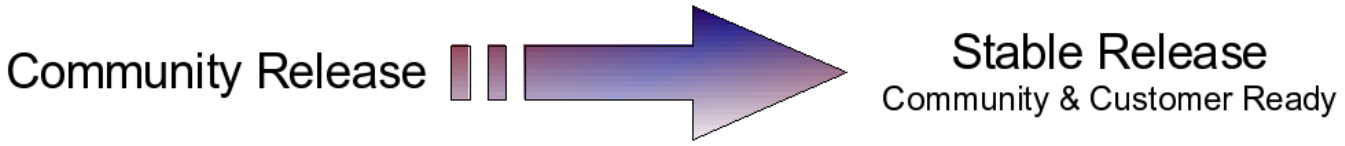
The above illustration is just one way to visualize the development life-cycle of a project. Certainly, it is not a true spiral or circle, since all of the tasks above can and do happen in parallel.

At a given point in time, this life-cycle above stops to release something that is considered more than just a work in-progress, but something that contains a feature set, and a level of quality that the lead developer(s) are happy with. This is considered the “stable” release. This release is the one that we will focus on in terms of becoming a product.



The above illustration shows that point-in-time when a “stable” release is dropped. Normally, this is solely decided by the lead developer(s) based on the feature set and quality that they deem fit for the label of “stable” release. In this transition from developing code, testing, community testing and feedback, to a stable release the process for turning this into a product for JBoss, Inc. must run in parallel.

Productize



In this transition from community releases to a stable release, or a final release, that is ready for customer consumption, the developer cannot make the decision in isolation about when that stable or final release will be. This must be done in conjunction with all of the stakeholders identified in the following sections. If the productizing steps have not been completed, yet the software is ready, it will not be officially released. We can call it a release candidate, but not stable or final until all the productizing steps, that have been agreed to, are complete.

There are seven main areas that I would like to focus on in regards to taking software that is being developed in this life-cycle, and getting it to a point where it is ready to be released as a product.

1. Product Road Map Creation and Maintenance
    a. Features
    b. Productizing Tasks
    c. Known Bugs
    d. Improvements to Existing Features
2. Reference documentation
    a. API Reference
    b. Administration Guide
    c. User Guide
3. On-line education
    a. Trailblazers
    b. Demonstrations
4. Training materials
    a. Internal training materials
        i. Support organization
        ii. System Engineers
        iii. Consultants
    b. External training materials
        i. Customers
        ii. Partners
5. Quality Assurance
    a. Performance testing

   b. Scalability testing
   c. Soak testing
   d. Integration testing
   e. Availability testing
   f. Certification testing
6. Development Tooling
   a. JBoss IDE support for developers
7. Release the stable or final release
   a. Community announcements

In the following sections, the “Who Does It?” column describes roles that are played. This is not meant to dictate that those tasks are done by non-development resources. The exception to this is product management and services. All other roles can be performed by the project developers, whether the are JBoss employees or outside contributors, if they so choose to do so. We would like the project to perform their work in as flexible a manner as possible. What we care about is delivering a high-quality product as quickly as possible, not who specifically does the work.

2.2.1. I. Product Road Map Creation and Maintenance

A product road map should be developed and maintained for each release of the project. It should contain at least the following:

1. List of planned new features for the release
2. The productizing tasks that are needed for the release
3. Address the high priority know bugs or issues with the previous release(s)
4. Improvements to existing features

The list of planned new features for the release should be discussed in the forums, and with developers from dependent projects and with product management. The interdependencies of many of our projects makes this critical. Feedback from our customers, in the form of surveys, support cases that are feature requests, feature requests from the forums, etc., should all be incorporated to come up with this list. The finalized road map should be in sync with product management’s product plans.

The productizing tasks that are discussed throughout this document should be incorporated into the road map. For example, let’s say that you don’t currently have a soak test, and that needs to be developed, tested, and executed to

complete the productizing tasks for the next release. Then you would add that to the road map, and that would turn into JIRA tasks.

Where the known bugs and issues are concerned any fixes from previous releases that have to be reconciled due to overlapping development needs to be addressed. The road map shouldn't necessarily address each one of those individually, but just make sure that the overall task is taken into account in the plan. Also, some of these issues may not be bugs, but merely that code needs to be re-factored or that performance enhancements have been identified in particular areas from either customers or our own testing, etc.

Improvements to existing features can take many forms, depending on the project. It may involve making a particular feature easier to use, it may involve making a feature easier to manage, etc.

A good example of a product road map is the following from the Portal project:

Portal 2.4 Road Map [http://wiki.jboss.org/wiki/Wiki.jsp?page=Portal_2_4Roadmap]

Some questions that are good to ask yourself as you prepare the road map are:

2.2.1.1. Questions for Constructing the Road Map

1. What do you plan to do for this release?
2. Have you prioritized the work?
3. Is the work specified?
4. Have you discussed it with others to validate the ideas?
5. Have you used feedback from your users?
6. What do you need from other projects?
7. Who uses your project? How will they be affected?
8. What do other projects want from you?
9. Do you have tasks for all the productizing that needs to be done?
10. Have you scheduled time for the productizing work, i.e. taken it into account, when estimating, what can be done for a release?

Other potential issues:

1. What do you plan to deprecate?
2. What do you plan to remove or retire?
3. Should you really be doing that in your project?

4. Is the work already done elsewhere? Don't fall into the "Not Invented Here" trap.
5. What third-party dependencies do you plan to introduce?
6. How will that third-party software be supported?
7. What is the license for that third-party software?

One thing that I would like to stress, is that we want to create releases of reasonable size. The product road map shouldn't contain every possible feature, fix, issue, etc. Use your best judgment in what can be delivered in a reasonable amount of time and prioritize accordingly. Remember, release early and often is the goal. There are no rules of thumb for how often releases should be made, because that is highly dependent on the project, its maturity, and the market demands.

2.2.1.2. Project road map checklist:

Task	Description	Who Does It?	When Is It Delivered?
Road map creation.	Define the features, fixes, issues, productizing tasks, and improvements to be made for a given release of the project.	Development with feedback from other projects, the community, customers and product management.	Delivered before work starts on the release.
Publish road map.	Post the road map on the jboss.org website (jboss wiki is a good tool for this), and create the JIRA release with associated tasks.	Development.	Delivered before work starts on the release.

2.2.2. II. Reference Documentation

Reference documentation is produced in parallel with developing the code, and should evolve through the community releases, such as Alpha, Beta and Release Candidate releases. Reference documentation is an area that we do quite well, as illustrated by the examples below:

The JBoss 4 Application Server Guide [http://docs.jboss.com/jbossas/jboss4guide/r4/html/]

HIBERNATE - Relational Persistence for Idiomatic Java [http://www.hibernate.org/hib_docs/v3/reference/en/html/]

JBoss Portal 2.2 Reference Guide [http://docs.jboss.com/jbportal/v2.2/reference-guide/en/html/]

JBoss jBPM 3.1 Workflow and BPM made practical [http://docs.jboss.com/jbpm/v3/userguide/]

TreeCache: a Tree Structured Replicated Transactional Cache [http://docs.jboss.com/jbcache/1.2.4/TreeCache/html/]

JBoss Microcontainer Reference [http://docs.jboss.org/nightly/microkernel/docs/reference/en/html/]

SEAM - Contextual Components A Framework for Java EE 5 [http://docs.jboss.com/seam/reference/en/html/index.html]

JBoss EJB 3.0 Reference Documentation [http://docs.jboss.org/ejb3/app-server/reference/build/reference/en/html/index.html]

These examples, all go over the public API of the software at a minimum. Installation, configuration and on-going administration, if applicable should also be covered. They are almost all published on the docs.jboss.org site, and they should all be accessible from there, even if they are not directly hosted there. Hibernate and Apache Tomcat documents are examples of this. The documents should be published in HTML, for easy on-line viewing, PDF for printing purposes, and if there are any examples, then the source code for those examples should be provided in an archive format such as zip.

Each project will certainly have different levels of documentation that is needed, but a plan for what the minimum for each project should be put together in conjunction with product management. Also, the documentation team, should be involved in creating the documentation, to make it navigable, presentable, and published in the two different formats required. A recommended set of documentation would be:

1. API Reference

    a. The API reference should contain a definition of the public API. This public API will be backward compatible between minor releases. Private APIs can change between all releases, with the exception that they need to preserve binary compatibility. There is a more complete statement on API stability and compatibility between releases in the JBoss Product Versioning Wiki.

2. Administration Guide

3. User Guides

    a. The user guide should contain, if applicable, a list of unsupported or experimental features that may be present, but are not recommended for production use.

2.2.2.1. Project reference documentation checklist:

Task	Description	Who Does It?	When Is It Delivered?
Minimum Content Definition.	Define the minimum documentation set that	Product management and development.	Delivered prior to the first alpha release[a].

Task	Description	Who Does It?	When Is It Delivered?
	should be produced for the project (e.g. API reference, administration guide, user guide).		
Documentation Creation.	Create the minimum defined documentation set for the project.	Development and the documentation team.	Delivered with each release, and iteratively updated as the project progresses through alphas, betas, release candidates through to a stable release.

[a]This may only be applicable to new projects that haven't had a stable release yet. Of course, a project may go through a significant structural change to warrant a change in the content definition.

Note: The definition for alpha, beta, and release candidate are in the JBoss Product Versioning Wiki page. It is under the heading, "Current Qualifier Conventions (Post 2006-03-01)". Here is the link:

JBoss Product Versioning [http://wiki.jboss.org/wiki/Wiki.jsp?page=JBossProductVersioning]

2.2.3. III. On-line Education

On-line education primarily consists of two elements. Trailblazers and demonstrations. The Trailblazers and demonstrations should be produced in parallel with the code and should evolve through the community releases, such as Alpha, Beta and Release Candidate releases. Examples of these are as follows:

EJB 3.0 Trailblazer [http://trailblazer.demo.jboss.com/EJB3Trail/]

JBoss Seam DVD Store Demonstration [http://dvdstore.demo.jboss.com/home.faces;jsessionid=C1B94FC67E91765EFFBFC1DE3831E9A8]

Overall page for Trailblazers and Demonstrations [http://www.jboss.com/docs/demos]

2.2.3.1. Project on-line education checklist:

Task	Description	Who Does It?	When Is It Delivered?
Needs assessment for trailblazer and demonstration.	Determine whether trailblazers and demos are needed to help market the	Product management and development.	Delivered prior to the first alpha release[a].

Task	Description	Who Does It?	When Is It Delivered?
	project, and to help with adoption.		
Trailblazer Creation.	Create the minimum defined documentation set for the project.	Development and the documentation team.	Delivered with each release, and iteratively updated as the project progresses through alphas, betas, release candidates through to a stable release.
Demonstration Creation.	Write an application or record a demo of the usage of the project, whichever is appropriate.	Development and the documentation team.	Delivered with each release, and iteratively updated as the project progresses through alphas, betas, release candidates through to a stable release.

[a]This may only be applicable to new projects that haven't had a stable release yet. Of course, a project may go through a significant structural change to warrant a change in the content definition.

2.2.4. IV. Training Materials

There are five distinct audiences for training materials. First, and foremost is our customers. This training targets the developers and administrators that will be using our technology to develop applications and support applications respectively. Second, is the support organization. In order for them to be able to be as self-sufficient as possible, they need training. This training needs to be detailed enough that it helps them be able to troubleshoot issues that customers have. Third is our consultants. They need the same level of training as support, in that they will not only be helping to develop solutions in concert with our customers, but they will be the first line of support in solving development related issues (troubleshooting ability is key). Fourth is our system engineers. They need training similar to the consultants, in that they will be in front of prospective customers, and may have to delve into technical details during pre-sales activities. And fifth, is our partners. The training for our customers is what is, and will still be, used for their training.

The training materials for developers, can certainly fill part of the training needs for support, consulting and our partners. It will need to be augmented with training that is helpful for troubleshooting customer problems. This training material should have instructions on where errors are logged, and typical reasons that exceptions are thrown, etc.

2.2.4.1. Training materials checklist:

Task	Description	Who Does It?	When Is It Delivered?
Define structure of training materials for customers.	Define the number and types of classes that the training materials need to support (e.g. Beginner, advanced, etc.).	Development with input from services.	The definition should be defined prior the first beta release[a].
Needs assessment for support, consulting and sales engineers.	Do a troubleshooting assessment for support/ consulting, so that additional training materials, or training sessions determined. At a minimum this should contain a triage list for first line support that identifies what information needs to be collected for problem resolution.	Development and product management with services.	The needs assessment should be complete prior to the first release candidate.
Develop training materials.	Develop the identified training materials for customers and from the needs assessment for support.	Development and documentation team.	The training materials should be delivered for testing purposes at the same time as the first release candidate[b].
Test training materials.	Test the training materials in a real classroom setting to make sure that the training materials are accurate, and that the labs actually work.	Services with feedback to development[c].	Testing of training materials should be done at the time of the first release candidate.

[a]This may only be applicable to new projects that haven't had a stable release yet. Of course, a project may go through a significant structural change to warrant a change in the current structure.
[b]This may have exceptions for newer technology or projects that we don't anticipate will have significant uptake in the market on their initial stable release. These exceptions will be made on a case-by-case basis by product management.
[c]The feedback should take the form of JIRA issue for the project, that would be defined as a blocking issue for a stable release.

Note: The definition for alpha, beta, and release candidate are in the JBoss Product Versioning Wiki page. It is under the heading, "Current Qualifier Conventions (Post 2006-03-01)". Here is the link:

JBoss Product Versioning [http://wiki.jboss.org/wiki/Wiki.jsp?page=JBossProductVersioning]

2.2.5.

2.2.6. V. Quality Assurance

All JEMS projects should be going through a standard quality assurance process[1]. There are five areas that need to addressed for projects where quality assurance is concerned.

The first area is performance testing. Every JEMS product should have a performance test that measures straight line performance of the product (single virtual user/client). The second area is scalability testing. Every JEMS product should have a scalability test that measures performance under high concurrent usage scenarios (many virtual users, clients, nodes, etc). This will be different depending on the JEMS product you are considering, but we should be able to sustain straight line performance levels (or at least not degrade very much) with high concurrency. The third area is soak testing. Every JEMS product should have a soak test that demonstrates sustained high performance with high concurrency over a long duration of time (catch issues like unintended object retention, leaked file descriptors, garbage collection issues, etc.). This test will run for a minimum of 24 hours. The fourth area is integration testing. All of the JEMS components that can be used in conjunction with each other for an application that our customers may develop or deploy, should be tested under scenarios that have them work together. The fifth area is availability testing. In this test, we should use our scalability scenarios and create fault conditions, so that we have a system under high concurrent usage, and are able to measure the ability to have failures and continue running.

These are all important quality aspects that our customers will expect to have nailed with each and every release of JEMS products. The testing described above is not meant to replace the existing unit test suites that each project already executes through their build process, or is it meant to replace any performance testing that each project may already have in place. What is described could very well leverage existing tests that projects already have.

2.2.6.1. Quality assurance checklist:

Task	Description	Who Does It?	When Is It Delivered?
Define performance test scenarios.	For each JEMS project, there should be a performance test(s) scenarios defined with a goal for what the straight line performance should be (this could be relative to a baseline release, or relative to a competitors number, etc.)	QA and development.	Complete prior to the first release candidate[a].
Define scalability test scenarios.	For each JEMS project, there should defined what constitutes high concurrency for that given project, and what levels	QA and development.	Complete prior to the first release candidate[b].

[1]The probable exception to this rule is JBoss IDE, as it is a development tool, and really doesn't fit the profile for these types of tests. Of course, JBoss IDE should do some form of performance testing around UI responsiveness and memory footprint.

Task	Description	Who Does It?	When Is It Delivered?
	of concurrency should be the goal.		
Define integration test scenarios.	For JEMS as a whole, scenarios should be defined that cross all of the integration points of the JEMS (e.g. Application that has a web tier that uses JSF/Seam, it has a middle tier that uses Stateless and Stateful session beans, and implements a workflow through jBPM, persists through EJB3/Hibernate, etc.)	QA and development.	Complete prior to the first release candidate of the application server (this is where most of the integration comes into play)[c].
Define availability test scenarios.	For each project, and for JEMS as a whole, test scenarios that inject failures in a high-availability configuration should be defined. Fault injection can be done through many techniques. Some as simple as unplug the network cable from a system, to some as sophisticated as having an aspect that is deployed that inject exceptions into the running application.	QA and development.	Complete prior to the first release candidate[d].
Build performance test.	Develop the appropriate test scripts to automate the performance test.	QA and development.	Complete and ready to execute by the time we offer silver support.
Build scalability test.	Develop the appropriate test scripts to automate the scalability test.	QA and development.	Complete and ready to execute by the time we offer silver support.
Build integration test.	Develop the appropriate	QA and development.	Complete and ready to

Task	Description	Who Does It?	When Is It Delivered?
	test scripts to automate the integration test.		execute by the time we offer silver support.
Build availability test.	Develop the appropriate test scripts to automate the availability tests[e].	QA and development.	Complete and ready to execute by the time we offer silver support.
Execute performance test.	Run the test.	QA and development.	Should be complete prior to moving project to support levels above silver.
Execute scalability test.	Run the test.	QA and development.	Should be complete prior to moving project to support levels above silver.
Execute soak test.	Run the test.	QA and development.	Should be complete prior to moving project to support levels above silver.
Execute integration test.	Run the test.	QA and development.	Should be complete prior to moving project to support levels above silver.
Execute availability test.	Run the test.	QA and development.	Should be complete prior to moving project to support levels above silver.

[a]This may not be needed for each and every release, as once they are developed the scenarios may not change. However, the goals of the runtime may change with each release. Therefore, at a minimum we should evaluate the goals with each release.
[b]Same as seven.
[c]This will probably only have to be done when there is a new JEMS component, once it is done the first time.
[d]Same as seven.
[e]This may be manual, unless it is identified that aspects are needed to be developed that will inject exceptions into the running application.

Note: The definition for alpha, beta, and release candidate are in the JBoss Product Versioning Wiki page. It is under the heading, “Current Qualifier Conventions (Post 2006-03-01)”. Here is the link:

JBoss Product Versioning [http://wiki.jboss.org/wiki/Wiki.jsp?page=JBossProductVersioning]

2.2.7. VI. Development and Management Tooling

Many of the JEMS projects have development and management tooling requirements. Developers in our target cus-

tomer base, are typically not the highest skilled developers[2], and certainly administrators need information and tools to make them more productive, especially in large deployments. Therefore, there is a need to make our platform as accessible to developers and administrators as possible through appropriate tooling.

2.2.7.1. Development tooling checklist:

Task	Description	Who Does It?	When Is It Delivered?
Define development tool needs.	Define what developers need to be productive developing against a specific project.	Product management and development.	Complete by the first alpha release.
Develop tools.	Build the tools that have been defined for developers.	Development.	Complete by the time the release is considered stable[a].
Define management needs.	Define what administrators need to be productive managing a production deployment.	Product management and development.	Complete by the first alpha release.
Develop management tools.	Build the tools, and the features within the product to expose management information for administrators.	Development.	Complete by the time the project moves to support levels above silver.

[a]This may or may not be a hard requirement depending on the project, and the market adoption rate. Exceptions to this should be approved by product management.

Note: The definition for alpha, beta, and release candidate are in the JBoss Product Versioning Wiki page. It is under the heading, “Current Qualifier Conventions (Post 2006-03-01)”. Here is the link:

JBoss Product Versioning [http://wiki.jboss.org/wiki/Wiki.jsp?page=JBossProductVersioning]

2.2.8. VII. Release the stable or final release

Again, to reiterate what has been said already. The designation of the final release should never be done in a vacuum. This should be coordinated through product management. No one within JBoss should read about a release of our software without knowing about it in advance. This allows us to coordinate all public relations activities, as

[2]If you have every heard the presentation given by Dave Thomas of The Pragmatic Programmers” titled “Herding Race Horses, and Racing Sheep” you have seen empirical evidence of the fact that most developers are either Novices or Advanced Beginners, and they are NOT competent.

well as do a final check on whether sales and services are truly ready to go.

2.2.8.1. Release checklist:

Task	Description	Who Does It?	When Is It Delivered?
Review release content and timing.	Review the plan for what will be released, in terms of web site content, and exactly what the timing will be.	Product management with development.	Approximately four weeks prior to final release.
Review product data sheet.	Review the product data sheet with product management, to make sure that it is accurate.	Product management with development.	Approximately one to two weeks in advance of the final release.
Review press release.	Review the press release with product management for accuracy of information.	Development with product management.	Approximately one week in advance of the final release.
Verify that agreed to productizing steps are complete.	Make sure that all the agreed and applicable productizing steps from sections one through five have been completed.	QA led with project management and development.	Just before final release.
Verify license conformance for distribution.	Verify that the final distribution conforms to the license that it uses, and that third-party libraries licenses are being complied to. A statement of all licenses involved in the distribution and what they apply to. This should included a source code header check for all the files.	QA led with developmenta.	Just before final release.
Create/Update JBoss.org product pages.	Create or update the Jboss.org product pages, documentation, and downloads with new re-	Development.	On release day.

Task	Description	Who Does It?	When Is It Delivered?
	lease.		
Test links on website.	Test all of the links from the new release information, downloads, etc., to make sure that everything is functional.	Development.	On release day.
Internal communication.	Development informs QA that the release is complete, and tags the source repository appropriately.	Development.	On release day.
Announcement.	Certification of the release, and communication is made to internal stakeholders and community forums.	Development and QA with review from product management[b].	On release day.
Official corporate announcement.	Marketing collateral, PR, JBoss ON, etc.	Product management.	Determined by product management.

[a]This could be generated by the build process, and not necessarily have to be done manually.
[b]It is important that release announcements are reviewed by product management to make sure that our messaging is in sync across all communication channels.

Note: The verify license conformance task is a part of producing the final build and distribution, and is not intended to be the time where all license issues are addressed. Issues around whether the licenses are compatible, whether we can use the code within our projects, etc., are a part of due diligence before productizing processes begin. There is a formal license policy in development, that will be linked to here, when it is complete.

2.3. Appendix A.

2.3.1. Key Contacts for the Productizing Process

Contact Name	Role	E-Mail Address	Related Mailing List
Andrig (Andy) Miller	Process owner.	andy.miller@jboss.com [mailto:Andy%20Miller%20%3Candy.miller@jboss.com%3E]	
Ivelin Ivanov	Development management.	ivelin@jboss.com [mailto:Ivelin%20Ivanov%20%3Civelin@jboss.com%3E]	
Shaun Connolly	Product management.	shaun.connolly@jboss.com [mailto:Shaun%20Connolly%20%3Cshaun.connolly@jboss.com%3E]	
Ryan Campbell	Quality Assurance.	ryan.campbell@jboss.com [mailto:Ryan%20Campbell%20%3Cryan.campbell@jboss.com%3E]	qa@jboss.com [mailto:qa@jboss.com]
Norman Richards	Documentation.	norman.richards@jboss.com [mailto:Norman.richards@jboss.com]	
Damon Sicore	JBoss.org website.	damon.sicore@jboss.com [mailto:damon.sicore@jboss.com]	

3

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.

3.1. 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.

3.2. 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.

3.2.1. 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.

3.2.2. 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].

3.3. Issues

3.3.1. 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.

3.3.2. 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.

3.3.3. 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.

3.3.4. 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.

3.4. 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.

3.5. 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.

3.6. 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.

3.7. 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).

4

Build Reference

This reference guide covers how to use the JBossBuild system.

4.1. 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.

4.2. 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.

4.2.1. Component Info Elements Reference

Table 4.1. 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 4.2. 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 4.3. 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>

4.2.2. Component Definition Elements Reference

Table 4.4. 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].

4.3. 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.

4.4. 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.

4.4.1. 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.

4.4.2. 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
```

4.4.2.1. 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.

4.4.3. 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.

4.5. 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>
```

5

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.

5.1. Understanding CVS

CVS (Concurrent Versions System) is an Open Source version control system that is used pervasively throughout the Open Source community. It keeps track of source changes made by groups of developers who are working on the same files and enables developers to stay in sync with each other as each individual chooses.

5.2. 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/].

5.3. Anonymous CVS Access

Note that the anonymous repository is a mirror of the comitter repository that is synched every 5 minutes.

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.com:/cvsroot/jboss login
cvs -z3 -d:pserver:anonymous@anoncvs.forge.jboss.com:/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.com:/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. To obtain more up-to-date information on module naming, refer to JBossAS Modules [http://wiki.jboss.org/wiki/Wiki.jsp?page=JBossASCVSModules] on our wiki.

To checkout the HEAD revision of jboss (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. For more information on Release Tagging Standards, refer to Chapter 14

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
```

You can also browse the repository using the web interface [http://anoncvs.forge.jboss.com/] . If you are stuck behind a firewall without pserver port access, you can even use fisheye to pull the repo using cvsgrab [http://cvsgrab.sourceforge.net/].

```
$ cd /tmp/cvsgrab/
$ cvsgrab -webInterface FishEye1_0 -url \
     http://anoncvs.forge.jboss.com/viewrep/JBoss/jrunit -destDir
```

This will create the JBoss/jrunit directory. Just replace jrunit with the module you want. If you want to check out the entire repo with cvsgrab, just omit the module:

```
$ cd /tmp/cvsgrab/
$ cvsgrab -webInterface FishEye1_0 -url \
     http://anoncvs.forge.jboss.com/viewrep/JBoss -destDir
```

Or, if you want a branch:

```
```

```
$ cd /tmp/cvsgrab/
$ cvsgrab -webInterface FishEye1_0 -url \
     http://anoncvs.forge.jboss.com/viewrep/~br=Branch_4_0/JBoss -destDir
```

Or a tag:

```
$ cd /tmp/cvsgrab/
$ cvsgrab -webInterface FishEye1_0 -url \
     http://anoncvs.forge.jboss.com/viewrep/~br=Branch_4_0,tag=sometag/JBoss -destDir
```

5.4. Committer Access to CVS and JIRA

Write access to the repository is granted only on approval by the Forge Administrator. To request write access send an email to forge-admin@jboss.com asking for committer access.

On approval, you will be given read/write access to the repository and a committer status in JIRA. It is required that you have a committer role in JIRA. The Forge Admin will make sure that you have the proper role and permission status.
To use the committer repository:

```
export CVS_RSH=ssh
export CVSROOT=:ext:username@cvs.forge.jboss.com:/cvsroot/jboss
```

If you are a JBoss employee, your username is the same as your existing cvs.jboss.com username.

If you are not a JBoss Employee, then your username is your existing SourceForge username OR your jboss.com username.

There is NO shell access, only cvs over ssh, similar to SourceForge.

All commiter access is authenticated via SSH. There is no password based committer access. You need to supply an SSH protocol verison 2 public key for access to be granted.
This could be done using the ssh-keygen utility as:

```
ssh-keygen -t dsa -C 'cvs.forge.jboss.com access' -f mykey
```

or

```
ssh-keygen -t rsa -C 'cvs.forge.jboss.com access' -f mykey
```

If you don't know your username or have any trouble, just send an email to forge-admin@jboss.com.

For committer access requests, please include:

- Your full name.

- Your SSH public key.
- A valid email address for us to use.
- Your SourceForge username IF you had committer access before the CVS migration
- Your jboss.org website username.

6

CVS Administration

This chapter describes the JBoss CVS administration policies for managing the CVS repository. Comments or questions regarding these policies should be directed to the JBoss Development forum.

6.1. Creating and Managing Release Branches

The CVS branching and release management procedures are outlined in this section. All development of new features occurs on the main trunk. Releases are done on branches off of the main trunk.

6.1.1. Release Numbering

Releases are tracked using CVS tags that have the following forms:

- Final Binary Releases: JBoss_(major).(even_minor).(patch)
- Beta Binary Releases: Rel__(major).(even_minor).(patch).(build)
- Development Binary Releases(optional): JBoss_(major).(odd_minor).(patch)
- Alpha Development Builds(optional): Rel_(major).(odd_minor).(patch).(build)

1. *A final binary release* is a tested and approved release of the JBoss server. The major and minor version numbers are fixed for a given branch. The minor version number is always even on a release branch. Example final release tags are: JBoss_2_2_0, JBoss_2_2_1, JBoss_2_4_13, JBoss_3_0_0.

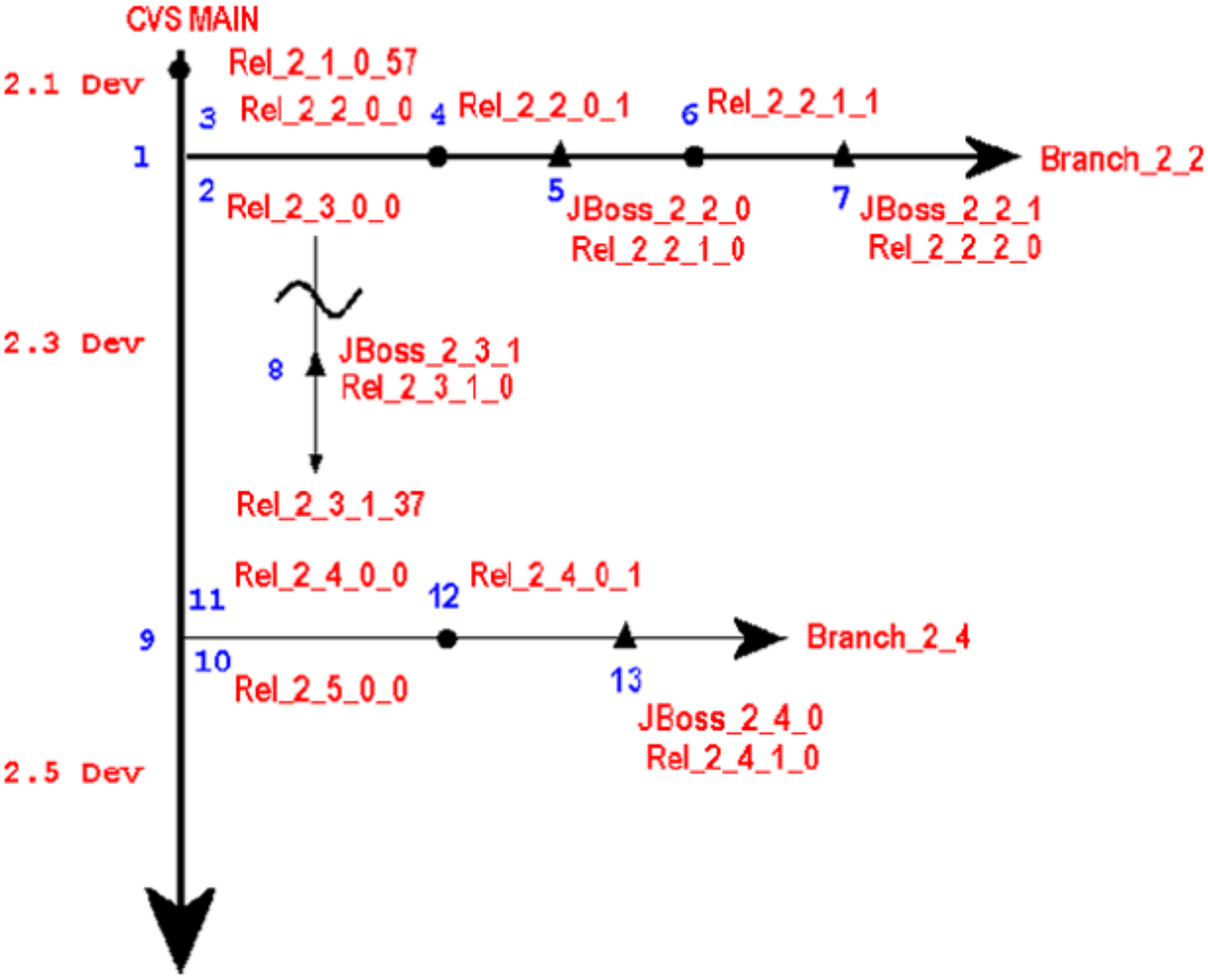
2. *A beta binary release* is a candidate final release that is being made available for testing. The major and minor version numbers are fixed for a given branch. The patch number is one greater than the current final binary. The build number indicates the number of patches that have been incorporated into the candidate release. For example, if the latest final release is JBoss_2_2_0, then next beta binary release patch number will be 1 and build numbers will start at 1. A build number of 0 is used to tag the previous final release code. So, if JBoss_2_2_0 were the latest final release, and three fixes were incorported into the 2.2 branch, there would be beta binary release tags of Rel_2_2_1_0, Rel_2_2_1_1 Rel_2_2_1_2, Rel_2_2_1_3. The idea is that beta binary releases are building to the next final binary release, in this case JBoss_2_2_1.

3. *A development binary release* is an alpha release of the JBoss server. It is a snapshot of the functionallity in the main trunk at some point in time. The major version number is greater than or equal to the latest final binary release. The minor version number is 1 greater than the latest final binary release minor version number. This means that minor versions of development binaries will always be odd. Example development binary releases are: JBoss_2_3_0, JBoss_2_3_1, JBoss_2_5_13, JBoss_3_1_0.

4. *An alpha development build* is a patch beyond a development binary release. The patch number is one greater than the current development binary. The build number indicates the number of patches that have been incorporated into the candidate build. For example, if the latest development build is JBoss_2_3_0, then next alpha build patch number will be 1 and build numbers will start at 1. A build number of 0 is used to tag the previous devlopment build code. So, if JBoss_2_3_0 were the latest development build, and three fixes were incorported into the main trunk, there would be alpha release tags of Rel_2_3_1_0, Rel_2_3_1_1 Rel_2_3_1_2, Rel_2_3_1_3. The idea is that alpha builds are leading to the next development build, in this case JBoss_2_3_1.

6.1.2. Example Release Scenarious

Consider events 1-13 in blue on the following figure:



Prior to event 1, the latest alpha development build is Rel_2_1_0_57. At this point it is decided to create a new binary release.

1. This is the creation of a 2.2 branch. It is labeled with a branch tag of Branch_2_2. This fixes the major version to 2 and the minor version to 2 for all tags on this branch.

2. This is the creation of a Rel_2_3_0_0 alpha release tag on the main trunk. It it is also an alias to the state of the main branch at the time of the 2.2 branch creation.

3. This is the creation of a Rel_2_2_0_0 beta release tag in the branch. It serves as an alias to the state of the main branch at the time the 2.2 branch was created.

4. This is the integration of the first patch/change into the 2.2 branch. After the code is commited the Rel_2_2_0_1 tag is applied.

5. This is the release of the initial 2.2 branch binary. The release is tagged as JBoss_2_2_0 as well as Rel_2_2_1_0 to start the next beta series.

6. This is the integration of the first patch/change after the 2.2.0 binary release. After the code is commited the Rel_2_2_1_1 tag is applied.

7. This is the release of the second 2.2 branch binary. The release is tagged as JBoss_2_2_1 as well as Rel_2_2_2_0 to start the next beta series.

8. This is the release of a development binary. The release is tagged as JBoss_2_3_1 as well as Rel_2_3_1_0 to start the next alpha series. Prior to this there had also been a JBoss_2_3_0 development binary not shown in the diagram.

9. This is the creation of a new binary release branch. After some period of development on the 2.3 portion of the trunk(Rel_2_3_0_0 to Rel_2_3_1_37), it is decided to release a final binary incorporating the main trunk functionality. The new 2.4 branch is labeled with a branch tag of Branch_2_4. This fixes the major version to 2 and the minor version to 4 for all tags on this branch.

10. This is the creation of a Rel_2_5_0_0 alpha release tag on the main trunk. It it is also an alias to the state of the main branch at the time of the 2.4 branch creation.

11. This is the creation of a Rel_2_4_0_0 beta release tag in the branch. It serves as an alias to the state of the main branch at the time the 2.4 branch was created.

12. This is the integration of the first patch/change into the 2.4 branch. After the code is commited the Rel_2_4_0_1 tag is applied.

13. This is the release of the initial 2.4 branch binary. The release is tagged as JBoss_2_4_0 as well as Rel_2_4_1_0 to start the next beta series.

6.2. Creating a New Binary Release Branch

1. Perform a clean check out of the jboss main branch without any tags to select the latest code:

```
cvs co jboss-head
```

2. Label the main branch with the next initial alpha development build tag: Rel_(major)_(odd_minor)_0_0. For the case of a 2.2 release case this would mean that main development would be for a 2.3 cycle and so main should be tagged with Rel_2_3_0_0 as follows from within the working directory created in step 1:

```
cvs tag Rel_2_3_0_0
```

3. Create the new branch giving it a branch tag of Branch_(major)_(even_minor). For example, to create a 2.2 branch, perform the following within the working directory created by the previous check out:

```
cvs tag -b Branch_2_2
```

4. Create a working directory for the new branch by checking it out using the Branch_2_2 tag:

```
cvs co -r Branch_2_2 jboss
```

5. Label the branch working directory with the initial beta release tag of Rel_(major)_(even_minor)_0_0. For the Branch_2_2 case this would be done by executing the following in the working directory created by the previous check out:

```
cvs tag Rel_2_2_0_0
```

6. Branch all non-jboss modules that contribute jars to the jboss module. Create a branch for each cvs module for which there is one or more jars included in the jboss module. This allows patches to be made to these modules and to be tagged with the JBoss_X_Y_Z final release tag so that all source can be obtained for the final release.

6.3. Checking Code into the MAIN Trunk

New features and bug fixes on unreleased code should go into the main trunk which is the latest development branch. The steps for doing this are:

1. Checkout the target module in which the changes are to be made. For example to commit changes to the `jboss` module do:

```
cvs co jboss-head
```

2. Make your chages to the source in the jboss working directory created by the previous check out.

3. Commit your changes. Do this by executing the following command in the directory you made the changes in, or any common parent directory:

```
cvs commit -m "commit-comment"
```

   You don't have to specify the commit msg on the commit command line. If you don't you will be prompted for the commit msg. Note that this will apply the same commit msg to all files you have changed. If you want specific commit msgs for each file then you can perform a seperate commit on each file.

4. *Optional* Tag the code with the next alpha build tag. For example, to tag the jboss source tree with a

Rel_2_3_1_3 tag, do:

```
cvs tag Rel_2_3_1_3
```

from within the jboss working directory.

6.4. Checking in a Patch on a Release Branch

When you have changes that need to go into the codebase of a release branch, you need to check out that branch and make the changes. So for example, if you need to add a patch the the 2.2 branch of the example CVS structure above, you need to first check out the 2.2 branch using the Branch_2_2 tag.

1. Checkout the module using the branch tag you want to work on. To checkout the 2.2 branch of the jboss module do:

```
cvs co -r Branch_2_2 jboss
```

   This will create a jboss working directory with a sticky tag that associates the source code with the 2.2 branch. If you look at the jboss/src/main/org/jboss/Main.java file in the jboss working directory that results from the previous command using the `cvs status` command you will see something like:

```
bash-2.04$ cd jboss/src/main/org/jboss/
bash-2.04$ cvs status Main.java
===================================================================
File: no file Main.java         Status: Needs Checkout

   Working revision:    1.30.2.6
   Repository revision: 1.30.2.6        /cvsroot/jboss/jboss/src/main/org/jboss/Main.java,v
   Sticky Tag:          Branch_2_2 (branch: 1.30.2)
   Sticky Date:         (none)
   Sticky Options:      (none)
```

   This shows that the "Sticky Tag:" is set to the Branch_2_2 tag as we requested.

2. Make your chages to the source in the jboss working directory created by the previous check out.

3. *Required* Run the jbosstest unit test suite. If there are any errors do NOT commit your change. Repeated failures to validate a change made to a branch will result in loss of CVS write priviledges.

4. Commit your changes. Do this by executing the following command in the directory you made the changes in, or any common parent directory:

```
cvs commit -m "commit-comment"
```

   As already noted, you don't have to specify the commit msg on the commit command line. If you don't you will be prompted for the commit msg. Note that this will apply the same commit msg to all files you have changed. If you want specific commit msgs for each file then you can perform a seperate commit on each file.

5. *Required* Tag the branch with the next beta binary release tag by incrementing the build number of the latest

tag. To determine what build number to use, look at all of the tags for a file using the `cvs status` command with the -v option. For example, looking at jboss/src/main/org/jboss/Main.java again:

```
bash-2.04$ cvs status -v Main.java
===================================================================
File: no file Main.java         Status: Needs Checkout

   Working revision:    1.30.2.6
   Repository revision: 1.30.2.6        /cvsroot/jboss/jboss/src/main/org/jboss/Main.java,v
   Sticky Tag:          Branch_2_2 (branch: 1.30.2)
   Sticky Date:         (none)
   Sticky Options:      (none)

   Existing Tags:
        Rel_2_3_1_0                     (revision: 1.34)
        Rel_2_2_2_0                     (revision: 1.30.2.6)
        JBoss_2_2_2                     (revision: 1.30.2.6)
        JBoss_2_2_1                     (revision: 1.30.2.3)
        Rel_2_2_1_0                     (revision: 1.30.2.3)
```

The Rel_2_2_2_0 tag is the latest tag on the 2.2 branch and indicates that no patches have been made since the JBoss_2_2_2 release. So to tag the changes you have made you need to use Rel_2_2_2_1. Do this using:

```
cvs tag Rel_2_2_2_1
```

from the top of the jboss working directory.

6. *Required* Merge the changes to the main trunk if they are missing. You need to validate that the changes you have made to the release branch are not already in the main trunk and merge the changes if they are.

7. *Required, if merge was done* Check out the latest trunk code:

```
cvs co jboss
```

8. *Required, if merge was done* Tag the main trunk with the next alpha build tag. Assuming the this is Rel_2_3_1_5, you would do:

```
cvs tag Rel_2_3_1_5
```

from within the jboss working directory you just checked out.

6.5. Checking in a Patch on a Non-JBoss CVS Module Release Branch

When you have changes that need to go into one of the modules other than the `jboss` cvs module for integration as a jar in a jboss release branch, perform the following steps. The example below describes how to make a change in the jbosscx module for incorporation into the jboss 2.4 release branch.

1. Checkout the module using the branch tag you want to work on (if the branch has not been created do so). To

checkout the 2.4 branch of the jbosscx module do:

```
cvs co -r Branch_2_4 jbosscx
```

2. Make your chages to the source in the jbosscx working directory created by the previous check out.

3. Commit your changes. Do this by executing the following command in the directory you made the changes in, or any common parent directory:

```
cvs commit -m "commit-comment"
```

4. *Required* Tag the branch with the next beta binary release tag on the jboss module release branch, not the jbosscx. The non-jboss modules are not labeled independent of the jboss module. This allows one to see what changes from the modules were merged into jboss. So, if the latest beta binary release tag in the jboss module is Rel_2_4_0_0, the jbosscx module would be tagged with Rel_2_4_0_1. Do this from within the root jbosscx working directory:

```
cvs tag Rel_2_4_0_1
```

   The Rel_2_2_2_0 tag is the latest tag on the 2.2 branch and indicates that no patches have been made since the JBoss_2_2_2 release. So to tag the changes you have made you need to use Rel_2_2_2_1.

```
cvs tag Rel_2_2_2_1
```

5. Perform the build of the module jars that are to be incorporated into the jboss module.

6. Copy the module jars into the approriate `jboss/src` subdirectory locations.

7. *Required* Run the jbosstest unit test suite. If there are any errors do NOT commit your change. Repeated failures to validate a change made to a branch will result in loss of CVS write priviledges.

8. Commit the jar changes in the jboss module by running the following from within the jboss/src directory:

```
cvs commit -m "commit-comment"
```

9. *Required* Tag the jboss module with the same tag used in step 4. From within the jboss root working directory tag the release:

```
cvs tag Rel_2_4_0_1
```

7

SVN Access for JBoss Sources

Source code for specific JBoss projects are located in the JBoss Subversion repository. Please see the project homepage to determine the source location.

7.1. Understanding SVN

Subversion is an Open Source version control system that is very similiar in functionality to CVS. It keeps track of source changes made by groups of developers who are working on the same files and enables developers to stay in sync with each other as each individual chooses.

7.2. Obtaining an SVN Client

The command line version of the Subversion program is freely available for nearly every platform. You can select the appropriate package here: Subversion downloads [http://subversion.tigris.org/project_packages.html].

Tortoise SVN is a popular GUI based client and can be found here: Tortoise SVN downloads [http://tortoisesvn.sourceforge.net/downloads]

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

For complete documentation on Subversion, check out The Subversion RedBook [http://svnbook.red-bean.com/].

7.3. Anonymous CVS Access

Note that the anonymous repository is a mirror of the comitter repository that is synched every 5 minutes.

All JBoss projects' Subversion repositories can be accessed through anonymously with the following instruction set. The project you want to check out must be specified as the `project`. You will also provide the path which contains either the correct branch, tag, or trunk.

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

```
svn co https://svn.jboss.org/repos/project/path
```

To checkout the HEAD revision of jboss (latest code on the main branch), you would use the project`jbossas/`

`trunk` as the project 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. For more information on Release Tagging Standards, refer to Chapter 14

Some checkout examples are:

```
svn co http://anonsvn.jboss.org/repos/jbossas/tags/JBoss_3_2_6
svn co http://anonsvn.jboss.org/repos/jbossas/trunk # Checkout the curent HEAD branch code
```

You can also browse the repository using the web interface [http://anonsvn.jboss.org/repos]

7.4. Committer Access to SVN and JIRA

Write access to the repository is granted only on approval by the Forge Administrator. To request write access send an email to forge-admin@jboss.com asking for committer access.

On approval, you will be given read/write access to the repository and a committer status in JIRA. It is required that you have a committer role in JIRA. The Forge Admin will make sure that you have the proper role and permission status.
To use the committer repository:

```
svn co https://svn.jboss.org/repos/project
```

If you are a JBoss employee, your username is the same as your existing cvs.jboss.com username.

If you are not a JBoss Employee, then your username is your existing SourceForge username OR your jboss.com username.

If you don't know your username or have any trouble, just send an email to forge-admin@jboss.com.

For committer access requests, please include:

- Your full name.
- A valid email address for us to use.
- Your jboss.org website username.

8

SVN Administration

This chapter describes the JBoss SVN administration policies for managing the SVN repository. Comments or questions regarding these policies should be directed to the JBoss Development forum.

8.1. Creating and Managing Release Branches

The CVS branching and release management procedures are outlined in this section. All development of new features occurs on the main trunk. Releases are done on branches off of the main trunk.

8.1.1. Release Numbering

Releases are tracked using SVN tags that have the following forms:

- Final Binary Releases: JBoss_(major).(even_minor).(patch)
- Beta Binary Releases: Rel__(major).(even_minor).(patch).(build)
- Development Binary Releases(optional): JBoss_(major).(odd_minor).(patch)
- Alpha Development Builds(optional): Rel_(major).(odd_minor).(patch).(build)

1. *A final binary release* is a tested and approved release of the JBoss server. The major and minor version numbers are fixed for a given branch. The minor version number is always even on a release branch. Example final release tags are: JBoss_2_2_0, JBoss_2_2_1, JBoss_2_4_13, JBoss_3_0_0.

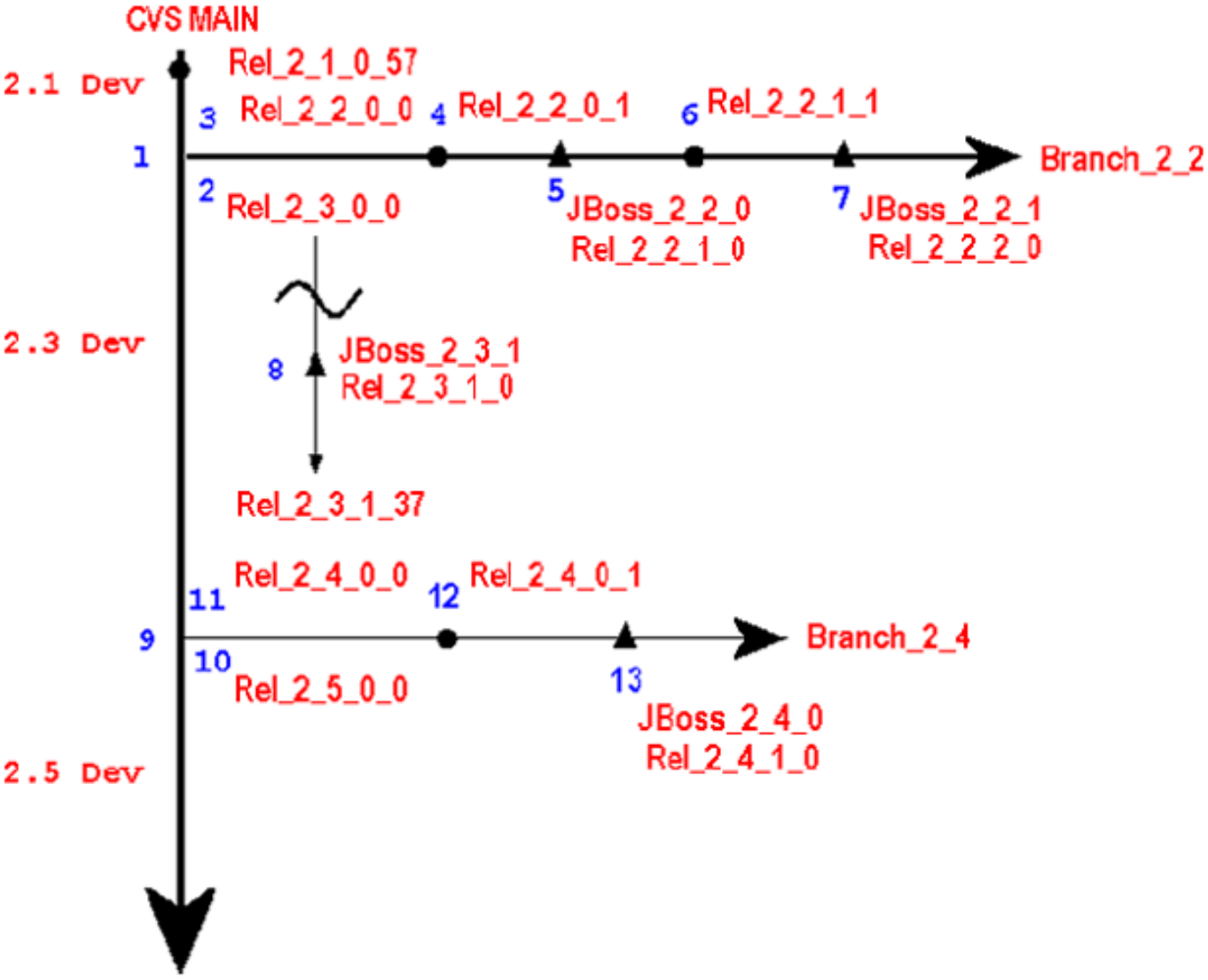
2. *A beta binary release* is a candidate final release that is being made available for testing. The major and minor version numbers are fixed for a given branch. The patch number is one greater than the current final binary. The build number indicates the number of patches that have been incorporated into the candidate release. For example, if the latest final release is JBoss_2_2_0, then next beta binary release patch number will be 1 and build numbers will start at 1. A build number of 0 is used to tag the previous final release code. So, if JBoss_2_2_0 were the latest final release, and three fixes were incorported into the 2.2 branch, there would be beta binary release tags of Rel_2_2_1_0, Rel_2_2_1_1 Rel_2_2_1_2, Rel_2_2_1_3. The idea is that beta binary releases are building to the next final binary release, in this case JBoss_2_2_1.

3. *A development binary release* is an alpha release of the JBoss server. It is a snapshot of the functionallity in the main trunk at some point in time. The major version number is greater than or equal to the latest final binary release. The minor version number is 1 greater than the latest final binary release minor version number. This means that minor versions of development binaries will always be odd. Example development binary releases are: JBoss_2_3_0, JBoss_2_3_1, JBoss_2_5_13, JBoss_3_1_0.

4. *An alpha development build* is a patch beyond a development binary release. The patch number is one greater than the current development binary. The build number indicates the number of patches that have been incorporated into the candidate build. For example, if the latest development build is JBoss_2_3_0, then next alpha build patch number will be 1 and build numbers will start at 1. A build number of 0 is used to tag the previous devlopment build code. So, if JBoss_2_3_0 were the latest development build, and three fixes were incorported into the main trunk, there would be alpha release tags of Rel_2_3_1_0, Rel_2_3_1_1 Rel_2_3_1_2, Rel_2_3_1_3. The idea is that alpha builds are leading to the next development build, in this case JBoss_2_3_1.

8.1.2. Example Release Scenarious

Consider events 1-13 in blue on the following figure:



Prior to event 1, the latest alpha development build is Rel_2_1_0_57. At this point it is decided to create a new binary release.

1. This is the creation of a 2.2 branch. It is labeled with a branch tag of Branch_2_2. This fixes the major version to 2 and the minor version to 2 for all tags on this branch.

2. This is the creation of a Rel_2_3_0_0 alpha release tag on the main trunk. It it is also an alias to the state of the main branch at the time of the 2.2 branch creation.

3. This is the creation of a Rel_2_2_0_0 beta release tag in the branch. It serves as an alias to the state of the main branch at the time the 2.2 branch was created.

4. This is the integration of the first patch/change into the 2.2 branch. After the code is commited the Rel_2_2_0_1 tag is applied.

5. This is the release of the initial 2.2 branch binary. The release is tagged as JBoss_2_2_0 as well as Rel_2_2_1_0 to start the next beta series.

6. This is the integration of the first patch/change after the 2.2.0 binary release. After the code is commited the Rel_2_2_1_1 tag is applied.

7. This is the release of the second 2.2 branch binary. The release is tagged as JBoss_2_2_1 as well as Rel_2_2_2_0 to start the next beta series.

8. This is the release of a development binary. The release is tagged as JBoss_2_3_1 as well as Rel_2_3_1_0 to start the next alpha series. Prior to this there had also been a JBoss_2_3_0 development binary not shown in the diagram.

9. This is the creation of a new binary release branch. After some period of development on the 2.3 portion of the trunk(Rel_2_3_0_0 to Rel_2_3_1_37), it is decided to release a final binary incorporating the main trunk functionality. The new 2.4 branch is labeled with a branch tag of Branch_2_4. This fixes the major version to 2 and the minor version to 4 for all tags on this branch.

10. This is the creation of a Rel_2_5_0_0 alpha release tag on the main trunk. It it is also an alias to the state of the main branch at the time of the 2.4 branch creation.

11. This is the creation of a Rel_2_4_0_0 beta release tag in the branch. It serves as an alias to the state of the main branch at the time the 2.4 branch was created.

12. This is the integration of the first patch/change into the 2.4 branch. After the code is commited the Rel_2_4_0_1 tag is applied.

13. This is the release of the initial 2.4 branch binary. The release is tagged as JBoss_2_4_0 as well as Rel_2_4_1_0 to start the next beta series.

8.2. Creating a New Binary Release Branch

1. Perform a clean check out of the jboss main branch without any tags to select the latest code:

```
svn co https://svn.jboss.org/repos/jbossas/trunk
```

2. To create a "tag" you simply execute a copy command. Tag the main branch with the next initial alpha development build tag: Rel_(major)_(odd_minor)_0_0. For the case of a 2.2 release case this would mean that main development would be for a 2.3 cycle and so main should be tagged with Rel_2_3_0_0 as follows from within

the working directory created in step 1:

```
svn copy https://svn.jboss.org/repos/jbossas/trunk https://svn.jboss.org/repos/jbossas/tags/Rel_2_3_0_0 "Creating a tag"
```

3. Create the new branch giving it a branch tag of Branch_(major)_(even_minor). For example, to create a 2.2 branch, perform the following within the working directory created by the previous check out:

```
svn copy https://svn.jboss.org/repos/jbossas/trunk https://svn.jboss.org/repos/jbossas/branches/Branch_2_2 "Creating a branch"
```

4. Create a working directory for the new branch by checking it out using the Branch_2_2 tag:

```
svn co https://svn.jboss.org/repos/jbossas/branches/Branch_2_2
```

5. Label the branch working directory with the initial beta release tag of Rel_(major)_(even_minor)_0_0. For the Branch_2_2 case this would be done by executing the following in the working directory created by the previous check out:

```
svn copy https://svn.jboss.org/repos/jbossas/branches/Branch_2_2 https://svn.jboss.org/repos/jbossas/tags/Rel_2_2_0_0 "Creating a b
```

8.3. Checking Code into the MAIN Trunk

New features and bug fixes on unreleased code should go into the main trunk which is the latest development branch. The steps for doing this are:

1. Checkout the target module in which the changes are to be made. For example to commit changes to the `jboss` module do:

```
svn co https://svn.jboss.org/repos/jbossas/trunk
```

2. Make your chages to the source in the jboss working directory created by the previous check out.

3. Commit your changes. Do this by executing the following command in the directory you made the changes in, or any common parent directory:

```
svn commit -m "commit-comment"
```

   Note that this will apply the same commit msg to all files you have changed. If you want specific commit msgs for each file then you can perform a seperate commit on each file.

8.4. Creating a service patch

The procedure defined below will take a developer through the process of creating a branch, making the necessary changes, and merging those changes into the main branch.

1

```
svn copy http://svn.jboss.org/repos/test/tags/JBoss_4_0_3_SP1/ http://svn.jboss.org/repos/test/branches/JBoss_4_0_3_SP1_JBAS-1234 -
```

2. Checkout the newly created branch

```
svn co http://svn.jboss.org/repos/test/branches/JBoss_4_0_3_SP1_JBAS-1234 jbas-1234_local_dir
```

3. Make your changes, perform testing, and commit them

```
svn commit -m "changes required for patch"
```

4. At this point you may wish to port this patch to the current code line. To do this we will use the svn merge command. The svn merge command requires 3 pieces of information.

   a. An initial repository tree

   b. A final repository tree

   c. A working copy to apply the changes to

   Essentially, you are finding the change set between 1 and 2 and applying them to 3. In our case 1 would be the tagged JBoss-4.0.3.SP1 and 2 would be the JBoss-4.0.3.SP1.PATCH branch that you created. 3 would be the current 4.0 branch (which will you need to check out).

Backporting procedure

1. checkout a working copy of the 4.0 branch

```
svn co http://svn.jboss.org/repos/test/branches/Branch_4_0 jboss-4.0
```

2. apply the changeset between the 4.0.3.SP1 tagged release and your patched branch to your working copy

```
svn merge http://svn.jboss.org/repos/test/tags/JBoss_4_0_3_SP1 http://svn.jboss.org/repos/test/branches/JBoss_4_0_3_SP1-JBAS-1234 j
```

3. The differences are now applied to your working copy. Ensure that no conflicts exist and then commit the work to current jboss-4.0 branch

```
svn commit
```

9

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
 * Copyright 2005, JBoss Inc., and individual contributors as indicated
 * by the @authors tag. See the copyright.txt in the distribution for a
 * full listing of individual contributors.
 *
 * This is free software; you can redistribute it and/or modify it
 * under the terms of the GNU Lesser General Public License as
 * published by the Free Software Foundation; either version 2.1 of
 * the License, or (at your option) any later version.
 *
 * This software is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
 * Lesser General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public
 * License along with this software; if not, write to the Free
 * Software Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA
 * 02110-1301 USA, or see the FSF site: http://www.fsf.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.

9.1. Templates

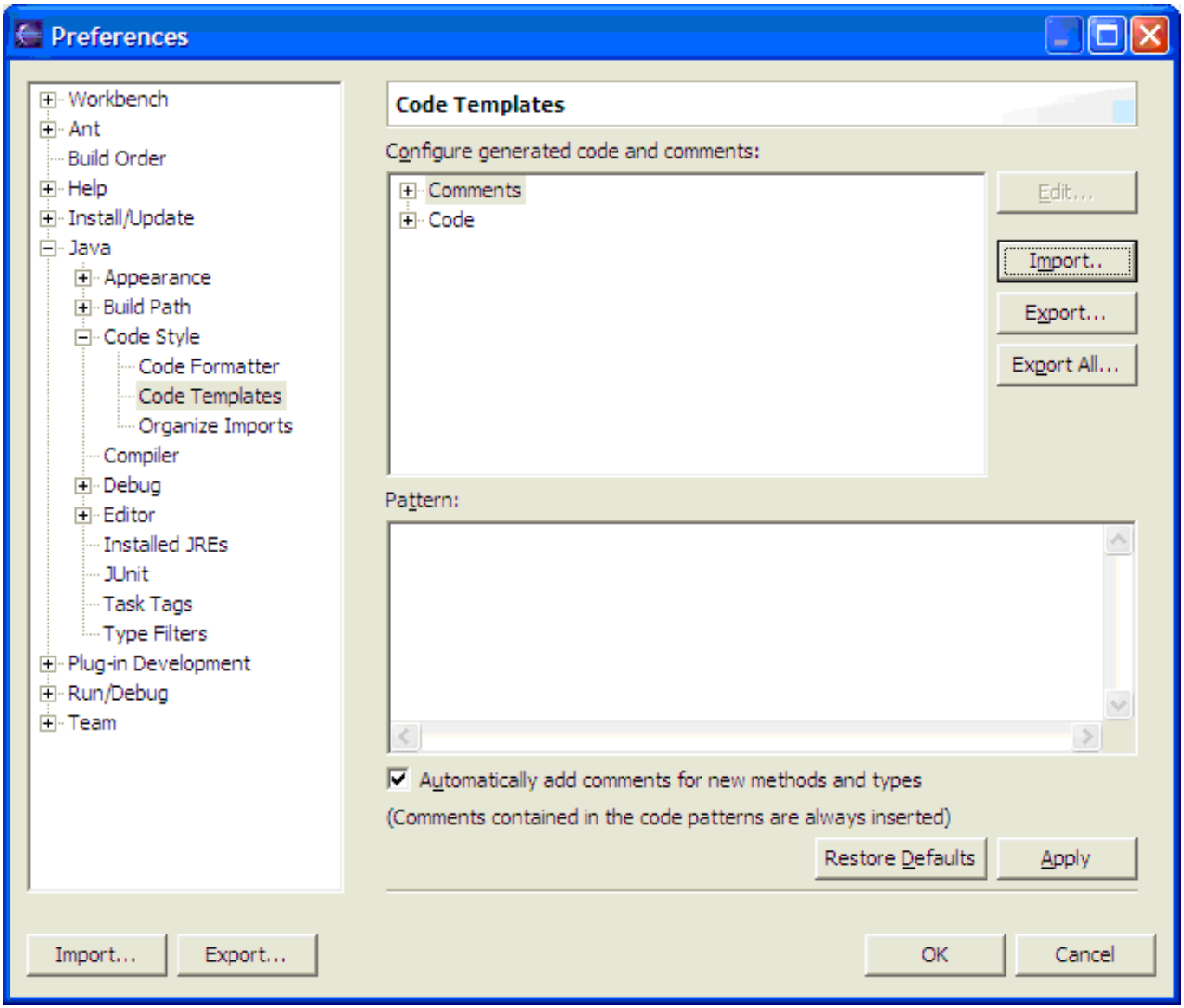
Template files for the Eclipse IDE can be found here: JBoss Eclipse Format [http://jira.jboss.com/jira/secure/attachment/12310313/jboss-format.xml/]. JBoss Eclipse Template [http://jira.jboss.com/jira/secure/attachment/12310312/jboss-template.xml/].

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

9.1.1. 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.

9.2. 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.

9.3. 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 9.1. A class that conforms to JBoss coding guidelines

```
 /*
  * JBoss, Home of Professional Open Source
  * Copyright 2005, JBoss Inc., and individual contributors as indicated
  * by the @authors tag. See the copyright.txt in the distribution for a
  * full listing of individual contributors.
  *
  * This is free software; you can redistribute it and/or modify it
  * under the terms of the GNU Lesser General Public License as
  * published by the Free Software Foundation; either version 2.1 of
  * the License, or (at your option) any later version.
  *
  * This software is distributed in the hope that it will be useful,
  * but WITHOUT ANY WARRANTY; without even the implied warranty of
  * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
  * Lesser General Public License for more details.
  *
  * You should have received a copy of the GNU Lesser General Public
  * License along with this software; if not, write to the Free
  * Software Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA
  * 02110-1301 USA, or see the FSF site: http://www.fsf.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: 1.4 $</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 9.2. An interface that conforms to JBoss coding guidelines

```
 /*
  * JBoss, Home of Professional Open Source
  * Copyright 2005, JBoss Inc., and individual contributors as indicated
  * by the @authors tag. See the copyright.txt in the distribution for a
  * full listing of individual contributors.
  *
  * This is free software; you can redistribute it and/or modify it
  * under the terms of the GNU Lesser General Public License as
  * published by the Free Software Foundation; either version 2.1 of
  * the License, or (at your option) any later version.
  *
  * This software is distributed in the hope that it will be useful,
  * but WITHOUT ANY WARRANTY; without even the implied warranty of
  * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
  * Lesser General Public License for more details.
  *
  * You should have received a copy of the GNU Lesser General Public
  * License along with this software; if not, write to the Free
  * Software Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA
  * 02110-1301 USA, or see the FSF site: http://www.fsf.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: 1.4 $</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;
```

```
}
```

10

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.

10.1. 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.

10.2. 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.

10.3. 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.

10.3.1. Separating Application Logs

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

Example 10.1. 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>
```

10.3.2. 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 specifed 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 10.2. 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>
```

10.3.3. 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 10.3. 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 10.4. 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 10.5. 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 10.6. `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>
```

10.3.4. 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 10.1. 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

10.3.5. 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 10.7. 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.

10.3.6. 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.

10.3.7. 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 10.8. 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.

10.3.8. 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 10.9. A `RepositorySelector`

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

```
 /*
  * JBoss, Home of Professional Open Source
  * Copyright 2005, JBoss Inc., and individual contributors as indicated
  * by the @authors tag. See the copyright.txt in the distribution for a
  * full listing of individual contributors.
  *
  * This is free software; you can redistribute it and/or modify it
  * under the terms of the GNU Lesser General Public License as
  * published by the Free Software Foundation; either version 2.1 of
  * the License, or (at your option) any later version.
  *
  * This software is distributed in the hope that it will be useful,
  * but WITHOUT ANY WARRANTY; without even the implied warranty of
  * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
  * Lesser General Public License for more details.
  *
  * You should have received a copy of the GNU Lesser General Public
  * License along with this software; if not, write to the Free
  * Software Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA
  * 02110-1301 USA, or see the FSF site: http://www.fsf.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;
      }
   }
}
```

10.4. 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
```

11

Logging
Logging

One can say that using a debugger may help to verify the execution of an application. However, in addition to the fact that a debugger decreases performance of an application, it is difficult to use it in a distributed computing environment.

This most basic form of logging involves developers manually inserting code into their applications to display small (or large) pieces of internal state information to help understand what's going on. It's a useful technique that every developer has used at least once. The problem is that it doesn't scale. Using print statements for a small program is fine, but for a large, commercial-grade piece of software there is far too much labor involved in manually adding and removing logging statements.

C programmers know, of course, that the way to conditionally add and remove code is via the C preprocessor and the #ifdef directive. Unfortunately, Java doesn't have a preprocessor. How can we make logging scale to a useful level in Java?

A simple way to provide logging in your program is to use the Java compiler's ability to evaluate boolean expressions at compile time, provided that all the arguments are known. For example, in this code, the println statements will not be executed if DEBUG not set to true.

```
class foo
{
    public bar()
    {
      if(DEBUG)
      {
        System.out.println("Debugging enabled.");
      }
    }
  }
```

A much better way, and the way that most logging is done in environments where the logged output is important, is to use a logging class.

A logging class collects all the messages in one central place and not only records them, but can also sort and filter them so that you don't have to see every message being generated. A logging class provides more information than just the message. It can automatically add information such as the time the event occurred, the thread that generated the message, and a stack trace of where the message was generated.

Some logging classes will write their output directly to the screen or a file. More advanced logging systems may instead open a socket to allow the log messages to be sent to a separate process, which is in turn responsible for

passing those messages to the user or storing them. The advantage with this system is that it allows for messages from multiple sources to be aggregated in a single location and it allows for monitoring remote systems.

The format of the log being generated should be customisable. This could start from just allowing setting the Log "level" - which means that each log message is assigned a severity level and only messages of greater importance than the log level are logged - to allowing more flexible log file formatting by using some sort LogFormatter objects that do transformations on the logging information.

The logging service should be able to route logging information to different locations based on the type of the information. Examples might be printing certain messages to the console, writing to a flat file, to a number of different flat files, to a database and so on. Examples of different types information could be for example errors, access information etc.

An appropriate logging library should provide these features:

1. Control over which logging statements are enabled or disabled,
2. Define importance or severity for logging statement via a set of levels
3. Manage output destinations,
4. Manage output format.
5. Manage internationalization (i18n)
6. Configuration.

11.1. Relevant Logging Framework

According to the features (described above) a logging framework should provide, we have considering the most common logging service is use.

11.1.1. Overview of log4j

11.1.1.1. Categories, Appenders, and Layout

Log4j has three main components:

1. Categories
2. Appenders
3. Layouts

11.1.1.2. Category Hierarchy

The `org.log4j.Category` class figures at the core of the package. Categories are named entities. In a naming scheme familiar to Java developers, a category is said to be a parent of another category if its name, followed by a

dot, is a prefix of the child category name. For example, the category named `com.foo` is a parent of the category named com.foo.Bar. Similarly, java is a parent of `java.util` and an ancestor of `java.util.Vector`.

The root category, residing at the top of the category hierarchy, is exceptional in two ways:

1. It always exists
2. It cannot be retrieved by name

In the `Category` class, invoking the static `getRoot()` method retrieves the root category. The static `getInstance()` method instantiates all other categories. `getInstance()` takes the name of the desired category as a parameter. Some of the basic methods in the `Category` class are listed below:

```
package org.log4j;

public Category class {
  // Creation and retrieval methods:
  public static Category getRoot();
  public static Category getInstance(String name);

  // printing methods:
  public void debug(String message);
  public void info(String message);
  public void warn(String message);
  public void error(String message);

  // generic printing method:
  public void log(Priority p, String message);
}
```

Categories may be assigned priorities from the set defined by the `org.log4j.Priority` class. Five priorities are defined: FATAL, ERROR, WARN, INFO and DEBUG, listed in decreasing order of priority. New priorities may be defined by subclassing the `Priority` class.

1. FATAL: The FATAL priority designates very severe error events that will presumably lead the application to abort.
2. ERROR: The ERROR priority designates error events that might still allow the application to continue running.
3. WARN: The WARN priority designates potentially harmful situations.
4. INFO: The INFO priority designates informational messages that highlight the progress of the application.
5. DEBUG: The DEBUG priority designates fine-grained informational events that are most useful to debug an application.

To make logging requests, invoke one of the printing methods of a category instance. Those printing methods are: `fatal(), error(), warn(), info(), debug(), log()`.

By definition, the printing method determines the priority of a logging request. For example, if c is a category in-

stance, then the statement `c.info("..")` is a logging request of priority INFO.

A logging request is said to be enabled if its priority is higher than or equal to the priority of its category. Otherwise, the request is said to be disabled. A category without an assigned priority will inherit one from the hierarchy.

11.1.1.3. Appenders and layouts

Log4j also allows logging requests to print to multiple output destinations called appenders in log4j speak. Currently, appenders exist for the console, files, GUI components, remote socket servers, NT Event Loggers, and remote UNIX Syslog daemons.

A category may refer to multiple appenders. Each enabled logging request for a given category will be forwarded to all the appenders in that category as well as the appenders higher in the hierarchy. In other words, appenders are inherited additively from the category hierarchy. For example, if you add a console appender to the root category, all enabled logging requests will at least print on the console. If, in addition, a file appender is added to a category, say C, then enabled logging requests for C and C's children will print on a file and on the console.

More often than not, users want to customize not only the output destination but also the output format, a feat accomplished by associating a layout with an appender. The layout formats the logging request according to the user's wishes, whereas an appender takes care of sending the formatted output to its destination.

For example, the PatternLayout with the conversion pattern `%r [%t]%-5p %c - %m%n` will output something like:

```
    176 [main] INFO org.foo.Bar #Hello World.
```

In the output above:

1. The first field equals the number of milliseconds elapsed since the start of the program
2. The second field indicates the thread making the log request
3. The third field represents the priority of the log statement
4. The fourth field equals the name of the category associated with the log request

The text after the - indicates the statement's message.

11.1.1.4. Configuration

The log4j environment can be fully configured programmatically. However, it is far more flexible to configure log4j by using configuration files. Currently, configuration files can be written in XML or in Java properties (key=value) format.

The following figure summarizes the different components when using log4j. Applications make logging calls on `Category` objects. The `Category` forwards to `Appender` logging requests for publication. `Appender` are registered with a `Category` with the `addAppender` method on the `Category` class. Invoking the `addAppender` method is made either by the `Application` or by `Configurator` objects. Log4j provides `Configurators` such as `BasicConfigurator`, which registers to the category the `ConsoleAppender` responsible to send logging requests to the console, or the `PropertyConfigurator`, which registers `Appender` objects based on `Appender` classes defined in a configuration

file. Both `Category` and `Appender` may use logging `Priority` and (optionally) `Filters` to decide if they are interested in a particular logging request. An `Appender` can use a `Layout` to localize and format the message before publishing it to the output world.

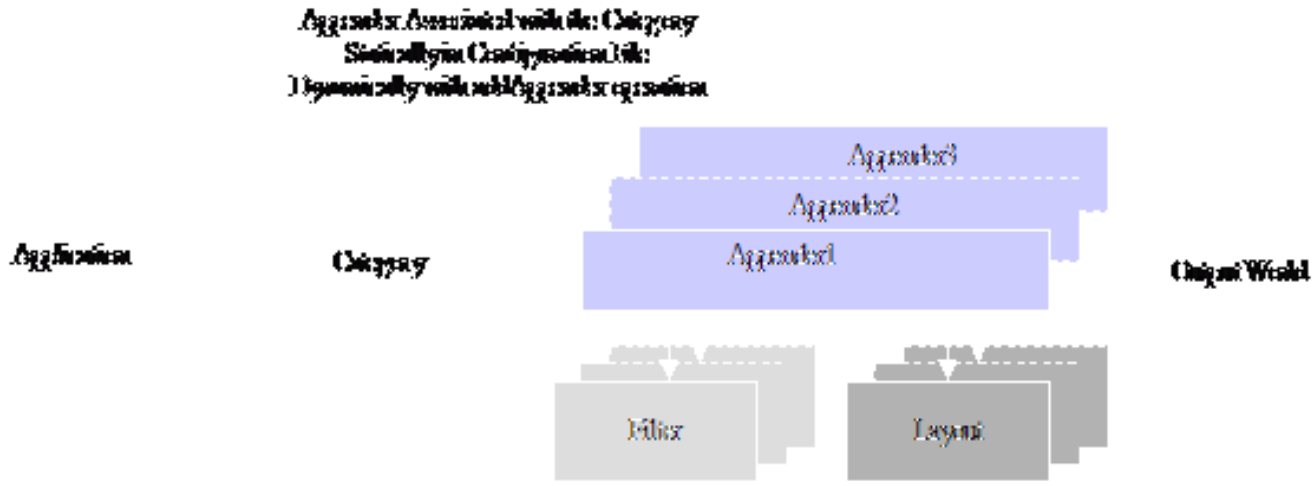


Figure 11.1. Example of log interactions.

11.1.2. HP Logging Mechanism

The HP Logging Mechanism consists of a log handler, zero or more log writers, and one or more log channels, as illustrated in Figure below.

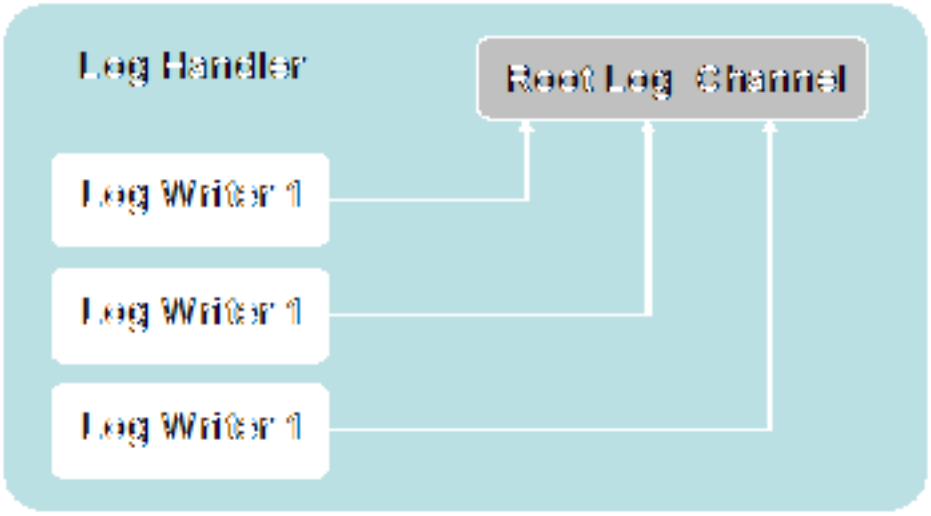


Figure 11.2. The LogHandler.

11.1.2.1. Log Handler

The log handler is implemented as a singleton Java Bean. It is accessible from the `com.hp.mw.common.util.LogHandlerFactory` which returns the single instance of `com.hp.mw.common.util.LogHandler`.

The following code illustrates how to obtain the LogHandler:

```
LogHandler handler;

handler = LogHandlerFactory.getHandler();
```

11.1.2.2. Log Channel

Log channels are virtual destinations; they receive messages and pass them to the log writers that are registered to receive them. They are not aware of the message formatting that might occur and are not aware of the logging tools that are used to view or store the messages. Log writers are registered for channels. When a log channel receives a message, and if that channel has a registered log writer(s), the message is passed along to that writer.

A client may obtain a channel with a specific name as follows.

```
LogChannel channel;

channel = LogChannelFactory.getChannel("myapplication" );
```

11.1.2.3. Log Writers

In order to abstract the destination of a log message (e.g., console, file, database), the Logging Mechanism relies on log writers. Log writers are defined by the `com.hp.mw.common.util.logging.LogWriter` interface and are given messages by the channel(s) they service. They are responsible for formatting messages and outputting to the actual destination.

11.1.2.4. Log Formatters

A log formatter is responsible for formatting a log message into a Java `String`. Since many log writers do not require the String representation, log formatters are not required for every log writer. As a result, the `com.hp.mw.common.util.logging.LogMessageFormat` interface would be used for formatting messages into Strings when applicable and necessary.

11.1.2.5. Log Levels and Thresholds

All log channels are created, initially, with a default log threshold. The threshold is the minimum severity of a log message that should be processed for that log channel. The log levels defined by the HP logging mechanisms are as follows:

Log Level Description

1. LOG_LEVEL_NONE This log level should be used to turn off all messages to a channel.
2. LOG_LEVEL_FLOW Flow messages indicate program flow and can be extremely frequent.

3. LOG_LEVEL_DEBUG Debug messages are fairly low-level messages that provide the developer(s) with information about events occurring within the application

4. LOG_LEVEL_INFO Informational messages are of higher severity than debug and should provide information that any user could understand, as opposed to debug messages, which provide code-specific information.

5. LOG_LEVEL_WARNING Warning messages are typically used to report an unusual or unexpected occurrence from which recovery is possible (e.g., a missing or incorrect configuration value that has a reasonable default).

6. LOG_LEVEL_ERROR Error messages are used to report an unusual or unexpected occurrence from which recovery is not possible. This does not indicate that the entire application or framework is incapable of continuing, but that the component involved might be defunct or the operation it was asked to perform is aborted.

7. LOG_LEVEL_CRITICAL Critical messages are typically used to report a very unusual or unexpected occurrence. For example, a component that was functioning correctly but suddenly experiences an unrecoverable error that prevents it from continuing should emit a critical message.

11.1.2.6. Interactions

The following figure summarizes the different components when using log4j. Applications make logging calls on `Channel` objects. The `Channel` forwards to LogWriter logging requests for publication. `LogWriter` are registered with the handler associated to a `Channel`. Both `LogChannel` and `LogWritter` may use logging `LogLevel` to decide if they are interested in a particular logging request. A `LogWriter` can use a `LogFormatter` to format the message before publishing it to the output world.

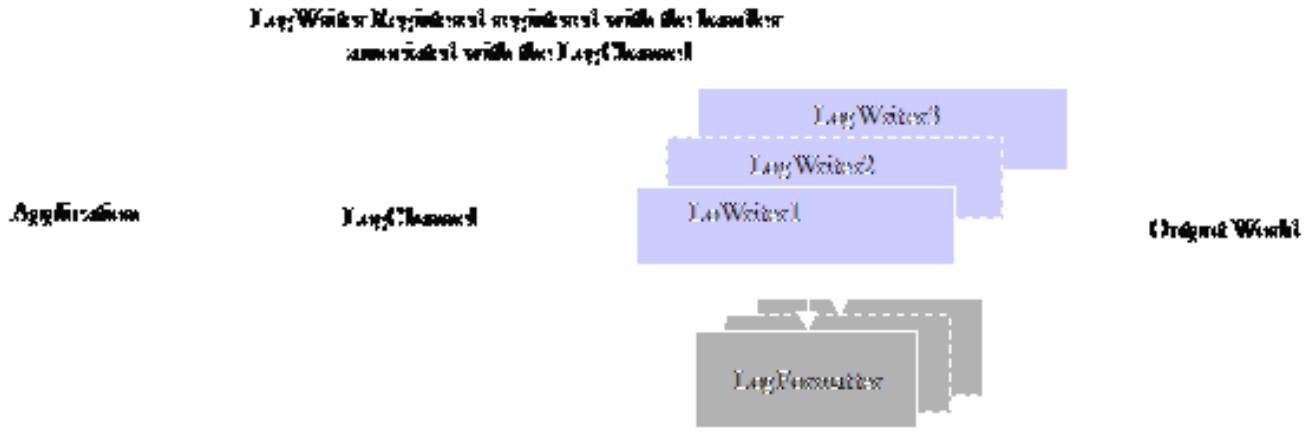


Figure 11.3. Basic interactions with HP Logger.

11.2. I18N and L10N

An application is internationalized, if it can correctly handle different encodings of character data. An application is localized, if it formats and interprets data (dates, times, timezones, currencies, messages and so on) according to rules specific to the user's locale (country and language).

Internationalization (I18N) is the process of designing an application so that it can be adapted to various languages and regions without engineering changes. Localization (L10N) is the use of locale-specific language and constructs at run time.

11.2.1. The Java Internationalization API

Java Internationalization shows how to write software that is multi-lingual, using Unicode, a standard system that supports hundreds of character sets. The Java Internationalization API is a comprehensive set of APIs for creating multilingual applications. The JDK internationalization features, from its version 1.1, include:

1. Classes for storing and loading language-specific objects.
2. Services for formatting messages, date, times, and numbers.
3. Services for comparing and collating text.
4. Support for finding character, word, and sentence boundaries.
5. Support for display, input, and output of Unicode characters.

11.2.2. Java Interfaces for Internationalization

Users of the Java internationalization interfaces should be familiar with the following interfaces included in the Java Developer's Kit (JDK):

1. `java.util.Locale` Represents a specific geographical, political, or cultural region.
2. `java.util.ResourceBundle` Containers for locale-specific objects
3. `java.text.MessageFormat` A means to produce concatenated messages in a language-neutral way.

11.2.3. Set the Locale

The concept of a Locale object, which identifies a specific cultural region, includes information about the country or region. If a class varies its behavior according to `Locale`, it is said to be locale-sensitive. For example, the `NumberFormat` class is locale-sensitive; the format of the number it returns depends on the `Locale`. Thus `NumberFormat` may return a number as 902 300 (France), or 902.300 (Germany), or 902,300 (United States). `Locale` objects are only identifiers.

Most operating systems allow to indicate their locale or to modify it. For instance Windows NT does this through the control panel, under the Regional Option icon. In Java, you can get the Locale object that matches the user's control-panel setting using `myLocale = Locale.getDefault();`. You can also create `Locale` objects for specific places by indicating the language and country you want, such as `myLocale = new Locale("fr", "CA");` for "Canadian French."

The next example creates Locale objects for the English language in the United States and Great Britain:

```
bLocale = new Locale("en", "US");
cLocale = new Locale("en", "GB");
```

The strings you pass to the Locale constructor are two-letter language and country codes, as defined by ISO standards.

11.2.4. Isolate your Locale Data

The first step in making an international Java program is to isolate all elements of your Java code that will need to change in another country. This includes user-interface text -- label text, menu items, shortcut keys, messages, and the like.

The `ResourceBundle` class is an abstract class that provides an easy way to organize and retrieve locale-specific strings or other resources. It stores these resources in an external file, along with a key that you use to retrieve the information. You'll create a ResourceBundle for each locale your Java program supports.

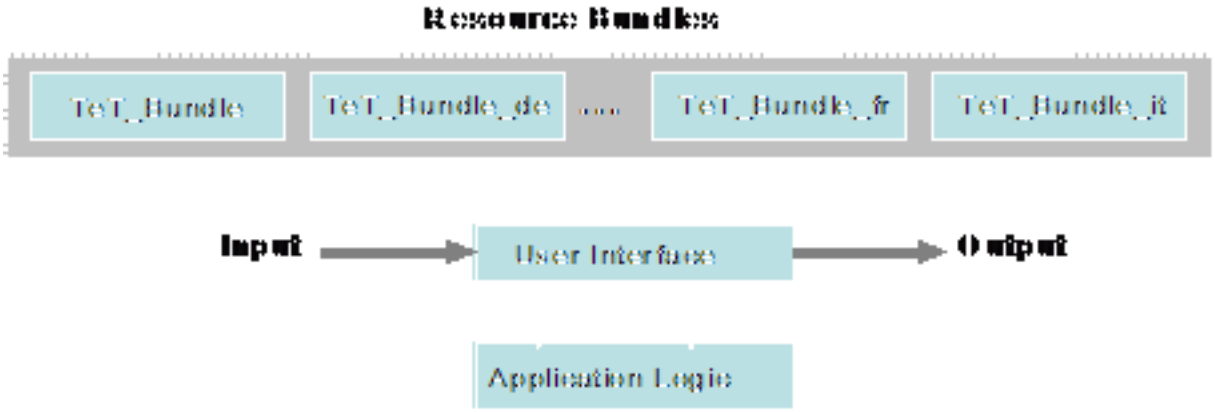


Figure 11.4. Resource Bundles.

The `ResourceBundle` class is an abstract class in the `java.util` package. You can provide your own subclass of `ResourceBundle` or use one of the subclass implementations, as in the case of `PropertyResourceBundle` or `ListResourceBundle`.

Resource bundles inherit from the `ResourceBundle` class and contain localized elements that are stored external to an application. Resource bundles share a base name. The base name `TeT_Bundle`, to display transactional messages such as *T#ransaction Commited#* might be selected because of the resources it contains. Locale information further differentiates a resource bundle. For example, `TeT_Bundle_it` means that this resource bundle contains locale-specific transactional messages for Italian.

To select the appropriate `ResourceBundle`, invoke the `ResourceBundle.getBundle` method. The following example selects the `TeT_Bundle ResourceBundle` for the Locale that matches the French language, the country of Canada.

```
Locale currentLocale = new Locale("fr", "CA");
```

```
    ResourceBundle introLabels = ResourceBundle.getBundle("TeT_Bundle", currentLocale);
```

Java loads your resources based on the locale argument to the `getBundle` method. It searches for matching files with various suffixes, based on the language, country, and any variant or dialect to try to find the best match. Java tries to find a complete match first, and then works its way down to the base filename as a last resort.

You should always supply a base resource bundle with no suffixes, so that your program will still work if the user's locale does not match any of the resource bundles you supply. The default file can contain the U.S. English strings. Then you should provide properties files for each additional language you want to support.

Basically, a resource bundle is a container for key/value pairs. The key is used to identify a locale-specific resource in a bundle. If that key is found in a particular resource bundle, its value is returned.

The jdk API defines two kinds of `ResourceBundle` subclasses -- the `PropertyResourceBundle` and `ListResourceBundle`.

A `PropertyResourceBundle` is backed by a properties file. A properties file is a plain-text file that contains translatable text. Properties files are not part of the Java source code, and they can contain values for `String` objects only. A simple default properties file, named `hpts_Bundle.properties`, for messages sent by HPTS could be.

```
    # Sample properties file for demonstrating PropertyResourceBundle
    # Text to inform on transaction outcomes in English (by default) trans_committed = Transaction Committed trans_rolledback=Tran
    # #
```

The equivalent properties file, `hpts_Bundle_fr_FR.properties`, for French would be:

```
    # Sample properties file for demonstrating PropertyResourceBundle
    # Text to inform on transaction outcomes in French trans_committed = La Transaction a #t# Valid#e trans_rolledback = La Transa
    # #
```

11.2.5. Example

The following example illustrates how to use the internationalization API allowing separating the text with a language specified by the user, from the source code.

```
    import java.util.*;
    import Demo.*;
    import java.io.*;
    import com.arjuna.OrbCommon.*;
    import com.arjuna.CosTransactions.*;
    import org.omg.CosTransactions.*;
    import org.omg.*;

    public class TransDemoClient {
      public static void main(String[] args) {
```

```
        String language; String country;
        if (args.length != 2) {
          language = new String("en");
          country = new String("US"); }
        else {
          language = new String(args[0]);
          country = new String(args[1]); }

        Locale currentLocale;
        ResourceBundle messages;
        currentLocale = new Locale(language, country);
        trans_message = ResourceBundle.getBundle( "hpts_Bundle", currentLocale);

        try {
          ORBInterface.initORB(args, null);
          OAInterface.initOA();

          String ref = new String();
          BufferedReader file = new BufferedReader(new FileReader("DemoObjReference.tmp"));
          ref = file.readLine();
          file.close();
          org.omg.CORBA.Object obj = ORBInterface.orb().string_to_object(ref);
          DemoInterface d = (DemoInterface) DemoInterfaceHelper.narrow(obj);

          OTS.get_current().begin();
          d.work();
          OTS.get_current().commit(true);
          System.out.println(tran_message.getString("trans_committed")); }
        catch (Exception e) {
          System.out.println(tran_message.getString("trans_rolledback")); }
        }
    }
```

In the following example the language code is *fr* (French) and the country code is *FR* (France), so the program displays the messages in French:

```
    % java TransDemoClient fr FR La Transaction a #t# valid#e
```

11.2.6. Creating Resource Bundles

The following ant task is provided in buildsystem.jar to automate the creation of resource bundles: `com.hp.mw.buildsystem.doclet.resbundledoclet.ResourceBundleDoclet`, which is a doclet for the JavaDoc tool that ships with the JDK. It produces resource bundle property files from comments placed in Java source. The comments have the following format:

```
/**
    * @message [key] [id] [text]
    * e.g., @message foo foo This is a message: {0}
    */
```

Where [key] is the key used to look up the corresponding message ([text]) in the resource bundle. The [id] field is typically the same as [key] but need not be: it is output with the internationalized message and is meant to be used by technical support in order to identify the [key][message] pair in a language independent manner.

It takes the following runtime options:

1. -resourcebundle [filename] This pecifies the name of the resource bundle to create, only use this if the Doclet is to produce a single resource bundle.
2. -basedir [directory] This specifies the base directory to generate the resource bundle property files within (MANDATORY).
3. -perclass This indicates that the doclet should produce resource bundles per class. If this is not specified then a single resource bundle properties file is produced for all of the source specified.
4. -ignorerepetition This indicates that the doclet should ignore key repetition and not flag an error.
5. -language [language code] This indicates which language is to be used
6. -locale [locale code] This indicates which locale is to be used.
7. -properties This indicates that the property filename should be postfixed with the .properties postfix.

The task can be declared within ant in the following way:

```
<doclet name="com.hp.mw.buildsystem.doclet.resbundledoclet.ResourceBundleDoclet">
                    <path>
                        <pathelement path="${com.hp.mw.ext.depends.classpath}"/>
                    </path>
                    <param name="-basedir" value="${com.hp.mwlabs.ts.arjuna.dest}"/>
                    <param name="-resourcebundle" value="${com.hp.mwlabs.ts.arjuna.resourcebundle}"/>
                </doclet>
```

11.2.7. Example of Use

Below is a sample of the internationalized messages used in the Transaction Service.

```
  /**
    * BasicAction does most of the work of an atomic action, but does not manage
    * thread scoping. This is the responsibility of any derived classes.
    *
    * @author Mark Little (mark@arjuna.com)
    * @version $Id: internationalization.xml,v 1.1 2006/03/07 17:59:23 mlittle Exp $
    * @since JTS 1.0.
    *
    *
    *
    * @message com.arjuna.ats.arjuna.coordinator.BasicAction_1
```

```
    * [com.arjuna.ats.arjuna.coordinator.BasicAction_1] - Action nesting
    * error - deletion of action id {0} invoked while child actions active
    * @message com.arjuna.ats.arjuna.coordinator.BasicAction_2
    * [com.arjuna.ats.arjuna.coordinator.BasicAction_2] - Aborting child
    * {0}
    * @message com.arjuna.ats.arjuna.coordinator.BasicAction_3
    * [com.arjuna.ats.arjuna.coordinator.BasicAction_3] - Destructor of
    * still running action id {0} invoked - Aborting
    * @message com.arjuna.ats.arjuna.coordinator.BasicAction_4
    * [com.arjuna.ats.arjuna.coordinator.BasicAction_4] - The Arjuna
    * licence only allows a single resource to be registered. Please apply
    * for a new licence.
    * @message com.arjuna.ats.arjuna.coordinator.BasicAction_5
    * [com.arjuna.ats.arjuna.coordinator.BasicAction_5] - Activate of
    * atomic action with id {0} and type {1} unexpectedly failed
    */
```

Which, when processed by the doclet, generates the following within the resource bundle:

```
com.arjuna.ats.arjuna.coordinator.BasicAction_1=[com.arjuna.ats.arjuna.coordinator.BasicAction_1] - Action nesting error - del
active
com.arjuna.ats.arjuna.coordinator.BasicAction_2=[com.arjuna.ats.arjuna.coordinator.BasicAction_2] - Aborting child {0}
com.arjuna.ats.arjuna.coordinator.BasicAction_3=[com.arjuna.ats.arjuna.coordinator.BasicAction_3] - Destructor of still runnin
com.arjuna.ats.arjuna.coordinator.BasicAction_4=[com.arjuna.ats.arjuna.coordinator.BasicAction_4] - The Arjuna licence only al
com.arjuna.ats.arjuna.coordinator.BasicAction_5=[com.arjuna.ats.arjuna.coordinator.BasicAction_5] - Activate of atomic action
```

11.3. The Common Logging Framework

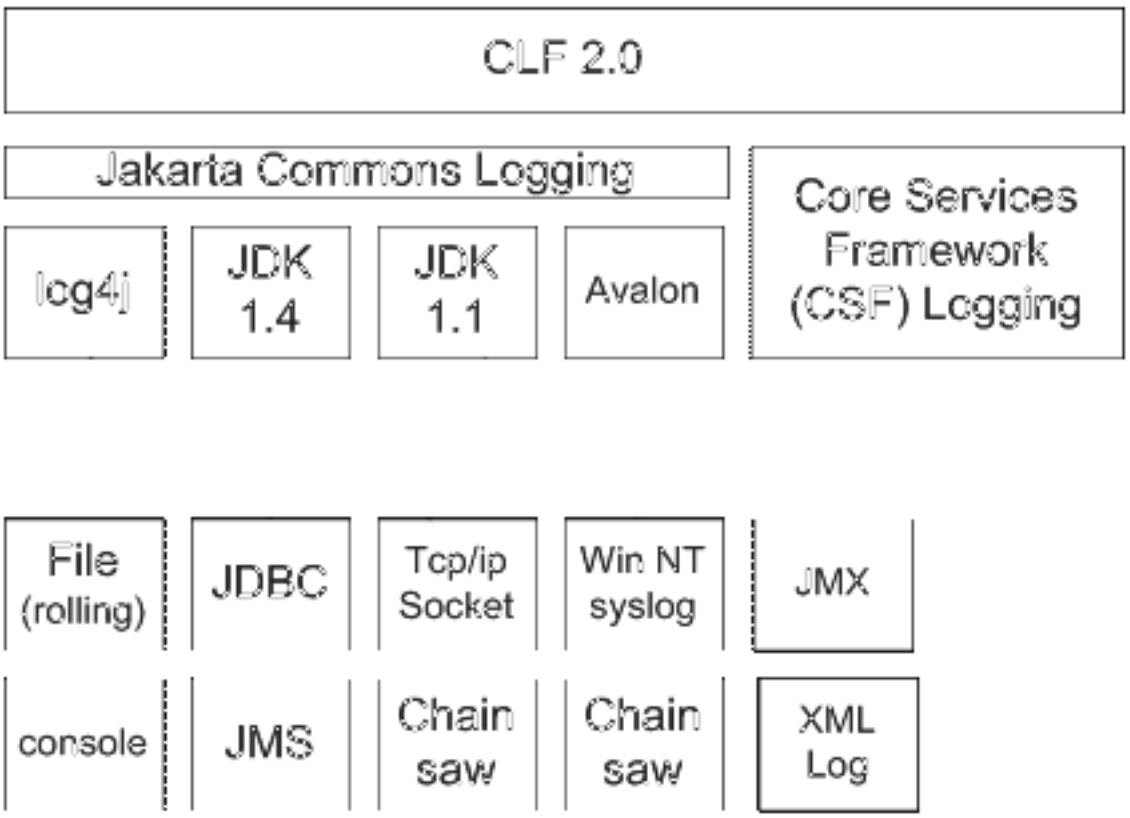


Figure 11.5. The Common Logging framework architecture.

11.3.1. Package Overview: `com.arjuna.common.util.logging`

11.3.1.1. Interface Summary

1. `Logi18n` A simple logging interface abstracting the various logging APIs supported by CLF and providing an internationalization layer based on resource bundles.

2. `LogNoi18n` A simple logging interface abstracting the various logging APIs supported by CLF without internationalization support

11.3.1.2. Class Summary

1. `CommonDebugLevel` The `CommonDebugLevel` class provides default finer debugging value to determine if finer debugging is allowed or not.

2. `CommonFacilityCode` The `CommonFacilityCode` class provides default finer facilitycode value to determine if

finer debugging is allowed or not.

3. `CommonVisibilityLevel` The `CommonVisibilityLevel` class provides default finer visibility value to determine if finer debugging is allowed or not.

4. `LogFactory` Factory for Log objects.

11.3.1.3. LogFactory

Factory for Log objects. LogFactory returns different subclasses of logger according to which logging subsystem is chosen. The log system is selected through the property `com.arjuna.common.utils.logger`. Supported log systems are:

1. `jakarta` Jakarta Commons Logging (JCL). JCL can delegate to various other logging subsystems, such as: log4j, JDK 1.4 logging, JDK 1.1 based logging (for compilation to Microsoft .net), Avalon

2. `dotnet` .net logging. (must be JDK 1.1 compliant for compilation by the Microsoft compiler)

> **Note**
> Rather than implementing CSF and .net logging as additional loggers for JCL they have been anchored at this level to maximise code reuse and guarantee that all .net dependent code is 1.1 compliant.

11.3.1.4. Setup of Log Subsystem

The underlying log system can be selected via the following property name:

1. `com.arjuna.common.util.logger` This property selects the log subsystem to use. Note that this can only be set as a `System` property, e.g. as a parameter to start up the client application: `java #com.arjuna.common.util.logger=log4j ....`

> **Note**
> Note: The properties of the underlying log system are configured in a manner specific to that log system, e.g., a `log4j.properties` file in the case that log4j logging is used.

The allowed values for the property are:

1. `log4j` Log4j logging (log4j classes must be available in the classpath); configuration through the `log4j.properties` file, which is picked up from the CLASSPATH or given through a `System` property: `log4j.configuration`

2. `jdk14` JDK 1.4 logging API (only supported on JVMs of version 1.4 or higher). Configuration is done through a file `logging.properties` in the jre/lib directory.

3. `simple` Selects the simple JDK 1.1 compatible console-based logger provided by Jakarta Commons Logging

4. `jakarta` Uses the default log system selection algorithm of the Jakarta Commons Logging framework

5. `dotnet` Selects a .net logging implementation. Since a dotnet logger is not currently implemented, this is currently identical to simple. Simple is a purely JDK1.1 console-based log implementation.

6. `noop` Disables all logging

To set log4j (default log system), provide the following System properties:

```
 -Dcom.arjuna.common.util.logger=log4j
 -Dlog4j.configuration=file://c:/Projects/common/log4j.properties
```

11.3.2. Getting Started

Simple use example:

```
import com.arjuna.common.util.logging.*;

public class Test {
static Log mylog =
LogFactory.getLog(Test.class);

public static void main(String[] args) {
String param0 = "foo";
String param1 = "bar";

// different log priorities mylog.debug("key1", new
Object[]{param0, param1});
mylog.info("key2", new Object[]{param0, param1});
mylog.warn("key3", new Object[]{param0, param1});
mylog.error("key4", new Object[]{param0, param1});
mylog.fatal("key5", new Object[]{param0, param1});

// optional throwable
Throwable throwable = new Throwable();
mylog.debug("key1", new Object[]{param0, param1}, throwable);
mylog.info("key2", new Object[]{param0, param1}, throwable);
mylog.warn("key3", new Object[]{param0, param1}, throwable);
mylog.error("key4", new Object[]{param0, param1}, throwable);
mylog.fatal("key5", new Object[]{param0, param1}, throwable);

// debug guard to avoid an expensive operation if the logger does not
// log at the given level:
if (mylog.isDebugEnabled()) {
String x = expensiveOperation(); mylog.debug("key6", new Object[]{x}); }

// ************************************************* //
fine-grained debug extensions
mylog.debug(CommonDebugLevel.OPERATORS, CommonVisibilityLevel.VIS_PUBLIC, CommonFacilityCode.FAC_ALL, "This debug message is e

mylog.setVisibilityLevel(CommonVisibilityLevel.VIS_PACKAGE);
mylog.setDebugLevel(CommonDebugLevel.CONSTRUCT_AND_DESTRUCT);
mylog.setFacilityCode(CommonFacilityCode.FAC_ALL);

mylog.mergeDebugLevel(CommonDebugLevel.ERROR_MESSAGES);

if (mylog.debugAllowed(CommonDebugLevel.OPERATORS, CommonVisibilityLevel.VIS_PUBLIC, CommonFacilityCode.FAC_ALL)) {
  mylog.debug(CommonDebugLevel.OPERATORS, CommonVisibilityLevel.VIS_PUBLIC, CommonFacilityCode.FAC_ALL, "key7", new Object[]{"
```

11.4. Default File Level Logging

Independent of the log system chosen, it is possible to log all messages over a given severity threshold into a file. This is useful to guarantee that e.g., error and fatal level messages are not lost despite a user has not set up a log framework, such as log4j

11.4.1. Setup

Usage of this feature is simple and can be controlled through a set of properties. These can be provided through the Property Manager or as System properties.

Table 11.1. Properties to control default file-based logging (default values are highlighted)

Property Name	Values	Description
`com.arjuna.common.logging.default`	true/ *false*	Enable/disable default file-based logging
`com.arjuna.common.util.logging.default.level`	*Info* /error/fatal	Severity level for this log
`com.arjuna.common.util.logging.default.showLogName`	true/ *false*	Record the fully qualified log name
`com.arjuna.common.util.logging.default.showShortLogName`	*true* /false	Record an abbreviated log name
`com.arjuna.common.util.logging.default.showDate`	*true* /false	Record the date
`com.arjuna.common.util.logging.default.logFile`	*error.log* (default)	File to use for default logging. This can be an absolute filename or relative to the working directory
`com.arjuna.common.util.logging.default.logFileAppend`	*true* /false	Append to the log file above in case that this file already exists

11.5. Fine-Grained Logging

11.5.1. Overview

Finer-grained logging in CLF is available through a set of debug methods:

```
public void debug(long dl, long vl, long fl, Object message);
public void debug(long dl, long vl, long fl, Throwable throwable);
public void debug(long dl, long vl, long fl, String key, Object[] params);
public void debug(long dl, long vl, long fl, String key, Object[] params, Throwable throwable);
```

All of these methods take the three following parameters in addition to the log messages and possible exception:

`dl` - The debug finer level associated with the log message. That is, the logger object will only log if the DEBUG level is allowed and `dl` is either equal or greater than the debug level assigned to the logger Object. See the table below for possible values.

`vl` - The visibility level associated with the log message. That is, the logger object will only log if the DEBUG level is allowed and `vl` is either equal or greater than the visibility level assigned to the logger Object. See the table below for possible values.

`fl` - The facility code level associated with the log message. That is, the logger object will only log if the DEBUG level is allowed and `fl` is either equal or greater than the facility code level assigned to the logger Object. See the table below for possible values.

The debug message is sent to the output only if the specified debug level, visibility level, and facility code match those allowed by the logger.

Note

The first two methods above do not use i18n. i.e., the messages are directly used for log output.

11.5.2. Usage

Possible values for debug finer level, visibility level and facility code level are declared in the classes `DebugLevel`, `VisibilityLevel` and `FacilityCode` respectively. This is useful for programmatically using fine-grained debugging.

Table 11.2. Possible settings for finer debug level (class DebugLevel)

Debug Finer Level	Value	Description
`NO_DEBUGGING`	0x0000	No debugging
`CONSTRUCTORS`	0x0001	Only output for constructors
`DESTRUCTORS`	0x0002	Only output for finalizers
`CONSTRUCT_AND_DESTRUCT`	CONSTRUCTORS | DESTRUCTORS	
`FUNCTIONS`	0x0010	Only output for methods
`OPERATORS`	0x0020	Only output for methods such as equals, notEquals etc.
`FUNCS_AND_OPS`	FUNCTIONS | OPERATORS	
`ALL_NON_TRIVIAL`	CONSTRUCT_AND_DESTRUCT | FUNCTIONS | OPERATORS	

Debug Finer Level	Value	Description
TRIVIAL_FUNCS	0x0100	Only output from trivial methods
TRIVIAL_OPERATORS	0x0200	Only output from trivial operators
ALL_TRIVIAL	TRIVIAL_FUNCS | TRIVIAL_OPERATORS	
ERROR_MESSAGES	0x0400	Only output from debugging error/warning messages
FULL_DEBUGGING	0xffff	Output all debugging messages

Table 11.3. Possible settings for visibility level (class VisibilityLevel)

Visibility Level	Value	Description
VIS_NONE	0x0000	No visibility
VIS_PRIVATE	0x0001	Only from private methods
VIS_PROTECTED	0x0002	Only from protected methods
VIS_PUBLIC	0x0004	Only from public methods
VIS_PACKAGE	0x0008	Only from package methods
VIS_ALL	0xffff	Output all visibility levels.

Table 11.4. Possible settings for facility code level (class FacilityCode)

Facility Code Level	Value	Description
FAC_NONE	0x0000	No facility
FAC_ALL	0xffffffff	Output all facility codes

At runtime, the fine-grained debug settings are controlled through a set of properties, listed in the table below:

Table 11.5. Controlling finer granularity

Property Name	Default Value
com.arjuna.common.util.logging.DebugLevel	NO_DEBUGGING
com.arjuna.common.util.logging.VisibilityLevel	VIS_ALL
com.arjuna.common.util.logging.FacilityCode	FAC_ALL

12

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.

12.1. 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.

12.1.1. Build JBoss

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

Unix

```
cd build
./build.sh
```

Windows

```
cd build
build.bat
```

12.1.2. 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.

12.1.3. 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
```

12.1.4. 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.

You can also use the udpGroup property to prevent your clustering tests from interfering with others on the same network using the udpGroup property. This can be passed at the command line or in the local.properties file. This will be passed to the servers under test using the -u option:

```
./build.sh -DudpGroup=128.1.2.3 tests
...
[server:start] java org.jboss.Main -c minimal -b localhost -u 128.1.2.3
```

12.1.5. 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/output/reports/text`.

12.2. 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.

12.2.1. Targets

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

Table 12.1. 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.

12.2.2. 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 12.2. Summary of build files

Build File	Description
`testsuite/build.xml`	Contains test targets. This file imports the macros and targets from the files below.
`testsuite/imports/server-config.xml`	Contains macros for creating and starting different server configurations.

Build File	Description
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).

12.3. 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`.

12.3.1. 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 12.1. 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 1.4 2006/02/22 20:56:55 rgenova 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>
```

12.4. 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>
```

12.5. 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;
   }
```

12.6. 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>
```

12.7. Excluding Bad Tests

If a test cannot be fixed for some reason, it should be added to the bad.test excludes. This is maintained near the top of the testsuite/build.xml. The patternset will be used to exclude tests all calls to JUnit within the testsuite.

```
<!-- Tests that are currently broken -->
<patternset id="badtest.excludes">
  <exclude name="org/jboss/test/aop/test/RemotingUnitTestCase"/>
  <!-- The media ejb is not active -->
  <exclude name="org/jboss/test/media/**"/>
  <!-- Needs apache ? -->
  <exclude name="org/jboss/test/cluster/httpsessionreplication/**"/>
  <exclude name="org/jboss/test/cache/test/local/ConcurrentTransactionalUnitTestCase.class"/>
</patternset>
```

13

Support and Patch Management

13.1. Introduction

Customer requested fix can be made in a cumulative patch, one-off patch, or both. Normally, most of the bug fixes are included into the cumulative patch, so there is no need to create one-off patches. One-off patches can still be created for the emergency production and security issues. However, it's strongly advised to merge all the bug fixes into the cumulative patches.

13.1.1. Cumulative Patch

1. Cumulative patches contain only customer requested bug fixes and security patches
2. Cumulative patches are additive, i.e. JBossAS-4.0.4.GA_CP02 includes all of the fixes from JBossAS-4.0.4.GA_CP01
3. Released on 2nd Tuesday of each month. Code freeze for all cumulative patch branches is on 1st day of each month.
4. Separate cumulative patch branch is created for every supported product release, i.e. https://svn.jboss.org/repos/jbossas/branches/JBoss_4_0_2_CP.
5. All customers get access to available cumulative patches
6. All the fixes included into a cumulative patch need to be included into the next maintenance release
7. It's required to have test case(s) for each fix included into the cumulative patch.
8. Cumulative patches are tested by QA so all fixes are tested in combination

13.1.2. One-off Patch

1. Isolated branch is created for every patch, i.e. https://svn.jboss.org/repos/jbossas/branches/JBoss_4_0_3_SP1_JBAS-2859.
2. Every one-off patch is tested by QA
3. It's advised to minimize number of one-off patches, and virtually all one-off patches are supposed to be merged into the cumulative patches

13.2. Support Workflow

1. Customer requests a fix.
2. If the issue is fixed in a newer GA/SP version of the product, Support verifies if upgrade is an option
3. Support engineer raises issue in the patch project (ASPATCH for the application server)
4. If support engineer has concerns, they mark the patch as "Request Triage". Patch is assigned to patch-triage team for resolution
5. If issues are resolved, the patch continues through the CP process. Support notifies customer when the next cumulative patch release is scheduled for
6. Otherwise, the patch is done as a one-off. For instance, if customer requires an emergency fix and can’t wait until the next cumulative patch release, a one-off patch is created. The fix is also merged into the upcoming cumulative patch
7. After Development, QA, and JBN are done, Support provides patch download link to the customer
8. If customer has a problem with the cumulative patch release, create a bug against it. For the app server, create a bug under the ASPATCH project. Make sure that every ASPATCH issue is public.

13.3. Cumulative Patch Process

13.3.1. Development Phase

1. Create an issue in JIRA with a type "Task" under the appropriate PATCH project. For the App Server it's AS-PATCH project. Such JIRA issues need to be public.
2. Set Fix Version field value appropriately. For instance, JBossAS-4.0.4.GA_CP03. It's easy to find out which Cumulative Patch release is next by looking at the roadmap in JIRA.
3. Link the issue with the appropriate bug issue using "incorporates".
4. Click the "Start Progress" link

   Check out appropriate cumulative patch branch, for instance for the AS4.0.2 release

```
svn checkout https://svn.jboss.org/repos/jbossas/branches/JBoss_4_0_2_CP
```

5. Implement the fix and check it in.

   If a bug is in app server component such as Web Services or Hibernate, a cumulative patch branch must be created off of the component release that was included into the app server. For instance, for Hibernate 3.1.3:

a. Create Hibernate_3_1_3_CP branch

b. Implement or merge the fix into the Hibernate_3_1_3_CP branch

c. Tag Hibernate_3_1_3_CP branch as Hibernate_3_1_3_CP0x where x is a number to be incremented with every fix.

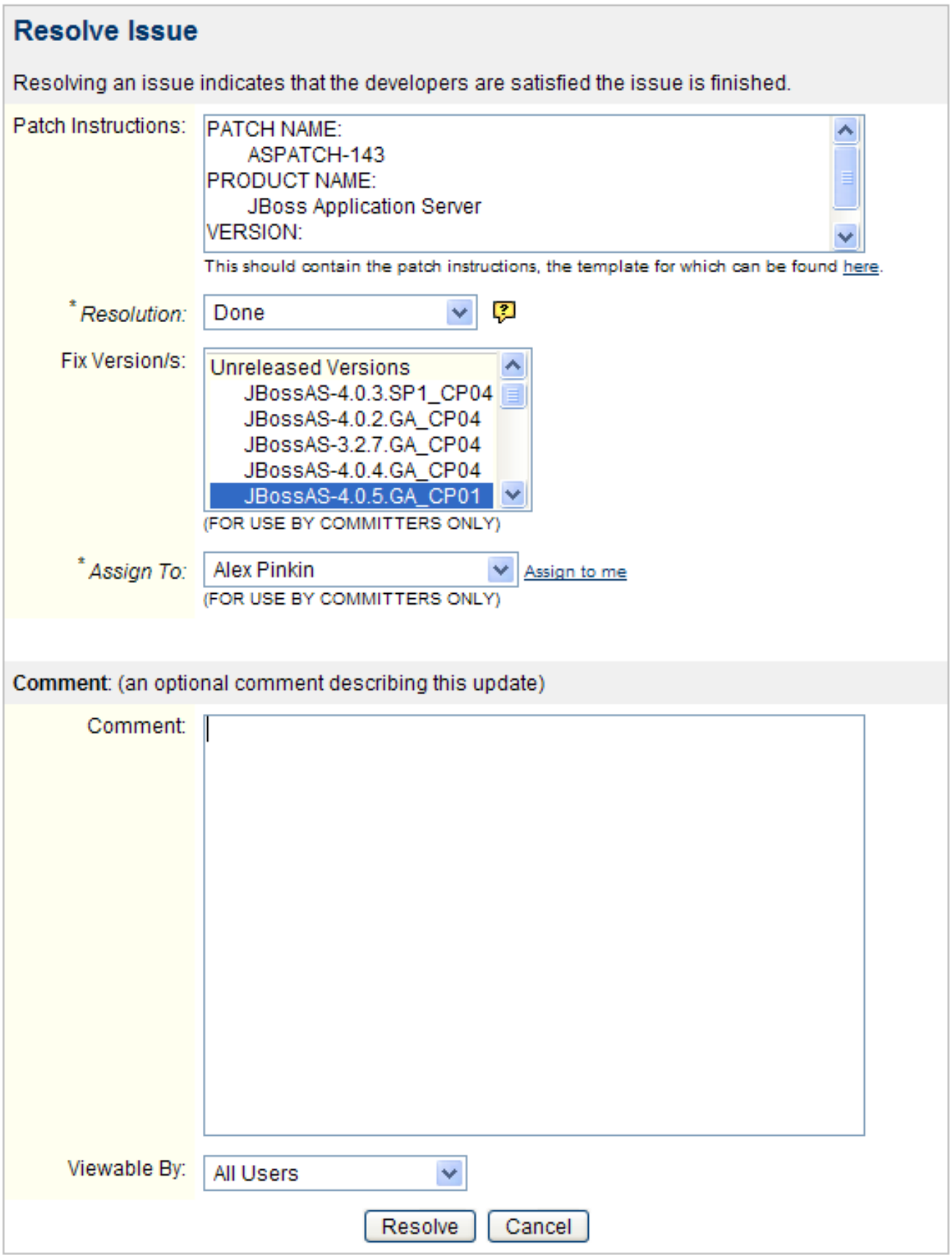
d. Follow normal release process for the component but treat the release as internal. For instance, you should end up with Hibernate 3.1.3_CP01 component in the repository.

e. Update App Server's build-thirdparty.xml with the appropriate version of the Hibernate CP release, i.e. 3.1.3_CP01

6. If the issue being resolved does not have a test case, it should be added to the testsuite.

7. If a regression is introduced, CruiseControl will notify.

8. Click "Resolve". In the "Patch Instructions" field enter installation instructions on how to apply the patch.



Use the Support Patch Template as a basis for these instructions.

13.3.2. QA Phase

1. Within 48 hours after the cumulative branch code freeze, QA will assign the release task and set a due date for the QA process to be complete.
2. QA will tag cumulative patch branch upon code freeze. For instance,

```
svn copy https://svn.jboss.org/repos/jbossas/branches/JBoss_4_0_2_CP https:/
```

3. QA will run the testsuite to verify that the cumulative patch produces no regressions. If the tests fail, QA will submit a blocker issue and it will be assigned to the developer for root cause analysis.
4. See How To QA a Cumulative Patch..
5. QA will mark the issue as "QA Tests Passed" if the tests passed.
6. QA will assemble cumulative patch package for the JBN
7. QA will test deployment of the cumulative patch with JBN team

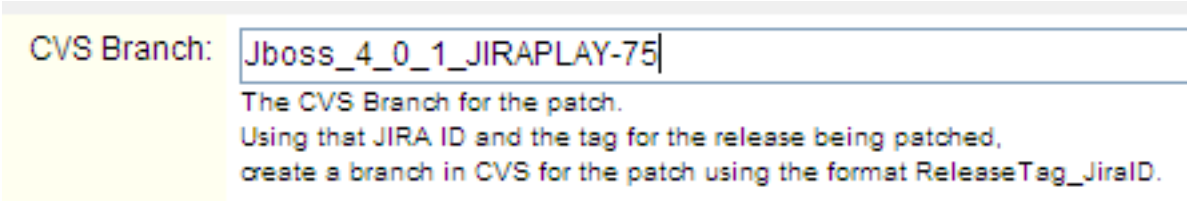
13.3.3. JBN Phase

1. The patch will be deployed by the JBossNetwork team.

13.4. One-Off Patch Process

13.4.1. Development Phase

1. Create an issue in JIRA with a type "Support Patch" under the appropriate project. For the App Server, it's JBAS project.
2. Be sure to make the issue either "Customers Only" or "JBoss Internal"
3. Click the "Start Progress" link to enter the CVS or SVN branch information.



   Enter the branch for the patch. Using that JIRA ID and the tag for the release being patched, create a branch for the patch using the format ReleaseTag_JiraID.

   For instance, if the patch issue ID is JBAS-1234 for JBoss 4.0.3, the patch's branch should be

JBoss_4_0_3_JBAS-1234.

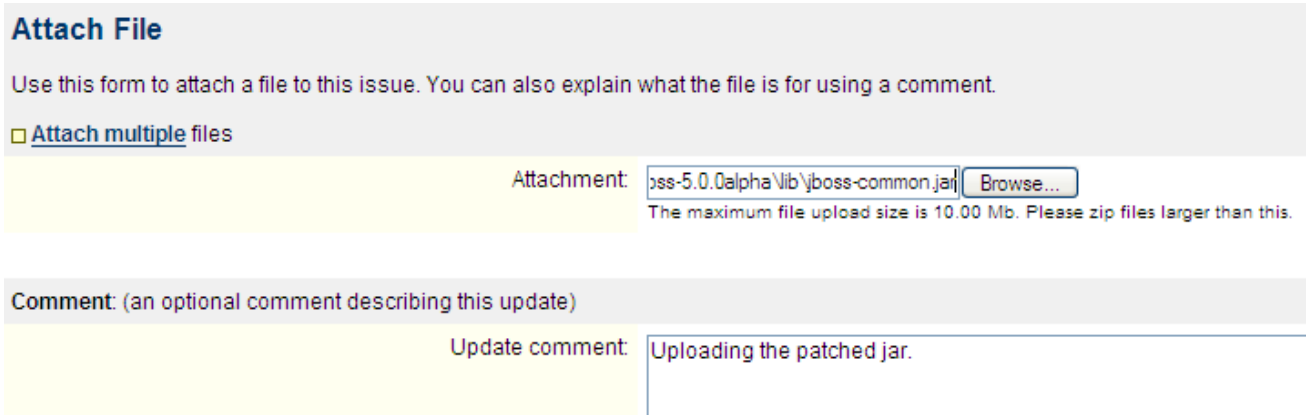
```
# svn
svn copy https://svn.jboss.org/repos/jbossas/tags/JBoss_4_0_3_SP1/ http:
svn checkout https://svn.jboss.org/repos/jbossas/branches/JBoss_4_0_3_SP

# cvs
cvs rtag -r JBoss_4_0_3 -b JBoss_4_0_3_JBAS-1234 jboss-4.0.x
cvs co -r JBoss_4_0_3_JBAS-1234 jboss-4.0.x
```

If the project being patched is not in JBoss hosted CVS, attach the source diff to this case.

4. If the issue being resolved does not have a test case, it should be added to the testsuite.

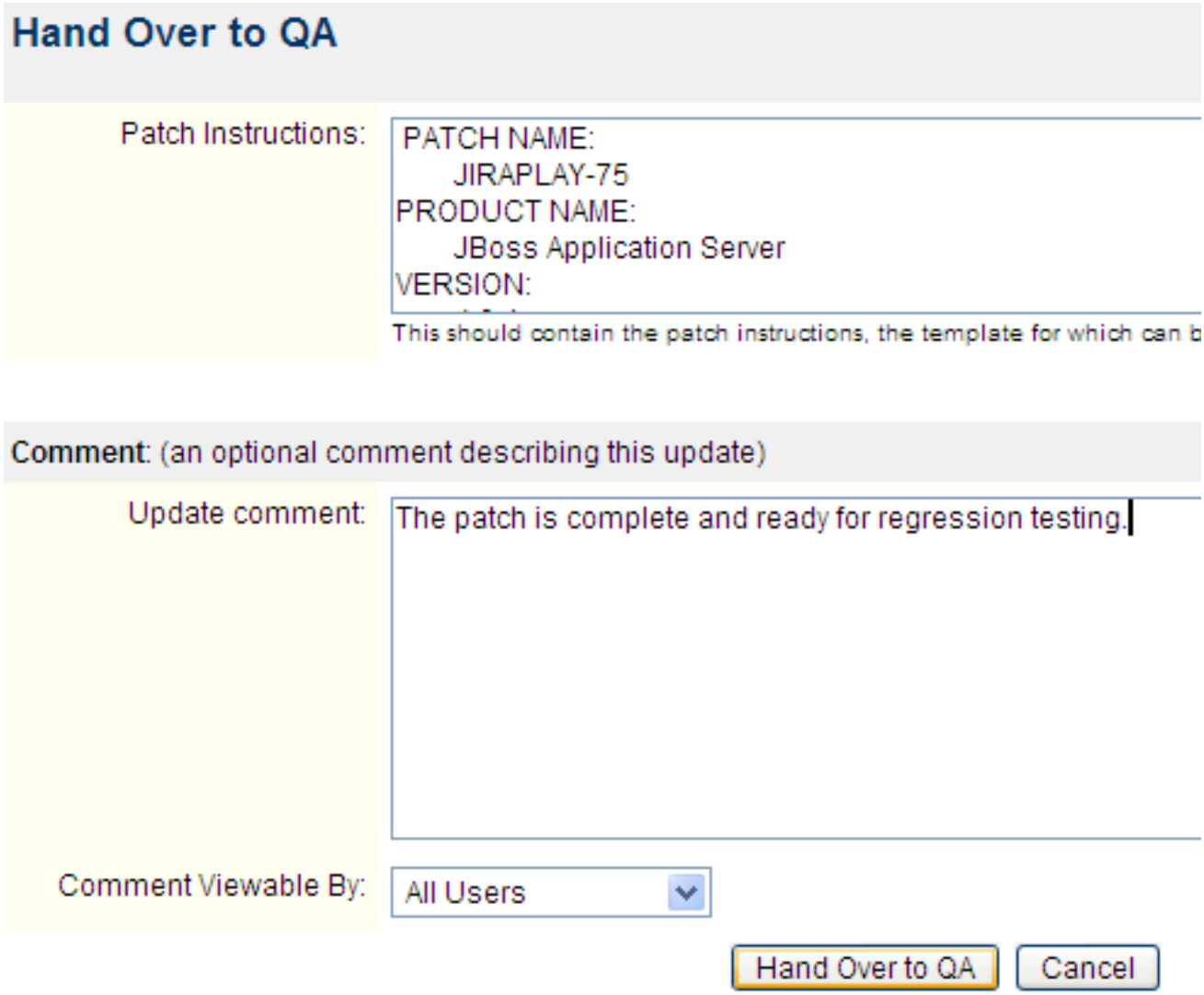
5. When committing the patch to the branch, be sure to include the JIRA ID of the patch in the commit comment.

6. Build the patched jars from the above branch and attach them to the JIRA issue.



7. Merge the patch into the appropriate cumulative patch branch(es) by following steps described in the cumulative patch section

8. Click "Hand Over to QA". In the "Patch Instructions" field enter installation instructions on how to apply the patch to an existing installation.



Use the Support Patch Template as a basis for these instructions.

9. Mark the issue as "Pass to QA".

13.4.2. QA Phase

1. Within 48 hours, QA will assign the task and set a due date for the QA process to be complete. If the due date is more than 3 days after the development is complete, QA will comment on the case and let the customer know when the patch will be ready.

2. QA will run the testsuite to verify that the patch produces no regressions. If the tests fail, QA will mark the issue as "QA Tests Failed" and it will be reassigned to the developer.

3. See How To QA a One-Off Support Patch..

4. QA will mark the issue as "QA Tests Passed" if the tests passed.

5. If tests pass, attach JARS to Support Portal case. Download jars and verify that MD5 sums match originals.

13.4.3. JBN Phase

1. The patch will be deployed by the JBossNetwork team. This should include linking the issue to the JBN patch.

13.5. Support Patch Instructions Template

Below is the skeleton of the directions that should be entered in the "Patch Instructions" field in JIRA.

```
PATCH NAME:
        [Not needed for the Cumulative Patch tasks]
        JBAS-XXXX
PRODUCT NAME:
        [Not needed for the Cumulative Patch tasks]
        JBoss Application Server
VERSION:
        [Not needed for the Cumulative Patch tasks]
        4.0.2
SHORT DESCRIPTION:
        [What problem this patch fixes.]
        ex: "A NullPointerException is no longer thrown when the password field is left blank."
LONG DESCRIPTION:
        [Detailed explanation of the problem.]
        ex: "Prior to this fix, blah happened. With this fix blah will happen instead.  This is because b
MANUAL INSTALL INSTRUCTIONS:
        [How a user should manually install this patch.]
        ex: "Rename %JBOSS_HOME%/lib/someJar.jar to "someJar.replacedBy.JBAS-xxxx.jar.old"
        Copy the new someJar.jar to %JBOSS_HOME%/lib/"
COMPATIBILITY:
        [known usages and known combinations that don't work]
        ex: "portal 2.x in a given jems bundle does not work with this change"
DEPENDENCIES:
        [list any patches this patch is dependent on. Not needed for the Cumulative Patch tasks]
        ex: 4.0.2-SP2
SUPERSEDES:
        [list any patches this patch supersedes]
        ex: JBAS-1450
SUPERSEDED BY:
        [list any patches this patch is superseded by. Not needed for the Cumulative Patch tasks]
        ex: 4.0.2-SP3
CREATOR:
```

```
        [author of this patch]
DATE:
        [date these instructions were written]
```

13.6. How To QA a One-Off Support Patch

```
# Howto test a patch in the QA lab

# create a directory which corresponds to the JIRA id for the patch
mkdir JBAS-1234
cd JBAS-1234

#set your JAVA_HOME correctly
#if jboss 3.x, use jdk 1.3
#export JAVA_HOME=/opt/jdk1.3.1_13/

#if jboss 4.x use jdk 1.4
export JAVA_HOME=/opt/j2sdk1.4.2_09/

#Make sure you are testing on a 32-bit box

# get the source distribution of the *targetted* version of jboss
tar xvzf /opt/src/jboss-4.0.2-src.tar.gz

# build Jboss & save the original output
cd jboss-4.0.2-src/build; sh build.sh
cp -r output output-orig
cd ../..

# download the patched binaries into a binaries directory
mkdir binaries; cd binaries; #download them locally & upload via scp (if in qa lab)

#TODO: look at using wget to retrieve patches

# capture the md5 of each jar and add it as comment to jira
md5sum *.jar

#use 'find' to locate where in the output each jar is, in
#order to create the right mirror of the install distro

find ../jboss-4.0.2-src/build/output -name jboss.jar

# create a mirror of the install distro using the patched jars
#for each configuration 'find' listed
mkdir -p jboss-4.0.2/server/all/lib
cp jboss.jar jboss-4.0.2/server/all/lib

# copy the jars to the output, verifying each jar copies correctly
# NOTE: make sure to replace all the jars in the source tree
# not just the ones under build/output since the client classpath
# is affected by the jars under /thirdparty and */output/lib.
cp -ivr jboss-4.0.2 ../jboss-4.0.2-src/build/output

# save the patched version for later (to switch back & forth)
cd ../jboss-4.0.2-src/build
cp -r output output-patched

# run the testsuite against the patched version (let node0 default to localhost)
cd ../testsuite
```

```
sh build.sh -Dnode1=$MYTESTIP_1 tests

# tests will take 1-2 hours, to save time, verify pass rate every 15 min
# if you see several failures, make sure no one else is on
who
ps --columns 1000 -ef | grep run.jar

# once tests complete, save the text report and take a look at it
cp output/reports/text/TESTS-Testsuite.txt ../../TESTS-patched.txt

# verify any failures also fail on the unpatched version
# upon any failure contact QA Lead for help
cd ../build;
mv output output-patched
cp -r output-orig output

#find out which mode the failing tests were using and
#start the server using the particular configuration
#the example below starts the 'all' mode
cd output/jboss-4.0.2/
./bin/run.sh -c all

#different shell
cd jboss-4.0.2-src/testsuite
sh build.sh -Dtest=org.jboss.tests.the.FailingTest one-test

# if failure only occurs on patched version, reject the patch
# Attach necessary information to JIRA issue
# ie, testuite/output/reports/TESTS-org.jboss.test.the.FailingTest.txt

#  When failing the patch notify developer that patched client jars
#  were note used, ie, add this as a comment:
#  NOTE: Patched client jars were not used to validate compatibility.

# Upon failure tar up the issue directory and copy to /home/issues
cd $ISSUE_HOME/..
tar cvzf JIRA-1234.tar.gz JIRA-1234
cp JIRA-1234.tar.gz /home/issues

#note path to tar.gz on JIRA issue
```

13.7. How To QA a Cumulative Patch

To be filled in by QA.

14

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.

15

Documentation and the Documentation Process

15.1. 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].

15.2. 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.

15.2.1. Responsibilities

15.2.1.1. 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.

15.2.1.2. 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.

15.2.2. 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.

15.2.3. Keep the documentation up-to-date

Since our technology is evolving fast, 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.

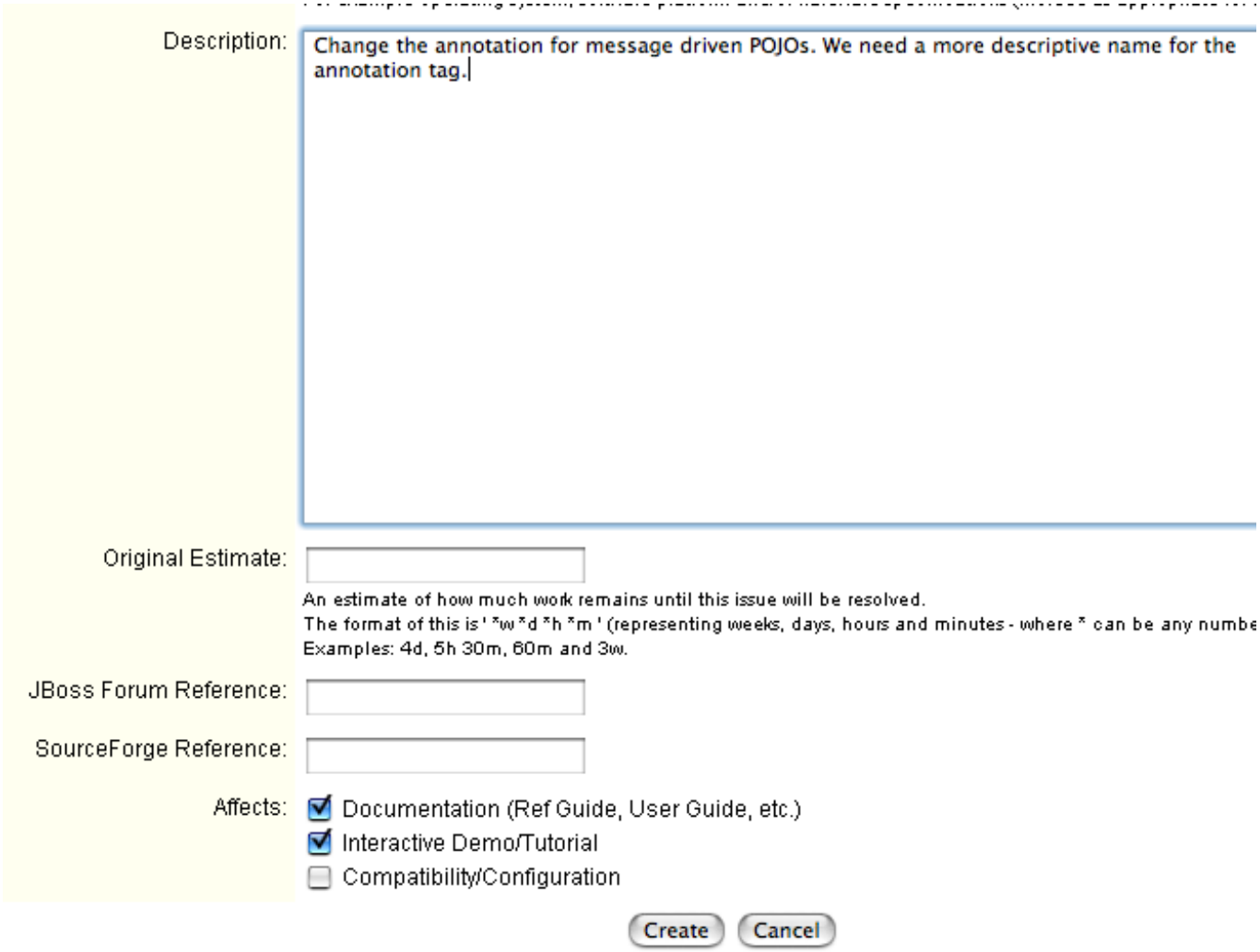


Figure 15.1. Check the "affects" boxes for a task that changes the public API

- 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.

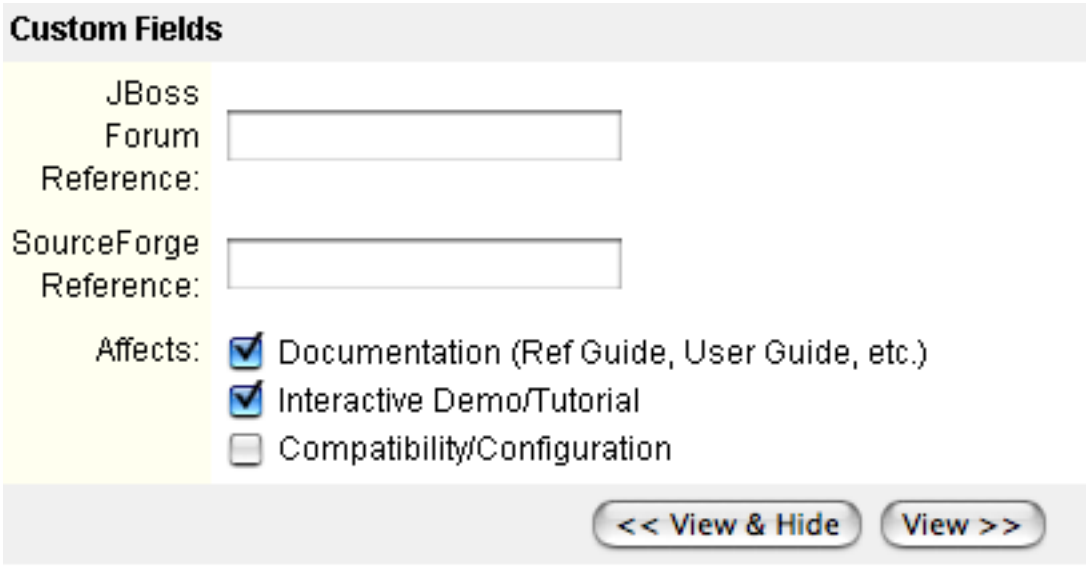


Figure 15.2. Find all tasks that affect docs

15.2.4. 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.

15.2.5. 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 docbook-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`

16

JBoss QA Lab Guide

This chapter provides JBoss developers with a guide to the QA lab. It covers usage guidelines as well as setup documentation

16.1. Quick Start Guide

Use ssh to log into dev02.pub.qa.atl.jboss.com. All of the servers in the Q/A lab have the same public keys and accounts as cvs.jboss.com. If you can't connect, contact it-ops@jboss.com. Dev02 is available for ad-hoc testing by developers and support engineers. No scheduling is required for this box.

To avoid port conflicts with others, bind listeners to your assigned address. Currently, each account has 4 IP addresses available. These are available as environment variables: $MYTESTIP_1,$MYTESTIP_2, $MYTESTIP_3, $MYTESIP_4. For instance:

```
./run.sh -b $MYTESTIP_1 -c default
```

You will find assorted tools and JVM's under /opt. If you can't find something you need, please raise the issue on the JBIT project in JIRA.

16.2. Lab Setup

16.2.1. Topology

Each node has a public IP address for services such as SSH and HTTP, in addition to 256 private IP addresses. The local network is Gigabit Ethernet, with connectivity to the public Internet via a DS3.

To prevent port conflicts, each listener/server process must be bound to a specific private IP address. These IP addresses are delineated per node and per user in the /opt/etc/assigned-ips file, available on each machine. Currently, each user is assigned 4 IP addresses per node. These are automatically set as the environment variables $MYTESTIP_1 $MYTESTIP_2, etc. All ports for a given private IP address may be accessed via SSH tunneling.

16.2.2. File System

The /opt and /home directories on each machine are mapped to the corresponding directories on dev01 via NFS. So your home directory is the same on each machine. The /opt directory holds common packages needed across all nodes, preventing unecessary duplication. Under /opt, you will find several useful packages such as:

- Many jdks, including IBM, Sun & JRockit
- JDBC drivers under /opt/jdbc-drivers
- Under /opt/src, several binary & src distros of JBoss
- Several versions of Ant

16.2.3. Databases

This section documents the available databases in the QA lab, how to start, stop & access them. JDBC drivers are available under /opt/jdbc-drivers.

For sudo acces to sybase & oracle, see "How do I get DB admin access?"

Table 16.1. Available Databases

Database	Connection URL	Start/Stop	New User

Oracle 10.1.0.3	jdbc:oracle:thin:@dev01-priv:1521:qadb01	`/etc/init.d/dbora start|stop`	`$ sudo su - oracle`
			`$ export ORACLE_SID=qadb01`
			`$ sqlplus "/ as sysdba "`
			`CREATE USER your_username_here`
			`IDENTIFIED BY your_password_here`
			`DEFAULT TABLESPACE users`
			`TEMPORARY TABLESPACE temp`
			`QUOTA UNLIMITED ON users;`
			
			`GRANT DBA to your_username_here;`

MySQL 4.1.10a	jdbc:mysql://dev01-priv/your_database_here	`/etc/init.d/mysql.server sta`	`$ /opt/mysql/bin/mysql -u qa -h qadb01 -p`
			`Enter password: #request from JBIT in JIRA`
			`GRANT ALL PRIVILEGES ON your_db_here.*`
			`TO 'your_username_here'@'%'`
			`IDENTIFIED BY 'your_password_here';`
			`flush privileges;`
			`create database your_database_here;`

Sybase ASE 12.5.2	jdbc:sybase:Tds:dev01-priv:4100	`sudo su - sybase`
		`startserver -f \`
		`~/sybase/ASE-12_5/install/RU`
		`#stop`
		`sudo su - sybase`
		`isql -Usa -P -Ssybase01`
		`shutdown`
		`go`
			`$ sudo su - sybase`
			`$ isql -Usa -P -Ssybase01`
			`sp_addlogin "yourusernamehere", "yourpassw`
			`go`
			`create database yourdbhere`
			`go`
			`use yourdbhere`
			`go`
			`sp_changedbowner yourusernamehere`
			`go`

16.2.4. Servers

This section lists the available servers and their configurations. We can move between Windows/SLES/RHEL very quickly. Please ask if you need a different OS. All servers have the domain name of dev??.pub.qa.atl.jboss.com.

Table 16.2. Available Servers

Host	Purpose	OS	CPU	Memory
dev01	Database/NFS server	RHEL 3/2.4 Kernel	2 x 3.06 GHz P4 Xeon	4GB
dev02	General usage	RHEL 3/2.4 Kernel	2 x 3.06 GHz P4 Xeon	2GB
dev03	General usage	RHEL 3/2.4 Kernel	2 x 3.06 GHz P4 Xeon	2GB
dev04	General usage	SLES 9/2.6 Kernel	2 x 3.06 GHz P4 Xeon	2GB
dev05	Cruisecontrol - no general access	RHEL 3/2.4 Kernel	2 x 3.06 GHz P4 Xeon	2GB
dev07	General usage	RHEL 4/2.6 Kernel/ 64 bit	2 x 1.4 GHz IA-64 Itanium 2	2GB
dev08	General usage	RHEL 4/2.6 Kernel/ 64 bit	2 x 1.4 GHz IA-64 Itanium 2	2GB
dev12	General usage	Solaris 8	2 x 1.2HGz Sun V210	4GB
dev13	General usage	Solaris ?	2 x 1.8GHz Sun V20z	
dev14	General usage	RHEL 4/2.6 Kernel/ 64 bit	1 x 1.5 GHz IA-64 Itanium 2	4 GB
dev15	General usage	RHEL 4/2.6 Kernel/ 64 bit	1 x 1.5 GHz IA-64 Itanium 2	4 GB
dev16	SPECJ	RHEL 4/2.6 Kernel/ 64 bit	2 x 3.6 GHz Xeon EM64T	4 GB
dev17	SPECJ	RHEL 4/2.6 Kernel/ 64 bit	2 x 3.6 GHz Xeon EM64T	4 GB
dev18	General usage	RHEL 4/2.6 Kernel/ 64 bit	4 x 1.5 GHz IA-64 Itanium 2	4 GB
dev19	General usage	RHEL 4/2.6 Kernel/ 64 bit	4 x 1.5 GHz IA-64 Itanium 2	4 GB

16.3. QA Lab FAQ

16.3.1. General

16.3.
16.3. How should I run the testsuite in the QA Lab?
1.1.

./build.sh tests -Dnode0=$MYTESTIP_1 -Dnode1=$MYTESTIP_2

16.3.
16.3. I'm getting port conflicts, how do I fix this?
1.2.

```
sudo /usr/sbin/lsof -i -P | grep 1099
```

This should display who has JBoss instances running. If they have not bound to their private IP address, it will conflict even if you are doing so.

16.3.
16.3. How do I get DB admin access?
1.3.

On dev01, check to see if you are in the oracle_admin group:

```
$ groups
```

If you don't see oracle_admin listed, open a JIRA issue in the JBIT project requesting access to this group.

16.3.
16.3. How do I add disk space for Sybase?
1.4.

Get into the isql prompt as above, and

```
disk resize name="master", size="200M"
go
```

17

Project Release Procedures

This section describes the JBoss Project Release procedure.

17.1. Tagging Standards

Tags on JBoss projects should consist of two parts - project identifier and version number. A list of existing modules can be found on the CVS Modules [http://fisheye.jboss.com/viewrep/JBoss/CVSROOT/modules] page. The version number must follow JBoss Versioning Conventions . A correctly tagged project would be JBoss_4_0_2, which is the tag for the JBoss Application Server, version 4.0.2. Note that all '.' from the version have been replaced with '_'.

17.2. JBoss Versioning Conventions

Product versions follow this format X.YY.ZZ.Q* (i.e. 1.2.3.GA, 1.2.3.CR1, 1.2.3.Alpha1-20060205091502)

- X signifies major version related to production release.
- YY signifies minor version with minor feature changes or additions (use of even numbers is preferred - 3.0, 3.2, 3.4, etc.).
- ZZ signifies patches and bug fixes. Minor features that do not introduce backward compatibility issues are ok.
- Q* is an alpha-numeric qualifier. The prefix of the qualifier needs to match the qualifier conventions listed below to ensure that versions can be compared consistently in terms of version ordering.

Major versions are usually developed in multiple iterations. Each iteration concludes with an intermediate version release. Intermediate versions are annotated with appropriate suffixes. This shows the progression of release versions. A given release may not have all stages of releases shown here.

17.2.1. Current Qualifier Conventions (Post 2006-03-01)

The following version conventions were put in place to have interop with eclipse/OSGI bundle version conventions.

1. X.Y.ZZ.Alpha# - An Alpha release includes all key features and is passing the main test cases. It still needs work on edge cases, bug fixes, performance tuning and other optimization tasks.
2. X.Y.ZZ.Beta# - A Beta release is the first release that the development and QA teams feel comfortable releasing for public testing. Some bug fixes and minor optimizations are expected, but no significant refactoring

should occur. No new major features are introduced from this phase on. Only stabilizing work.

3. X.Y.ZZ.CR# - Each candidate for release is a potential target for final release. Only if unexpected bugs are reported during the iteration timeframe the CR number is incremented (e.g. jboss-4.0.1.CR1 to jboss-4.0.1.CR2) and another release candidate is published. Generally only minor bug fixes are introduced to code and documentation.

4. X.Y.ZZ.GA - A Final version is released when there are no outstanding issues from the last CR version. Usually it's a matter of renaming the version from CR# to Final and repackaging the software.

5. X.Y.ZZ.SP# - A service pack release to a final release. A service pack may be made when there are significant issues found after a final release to provide a bug fix release before the next scheduled final release.

17.2.2. Practices

The standard qualifiers are the required prefixes. Additional information may be included in the qualifer as a suffix to incorprate information such as the build id to allow for distinction between nightly builds for example. If a given branch of a project is at 1.2.3.Beta1, the full version used could include a build id based on a GMT timestamp, the actual number of builds, etc. using a full qualifier syntax like Beta1-NNN where NNN is the numeric build id.

The key thing is that all version usage be consistent for a given project. A project cannot include a build id in the nightly builds, and then fail to include a build id of greater value when 1.2.3.Beta1 is actually released. The reason is that 1.2.3.Beta1 cannot be seen to be older than some previous 1.2.3.Beta1-NNN nightly build.

17.2.3. Legacy Qualifier Conventions (Pre 2006-03-01)

1. X.Y.ZZ.DR# - DR stands for Development Release. There could be a number of development releases. For example jboss-4.0.0DR1. A development release is a significant project milestone, but it does not implement all of the key features targeted for the public release.

2. X.Y.ZZ.Alpha - An Alpha release includes all key features and is passing the main test cases. It still needs work on edge cases, bug fixes, performance tuning and other optimization tasks.

3. X.Y.ZZ.Beta - A Beta release is the first release that the development and QA teams feel comfortable releasing for public testing. Some bug fixes and minor optimizations are expected, but no significant refactoring should occur. No new major features are introduced from this phase on. Only stabilizing work.

4. X.Y.ZZ.RC# - Each release candidate is a potential target for final release. Only if unexpected bugs are reported during the iteration timeframe the RC number is incremented (e.g. jboss-4.0.1RC1 to jboss-4.0.1RC2) and another release candidate is published. Generally only minor bug fixes are introduced to code and documentation.

5. X.Y.ZZ.Final - A Final version is released when there are no outstanding issues from the last RC version. Usually it's a matter of renaming the version from RC# to Final and repackaging the software (jboss-4.0.1).

6. X.Y.ZZ.SP# - A service pack release to a final release. A service pack may be made when there are significant issues found after a final release to provide a bug fix release before the next scheduled final release.

17.3. JBoss Naming Conventions

17.3.1. Naming of Build Artifacts

When creating jars as a result of a project's build, do not include the version element in the jar name. An example of that would be the current JBoss Messaging component of the Application Server - jbossmq.jar and not jbossmq-1.1.jar

17.3.2. Jar Manifest Headers

The standard Java version information and OSGI bundle version headers and their usage needs to be defined. The standard java jar manifest headers should include:

1. Manifest-Version: 1.0
2. Created-By: @java.vm.version@ (@java.vm.vendor@)
3. Specification-Title: @specification.title@
4. Specification-Version: @specification.version@
5. Specification-Vendor: @specification.vendor@
6. Implementation-Title: @implementation.title@
7. Implementation-URL: @implementation.url@
8. Implementation-Version: @implementation.version@
9. Implementation-Vendor: @implementation.vendor@
10. Implementation-Vendor-Id: @implementation.vendor.id@

where:

- Specification-Title: whatever name/standard name the jar implements
- Specification-Version: the version number
- Specification-Vendor: JBoss (http://www.jboss.org/)
- Implementation-Title: name with additional implementation details
- Implementation-URL: http://www.jboss.org/
- Implementation-Version: a full implementation version with addition build info. For example: ${version.major}.${version.minor}.${version.revision}.${version.tag} (build: CVSTag=${version.cvstag} date=${build.id})
- Implementation-Vendor: JBoss Inc.

- Implementation-Vendor-Id: http://www.jboss.org/

17.4. Pre-Release Checklist

A few tasks need to be completed before a project is handed off for release QA.

1. The files to be released should be tagged using the correct tagging convention, and the tags should match the appropriate version, refer to Tagging Standards
2. A roadmap which corresponds to the tag (eg. an RC1 release) should be present in JIRA, and each task in the roadmap must be marked off as completed
3. Product version should follow Versioning Conventions
4. The binary outputs for the project should be built and added to the repository
5. MD5 checksums should be generated for the binary outputs of the project
6. The testsuite should be able to run with a 100% success rate
7. Create a JBQA issue in JIRA for coordination with QA

Once all items on the Pre-Release Checklist have been completed, the project is ready for release testing.

17.5. QA Release Process

When a project is ready for a release and the Pre-Release Checklist has been completed, the QA team follows a standard release procedure outlined below.

1. 2 weeks prior to release the project team should open a JIRA issue in the JBoss QA project detailing what will be released, the date it is expected to be released on, and the CVS tag which will be used for the release
2. On release day the team will tag their project appropriately and enter a comment on the JIRA issue notifying QA that the project is now ready for the QA process.
3. QA team checks out the project and any dependent modules from cvs by the specified tag
4. QA team then builds the project using the target distr from the build script
5. QA team will then run the testsuite for the specific project and analyze their results - if any failures are present those issues need to be resolved by the QA or project teams before the release process could resume
6. QA team will verify documentation is present and correct
7. After all tests are passing, QA team will upload the disctribution archives
8. QA team makes a release on Sourceforge.net and a binary release to jboss-head

For more detailed release process on existing JBoss Projects, refer to JBoss QA Wiki

[http://wiki.jboss.org/wiki/Wiki.jsp?page=QualityAssurance] page

17.6. Release Notes

The Project Management System (JIRA) automatically generates Release Notes for a project. This is covered in Release Notes section of the JIRA chapter

18

Serialization

18.1. Performance Consideration - Use Externalization

The best way to achieve performance on Serialization, is to use Externalization without using writeObject.

Example 18.1. Externalization Code

```
public class FirstClass implements Externalizable
{
	SecondClass secondClass;
	public void writeExternal(ObjectOutputStream out)
	{
		secondClass.writeExternal(out);
	}
	public void readExternal(ObjectInputStream inp)
	{
		secondClass = new SecondClass();
		secondClass.readExternal(inp);
	}
}

class SecondClass implements Externalizable
{
	String str;
	public void writeExternal(ObjectOutputStream out) { out.writeUTF(str); }
	public void readExternal(ObjectInputStream inp) { str = inp.readUTF(); }
}
```

This is because writeObject will call a heavy discovery meta data for reflection and serialization's constructors. Dealing directly with the life cycle of objects on externalization routines (calling new) will save you from execution this meta code.

Of course there are situations where you have to use writeObject/readObject, specially if the written object was already described as part of other object (for solving circular references), but most of times when doing writeExternal routines you can guarantee the life cycle of objects.

18.2. Version Compatibility

A rule of thumbs is always define serialVersionUID.

Example 18.2. serialVersionUID

```
private static final long serialVersionUID = 39437495895819393L;
```

If you don't specify the uniqueID for the object, you can't guarantee version compatibility as minor changes could end up in different serialVersionUID, as they would be calculated according to rules specified on this URL:

http://java.sun.com/j2se/1.5.0/docs/guide/serialization/spec/class.html#4100

18.2.1. In Externalizable Objects

For an externalizable class, writeExternal and readExternal will have to control its version compatibility.

ObjectInputStream encapsulates Streaming in such way that if you try to read more fields from a readExternal method, you would get an EOFException. You could use the exception to determine if the end of a streaming was reached, like in the example

Example 18.3. writeExternal/readExternal among different versions

```
public void writeExternal(ObjectOutputStream out)
{
  out.writeUTF("FirstString");
  // code added in a newer version
  out.writeUTF("SecondString");
}

public void readExternal(ObjectInputStream inp)
{
  String str1 = inp.readUTF();
  try
  {
    String str2 = inp.readUTF();
  }
  catch (EOFException e)
  {
  }
}
```

On the example above if an older version was used to write the object an EOFException would happen and it

would be ignored. This would guarantee compatibility between different versions.

Any change made to an Externalizable class will be compatible as long as its read and writeExternal methods are compliant.

18.2.2. Regular Serialization

Serialization's specification describe lots of scenarios on exchanging information between different class versions:

http://java.sun.com/j2se/1.5.0/docs/guide/serialization/spec/version.html

Basically there is one simple and basic rules that will summarize the list above

- Add fields, don't delete them.

You need to take extra care when adding fields if the same Class is used back and forth different versions. For example a Class that is for communications on both sides.

18.2.3. Compatible and Incompatible Changes

The following URL lists all the possible situations where a class will and won't be compatible.

http://java.sun.com/j2se/1.5.0/docs/guide/serialization/spec/version.html

19

How to Update the Development Guide

This chapter discusses the process of updating the JBoss Development Process Guide.

The Process Guide is written using DocBook schema. To be able to keep it updated, a basic knowledge of DocBook is assumed. For reference and style manuals check the DocBook website [http://www.docbook.org/]

19.1. Checking Out The Guide As A Project

The Development Guide Project has two modules that need to be checked out separately.

- guide modules - checked out from private cvs
- guide build scripts (docbook-support module) - checked out from public cvs

To checkout the guide modules:

```
cvs -d:ext:username@cvs.jboss.com:/opt/cvs/private/development/management co -r guide
```

To checkout the build scripts:

```
cvs -d:pserver:anonymous@cvs.sourceforge.net:/cvsroot/jboss export -r HEAD docbook-support
```

The guide module includes the docBook content of the Development Guide - modules, stylesheets and images and the docbook-support module includes DocBook build scripts and is used to generate the html, single-html and pdf versions of the Development Guide. For any questions on accessing the cvs repository, refer to CVS Access .

19.2. Building The Modules

Building the modules is done through the guide's `build.xml` file, which is using targets defined in the docbok-suport module's `build.xml` . Make sure the following line in guide/build.xml is correct:

```
<import file="docbook-support/support.xml" />
```

The current version assumes the docbook module is in a directory called `docbook-support` inside your guide project folder. By executing the default target you will generate three different formats of the Guide - single-html, html and pdf, located in the build directory of your project. Currently the `master.xml` file specifies which modules of the guide are to be included in the build. If you add new files to the modules directory, you need to specify them in this file.

```
<! ENTITY qalab                    SYSTEM "modules/qalab.xml">
```

This line declares a module entity later to be added to the build list.

19.3. Request Development Guide Update

Any updates to the Development Guide need to be requested by creating a JBQA issue in JIRA.