

Jenkins-Koji Integration Plugin

Zásuvný modul integrující systémy
Koji a Jenkins

Bc. Václav Tunka

Master's thesis
2014



Tomas Bata University in Zlín
Faculty of Applied Informatics

Univerzita Tomáše Bati ve Zlíně

Fakulta aplikované informatiky

akademický rok: 2013/2014

ZADÁNÍ DIPLOMOVÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení: **Bc. Václav Tunka**
Osobní číslo: **A12667**
Studijní program: **N3902 Inženýrská informatika**
Studijní obor: **Informační technologie**
Forma studia: **kombinovaná**

Téma práce: **Zásuvný modul integrující systémy Koji a Jenkins**

Téma anglicky: **Jenkins - Koji Integration Plugin**

Zásady pro vypracování:

1. **Prostudujte možnosti systému Koji.**
2. **Analyzujte možnost použití Koji v systému Jenkins CI (Continuous Integration).**
3. **Navrhňte zásuvný modul integrující systémy Koji a Jenkins CI na základě požadavků vývojářských komunit obou systémů.**
4. **Publikujte plugin pod opensource licencí a získejte zpětnou vazbu od vývojářské komunity Jenkins CI a JBoss / Red Hat.**
5. **Shrňte a diskutujte výsledky práce.**

Rozsah diplomové práce:

Rozsah příloh:

Forma zpracování diplomové práce: **tištěná/elektronická**

Seznam odborné literatury:

1. SMART, John Ferguson. Jenkins: The definitive guide. Sebastopol, Calif: O'Reilly Media, 2011. ISBN 978-144-9305-352.
2. DUVALL, Paul M, Steve MATYAS a Andrew GLOVER. Continuous integration: Improving software quality and reducing risk. Upper Saddle River, NJ: Addison-Wesley, 2007, 283 s. ISBN 03-213-3638-0.
3. HUMBLE, Jez. Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation. Boston: Addison-Wesley, 2010. ISBN 978-0321601919.
4. BECK, Kent a Cynthia ANDRES. Extreme programming explained: embrace change. 2nd ed. Boston, MA: AddisonWesley, 2005, 189 s. ISBN 03-212-7865-8.
5. ROEBUCK, Kevin. Continuous Integration: High-impact Strategies - What You Need to Know: Definitions, Adoptions, Impact, Benefits, Maturity, Vendors. Ruislip: Tebbo, 2011. ISBN 978-1743044841.
6. Koji: RPM building and tracking system [online]. 2013 [cit. 2013-10-21]. Dostupné z: <https://fedorahosted.org/koji/wiki>
7. ROSHEN, Waseem. SOA-based enterprise integration: a step-by-step guide to services-based application integration. New York: McGraw-Hill, 2009, xix, 364 p. ISBN 00-716-0552-5.

Vedoucí diplomové práce:

Ing. Tomáš Dulík, Ph.D.

Ústav informatiky a umělé inteligence

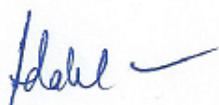
Datum zadání diplomové práce:

25. července 2014

Termín odevzdání diplomové práce:

26. srpna 2014

Ve Zlíně dne 31. července 2014



doc. Mgr. Milan Adámek, Ph.D.
děkan



doc. Mgr. Roman Jašek, Ph.D.
ředitel ústavu

ABSTRAKT

Práce pojednává o procesu průběžného doručení (Continuous Delivery) z pohledu vývoje zásuvného modulu pro prostředí Jenkins CI, který poskytuje integraci s prostředím Koji, sloužícím pro sestavování produkčních aplikací. Praktická část práce je publikována pod open-source MIT licencí a je volně dostupná ke stažení. Dále jsou popsány principy a architektura systému Koji, aby se vytvořil prostor pro popsání vývoje Jenkins-Koji zásuvného modulu.

Klíčová slova:

Průběžné doručování, Průběžné nasazení, Průběžná integrace, Jenkins, Koji, zásuvný modul, produkční sestavení, Maven.

ABSTRACT

This thesis describes the process of continuous delivery in terms of plugin development for Jenkins CI that provides integration with a clean-room build environment for production builds called Koji. The resulting software is published under the MIT open-source license and is available for download from the Jenkins CI community pages. Afterwards, release engineering principles and Koji architecture is described a necessary introduction for description of Jenkins-Koji plugin development.

Keywords:

Continuous Delivery, Continuous Deployment, Continuous integration, Jenkins, Koji, plugin, production builds, clean-room environment, release engineering, Maven.

ACKNOWLEDGEMENTS

I would like to thank my thesis supervisor Ing. Tomáš Dulík, PhD. for leading my thesis. Then my thanks belong to my colleague from Red Hat Salim Badakchani, who is one of the authors of Kojak, an environment that enables easy local installation of Koji. I would also like to thank Mike Bonnet, MSc., principal software engineer from Red Hat and one of two main developers of Koji, for insightful technical conversations regarding development. Last but not least, I would like to thank my colleague Mgr. Milan Navrátil for proofreading this text.

Acknowledgements, motto, and a declaration of honor stating that the print version of the Master's thesis and the electronic version of the thesis deposited in the IS/STAG system are identical, worded as follows:

I hereby declare that the print version of my Master's thesis and the electronic version of my thesis deposited in the IS/STAG system are identical.

THESIS AUTHOR STATEMENT

Prohlašuji, že

- beru na vědomí, že odevzdáním diplomové práce souhlasím se zveřejněním své práce podle zákona č. 111/1998 Sb. o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších právních předpisů, bez ohledu na výsledek obhajoby;
- beru na vědomí, že diplomová práce bude uložena v elektronické podobě v univerzitním informačním systému dostupná k prezenčnímu nahlédnutí, že jeden výtisk diplomové práce bude uložen v příruční knihovně Fakulty aplikované informatiky Univerzity Tomáše Bati ve Zlíně a jeden výtisk bude uložen u vedoucího práce;
- byl/a jsem seznámen/a s tím, že na moji diplomovou práci se plně vztahuje zákon č. 121/2000 Sb. o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) ve znění pozdějších právních předpisů, zejm. § 35 odst. 3;
- beru na vědomí, že podle § 60 odst. 1 autorského zákona má UTB ve Zlíně právo na uzavření licenční smlouvy o užití školního díla v rozsahu § 12 odst. 4 autorského zákona;
- beru na vědomí, že podle § 60 odst. 2 a 3 autorského zákona mohu užít své dílo – diplomovou práci nebo poskytnout licenci k jejímu využití jen s předchozím písemným souhlasem Univerzity Tomáše Bati ve Zlíně, která je oprávněna v takovém případě ode mne požadovat přiměřený příspěvek na úhradu nákladů, které byly Univerzitou Tomáše Bati ve Zlíně na vytvoření díla vynaloženy (až do jejich skutečné výše);
- beru na vědomí, že pokud bylo k vypracování diplomové práce využito softwaru poskytnutého Univerzitou Tomáše Bati ve Zlíně nebo jinými subjekty pouze ke studijním a výzkumným účelům (tedy pouze k nekomerčnímu využití), nelze výsledky diplomové práce využít ke komerčním účelům;
- beru na vědomí, že pokud je výstupem diplomové práce jakýkoliv softwarový produkt, považují se za součást práce rovněž i zdrojové kódy, popř. soubory, ze kterých se projekt skládá. Neodevzdání této součásti může být důvodem k neobhájení práce.

Prohlašuji,

- že jsem na diplomové práci pracoval samostatně a použitou literaturu jsem citoval. V případě publikace výsledků budu uveden jako spoluautor.
- že odevzdaná verze diplomové práce a verze elektronická nahraná do IS/STAG jsou totožné.

Ve Zlíně

.....
podpis diplomanta

TABLE OF CONTENTS

INTRODUCTION.....	9
I THEORETICAL PART.....	10
1 STATE OF THE ART.....	11
1.1 LATEST DEVELOPMENT IN JENKINS CI.....	11
1.1.1 Replacement of Winston servlet container.....	11
2 CONTINUOUS DELIVERY.....	13
2.1 WORK FLOW OF SOFTWARE CHANGES.....	13
2.2 PROBLEM STATEMENT.....	14
2.3 CONTINUOUS DELIVERY PIPELINE.....	15
2.4 API/ABI COMPATIBILITY.....	16
2.5 CONFIGURATION MANAGEMENT.....	17
2.5.1 Managing third-party dependencies.....	17
2.5.2 Managing components.....	17
2.5.3 Managing configuration.....	18
2.5.4 Managing environments.....	20
3 CONTINUOUS DEPLOYMENT.....	22
3.1 RELEASE AND DEPLOYMENT ANTI-PATTERNS.....	22
3.1.1 Manual release and deployment.....	22
3.1.2 Delaying deployment to production-like environment.....	24
3.1.3 Manual configuration management.....	26
3.2 INFRASTRUCTURE.....	27
3.2.1 Bare-metal deployment.....	27
3.2.2 Virtualized infrastructure.....	27
3.2.3 Cloud deployment.....	27
3.3 DEPLOYMENT TECHNIQUES FOR CLUSTERED ENVIRONMENT.....	28
3.3.1 Blue-Green deployment.....	28
3.3.2 Canary releasing.....	29
4 KOJI OVERVIEW.....	32
4.1 RELEASE ENGINEERING.....	32
4.1.1 Identifiability.....	32
4.1.2 Reproducibility.....	33
4.1.3 Consistency.....	33
4.2 KOJI, BREW AND MEAD.....	33
4.3 KOJI TERMINOLOGY.....	34
4.4 KOJI ARCHITECTURE.....	36
4.4.1 Koji-Hub.....	37
4.4.2 Kojid.....	37
4.4.3 Kojira.....	37
4.4.4 Koji web.....	38
4.4.5 Koji CLI.....	38
4.4.6 Koji XML-RPC client API.....	39

4.5 VARIOUS PLATFORM BUILDS.....	39
4.6 ARCHIVAL AND AUDITING.....	40
4.7 KOJI USAGE AT RED HAT.....	40
4.7.1 Layered products.....	41
4.7.2 Source builds and mass rebuilds.....	41
4.7.3 Stable vs. developer friendly.....	41
4.7.4 Middleware usage.....	42
5 KOJAK.....	44
II PRACTICAL PART.....	45
6 KOJI PLUGIN IMPLEMENTATION.....	46
6.1 REQUIREMENTS.....	46
6.2 ANALYSIS.....	46
6.2.1 Orchestration.....	46
6.2.2 Jenkins-Koji XML-RPC communication.....	47
6.2.3 Other design considerations.....	47
6.3 IMPLEMENTATION.....	48
6.3.1 Solving varying XML-RPC extension support.....	48
6.3.2 Type casting for Jenkins-Koji XML-RPC communication.....	49
6.3.3 OpenSSL and Kerberos in XML-RPC API.....	50
6.3.4 Kojak contributions.....	51
6.4 EXTENSION POINTS.....	51
6.5 BUILD PROCESS.....	52
7 KOJI PLUGIN USAGE.....	53
7.1 INSTALLATION.....	53
7.2 CONFIGURATION.....	54
7.3 PLUGIN USAGE.....	55
8 OPEN-SOURCE COMMUNITY FEEDBACK.....	56
8.1 JBOSS / RED HAT COMMUNITY.....	56
8.2 JENKINS COMMUNITY.....	57
8.3 FUTURE PLANS.....	57
CONCLUSION.....	58
BIBLIOGRAPHY.....	59
LIST OF ABBREVIATIONS.....	60
LIST OF FIGURES.....	62

INTRODUCTION

Nowadays, continuous integration (CI) is quite widely adopted practice helping developers to get instant feedback on changes introduced to a software build. Advanced practices such as continuous delivery and continuous deployment are finding its way into the mature software teams, helping teams to automate configuration management, release process and deployment. There is a lot of challenges in making continuous delivery and deployment easier and more manageable to help increase its adoption. Author's work is trying to connect world of continuous integration with the world of production-ready systems that have reproducibility and auditability in mind, ultimately leading to continuous delivery process.

The goal of this thesis is to develop a plugin for Jenkins enabling integration with the Koji production build environment. Whereas Jenkins CI serves for fast developer feedback and is mainly developer oriented [2], Koji is oriented towards delivering production builds using a clean-room secure environment with auditability and build reproducibility in mind.

So far there was no integration of Jenkins CI and Koji [6] and software engineers had to manually operate those two systems. Combination of these systems is used at Red Hat, but probably also by other software communities and companies around the world, so a plugin, providing integration of those two systems could benefit these communities. Koji used to be mainly RPM based environment, however it also supports Maven builds, which are going to be the main integration point for this thesis. Considering that clean-room production build environments are very expensive and usually proprietary, Jenkins-Koji integration could allow developers to easily leverage this OSS tool-chain free of charge with all advantages that it brings.

In this diploma thesis, author is following up and extending his bachelor thesis [1]. outcome of this is a completely new Jenkins plugin. The basic theory about continuous integration [8], refactoring [11], development process [10], etc. can be read in author's bachelor thesis [1]. Theoretical part of this diploma thesis is following advanced concepts like continuous delivery [3], continuous deployment and latest trends in these fields.

I. THEORETICAL PART

1 STATE OF THE ART

This chapter describes latest development in the field of continuous integration and follows up with advanced concepts such as continuous delivery and continuous deployment, which are related to author's practical contributions described in latter chapters.

Please refer to author's bachelor's thesis [1] for general concepts of continuous integration [5], refactoring, unit tests, versioning systems, various CI implementations like Jenkins [2], TeamCity, etc and for information about Jenkins plugin development [12].

1.1 Latest development in Jenkins CI

A lot has improved since the author wrote his bachelor thesis. There are improvements related to plugin development such as change of servlet container and other changes in Jenkins CI core [9], enablement of Ruby or Groovy based plugins, better security integration, improved cross-plugin integration etc., which are going to be described later in more detail.

1.1.1 Replacement of Winston servlet container

Jenkins CI was previously using custom servlet container called Winston [9], which was a custom container used only by Jenkins and which had significant associated maintenance and security cost. Later in 2013 an abstract layer was created, which is shielding Jenkins from direct communication with Winston. Most of the tasks performed by Winston were outsourced to Jetty, which is embeddable servlet container. Parallely an effort to integrate Jenkins with new servlet container called Undertow was progressing. Undertow was created to have high-performance, asynchronous IO and with high parallelization in mind. Undertow is developed by JBoss community / Red Hat and has been part of the Wildfly application server since Wildfly 8.0.0.Final release. It is not yet clear, if Undertow will be integrated fully inside Jenkins CI core, however there is a proof of concept available already.

Jenkins community takes backward compatibility for API/ABI and for runtimes very seriously. Jenkins is currently at version 1.570 and uses Java 6 language level / API, there is even a legacy Java 5 support, which is however done as best effort, with no guarantees.

Correct API/ABI compliance policy is to never do major changes such as runtime switch in minor or micro version. As Jenkins does not use micro versioning scheme, it means Java 6 is going to be supported till Jenkins 2.0, which will probably transfer to Java 7 or Java 8. Just to clarify, Jenkins of course runs on Java 7 or Java 8, however the language level and API usage is currently on Java 6 Standard Development Kit (SDK) levels.

Undertow recently moved to Java 7 SDK, abandoning the Java 6 SDK backward compatibility. This is not optimal for the Jenkins community, so discussion are being led at the moment how to provide Java 6 SDK support for Undertow, so it can be tightly integrated into Jenkins.

If developers do not use new API from Java 7 SDK, but just the new language constructs, there is a way to retro-translate the bytecode to Java 6 SDK language level. However Undertow might have used new API, so either a workaround will have to be found, or Jenkins community need to wait till transition to Java 7 SDK language level to adopt Undertow as embedded servlet container.

2 CONTINUOUS DELIVERY

Continuous delivery is a method of doing small incremental releases in frequent intervals instead of larger monolithic releases, that happen after longer time intervals and have increased migration and maintenance cost [3]. As continuous integration helps developers to integrate with every small change and have instant feedback, continuous delivery helps to solve similar problem on the level of delivery of individual software features in release. This way both software vendor and customers adapt to rapid cycle, that delivers targeted functionality. Customers then have up-to-date software with frequent bug fixes and security fixes. Continuous integration is a prerequisite to continuous delivery.

2.1 Work flow of software changes

Typically a software feature follows a similar work flow, like the one outlined below:

1. Feature specification
2. Analysis
3. Implementation
4. Testing
5. Release
6. Delivery
7. Maintenance
8. End of Life

However a lot of tasks performed by the development team, release engineering team or quality assurance team are implicit and undefined. For example after source code is stored in versioning system, and testing is complete, the releasing process itself can be fairly complex with a lot of variables. Deployment to a production instance can involve manual actions and can sometimes be very tedious. Configuration or data can affect the running application significantly and can be out of the scope of testing. Continuous delivery tries to formalize and automate this whole process, leading to more predictable process while eliminating the problematic manual parts.

2.2 Problem statement

For complex modularized software releases, features are delivered in a bulk release together with other features and bug fixes. From customer point of view, it is important if the feature developed for a software product fulfills all the requirements and if so, when can customer start using it. For this reason it is important to have lean delivery and deployment of individual features in smaller releases, so software can be deployed and customized quickly for customers, leading to immediate benefit.

In cases, where software is missing some functionality required by customer, we need to get back to point 1 – Specification. Usually, a software feature is waiting for other features or bug fixes to be delivered. In Illustration 1, time from specification to release and delivery is visualized and in the example case, a feature can wait for months even if it is completed and tested, just for the delivery mechanism to be picked out for a release. Continuous delivery aims to fix this problem by introducing continuous delivery pipeline described in latter chapters.

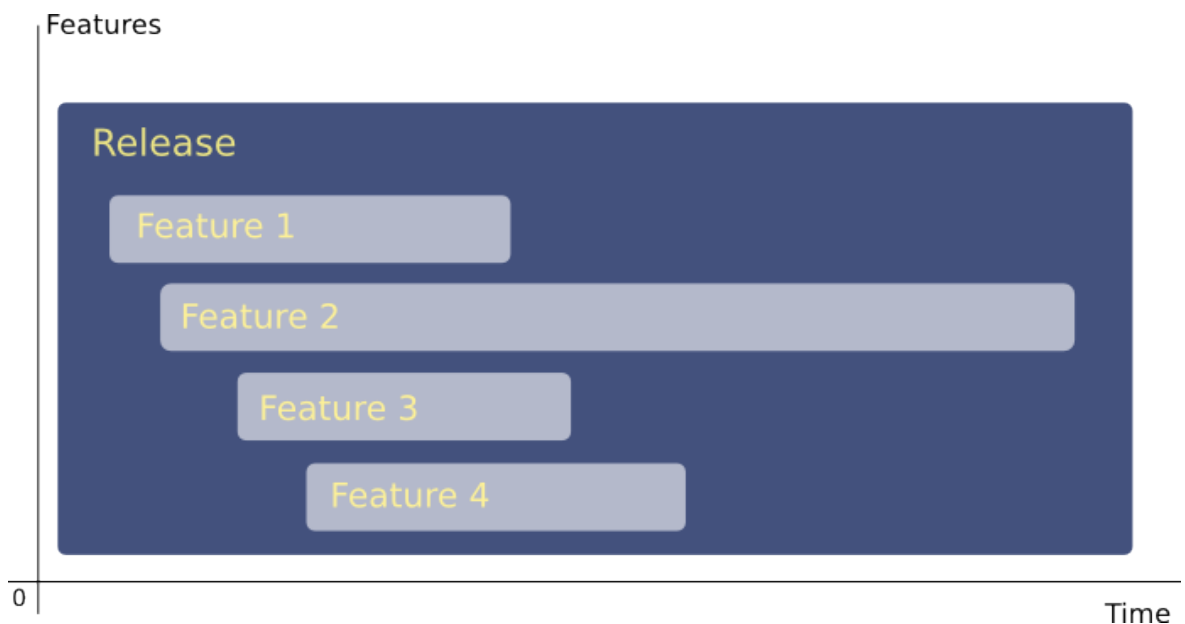


Illustration 1: Release dependency on features

2.3 Continuous delivery pipeline

As you can see in Illustration 2, each change goes through version control and from there to series of cascading checks ranging from unit tests, through smoke tests and acceptance tests. Each of these phases are more complex and time consuming, the latter stages can involve manual actions although it is recommended to automate as much as possible [3]. Red color indicates failed phase, green color indicates passing phase in the diagram.

You can see that first check was faulty and triggered unit test failure. After another check in, unit tests passed, but more complex acceptance tests failed. This lead to third iteration, where developers fixed issues in the code causing failing acceptance tests and in this iteration, the change progressed all the way to user acceptance tests after which release was performed.

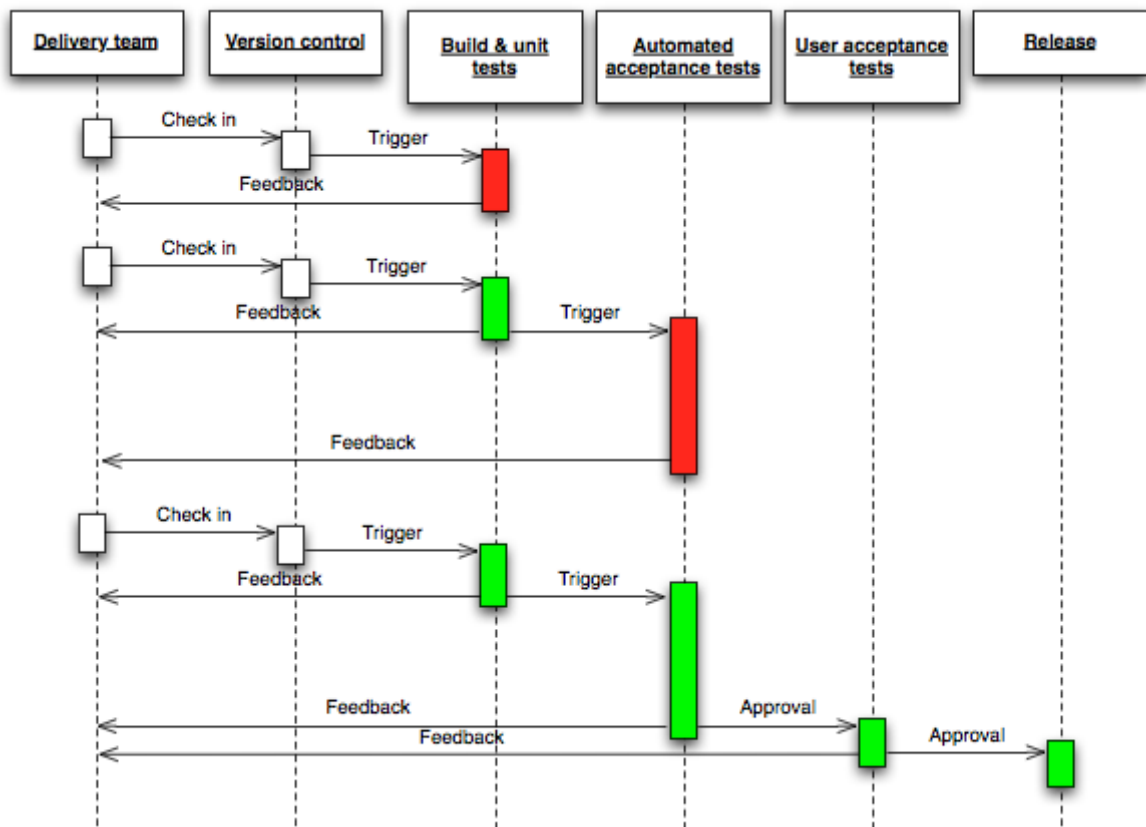


Illustration 2: Continous Delivery pipeline – cited from [3]

Each change in source code, configuration, database or environment triggers a creation of individual delivery pipeline [3]. First steps of the pipeline involve compilation to binary form or bundling into an installers. All the other steps involve various tests, each giving the development team more confidence, that this specific combination of code, environment, data and configuration works and can be released.

Continuous delivery pipeline has following goals:

- Increase collaboration – by making each part of the process (build, testing, release) visible to everyone.
- Immediate feedback – problems are identified as soon as possible in the cycle.
- Automated releases – deployment and releasing is automated process, so it becomes possible to do it early and often.

2.4 API/ABI compatibility

Mature software companies and software distributions, such as various Linux distributions, have standards for API/ABI compatibility. The industry standard for version scheme is – X.Y.Z:

- X – Major version
- Y – Minor version
- Z – Micro version

Complex changes, module rewrites, additions of large parts of completely new functionality can only happen in major software release. It is allowed to discontinue API/ABI compatibility for major software releases, however this should have reasonable explanation, such as End of Live (EOL) of a runtime, such as Java 6 SDK EOL.

Minor software release is used for larger software bug-fixes or for smaller feature sets, which are targeted for a baseline major version of the given software product. Typically the release is stable and maintains API/ABI compatibility. Only case, when the compatibility can be broken in minor release is a security fix, which fixes a known or embargoed vulnerability.

Micro software releases only serve as a vehicle to deliver security fixes and bug fixes to customers. It can never contain a new feature or break existing API/ABI or any of the functionalities, as this would be considered a regression. Again the only justifiable case for an exception can be a security fix, that fixes known or embargoed functionality.

2.5 Configuration management

Version control best practices were described in previous author's work [1]. This chapter is going to concentrate on dependency management, software configuration management and environment management.

2.5.1 Managing third-party dependencies

Most complex applications will depend on third-party libraries, which are usually not updated very frequently by the development team. For compiled programming languages, dependencies come typically in a binary form, for interpreted programming languages, dependencies come as source code modules. Either way, these libraries will be installed locally on the file system using integrated package management system such as RPM, or will be managed by special package management tools for the given programming language, such as Maven, Ruby gems or Node.JS packaged modules.

It is a good practice to set up a local package repository inside the organization, which will contain approved artifacts, that can be used by the development team. This is not only good for auditing and compliance reasons, but also for faster start of new team members. Also if anything changes in the outside world, local maven repository will still contain artifacts needed for build reproducibility of developed software [3]. To achieve greater reproducibility it is recommended to specify exact versions of dependencies to be used (some tools allow automated resolving without specifying versions). This practice can prevent spending hours of debugging caused by people using different versions of the same library.

2.5.2 Managing components

For modularized applications, each module can be developed by a different team in the same organization, however with individual release schedule, so the treatment is similar to third-party libraries. Every larger application should be split into components. This support

re-usability of components, makes testing of software changes easier as they happen on smaller scope and helps to locate and limit regressions.

At the beginning of the development phase, the build will typically be performed as a whole during single execution, running the tests during the build as well.

After monolithic build starts to become unmanageable, it is a good practice to start splitting component builds and setting binary dependencies. Binary dependencies are better than source dependencies, because they are compiled only once with same set of libraries and developers can be sure they are not using unsupported configuration. Even if all the other variables do not change, there might be different version of compiler or different version of the OS, leading to different binaries, which could be cause of potential issues.

Using binary dependencies has a small disadvantage manifesting in increased difficulty when tracking down which commit caused a regression or change in behavior, however good CI server can help with this.

2.5.3 Managing configuration

Configuration can significantly affect software behavior, some authors argue, that configuration is on similar level like source code and should be treated as such [3]. One of the hardest tasks when managing configuration can be consistent design of configuration options and consistent configuration behavior in different parts of the system.

Configuration can affect software in following stages:

1. Build time – build scripts incorporate configuration from various locations into the binary at build time, or affecting binary creation during build time.
2. Packaging time – configuration can be inserted into packaged software during various assemblies, composes, or installer creation.
3. Deploy time – deployment scripts or installers can use various configuration files, or ask users for input during deploy time.
4. Run time – application can receive configuration at start up or when fully running.

A lot of customers want high configuration flexibility for the developed software. However the higher configuration flexibility will be, the more difficult it will become to process

configuration in the application, in extreme cases leading to combinatoric explosion. Changes to source code are verified by compiler, static analysis, unit tests and more complex tests scenarios. However configuration changes are more difficult to test and usually the logic for validating inputs is not verified with the same rigorosity as the source code is. It is common, that configuration handling can reveal leaking abstraction pattern to end user. For example when user tries to configure e.g. backup location setting and receives exception message from class managing paths on a filesystem, which complains about wrong delimiter of file path.

Configuration should be managed consistently and go through proper integration, deployment and testing processes like source code does. Deploying to production-like environment during testing and deployment smoke tests can help to limit potential issues with configuration.

If build-time or packaging-time configuration has effect on deployed application or its runtime, it is generally a bad practice to embed it during the first two phases. It should be possible for the application to be fully configured during deployment and later stages without restriction from the previous phases. This is especially true for database environment, authentication, backups, and other services.

Principles for configuration management [3]:

- Carefully decide where to put configuration settings – at build-time, packaging-time, deploy-time, or run-time. Feedback from operations team or from customers can be very valuable.
- Configuration options should be stored in the same source code management repository as application source code. Configuration values should be stored at different place however, as they have distinct life-cycle. Values like passwords or sensitive information should not be stored at version control at all, various Single Sign On (SSO), Kerberos, or other solutions should be used instead.
- Automated script should be used to perform configuration stored in a configuration repository. This enables auditing and helps to immediately find out, which environment is using which configuration.

- Scripts should support multiple versions of application (which can have different configuration options), and deploy to different environments. This way, developers and operations can easily see, what options are available for particular version of the application.
- Descriptive names should be used for configuration names. It is recommended to try and configure the system without a manual, using just the configuration names as a reference. This is a good test if the names are clear enough.
- Configuration should be encapsulated, so there are not unexpected side effects on completely unrelated part of the system. Good configuration tests can verify this.
- Configuration system and configuration options should be as simple as possible. Developers should avoid creating configuration options, unless there is a requirement for it, or unless the missing configuration option would seriously affect the system.
- Configuration options should be tested at multiple stages to ensure it works as it should. There should be tests, checking presence of external tools, which the application depends on.

2.5.4 Managing environments

Environment is meant as a combination of operating system, software, hardware and external infrastructure. Application environment has similar importance as application configuration. If for example application requires an LDAP server and will not be able to properly function without it, it must be configured as part of environment configuration. Similarly number of open TCP connections, or other OS controlled values must be taken care of as part of environment configuration, if the developed application depends on these prerequisites [3].

If environmental configuration is managed manually, e.g. the system administrator edits multiple configuration files across each system, it is often difficult to identify source of regression, as there is no track record that would help the administrator identify problematic changes. It is also difficult to roll back to last working configuration.

Symptoms of problematic environment management:

- Small change can affect application performance or lead to issues.
- It is difficult to replicate production environment, as nobody knows the actual state of live environments.
- Manually maintaining multiple nodes in a cluster become difficult and differences between node configuration arise.

It is advised to use tools such as Puppet or CfEngine for managing your environments. These tools have agents installed on the managed systems and the agents are periodically pulling the changes from version control systems to ensure system configuration is up to date. Advantage of these tools is, that the whole process becomes auditable, so you can easily find out who did the changes, which machines are affected and what happened to them.

It is generally a good practice to use virtual machines, as the VM can be easily cloned, without having to run extensive scripts to prepare a fresh system.

3 CONTINUOUS DEPLOYMENT

Whereas continuous delivery solves only the shipment of a software release to customers and leaves the choice on how to deploy to customers, continuous deployment goes one step further. Continuous deployment means, that each change that goes through continuous delivery pipeline is deployed to production. Whereas with continuous delivery, not each change needs to be deployed and what is deployed is typically customer decision, continuous deployment releases every successful change to production each time, which is typically multiple times a day. Continuous deployment solves the additional steps as well such as maintenance of the infrastructure, deployment on staging and production environments, database migrations, etc. In this chapter, these parts of continuous deployment will be described in more detail to provide reader with complete description of the whole process, its advantages and requirements.

3.1 Release and deployment anti-patterns

This chapter describes common anti-patterns that are slowing, complicating or otherwise damaging the release process. Anti-patterns such as manual release and deployment, delayed deployment to production environment or manual configuration management are described in this chapter.

3.1.1 Manual release and deployment

Software release and deployment is a process that consists of working with many moving parts. A lot of software companies still release manually, treating individual tasks as atomic and isolated. In the end, this has effects on order and timing of individual tasks leading to unpredictable elements in the software release process [3].

Signs, that release process is not properly automated:

- Teams rely on manual testing to verify software is running correctly.
- Many changes happen to the release process itself while release is in progress.
- There is extensive documentation of release steps with many conditionals and description of what do do if a step fails.

- There is different configuration for clustered environments e.g. different layouts of file system, different DB settings.
- Releases are unpredictable and often have to be rolled back.

Automation is crucial in the release and deployment phase, however there are some tasks where human intervention is acceptable – choosing the environment and starting the deployment process.

- If the deployment process is not automated, the order of deployment tasks is not repeatable and the whole process can be unreliable.
- There is a bigger room for errors to happen during the manual deployment phase. Automated deployment will typically try to minimize room for error.
- Manual deployment requires maintaining of extensive documentation, involving people from various teams. Maintaining documentation for this process will likely be very time consuming and the documentation is likely to be out of date. Once deployment is written down as executable script, it must always be up to date, so with reasonable comments, it can be considered executable documentation.
- Automated deployment sets clear boundaries for collaboration, encouraging creation of clearly interfaces for machine communication.
- Manual release and deployment process requires person with expert knowledge, when this person is not available, releases can get delayed.
- Although the manual deployment usually requires expert knowledge, it is a repetitive tasks which is not creative. Once deployment is automated, engineers can start working on more creative tasks, instead of donating their time to manual deployment.
- Until manual deployment process is started, there is usually no way how to test it works. Automated deployments have embedded checks and can easily test the prerequisites.
- When process is manual, it is not auditable, as there is no guarantee, that the documentation was followed. Once the process becomes automated, audit becomes easy, as there is a log will all the activities, that happened as part of the process.

Once deployment process is automated, it must be the only canonical way how to deploy. It is no longer possible to do deployment multiple ways, as this would go around the deployment scripts. When deployment becomes more frequent, it is a good smoke test, that all the scripts work properly.

Scripts should be designed in a way, that deployment to multiple environments is possible. This way, future deployment to production environment gets tested many times, before it actually happens. If error occurs in the production environment, engineer doing the production release can be certain this occurred because of an error in the configuration of production environment and not in the deployment scripts or in the software itself.

3.1.2 Delaying deployment to production-like environment

This anti-pattern describes situation, where development teams waits with the deployment to production-like environment till the development is complete [3].

Definition of this anti-pattern:

- QE team tested on development machines, if they were involved at all in these stages of the process.
- Engineering team assembles the software package, installers, configuration and database migrations and passes it to different team doing operations. The software was never deployed to staging or production-like environment before this step.
- System administrators doing the operations see the release for the first time, when they do deploy on staging or on production.
- Production-like environment is absent, as it is under strict lock-down, it is expensive to have one, or it is not present for other reasons.
- Little collaboration between the engineering team and the operations team.

In large organizations the operation teams can contain various experts, such as database experts, administrators taking care of application server, someone else maintains load-balancers and network configuration. This way the deployment process is divided between multiple roles, which lead to different errors being solved by different people, who do not have the full picture. Unexpected issues can be encountered by the deployment team, as they are the first people deploying to staging environment. Development team then has

to patch a lot of issues during deployment – these patches are created in rush and can lower the quality of the software as a whole.

When development teams test only on development machines, clustering and environment issues can appear during the deployment to staging. Clustering issues are usually quite hard to debug and to fix, however there is usually a pressure to deploy and release as quickly as possible, leading to incomplete fixes, which can become problematic later on. Generally, a lot of issues is found during the deployment to staging environment, as the team did not perform deploys before, but there is not enough time to fix all those issues so late in the cycle. Therefore, software with inadequate quality gets released.

- The development team has incorrect assumptions before the deployment occurs. The longer the release process lasts, the longer the team has incorrect assumptions and the longer it can take to fix those.
- If the application is developed from scratch, first deployment to staging environment is usually the most problematic one.
- In large organizations, where deployment is divided between multiple roles such as database administrator, system administrator, networking personnel, etc. the communication overhead can be quite intensive. A lot of various tickets or emails need to be send, these usually have inter-dependencies and it becomes difficult to orientate in the current state.
- Developer assumptions are less realistic as the difference between staging and production machines increases. E.g. it can be problematic, when developers use workstations with Mac OS X and in production they deploy to Linux or Solaris cluster.
- Once a platform is developed, the development team does not have control of various environments customers will deploy to. This involves a great deal of testing to be performed by quality engineering teams to certify various platform combinations.

The whole delivery team encompassing developers, testers, operations, release engineering should be working together from the start of the project. Activities such as testing, releasing and deployment should be part of the regular development process, so they are performed

over and over before the final release. The testing environment should become more production-like as the software change goes through continuous delivery pipeline.

3.1.3 Manual configuration management

Large organizations typically have engineering operations team, which performs changes on production systems. If a change is needed in production, developers typically create a ticket that describes desired change such as increase of thread pool on application servers, or modification to database connections. Such change is then performed manually on a server and logged in the ticket, wiki or change management database in better cases [3].

Signs of this anti-pattern:

- Differences on cluster nodes – different nodes serve different amount of load, etc.
- Although multiple deployments on staging work, production deployment is problematic.
- It takes long to prepare production environment by the operations team.
- Rollback to previous configuration, OS, application server, database, runtimes is difficult and operations are reluctant to perform it.
- Different cluster nodes do not have the same OS version, same environment or runtimes.
- Manual configuration of the production systems by operations.

Changes to staging or production environment should be applied using version control systems. Configuration management solves creation of environment including operating system, runtimes, configuration and other services in automated and repeatable manner. Each change done to production environment must be tracked and recorded in version control system, if someone does manual change to production system in the meantime, scripts become out of sync, thus the only correct way how to alter the production environment should be in automated manner. Essentially it is important to follow the same continuous delivery pipeline not only for code changes, but for all the other configuration or environment changes as well.

3.2 Infrastructure

This chapter describes various infrastructure options for deployment, from bare-metal deployment, through virtualized environment to cloud based deployment.

3.2.1 Bare-metal deployment

Historically most of the infrastructure deployed at customer side was self-hosted and self-managed for most of the installations. Customer usually purchased a machine and installed operating system directly without virtualization, taking care of the maintenance and administration themselves. Only the largest customers could afford redundancy and clustering, so in case of software or hardware failure there was a risk of services not being available.

3.2.2 Virtualized infrastructure

During the last decade a lot of customers migrated to various virtualization solutions, either providing dual-stack, e.g. memory and process management on top of the host system, or coming with a hypervisor, enabling usage of host memory and process management. In case of failure on the VM level, customers could revert VM to previous snapshot, easily provision a new VM, or solve this otherwise with better tools compared to bare-metal deployment. However when issue emerged on HW level of the host systems, customers lost availability of critical systems till they migrated the VMs on a new host.

3.2.3 Cloud deployment

A development of last years was to migrate to cloud solutions. The main difference and advantage to previous infrastructure solutions was virtually no delays in case of failure, as the whole process of maintaining virtual images, maintaining the core infrastructure, storage and backups was outsourced to additional software layer that is able to provide fail-over and clustering by default.

Types of cloud solutions:

- IaaS – Infrastructure as a Service
- PaaS – Platform as a Service
- SaaS – Software as a Services

Infrastructure as a Service provides customer with a complete control over the system, so customer is able to install OS, various tools and automatically receives root level access to the system. However customer is responsible for managing the system as is. Cloud provider provides a new system in case of failure of existing systems or in case of increased demand. Example of IaaS clouds is Amazon EC2, Microsoft Azure, etc.

Platform as a Service provides customers with tools and runtimes, that are already pre-configured and ready for deployment of customer's application. In case there is issue on the application level, customers are expected to solve it themselves. The cloud provider offers elasticity of the cloud, so if customer application needs to span across more nodes in the cluster, this can be achieved. Example of PaaS cloud is Openshift, Heroku, etc.

Software as a Service provides customers with deployed application, that can be configured using web UI and is ready for usage afterwards. Cloud provider has responsibility to ensure the application hosting will be available for customers. Example of SaaS cloud is Salesforce, Facebook, Google services, etc.

3.3 Deployment techniques for clustered environment

When deploying to clustered environment, it is necessary to have options for rollback and to ensure transactions are not going to get lost during deployment of a new version. There are various techniques how to achieve these goal, which are going to be described in this chapter.

3.3.1 Blue-Green deployment

In Blue-Green deployment, there are two production environments, which are similar as much as possible, but separated from each other. Each environment has its web server, an application server and a database. At a given time, only one of those environments is live for users. A router or load-balancer is used to determine, which environment serves users currently [3].

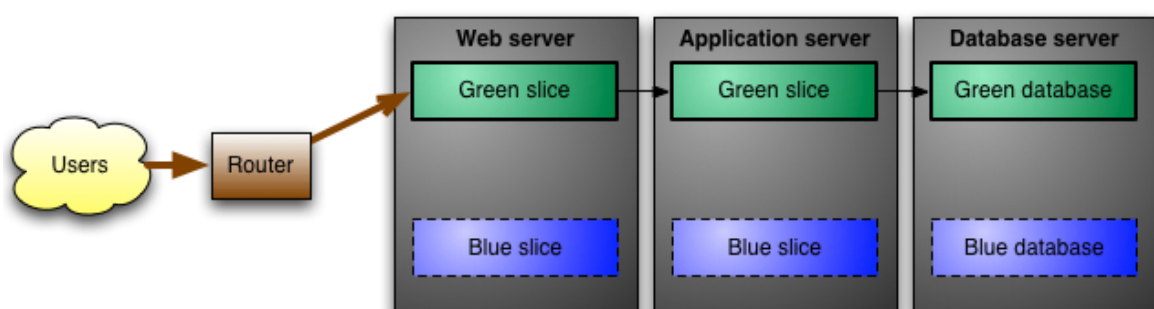


Illustration 3: Blue-Green deployment, cited from [3]

In this scenario, green deployment is live and the development team is preparing a new release in the blue environment. After the release is completed and tested, the production environment is switched to blue.

Development team can decide to do a rollback quite quickly, blue-green deployment provides easy way how to rollback just by switching the router to the previous environment. During the switch, development team must make sure, no transactions are lost. It is possible to put transactions on both environments, where one is serving as a backup, or it is possible to set one of the environments read-only, before the transition and then set it back to read-write.

The two environments need to be separated, but as similar as possible in configuration. They might use the same type of hardware, or be identical virtual machines. Both environments can also run on the same operating system and just be separated by zones, using different IP addresses.

Advantage of this deployment technique is, that the development team is cycling between these two environments at each release, keeping both environments production-ready. Blue serving as a final staging area, while green deployment is live and then vice versa, after blue become the live environment. This way there is always possibility for rollback and teams have working solution for disaster recovery.

3.3.2 Canary releasing

Canary releasing is a technique similar to blue-green deployment. Before rolling out the change for all users, a smaller subset of users receives the change first. The subset of users who will be routed to the new deployment can be random, or can be chosen per geography. Advanced method of choosing the set of users involve profile and other demographics [3].

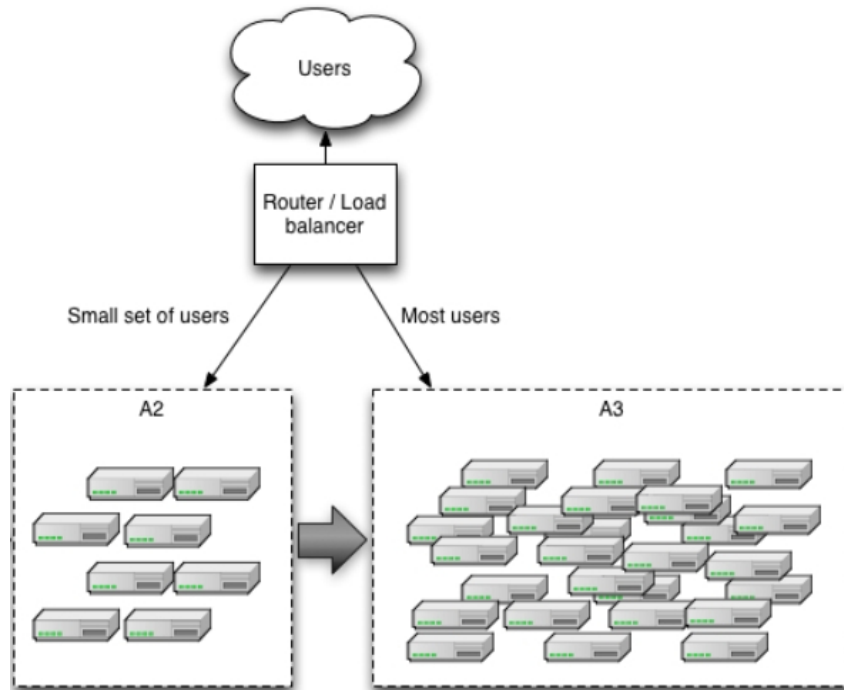


Illustration 4: Canary releasing, cited from [3]

Similarly to the blue-green deployment, both infrastructures are running in parallel, until all users are migrated to the new infrastructure. Advantage to blue-green deployment is that only a part of users are testing the new features, so the whole user base is not impacted by potential issue.

Another advantage of canary releasing is capacity testing. The development team can increase the number of users routed to the new deployment and perform load testing at the same time. If issue occurs, users can be rolled back to the old deployment, giving the development team enough time to fix the performance issues.

Disadvantage of canary releases is, that the development team has to maintain multiple versions of the software until all users are migrated to a single version. Given releases happen often, the state when development teams maintain multiple releases is almost permanent. It is best to limit number of concurrently deployed features at minimum, so there are not more than two concurrently maintained environments.

4 KOJI OVERVIEW

This chapter provides an overview of Koji build system [6], starting with an introduction to release engineering practices that heavily influenced the development of Koji, later describing Koji terminology and its various extensions and implementations such as Brew / Multiplatform Enablement And Delivery (MEAD).

Koji is an open source build system written mostly in Python, designed for clean-room production-ready software builds that ships to multiple platforms and distribution formats. Koji was primarily developed to be a build system for the Fedora project, however it is used in many other organizations as well¹, for example in Red Hat, Amazon, TomTom, Scientific linux, Caltech or in CERN [6].

4.1 Release engineering

Release engineering is a discipline in software engineering that deals with build, assembly, maintenance, and delivery of a software product [4]. Release engineering as a practice is crucial especially for complex products assembled from many subsystems, that are developed by various development teams across the organization using wide variety of different software tools and practices. Generally, the practices are suitable for all development teams regardless of their size.

Release engineering has three main axes:

1. Identifiability
2. Reproducibility
3. Consistency

4.1.1 Identifiability

Identifiability is an ability to identify all the source codes, tools, components and other parts that compose a release. In a complex system, various parts might be stored in different SCM repositories (Git, Mercurial, SVN, etc.), they might be using different build tools (Maven, Gradle, Ant, Make) and mainly they can contain a larger number of 3rd party dependencies

¹ <https://fedoraproject.org/wiki/Koji/RunsHere>

that might be out of control of the development team. All of these tools and variables might change over time.

4.1.2 Reproducibility

Reproducibility means that the team needs to be able to reproduce the same binaries or binary compatible outputs using the input sources throughout a larger period of time. For example, a customer is running an older version of a software product that was released 5 years ago. For some development teams, it might be challenging even to find the source codes for that kind of a historic release. Next step is to ensure that all the tools used to build and release the software product are available and ready for use. For example some of the tools might have already discontinued development, are in the EOL (End of Life) phase or might not work with today's hardware or software. Ideally, the release engineering team should have a system that contains snapshots of all the required versions of tools in tool-chains that are associated with the given software release.

4.1.3 Consistency

Consistency has to do mainly with the process-oriented side of release engineering and is dependent on the two previous axes. The main goal of this axis is to provide an environment for developers that enables reproducibility, enforces clear rules for production builds, enables delivery of software patches and is auditable. For auditability, it is important to know what chunks of changes were distributed with the given release version, who built the various parts of release, when it happened and what tools were used. This also greatly helps software security.

4.2 Koji, Brew and MEAD

Koji is an open-source build system designed with proper release engineering practices in mind. The term *build system* is used in a wider meaning in this context, standing for a robust environment for multi-platform builds which enable the usage of various builds tools and allows reproducibility, modularity, and isolation of individual builds.

Koji is an upstream project for Red Hat build system called Brew. Upstream is a common term used for an open-source project that is being a basis for a commercially used or supported product. We can say that Brew is Koji with a few extensions that are described

later. In the beginning, Brew was focused on the standard RPM builds however, later on the support for native Java builds in Maven was developed. This Brew/Koji extension enabling Maven support is called MEAD. Brew also enables builds for the Windows platform, or builds for Solaris platform for example. Generally, Koji/Brew developers try to push everything upstream first to enable community testing and usage, however, the Fedora community being the primary user, is somehow selective in what gets accepted. For example the MEAD Maven extension was for long time not part of the base Koji installation.

Please assume that Koji and Brew are generally very similar, nearly identical terms and when Koji is mentioned, Brew is also referenced, or the other way around in the scope of this thesis.

4.3 Koji terminology

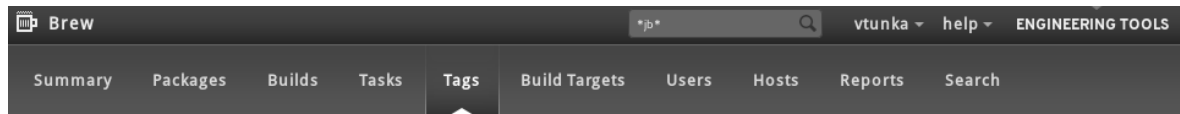
Koji primarily deals with packages – packages have fields such as a name, description and some other meta-data. Each package can contain builds, which can be tagged to multiple packages at once. Each build is built in its own isolated environment called build-root. A build-root is set up individually for each build, and all the required tools and dependencies are installed from scratch. This clean room environment ensures that all the contents are buildable and that the remains of older builds are not accidentally shipped to customers. Please see Illustration 5: Koji terminology for a graphical visualization.



Illustration 5: Koji terminology

Koji also allows for an advanced package organization to something called build tags. You can imagine build tags as sets of builds that have dependencies between each other. Also anything that is not in a build tag cannot be used as a build dependency. The build tags also support separating packages used just as build time dependencies, but not being shipped in the final software product, from packages that are run-time dependencies which are integral part of the shipped software product. Koji allows for inheritance, white listing,

and black listing between the build tags, which ensures a very fine-grained control over the builds. See Illustration 6: Brew/Koji Build Tag Inheritance for details.



Information for tag [dist-mead-jboss-build](#)

Name	dist-mead-jboss-build
ID	1648
Arches	i386
Locked	no
Permission	admin
Maven Support?	yes
Include All Maven Builds?	no
Inheritance	<ul style="list-style-type: none"> — dist-mead-jboss-build — dist-mead-jboss — mead-import-jboss — mead-import-maven — mead-import-custom

Illustration 6: Brew/Koji Build Tag Inheritance

4.4 Koji architecture

Koji has a centralized architecture with one master server that is acting as hub which is distributing the load across various build hosts, controls the communication, and stores the resulting artifacts. The difference between Jenkins and Koji is that Koji is provisioning clean bare metal or virtual machines each time a build starts, whereas Jenkins is using existing systems.

Koji has multiple clients that access the information stored in Koji and can control the given instance. These are Koji Web interface, Koji CLI and any other clients accessing XML-RPC Koji API.

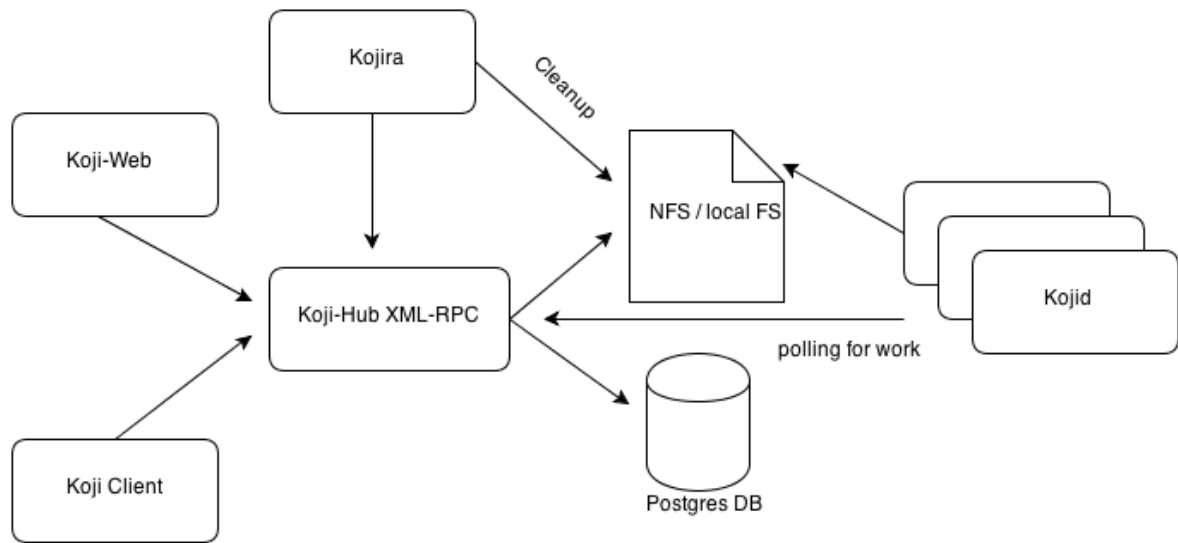


Illustration 7: Koji Architecture

4.4.1 Koji-Hub

Koji-Hub is a XML-RPC server running in the center of Koji operations. It is running `mod_wsgi` inside `apache httpd`. It is a passive component awaiting XML-RPC communication initiated by other Koji components.

Koji-Hub is the only component of Koji to have direct access to PostgreSQL database and one of the two components that can write to the file system [7].

4.4.2 Kojid

The daemon running on the build executors is called `Kojid` [7]. `Kojid` is polling `Koji-Hub` for work and then determines which kind of build is to be executed e.g. MEAD Maven build, RPM build, build on specific platform, etc., and initializes the build host using `Mock` tool, which is described later. Each time the build hosts installs all the packages including the kernel from scratch, which creates a fresh system with clean-room environment.

4.4.3 Kojira

`Kojira` is a daemon responsible for keeping the information about build-roots up to date, it also performs garbage collection of old build-roots and cleanup after build request completes [7].

4.4.4 Koji web

The Koji website displays a graphical interface with all the information about packages, builds, tasks, build-roots, artifacts, users, build tags, targets, etc.

The Koji website is intended to act more as a reference than remaining clients, for example, the developer cannot submit a build using the Koji web, tag builds into various packages, or do change operations.

Technically, Koji website is a set of scripts running `mod_python` providing web interface using Cheetah templating engine [7].

4.4.5 Koji CLI

Koji Command Line Interface (CLI) is a simple console client, that allows for gathering information from Koji, starts the remote builds, monitors progress, and manages Koji in all the possible ways. It even can even show build inheritance and other graph listings using the console semi-graphics.

```
[vtunka@voidray jenkins-koji-plugin]$ brew latest-build jb-eap-6-rhel-6-build resteasy
Build                               Tag                               Built by
-----
resteasy-2.3.8-4.Final_redhat_3.1.ep6_e16  jb-eap-6-rhel-6-candidate  mead-scheduler
```

Illustration 8: Koji CLI

Koji CLI is using Kerberos authentication and unless the user has a valid Kerberos ticket or OpenSSL key, he cannot do any change operations or start a build, but only has a read-only access.

Koji CLI enables the user to:

- Build one or more packages from source .
- Print basic information about a build.
- Print the list of tags, build targets, builds, users, hosts, etc.
- Cancel tasks and/or builds.
- Download a built package, download logs for the package.
- Create a disk image given an install tree.

- Print the latest builds for a tag , tag inheritance.
- Print the list of Koji XML-RPC API methods.
- Print the package listing for tag or for owner.
- Print a history of operations.
- Build a Maven package from source.
- Create a mock config.
- Retry a canceled or failed task, using the same parameter as the original task.
- Apply or remove a tag to or from builds.
- Show information about a task , track progress of a task.
- Wait for a repository to be regenerated.
- Watch logs in real-time.
- Build a Windows package from source.
- Etc.

4.4.6 Koji XML-RPC client API

The XML RPC API is meant for machine communication of Koji with various systems and thus it is a preferred way also for the sake of this diploma thesis for Jenkins-Koji integration. The XML RPC API uses primarily plain authentication method using login and password, authors claim OpenSSL login is supported as well in the API, see 4.4.6 OpenSSL and Kerberos in XML-RPC API for actual state.

This method is practically equivalent to the CLI interface with few other methods, making it a superset of the CLI to be more precise. The CLI interface is internally using the XML-RPC interface as well.

4.5 Various platform builds

Koji uses RPM as an underlying platform and is also heavily dependent on the Mock tool. Mock serves to create isolated environment simulating the standard RPM setup on RPM based Linux distributions such as Fedora, RHEL, CentOS, OpenSUSE, etc. For example

when MEAD Maven build starts, the systems dependencies including the kernel and basic packages are installed first, then the MEAD build fetches Java compilers and development tools, etc. which are installed from RPM channels as well. Mock also ensures, that the builds are accessing only the files and directories it should, preventing various kinds of security breaches or bundling.

As Java is multi-platform, it does not matter which underlying system architecture is used, e.g. X86_64, i686, PowerPC, etc. However, for the standard RPM builds, Koji hub splits the build between multiple build hosts on various platforms and collects the results afterwards.

Windows native builds are also an exception, as they are obviously out of the RPM ecosystem, so virtual machines are used instead. Developers can define tool-chains as templates that are automatically installed on the fresh virtual machine. Then for each build a fresh virtual machine is provisioned, that is destroyed as soon as the build completes.

4.6 Archival and auditing

Koji uses standard NFS mounts to store the build-roots for some time, till they are garbage collected. These build-roots will never be used again, so it is just for reference for developers if they want to, for example, study debug logs, or find detailed information. However, the resulting build artifacts and build logs are always stored in secure storage that is read only unless the build is explicitly deleted. Even if the build is deleted from Koji build system, the artifacts still remain stored for a period of time, so the delete operation can be reversed. Even after this time period expires, the meta-data about the build or task are always visible in Koji and marked as "deleted" build with information who executed the build and who deleted it.

4.7 Koji usage at Red Hat

As mentioned previously, Koji is an upstream project for the production instance running at Red Hat called Brew, at the same time Fedora has its own Koji instance. Koji has upstream first philosophy, so everything appearing on the production Brew instance should go through Koji first.

Brew is currently used to build all the Enterprise products supported by Red Hat ranging from Red Hat Enterprise Linux (RHEL), Red Hat Enterprise Virtualization (RHEV), JBoss Middleware (multiple products), storage and cloud offerings (OpenStack, OpenShift, etc.). Previously Koji was designed mainly for Fedora and RHEL, which are all based on the RPM model, however as Red Hat grew, there was a need to support other various models such as maven support for Java builds or multi-platform enablement. In the past Koji was heavily dependent on RPM tools such as mock, yum, rpmbuild, etc. which are still core for RPM packaging done for Koji and are helping Koji to isolate build roots, and assure auditability. However, for MEAD maven builds these RPM tools might act as a lower layer, which the developer does not directly interface with.

4.7.1 Layered products

Brew allows to create a structure mapping layering between various Red Hat products, leading to re-use of software components from downstream products to upstream products. In this way JBoss EAP (based on Wildfly application server project) can for example re-use some of the native components built for RHEL.

4.7.2 Source builds and mass rebuilds

Koji was created to be able to handle source builds for all dependencies in the dependency tree of given component or product. This can span hundreds to thousands of software components that have to be maintained and built. Some automation was needed to assure this process is reliable, reproducible, auditable and automated. Koji has embedded support for chain builds, which allow to conveniently rebuild the whole distribution when runtime changes, e.g. there is a new version of gcc compiler.

4.7.3 Stable vs. developer friendly

Historically Fedora and RHEL have been supporting single version of a component on a given system. For example if some application wanted to run on Java 6 and some other required Java 7, operating system was able to provision only one version and the other one had to be installed, maintained and patched by the system administrator. For specific packages, such as Python, which was needed by yum, an older version with different package name was maintained simultaneously with the newer Python 3 runtime.

However, this was a static configuration, which had to be supported by OS developers, and if the end-user wanted their own version of package, they had to maintain it themselves and take care of all the different versions of dependencies required to build those.

Fedora and RHEL use rolling update model of package management, which means, that all the tools on the system using a library must use the same version of that library. E.g. if different applications X, Y and Z use library foo, they all have to use foo 1.2 which is available on the system.

System administrators are usually more conservative and want to maintain system with stable components, however this is in contrast with needs of developers, who usually want latest and greatest packages. For this reason, dynamic software collections or software stacks were introduced into the RPM model and Koji had to adapt to this change. This is allowing users to install and run components at various versions and receive Red Hat support and maintenance for all these components, at the same time satisfying both the system administrator for stable core applications and services, and developers for deployment with newer versions of runtimes and libraries, that can be used by their applications.

4.7.4 Middleware usage

Later on after JBoss team joined Red Hat, a better solutions for Java and multi-platform builds had to be found, leading to Brew / MEAD Maven extension and support for other platforms than linux as described in chapter 4.2.

Middleware team is using Brew / MEAD Maven extension to ensure the same principles adhered across Red Hat such as reproducibility, auditability, security, etc. are met also for Java and Maven builds.

In Java world and for developer oriented audience it is common, to use multiple versions of the same software component at different places. E.g. your core application can be using logging library foo 1.4, whereas some underlying system service can use older version of library foo 1.2. In Java this is solved by class loading or at build level by maven, which takes care of choosing the right version in its dependency resolution mechanism.

The MEAD Maven extension is solving this requirement by providing various certified internal repositories, which are created based on the build target and build tag inheritance

structure, carefully isolating build-time and run-time dependencies and making sure, that Maven builds do not download random builds from public repositories available on the internet such as maven central.

5 KOJAK

Kojak² is a set of scripts that provide an easy way of deploying a Koji instance either on local Linux installation or on a Virtual Machine (VM) provisioned by libvirt. Koji requires certain software packages, paths on file system with specified permission, specific Koji services installation and life-cycle management, etc. [7]. Kojak fulfills all those requirements and provides an easy out of the box installation.

At first user has to select if he wishes to install Koji on a local file system, or if he wants to create a VM. After either of those steps is finished, the user can execute a script that creates required paths, configuration files, starts various Koji services and otherwise prepares the environment for Koji instance. After those steps are finished, the user has a running, functional Koji instance.

To enable support for Maven builds, the user has to import Maven tool-chain and dependencies. Kojak can assist in this phase as well and provide scripts that install base Maven on the file system. Subsequently, all the various Maven build-time and run-time dependencies can be imported to provide all the tools and dependencies for basic Maven builds.

² <https://github.com/sbadakhc/kojak>

II. PRACTICAL PART

6 KOJI PLUGIN IMPLEMENTATION

This chapter describes Koji plugin requirements, analysis, design and implementation. Contributions to various OSS projects and libraries will be described in this chapter.

6.1 Requirements

The goal of the Jenkins-Koji integration should be orchestration of the Koji build system from Jenkins leading to a semi-automated release engineering process. Jenkins will run nightly builds in addition to the continuous integration builds running after each commit. A complete test suite will be run for each of the nightly builds and after builds are stable, Jenkins can be configured to trigger a production build in Koji. After the build in Koji completes, Jenkins will download the artifacts from Koji to a staging location, and again run full test suite to ensure that the process in Koji finished correctly and the build is valid. A number of other smoke tests can be executed on stage as well.

Koji is quite a complex system, so the requirements for analysis had to be narrowed down to simpler use cases. As most of the requirements come from the JBoss middleware team, the main focus is on Java builds. Inside the middleware team, RPM features of Koji are used only in a limited manner, whereas the Koji/Brew extension for Java Maven builds called MEAD is used heavily, so the main focus will be to provide MEAD Maven support for Jenkins-Koji plugin. The plugin should be written in a generic manner that would allow for easy extension for standard RPM builds, which are already supported in Koji, in the future.

6.2 Analysis

This section describes various analysis concerns, namely orchestration, communication protocol between Jenkins and Koji and other considerations for plugin analysis.

6.2.1 Orchestration

As described in previous chapters Koji is a passive XML-RPC driven set of services, that require active orchestration. There is also a complication, that Koji does not provide push notifications and the service needs to be periodically queried to find out whether a task has already finished and artifacts are available for further processing.

Theoretically the orchestration could be:

1. Provided by Jenkins.
2. Provided by Koji.
3. Provided in hybrid manner.

Based on the current Koji design, option 1 seems like the best choice and has been selected for the practical part of this diploma thesis. Option 2 is currently not viable, as Koji is more or less a passive service awaiting either user or machine input. Option 3 is not possible as well, as Koji does not provide events like push notifications.

6.2.2 Jenkins-Koji XML-RPC communication

XML-RPC is a multi-platform standard for machine communication that was created to simplify previous efforts like CORBA and before various Web Services (WS) and REST API came into existence and wider usage.

Jenkins can communicate with other services by various means, spanning across all technologies listed in the previous paragraph. However, Koji options are quite limited in this regard. Koji was originally designed 8 years ago, when there were no better options than XML-RPC communication. Author has commented on a change request³ to provide REST like interface using JSON in the future, however during the defense of this thesis XML-RPC and Koji-CLI are likely to be the only means how to communicate with Koji.

6.2.3 Other design considerations

Currently Jenkins allows a post-build task to be triggered under three conditions:

- Always
- If build is successful
- If build is unsuccessful

Jenkins users can also set up a downstream job containing Koji integration build step, that will get triggered under similar circumstances.

³ <https://fedorahosted.org/koji/ticket/242>

6.3 Implementation

This section provides readers with an overview of the implementation process, describing various engineering challenges and proposed solutions.

Please have a look at Appendix P III: CLASS DIAGRAM, which portrays the overall class diagram for the Jenkins-Koji plugin including the used Jenkins API extension points.

6.3.1 Solving varying XML-RPC extension support

As was described in the previous section, XML-RPC is the only current viable option for machine communication between Jenkins and Koji. XML-RPC should provide a bridge between various languages and enable intermediate multi-platform format of communication. However, its implementations in different languages vary and can sometimes violate the specification⁴ for XML-RPC. Koji is written in Python and uses the XML-RPC library called `xmlrpc.lib` integrated into base Python runtime. Here you can see a method from Koji XML-RPC API:

```
listTagged(tag, event=None, inherit=False, prefix=None,
latest=False, package=None, owner=None, type=None)
```

Python enables a feature called *keyword arguments*⁵, which essentially means that a Python developer can pass arguments to a function or to a method in random order, as long as he states which argument takes which value. Koji-hub component which provides XML-RPC API on the server side is written also in this manner and a lot of optional arguments are in the middle of method arguments. In Java language, there is no such feature as *keyword arguments*, however there are various ways of achieving a similar result. Nevertheless, concerning communication with the Python XML-RPC server this creates issues, as in Java you have to pass `null` values.

If this was not a XML-RPC call but a local call, you could call this method in that way in Python if you wanted to pass just a `tag` and `latest` arguments:

```
listTagged(tag="example-tag", latest=True)
```

4 <http://xmlrpc.scripting.com/spec.html>

5 <https://docs.python.org/2/tutorial/controlflow.html#keyword-arguments>

Whereas in Java you would need to call it like:

```
listTagged("example-tag", null, false, null, true, null, null,
null)
```

Normally this would not be a problem, as the `null` values passed the method would be translated to default arguments on the side of the Python server, however XML-RPC does not officially support `null` types.

None or null types (identical meaning) were not included in the official XML-RPC specification – the type was added later on by Apache Software Foundation (ASF), but only as an extension type⁶. The ASF WS XML-RPC library claims to support `null` types, but the reality is different.

Anytime Jenkins-Koji client tried to pass or receive null values when communicating with the Koji XML-RPC server, the communication failed. Years ago there was a bug⁷ reported to the Koji community, that the API is not XML-RPC compliant, but was waived as `None` type should be widely supported.

The author had to write his own implementation of `TypeFactory` used to handle XML-RPC types in the ASF WS XML-RPC library, that provides correct implementation to parse `None` values both in requests and responses, which solved this issue.

6.3.2 Type casting for Jenkins-Koji XML-RPC communication

As the Koji hub providing the XML-RPC API is written in Python, which has dynamic typing, it is not clear which types are expected to be passed to API methods and what type is returned from those methods. For a long time there was an unresolved ticket⁸ in the Koji community tracker to provide automatically generated method signatures using introspection, which the author has commented on, however it was unlikely this would be fixed before this thesis is finished, so the author had to solve this problem in another way.

⁶ <http://ws.apache.org/xmlrpc/types.html>

⁷ <https://fedorahosted.org/koji/ticket/54>

⁸ <https://fedorahosted.org/koji/ticket/143>

Supposedly, any XML-RPC library should offer ways of debugging incoming and outgoing communication. However, developers of ASF WS XML-RPC library did not include debugging support for showing raw XML requests and responses.

The author had to develop custom implementation of `TransportFactory`, so that XML-RPC communication can be properly read and debugged. This included custom logging infrastructure, as debugging the client inside Jenkins container would be time-consuming and inconvenient for development.

Thanks to this implementation, the author was able to at least partly decipher what arguments are expected by the XML-RPC methods and which types are returned to the client leading to creation of `KojiClient` class handling XML-RPC communication.

The author is considering to contribute both enhancements described in 6.3.2 and 6.3.3 back to the Apache WS XML-RPC library. However, as this library had the latest binary release in February 2010, it might take longer till these contributions are accepted and a binary release is made.

6.3.3 OpenSSL and Kerberos in XML-RPC API

The author of this thesis tried accessing the XML-RPC hub using the HTTPS protocol as a premise for calling SSL enabled login methods. The author developed `initSSL()` method as a proof of concept in `KojiClient` class to test communication with Koji-Hub under a secure protocol. However, Koji-hub was returning handshake failures, even if certificate validation was overridden for testing purposes. After consulting the details with Koji developers this turned out to be an issue in Koji itself.

There is a method for SSL login embedded into the Koji XML-RPC API, but this method is built for Python implementation only and is accessing members that are not a part of the API. Koji developers currently encouraged the author of this thesis to use the user name and password authentication as a sole method for XML-RPC access and authentication given OpenSSL support is not properly configured for cross-language usage in Koji for the time being. The same holds true for the Kerberos login, which does not even have an API support in Koji.

Koji developers recommended calling the Python CLI client from Jenkins in case users wish to authenticate using OpenSSL or using Kerberos. Currently if the user configures OpenSSL or Kerberos authentication in the Jenkins global configuration for Koji plugin, Koji CLI needs to be properly installed and configured. The author sincerely hopes that this issue is going to be fixed in upcoming Koji releases, so OpenSSL and Kerberos authentication can be offered to Jenkins users using non-Linux executors.

6.3.4 Kojak contributions

Kojak was released in a usable state for the purposes of this thesis in the middle of April, the author was one of the first users of Kojak and contributed mainly by testing this set of scripts and defining a structure for Maven build process, which can be used for testing Jenkins-Koji plugin. As this software is under active development a lot is changing at the moment and sometimes Kojak becomes unstable, or has installation issues. The author encountered an installation issue due to a missing libvirt installation and contributed by a simple pull-request to fix this issue⁹. Generally, the maintainer of this tool was very active and reacted to the author's feedback in a timely manner, so author's further contributions or maintaining custom fork of the Kojak project was not necessary.

6.4 Extension points

The base class of every plugin is the `Plugin` class. However, the author has decided to extend the `Builder` class, which is derived from the `Plugin` class instead. `Builder` class serves as an extension point for the build action and everything that is performed during the build action must be defined in the `perform()` method [12]. Therefore the resulting `KojiBuilder` class contains logic to handle Koji configuration for the given projects and also contains the descriptor subclass of `BuildStepDescriptor`, which takes care of global plugin settings such as the Koji instance location, authentication settings and so on.

Logic to execute Koji using XML-RPC is handled in the `KojiClient` and `KojiSession` classes, logic to execute Koji-CLI for OpenSSL or Kerberos authentication

⁹ <https://github.com/sbadakhc/kojak/pull/19/files>

is in the `KojiLauncher` class. `KojiBuilder` creates instance of these classes and outsources the responsibility to run and configure Koji to them.

6.5 Build process

Each project in Jenkins can be executed either periodically, when SCM event happens; or when other conditions occur. When one of these events triggers a build, Jenkins automatically determines a machine that is allocated for this build job based on the executor filter for example only on 64-bit Linux machine. Then the build sections are executed in the order the user has defined them. Once the Koji plugin build action starts to execute, Jenkins passes the `Build` object, `Launcher` and `BuildListener` objects, taking care about the build action of Koji plugin to `KojiBuilder`.

The Koji plugin then starts to log information about the current state of execution to the build console. The Koji client calls the Koji server using XML-RPC methods based on what action was selected by the user and reports back results. The build action is finished by a successful or unsuccessful execution of the Koji plugin. The outcome of the build is passed to the Jenkins instance using the `Builder` subclass.

7 KOJI PLUGIN USAGE

This chapter describes how to install, configure and run the Koji plugin in the Jenkins CI environment. The Author has also provided a simple installation and usage guide to the Jenkins CI community at the Koji plugin wiki page¹⁰.

Short summary of steps to install and use Koji plugin:

1. Download Kojak and install Koji environment on a virtual machine or use an existing Koji installation with Maven support. If you want to use OpenSSL or Kerberos authentication, make sure you have Koji-CLI installed and configured properly.
2. Install plugin: Manage Jenkins > Manage plugins > Available.
3. Configure Koji instance URL and authentication options.

Illustration 9: Koji plugin global configuration

4. Restart Application server / Jenkins.
5. If you want the plugin to execute Koji add "Koji integration" build step and configure task you want Koji to execute.

Illustration 10: Koji plugin project configuration

6. Explore results in Jenkins log or at Koji web.


Appendix P II: Koji web results

More detailed description of these steps follows in this chapter.

7.1 Installation

Users can either directly download Koji plugin .hpi package containing plugin installation or install the Koji plugin automatically using the plugin repository manager embedded in Jenkins. After plugins are installed, it is required to restart the Jenkins master server, so that the plugin can be loaded. Jenkins developers enabled plugin deployment without restarts. However, it is still safer to restart Jenkins when deploying a new plugin.

¹⁰ <https://wiki.jenkins-ci.org/display/JENKINS/Koji+Plugin>



Koji plugin

Koji instance XML-RPC hub URL

Choose authentication for Koji instance

Plain authentication

Koji Username

Koji Password

Plain authentication

Path to SSL certificate for Koji authentication

E-mail Notification

SMTP server

Default user e-mail suffix


Test configuration by sending test e-mail

Illustration 9: Koji plugin global configuration

Users can use existing Koji instance, or provision a new Koji instance in a virtual machine using Kojak or installing it manually. As the VM created has ~37 GB and involves a lot of additional steps to install, it was not desirable for the Jenkins-Koji plugin to manage the installation of Koji. Users can refer to the Kojak documentation² for Koji installation instructions.

7.2 Configuration

Users can invoke Koji operations by adding a Koji integration build step as shown below. This step can be added to various types of Jenkins projects.



Run only if build succeeds
 Run only if build succeeds or is unstable
 Run regardless of build result
 Should the post-build steps run only for successful builds, etc.

Koji integration

Choose task for Koji instance

Koji build

Koji package

Koji target

Koji SCM URL

Other options

Scratch build

Check if you don't want this build to be tagged into Koji database

Add post-build step ▼

- Execute Windows batch command
- Execute shell
- Generate random data
- Invoke Ant
- Invoke top-level Maven targets
- Koji integration**

Illustration 10: Koji plugin project configuration

Please refer to Appendix P I: Koji plugin configuration for larger image of Koji configuration.

Users can select various Koji tasks to be executed from Jenkins, for example:

- Download a Koji Maven build to workspace.
- Run a new Koji Maven build.
- List latest build for a given package.
- Validate Koji configuration and authentication.

7.3 Plugin usage

Users can execute any project containing Koji build action or post-build action and the plugin takes care of the Koji configuration and task execution for them. Below, you can see the output to the Jenkins console when the user selects a new Maven build. Users are then able to navigate to Koji web to monitor provisioning of a fresh VM to execute the Maven build and monitor progress there.

```
[Koji integration] Running maven build build for package  
com.redhat.rcm.maven.plugin.buildmetadata-maven-plugin in tag  
rcm-mw-tools-candidate
```

```
[workspace] $ koji maven-build -Dmaven.test.skip=true rcm-mw-  
tools-candidate git+https://github.com/vtunka/buildmetadata-  
maven-plugin#ce68bfc08000ada70a3aa04d92d7c88271ac5b5e
```

```
Created task: 380
```

```
Task info: http://koji.localdomain/koji/taskinfo?taskID=380
```

8 OPEN-SOURCE COMMUNITY FEEDBACK

This chapter follows up on the initial requirements by the JBoss / Red Hat and Jenkins developer communities described in section 6.1 and their feedback during plugin development and after the initial release of the plugin.

8.1 JBoss / Red Hat community

JBoss Community is an open-source developer community that is mostly known for the JBoss Application server, now called Wildfly. However, there are many other JBoss community projects such as Drools (rules engine), Hibernate (known ORM tool implementing JPA), Infinispan (in memory NoSQL database), etc.

These community projects form the basis for Red Hat supported products, which are also open-source software on their own. The difference between community projects and Red Hat products is in the added value to stability, supportability, security, and other axes of the products.

JBoss product engineers use Jenkins heavily to leverage CI for various products and their components, however for any software components that are shipped to customers Red Hat requires its engineers to run builds in an internal Koji instance called Brew.

So far there has been no integration between Jenkins and Koji. This was especially problematic, as Koji instances are not meant to run tests during the builds. Koji was never designed to run CI, but was designed rather for a production build once the software is at the end of development cycle and is stabilized. If engineers wanted to be sure, that product passes all the integration tests they had to run the tests in Jenkins and then pass the SCM URL to Koji with the SCM tag they knew passes tests in Jenkins.

Also once the production build finished in Koji, engineers needed to check if the release was produced correctly, running smoke tests on binaries downloaded for Koji. This again was a manual task.

The author has received positive feedback from JBoss software engineers that this plugin could be very helpful for the improvement of the current state.

8.2 Jenkins community

Initially the author has posted a note to the Jenkins developer mailing list to announce he is going to develop this plugin, so their potential requirements could be taken into consideration.

Jenkins developer community has accepted the author's plugin and the author's personal Github repository¹¹, which was used for the plugin development, was cloned to the official Jenkins repository¹². The author has created a simple wiki page¹³ in the Jenkins plugin documentation and has released the plugin artifacts to the Jenkins Maven plugin repository, so users can automatically install the plugin using a plugin manager.

8.3 Future plans

Authors of Kojak plan to include a Jenkins instance into the Kojak created VM, that would come with pre-installed Jenkins-Koji plugin. The ultimate goal could be to allow community developers to use environment they are used to, e.g. standard Java build chain with Maven executed from Jenkins CI and handle the clean-room, audited builds on the background without directly interfering with Koji from the side of developers. Afterwards, the meta-data from a local Koji build and Maven repository could be easily exported and verified before executed on a central secure Koji instance.

As plugin maintenance is a long term commitment, the author would like to continue developing and maintaining this plugin after finishing this thesis, so that the development communities can benefit from it.

11 <https://github.com/vtunka/jenkins-koji-plugin>

12 <https://github.com/jenkinsci/koji-plugin/>

13 <https://wiki.jenkins-ci.org/display/JENKINS/Koji+Plugin>

CONCLUSION

The goal of this thesis was to develop a plugin that integrates Koji production build environment with Jenkins CI. I have developed the plugin according to the requirements of development communities. I released the plugin under the open-source MIT license and it was accepted both by JBoss / Red Hat and Jenkins CI development communities. Jenkins-Koji plugin can now be automatically installed on any Jenkins instance around the world from the plugin manager.

In the theoretical part, I described some of the key changes in the Jenkins ecosystem, that happened in the past few years. Advanced techniques and concepts expanding and following up on continuous integration such as continuous delivery and continuous deployment were described.

Related techniques to continuous delivery and deployment such as configuration management, API/ABI compatibility were introduced together with associated best practices and anti-patterns.

Moreover I described release engineering concepts such as clean-room environment, reproducibility, stability, and introduced the Koji build system. Koji architecture and principles were described in detail.

In the practical part, I introduced readers to Koji plugin development including topics such as plugin requirements, analysis, resulting architecture and various implementation options. Then I presented the Koji plugin from the plugin's user perspective along with a description of various configuration options and general usage.

I believe I have fulfilled all the official required points for this thesis. I hope my plugin benefits a number of Jenkins CI and Koji users around the world and will make their work easier.

BIBLIOGRAPHY

- [1] TUNKA, Václav. Plugin Development for Jenkins CI. 2012 [cit. 2014-05-01]. Bachelor Thesis. TBU in Zlín, Faculty of Applied Informatics. Thesis advisor: Tomáš Dulík.
- [2] SMART, John Ferguson. Jenkins: The definitive guide. Sebastopol, Calif: O'Reilly Media, 2011. ISBN 978-144-9305-352.
- [3] HUMBLE, Jez. Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation. Boston: Addison-Wesley, 2010. ISBN 978-0321601919.
- [4] DUVALL, Paul M, Steve MATYAS a Andrew GLOVER. Continuous integration: Improving software quality and reducing risk. Upper Saddle River, NJ: Addison-Wesley, c2007, 283 s. ISBN 03-213-3638-0.
- [5] ROEBUCK, Kevin. Continuous Integration: High-impact Strategies - What You Need to Know: Definitions, Adoptions, Impact, Benefits, Maturity, Vendors. Ruislip: Tebbo, 2011. ISBN 978-1743044841.
- [6] Koji build system [online]. 2014 [cit. 2014-04-25]. Available from: <https://fedorahosted.org/koji/>
- [7] Koji server. In: Fedora project [online]. 2014 [cit. 2014-04-25]. Available from: <https://fedoraproject.org/wiki/Koji>
- [8] FOWLER, Martin. Continuous Integration. In: Martin Fowler [online]. 1.5.2006 [cit. 2014-04-30]. Available from: <http://www.martinfowler.com/articles/continuousIntegration.html>
- [9] Jenkins Documentation [online]. 2014 [cit. 2014-04-30]. Available from: <https://wiki.jenkins-ci.org/display/JENKINS/Home>
- [10] BECK, Kent a Cynthia ANDRES. Extreme programming explained: embrace change. 2nd ed. Boston, MA: Addison-Wesley, 2005, 189 s. ISBN 03-212-7865-8.
- [11] KERIEVSKY, Joshua. *Refactoring to patterns*. Boston: Addison-Wesley, 2005, 367 s. ISBN 03-212-1335-1.
- [12] Jenkins architecture. In: *Jenkins* [online]. 2014 [cit. 2014-05-01]. Available from: <https://wiki.jenkins-ci.org/display/JENKINS/Architecture>

LIST OF ABBREVIATIONS

CI	Continuous Integration
XP	Extreme Programming
RUP	Rational Unified Process
QE	Quality Engineering (usually meaning a QA department)
QA	Quality Assurance (same as QE, if not referred to as a process)
UI	User Interface
UX	Usability (in a sense of intuitiveness and ease of use for given user interface)
SCM	Source Code Management system
VCS	Version Control System
CVS	Concurrent Versioning System (centralized VCS implementation)
SVN	Subversion VersioNing system (next generation CVS like system)
API	Application Programming Interface
SPI	Service Provider Interface
IDE	Integrated Development Environment
OS	Operating System
JDK	Java Development Kit
JRE	Java Runtime Environment
JSP	Java Server Pages
JSTL	Java Server Pages Standard Tag Library
POM	(Maven) Project Object Model
ORM	Object Relational Mapping
JPA	Java Persistence API
CLI	Command Line Interface
VM	Virtual Machine
OSS	Open-Source Software
XML	Extensible Markup Language
RPC	Remote Procedure Call
RPM	RPM Package Manager (recursive abbreviation)
MEAD	Multiplatform Enablement And Delivery
SDK	Standard Development Kit

RHEL Red Hat Enterprise Linux

EAP JBoss Enterprise Application Platform (based on Wildfly application server)

LIST OF FIGURES

Illustration 1: Release dependency on features.....	14
Illustration 2: Continuous Delivery pipeline – cited from [3].....	15
Illustration 3: Blue-Green deployment, cited from [3].....	28
Illustration 4: Canary releasing, cited from [3].....	30
Illustration 5: Koji terminology.....	34
Illustration 6: Brew/Koji Build Tag Inheritance.....	35
Illustration 7: Koji Architecture.....	36
Illustration 8: Koji CLI.....	37
Illustration 9: Koji plugin global configuration.....	53
Illustration 10: Koji plugin project configuration.....	53

LIST OF APPENDICES

Appendix P I: Koji plugin configuration

Appendix P II: Koji web results

Appendix P III: Class diagram

APPENDIX P I: KOJI PLUGIN CONFIGURATION

- Run only if build succeeds
 - Run only if build succeeds or is unstable
 - Run regardless of build result
- Should the post-build steps run only for successful builds, etc.

Koji integration

Choose task for Koji instance

Run a new maven build

Koji build

Koji package

com.redhat.rcm.maven.plugin.buildmetadata-maven-plugin

Koji target

rcm-mw-tools-candidate

Koji SCM URL

git+https://github.com/vtunka/buildmetadata-maven-plugin#ce68bfc08000ada70a3aa0492d7c88271ac5b5e

Other options

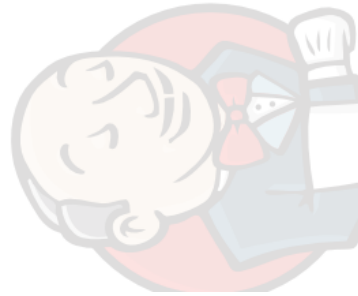
Scratch build




Check if you don't want this build to be tagged into Koji database

Add post-build step

- Execute Windows batch command
- Execute shell
- Generate random data
- Invoke Ant
- Invoke top-level Maven targets
- Koji integration



APPENDIX P II: KOJI WEB RESULTS



Summary
Package
Builds
Tasks
Tags
Build Targets
Users
Hosts
Reports
Search

Search

Package
Wed, 14 May 2014 19:02:02 EDT | login

Information for task buildMaven (rcm-mw-tools-build)

ID 377

Method buildMaven

Parameters <https://github.com/vrnunika/buildmetadata-maven-plugin/tree/08000ada70a3aa04952d7c88271ac5b5e>

Build Tag rcm-mw-tools-build

Options

- repo.id = 14
- properties = maven.test.skip=true

State closed

Created Wed, 14 May 2014 18:50:21 EDT

Started Wed, 14 May 2014 18:50:35 EDT

Completed Wed, 14 May 2014 18:54:53 EDT

Owner kojadmin

Channel maven

Host kojibuilder1.localdomain

Arch noarch

Buildroot /var/lib/mock/rcm-mw-tools-build-13-14

Parent maven (rcm-mw-tools-candidate, /vrnunka/buildmetadata-maven-plugin/tree/08000ada70a3aa04952d7c88271ac5b5e)

Descendants

Waiting? no

Awaited? no

Priority 19

Weight 1.50

Result Files = ["buildmetadata-maven-plugin-1.3.0-scm-sources.zip", com/redhat/rcm/maven/plugin/buildmetadata-maven-plugin/1.3.0-[buildmetadata-maven-plugin-1.3.0.pom], buildmetadata-maven-plugin-1.3.0.jar]

logs = checkour.log, build.log, state.log, root.log


maven_info = group_id=com.redhat.rcm.maven.plugin, artifact_id=buildmetadata-maven-plugin, version=1.3.0

Output

```

com/redhat/rcm/maven/plugin/buildmetadata-maven-plugin/1.3.0/buildmetadata-maven-plugin-1.3.0.jar
build.log (tail) (tail)
mock_output.log (tail)
root.log (tail)
state.log (tail)
com/redhat/rcm/maven/plugin/buildmetadata-maven-plugin/1.3.0/buildmetadata-maven-plugin-1.3.0.pom
buildmetadata-maven-plugin-1.3.0-scm-sources.zip
                    
```

Copyright © 2016-2010 Red Hat



APPENDIX P III: CLASS DIAGRAM

