

Advanced Software Engineering

19.11.2015:
Release Your Stuff 3 Times a Day

DI Stefan Strobl



Agenda

- Dependency & Repository Management
- Build Management and Automation
- Release Management
- Continuous Integration & Deployment



A real world example (2013)

- Large government agency
- 40+ developers
- Source code managed in repository suitable for documents
 - Versioning & labelling per file
 - No notion of a change-set
- Dependencies are stored in the SCM tool without version information
- Releases are built from IDE on developer workstation
- Releases are not versioned
- Releases are stored on a share in the local network
- Releases are manually deployed to test and production stages by operators



Terminology

- Continuous Integration
 - Constantly merge development work with Master/Trunk/Mainline
 - Build & test automatically
- Continuous Delivery
 - Continuously deliver your code to a staging environment for your customer (!) to test
- Continuous Deployment
 - Automatic deployment of code from SCM to production (!)
 - Requires both continuous integration and delivery to be in place



Handling Dependencies...

- Default/naive approach: keeping in SCM (or other file storage)
- Problems:
 - Difficult to (manually) find new libraries and updated versions
 - Loose trace to source (e.g. download page)
 - No standardized naming (convention)
 - Loose version information (unless included in filename or package/Manifest)
 - No information about transitive dependencies
 - SCM not built for versioning binaries (no diff, bad handling of binary files, high resource usage)
 - DSCM (e.g. Git) especially bad (by design) at working with

large binaries



Dependency Management

- Declare which libraries you are using
- Declare which version of a library you are using
- Declare where these libraries are coming from
- Declare in which context you are using the library (test vs. production)
- Have these libraries declare, which libraries they are using
- Automatically retrieve all required libraries from a repository
- Each dependency has a unique name and version
- In a Maven environment, the following triple is used (commonly known as GAV)
 - GroupId (e.g.: at.vie.mach2)
 - ArtifactId (e.g.: info-webapp)
 - Version (e.g.: 3.1.0)



VIE Example: Versioning

- Standard (Maven) semantic versioning (X.Y.Z-Qualifier)
 - X: Major Version: Release
 - Y: Minor Version: (unplanned) feature releases
 - Z: Incremental Version: bugfix-/hotfix-Releases
 - Qualifier:
 - -SNAPSHOT: nightly/local build
 - Mxx: Milestone Version
 - RCx: Release Candidate
 - Final versions: no qualifier
- Examples:
 - 2.0.4 current version, fourth hotfix release
 - 2.1-RC3 release candidate for upcoming feature release
 - 3.0.0-M17 current milestone, in test
 - 3.0.0-M18-SNAPSHOT current development version



Dependency Management - Tools

■ Maven

- multiple tools in one
- steep learning curve, a lot of "magic"



■ Apache Ivy

- pure dependency management
- Builds on same principles and resources



■ Gradle, Buildr & Co

- Different build tools
- (basically) same dependency management



■ OSGi

- Apache Felix OSGi Bundle Repository (OBR)



Repository Management

- Manage all used third party dependencies & repositories
 - Even the ones that are not readily available in public repositories (e.g. Oracle JDBC driver)
- Manage all artifacts created by the project (binaries but also source, documentation, configuration)
 - Central location for all artifacts ensures accessibility (and easy backup)
 - No need to always build complete project
 - Archival of past releases
- Proxy & cache remote repositories
 - results in faster build times
 - easy traceability
 - fault tolerance (e.g. if internet connection and/or remote repositories are unreliable)



Repository Management - Tools

- Sonatype Nexus
 - OSS version sufficient for most users
 - Pro version & support available (e.g. HA, Staging)
- Apache Archiva
 - Completely free & OS, but fewer features
 - Still full fledged repository
- jFrog Artifactory
 - The “newcomer”
 - Pro version with lots of features (e.g. RPM and P2 repositories)
- Detailed comparison: <http://bit.ly/3yey7>

The logo for Sonatype Nexus, featuring a stylized icon of three horizontal bars on the left and the word "Nexus" in a sans-serif font on the right.The logo for Apache Archiva, featuring a stylized orange sunburst icon above the word "archiva" in a bold, lowercase sans-serif font, with a trademark symbol.The logo for jFrog Artifactory, featuring the word "artifactory" in a lowercase sans-serif font, with the "o" in "factory" highlighted in red.

Outside the Java Ecosystem

- Dependency and Repository Management is usually specific to one environment
- Most ecosystems have similar mechanisms, e.g:
 - Pearl: CPAN (Comprehensive Perl Archive Network)
 - Tex: CTAN (Comprehensive Tex Archive Network)
 - Python: PyPI (Python Package Index)
 - PHP: PEAR (PHP Extension and Application Repository)
 - Node.js : Node Package Manager (NPM)
 - Ruby and Rails: RubyGems (RPM format)
- While Java per se does not have such an archive, Maven and Maven Central might be considered equivalent



Build Management and Automation

- Compile source code to binary format
- Package binaries
- Execute automated test cases
- Execute static code analysis and reporting
- Generate documentation
- Run your application locally
- Deploy your application
- Release & publish your artifacts



Build Management – Tools

- GNU Make (<http://www.gnu.org/software/make/>)
 - GNU Make is a tool which controls the **generation of executables** and other non-source files of a program from the program's source files.
 - Make gets its **knowledge of how to build** your program from a file called the *makefile*, which lists each of the non-source files and how to compute it from other files. When you write a program, **you should write a makefile** for it, so that it is possible to use Make to build and install the program.
- Apache Maven
- Apache Ant/NAnt, Gradle, Buildr, MSBuild, Rake, ...



The maven build lifecycle

- **validate** - validate the project is correct and all necessary information is available
- **compile** - compile the source code of the project
- **test** - test the compiled source code using a suitable unit testing framework. These tests should not require the code be packaged or deployed
- **package** - take the compiled code and package it in its distributable format, such as a JAR.
- **integration-test** - process and deploy the package if necessary into an environment where integration tests can be run
- **verify** - run any checks to verify the package is valid and meets quality criteria
- **install** - install the package into the local repository, for use as a dependency in other projects locally
- **deploy** - done in an integration or release environment, copies the final package to the remote repository for sharing with other developers and projects.



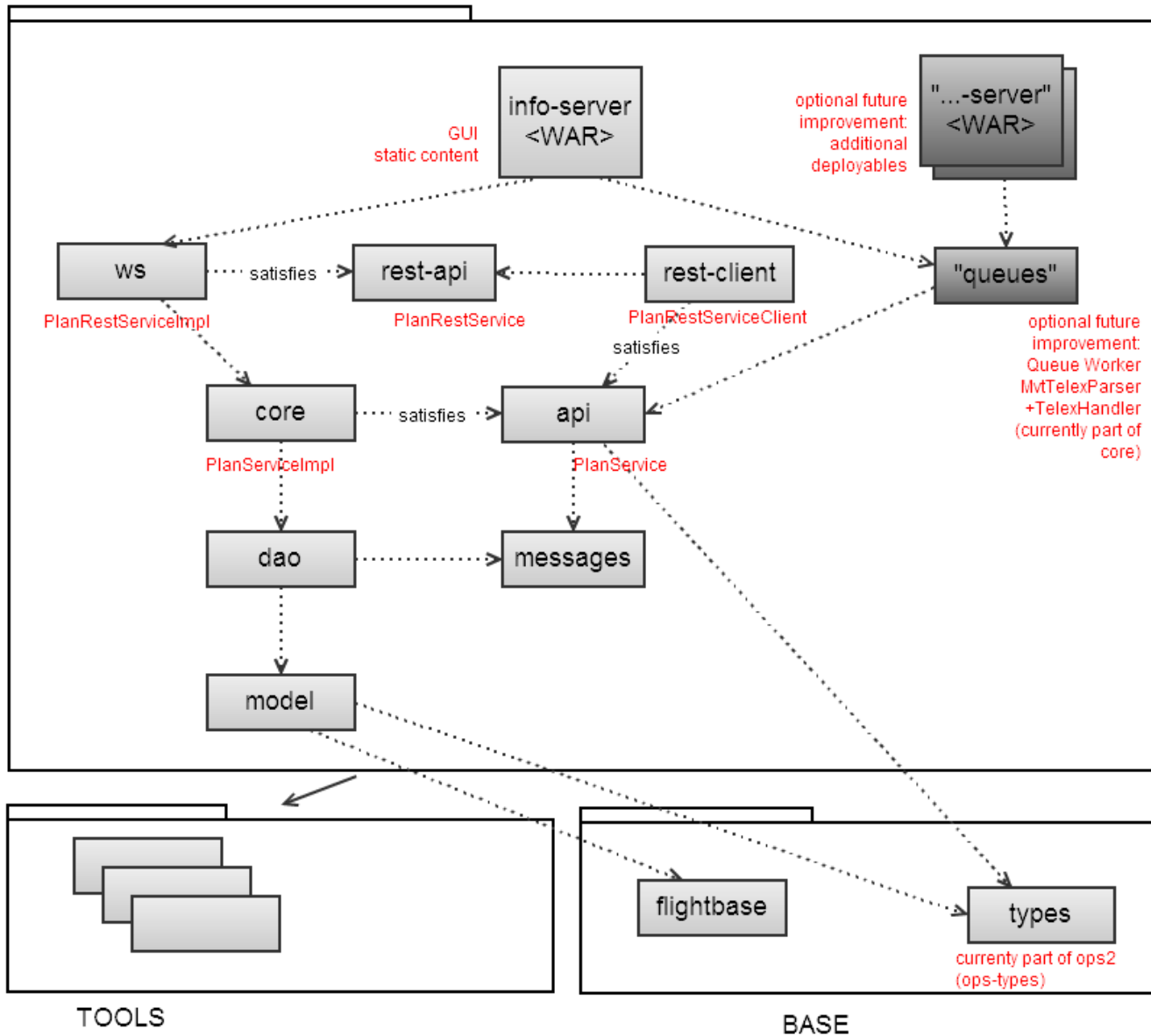
VIE Example: Maven Build

- Maven Multi-Module (Reactor Build)
 - Level 1 Parent-POM: Reactor Configuration
 - Level 2 Parent-POM: Dependency-, Plugin-Management
 - Level 3 Project POMs: Individual Modules
- Auto-versioning – all modules are released together
- Repository: Nexus
- Build Time: ~5 minutes, including ~3000 Unit Tests
- Milestone Releases: 1-2/week, minimum 1/sprint



VIE Example: Modules

Modulstruktur



Focus: releasing your project

Goal: create stable, repeatable artefacts

- maven-release-plugin codifies best practices
- Step 1: mvn release:prepare
 - verify no uncommitted changes & no SNAPSHOT dependencies
 - remove -SNAPSHOT qualifier from project version & update SCM URLs to point to tag destination
 - perform build & execute tests
 - commit to SCM and create tag
 - increase version number, append –SNAPSHOT, update SCM section
 - commit to SCM
- Step 2: mvn release:perform
 - checkout previously created tag
 - build and deploy artifact
 - deploy additional resources (site-deploy)
- tip: always include javadoc and sources in the release



Focus: Change Logs

- Communicate to the stakeholders of your project (QA, project management, dependent projects/systems, end users) what has changed since the last release
- Distinguish between technical and non-technical recipients
 - Technical: simple issue tracking report
 - Non-technical: focus on features and functional bugs, frequently written in “prose”
- Tool support is crucial (Issue Tracking)
 - Minimize overhead
 - Align versioning between issue tracking and code base
 - Developer discipline is crucial (setting fix/target version)



some more preconditions for releasing 3 times a day

- SCM (SVN, Git, Mercurial, ...)
 - no (bigish) project can (and should) do without
- Testing (code quality!)
 - a lot about CI stands (or falls) with sufficient test automation
 - especially hard to build into larger existing projects
 - flaky/bad tests do more harm than good
 - Use code coverage tools
- Static Code Analysis (more code quality!)
 - Sonar, FindBugs, PMD, Checkstyle
 - make the build fail on violations!
- Configuration Management
 - a project is much more than source code
 - problems of keeping configuration in the DB
 - (some) more insight in an upcoming lecture



Continuous Integration – Principles (by M.Fowler)

1. Maintain a code repository
2. Automate the build
3. Make the build self-testing
4. Everyone commits to the baseline every day
5. Every commit (to baseline) should be built
6. Keep the build fast
7. Test in a clone of the production environment
8. Make it easy to get the latest deliverables
9. Everyone can see the results of the latest build
10. Automate deployment



Continuous Integration

- Execute a full build of the project (ideally) after every commit
 - Example: Google does it, at very large scale (see references)
- Always know & communicate the state of your repository
- Publish your build artifacts (binaries, documentation, configuration, reports)
- Deploy and run your application
 - Probably the hardest step
 - Binaries and configuration have to fit together (perfectly)
 - Usually not done continuously (as in every few minutes) - preferably nightly or on an “as needed” basis



Continuous Integration - Tools

- Hudson/Jenkins



- Apache Continuum



- CruiseControl



- Atlassian Bamboo



- many, many more...

Continuous Integration - Dashboard

Jenkins

Jenkins Suisse Stop-tabac dev ENABLE AUTO REFRESH

[Back to Dashboard](#)

Status

- [Changes](#)
- [Workspace](#)
- [Build Now](#)
- [Delete Project](#)
- [Configure](#)
- [Set Next Build Number](#)
- [Duplicate Code](#)
- [Coverage Report](#)
- [SLOCCount](#)
- [Git Polling Log](#)

Project

CI build [edit description](#) [Disable Project](#)

- [Coverage Report](#)
- [Workspace](#)
- [Recent Changes](#)
- [Latest Test Result \(no failures\)](#)

Test Result Trend

Build	Count
#90	0
#171	45
#210	50
#263	75
#345	125
#426	135
#438	140
#977	140

[\(just show failures\)](#) [enlarge](#)

Code Coverage

Classes 45% Conditionals 74% Files 45% Lines 28% Packages 88%

Build	Classes	Conditionals	Files	Lines	Packages
#171	20	70	10	10	20
#210	0	100	0	0	0
#263	65	75	30	30	25
#345	65	75	30	30	25
#426	65	75	30	30	25
#438	65	75	30	30	25
#977	85	75	45	30	28

Permalinks

- [Last build \(#977\), 3 min 17 sec ago](#)
- [Last stable build \(#977\), 3 min 17 sec ago](#)
- [Last successful build \(#977\), 3 min 17 sec ago](#)

Build History (trend)

- [#977 Aug 27, 2012 4:37:27 PM](#)
- [#438 Jun 28, 2012 8:47:42 AM](#)
- [#426 Jun 26, 2012 1:39:39 PM](#)
- [#345 Jun 19, 2012 9:02:20 AM](#)
- [#263 Jun 6, 2012 9:14:42 PM](#)
- [#210 May 31, 2012 8:42:29 AM](#)
- [#171 May 23, 2012 9:58:18 PM](#)
- [#90 May 15, 2012 11:49:41 AM](#)

[RSS for all](#) [RSS for failures](#)

SLOCCount Trend

Build	Lines
#171	2000
#210	3500
#263	5500
#345	7500
#426	8500
#438	8500
#977	8500

[Help us localize this page](#) Page generated: Aug 27, 2012 4:40:45 PM Jenkins ver. 1.470



VIE Example: Continuous Integration Builds

- Tool: Atlassian Bamboo
 - Integrates nicely with Atlassian Tool Suite (JIRA, Confluence, FishEye, ...)
- Build plans:
 - Continuous (~every 5-10 min)
 - Nightly (@02:00)
 - Deploys the current SNAPSHOT binaries to the repository
 - Triggers Site, Tomcat Builds
 - Site (Sonar, JavaDoc)
 - Tomcat (Deploy on Dev-Server)
 - Release (manually triggered)
 - Maintenance Branch Continuous (~15 min)
 - Separate builds for tools and external APIs/contracts

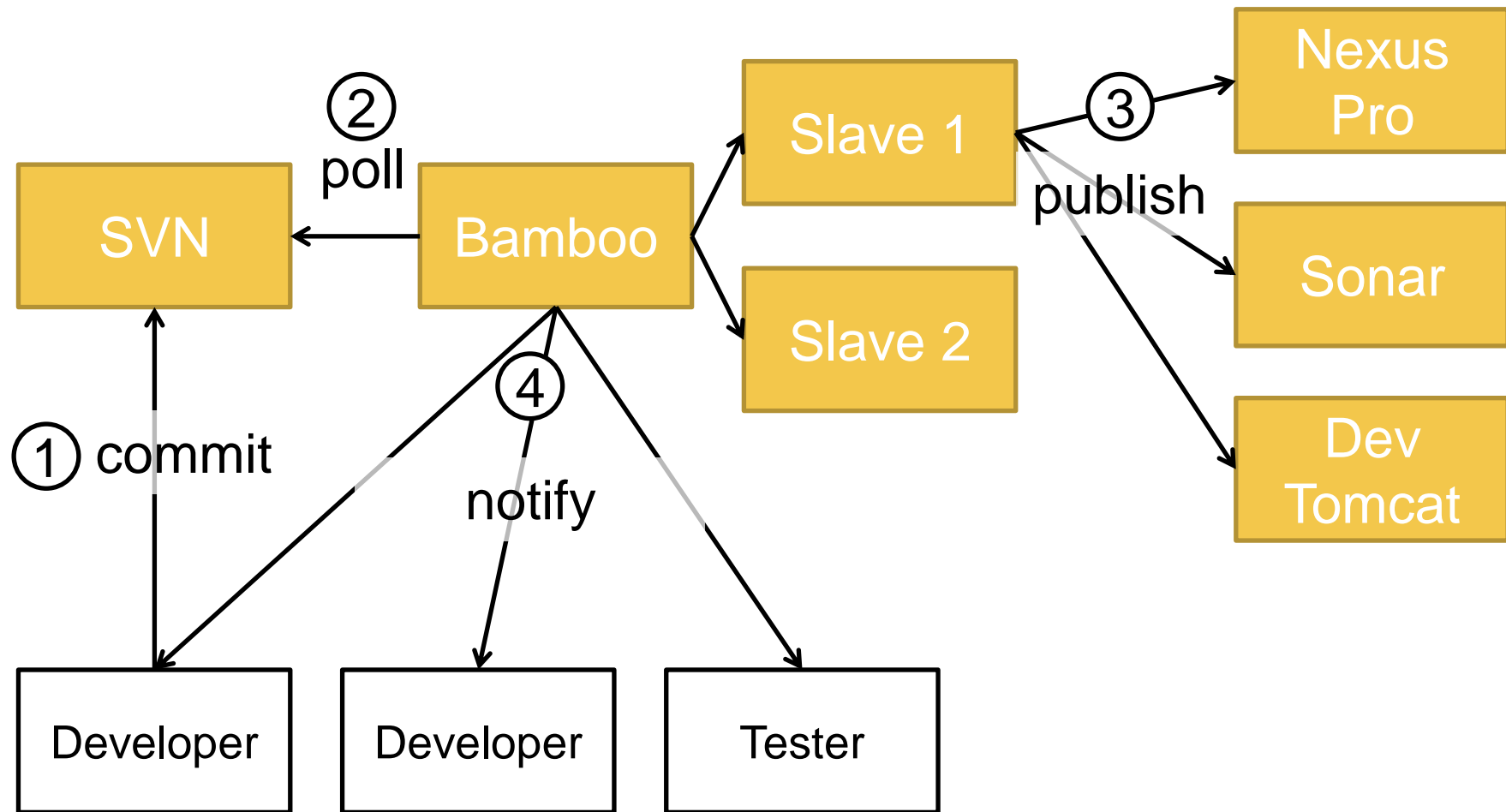


How to put it all together – “Enterprise” Edition

- Private, usually on premise, setup
- Consumes quite a bit of resources
 - Hardware (Servers, Storage, Network, Rack, ...)
 - Human (Administration, Configuration, Know How, ...)
- High cost of entry
- Full control & flexibility
 - tightly integrated with existing resources – e.g. LDAP/Active Directory
 - Choose the tools you need/want
 - Use your infrastructure (e.g. platform, versions)
- Integration of tools can be painful at times
- Scalability can become an issue on large projects
- Support only for individual, commercial tools



VIE Example: CI Environment



How to put it all together – “Cloud” Edition

- Everything is hosted by external service provider(s)
 - No need for expensive hardware & maintenance
 - Low barrier of entry
- Easy setup
 - Fully web based configuration
- Good integration of selected tools
 - Service provider is responsible for and supports a complete tool chain
- Good scalability
 - Scaling up means a few clicks (and a few coins)
 - Possibility for automatic scale up (& down)
- Source code etc has to leave your servers/network
 - ... and often your country
 - Legal implications or barriers



Tools for getting started in the “Cloud”

■ Source Code Management

- GitHub (including issue tracking and further collaboration)
- BitBucket



■ Continuous Integration

- Travis CI (integrated with GitHub)
- CloudBees, Bamboo OnDemand (both commercial)



■ Repository Management

- BinTray
- CloudBees – together with CI



■ Development Server/PaaS

- AWS Beanstalk, Heroku, Google App Engine, ...
- Many others – depending on your platform/needs



Summary

- Invest in your dev-infrastructure, it will pay off
- Tools can only do so much, discipline is required
- The principals of CI are a cornerstone for agile development
- Properly releasing your software (even for QA-builds) ensures traceability



Links & Resources

- <http://www.martinfowler.com/articles/continuousIntegration.html>
- Beck, Kent (1999). Extreme Programming Explained.
- Cunningham, Ward: <http://c2.com/cgi/wiki?IntegrationHell>
- <http://www.methodsandtools.com/archive/archive.php?id=42>
- <http://www.ibm.com/developerworks/java/library/j-ap11297/>
- <http://semver.org/>
- <http://www.eclipsecon.org/2013/sites/eclipsecon.org.2013/files/Continuous%20Integration%20at%20Google%20Scale.pdf>
- <http://devopsnet.com/2011/08/04/continuous-delivery/>
- <http://devopsnet.com/2012/10/25/why-do-we-do-continuous-integration/>
- <http://blog.assembla.com/assemblablog/tabid/12618/bid/92411/Continuous-Delivery-vs-Continuous-Deployment-vs-Continuous-Integration-Wait-huh.aspx>
- <http://www.adaptavist.com/w/atlassian-ondemand-pros-and-cons-cloud/>

