# Tooling for Java EE applications

PA165

Jiří Uhlíř, Martin Kotala

26. 9. 2017

solarwinds

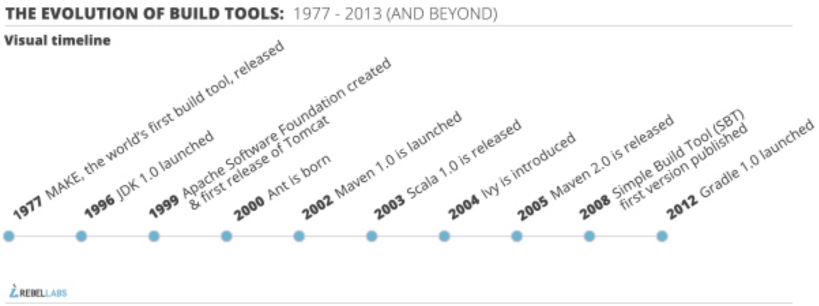
## Contents

## Contents

## Building java applications

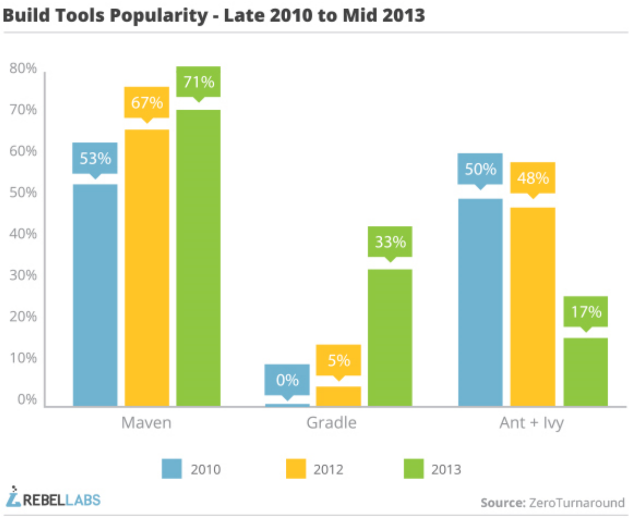
- Motivation aka *Why should we care, let's have bash script with bunch of javac commands*
- Brief look into history
  - Make
  - Ant (with Ivy)
  - Maven
  - Gradle

## Build tools - history overview


THE EVOLUTION OF BUILD TOOLS: 1977 - 2013 (AND BEYOND)
Visual timeline
1977 MAKE, the world's first build tool, released
1996 JDK 1.0 launched
1999 Apache Software Foundation created & first release of Tomcat
2000 Ant is born
2002 Maven 1.0 is launched
2003 Scala 1.0 is released
2004 Ivy is introduced
2005 Maven 2.0 is released
2008 Simple Build Tool (SBT) first version published
2012 Gradle 1.0 launched
REBELLABS

solarwinds

## Build tools - popularity



## Desired properties of quality build tool

- How steep is learning curve
- Time required for build
- Complexity of build script (creation, maintenance)
- Extensibility and flexibility (plugins)
- Build environment consistency
- Extra features (docs, deployment, etc...)
- Integration with developer tools (IDEs, CI servers,...)

## Maven basics

- Software project management and comprehension tool
  - Describes how software project is built
  - Describes software project dependencies
- Developed by Apache Software Foundation
- Created 2004, current version 3.5.0 (Apr 2017)
- Written in Java
- XML configured

## Maven characteristics

- Consistency across projects, standardized build environment
- Lot of implicit functionality (mvn clean is standardized)
- Inheritance via parent poms
- Dependency management, transitive dependencies
- Convention over configuration
  - If mvn conventions are followed then it's easy, if not, it can become very complicated
- Centered around managing entire projects lifecycle

## Maven - POM file

- XML definition of a project
- Defines:
  - Project information
    - Name
    - Company allegiance
    - Docs
    - Test coverage
    - Version
  - Parent relation
  - Dependencies
  - Plugins to be used
  - Build steps

## Maven - Features

- Dependency management
  - Scope
    - compile (default)
    - provided
    - test
  - Library versions
- Versioning
  - SNAPSHOTs
  - Deployment of stable versions into repository
    - Central Maven repository
    - Company proxy - Artifactory, Nexus

## Maven - Features

- Packaging
  - jar (default)
  - war
  - ear
  - pom
- Archetypes
  - Ability to quickly setup template projects
  - Way of standardization of company guidelines and frequently used patterns

solarwinds

## Maven - Plugins

- Standardized shared functionality execution
- All work in Maven is done by plugins
- Build plugins (project consolidation)
- Reporting plugins (tests, site, docs)
- Each plugin can have several goals (e.g. **mvn jetty:run**)
- Standard plugins `https://maven.apache.org/plugins/`
  - **mvn clean**
  - **mvn package**
  - **mvn install**
  - **mvn deploy**

## Maven - Lifecycle

- Clearly defined process for project build and distribution
- Consisting of phases
  - Phase consists of plugin goals
- 3 built-in lifecycles
  - default (validate, compile, test, package, verify, install, deploy)
    - Each phase triggers all previous (eg. mvn test executes validate and compile before test)
  - clean
  - site
- Customizable (defining own lifecycles)

## Maven - Best practices

- Single artifact per pom.xml
- Using POM recommended layouts - ref
- Versions for plugins and dependencies to be kept in parent mgmt section
- Put all artifacts into target folder (follow conventions)
- Use dependency plugin to analyze goals to identify issues
- Follow conventions (e.g. directory structure - ref)
- Do not make Maven act like Ant

## Maven - Demo

## Contents

## Version control

- Motivation
- History
  - One file at a time
  - Centralized (CVS, Subversion)
  - Distributed (Git, Mercurial)

solarwinds

## Git history

- Created in 2005 by Linus Torvalds
  - described by himself as "stupid content tracker"
  - Originally created for linux kernel development
- Inspired by BitKeeper, aiming to be performant and free
- CVS taken as example of what *not to do*
- git - no exact meaning
  - random three-letter combination that is pronounceable, and not actually used by any common UNIX command. The fact that it is a mispronunciation of "get"may or may not be relevant.
  - "global information tracker": you're in a good mood, and it actually works for you. Angels sing, and a light suddenly fills the room.
  - "g*dd*mn idiotic truckload of sh*t": when it breaks
  - `https://github.com/git/git/blob/master/README.md`

## Git characteristics

- Strong support for non-linear development
  - Rapid branching and merging
  - Tools for visualisation and navigation in development history
  - Lightweight branches
- Distributed development
  - Each developer has full history
    - Prevents data loss
    - Subteams can share reposities without access to central repository
  - No need to have access to central repository all the time
  - Changes are commited locally and then pushed to central repository

## Git characteristics

- Variety of protocols supported
  - HTTP/HTTPS
  - FTP
  - SSH
- Efficient handling of large projects
  - Fast (when applying patches)
  - Scalable
  - Fetching version history from locally stored repository is faster then from remote
- Allows various workflows
  - Centralized (enterprise companies)
  - Hierarchical (Linux kernel)
  - Distributed (open source projects, pull requests)

## Git Basics - commands

**git init**

- Initializes empty local repository

**git status**

- Shows current file differences between HEAD commit and current working copy

**git add <filename>**

- Adds a file/directory into commit checklist
- -A (all files not versioned, or not ignored), -u (only updated files already under version control)

**git commit -m <message>**

- Records working copy changes into repository

## Git Basics - commands

**git log**

- Shows latest commits for local repository
- --oneline (condensed view), --graph (includes branches)

**git diff**

- Shows code difference between HEAD commit and current working copy

**git checkout / git reset**

- Removes local uncommited changes

**git reset --soft HEAD 1 / --hard <commithash>**

- Reverts working copy to given commit (soft keeps changes as *to be commited*, hard removes them completely)

**git tag**

- Annotates current version of local repository with tag (such as version)

solarwinds

## Git Basics - commands

**git clone**

- Clones remote repository into local repository and fetches latest changes

**git push**

- Pushes local commited changes into remote repository
- --tags Pushes tags into remote repository
- Cleanup local commits before push using amend or rebase

**.gitignore**

- File for specifying files not to be tracked under version control (binary files, log files, temporary build files, etc.)

**.gitattributes**

- File for specifying attributes to apply for certain paths
- Used for example for specifying line endings

## Git Basics - Demo

## Contents

solarwinds

## Git Branching - Overview

- A branch represents an independent line of development.
- Lightweight implementation of branching – Git stores a branch as **a reference to a commit**.
- Keeps history as a tree, where **each commit is a node** in the tree, and has one or more parents.
- History is **extrapolated through the commit relationships**.
- It's a good practice to **spawn a new branch to encapsulate your changes** no matter how big the changes are.

## Git Branching - Local Branches

- **Non-tracking local branches**
  - Exist on user's machine.
  - Not associated with any other branch.
  - User needs to specify upstream branch when running push or pull commands.
- **Tracking local branches**
  - Exist on user's machine.
  - Tracking branch is a branch that has a direct relationship to another branch.
  - Local tracking branches in most cases track a remote tracking branch.
  - Allow user to run git pull and git push without specifying which upstream branch to use.

## Git Branching - remote-tracking branches

- **Remote**
  - Remote connection (bookmark) into other repository.
- **Remote branch**
  - Branch on a remote location.
- **Remote-tracking branch**
  - Local cache for what the remote repositories contain.
  - (remote)/(branch)
    - origin/master
    - origin/test-branch
- **Note:**
  - “origin” and “master” are not special.

## Git Branching - merge

- Way of putting a forked **history back together** again.
- **Non-destructive** operation.
- All the operations always **merge into the current** branch.
- Git has **several distinct algorithms** to accomplish the merge.

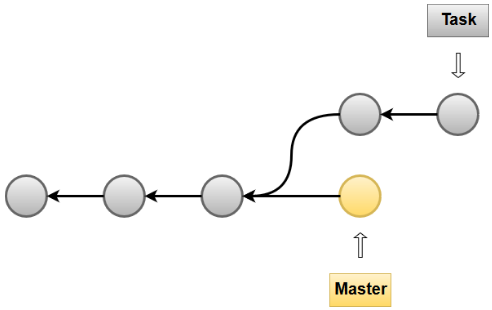
Note:

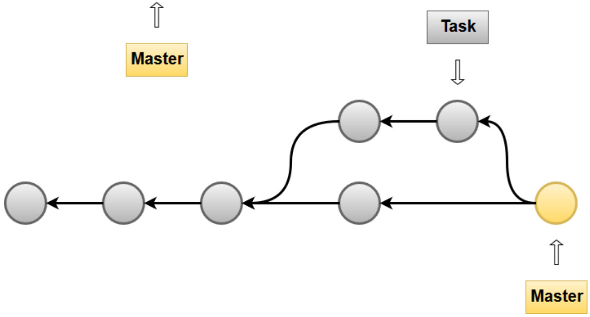
- git pull command effectively runs git fetch and git merge.

## Git Branching - merge

- **3-Way Merge**
  - Creates **merge commit** that ties together the histories of both branches.
  - Merge commit as a **symbolic joining** of the two branches.
  - Original **context is maintained**.
- **Fast-Forward Merge**
  - Requires **linear path** from the current branch tip to the target branch.
  - Usually **facilitated through rebasing** – suitable for small tasks and fixes.
  - Context of the affected commits as part of an earlier feature branch is lost.

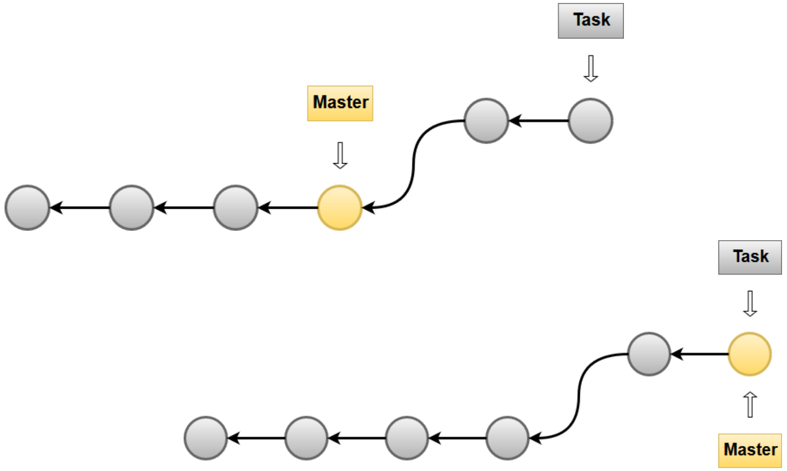
## Git Branching - 3-way merge





solarwinds

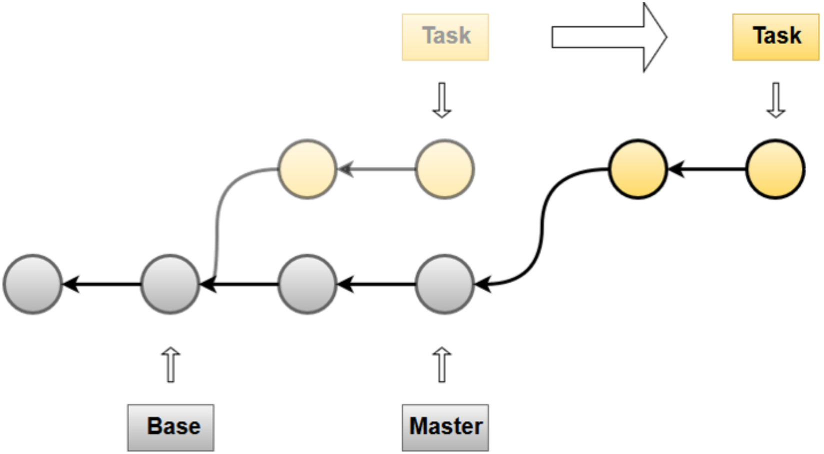
## Git Branching - fast-forward merge



## Git Branching - rebase

- Process of **moving or combining** a sequence of commits to a new base commit – alternative to merge.
- Makes the branch appear as if you'd created it from a different commit.
- Git takes changes from your branch and **replays them** on top of the destination branch.
- Result branch **looks the same** but it's composed of entirely new commits.
- Do not rebase commits that exist **outside your repository** (unless you have a good reason to do so).

## Git Branching - rebase



## Git Branching - rebase

- Rebase helps to maintain a **linear project history** – that allows an easier investigation of regression issues.
- **Workflow example:**
  1. User creates the new task branch from master and starts working in it.
  2. There is an active development on a master branch.
  3. User wants to get the latest updates from master to task branch.
  4. User performs regular rebase operation to move his commits on top of latest master commits.
  5. User is done with his task and performs the final rebase and merge to master.
  6. Git is able to apply fast-forward algorithm for the merge.

## Git Branching - rebase

- **Interactive rebase**
  - Allows user to **alter individual commits** in the process of rebasing.
  - Support for powerful **history rewriting** features.
  - Useful for history cleanup: reword, edit, squash, fixup
  - Always amend commits that have **not been pushed** yet to avoid confusion.

solarwinds

## Git Branching - conflicts

- Conflicts may occur during merge and rebase operations.
- Use the suitable **merge strategy** to avoid conflicts.
- When the conflict occurs:
  - **Abort** the merge with.
  - **Work through** the conflict and **continue**.
- **Visualize and resolve** conflict in merge tool.

Note:

- Rebase has an option to **skip/bypass** conflicting commit.

## Git Branching - commands

**git branch**

- List all of the branches in your repository.

**git branch <new-branch-name>**

- Create a new branch called <new-branch-name>.

**git branch -d <existing-branch-name>**

- Delete the existing branch, safely. Use –D to force it.

**git checkout <existing-branch-name>**

- Navigate between the existing branches.

**git checkout -b <new-branch-name> <remote>/<branch-name>**

- Create and checkout a new local tracking branch.
- Or simply use the previous command if there is only one remote-tracking branch called <existing-branch-name>. This applies to Git 1.6.6+.

## Git Branching - commands

**git remote**

- List the connections to remote repositories. Use -v to get URLs.

**git remote add <remote-name> <url>**

- Add a new connection to a remote repository.

**git remote rm <remote-name>**

- Remove the connection to a remote repository.

**git fetch <remote-name>**

- Fetch all of the branches from the remote repository.

**git pull <remote-name>**

- Fetch the remote's copy of the current branch and merge it into the local copy.

**git push <remote-name> <branch-name>**

- Push the specified branch to remote repository

## Git Branching - commands

**git merge <existing-branch-name>**

- Merge the specified branch into current one and let Git to choose an algorithm.

**git merge --no-ff <existing-branch-name>**

- Merge the specified branch into current one and generate merge commit.

**git merge --ff-only <existing-branch-name>**

- Merge the specified branch into current one and refuse to merge when fast-forward is not possible.

**git checkout task**
**git rebase master**

- Move the entire task branch to begin on the tip of the master branch, effectively incorporating all of the new commits in master.

## Contents

## Pull Requests - overview

- Mechanism for a developer to **notify coworkers.**
- **Not a Git feature**, functionality provided by e.g. Bitbucket or GitHub.
- Interface for **discussing proposed changes** before integrating them into the official project code base.
- **4 pieces** of information:
  - source repository
  - source branch
  - destination repository
  - destination branch

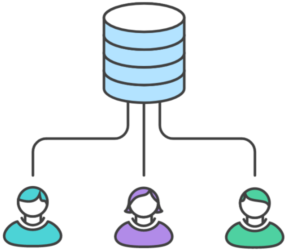
## Pull Requests - workflow

1. Developer **creates task/feature** branch.
2. Once done with coding the **developer pushes his dedicated local branch** to public repository.
3. Developer **creates pull request** (it's good idea to rebase local branch).
4. Team members **review the code** and provide feedback.
5. Once approved the project maintainer **merges the changes** to target branch.

## Contents

## Centralized Workflow

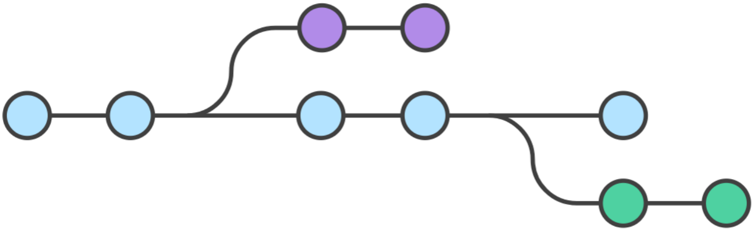
- Master branch only.
- Easier transition to Git.
- Users don't need to change their workflow.



solarwinds

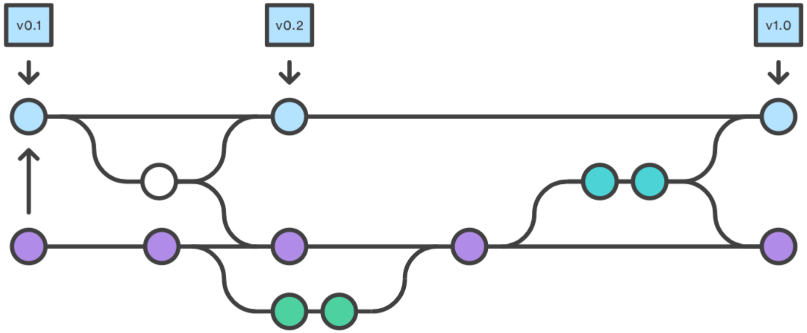
## Feature Branch Workflow

- Feature development isolated in feature branches – pull request for discussing the changes.
- Master branch should always hold stable code.



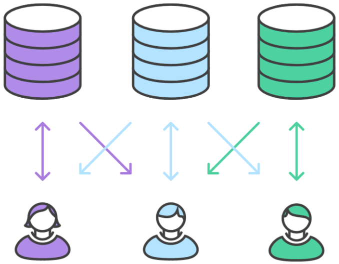
## Gitflow Workflow

- Strict branching model designed around the project release.
  - The master branch stores the official release history.
  - The develop branch serves as an integration branch for features.



## Forking Workflow

- Forks of the repository need to be created.
- Each user has a personal public repository.
- Project maintainer integrates the changes into official repository via pull requests.
- official repository = public repository of project maintainer.



## Tools Overview

- GitHub, Bitbucket, GitLab etc.
- Feature overview:
  - Git repository management.
  - Access control.
  - Branch permissions.
  - Pull requests with code reviews and comments.
  - Code aware search.
  - Git Large File Storage (LFS).
  - 3rd party integrations (Jira, CI server etc.)

## Contents

## Continuous Integration

- Definition by Martin Fowler:
    - *"Continuous Integration is a software development practice where members of a team integrate their work frequently, usually* ***each person integrates at least daily*** *- leading to multiple integrations per day. Each integration is* ***verified by an automated build (including test)*** *to detect integration errors as quickly as possible."*
- Team of developers integrate and test their code early and often.
- Reduce the risk of seeing "integration hell".
- Automated testing is a key part of Continuous Integration.
- Address the conflicts and issues early.
- Get fast feedback on code changes.

solarwinds

## 11 Practices by Martin Fowler

1. Maintain a Single Source Repository.
2. Automate the Build
3. Make Your Build Self-Testing
4. Everyone Commits To the Mainline Every Day
5. Every Commit Should Build the Mainline on an Integration Machine
6. Fix Broken Builds Immediately
7. Keep the Build Fast
8. Test in a Clone of the Production Environment
9. Make it Easy for Anyone to Get the Latest Executable
10. Everyone can see what's happening
11. Automate Deployment

## Typical Issues

- Builds fails too often.
- Developers are not able to verify the code before commit.
- Test engineers are waiting too long to get new version.
- Single step of build pipeline cannot be repeated.
- Number of muted tests is growing.
- It's hard to identify source of the issue – single build contains a lot of changes.

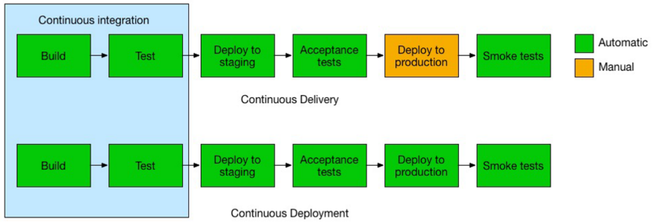
## Improvements

- Automate everything that can be automated (build, environment preparation, tests etc.).
- Let developers to run all validation steps locally.
- Isolate each build step and make it standalone.
- Split monolithic build into smaller build configurations.
- Execute some of the build steps in parallel (e.g. tests and package preparation).
- Scale build configurations: Fast/Common/Nightly

## Continuous Delivery and Deployment

- **Continuous Delivery**
  - Definition by Martin Fowler:
    - *"Continuous Delivery is a software development discipline where you build software in such a way that the **software can be released** to production at any time."* by Martin Fowler
  - Exists on top of Continuous Integration.
  - Deployment and release process automated.
  - Every change **can be deployed** to production but you may choose not to do it because of business priorities.
- **Continuous Deployment**
  - Sometimes confused with Continuous Delivery but goes one step further.
  - Every change that passes all stages of the production pipeline is released to customers.

## Continuous Delivery and Deployment



## Tools Overview

- Jenkins
- TeamCity
- GitLab CI/CD
- Bamboo
- Bitbucket Pipelines

## References

- Git Documentation
- Martin Fowler's web
- Atlassian Git Tutorial
- Atlassian Documentation
- ZeroTurnaround web

Questions?