Software-Engineering 2

Prof. Dr.-Ing. Gerhard Wanner
Email: wanner@hft-stuttgart.de

BUILD MANAGEMENT

Overview

- Build automation
  - Motivation
  - Tool examples
    - Ant
    - Maven
    - Gradle
- Continuous Integration
  - Motivation
  - Tool examples
    - Jenkins
    - Cloud.

Build automation - Motivation 1

- Within the software development several tasks have to be done over and over again…
  - Compiling
  - Building for Test
  - Testing
  - Configuration
  - Building for Deployment
  - Deployment
  - Generation of Reports
  - Generation of Documentation
  - Cleaning up
  - …

Build automation - Motivation 2

- Doing these tasks by hand is …
  - complicated,
  - tedious and
  - error-prone

Build automation

- Scripting or automating the process of compiling computer source code into binary code
- Ideally a one-step process for turning source code into a working system
  - → Saves time and reduces errors.

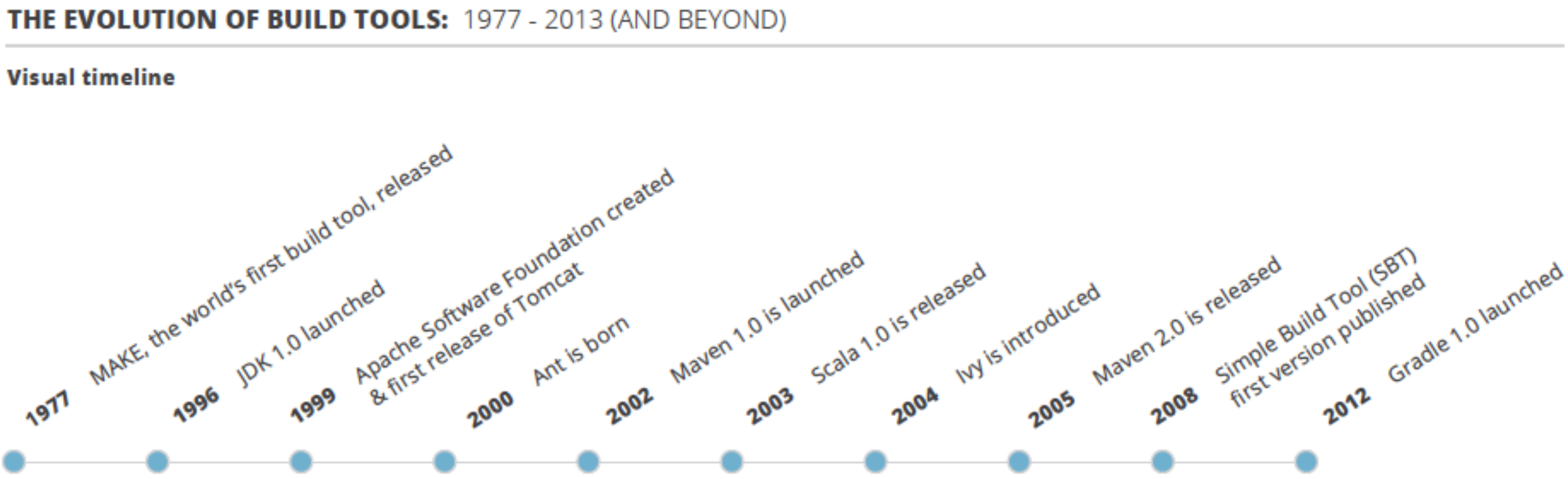
Kinds of build automation

- Commanded Automation
  - Such as a user running a script on the command line
  - Examples: Make, Ant, Maven, Gradle
- Continuous Integration
  - Scheduled Automation
    - Such as a Continuous Integration server running a nightly build
    - Examples: Cruise Control, Hudson, Jenkins, TeamCity, Bamboo
  - Triggered Automation
    - Such as a Continuous Integration server running a build on every commit to a version control system
    - Examples: Cruise Control, Hudson, Jenkins, TeamCity, Bamboo.

Overview

- Build automation
  - Motivation
  - Tool examples
    - Ant
    - Maven
    - Gradle
- Continuous Integration
  - Motivation
  - Tool examples
    - Jenkins
    - Cloud.

The evolution of build tools



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

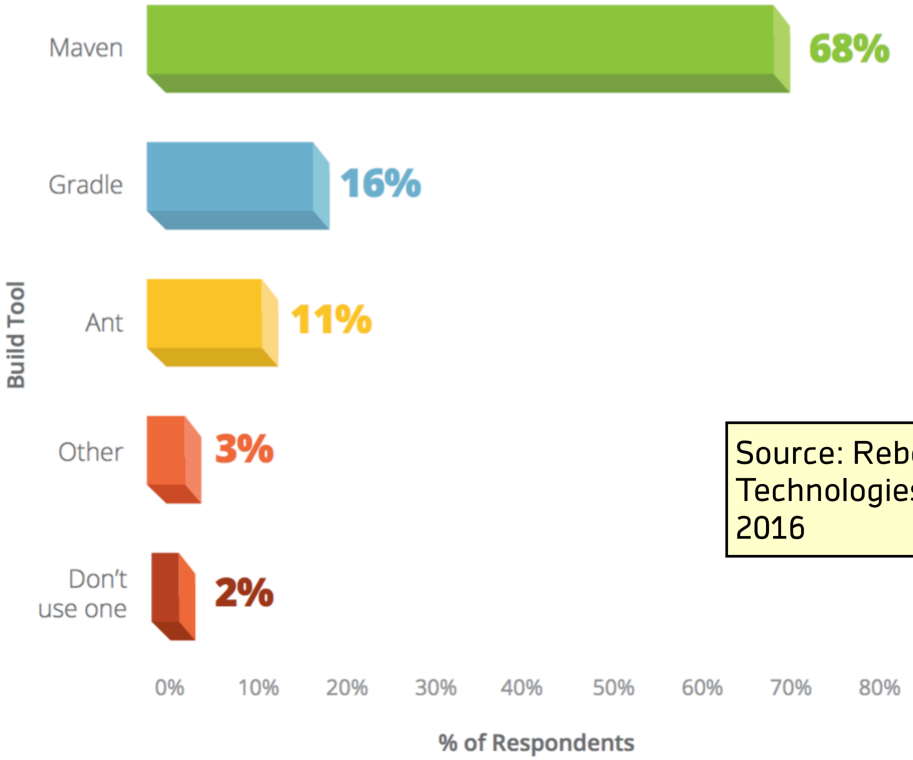
2005 Maven 2.0 is released

2008 Simple Build Tool (SBT) first version published

2012 Gradle 1.0 launched

Source: Rebellabs, Java Build Tools

Which build tool do you use most often?



Source: Rebellabs, Java Tools and Technologies Landscape Report 2016

Overview

- Build automation
  - Motivation
  - Tool examples
    - Ant
    - Maven
    - Gradle
- Continuous Integration
  - Motivation
  - Tool examples
    - Jenkins
    - Cloud.

Ant - History

- Ant was originally part of the Tomcat code base, and ONLY supported the Tomcat project
  - When Tomcat became part of the Jakarta project, it became evident that it could solve some of the problems associated with **`make`**
- Releases
  - Released publicly in April of 2000 with Tomcat (Version 0.3.1)
  - First independent release was Ant 1.1 in July of 2000
  - Actual Version is 1.10.x (March 2020).

What is Ant?

- A build tool like **`make`**
- Open Source
- Implemented in Java and written for Java
- Used to build many open-source Java projects
- Why is it called Ant?
    - According to the author (James Duncan Davidson) …
        - Because ants do an extremely good job of building things
        - Because ants are very small and can carry a dozen times their own weight
        - Stands for "Another Neat Tool".

Ant vs. Make

- How is Ant like *Make?
  - It’s a build tool
  - It has build “targets”

```
make [options] [target]
ant  [options] [target]
```

- How is Ant different?
  - XML-based instead of script based
  - Extendable using Java classes
  - Built-in multi-platform support
  - Automagically builds projects recursively
    - So, there is no need for multiple build files.

Ant - Features

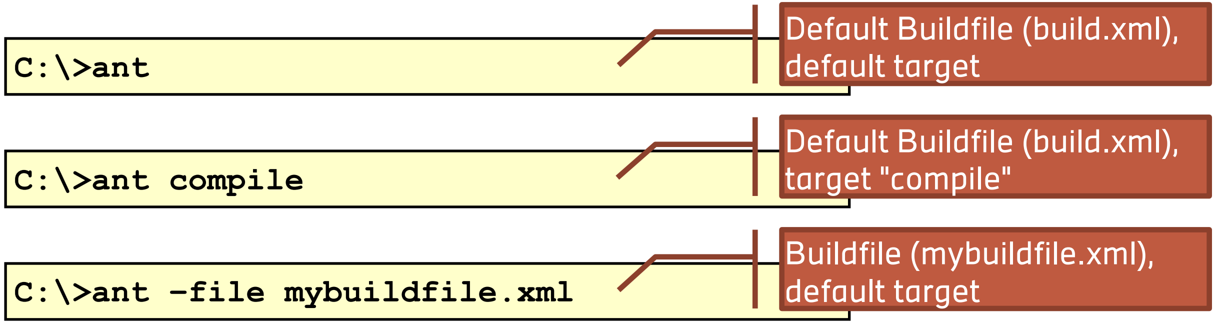
- Java based
  - Only requires a Java VM to run
  - Easily expandable (using Java → no additional language necessary), and inherently cross-platform
  - All Java IDEs have built-in support for Ant
- Many built-in Java-specific tasks
  - Such as JavaDoc, creation of WAR and JAR files, EJB support, …
- Configuration in a XML script
  - No cryptic shell commands
  - Each task is defined in a separate XML block, so adding or removing tasks is simple
- Fast
  - Uses the same VM for the entire process
- It can operate recursively, so only one Ant build script is required for most projects
- Built-in support for git, SVN, FTP, JUnit, …

How does Ant work?

- Ant commands (or tasks) are implemented by Java classes
  - Many tasks are built-in
  - Others come in optional JAR files
  - Custom commands can be easily created
- Each project using Ant will have a build file
  - Typically called `build.xml` – Ant's default
- Each build file is composed of any number of targets
  - These correspond to common activities like compiling and running code
- Each target is composed of tasks
  - Executed in sequence when the target is executed
  - Like make, Ant targets can have dependencies
    - For example, the build file may have a task to build a JAR file and can specify that before the JAR file can be created, package B needs to be compiled, and before package B is compiled, package A must be compiled.

How does Ant work? (cont)

- Ant Targets …
  - Can be specified on the command-line
  - A default can be defined, and will be used if no target is specified on the command-line
  - Ant stops execution of the build when any errors are encountered
    - Generally, a good thing, but frustrating on large projects
  - Each target is executed only once, no matter how many other targets define it as a dependency
  - Some tasks may be skipped, such as a compile task if no source has changed.



```
C:\>ant
```

Default Buildfile (build.xml), default target

```
C:\>ant compile
```

Default Buildfile (build.xml), target "compile"

```
C:\>ant –file mybuildfile.xml
```

Buildfile (mybuildfile.xml), default target

What can Ant do?

- Built in commands
  - Compile

```
<target name="compile">
    <javac srcdir="${src}" destdir="${build}"/>
</target>
```

  - Package
    - Jar, Tar, War, Zip

```
<target name="dist" depends="compile">
    <jar jarfile="${dist}/lib/MyProject.jar"
        basedir="${build}"/>
</target>
```

  - Test
    - JUnit, Cactus, <java>
  - Document
    - JavaDoc, JUnit Reports, Build Log, Mail/MimeMail
  - Deploy
    - SCP, FTP, Telnet, Unwar, Untar, Unjar, Unzip
  - Source code management
    - Subversion, GIT, CVS, SourceSafe, ClearCase, Perforce
  - Full list of integrated tasks:
    - https://ant.apache.org/manual/tasklist.html

Ant - Build file example (1/2)

```
<project name="MyProject" default="dist" basedir=".">
    <description>
        simple example build file
    </description>
    <!-- set global properties for this build -->
    <property name="src" location="src"/>
    <property name="build" location="build"/>
    <property name="dist" location="dist"/>
    <target name="init">
        <!-- Create the time stamp -->
        <tstamp/>
        <!-- Create the build directory used by compile -->
        <mkdir dir="${build}"/>
    </target>
    <target name="compile" depends="init"
        description="compile the source " >
        <!-- Compile code from ${src} into ${build} -->
        <javac srcdir="${src}" destdir="${build}"/>
    </target>
```

Ant - Build file example (2/2)

```
    <target name="dist" depends="compile"
        description="generate the distribution" >
        <!-- Create the distribution directory -->
        <mkdir dir="${dist}/lib"/>
        <!-- Put everything in ${build} into the
             MyProject-${DSTAMP}.jar file -->
        <jar jarfile="${dist}/lib/MyProject-${DSTAMP}.jar"
             basedir="${build}"/>
    </target>
    <target name="clean"
        description="clean up" >
        <!-- Delete ${build} and ${dist} dirs -->
        <delete dir="${build}"/>
        <delete dir="${dist}"/>
    </target>
</project>
```

Typical Ant problems

- Ant is hard to debug (echo and option –v helps)
- 95% of Ant-Problems are path-problems (which are hard to detect)
  - Ant may have problems with spaces in filenames
  - Ant assumes you are using Sun-recommended Java package structures → Anything in the **`com.myco.demo`** package will be found in **`./com/myco/demo/`** directory)
- Ant depends exclusively on timestamps to decide what needs to be rebuilt, so be careful!

Executing an Ant script

- In its basic form, an Ant script can be run by simply typing **`ant`**
- Command-line option summary:

```
ant [options] [target [target2 [target3] ...]]
```

  - Options (trimmed to fit page):

```
-help                         print this message
-projecthelp                  print project help information
-version                      print the version information and exit
-quiet                        be extra quiet
-verbose                      be extra verbose
-logfile                      file use given file for log output
-logger classname             the class that performs logging
-listener classname           add class as a project listener
-file                         use specified buildfile
-Dproperty=value              set property to value.
```

Ant - Commands

- List targets in build.xml of the current directory
  - Command

```
ant -projecthelp
```

  - Output

```
Searching for build.xml ...
Buildfile: C:\MyJava\Samples\build.xml
Main targets:
 clean                  deletes all generated files
 compile                compiles source files
 deploy                 deploys the war file to Tomcat
 dtd                    generates a DTD for Ant build files
 javadoc                generates javadoc from all .java files
 prepare                create output directories
 test                   runs all JUnit tests
 undeploy               undeploys the war file from Tomcat
 war                    builds the war file
```

Ant - Creating custom tasks (1/2)

- Steps
  - Create a Java class that
    - extends **`org.apache.tools.ant.Task`**
    - has a no-arg constructor
  - Plan the attributes, text and child elements that your task element will use
  - For each attribute, add a set method
    - **`type`** can be a String or any Java primitive type
    - Example for the attribute **`attrName`** (name-resolution is done using reflection)

```
public void setAttrName(type param)
```

  - Add the method that implements the tasks

```
public void execute()
```

  - Compile the class
  - Ensure that it can be found on the CLASSPATH.

Ant - Creating custom tasks (2/2)

- Example

```
import org.apache.tools.ant.BuildException;
import org.apache.tools.ant.Task;

public class MyVeryOwnTask extends Task
{
    private String msg;

    // The method executing the task
    public void execute() throws BuildException {
        System.out.println(msg);
    }

    // The setter for the "message" attribute
    public void setMessage(String msg) {
        this.msg = msg;
    }
}
```

Ant - Use custom tasks

- Steps
  - Define the task in the build-file

```
<taskdef name="mytask" classname="MyVeryOwnTask"/>
```

Fully qualified class name of the task

Name of the task

  - Use the custom task

```
<target name="owntask" depends="init"
    description="An own Ant extension">
    <!-- Just output a Message -->
    <mytask message="Hello Task!"/>
</target>
```

Parameter

Name of the task as defined

Ant - External tasks/tools

- Anakia
  - XML transformation tool based on JDOM, Velocity and Ant
- Anteater
  - A set of Ant tasks for the functional testing of websites and web services
- Checkstyle
  - Help programmers write Java code that adheres to a coding standard
- CleanImports
  - Removes unneeded imports and formats import sections
- Clover
  - An Ant-based Code Coverage tool
  - Provides method, statement, and branch coverage analysis
- jMetra
  - Tool for collecting code metrics
  - Compiles the results into JavaDoc-styled documentation to analyze project metrics over time

More see http://ant.apache.org/external.html

Ant & ivy

- Ant lacks a dependency manager
- Apache ivy is a very powerful dependency manager oriented toward Java dependency management (not exclusively)
- Features
  - Integrated with Apache Ant
    (Apache Ivy is a subproject of Apache Ant)
  - Simple to use (see next page)
  - Dependency reports
  - Non intrusive
  - Transitive dependencies
  - ... and much more. See http://ant.apache.org/ivy/

Ant & ivy - Example

- ivy.xml

This file is used to describe the dependencies of the project on other libraries

```
<ivy-module version="2.0">
    <info organisation="org.apache" module="hello-ivy"/>
    <dependencies>
        <dependency org="commons-lang" name="commons-lang" rev="2.0"/>
        <dependency org="commons-cli" name="commons-cli" rev="1.0"/>
    </dependencies>
</ivy-module>
```

Exact information for the libraries you depend on is necessary for the attributes. Ivy uses the maven 2 repository by default, so information about that can be found at mvnrepository.com

- build.xml

```
<project xmlns:ivy="antlib:org.apache.ivy.ant" name="hello-ivy">
    ...
    <target name="resolve" description="retrieve dependencies">
        <ivy:retrieve />
    </target>
</project>
```

The retrieve-task with no attributes will use the default settings and look for a file named ivy.xml for the dependency definitions

Overview

- Build automation
  - Motivation
  - Tool examples
    - Ant
    - Maven
    - Gradle
- Continuous Integration
  - Motivation
  - Tool examples
    - Jenkins
    - Cloud.

Maven

- What is Maven?
  - Intelligent Project-Management-, Build- and Deployment-Tool
- Comparison with Ant?
  - Ant: Build-Tool to automate standard-tasks
    - Clean, Compile, Pack (JAR, WAR, EAR), Deploy, JavaDoc, ...
- Maven is based on Ant
  - Maven can do anything that Ant can
  - Adds an abstraction-layer
  - Added value services.

Java EE build and deployment as we know it today is pretty much standardized. Every enterprise has some variations, but *in general it is all the same: deploying EARs, WARs, and EJB-JARs*. Maven captures this intelligence and lets you achieve the build and deployment in *about 5-6 lines of Maven script compared to dozens of lines in an Ant build script*.

(Maven Magic – theserverside.com)

Maven vs. Ant

- Process using Ant
  - Write build-scripts
  - Run target

Weak points of Ant	Solution process when using Ant
Bad Cross-Project-Reuse / Cross-Developer-Reuse No possibility to reuse targets for different projects	Copy-and-Paste
No support for conditional logic / loops • If … / For each …	Custom targets Usage of conditional logic in Java
Handling with libraries/dependencies	Apache ivy
No defined build lifecycle	Every project defines its own build lifecycle

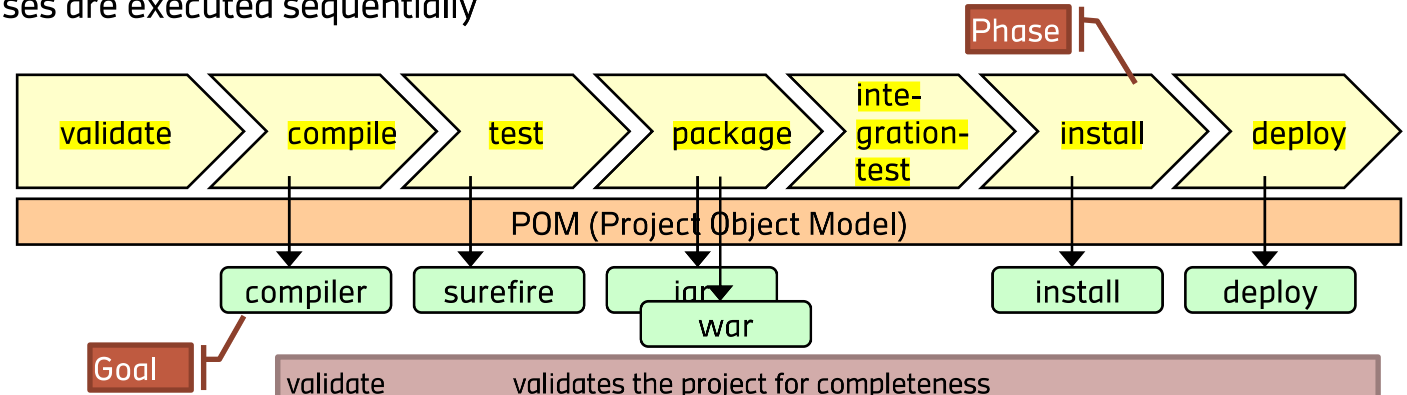
How Maven addresses these problems

- Plug-In-concept
  - Definition of reusable, project-independent goals
  - Plug-Ins are primarily written in Java, though there are providers for other scripting languages
- Project Object Model (POM)
  - Description of the project
    - Structure, dependencies, project-management-details
  - Central XML-file (project.xml) for the project-artifacts
- Repository-concept
  - Maven builds up an own local repository
  - Maven uses remote-repositories to load libraries

- Solution process with Maven
  1. Describe project and configure plug-Ins
  2. Run existing plug-Ins.

Maven - Build Lifecycle

- Maven arranges the build-lifecycle into phases
  - The goals are assigned to phases, not to the artifacts the developer wants to create
  - Phases are executed sequentially



Important phases

validate	validates the project for completeness
compile	compiles the sourcecode
test	executes unit-tests, e.g. using the JUnit-plugin
package	packages the compiled file into archives (Jar, War, etc)
integration-test	deploys the artifacts into test-environments and executes tests
install	installs the artifact into a local repository
deploy	releases the artifact into a remote-repository to be used by other developers and projects

Maven - Assignment of a plug-in to a phase

- **Usually, all plug-ins are attached to a phase**
  - There are some exceptions if a plug-in doesn't belong to a phase
  - Examples:
    - clean – Delete all generated artifacts
    - eclipse:eclipse – Create application frame.

```xml
<build>
 ...
 <plugins>
  <plugin>
   <groupId>org.apache.maven.plugins</groupId>
   <artifactId>maven-checkstyle-plugin</artifactId>
   <version>2.1</version>
   <executions>
    <execution>
     <phase>validate</phase>
     <goals>
      <goal>check</goal>
     </goals>
    </execution>
   </executions>
  </plugin>
 </plugins>
 ...
</build>
```

Example for the integration of the checkstyle plug-in into the phase "validate" as goal "check"

Maven - Full "Default Lifecycle" (1)

Lifecycle Phase	Description
validate	Validate the project is correct and all necessary information is available to complete a build
generate-sources	Generate any source code for inclusion in compilation
process-sources	Process the source code, for example to filter any values
generate-resources	Generate resources for inclusion in the package
process-resources	Copy and process the resources into the destination directory, ready for packaging
compile	Compile the source code of the project
process-classes	Post-process the generated files from compilation, for example to do bytecode enhancement on Java classes
generate-test-sources	Generate any test source code for inclusion in compilation
process-test-sources	Process the test source code, for example to filter any values
generate-test-resources	Create resources for testing
process-test-resources	Copy and process the resources into the test destination directory
test-compile	Compile the test source code into the test destination directory

Maven - Full "Default Lifecycle" (2)

Lifecycle Phase	Description
test	Run tests using a suitable unit testing framework. These tests should not require the code be packaged or deployed
prepare-package	Perform any operations necessary to prepare a package before the actual packaging. This often results in an unpacked, processed version of the package
package	Take the compiled code and package it in its distributable format, such as a JAR, WAR, or EAR
pre-integration-test	Perform actions required before integration tests are executed. This may involve things such as setting up the required environment
integration-test	Process and deploy the package if necessary into an environment where integration tests can be run
post-integration-test	Perform actions required after integration tests have been executed. This may include cleaning up the environment
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	Copies the final package to the remote repository for sharing with other developers and projects (usually only relevant during a formal release)

Maven - "Clean" and "Site" Lifecycle

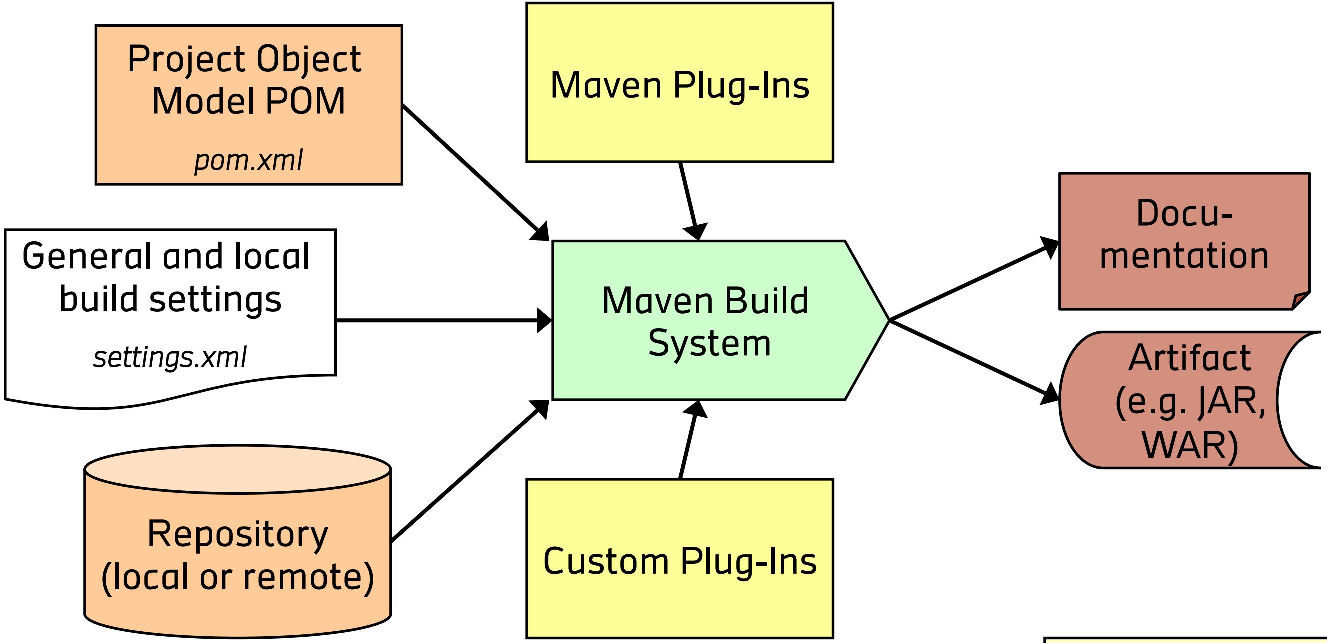
- Clean Lifecycle
  - Running `mvn clean` invokes the clean lifecycle which consists of three lifecycle phases

Lifecycle Phase
pre-clean
clean
post-clean

- Site Lifecycle
  - Maven can also generate project documentation and reports about the project, or a collection of projects
  - Project documentation and site generation have a dedicated lifecycle which contains four phases.

Lifecycle Phase
pre-site
site
post-site
site-deploy

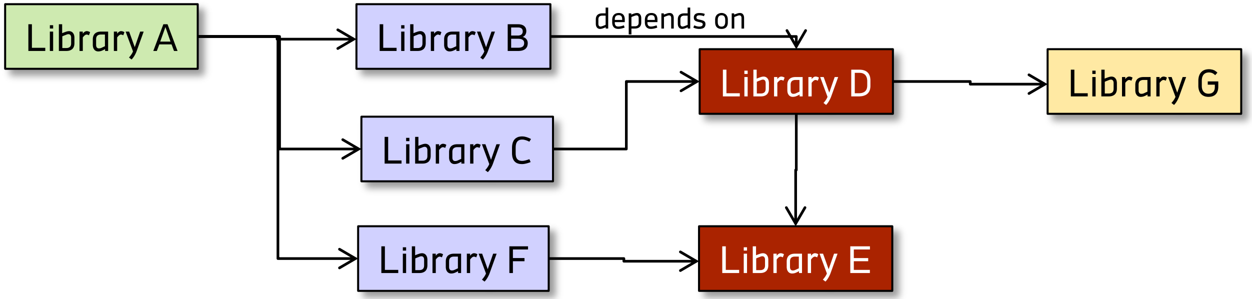
Elements of Maven



Elements and process of Maven

Dependencies in Maven (1/3)

- Maven includes a sophisticated dependencies-concept
  - Transitive dependencies: every component knows on what other components it depends (including version number)
    - Recursive
    - For huge applications with many libraries essential.



Example: Transitive dependency-management in Maven

Dependencies in Maven (2/3)

- Maven includes a sophisticated dependencies-concept
  - Scope: Defines which dependencies are valid at a specific time.

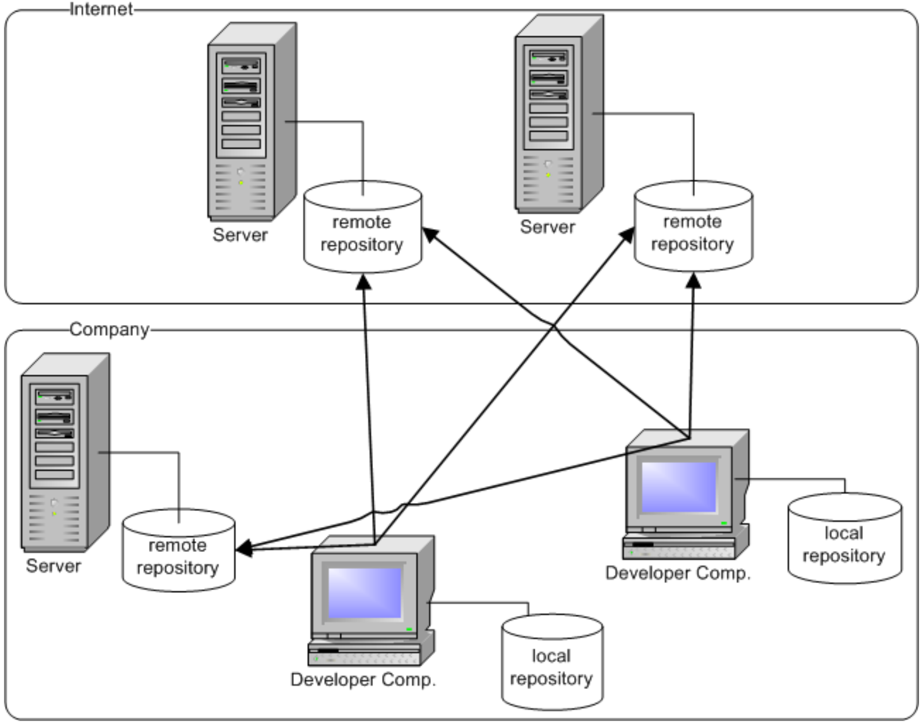
compile	Library is included always (default)
provided	Library is provided through the runtime system (e.g. servlet-container)
runtime	Library only is used during runtime, not for compilation and test
test	Library is only used for tests

```
<dependency>
           <groupId>org.apache.myfaces.core</groupId>
           <artifactId>myfaces-api</artifactId>
           <version>2.0.0-SNAPSHOT</version>
           <scope>compile</scope>
</dependency>
<dependency>
           <groupId>org.apache.myfaces.core</groupId>
           <artifactId>myfaces-impl</artifactId>
           <version>2.0.0-SNAPSHOT</version>
           <scope>compile</scope>
</dependency>
```

Example: Dependency-definition for Apache MyFaces 2.0.0 SNAPSHOT (developer-version)

Dependencies in Maven (3/3)

- Repository concept
  - Local and remote repositories possible
  - Local repositories are…
    - faster
    - more reliable
    - integration into version control possible
  - Remote repositories are…
    - more actual.



Maven - Support for complex projects

- To support complex projects, projects can be aggregated
  - Better team-support
    - When using the top-level POM, goals are executed for every subproject
    - Separate usage of subprojects possible.

Aggregation of projects

```
<project>
        <modelVersion>4.0.0</modelVersion>
        <groupId>com.company</groupId>
        <artifactId>myshop</artifactId>
        <version>2.0</version>
        <packaging>pom</packaging>
        <modules>
                <module>myshop-ear</module>
                <module>myshop-users-war</module>
                <module>myshop-admin-war</module>
                <module>myshop-catalog-ejb</module>
        </modules>
</project>
```

Maven – Extensions using plug-ins

- Plug-ins in Maven are called mojo
- Implementation
  - Implementation is similar to an Ant-task
  - The mojo has to extend **`org.apache.maven.plugin.AbstractMojo`** and override the method **`execute()`**
- Annotations
  - Using annotations, the lifecycle (**`@phase`**), the goal (**`@goal`**) and the description (**`@description`**) of the mojo can be set
  - For parameters of the plug-in there's the annotation **`@parameter`**
  - Parameters can be defined as required (**`@required`**) and/or readonly **`(@readonly`**)
- Usage
  - The plug-in can be installed in a repository using **`mvn install`** or **`mvn deploy`**
  - There it can be used directly by other projects.

Overview

- Build automation
  - Motivation
  - Tool examples
    - Ant
    - Maven
    - Gradle
- Continuous Integration
  - Motivation
  - Tool examples
    - Jenkins
    - Cloud.

Gradle

- Gradle is one of the youngest build tools and its developers tried to combine Ant's power and flexibility with Maven's dependency management and conventions
  - After several years of developers, Gradle v1.0 was released in 2012
  - It's developing fast and already adopted by some big enterprises – Gradle, for example, was selected to be the build tool for Google's Android OS
- One interesting point is that with Gradle, XML is not used anymore
  - Instead, developers have a Domain Specific Language (DSL) based on the JVM language Groovy, which was invented so that developers could ditch the verbosity of XML and write more simple and clear statements.

Gradle - Example

```
apply plugin: 'java'

sourceCompatibility = 1.5
version = '1.0'
jar {
    manifest {
        attributes 'Implementation-Title': 'Gradle Quickstart',
                   'Implementation-Version': version
    }
}

repositories {
    mavenCentral()
}

dependencies {
    compile group: 'commons-collections', name: 'commons-collections',
            version: '3.2'
    testCompile group: 'junit', name: 'junit', version: '4.+'
}
```

This will apply the Java plug-in to the project, which adds a number of tasks to the project

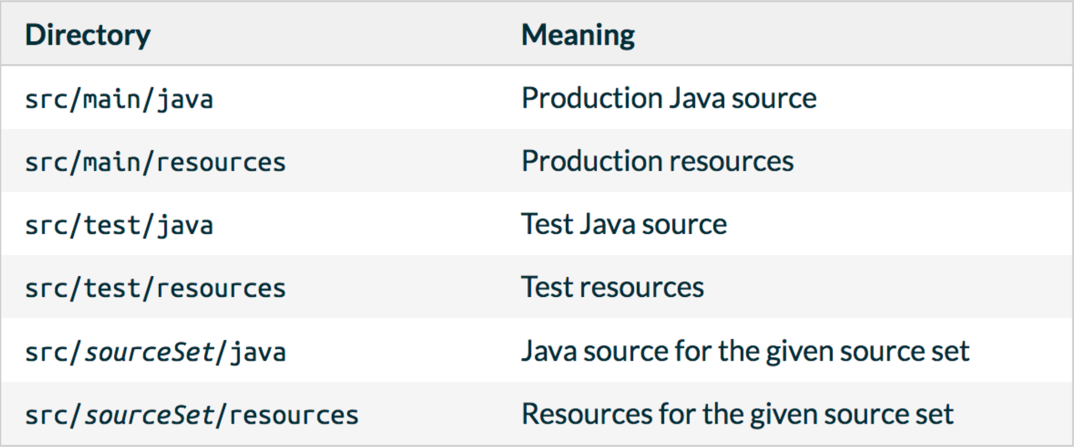
The Java plugin adds properties to the project. These default values can be changed. Here the version number for the Java project and the Java version of the sources is specified. Also some attributes to the JAR manifest are added

Use the public Maven repository

Dependencies: production classes have a compile-time dependency on commons collections, and the test classes have a compile-time dependency on junit

Gradle – Example (2)

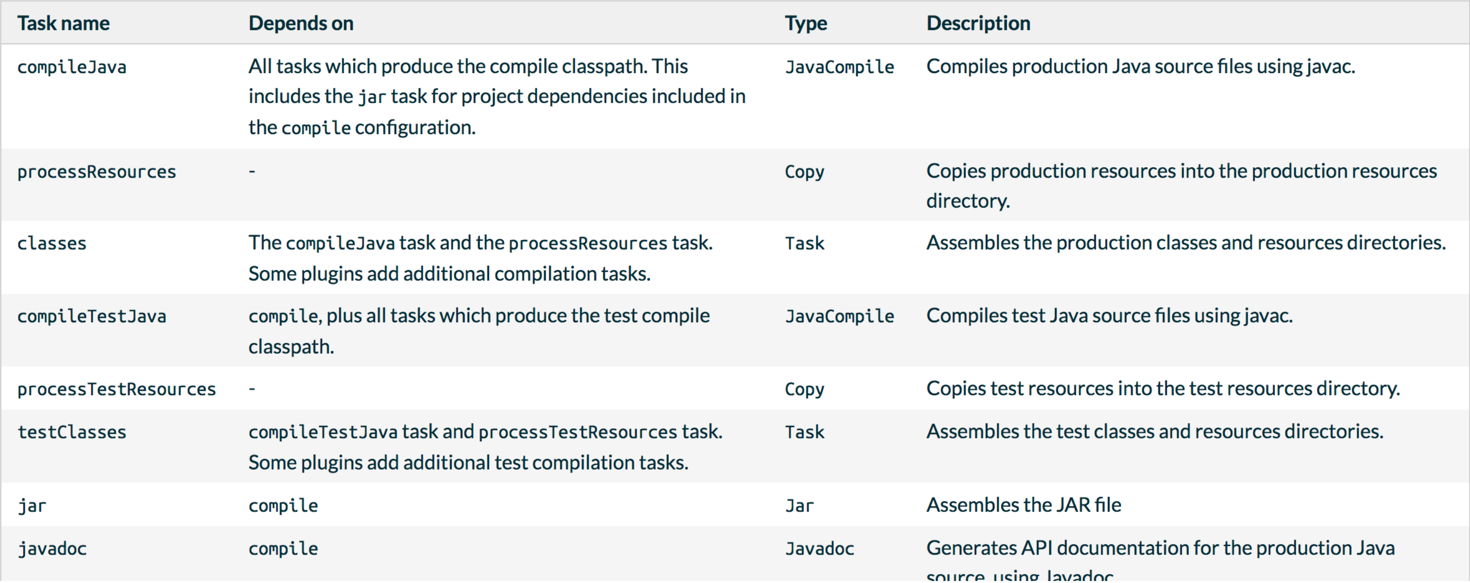
- Java plug-in – project layout
  - Gradle assumes a default project layout when applying the Java plug-in



Directory	Meaning
`src/main/java`	Production Java source
`src/main/resources`	Production resources
`src/test/java`	Test Java source
`src/test/resources`	Test resources
src/*sourceSet*/java	Java source for the given source set
src/*sourceSet*/resources	Resources for the given source set

Gradle – Example (3)

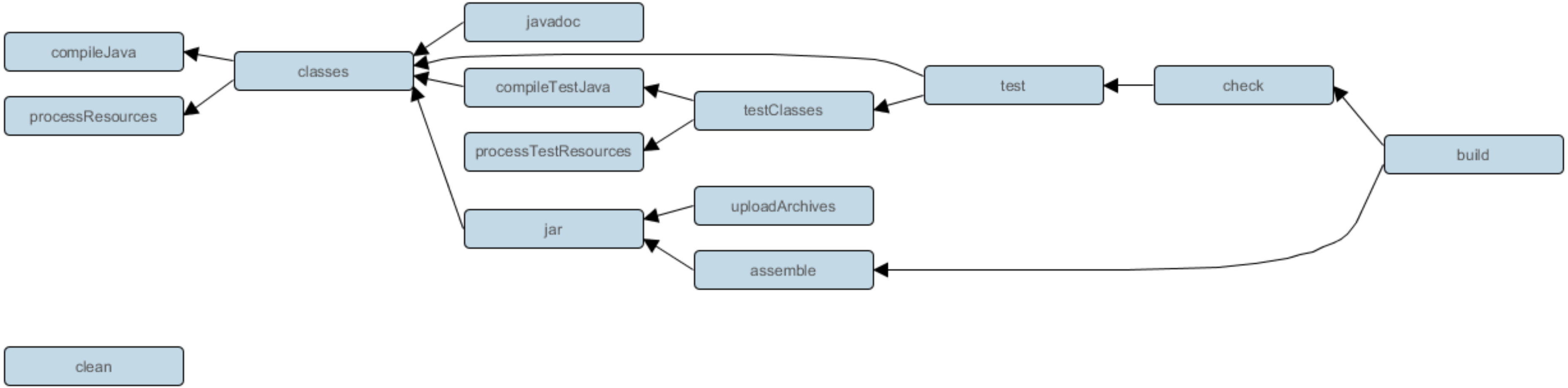
- Java plug-in - tasks
  - Applying the Java plug-in to the build adds tasks to the project
  - More information at https://docs.gradle.org/current/userguide/java_plugin.html



Task name	Depends on	Type	Description
`compileJava`	All tasks which produce the compile classpath. This includes the `jar` task for project dependencies included in the `compile` configuration.	`JavaCompile`	Compiles production Java source files using javac.
`processResources`	-	`Copy`	Copies production resources into the production resources directory.
`classes`	The `compileJava` task and the `processResources` task. Some plugins add additional compilation tasks.	`Task`	Assembles the production classes and resources directories.
`compileTestJava`	`compile`, plus all tasks which produce the test compile classpath.	`JavaCompile`	Compiles test Java source files using javac.
`processTestResources`	-	`Copy`	Copies test resources into the test resources directory.
`testClasses`	`compileTestJava` task and `processTestResources` task. Some plugins add additional test compilation tasks.	`Task`	Assembles the test classes and resources directories.
`jar`	`compile`	`Jar`	Assembles the JAR file
`javadoc`	`compile`	`Javadoc`	Generates API documentation for the production Java source using Javadoc

Gradle – Example (4)

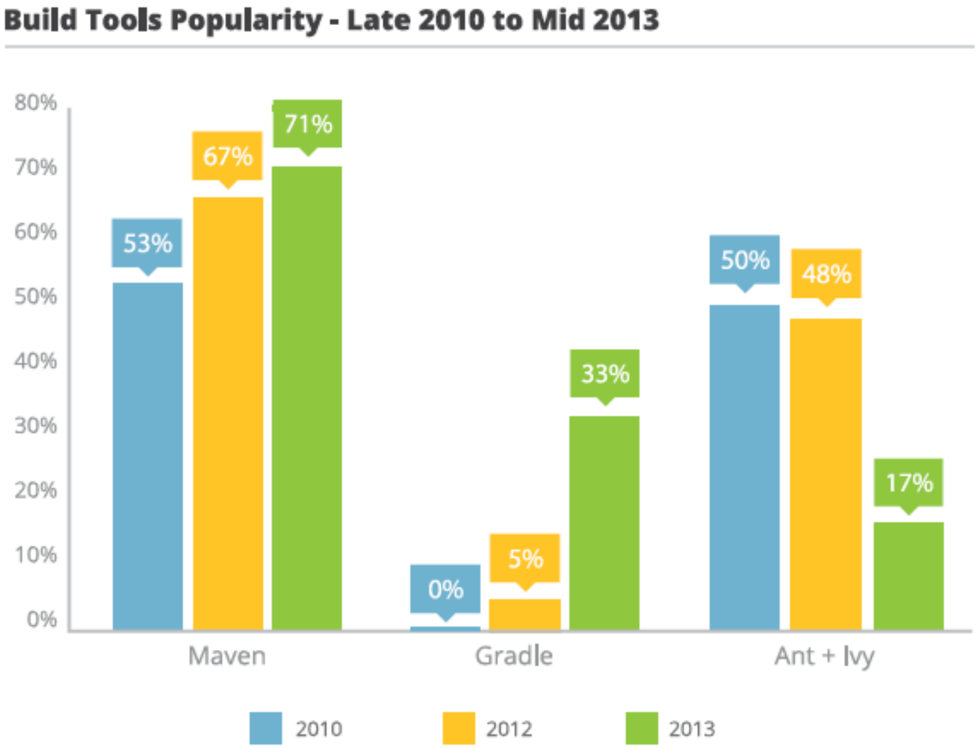
- Java plug-in – relationship between tasks



Gradle – Features & Benefits

- Declarative builds and build-by-convention
  - At the heart of Gradle lies a Domain Specific Language (DSL) based on Groovy
  - Those elements also provide build-by-convention support for Java, Groovy, OSGi, Web and Scala projects
- Language for dependency based programming
  - The declarative language lies on top of a general purpose task graph, which you can fully leverage in your builds
- Gradle scales
  - Gradle scales very well. With the incremental build function, this is also true for tackling the performance pain many large enterprise builds suffer from
- Many ways to manage dependencies
  - Gradle provides convenient support for any external dependency strategy
  - From transitive dependency management with remote Maven and Ivy repositories to jars or directories on the local file system.

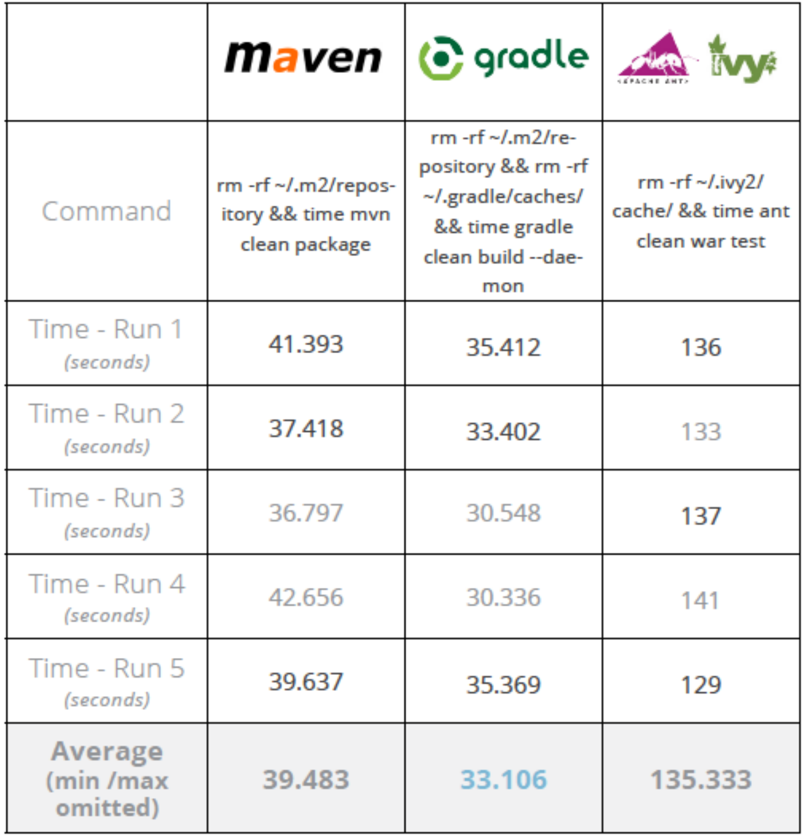
Build tools popularity



Source: Rebellabs, Java Build Tools

Build tools performance

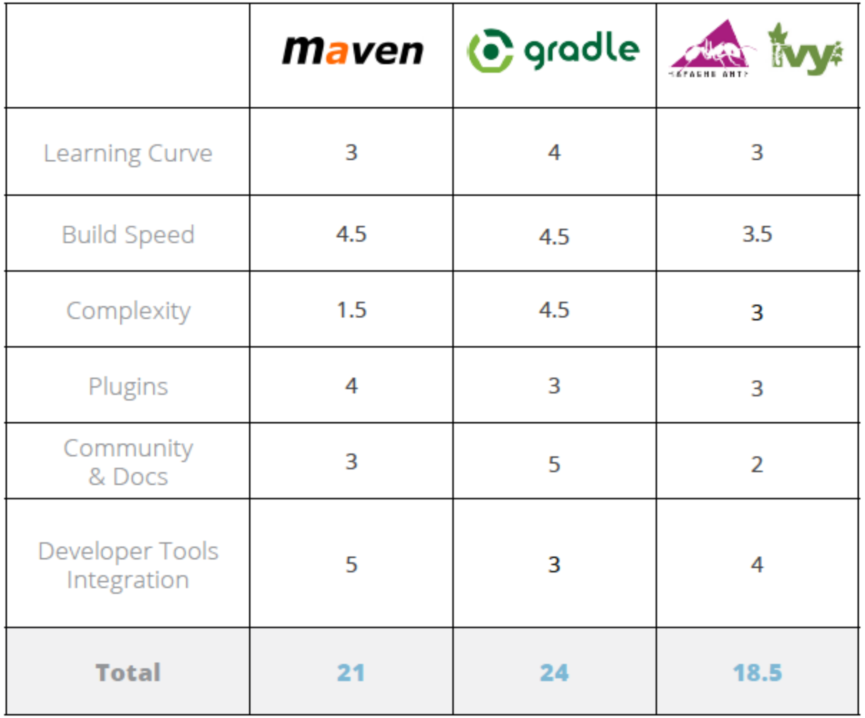
- ➢ Doing a clean build with tests.



	maven	gradle	APACHE ANT ivy
Command	rm -rf ~/.m2/repository && time mvn clean package	rm -rf ~/.m2/repository && rm -rf ~/.gradle/caches/ && time gradle clean build --daemon	rm -rf ~/.ivy2/cache/ && time ant clean war test
Time - Run 1 *(seconds)*	41.393	35.412	136
Time - Run 2 *(seconds)*	37.418	33.402	133
Time - Run 3 *(seconds)*	36.797	30.548	137
Time - Run 4 *(seconds)*	42.656	30.336	141
Time - Run 5 *(seconds)*	39.637	35.369	129
Average (min /max omitted)	**39.483**	**33.106**	**135.333**

Source: Rebellabs, Java Build Tools

Rebellabs comparison of build tools – final results



	maven	gradle	APACHE ANT ivy
Learning Curve	3	4	3
Build Speed	4.5	4.5	3.5
Complexity	1.5	4.5	3
Plugins	4	3	3
Community & Docs	3	5	2
Developer Tools Integration	5	3	4
Total	**21**	**24**	**18.5**

Source: Rebellabs, Java Build Tools

- For specific environments (e.g. enterprise development) Maven could be the better choice
- Details see report.

Overview

- Build automation
  - Motivation
  - Tool examples
    - Ant
    - Maven
    - Gradle
- Continuous Integration
  - Motivation
  - Tool examples
    - Jenkins
    - Cloud.

CI - Starting point

- Software projects are developed in a team
- Growing size and complexity
  - ... makes the management and the build of the software difficult
- Integration errors due to the interaction of the programmers
  - Phenomenon: „It works on my machine!“
- Finding integration errors late
  - … makes their correction costly.

CI - Strategy to avoid these problems

- At a regular frequency (ideally at every commit), the system is …
  - Integrated
    - All changes up until that point are combined into the project
  - Built
    - The code is compiled into an executable or package
  - Tested
    - Automated test suites are run
  - Archived
    - Versioned and stored so it can be distributed as is, if desired
  - Deployed
    - Loaded onto a system where the developers can interact with it

- → Continuous Integration.

CI – Where it all began

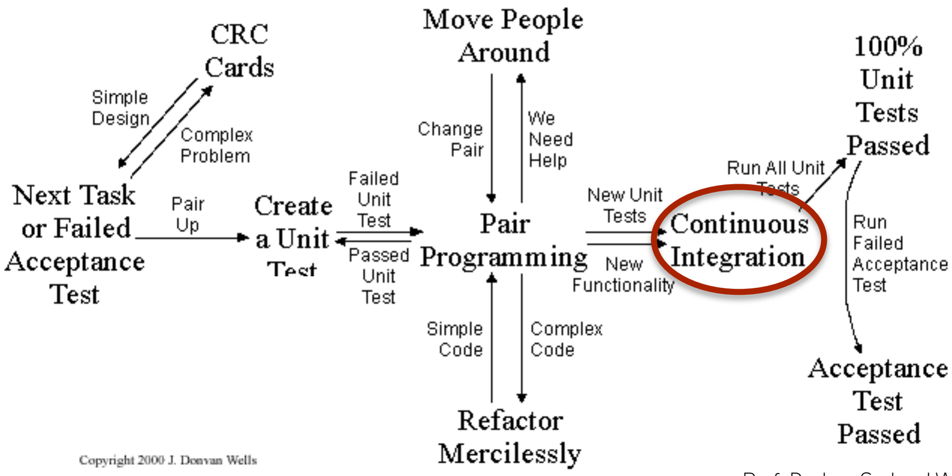
Continuous integration is a software development practice where members of a team integrate their work frequently, usually each person integrates at least daily – leading to multiple iterations per day.

Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible. Many teams find that this approach leads to significant reduced integration problems and allows a team to develop cohesive software more rapidly.

Martin Fowler, 2006
(original text from 2000)

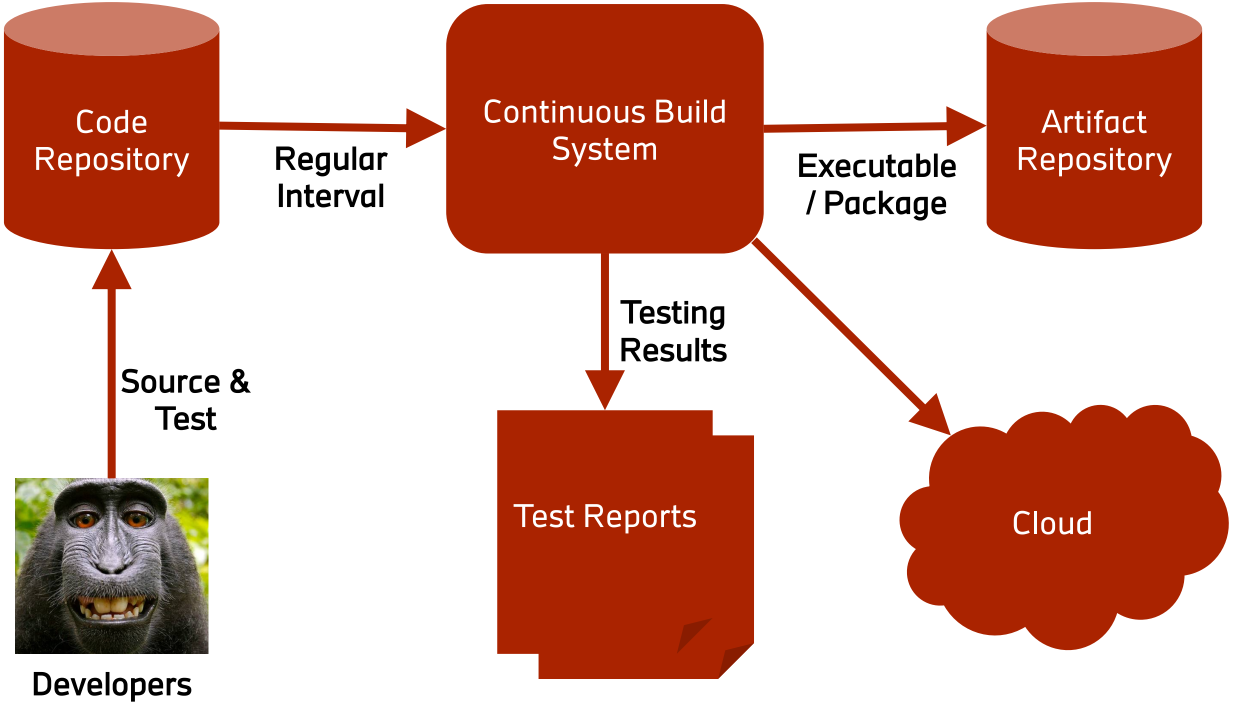
Example: CI in the context of XP

- XP rule: Integrate often
  - Continuous integration often avoids diverging or fragmented development efforts, where developers are not communicating with each other about what can be re-used, or what could be shared
    - Everyone needs to work with the latest version
    - Changes should not be made to obsolete code causing integration headaches
  - Continuous integration avoids or detects compatibility problems early
    - That is, if you integrate throughout the project in small amounts you will not find your self trying to integrate the system for weeks at the project's end.



Copyright 2000 J. Donvan Wells

CI - Workflow


Code Repository
Regular Interval
Continuous Build System
Executable / Package
Artifact Repository
Source & Test
Testing Results
Test Reports
Cloud
Developers

CI – Benefits & Tools

- Benefits
  - Immediate bug detection
  - No integration step in the lifecycle
  - A deployable system at any given point
  - Record of evolution of the project

> → Continuous integration is a necessity on complex projects due to the benefits it provides regarding early detection of problems
>
> → A good continuous build system should be flexible enough to fit into pre-existing development environments and provide all the features a team expects from such a system

- Tools
  - Jenkins (Hudson)
  - Bamboo
  - Cruise Control.

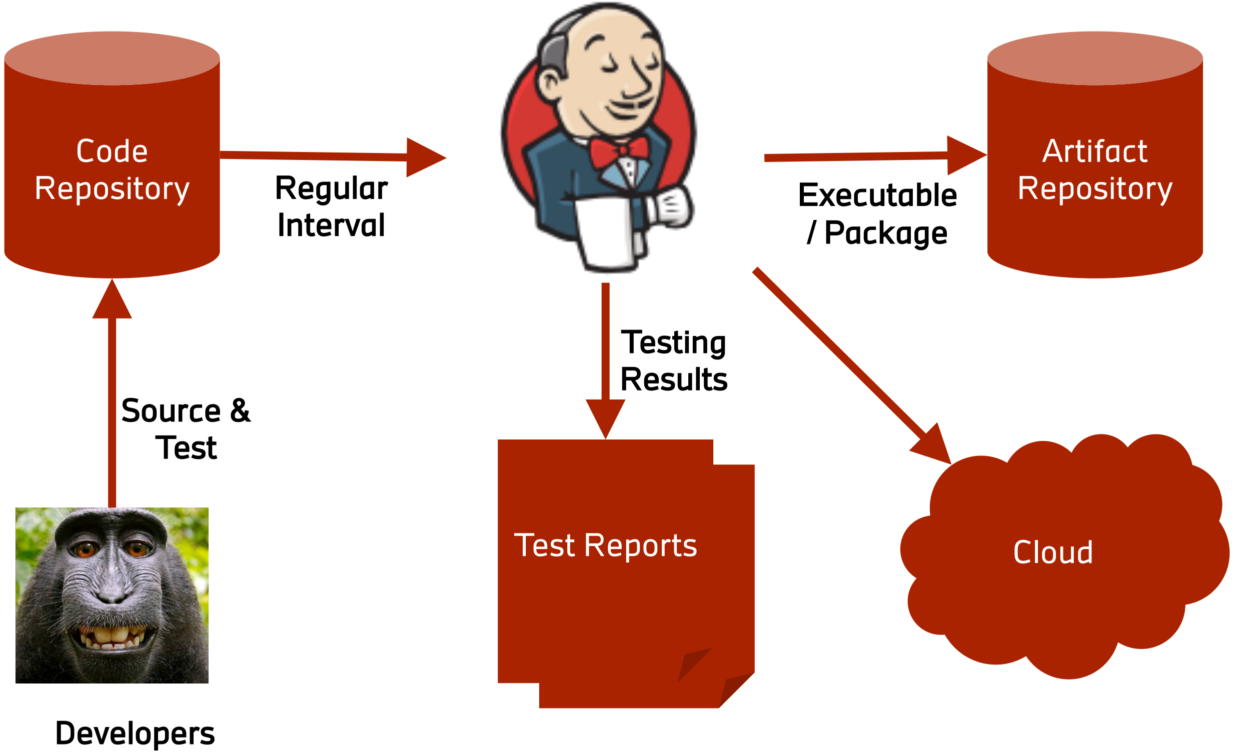
Overview

- Build automation
  - Motivation
  - Tool examples
    - Ant
    - Maven
    - Gradle
- Continuous Integration
  - Motivation
  - Tool examples
    - Jenkins
    - Cloud.

Jenkins - a Hudson fork

- Branched from Hudson
  - Because of differences on where to host the Hudson-Project there was a fork of Hudson in early 2011
  - Also a common name for butlers
- Java based Continuous Build System
- Runs in servlet container
  - Glassfish, Tomcat
- Supported by over 400 plugins
  - SCM, Testing, Notifications, Reporting, Artifact Saving, Triggers, External Integration
- Under development since 2005
  - Today Jenkins is the leading Open Source CI software
- http://jenkins-ci.org/

CI - Workflow


Code Repository
Regular Interval
Executable / Package
Artifact Repository
Testing Results
Source & Test
Test Reports
Cloud
Developers

Overview

- Flexibility
  - Jenkins is a highly configurable system by itself
  - The additional community developed plugins provide even more flexibility
  - By combining Jenkins with Ant, Gradle, or other Build Automation tools, the possibilities are limitless

- Free / OSS and award winning
  - See: https://jenkins.io/awards/
  - Jenkins is released under the MIT License

- What can Jenkins do?
  - Generate test reports
  - Integrate with many different Version Control Systems
  - Push to various artefact repositories
  - Deploys directly to production or test environments
  - Notify stakeholders of build status
  - …and much more.

How Jenkins works - Setup

- When setting up a project in Jenkins, out of the box you have the following general options:
  - Associating with a version control server
  - Triggering builds
  - Polling, Periodic, Building based on other projects
  - Execution of shell scripts, bash scripts, Ant targets, and Maven targets
  - Artefact archival
  - Publish JUnit test results and Javadocs
  - Email notifications
- As stated earlier, plugins expand the functionality even further.

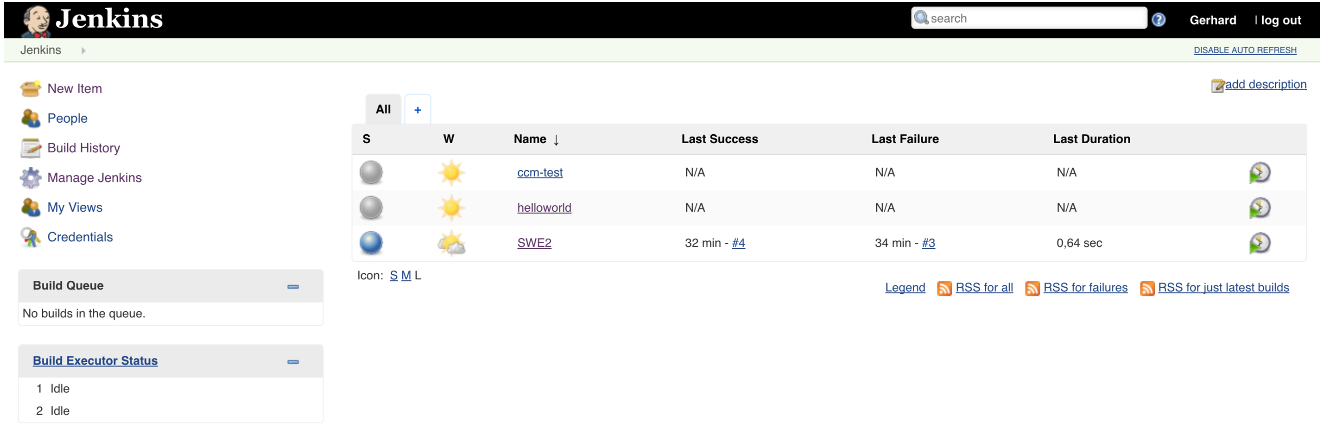
How Jenkins works - Building

- Once a project is successfully created in Jenkins, all future builds are automatic
- Building
  - Jenkins executes the build in an executer
    - By default, Jenkins gives one executer per core on the build server
  - Jenkins also has the concept of slave build servers
    - Useful for building on different architectures
    - Distribution of load.

How Jenkins works - Reporting

- Jenkins comes with basic reporting features
  - Keeping track of build status
    - Last success and failure
    - “Weather” – Build trend
- These can be greatly enhanced with the use of pre-build plugins
  - Unit test coverage
  - Test result trending
  - Findbugs, Checkstyle, PMD.

Jenkins by example – Main page



S	W	Name ↓	Last Success	Last Failure	Last Duration
●	☀	ccm-test	N/A	N/A	N/A
●	☀	helloworld	N/A	N/A	N/A
●	⛅	SWE2	32 min - #4	34 min - #3	0,64 sec

Icon: S M L

Legend RSS for all RSS for failures RSS for just latest builds

Build Queue

No builds in the queue.

Build Executor Status

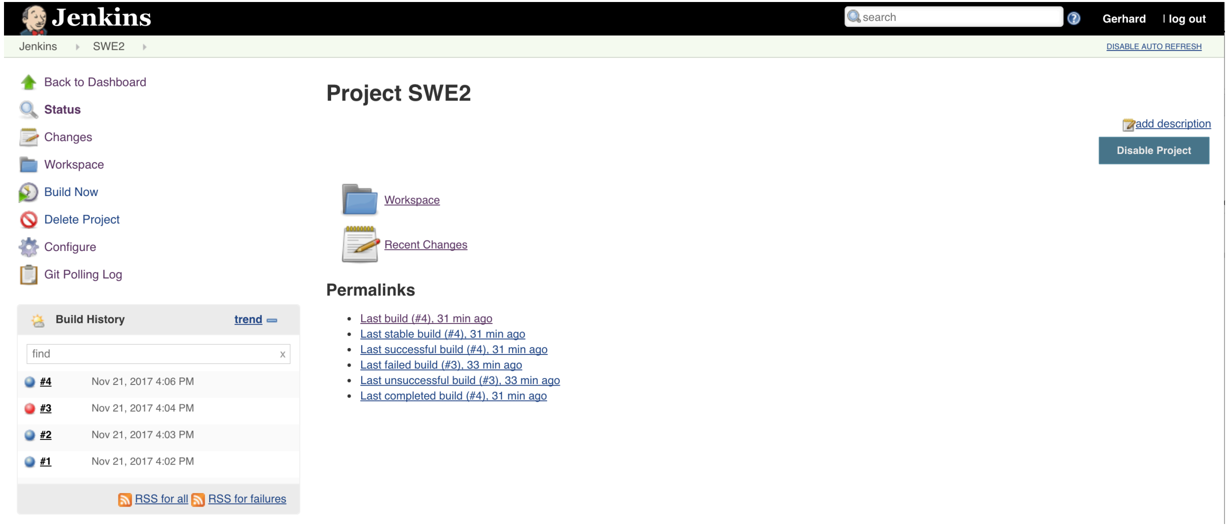
1 Idle

2 Idle

- The main page provides a summary of the projects
- Quick view of
  - What’s building (“No builds in the queue”)
  - Build Executor Status (both “Idle”)
  - Status of the projects.

Jenkins by example – Project status

- Project status pages provide more details about a given project
  - The status of the last several builds
  - Charting (depending on plugins)
  - Dependencies.



Project SWE2

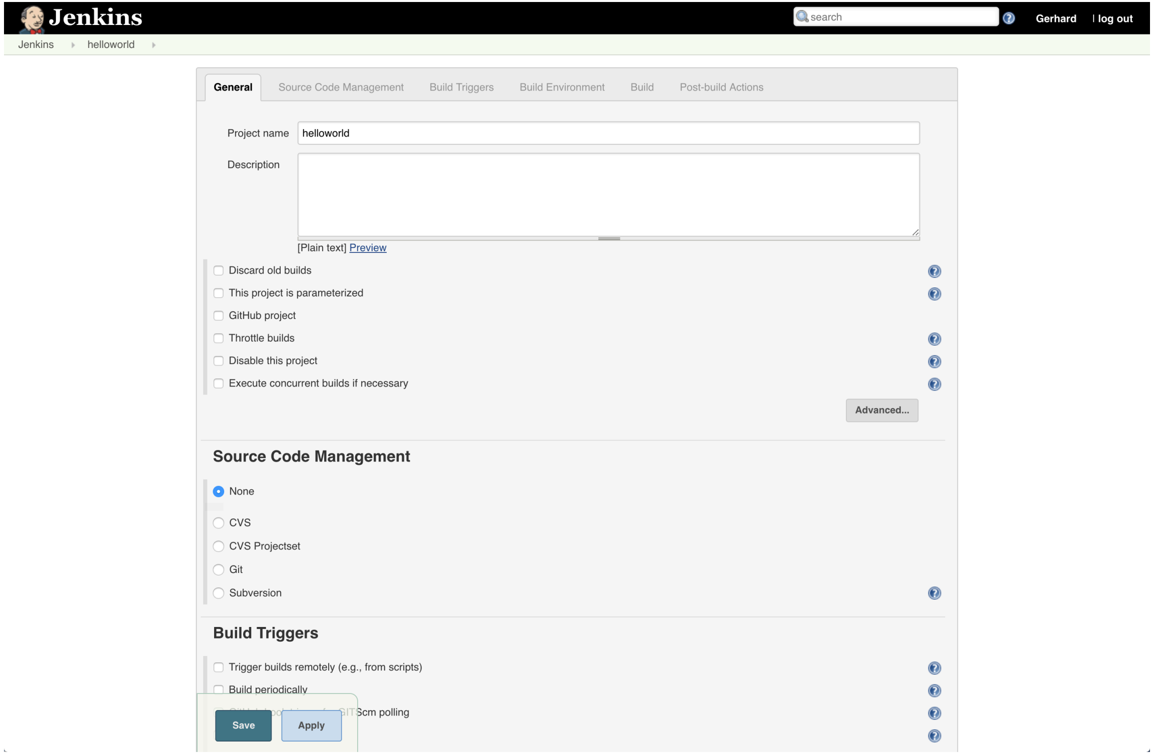
Build History

#4	Nov 21, 2017 4:06 PM
#3	Nov 21, 2017 4:04 PM
#2	Nov 21, 2017 4:03 PM
#1	Nov 21, 2017 4:02 PM

Permalinks

- Last build (#4), 31 min ago
- Last stable build (#4), 31 min ago
- Last successful build (#4), 31 min ago
- Last failed build (#3), 33 min ago
- Last unsuccessful build (#3), 33 min ago
- Last completed build (#4), 31 min ago

Jenkins by example – New project



Project name: helloworld

Description:

[Plain text] Preview

□ Discard old builds

□ This project is parameterized

□ GitHub project

□ Throttle builds

□ Disable this project

□ Execute concurrent builds if necessary

Source Code Management

● None

○ CVS

○ CVS Projectset

○ Git

○ Subversion

Build Triggers

□ Trigger builds remotely (e.g., from scripts)

□ Build periodically

□ GitHub hook trigger for GITScm polling

Enhancing Jenkins

- Jenkins plugin system can enable a wide range of features including (but certainly not limited to)
  - SCM
    - Mercurial, Git, Subversion
  - Testing
    - Selenium, Windmill, TestLink
  - Notifications
    - IRC, Twitter, Jabber
  - Reporting
    - Doxygen, PMD, Findbugs
  - Artifact Saving
    - Artifactory, Amazon S3, SCP
  - Triggers
    - Jabber, Directory Watchers
  - External Integration
    - GitHub, Bugzilla, JIRA
  - And most importantly – The CI Game (see next page).

Running Jenkins yourself

- Jenkins is packaged as a WAR, so you can drop it into whichever servlet container you prefer to use
- Jenkins comes pre-packaged with a servlet if you just want a lightweight implementation
- Native/Supported packages exist for
  - Windows, Ubuntu/Debian, Redhat/Fedora/CentOS, Mac OSX, …

Putting it all together

- While an integral part of a CI system, Jenkins is by no means the only component
  - In order for a CI system to function, a common repository for the codebase needs to exist
  - A database of artefacts needs to exist, so deliveries can be made at past iterations
  - The last step in a CI process is the deployment of the components built
- … and none of this matters if the developers don't use the system; procedures need to ensure the system is used as intended

- Jenkins, a continuous build system, can be an integral part of any continuous integration system due to it's core feature set and extensibility through a plugin system.

Overview

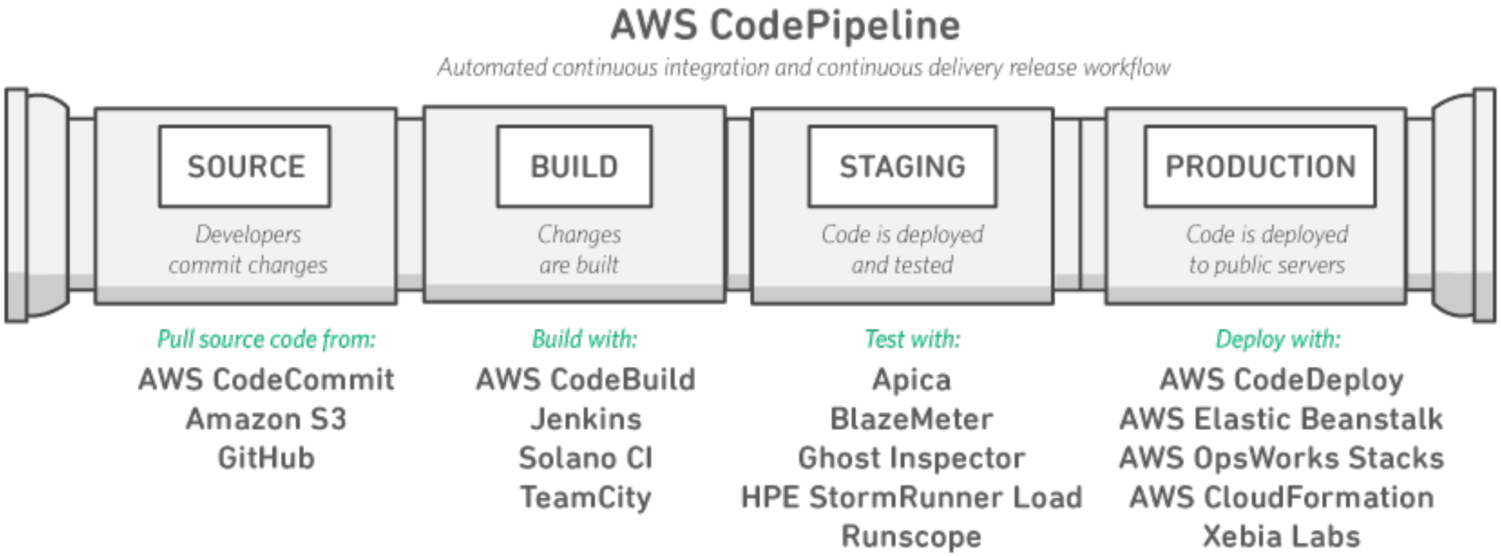
- Build automation
  - Motivation
  - Tool examples
    - Ant
    - Maven
    - Gradle
- Continuous Integration
  - Motivation
  - Tool examples
    - Jenkins
    - Cloud.

CI in the cloud

- Problem
  - Integrating a large software-system temporarily needs a huge amount of computing power
    - In between two builds this system is idle
  - Huge costs, to achieve short integration times
- Solution
  - Run continuous integration in the cloud
  - Only pay for computing power when needed
    - No idle systems. No waste of computing power
  - Tools that are used inside the build pipeline (scm, build, test, deployment) usually can be chosen.

CI in the cloud

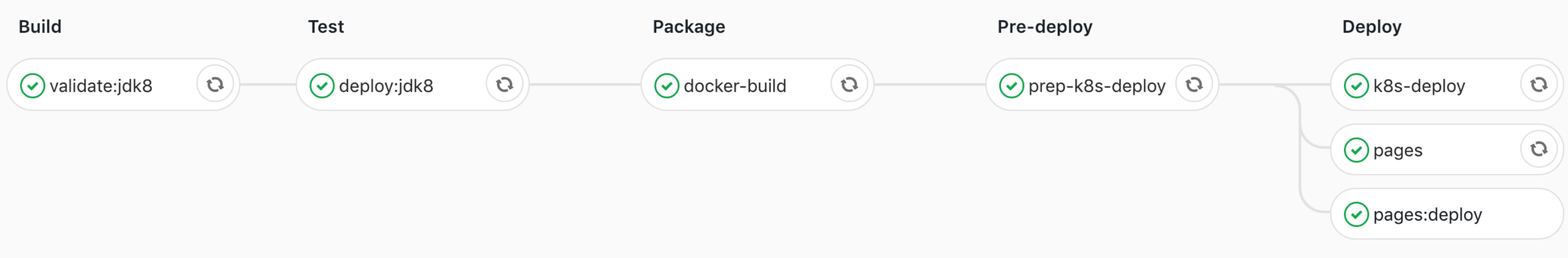
- ➢ Example: AWS CodePipeline



Source:
https://aws.amazon.com/de/codepipeline/

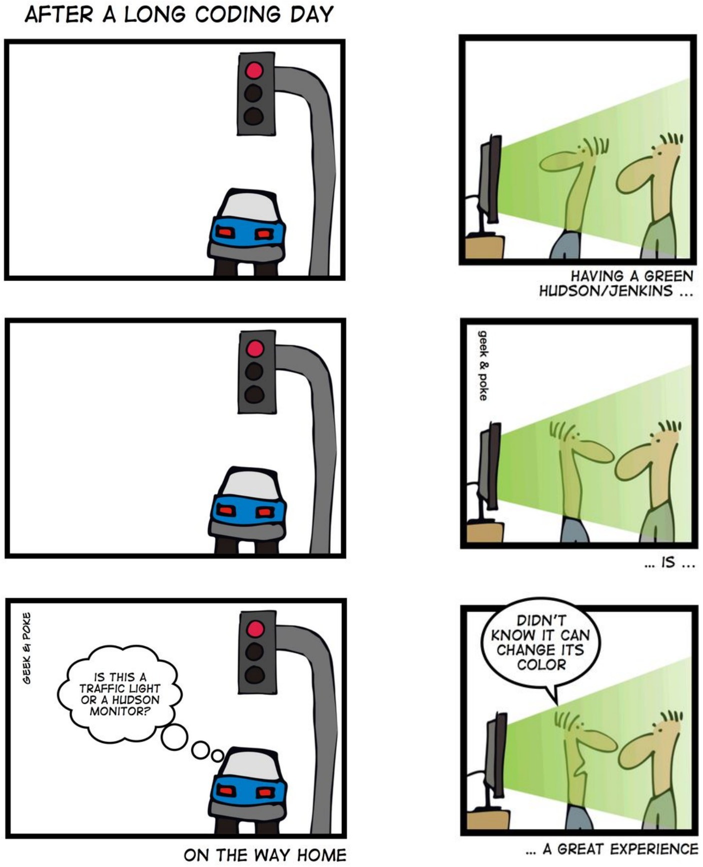
CI in the cloud

- Example: Gitlab DevOps pipeline



  - Build: validate environment
  - Test: run unit-tests
  - Package: build Docker-image
  - Pre-Deploy: check Kubernetes-environment
  - Deploy: deploy Docker-image on Kubernetes, create project-pages.

Typical effects when using Hudson/Jenkins


AFTER A LONG CODING DAY
ON THE WAY HOME
IS THIS A TRAFFIC LIGHT OR A HUDSON MONITOR?
GEEK & POKE
HAVING A GREEN HUDSON/JENKINS ...
... IS ...
geek & poke
DIDN'T KNOW IT CAN CHANGE ITS COLOR
... A GREAT EXPERIENCE

Source:
GeekAndPoke, 2011

References

- Ant Web Pages
  - http://ant.apache.org/
  - http://ant.apache.org/resources.html
- Maven
  - http://maven.apache.org/
- Gradle
  - https://gradle.org/
- Cruise Control
  - http://martinfowler.com/articles/continuousIntegration.html
  - http://cruisecontrol.sourceforge.net/
  - http://www.cruisecontrol.com/
- Hudson CI
  - http://hudson-ci.org/
- Jenkins CI
  - http://jenkins-ci.org/