Gute BUILDung

Lifecycle-Management mit der Jenkins Job DSL

Marcel Birkner

Wer möchte nicht wertvolle Zeit sparen? Dieser Artikel ist ein Erfahrungsbericht, wie uns das „Jenkins Job DSL Plugin" sehr viel Arbeit und Zeit bei der Verwaltung von Jobs erspart hat und zukünftig weiter ersparen wird.

Automatisierung des Softwareentwicklungsprozesses

Continuous Integration (CI)-Server für die Automatisierung der Build- und Deployment-Prozesse gehören in den meisten Unternehmen zum Entwicklungsprozess. Oft wird dabei Jenkins als CI-Server eingesetzt. Abbildung 1 zeigt den Lebenszyklus eines typischen Jenkins-Jobs.

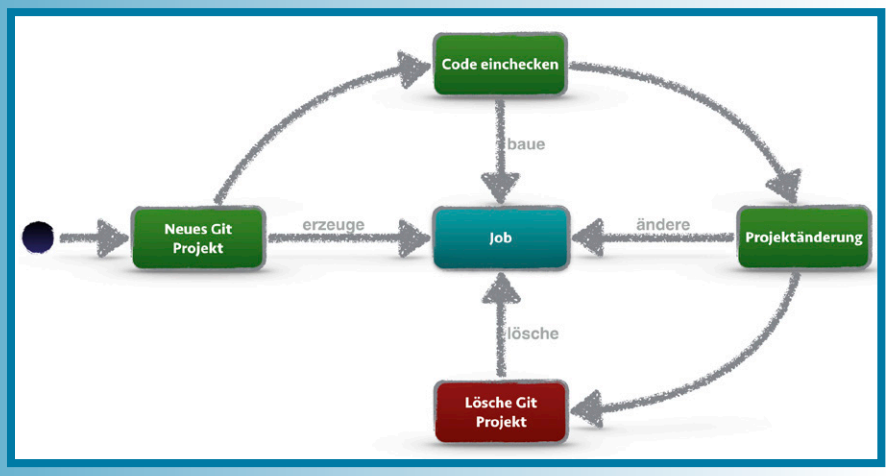

Abb. 1: Jenkins-Job-Lebenszyklus

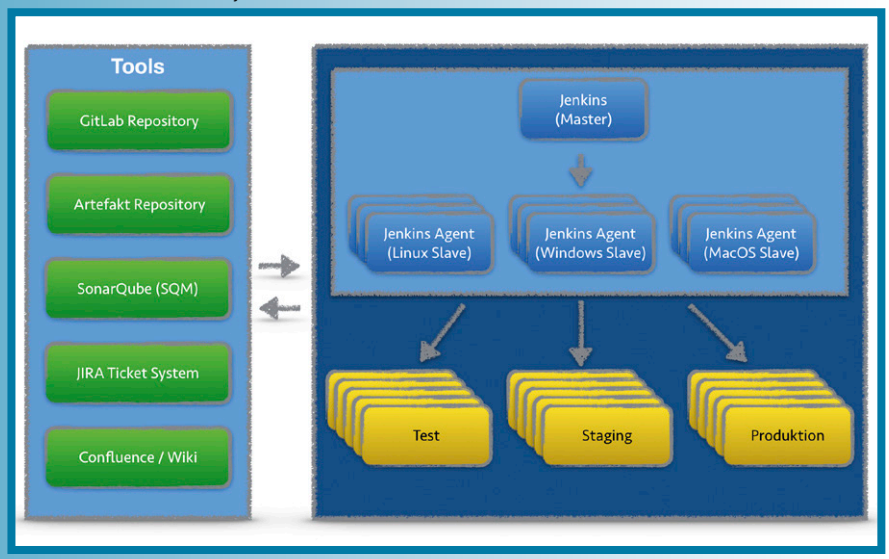

Abb. 2: Typische Softwareentwicklungstools

Der Lebenszyklus eines CI/CD-Jobs sieht wie folgt aus: Beim Start eines neuen Softwareprojektes wird ein Git-Repository angelegt und eine Reihe von Build-, Test- und Deployment-Jobs erstellt. Jede Änderung am Quellcode startet den CI-Job.

Nachdem die CI- und Deployment-Jobs eine Weile ohne Probleme liefen, stehen irgendwann Upgrades von diversen Komponenten an. Zum Beispiel muss bei einem Update von Java 7 auf Java 8 der Build-Job angepasst werden, da das JDK 8 zum Compilieren benötigt wird. Manchmal stehen komplexere Umstellungen an, wie zum Beispiel die Umstellung des Build-Systems von Ant auf Maven. In diesen Fällen stehen größere Änderungen für die Jenkins-Jobs an.

Nach ein paar Jahren kommt die Zeit, dass ein Projekt ausläuft und nicht mehr benötigt wird. Alle relevanten Jobs können dann gelöscht werden. Die obsoleten Jobs sollten regelmäßig entfernt werden, damit die CI-Infrastruktur sauber bleibt und keine unnötigen Ressourcen gebunden werden.

Jenkins ist ein großartiges Tool für die Automatisierung des Softwareentwicklungsprozesses. Es ist sehr einfach, einen Jenkins-CI-Server aufzusetzen, und nachdem die ersten Jobs automatisiert wurden, möchte man Jenkins nicht mehr missen.

In den meisten Unternehmen sehen die Entwicklungs- und CI-Umgebung nach einer Weile so aus, wie in Abbildung 2 zu sehen. Es gibt die typischen Entwicklungstools für die Quellcodeverwaltung, ein Artefakt-Repository, ein Ticket-System, ein Wiki und eventuell SonarQube für die statischen Codeanalysen. Zusätzlich gibt es einen oder mehrere Jenkins-Instanzen mit diversen Slaves, die unterschiedliche Aufgaben erfüllen.

Im Jenkins werden sehr schnell eine Vielzahl unterschiedlicher Jobs eingerichtet. Die Jobs wurden entweder manuell erstellt, von Vorlagen kopiert, über das REST-API oder die Client-JAR erstellt. Es gibt verschiedene Wege, Jobs zu erstellen, die alle zuverlässig funktionieren.

Die Einsatzszenarien für Jenkins sind vielfällig. Eine Beschränkung auf Programmiersprachen gibt es nicht und man bekommt schnell das Gefühl, alles automatisieren zu können. Aktuell gibt es über 1000 Plug-ins, welche Jenkins mit diversen anderen Tools und Frameworks integrieren.

Auswirkungen von Änderungen

Es könnte alles so schön sein, doch dann kommen die ersten Veränderungen. Im Folgenden liste ich eine Reihe von Gründen für typische Veränderungen in Softwareprojekten auf und die daraus resultierenden Anpassungen der Jenkins-Jobs. Ich denke, jeder wird das ein oder andere aus seinem Umfeld wiedererkennen.

Anpassung 1

Zuerst der Klassiker: Projekte entscheiden sich, die Java-Version zu aktualisieren.

Folgen:
- Das JDK muss in den Build-Jobs aktualisiert werden.

Anpassung 2
Das Build-Tool wird ersetzt.
Folgen:
- Build-Jobs müssen auf das neue Tool umgestellt werden.
- Deployment- und Test-Skripte müssen angepasst werden, da sich der Deployment-Prozess ändert.

Anpassung 3
Wechsel des Version-Control-Systems (VCS).
Folgen:
- Alle Jobs zu den Projekten müssen auf das neue VCS umgestellt werden.
- Bei Big-Bang-Umstellungen oft alle Jobs innerhalb sehr kurzer Zeit.

Anpassung 4
Der verwendete Applikationsserver bekommt ein Upgrade.
Folgen:
- Die automatisierten Deployment-Skripte müssen angepasst werden.
- Job-Parameter können sich ändern.
- Konfigurationsvorlagen müssen aktualisiert werden.

Anpassung 5
Im Unternehmen findet eine Umstrukturierungen statt, wodurch sich Projekt- und Gruppennamen ändern.
Folgen:
- Git-Projekte/Gruppennamen ändern sich.
- Jenkins-Job-Namen müssen aktualisiert werden.
- SCM-Einstellungen in den Jobs müssen an die neuen Namen angepasst werden.

Anpassung 6
Projekte werden in Git umbenannt, umstrukturiert oder zusammengelegt.
Folgen:
- Auch hier müssen die Job-Namen und SCM-Einstellungen aktualisiert werden.

Anpassung 7
Projekte wollen den CI-Build auf speziellen Branches ausführen und nicht mehr auf dem Master-Branch.
Folgen:
- SCM-Einstellungen im Jenkins-Job müssen individuell angepasst werden.

Anpassung 8
Framework-Änderungen.
Folgen:
- Ihre Java-Projekte verwenden neuere Bibliotheken, die wiederum eine neuere JDK-Version benötigen.
- Hadoop benötigt zum Beispiel mindestens JDK 7.

Anpassung 9
Es werden vermehrt kleinere eigenständige Anwendungen anstelle von Monolithen gebaut.
Folgen:
- Hier müssen ständig neue Build-, Test-, Release- und Deployment-Jobs angelegt werden.

Anpassung 10
Projekte werden abgeschlossen oder ersetzt.
Folgen:
- Alle dem Projekt zugeordneten Jobs müssen aus dem CI-System gelöscht und aufgeräumt werden.

Sicherlich fallen Ihnen noch weitere Gründe ein, die im Laufe der Zeit eintreten können.

Jenkins Job DSL Plugin

Jetzt stellen Sie sich vor, Sie verwalten die CI-Umgebung für 140 Java-Projekte in Ihrem Unternehmen. Auf Ihrem Jenkins-Server laufen täglich Tausende Jobs und ein Großteil der Projekte durchläuft eine der diversen Änderungen, die ich oben aufgelistet habe. Manche Projekte migrieren von Java 6 nach Java 7, von Ant nach Maven, von Tomcat 6 nach Tomcat 7 oder durchlaufen gerade organisatorische Änderungen, sodass alle Projekte neuen Abteilungen zugewiesen werden und neue Namen bekommen. Es gibt für das CI-Team immer etwas zu tun.

Zusätzlich haben wir für Testzwecke eine exakte Kopie aller Entwicklungstools in einer eigenen Testumgebung. Daher müssen die Jobs nicht nur auf dem produktiven Jenkins aktualisiert werden, sondern auch auf den Testinstanzen. In der Testumgebung werden Upgrades simuliert oder neue Plug-ins auf ihre Funktionalität hin getestet.

Am Anfang haben wir die Änderungen manuell durchgeführt, da die Anzahl der Projekte und Jobs übersichtlich war. Dafür haben wir die Jenkins CLI intensiv genutzt, um mehrere Jobs gleichzeitig über Groovy-Skripte ändern zu können [JenkinsCLI]. Zusätzlich haben wir Shell-Skripte geschrieben, zusammen mit der Jenkins-Client-JAR, um Jobs über XML-Vorlagen anzulegen. Mit der Zeit wurden diese Skripte jedoch immer komplexer und unübersichtlicher.

Zu diesem Zeitpunkt stieß ich auf zwei Blog-Einträge meiner Kollegen Dennis Schulte [Schu15] und Daniel Reuter [Reut15], die über die Vorteile des „Jenkins Job DSL Plugins“ [JobDSL] geschrieben haben.

Erste Schritte

Hier ist ein einfaches Beispiel des Job DSL Plugins. Die Job-DSL ist eine domänenspezifische Sprache, die in Groovy geschrieben ist. Nach der Installation des Job DSL Plugins erstellt man einen Seed-Job. Im nächsten Schritt wird ein Groovy-Skript geschrieben, welches die Job-DSL verwendet. Die vollständige Programmierschnittstelle können Sie im Job DSL API Viewer durchsuchen [JobDSLAPI]. Ein einfaches Skript könnte wie in Code-Snippet 1 aussehen. Alle Snippets finden Sie auch unter [SRC].

```
def gitUrl = "git@github.com:marcelbirkner/jersey-rest-server.git"
job('jersey-rest-server') {
  scm {
    git {
      remote {
        url(gitUrl)
      }
      createTag(false)
    }
  }
  triggers {
    scm('H/15 * * * *')
  }
  steps {
    maven {
      goals('-e clean install')
      mavenOpts('-Xms256m')
      mavenOpts('-Xmx512m')
      properties skipTests: true
```

```
      mavenInstallation('Maven 3.3.3')
    }
  }
}
```

Code-Snippet 1: jobDslExample.groovy

Dieses Skript erzeugt einen Freestyle-Jenkins-Job, der ein GitHub-Repository auscheckt und alle 15 Minuten prüft, ob es SCM-Änderungen gibt. Bei Änderungen wird ein Maven-Build mit `mvn clean install`, inkl. verschiedener `MAVEN_OPTS`, ausgeführt. Die Syntax der Job-DSL ist sehr sprechend und im Vergleich zur manuellen Konfiguration sehr übersichtlich. Wer die Job-DSL einmal selbst ausprobieren möchte, kann dafür den Playground auf Heroku nutzen [JobDSLPG].

Das DSL-API unterstützt die Erzeugung verschiedener Job-Typen, Ansichten, Ordner und vieler weiterer nützlicher Dinge. Bisher haben über 100 Entwickler zur Weiterentwicklung des Plug-ins beigetragen. Jeden Monat gibt es ein Release mit neuen Features. Ein weiterer Pluspunkt liegt darin, dass in den Seed-Skripten die Vorteile von Groovy zum Tragen kommen. Sie können HTTP-Requests abschicken, JSON und XML verarbeiten, Informationen aus Datenbanken lesen sowie vieles mehr.

Bevor wir auf die Job-DSL gewechselt sind, haben wir ein paar Tests durchgeführt, inwieweit wir alle benötigten Informationen aus bestehenden Systemen lesen konnten. Hier ein Ausschnitt unserer Tests.

GitLab-REST-API mit Groovy verarbeiten

Als zentrales Git-Repository setzen wir GitLab ein. GitLab bietet ein REST-API an, über das alle Repositories und Projekt-Details ausgelesen werden können. Das REST-API ist mit einem privaten Token gesichert [GitLabAPI]. Dieser ist im Benutzerprofil hinterlegt. Mit Hilfe des Groovy *JsonSlurpers* ist es sehr einfach, das Antwort-JSON zu parsen und in ein Objekt umzuwandeln (Code-Snippet 2).

```
// GitLab Settings
def gitUrl = 'http://git/api/v3'
def privateToken = 'xYxYxYxYxYxYxYxYxYxY'
def currentPage = 1
def projectsPerPage = 100

// Reading projects from GitLab REST API
def restApiUrl = "${gitUrl}/projects/all?" +
  "page=${currentPage}&" +
  "per_page=${projectsPerPage}&" +
  "private_token=${privateToken}"
def projectList = new URL(restApiUrl)
def reader = projectList.newReader()
def projects = new groovy.json.JsonSlurper().parse(reader)
projects.each {
  println it.name
}
```

Code-Snippet 2: accessGitLabRESTApi.groovy

Oracle-Datenbank mit Groovy auslesen

Für das Konfigurationsmanagement wird bei unserem Kunden ein Tool eingesetzt, welches die Daten in einer Oracle-Datenbank speichert. Alle Projekte verwenden das gleiche Namensschema, sodass die Git-Repository-Namen identisch mit den Anwendungsnamen im Konfigurationsmanagementtool sind. Daher können wir alle Anwendungsinformationen wie Build-Tool (Ant, Maven), Java-Version (6/7/8), Framework-Version, Deployment-Server (WebSphere, Tomcat, Batch, None) und viele weitere Details aus der Oracle-Datenbank lesen. Wir verwenden dazu die `groovy.sql.Sql`-Klasse, wie in Code-Snippet 3 zu sehen.

```
import groovy.sql.Sql

// Oracle DB Settings
def dbSchema = "schema"
def dbServer = "oracleserver"
def dbUser = "user"
def dbPassword = "password"
def dbDriver = "oracle.jdbc.driver.OracleDriver"
def dbUrl = "jdbc:oracle:thin:@${dbServer}:${dbSchema}"
sql = Sql.newInstance( dbUrl, dbUser, dbPassword, dbDriver )

// Read data
def selectStatement = "SELECT * FROM application_configuration" +
  "WHERE application_name = ${gitProjectName}"
def row = sql.firstRow(selectStatement)
println row
```

Code-Snippet 3: accessOracleDatabase.groovy

MySQL-Datenbank mit Groovy auslesen

Statistiken zu allen Projekten und Jobs speichern wir in einer MySQL-Datenbank. Auch das ist sehr leicht mit einem Groovy-Skript umzusetzen.

```
import groovy.sql.Sql

// MySQL DB Settings
def dbSchema = "schema"
def dbServer = "mysqlserver"
def dbUser = "user"
def dbPassword = "password"
def dbDriver = 'com.mysql.jdbc.Driver'
def dbUrl = 'jdbc:mysql:@' + dbServer + ':' + dbSchema
sql = Sql.newInstance( dbUrl, dbUser, dbPassword, dbDriver )
def insertStatement =
  "INSERT INTO table_name (column1, column2, column3)" +
  "VALUES (value1, value2, value 3)"
sql.execute insertStatement
```

Code-Snippet 4: accessMySqlDatabase.groovy

Erfahrungsbericht

Nachdem wir genau wussten, was mit dem Job DSL Plugin und Groovy alles möglich ist, begannen wir unsere nächsten Schritte zu planen.

Planungsphase

Abbildung 3 zeigt den Prozess, den wir letztendlich umgesetzt haben. Es gibt einen zentralen Seed-Job, der nachts ausgeführt wird, alle Jobs erzeugt und bestehende aktualisiert. Der Seed-Job kann auch öfters ausgeführt werden, da er laufende Jobs nicht verändert.

Schritt 1: Über das GitLab-REST-API erhält das Seed-Job-Groovy-Skript eine Liste aller Git Repositories.

Schritt 2: Jetzt iterieren wir über alle Git-Projekte, die nicht als archiviert gekennzeichnet sind.

Schritt 3: Wir prüfen alle Git-Projekte, ob es in der Konfigurationsmanagement-Datenbank Informationen gibt:

- werden Informationen gefunden, machen wir mit Schritt 4 weiter,
- wenn nicht, überspringen wir das Projekt.

Schritt 4: Abhängig von den Informationen erstellen wir Ant/Maven-Build-Jobs, Tomcat/WebSphere-Deploy-Jobs und alle weiteren Jobs, die notwendig sind.

Schritt 5: Nachdem alle Jobs erzeugt wurden, erstellen wir zusätzlich Jenkins-Ansichten, um die Jobs übersichtlich zu gruppieren. Projekte, die aus einer Kette von Build-, Deploy- und Akzeptanztest-Jobs bestehen, bekommen eine eigene

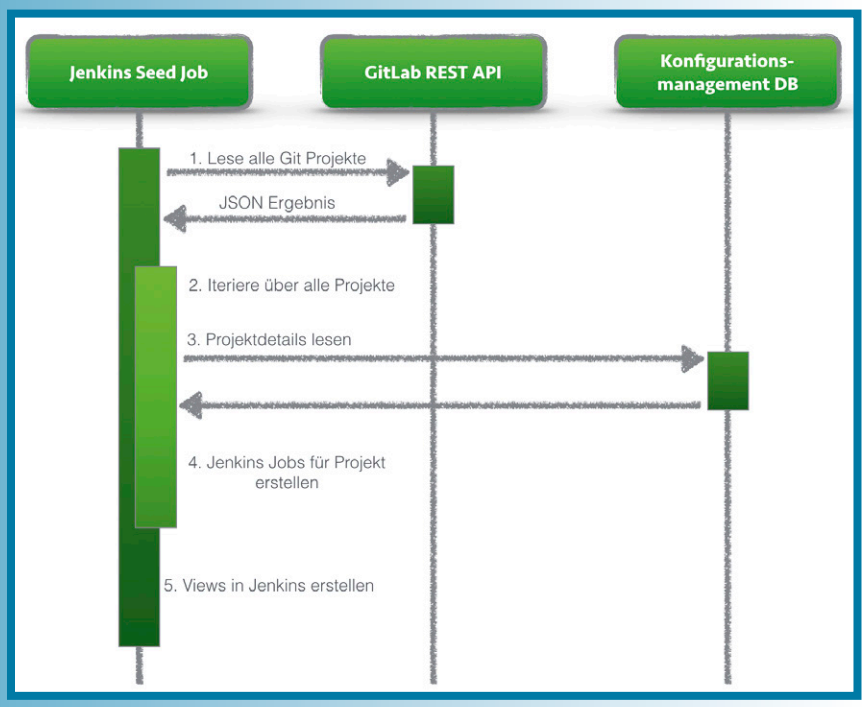

Abb. 3: Final implementierter Prozess

Systems wurde stark vereinfacht und wir mussten uns nicht mehr um diverse Shell-Skripte, Groovy-Skripte und Jenkins-Job-Vorlagen kümmern. Es gibt jetzt nur noch ein Groovy-Seed-Skript, das alles erzeugt und das versioniert in einem Git-Repository gespeichert wird. Zusätzlich benötigte Informationen werden aus bestehenden Systemen gelesen und dort von den Projektteams und Entwicklern gepflegt. Alle Jobs und Projekte sind am Morgen immer aktuell. Die CI/CD-Testumgebung hat identische Jobs wie die Produktionsumgebung. Jobs von archivierten Git-Projekten werden automatisch entfernt. Entwicklerteams können ihre Projekte jederzeit umstellen, ohne von den CI-Infrastruktur-Entwicklern abhängig zu sein.

Wenn Sie ebenfalls einen großen Zoo an Projekten und Programmiersprachen haben, dann könnte dieses Plug-in Ihr Leben einfacher gestalten. Für uns war der Umstieg eine große Erleichterung, alle Projekte müssen sich an ein paar einfache Konventionen halten, so lässt sich der Softwareentwicklungs- und Release-Prozess so gut wie möglich automatisieren.

Build-Pipeline-Ansicht. Zuletzt schreiben wir die gesammelten Statistikdaten in eine MySQL-Datenbank.

Status quo

Hier sind noch ein paar Zahlen aus dem Projekt, in dem wir dieses Verfahren eingesetzt haben:

- über 140 Java-Projekte,
- über 700 automatisch generierte Build-, Test-, Deploy- und Release-Jobs,
- über 1000 Builds pro Tag,
- die komplette CI/CD-Umgebung läuft auf virtuellen XEN-CentOS-Servern,
- der Seed-Job benötigt 2 bis 3 Minuten, um alle Jobs zu erzeugen beziehungsweise zu aktualisieren,
- die „alte" Lösung hat 15 bis 30 Minuten benötigt, um alle Jobs zu erzeugen.

Docker-Beispiel

Wenn Sie interessiert sind, das Jenkins Job DSL Plugin zu testen, können Sie dies recht einfach mit dem folgenden Docker-Container machen. Ich habe ein Git-Repository angelegt, in dem Sie einen fertig konfigurierten Jenkins inkl. Job-DSL-Beispielen starten und testen können, siehe Code-Snippet 5.

```
#!/bin/sh
git clone git@github.com:marcelbirkner/docker-jenkins-job-dsl.git
cd docker-jenkins-job-dsl
docker build -t docker-jenkins-job-dsl .
docker run -p=8080:8080 docker-jenkins-job-dsl
```

Code-Snippet 5: docker-jenkins-job-dsl.sh

Fazit

Im Nachhinein hat uns das Job DSL Plugin viel mehr Arbeit abgenommen, als wir erwartet hatten. Die Komplexität des

Weiterführende Links

[GitLabAPI] GitLab REST API,
`https://github.com/gitlabhq/gitlabhq/tree/master/doc/api`
[JenkinsCLI] Jenkins CLI, Command Line Interface,
`https://wiki.jenkins-ci.org/display/JENKINS/Jenkins+CLI`
[JobDSL] Job DSL Plugin – Jenkins,
`https://wiki.jenkins-ci.org/display/JENKINS/Job+DSL+Plugin`
[JobDSLAPI] Job DSL API Viewer – Jenkins,
`https://jenkinsci.github.io/job-dsl-plugin`
[JobDSLPG] Jenkins Job DSL Playground,
`http://job-dsl.herokuapp.com/`
[Reut15] D. Reuter, Generated Jenkins Jobs and automatic Branch merging for Feature branches,
`https://blog.codecentric.de/en/2015/04/generated-jenkins-jobs-and-automatic-branch-merging-for-feature-branches`
[Schu15] D. Schulte, Continuous Delivery for Microservices with Jenkins and the Job DSL Plugin, `https://blog.codecentric.de/en/2015/01/continuous-delivery-microservices-jenkins-job-dsl-plugin/`
[SRC] M. Birkners Gists, Code zum Artikel,
`https://gist.github.com/marcelbirkner/`

Marcel Birkner ist Software Consultant bei der codecentric AG. Sein Fokus liegt auf Java, Spring, Cloud-Technologien und allem rund um die Automatisierung von Softwareentwicklungsprozessen.
E-Mail: marcel.birkner@codecentric.de