Jenkins

Coding Continous Delivery – Grundlagen des Jenkins-Pipeline-Plug-ins

Johannes Schnatterer und Daniel Behrwind, Triology GmbH

Wer schon einmal eine Continuous-Delivery-Pipeline mit einem herkömmlichen CI-Tool durch Verketten einzelner Jobs und ohne direkten Pipeline-Support eingerichtet hat, der weiß, wie unübersichtlich ein solches Unterfangen sein kann. Diese Artikelserie zeigt, wie sich eine Pipeline mithilfe des Jenkins-Pipeline-Plug-ins an zentraler Stelle als Code definieren lässt. Im ersten Teil geht es um die Grundlagen und um praktische Tipps für den Einstieg.

Continuous Delivery hat sich im Umfeld agiler Software-Entwicklung als adäquates Vorgehen erwiesen, qualitativ hochwertige Software in kurzen Zyklen zuverlässig und wiederholbar zu veröffentlichen. Dabei durchlaufen Änderungen an der Software eine Reihe von qualitätssichernden Schritten, bevor sie in Produktion gehen. Eine typische Continuous-Delivery-Pipeline könnte beispielsweise so aussehen:

- Checkout aus der Versionsverwaltung
- Build
- Unit-Tests
- Integrationstests
- Statische Code-Analyse
- Deployment in einer Staging-Umgebung
- Funktionale und/oder manuelle Tests
- Deployment in Produktion

Dabei ist es essenziell, all diese Schritte zu automatisieren, was typischerweise mit Continuous-Integration-Servern wie Jenkins geschieht.

Zwar eignen sich herkömmliche Jenkins-Jobs gut, um einzelne Schritte einer Continuous-Delivery-Pipeline zu automatisieren. Da die Schritte aber aufeinander aufbauen, ist deren Reihenfolge zwingend einzuhalten. Mit herkömmlichen Jenkins-Jobs (oder anderen Continuous-Integration-Tools ohne direkten Pipeline-Support) kann das schnell unübersichtlich sein. Einzelne Jobs, oft angereichert um etliche Pre- und Post-Build-Schritte, werden verkettet. Dies führt dazu, dass man sich von Job zu Job hangeln muss, um zu verstehen, was passiert. Zusätzlich sind solche komplexen Konfigurationen weder test- noch versionierbar und müssen für jedes Projekt neu aufgesetzt werden.

Hier schafft das Jenkins-Pipeline-Plug-in Abhilfe. Es bietet die Möglichkeit, die gesamte Pipeline mithilfe einer Groovy-DSL an zentraler Stelle, einer versionierten Skript-Datei („Jenkinsfile"), als Code zu definieren. Dabei stehen dem Anwender zwei Varianten der DSL zur Auswahl: ein imperativer, tatsächlich eher geskripteter Stil (nachfolgend als „scripted Syntax" bezeichnet) und seit Februar 2017 auch eine deklarative Variante (nachfolgend „declarative Syntax" genannt).

Die deklarative Syntax ist dabei eine Teilmenge der „scripted"-Syntax und bietet mit einem vorgegebenen Aufbau und mehr beschreibenden Sprachelementen eine Grundstruktur (ähnlich einer „maven-pom"-Datei), die den Einstieg vereinfachen kann. Damit führt sie, auch wenn sie umfangreicher und unflexibler ist, zu Build-Skripten, die intuitiver verständlicher sind als solche, die in „scripted"-Syntax verfasst sind. Diese bietet dafür fast alle Freiheiten, die die Sprache Groovy mit sich bringt (Einschränkungen siehe *[1]*), erfordert aber gegebenenfalls auch eine tiefergehende Auseinandersetzung mit Groovy.

Die Entscheidung für die eine oder die andere Variante kann aktuell noch als Geschmacksfrage angesehen werden; es ist nicht absehbar, ob sich eine der beiden Richtungen durchsetzen und die andere letztlich verdrängen wird. Neuere offizielle Beispiele sind jedoch meist in der „declarative"-Syntax verfasst und es gibt einen visuellen Editor, der nur auf diese ausgelegt ist. Um einen direkten Vergleich zu bieten, sind die Beispiele in diesem Artikel in beiden Stilen formuliert.

Kernkonzepte

Die Beschreibung einer Build-Pipeline mit der Jenkins-Pipeline-DSL gliedert sich im Wesentlichen in Stages und Steps. Stages sind frei wählbare Gruppierungen der Schritte einer Pipeline. Die Punkte der

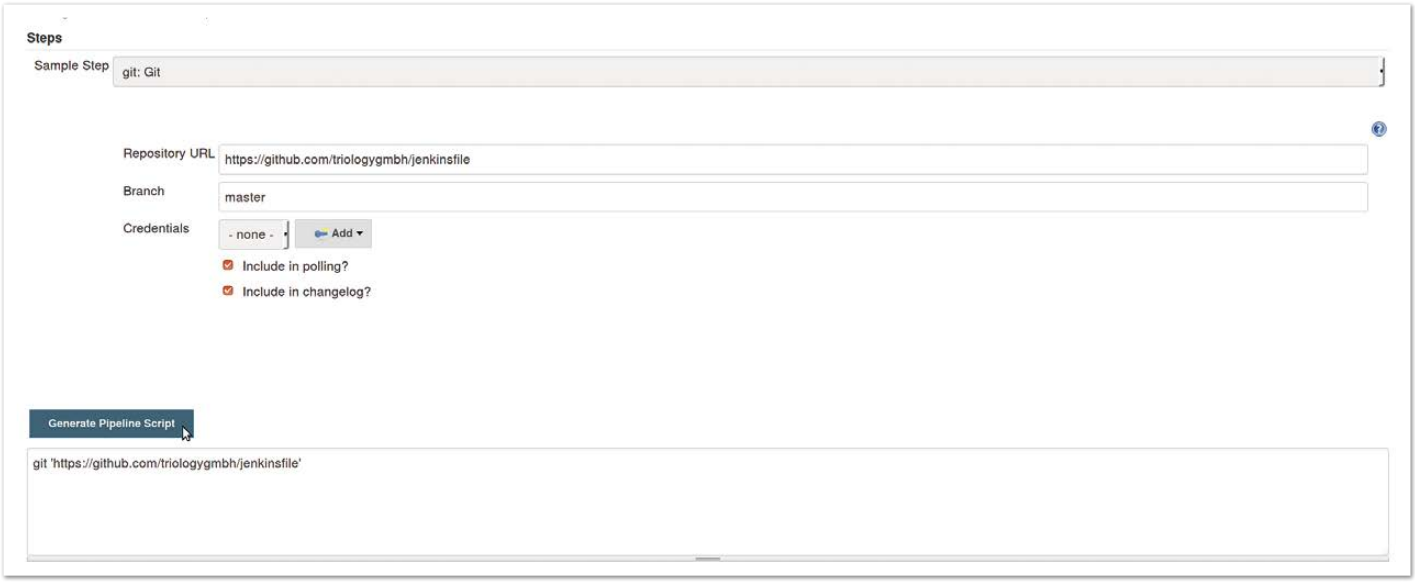

Abbildung 1: Der Snippet-Generator

```
pipeline {
  agent any

  tools {
     maven 'M3'
  }

  stages {
     stage('Checkout') {
        steps {
           git 'https://github.com/triologygmbh/jen-
kinsfile'
        }
     }

     stage('Build') {
        steps {
           sh 'mvn -B package'
        }
     }
  }
}
```

Listing 1

```
node {
  def mvnHome = tool 'M3'

  stage('Checkout') {
     git 'https://github.com/triologygmbh/jenkinsfile'
  }

  stage('Build') {
     sh "${mvnHome}/bin/mvn -B package"
  }
}
```

Listing 2

oben genannten Beispiel-Pipeline könnten beispielsweise jeweils eine Stage darstellen. Ein Step ist dabei ein Befehl, der einen konkreten Build-Schritt beschreibt und der letztlich von Jenkins ausgeführt wird. Eine Stage umfasst also einen oder mehrere Steps.

Für eine minimale Pipeline-Definition muss neben mindestens einer Stage mit einem Step noch ein Build-Executor allokiert sein, also etwa ein Jenkins-Build-Slave. Dies geschieht in der „declarative"-Syntax mithilfe der „agent"-Section, in der „scripted"-Syntax per „node"-Step. In beiden Varianten kann der Executor mit Labels weiter qualifiziert werden, sodass sichergestellt ist, dass er bestimmte Bedingungen erfüllt (etwa eine bestimmte Java-Version oder eine Docker-Installation zur Verfügung stellt).

Bevor es an der Zeit ist, die erste Pipeline einzurichten, noch ein Hinweis auf die verschiedenen Arten von Pipeline-Jobs, die Jenkins anbietet:

- *Pipeline*
  Ein einfacher Pipeline-Job, der die Script-Definition direkt über die Weboberfläche von Jenkins oder in einem „Jenkinsfile" aus dem Source Code Management (SCM) erwartet.
- *Multibranch-Pipeline*
  Erlaubt es, ein SCM-Repository mit mehreren Branches anzugeben. Steht in einem Branch ein „Jenkinsfile", wird die darin definierte Pipeline bei Änderungen am Branch ausgeführt. Pro Branch wird dabei on-the-fly ein Jenkins-Job angelegt.
- *GitHub-Organization*
  Eine Multibranch-Pipeline für eine GitHub-Organization beziehungsweise einen GitHub-User. Dieser Job scannt alle Repositories einer GitHub-Organization und legt für alle, deren Branches ein „Jenkinsfile" enthalten, einen Folder an, der eine Multibranch-Pipeline enthält. Es handelt sich also quasi um eine geschachtelte Multibranch-Pipeline *[2]*.

Erste Schritte

Um sich mit den Möglichkeiten des Pipeline-Plug-ins vertraut zu machen, bietet sich das simpelste Setup an: Man baut ein Projekt ohne vorhandenes „Jenkinsfile", indem man das Pipeline-Script direkt in einem Pipeline-Job über die Weboberfläche von Jenkins beschreibt.

Zum Einstieg ist es wichtig zu wissen, dass es in jeder Art von Pipeline-Job auf der Weboberfläche von Jenkins Links zur Dokumentation der auf der aktuellen Jenkins-Instanz verfügbaren Pipeline-Features gibt. Die in allen Jenkins-Instanzen verfügbaren Basic Steps *[3]* lassen sich durch zusätzliche Plug-ins erweitern.

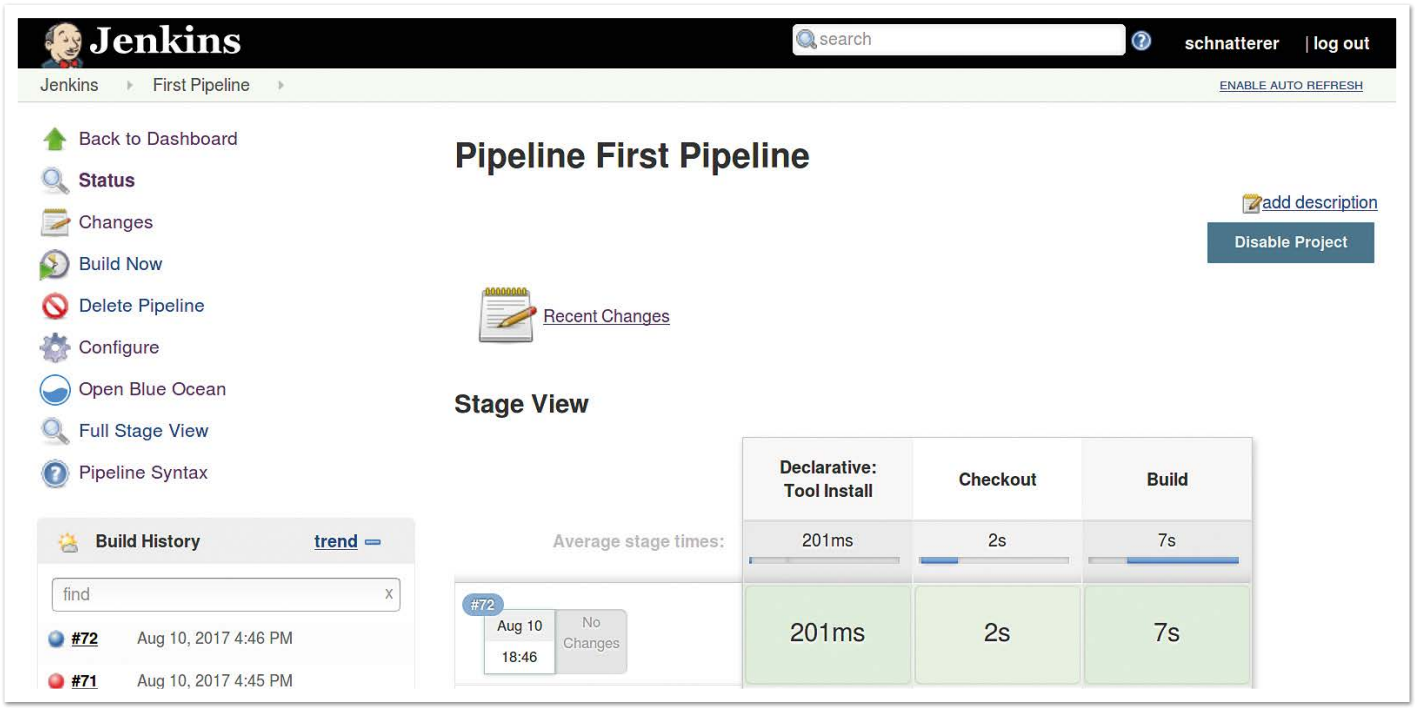

Abbildung 2: Jenkins-Stage-View im „classic Theme“

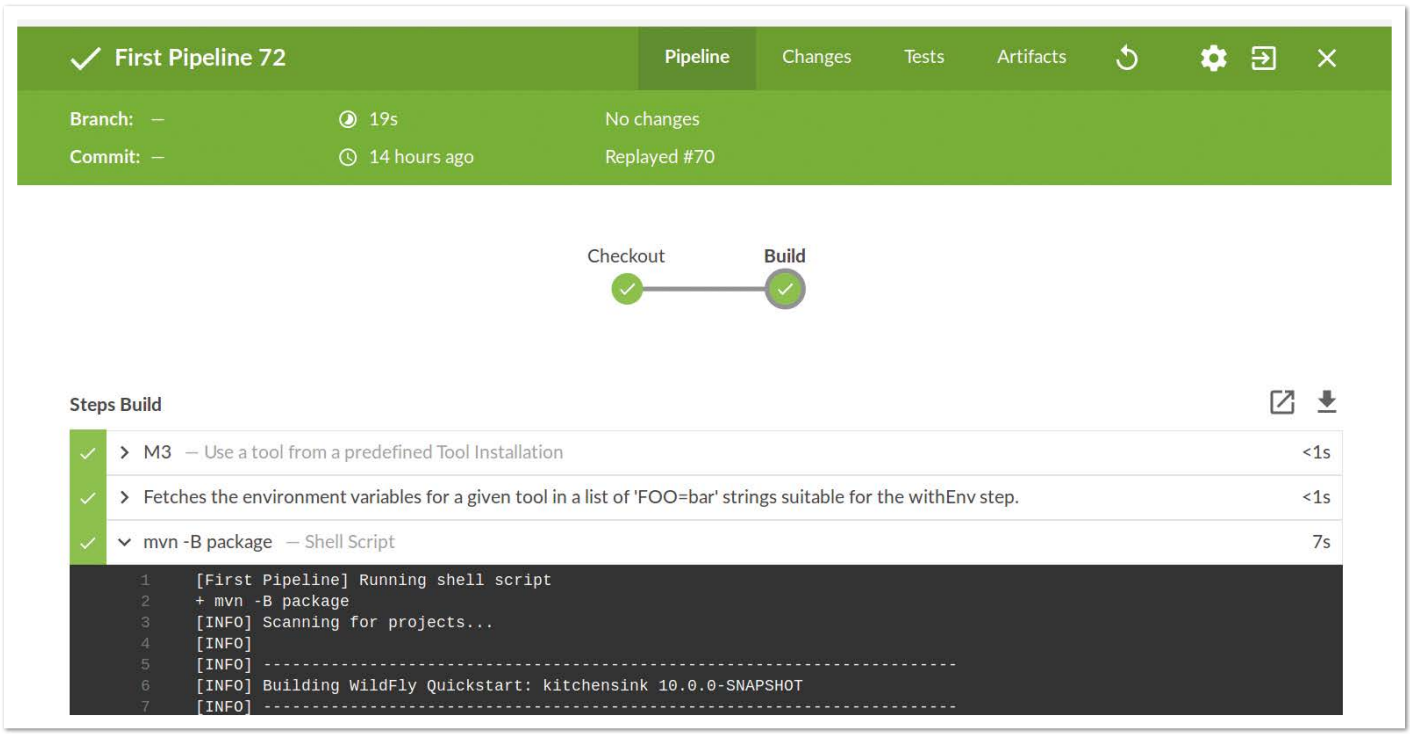

Abbildung 3: Ansicht eines Step-Logs im Blue-Ocean-Theme

Klickt man im Job auf Pipeline-Syntax, gelangt man auf den Snippet-Generator. Zusätzlich gibt es einen globalen Link in jeder Jenkins-Instanz. Die URL lautet *„https://jenkins/pipeline-syntax“*, also zu Beispiel: *„https://opensource.triology.de/jenkins/pipeline-syntax“*.

Der Snippet-Generator ist ein hilfreiches Tool für den Übergang vom bisherigen Jenkins-Job „Zusammenklicken“ zur Pipeline-Syntax. Hier kann man wie gewohnt Teile seines Build-Jobs mit der Maus zusammenstellen und sich daraus ein Snippet in Pipeline-Syntax generieren lassen *(siehe Abbildung 1)*. Der Snippet-Generator kann aber auch später hilfreich sein, um die Syntax neuer Plug-ins kennenzulernen oder wenn man keine Autovervollständigung in seiner IDE hat. Apropos Autovervollständigung: Im Snippet-Generator gibt es weitere Links auf hilfreiche Informationen:

- Die für diese Instanz verfügbaren globalen Variablen. Dazu gehören environment variables, Parameter des Build-Jobs und Informationen zum aktuellen Build-Job, zum Beispiel *„https://opensource.triology.de/jenkins/pipeline-syntax/globals“*
- Die ausführliche Dokumentation aller für diese Instanz verfügbaren Steps und Classes sowie ihre zugehörigen Parameter, zum Beispiel *„https://opensource.triology.de/jenkins/pipeline-syntax/html“*

Abbildung 4: Pipeline-Script from SCM

- Eine IntelliJ-Groovy-DSL-Datei (GDSL), mit der man Autovervollständigung aktivieren kann. Wie das funktioniert, zeigt ein Blog Post *[4]*

Pipeline-Scripts

Mit diesem Wissen kann das Schreiben von Pipeline-Scripts beginnen. Die grundlegenden Features des Jenkins-Pipeline-Plug-ins werden anhand eines typisches Java-Projekts gezeigt. Als Beispiel dient hier der „kitchensink"-Quickstart von WildFly, einer typischen JEE-Webanwendung mit CDI, JSF, JPA, EJB, JAX-RS und Integrationstests mit Arquillian. *Listing 1* zeigt die „declarative"-Syntax für ein minimales Pipeline-Script zum Bauen dieses Projekts auf Jenkins.

Dieses Script allokiert einen beliebigen Build-Executor, holt sich die unter „Tools" konfigurierte Maven-Instanz, checkt den Default-Branch der Git-URL aus und führt einen nicht-interaktiven Maven Build aus. Hier zeigt sich der einheitliche Aufbau der „declarative"-Pipeline. Jede Pipeline ist umschlossen vom „pipeline"-Block, der wiederum aus „Sections" beziehungsweise „Directives" *[5]* besteht. Diese spiegeln unter anderem die beschriebenen Kernkonzepte Stage und Step wider. *Listing 2* zeigt zum Vergleich die gleiche Pipeline in „scripted"-Syntax.

Auch hier sieht man das Konzept der Stage, dies beinhaltet dann direkt die Steps. Das Konzept der Sections beziehungsweise Directives gibt es hier nicht. Bei „node", „tool", „stage", „git" etc. spricht man hier jeweils nur von Steps. Diese Syntax bietet deutlich mehr Freiheiten. Anders als bei der „declarative"-Syntax, die immer von einem „pipeline"-Block umgeben sein muss, könnte man hier auch Steps außerhalb des „node"-Blocks ausführen. Dabei zeigt sich, warum die deklarative Syntax eine Teilmenge der „scripted"-Syntax ist: Im Prinzip ist der die „declarative"-Pipelines umschließende „pipeline"-Block ein Step der „scripted"-Syntax.

Damit diese Scripts auf einer Jenkins-Instanz ausführbar sind, muss auf einem Jenkins 2.60.2 auf Linux im Auslieferungszustand unter „Global Tool Configuration" nur eine Maven-Installation mit dem Namen „M3" angelegt sein. Danach kann dieses mithilfe der „tools"-Declarative beziehungsweise Step im Pipeline-Job bekannt gemacht und ausgeführt werden.

Im Hintergrund führt die Angabe des Tools „M3" dazu, dass Maven auf dem aktuellen Build-Executor zur Verfügung gestellt wird. Falls nötig, wird es dazu installiert und im „PATH" bekannt gemacht. Das Ergebnis des Builds ist im „classic Theme" von Jenkins in einer Stage-View angezeigt, in der je Build die Stages und deren Ausführungszeiten visualisiert sind *(siehe Abbildung 2)*.

```
stage('Checkout') {
checkout scm
}
```

Listing 3

Eine spezielle, für Pipelines designte Ansicht bietet das offizielle Jenkins-Theme „Blue Ocean" *[6]*. Mit deutlich modernerem UX kann man hier sehr übersichtlich mit Pipelines arbeiten. Man sieht auf einen Blick, welche Steps pro Stage ausgeführt wurden, und kann beispielsweise das Console-Log bestimmter Steps mit einem Klick einsehen. Für „declarative"-Pipelines gibt es außerdem einen visuellen Editor *[7]*. Wenn das Plug-in installiert ist, kann man über Links in jedem Build-Job zwischen Blue Ocean und Classic wechseln. *Abbildung 3* zeigt die Pipeline aus *Abbildung 2* in Blue Ocean.

Die gezeigten ersten Pipeline-Beispiele werden im Folgenden sukzessive erweitert, um die grundlegenden Features der Pipeline zu zeigen. Dabei sind die Änderungen jeweils sowohl in „declarative"- als auch in „scripted"-Syntax zu sehen. Den aktuellen Stand jeder Erweiterung kann man bei *[8]* nachverfolgen und ausprobieren. Hier gibt es für jeden Abschnitt unter der in der Überschrift genannten Nummer jeweils für „declarative" und „scripted" einen Branch, der das vollständige Beispiel umfasst. Das Ergebnis der Builds jedes Branch lässt sich außerdem direkt auf der Jenkins-Instanz der Autoren *[9]* einsehen.

Umzug ins SCM-/Jenkins-File

Einer der größten Vorteile von Jenkins-Pipelines ist, dass man sie unter Versionsverwaltung stellen kann. Dazu ist es Konvention, eine Datei „Jenkinsfile" ins Root-Verzeichnis des Repository zu legen. Dann kann der beschriebene Pipeline-Job auf „Pipeline-Script from SCM" *(siehe Abbildung 4)* umgestellt oder ein Multibranch-Pipeline-Job angelegt werden.

Die URL des SCM (in diesem Fall Git) wird im Job konfiguriert. Man könnte die Pipeline-Scripts wie gezeigt einchecken, allerdings würde man dann die URL des Repository im Repository wiederholen. Dies würde allerdings das DRY-Prinzip *[10]* verletzen. Die Lösungen sind je nach Syntax unterschiedlich:

- *Declarative*
  Der Checkout erfolgt standardmäßig durch die „agent"-Section; die Checkout Stage kann hier also komplett entfallen.
- *Scripted*
  Der Checkout wird nicht standardmäßig ausgeführt; es gibt jedoch die Variable „scm", die die im Job konfigurierte Repository-URL enthält, sowie den Step „checkout", der den im Job konfigurierten SCM-Provider (in diesem Falle Git) enthält *(siehe Listing 3)*.

Lesbarkeit mit eigenen Steps verbessern

Groovy macht es als Grundlage der Pipeline sehr einfach, die vorhandenen Steps zu erweitern. Am bestehenden Beispiel kann der Aufruf von Maven ausdrucksstärker gemacht werden, indem er in eine eigene Methode gekapselt wird *(siehe Listing 4)*.

Diese erlaubt es, sowohl in „declarative"- als auch in „scripted"-Syntax Maven mit „mvn 'package'" aufzurufen. Die Definition der Tools ist in die Methode verlagert. Damit werden für die Ausführung auf Jenkins sinnvolle Maven-Parameter (Batch Mode, Maven-Version ausgeben, Snapshots updaten, fehlgeschlagene Tests auf Konsole ausgeben) von den im Kontext des jeweiligen Aufrufs interessanten Maven-Parametern (hier „package"-Phase) getrennt. Dies erhöht die Lesbarkeit, weil man beim Aufruf nur die wesentlichen Parameter übergibt. Zudem muss man nicht die Parameter wiederholen, die ohnehin bei jedem Aufruf mitgegeben werden sollten.

Zusätzlich kommt hier ein spezifisches JDK zum Einsatz. Dies erfordert zwar, dass analog zu Maven unter „Global Tool Configuration" nun eine JDK-Installation mit dem Namen „JDK8" installiert ist. Dadurch wird jedoch der Build deterministischer, da nicht implizit das JDK von Jenkins verwendet wird, sondern ein explizit benanntes.

Diese Maven-Methode ist den offiziellen Beispielen entnommen *[11]*. Eine Methode wie „mvn" ist ein guter Kandidat für die Auslagerung in eine Shared-Library. Diese ist in einem folgenden Teil dieser Artikelserie beschrieben.

Unterteilung in Stages

Ähnlich wie beim Schreiben von Methoden oder Funktionen in der Software-Entwicklung ist es auch für die Wartbarkeit von Pipeline-Scripts sinnvoll, kleine Stages zu verwenden. Diese Unterteilung erlaubt es, bei scheiternden Builds mit einem Blick zu erkennen, wo etwas schiefging. Außerdem werden die Zeiten pro Stage gemessen, wodurch schnell ersichtlich wird, welche Teile des Builds am meisten Zeit benötigen.

Der Maven Build in diesem Beispiel lässt sich weiter unterteilen in „Build"- und „Unit"-Test. An dieser Stelle werden zudem erstmals Integration-Tests ausgeführt. In diesem Beispiel erfolgt die Ausführung mit Arquillian und WildFly Swarm. In „declarative"-Syntax sieht das wie in *Listing 5* aus. In „scripted"-Syntax entfallen die „steps"-Sections.

Als Nachteil ist zu nennen, dass der gesamte Build dadurch etwas langsamer wird, da verschiedene Maven-Phasen in mehreren Stages durchlaufen werden. Dies ist hier schon optimiert, indem „clean" nur einmal am Anfang aufgerufen wird und die „pom.xml" um ein Property „skipUnitTests" erweitert ist, was die erneute Ausführung der Unit-Tests in der Integration-Test-Stage verhindert.

Bei Integration-Tests besteht generell die Gefahr von Port-Konflikten. Beispielsweise kann die Infrastruktur gleichzeitig laufender Builds an die gleichen Ports binden. Dadurch kommt es zu unerwartet fehlschlagenden Builds. Dies lässt sich durch die Verwendung von Docker effektiv umgehen. Dies ist in einem folgenden Teil dieser Artikelserie beschrieben.

Ende des Pipeline-Laufs und Fehlerbehandlung

Üblicherweise gibt es Schritte, die immer am Ende eines Pipeline-Laufs ausgeführt werden sollen, unabhängig davon, ob der Build erfolgreich war oder nicht. Das beste Beispiel dafür sind Test-Ergebnisse. Schlägt ein Test fehl, wird auch der Build fehlschlagen. In jedem Fall sollten aber die Test-Ergebnisse in Jenkins erfasst werden.

Zudem sollte bei fehlschlagenden Builds oder wenn sich der Status des Builds ändert, eine spezielle Reaktion erfolgen. Üblich ist hier das Versenden von E-Mails, denkbar wären aber auch Chat-Benachrichtigungen oder Ähnliches. Beide Fälle sind in Pipelines mit den ähnlichen syntaktischen Konzepten spezifiziert. Allerdings unterscheiden sich hier die Ansätze zwischen „declarative"- und „scripted"-Syntax.

Generell stünden in beiden Fällen die Sprachmittel von Groovy, also „try-catch-finally"-Blöcke, zur Verfügung. Diese sind allerdings nicht ideal, da gefangene Exceptions dann keine Auswirkungen auf den Build-Status haben. In der „declarative"-Syntax stehen hier die „post"-Section mit

```
def mvn(def args) {
  def mvnHome = tool 'M3'
  def javaHome = tool 'JDK8'

  withEnv(["JAVA_HOME=${javaHome}",
"PATH+MAVEN=${mvnHome}/bin:${env.JAVA_HOME}/bin"]) {
     sh "${mvnHome}/bin/mvn ${args} --batch-mode -V -U
-e -Dsurefire.useFile=false"
  }
}
```

Listing 4

```
stages {
  stage('Build') {
     steps {
        mvn 'clean install -DskipTests'
     }
  }

  stage('Unit Test') {
     steps {
        mvn 'test'
     }
  }

  stage('Integration Test') {
     steps {
        mvn 'verify -DskipUnitTests -Parq-wildfly-
swarm '
     }
  }
}
```

Listing 5

```
post {
  always {

     junit allowEmptyResults: true,
        testResults: '**/target/surefire-reports/TEST-
*.xml, **/target/failsafe-reports/*.xml'
  }
  changed {
     mail to: "${env.EMAIL_RECIPIENTS}",
        subject: "${JOB_NAME} - Build #${BUILD_NUMBER}
- ${currentBuild.currentResult}!",
        body: "Check console output at ${BUILD_URL} to
view the results."
  }
}
```

Listing 6

```
node {
  catchError {
     // ... Stages ...
  }

  junit allowEmptyResults: true,
        testResults: '**/target/surefire-reports/TEST-*.xml, **/target/failsafe-reports/*.xml'
  statusChanged {
     mail to: "${env.EMAIL_RECIPIENTS}",
        subject: "${JOB_NAME} - Build #${BUILD_NUMBER} - ${currentBuild.currentResult}!",
        body: "Check console output at ${BUILD_URL} to view the results."
  }
}
def statusChanged(body) {
  def previousBuild = currentBuild.previousBuild
  if (previousBuild != null && previousBuild.result != currentBuild.currentResult) {
     body()
  }
}
```

Listing 7

```
node {
  // ... catchError und nodes
  mailIfStatusChanged env.EMAIL_RECIPIENTS
}

def mailIfStatusChanged(String recipients) {

  if (currentBuild.currentResult == 'SUCCESS') {
     currentBuild.result = 'SUCCESS'
  }
  step([$class: 'Mailer', recipients: recipients])
}
```

Listing 8

```
pipeline {
  agent any

  options {
     disableConcurrentBuilds()
     buildDiscarder(logRotator(numToKeepStr: '10'))
  }

  stages { /* .. */ }
}
```

Listing 9

```
node {
  properties([
     disableConcurrentBuilds(),
     buildDiscarder(logRotator(numToKeepStr: '10'))
  ])

  catchError { /* ... */ }
}
```

Listing 10

```
stage('Build') {
steps {
mvn 'clean install -DskipTests'
archiveArtifacts '**/target/*.*ar'
}
}
```

Listing 11

den Conditions „always“, „changed“, „failure“, „success“ und „unstable“ zur Verfügung. Damit lässt sich ausdrucksstark definieren, was am Ende jeder Ausführung erfolgen soll. *Listing 6* zeigt das beschriebene Szenario.

Erwähnenswert ist hier, dass man mit wenig Aufwand auch den bestehenden E-Mail-Mechanismus von Jenkins verwenden kann, was später in diesem Abschnitt erklärt ist. Zudem ist die Herkunft der E-Mail-Adresse der Empfänger hier von Relevanz. Im Beispiel werden diese aus der Environment-Variable „EMAIL_RECIPIENTS“ geladen. Diese muss von einem Administrator in der Jenkins-Konfiguration festgelegt sein. Alternativ kann man die Empfänger natürlich auch direkt ins „Jenkinsfile“ schreiben. Dann werden sie allerdings mit ins SCM eingecheckt.

In der „scripted“-Syntax steht hier nur der „catchError“-Step zur Verfügung. Dieser verhält sich im Wesentlichen wie ein „finally“-Block. Um obiges Szenario abzubilden, muss mit „if“-Bedingungen gearbeitet werden *(siehe Listing 7)*. Auch hier empfiehlt es sich, aus Gründen der Wartbarkeit einen eigenen Step zu definieren.

Wie erwähnt, lässt sich das Thema „E-Mails“ sowohl in der „declarative“- als auch in der „scripted“-Syntax vereinfachen, indem der bestehende E-Mail-Mechanismus von Jenkins zum Einsatz kommt. Hier werden die bekannten „Build failed in Jenkins“-und „Jenkins build is back to normal“-E-Mails versendet. Dazu wird die Klasse „Mailer“ verwendet, für die es keinen dedizierten Step gibt. Dies ist über den generischen Step „step“ möglich.

Wenn man auch die „back to normal“-E-Mails erhalten möchte, ist noch eine Besonderheit zu beachten: Die Klasse „Mailer“ liest den Wert aus der Variable „currentBuild.result“ aus. In der Pipeline wird diese im Erfolgsfall erst ganz am Ende der Pipeline gesetzt, dadurch erfährt die Klasse „Mailer“ es nie. Also bietet sich auch hier die Implementierung als eigener Step an. In „scripted“-Syntax lässt sich dies wie in *Listing 8* realisieren, die Lösung lässt sich aber auch mit „declarative“-Syntax verwenden. Abschließend sei in Bezug auf Notification mit Hipchat, Slack etc. ein Blogbeitrag von Jenkins empfohlen *[12]*.

Properties und Archivierung

Bei herkömmlichen Jenkins-Jobs gibt es viele kleinere Einstellungen, die man über die Oberfläche vornimmt, wie beispielsweise die Größe der Build-Historie, Verhindern paralleler Builds etc. Selbstverständ-

lich sind diese bei Verwendung des Pipeline-Plug-ins im „Jenkinsfile" beschrieben. In der „declarative"-Syntax heißen diese Einstellungen „options" *(siehe Listing 9)*, in der „scripted"-Syntax „properties". Sie werden über den gleichnamigen Step gesetzt *(siehe Listing 10)*.

Ein weiterer nützlicher Step ist „archiveArtifacts". Er speichert durch den Build erstellte Artefakte („.jar", „.war", „.ear" etc.), sodass diese über die Weboberfläche von Jenkins eingesehen werden können. Dies kann für Debugging nützlich sein oder tatsächlich zur Archivierung von Versionen, wenn man kein Maven Repository verwendet. In „declarative"-Syntax lässt sich dies wie in *Listing 11* formulieren. In „scripted"-Syntax entfallen die „steps"-Sections. Sie speichert alle „.jar", „.war", „.ear" etc., die in einem der Maven-Module erzeugt wurden.

Tipps für den Einstieg

Es gibt einige weitere grundlegende Directives wie beispielsweise „parameters" (deklariert Build-Parameter) und „script" (zum Ausführen eines Blocks in „scripted"-Syntax innerhalb der „declarative"-Syntax). Hier ist die Lektüre *[5]* empfehlenswert. Darüber hinaus gibt es viele weitere Steps, die im Wesentlichen durch Plug-ins bereitgestellt werden, siehe offizielle Übersicht *[13]*. Damit diese in der Pipeline verfügbar sind, müssen Entwickler ein entsprechendes API verwenden. Die Pipeline-Kompatibilität einzelner Plug-ins ist hier zusammengefasst *[14]*. Zum jetzigen Zeitpunkt haben die meisten gängigen Plug-ins Unterstützung für das Jenkins-Pipeline-Plug-in. Eine weitere empfehlenswerte Literatur sind die Top 10 Best Practices for Jenkins Pipeline *[15]*.

Abschließend noch einige nützliche Tipps für die Arbeit mit dem „Jenkinsfile". Beim ersten Aufsetzen einer Pipeline empfiehlt es sich, mit einem normalen Pipeline-Job zu starten und das „Jenkinsfile" erst unter Versionsverwaltung zu stellen, wenn der Build läuft. Ansonsten läuft man Gefahr, seine Commit-Historie zu verunreinigen. Bei Änderungen einer bestehenden Multibranch-Pipeline bietet sich das „Replay"-Feature an, mit dem temporär für die nächste Ausführung die Pipeline über die Weboberfläche von Jenkins editieren kann, ohne das „Jenkinsfile" im SCM zu verändern.

Ein letzter Tipp: Auch bei Pipelines kann man den Workspace über die Weboberfläche von Jenkins einsehen. Durch die „agent"-Section beziehungsweise den „node"-Step ist es möglich, mehrere Build-Executors zu belegen. Das ist mit dem Thema „Parallelisierung" in einem folgenden Teil dieser Artikelserie näher beschrieben. Deshalb kann es auch mehrere Workspaces geben. Diese kann man im „classic Theme" jeweils einsehen, indem man in einem Build-Job auf „Pipeline Steps" klickt und dort auf „Allocate node : Start". Hier gibt es dann den bekannten „Workspace"-Link auf der linken Seite.

Fazit und Ausblick

Dieser Artikel zeigt die Grundlagen des Jenkins-Pipeline-Plug-ins. Er beschreibt die Grundkonzepte und Begriffe, die verschiedenen Job-Arten, führt theoretisch sowie anhand von Beispielen in die Syntax des „Jenkinsfile" ein und gibt Praxistipps für das Arbeiten mit Pipelines. Das beschriebene Beispiel konfiguriert Jenkins, baut den Code, führt darauf Unit- und Integration-Tests aus, archiviert Test-Ergebnisse sowie Artefakte und versendet E-Mails – das alles mit einem ungefähr dreißig Zeilen langen Skript.

Um von Continous Delivery zu sprechen, fehlen hier natürlich noch Schritte wie statische Code-Analyse, beispielsweise mit SonarQube, und natürlich Deployments auf Staging- und Produktiv-Umgebungen. Um dies zu realisieren, bietet sich die Verwendung einiger Werkzeuge und Methoden wie Nightly Builds, Wiederverwendung über verschiedene Jobs hinweg, Unit Testing, Parallelisierung und Docker an. Dies ist in folgenden Teilen dieser Artikelserie beschrieben.

Weitere Informationen

[1] *https://jenkins.io/doc/book/pipeline/syntax/#differences-from-plain-groovy*
[2] *https://opensource.triology.de/jenkins/job/triologygmbh-github*
[3] *https://jenkins.io/doc/pipeline/steps/workflow-basic-steps*
[4] *https://www.triology.de/blog/jenkins-pipeline-plugin-code-vervollstaendigung*
[5] *https://jenkins.io/doc/book/pipeline/syntax/#declarative-pipeline*
[6] *https://jenkins.io/doc/book/blueocean*
[7] *https://www.cloudbees.com/blog/getting-started-blue-oceans-visual-pipeline-editor*
[8] *https://github.com/triologygmbh/jenkinsfile*
[9] *https://opensource.triology.de/jenkins/job/triologygmbh-github/job/jenkinsfile*
[10] *https://pragprog.com/the-pragmatic-programmer/extracts/tips*
[11] *https://github.com/jenkinsci/pipeline-examples*
[12] *https://jenkins.io/blog/2017/02/15/declarative-notifications*
[13] *https://jenkins.io/doc/book/pipeline/syntax*
[14] *https://github.com/jenkinsci/pipeline-plugin/blob/master/compatibility.md*
[15] *https://www.cloudbees.com/blog/top-10-best-practices-jenkins-pipeline-plugin*

Johannes Schnatterer
johannes.schnatterer@triology.de

Johannes Schnatterer ist Solution Architect bei der TRIOLOGY GmbH in Braunschweig. Technologisch ist er dort in den Bereichen „Java EE" und „Web" tätig und versucht, mit besonderem Fokus auf Qualität, Open-Source-Enthusiasmus, einem Hauch von Pedantismus und der Pfadfinderregel die IT-Welt jeden Tag ein bisschen besser zu machen.

Daniel Behrwind
d.behrwind@gmail.com

Daniel Behrwind ist als Software-Entwickler bei der TRILOGY GmbH tätig. Dort befasst er sich überwiegend mit der Entwicklung von Individual-Software, wobei er schwerpunktmäßig an Java-Webprojekten arbeitet. Als leidenschaftlicher Clean-Code-Verfechter ist er begeistert von kreativen Lösungen für komplexe Probleme, die so offensichtlich aussehen, als seien sie von selbst entstanden.