Java aktuell

iJUG Verbund
www.ijug.eu

Coding Continuous Delivery
Hilfreiche Werkzeuge für die Jenkins-Pipeline

Besser Programmieren
Die Entwicklungsumgebung optimal nutzen

Aus der Praxis
Microservices richtig einsetzen

Java und mehr

Ausgabe 03/2018
Java aktuell
Praxis. Wissen. Networking. Das Magazin für Entwickler
D: 4,90 EUR | CH: 9,80 CHF | A: 5,60 EUR | Benelux: 5,80 EUR
4 191978 304903 03

Jenkins

Coding Continuous Delivery – hilfreiche Werkzeuge für die Jenkins-Pipeline

Johannes Schnatterer, Triology GmbH

Nachdem in den letzten beiden Ausgaben Grundlagen und Performance von Jenkins-Pipelines thematisiert wurden, beschreibt dieser Artikel nützliche Werkzeuge und Methoden: Mit Shared Libraries lassen sich die Wiederverwendung über verschiedene Jobs hinweg und Unit Testing des Pipeline-Codes realisieren. Außerdem bietet der Einsatz von Containern mittels Docker auch hier seine Vorzüge.

Nachfolgend sind die Pipeline-Beispiele aus den ersten beiden Artikeln sukzessive erweitert, um die Features der Pipeline zu zeigen. Dabei werden die Änderungen jeweils in „declarative"- und in „scripted"-Syntax realisiert. Den aktuellen Stand jeder Erweiterung kann man bei GitHub *[1]* 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 *[2]* einsehen.

Wie in den ersten beiden Teilen werden auch in diesem die Features des Jenkins-Pipeline-Plug-ins anhand eines typischen

Java-Projekts gezeigt. Als Beispiel dient damit auch hier der „kitchensink"-Quickstart von WildFly. Da dieser Artikel auf den Beispielen aus dem ersten Artikel aufbaut, setzt sich die Nummerierung aus dem ersten Teil fort. Dort wurden mit simpler Pipeline, eigenen Steps, Stages, Fehlerbehandlung und Properties/Archivierung, Parallelisierung und Nightly Builds bereits sieben Beispiele gezeigt.

Shared Libraries

In den Beispielen aus dieser Artikelserie gibt es bereits einige selbst geschriebene Steps wie beispielsweise „mvn()" oder „maillfStatusChanged()". Sie sind nicht projektspezifisch und könnten aus dem „Jenkinsfile" ausgelagert und damit auch in anderen Projekten wiederverwendet werden. Für das Auslagern gibt es bei Jenkins-Pipelines derzeit zwei Möglichkeiten:

- **„load"-Step**
  Lädt eine Groovy-Skript-Datei aus dem Jenkins-Workspace (also dem gleichen Repository) und evaluiert sie; damit lassen sich dynamisch weitere Steps nachladen
- **Shared Libraries**
  Erlauben Einbinden von externen Groovy-Skripten und -Klassen

Der „load"-Step unterliegt gewissen Einschränkungen:

- Es können nur Groovy-Skripte und keine Klassen geladen werden *[3]*. Dadurch lassen sich in diesen Skripten beispielsweise nicht ohne Weiteres zusätzliche Klassen laden und Vererbung ist nicht möglich. Damit die Skripte in der Pipeline einsetzbar sind, muss jedes Skript mit „return this;" enden.
- Man kann nur Dateien aus dem Workspace verwenden. Dadurch ist keine Wiederverwendung über Projekte möglich.
- Die über diesen Step geladenen Skripte werden in dem im ersten Artikel beschriebenen „Replay"-Feature nicht angezeigt. Dadurch sind sie schwerer zu entwickeln und zu debuggen.

Shared Libraries unterliegen diesen drei Einschränkungen nicht, wodurch sie wesentlich flexibler sind. Deshalb ist ihre Verwendung nachfolgend näher beschrieben: Aktuell muss eine Shared Library aus einem eigenen Repository geladen werden. Das Laden aus dem zu bauenden Repository ist momentan noch nicht möglich, wird es aber wahrscheinlich in Zukunft sein *[4]*. Dadurch ist es zukünftig möglich, das „Jenkinsfile" auf verschiedene Klassen/Skripte aufzuteilen. Dies erhöht die Wartbarkeit und schafft die Möglichkeit für das Schreiben von Unit-Tests. Außerdem ist dies für die Entwicklung von Shared Libraries hilfreich, weil diese in ihrem eigenen „Jenkinsfile" verwendet werden können. Das Repository jeder Shared Library muss eine bestimmte Verzeichnisstruktur aufweisen:

- „src" enthält Groovy-Klassen
- „vars" enthält Groovy-Skripte und Dokumentation
- „resources" enthält weitere Dateien

Zudem sind ein „test"-Verzeichnis für Unit-Tests und ein eigener Build empfehlenswert. Um die Komplexität des „Jenkinsfile" aus den Beispielen zu reduzieren und die Funktionalität für andere Projekte wiederverwendbar zu machen, wird im Folgenden exemplarisch ein Step in eine Shared Library ausgelagert. Für den Step „mvn" legt man beispielsweise im Repository der Shared Library im Verzeichnis „vars" eine Datei „mvn.groovy" an *(siehe Listing 1)*. Diese enthält die aus dem ersten Teil dieser Artikelserie bekannte Methode.

Im Groovy-Skript in *Listing 1* wird diese Methode allerdings nach Groovy-Konvention „call()" genannt. Technisch gesehen legt Jenkins für alle „.groovy"-Dateien im Verzeichnis „vars" eine globale Variable an und benennt sie entsprechend dem Dateinamen. Ruft man diese Variable jetzt mit dem Call-Operator „()" auf, wird implizit deren Methode „call()" aufgerufen *[5]*. Da bei Groovy die Klammern beim Aufruf optional sind, bleibt der Aufruf der Steps in „scripted"- und „declarative"-Syntax wie bisher, beispielsweise „mvn 'test'".

Um die Shared Library in der Pipeline zu verwenden, gibt es mehrere Möglichkeiten. Zunächst müssen die Shared Libraries in Jenkins bekannt gemacht sein. Dazu gibt es folgende Möglichkeiten der Definition von Shared Libraries:

- **Global**
  Muss durch einen Jenkins-Administrator in der Jenkins-Konfiguration eingestellt sein. Dort definierte Shared Libraries sind in allen Projekten verfügbar und werden als vertrauenswürdig behandelt; sie dürfen also alle Groovy-Methoden, interne Jenkins-APIs etc. ausführen. Hier ist also Vorsicht geboten. Man kann dies allerdings auch nutzen, um beispielsweise die unter Nightly Builds beschriebenen Aufrufe zu kapseln, für die sonst Script Approval notwendig wäre.
- **Folder/Multibranch**
  Kann von entsprechend berechtigten Projekt-Mitgliedern für eine Gruppe von Build-Jobs eingestellt werden. Dort definierte Shared Libraries gelten nur für zugehörige Build-Jobs und werden nicht als vertrauenswürdig behandelt. Das heißt, sie laufen in der Groovy-Sandbox – wie normale Pipelines auch.
- **Automatisch**
  Plug-ins wie das Pipeline-GitHub-Library-Plug-in *[6]* erlauben das automatische Definieren von Libraries innerhalb von Pipelines, die in einem GitHub Organization Folder definiert sind. Dadurch können Shared Libraries ohne vorherige Definition im Jenkins direkt in „Jenkinsfile" verwendet werden. Auch diese Shared Libraries laufen in der Groovy-Sandbox.

```
def call(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 1

```
@Test
void mvn() {
    def shParams = ""
    helper.registerAllowedMethod("tool", [String.class], { paramString -> paramString })
    helper.registerAllowedMethod("sh", [String.class], { paramString -> shParams = paramString })
    helper.registerAllowedMethod("withEnv", [List.class, Closure.class], { paramList, closure ->
        closure.call()
    })
    def script = loadScript('vars/mvn.groovy')
    script.env = new Object() {
        String JAVA_HOME = "javaHome"
    }
    script.call('clean install')
    assert shParams.contains('clean install')
}
```

Listing 2

In unserem Beispiel bietet sich die Verwendung des GitHub-Branch-Source-Plug-ins an, da es bei GitHub verfügbar und deshalb keine weitere Konfiguration in Jenkins notwendig ist. Sowohl in den „scripted"- als auch in den „declarative"-Syntax-Beispielen können die ausgelagerten Steps (beispielsweise „mvn") durch das Einbinden der Shared Library in der ersten Zeile des Skripts mit „@Library('github.com/triologygmbh/jenkinsfile@e00bbf0') _" definiert werden. Dabei ist „github.com/triologygmbh/jenkinsfile" der Name der Shared Library und nach dem „@" steht die Version, in diesem Fall ein Commit Hash. Hier könnte man auch einen Branch- oder Tag-Namen nehmen.

Es empfiehlt sich, einen definierten Stand (Tag oder Commit anstelle von Branches) zu verwenden, um deterministisches Verhalten zu gewährleisten. Da die Shared Library in jedem Build neu aus dem Repository abgerufen wird, besteht sonst die Gefahr, dass eine Änderung daran ohne Änderung des eigentlichen Pipeline-Skripts oder -Codes Auswirkung auf den nächsten Build hat. Dies kann zu unerwarteten Ergebnissen führen, deren Ursache schwer zu finden ist. Alternativ kann man Libraries dynamisch (mit dem „library"-Step) laden. Diese können dann erst nach dem Aufruf des Steps verwendet werden.

Wie erwähnt, kann man in Shared Libraries außer Skripten auch Klassen anlegen (im „src"-Ordner). Liegen diese in Packages, können sie über Import-Statements nach der „@Library"-Annotation angegeben werden. Diese Klassen können in „scripted"-Syntax überall in der Pipeline instanziiert werden, in „declarative"-Syntax nur innerhalb des „script"-Steps. Ein Beispiel dafür ist die Shared Library des Cloudogu EcoSystem *[7]*.

Außerdem bieten Shared Libraries die Möglichkeit, Unit-Tests zu schreiben. Für Klassen ist dies häufig mit Groovy-Bordmitteln möglich *[7]*. Für Skripte bietet sich die Verwendung von „JenkinsPipelineUnit" *[8]* an. Mit diesem Framework kann man Skripte laden und einfach Mocks der eingebauten Pipeline-Steps definieren. *Listing 2* zeigt, wie ein Test für den in *Listing 1* beschriebenen Step aussehen könnte.

Dort wird überprüft, ob der übergebene Parameter korrekt an den „sh"-Step weitergereicht wird. Die Variable „helper" wird dabei der Test-Klasse durch das Framework über Vererbung bereitgestellt. Wie man in *Listing 2* sieht, kommt hier viel Mocking zum Einsatz: etwa die „tool"- und „withEnv"-Steps sowie die globale Variable „env". Daran zeigt sich schon, dass der Unit-Test nur die grundlegende Logik prüft und natürlich nicht den Test in einer echten Jenkins-Umgebung ersetzt.

Diese Integrationstests kann man derzeit noch nicht automatisiert ausführen. Für die Entwicklung von Shared Libraries bietet sich das im ersten Artikel beschriebene „Replay"-Feature an: Neben dem „Jenkinsfile" kann man hier auch temporär die Shared Library verändern und ausführen. Dadurch vermeidet man viele unnötige Commits ins Repository der Shared Library. Dieser Tipp ist auch in der umfangreichen Dokumentation zu Shared Libraries beschrieben *[9]*. Zusätzlich zum Auslagern von Steps kann man ganze Pipelines in Shared Libraries definieren *[10]* und so beispielsweise seine Stages standardisieren. Zum Abschluss des Themas noch einige Open Source Shared Libraries:

- Offizielle Beispiele mit Shared Library und Jenkinsfile *[11]*; enthält Klassen und Skripte
- Shared Library, die von Docker Inc. für die Entwicklung verwendet wird *[12]*; enthält Klassen und Skripte
- Shared Library, die vom Firefox Test Engineering verwendet wird *[13]*; enthält Skripte mit Unit Tests und Groovy Build
- Shared Library des Cloudogu EcoSystem *[7]*; enthält Klassen und Skripte mit Unit-Tests und Maven Build

Docker

Durch den Einsatz von Docker in Jenkins-Builds lassen sich Build- und Test-Umgebung vereinheitlichen und Anwendungen deployen. Außerdem können, wie bereits im ersten Artikel dieser Serie erwähnt, durch die Isolierung Port-Konflikte bei parallelen Builds verhindert werden. Ein weiterer Vorteil ist, dass weniger Konfiguration in Jenkins notwendig ist. Auf Jenkins muss nur Docker bereitgestellt werden. Die Pipelines können dann benötigte Tools (Java, Maven, Node.js, PaaS-CLIs etc.) einfach per Docker-Image beziehen.

Damit man in Pipelines Docker nutzen kann, muss natürlich ein Docker-Host verfügbar sein. Dies ist ein Infrastruktur-Thema, das außerhalb von Jenkins zu lösen ist. Auch unabhängig von Docker ist es empfehlenswert, in Produktion den Build-Executor getrennt vom Jenkins-Master zu betreiben, um die Last zu verteilen und Reaktionszeiten der Jenkins-Web-Anwendung nicht durch Builds zu verlangsamen. Das gilt auch bei der Bereitstellung von Docker auf den Build-Executors: Der Docker-Host des Masters (falls vorhanden) sollte getrennt vom Docker-Host der Build-Executors sein. Auch

```
pipeline {
    agent {
        docker {
            image 'maven:3.5.0-jdk-8'
            label 'docker'
        }
    }
    //...
}
```

Listing 3

```
node('docker') {
    // ...
    docker.image('maven:3.5.0-jdk-8').inside {
            // ...
     }
}
```

Listing 4

dies stellt sicher, dass die Jenkins-Web-Anwendung unabhängig von den Builds reaktionsfreudig bleibt. Außerdem bietet die Trennung der Hosts zusätzliche Sicherheit, da im Falle von Container-Breakouts *[14]* kein Zugriff auf den Jenkins-Host möglich ist.

Wenn man einen speziellen Build-Executor mit Docker aufsetzt, ist es empfehlenswert, darin auch direkt den Docker-Client zu installieren und im „PATH" verfügbar zu machen. Alternativ kann man den Docker-Client auch als Tool in Jenkins installieren. Dieses Tool muss dann (wie Maven und JDK in den Beispielen im ersten Artikel dieser Serie) explizit in der Pipeline-Syntax angegeben sein. Das ist derzeit jedoch nur in „scripted"- und nicht mit „declarative"-Syntax möglich *[15]*.

Sobald Docker eingerichtet ist, bietet die „declarative"-Syntax die Möglichkeit, entweder die gesamte Pipeline oder einzelne Stages innerhalb eines Docker-Containers auszuführen. Das dem Container zugrunde liegende Image kann entweder aus einer Registry gezogen *(siehe Listing 3)* oder aus einem „Dockerfile" gebaut werden.

Durch die Verwendung des „docker"-Parameters in der „agent"-Section wird die gesamte Pipeline innerhalb eines Containers ausgeführt, der aus dem angegebenen Image erstellt wird. Das in *Listing 3* verwendeten Image sorgt dafür, dass die Executables von Maven und des JDK im „PATH" bereitgestellt werden. Man kann damit hier ohne weitere Konfiguration von Tools in Jenkins (wie Maven und JDK in den Beispielen im ersten Artikel dieser Serie) beispielsweise den Step „sh 'mvn test'" ausführen.

Das in *Listing 3* gesetzte Label bezieht sich in diesem Fall auf den Build-Executor. Dies bewirkt, dass die Pipelines nur auf Build-Executors ausgeführt werden, bei denen ein Label „docker" gesetzt ist. Insbesondere, wenn man verschiedene Build-Executors hat, ist diese Best-Practice hilfreich. Wird diese Pipeline auf einem Build-Executor ausgeführt, auf dem kein Docker-Client im „PATH" verfügbar ist, scheitert dieser. Ist jedoch kein Build-Executor mit dem Label verfügbar, bleibt der Build in der Queue.

Ein weiterer Punkt, den man bei in Containern ausgeführten Builds oder Steps bedenken muss, ist das Speichern von Daten außerhalb der Container. Da jeder Build in einem neuen Container ausgeführt wird, sind die darin enthaltenen Daten bei der nächsten Ausführung nicht mehr verfügbar. Jenkins sorgt zwar dafür, dass der Workspace als Working-Directory in den Container gemountet wird. Dies geschieht jedoch beispielsweise nicht für das lokale Maven-Repository. Während der in den Beispielen bisher verwendete „mvn"-Step (basierend auf den Jenkins-Tools) das Maven-Repository des Build-Executor verwendet, legt der Docker-Container ein Maven-Repository im Workspace jedes Builds an. Das kostet zwar etwas mehr Speicher und der jeweils erste Build wird langsamer. Dafür schließt man unerwünschte Seiteneffekte aus, etwa wenn zwei gleichzeitig laufende Builds eines Maven-Multi-Module-Projekts sich gegenseitig Snapshots im gleichen lokalen Repository überschreiben.

Wenn trotzdem das Repository des Build-Executor verwendet werden soll, sind einige Anpassungen am Docker-Image notwendig *[16]*. Was man eher vermeiden sollte, ist das Ablegen des lokalen Maven-Repository im Container. Dies würde dazu führen, dass alle Dependencies bei jedem Build neu aus dem Internet geladen werden, was wiederum die Dauer jedes Builds deutlich verlängern würde. Das in *Listing 3* in „declarative"-Syntax beschriebene Verhalten lässt sich auch in „scripted"-Syntax realisieren *(siehe Listing 4)*.

Wie in „declarative"-Syntax *(Listing 3)* lassen sich auch in „scripted"-Syntax Build-Executors durch Label auswählen. Dort *(Listing 4)* erfolgt dies als Parameter des „node"-Steps. Hier wird Docker über eine globale Variable angesprochen *[17]*. Diese Variable bietet noch weitere Features, unter anderem:

- Man kann bestimmte Docker-Registries verwenden (unter anderem für Continuous Delivery auf Kubernetes hilfreich, was im nächsten Teil dieser Serie beschrieben wird)
- Man kann einen bestimmten Docker-Client (definiert als Jenkins-Tool, wie oben beschrieben) verwenden
- Man kann Images bauen, mit Tags versehen und in eine Registry schieben
- Man kann Container starten und stoppen

Die „docker"-Variable unterstützt nicht immer die neuesten Docker-Features; beispielsweise fehlt das Bauen von Multi-Stage-Docker-

```
def call(def args) {
    docker.image('maven:3.5.0-jdk-8').inside {
        sh "mvn ${args} --batch-mode -V -U -e -Dsurefire.useFile=false"
    }
}
```

Listing 5

Images *[18]*. In diesem Fall kann man auf den CLI-Client von Docker zurückgreifen, beispielsweise mit „sh 'docker build …'".

Beim Vergleich der Listings 3 und 4 zeigt sich deutlich der Unterschied zwischen beschreibender („declarative") und imperativer („scripted") Syntax. Statt deklarativ am Anfang anzugeben, welcher Container zu verwenden ist, wird imperativ festgelegt, ab wo etwas in diesem Container auszuführen ist. Damit ist man auch flexibler: Während man in „declarative"-Syntax darauf beschränkt ist, die ganze Pipeline oder einzelne Stages in Containern auszuführen, kann man in „scripted"-Syntax beliebige Abschnitte in Containern ausführen.

Wie schon mehrfach erwähnt, kann man in „declarative"-Syntax allerdings auch innerhalb des „script"-Steps „scripted"-Syntax ausführen oder eigene Steps aufrufen, die in „scripted"-Syntax geschrieben sind. Das Auslagern wird nachfolgend genutzt, um den „mvn"-Step in der Shared Library *(Listing 1)* von Jenkins-Tools auf Docker umzustellen *(siehe Listing 5)*. Nach der Aktualisierung der Shared Library läuft jeder „mvn"-Step in den „scripted"- und „declarative"-Pipeline-Beispielen dann ohne weitere Änderung in einem Docker-Container. Zum Abschluss noch ein fortgeschrittenes Docker-Thema. Die „scripted"-Pipeline-Syntax lädt fast dazu ein, Docker-Container zu schachteln, also „Docker in Docker" auszuführen. Dies ist nicht ohne Weiteres möglich, da in einem Docker-Container zunächst kein Docker-Client verfügbar ist. Allerdings lassen sich mit „docker.withRun()" *[19]* mehrere Container gleichzeitig ausführen.

Es gibt jedoch auch Builds, die Docker-Container starten, beispielsweise mit dem Docker-Maven-Plug-in *[20]*. Damit können unter anderem Test-Umgebungen hochgefahren oder UI-Builds ausgeführt werden. Für diese Builds muss man tatsächlich „Docker in Docker" verfügbar machen. Dafür ist es jedoch nicht naheliegend, einen weiteren Docker-Host in einem Docker-Container zu starten, auch wenn das möglich wäre *[21]*. Stattdessen kann man den Docker-Socket des Build-Executor in den Docker-Container des Builds mounten.

Auch bei diesem Vorgehen sollte man sich jedoch gewisser sicherheitsrelevanter Einschränkungen bewusst sein *[22]*. Hier ist die oben erwähnte Trennung des Docker-Hosts des Masters von den Docker-Hosts der Build-Executors noch wichtiger. Damit der Zugriff auf den Socket möglich ist, sind außerdem einige Anpassungen am Docker-Image notwendig. So muss der User, mit dem der Container gestartet wird, in der „docker"-Gruppe sein, um Zugriff auf den Socket zu bekommen. Dazu müssen im Image User und Gruppe erzeugt werden *[16]*.

Fazit und Ausblick

Dieser Artikel beschreibt zum einen, wie man die Wartbarkeit der Pipeline durch Auslagerung von Code in eine Shared Library verbessern kann. Dieser Code ist dann auch wiederverwendbar und seine Qualität lässt sich durch Unit-Tests prüfen. Außerdem wird Docker als Werkzeug vorgestellt, mit dem Pipelines in einheitlicher Umgebung isolierter und unabhängiger von der Konfiguration der jeweiligen Jenkins-Instanz ausgeführt werden können. Diese nützlichen Werkzeuge schaffen die Grundlage für den Artikel in der nächsten Ausgabe, mit dem die Continuous-Delivery-Pipeline vollendet wird.

Weitere Informationen

[1] Jenkinsfile Repository GitHub: *https://github.com/triologygmbh/jenkinsfile*
[2] Triology Open Source Jenkins: *https://opensource.triology.de/jenkins/job/triologygmbh-github/job/jenkinsfile*
[3] Groovy Scripts vs. Classes: *http://docs.groovy-lang.org/latest/html/documentation/index.html#_scripts_versus_classes*
[4] cps-global-lib-plugin Pull Request 37: *https://github.com/jenkinsci/workflow-cps-global-lib-plugin/pull/37*
[5] Groovy Call Operator: *http://docs.groovy-lang.org/latest/html/documentation/#_call_operator*
[6] Pipeline GitHub Library Plug-in: *https://wiki.jenkins.io/display/JENKINS/Pipeline+GitHub+Library+Plugin*
[7] Cloudogu ces-build-lib: *https://github.com/cloudogu/ces-build-lib*
[8] JenkinsPipelineUnit: *https://github.com/jenkinsci/JenkinsPipelineUnit*
[9] Jenkins Shared-Libraries: *https://jenkins.io/doc/book/pipeline/shared-libraries*
[10] Standard-Build-Beispiel: *https://github.com/jenkinsci/pipeline-examples/blob/master/global-library-examples/global-function/standardBuild.groovy*
[11] Shared Library Demo: *https://github.com/jenkinsci/workflow-aggregator-plugin/tree/master/demo*
[12] Shared Library Docker: *https://github.com/docker/jenkins-pipeline-scripts*
[13] Shared Library Firefox: *https://github.com/mozilla/fxtest-jenkins-pipeline*
[14] Security Concerns when using Docker: *https://www.oreilly.com/ideas/five-security-concerns-when-using-docker*
[15] Pipeline-Syntax-Tools: *https://jenkins.io/doc/book/pipeline/syntax/#tools*
[16] „Cloudogu-ces-build-lib"-Docker: *https://github.com/cloudogu/ces-build-lib/blob/develop/src/com/cloudogu/ces/cesbuildlib/Docker.groovy*
[17] Global Variable Reference Docker: *https://opensource.triology.de/jenkins/pipeline-syntax/globals#docker*
[18] Jenkins Issue 44609 : *https://issues.jenkins-ci.org/browse/JENKINS-44609*
[19] Pipeline Docker: *https://jenkins.io/doc/book/pipeline/docker*
[20] Docker-Maven-Plug-in: *https://github.com/fabric8io/docker-maven-plugin*
[21] Do Not Use Docker In Docker for CI: *http://jpetazzo.github.io/2015/09/03/do-not-use-docker-in-docker-for-ci*
[22] Never Expose Docker Socket: *https://dzone.com/articles/never-expose-docker-sockets-period*

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. Derzeit arbeitet er am Cloudogu Ecosystem.