Jenkins

Coding Continuous Delivery – statische Code-Analyse mit SonarQube und Deployment auf Kubernetes et al. mit dem Jenkins-Pipeline-Plug-in

Johannes Schnatterer, Cloudogu GmbH

Die ersten drei Teile dieser Artikelserie in den letzten drei Ausgaben der Java aktuell zum Thema „Jenkins Pipelines" beschreiben Grundlagen, Performance und Werkzeuge wie Shared Libraries und Docker. Dieser letzte Teil widmet sich der Integration statischer Code-Analyse mittels SonarQube. Er zeigt, wie man kontinuierlich auf Kubernetes ausliefern kann, und gibt Anregungen für Continuous Delivery (CD) auf anderen Plattformen.

Die Pipeline-Beispiele aus den Vorgänger-Beiträgen werden in diesem Artikel sukzessive erweitert. Das Beispiel-Projekt ist weiter der „kitchensink"-Quickstart von WildFly. Alle Pipeline-Beispiele sind sowohl in „declarative"- als auch in „scripted"-Syntax realisiert. Den aktuellen Stand jeder Erweiterung kann man bei GitHub [1] nachverfolgen und ausprobieren. Hier gibt es für jeden Abschnitt 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. Die Nummerierung der Branches setzt sich damit aus den vorhergehenden Teilen fort. Docker war Nummer neun, also ist die statische Code-Analyse das zehnte Beispiel.

Statische Code-Analyse mit SonarQube

Die statische Code-Analyse erfasst gängige Fehlermuster, Code Style und Metriken (wie Code Coverage) anhand des Source- oder Byte-Codes. Auf Basis der Metriken können dann Quality Goals festgelegt werden, etwa „Code Coverage muss bei neuem Code größer 80 Prozent sein".

Diese Schritte sind gut automatisierbar und deshalb ein weiteres Mittel zur Qualitätssicherung in einer CD-Pipeline, neben den bereits gezeigten Unit- und Integrationstests. Weit verbreitet ist dabei das Werkzeug SonarQube (SQ), das in einer eigenen Web-Anwendung gekapselt die Möglichkeit zur statischen Code-Analyse bietet [3]. Dabei können über Plug-ins viele Programmiersprachen und die Regeln von Tools wie FindBugs, PMD und Checkstyle integriert werden.

Die Integration in die Jenkins Pipeline erfolgt per Jenkins-Plug-in *[4]*. Nach der Installation wird in Jenkins die SQ-Instanz konfiguriert, typischerweise mithilfe von URL und Authentication Token. In SQ wird zudem ein Webhook eingerichtet, der Jenkins über die Ergebnisse der Quality Gate Checks asynchron informiert. Der Webhook wird vom SQ-Plug-in auf Jenkins bereitgestellt: „/sonarqube-webhook". Die Analyse kann man dann unter anderem vom SQ-Plug-in für Maven auf Jenkins durchführen lassen. *Listing 1* zeigt, wie das mit „scripted"-Syntax in der Jenkins Pipeline abgebildet werden kann.

Das SQ-Plugin stellt auf der Pipeline verschiedene Steps bereit:

- „withSonarQubeEnv()" injiziert die in der Konfiguration für eine SQ-Instanz (hier mit der ID „sonarcloud.io") angegebenen Werte wie die URL als Environment-Variablen in den entsprechenden Block
- „waitForQualityGate()" wartet auf den Aufruf des Webhook, der über den Zustand des Quality Gate informiert

In *Listing 1* wird bei negativem Ergebnis der Build auf „unstable" (gelb) gesetzt. Hier ist alternativ auch ein Aufruf des „error()"-Steps möglich, der den Build auf „failed" (rot) setzt. Um zu verhindern, dass der Build hier unbegrenzt wartet, wird ein Timeout gesetzt. Ist beispielsweise kein Webhook konfiguriert, bricht der Build nach zwei Minuten ab. Auch hier empfiehlt es sich, die Logik in einen eigenen Step „analyzeWithSonarQubeAndWaitForQualityGoal()" auszulagern, um den Pipeline-Code lesbarer zu gestalten. In „declarative"-Syntax muss mindestens die Prüfung des Quality Gate in einen eigenen Step oder „script"-Block geschrieben werden.

Die offizielle Pipeline-Dokumentation *[5]* empfiehlt zwar, den Step „waitForQualityGate()" außerhalb eines Node durchzuführen, um diesen so wenig wie möglich zu blockieren. Dies führt allerdings zu schwerer wartbarem Code (siehe dazu die Beispiele bei GitHub im Branch 10a, jeweils für „scripted" und „declarative" *[1]*). In „declarative"-Syntax führt dies unter anderem dazu, dass jede Stage in einem eigenen Build Executor ausgeführt wird, womit sich zudem die Gesamtlaufzeit der Pipeline deutlich verlängert *[2]*. Da die Antwortzeiten von SQ typischerweise bei wenigen Millisekunden liegen, zeigt *Listing 1* den pragmatischen Weg innerhalb des Node.

Es gibt noch einige fortgeschrittene Themen im Umgang mit SQ, etwa die Ergebnisse der Analyse direkt als Kommentar in Pull Requests schreiben zu lassen oder die Verwendung des Branch-Features (nicht in der Community Edition). Diese Features sind zum Beispiel von der Shared Library „ces-build-lib" *[6]* komfortabel bereitgestellt.

Deployment

Wenn alle qualitätssichernden Maßnahmen erfolgreich waren, kann zum Abschluss der CD-Pipeline das Deployment erfolgen. Je nach Grad der Automatisierung steht hier am Ende das Deployment in Produktion. Um dabei die Risiken zu verringern, ist es empfehlenswert, mindestens eine Staging-Umgebung zu betreiben. Eine einfach umsetzbare Logik, um sowohl Staging als auch Produktion automatisiert einzurichten, ist die Verwendung von Branches im Source Code Management (SCM).

```
node {
    stage('Statical Code Analysis') {
        analyzeWithSonarQubeAndWaitForQualityGoal()
    }
}
// ...
void analyzeWithSonarQubeAndWaitForQualityGoal() {
    withSonarQubeEnv('sonarcloud.io') {
        mvn "${SONAR_MAVEN_GOAL} -Dsonar.host.
url=${SONAR_HOST_URL} -Dsonar.login=${SONAR_AUTH_TOKEN}
${SONAR_EXTRA_PROPS} "
    }
    timeout(time: 2, unit: 'MINUTES') {
        def qg = waitForQualityGate()
        if (qg.status != 'OK') {
            currentBuild.result = 'UNSTABLE'
        }
    }
}
```

Listing 1

Viele Teams arbeiten mit Feature-Branches oder Git Flow, in denen der integrierte Entwicklungsstand auf dem Develop-Branch zusammenfließt und der Master-Branch die produktiven Versionen enthält. Darauf kann man einfach seine CD-Strategie aufbauen: Jeder Push auf Develop führt zu einem Deployment auf die Staging-Umgebung, jeder Push auf Master geht in Produktion. So hat man stets die letzte integrierte Version auf Staging und kann dort funktionale oder manuelle Tests durchführen, bevor man durch einen Merge auf Master das Deployment in Produktion anstößt. Zudem ist ein Deployment pro Feature-Branch denkbar.

Eine solche Deployment-Logik lässt sich mit Jenkins Pipelines einfach realisieren, da man den Branch-Namen in Multibranch Builds aus dem Environment abfragen kann (siehe *Listing 2*, Stage „deploy"). Wohin und wie man Software einrichtet, hängt vom Projekt ab. In den letzten Jahren haben sich Container Orchestration Platforms als flexibles Mittel für DevOps-Teams erwiesen. Hier hat sich Kubernetes (K8s) als De-facto-Standard herauskristallisiert, weshalb dieser Artikel exemplarisch das Deployment auf K8s beschreibt. Um eine Anwendung auf K8s einzurichten, sind vier Schritte notwendig:

1. Versionsnamen festlegen
2. Docker-Image bereitstellen (Image bauen, mit Version als Tag und in Registry hochladen)
3. Image-Version in Deployment-Beschreibung (typischerweise in YAML) aktualisieren
4. YAML-Datei auf K8s-Cluster anwenden.

Listing 2 zeigt die Umsetzung dieser Schritte in „scripted"-Syntax.

Der Versionsname lässt sich mit Groovy erzeugen, etwa als Zeitstempel in *Listing 2*. Hier könnte man durch Anhängen des Git Commit Hash mehr Eindeutigkeit schaffen. Denkbar wäre es außerdem, aus der Pipeline einen Git Tag zu setzen. Um diese Version auch in Maven zu verwenden, bietet sich die Nutzung der ab Maven 3.5.0 verfügbaren „CI Friendly Versions" *[8]* an. *Listing 3* zeigt, wie dies in der „pom.xml" umgesetzt wird.

Im Build wird die Version dann mithilfe des Arguments „-Drevision" *(siehe Listing 2)* an Maven übergeben. Die weiteren Schritte in *Listing 2* sind in einem eigenen Step „deployToKubernetes()" realisiert. Dieser wird jedoch erst nach der Prüfung daraufhin, ob der Build noch

```
node {
    String versionName = createVersion()
    stage('Build') {
        mvn "clean install -DskipTests -Drevision=${versionName}"
    }
    // ...
    stage('Deploy') {
        if (currentBuild.currentResult == 'SUCCESS') {
            if (env.BRANCH_NAME == "master") {
                deployToKubernetes(versionName, 'kubeconfig-prod', 'hostname.com')
            } else if (env.BRANCH_NAME == 'develop') {
                deployToKubernetes(versionName, 'kubeconfig-staging', 'staging-hostname.com')
            }
        }
    }
}
String createVersion() {
    String versionName = "${new Date().format('yyyyMMddHHmm')}"

    if (env.BRANCH_NAME != "master") {
        versionName += '-SNAPSHOT'
    }
    currentBuild.description = versionName
    return versionName
}
void deployToKubernetes(String versionName, String credentialsId, String hostname) {

    String dockerRegistry = 'your.docker.registry.com'
    String imageName = "${dockerRegistry}/kitchensink:${versionName}"
    docker.withRegistry("https://${dockerRegistry}", 'docker-reg-credentials') {
        docker.build(imageName, '.').push()
    }

    withCredentials([file(credentialsId: credentialsId, variable: 'kubeconfig')]) {
        withEnv(["IMAGE_NAME=${imageName}"]) {
            kubernetesDeploy(
                    credentialsType: 'KubeConfig',
                    kubeConfig: [path: kubeconfig],
                    configs: 'k8s/deployment.yaml',
                    enableConfigSubstitution: true
            )
        }
    }

    timeout(time: 2, unit: 'MINUTES') {
        waitUntil {
            sleep(time: 10, unit: 'SECONDS')
            isVersionDeployed(versionName, "http://${hostname}/rest/version")
        }
    }
}
boolean isVersionDeployed(String expectedVersion, String versionEndpoint) {
    def deployedVersion = sh(returnStdout: true, script: "curl -s ${versionEndpoint}").trim()
    return expectedVersion == deployedVersion
}
```

Listing 2

stabil ist, aufgerufen. Wenn das Quality Gate fehlschlug, soll natürlich nicht eingerichtet werden.

Das Bauen und Hochladen des Image lässt sich dank der in der letzten Ausgabe beschriebenen Docker-Integration leicht mit Jenkins-Bordmitteln realisieren. Dabei muss man sich bei der Docker-Registry authentifizieren. Dazu hinterlegt man in Jenkins Username und Password-Credentials an („docker-reg-credentials" in *Listing 2*). Deren Herkunft hängen vom Anbieter der Registry ab, beispielsweise ist das Passwort bei der Google-Container-Registry eine JSON-Datei *[7]*, die man in einfachen Anführungszeichen ohne Zeilenumbrüche in Jenkins einfügt.

Um den Versions-Namen in die YAML-Datei zu schreiben, kann man einen eigenen Step zum Ersetzen im Jenkinsfile schreiben oder man verwendet ein Plug-in. Eine komfortable Möglichkeit ist das „kubernetes-cd-plugin" *[9]*. Es stellt den Step „kubernetesDeploy()" zur Verfügung, der YAML-Dateien filtert und direkt auf den Cluster anwendet. Dabei werden alle Einträge mit der „$VARIABLE"-Syntax in den YAML-Dateien durch entsprechende Environment-Variablen aus der Jenkins Pipeline ersetzt (in *Listing 2*, zum Beispiel „IMAGE_NAME").

Um die YAML-Datei auf den Cluster anwenden zu können, muss sich das Plug-in beim K8s-Master authentifizieren. Dafür erzeugt man einen K8s-Service-Account für Jenkins und legt dessen Rechte mit Role-Based Access Control fest. *Listing 4* zeigt exemplarisch, wie

```
<project>
    <version>${version}</version>
    <properties>
        <version>-SNAPSHOT</version>
    </properties>
</project>
```

Listing 3

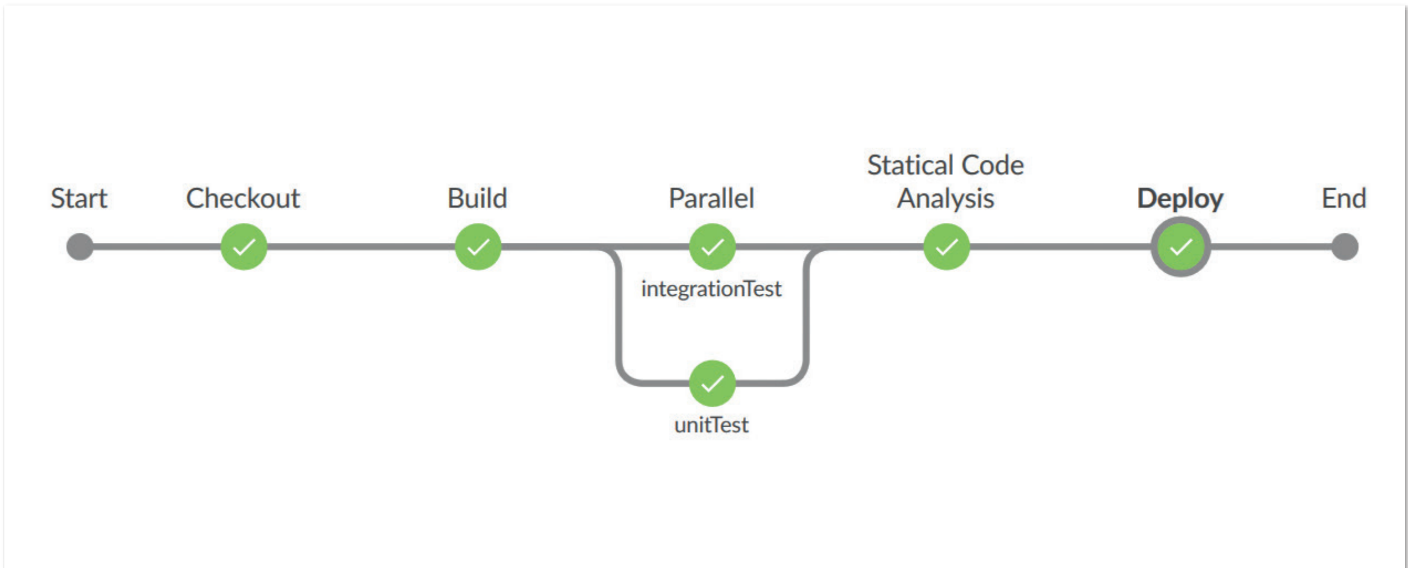

Abbildung 1: Die finale Pipeline im „Blue Ocean"-Theme

man imperativ einen Service-Account auf einen K8s-Namespace einschränkt. Für die deklarative Variante (in YAML) siehe *[1]*.

Mit diesem Service-Account kann man über das K8s-HTTP-API, „kubectl" oder „kubernetes-cd-plugin" auf den Cluster zugreifen. An das Plug-in übergibt man den Service-Account als „kubeconfig"-Datei. Diese lässt sich mit einem Script von GitHub *[10]* erstellen, wie die letzte Zeile in *Listing 4* zeigt. Die durch das Script erstellte Datei „kubeconfig" lädt man in Jenkins in ein Secret File Credential hoch, etwa mit der ID „kubeconfig-prod" *(siehe Listing 2)*. Wenn man Staging-Umgebungen in einem anderen Namespace hat, würde man an dieser Stelle weitere „kubeconfig"-Dateien für diese erstellen (wie „kubeconfig-staging" in *Listing 2)*.

Da das Anwenden der Datei auf den Cluster bei K8s nur das serverseitige Deployment anstößt, ist danach noch nicht klar, ob dieses erfolgreich war. Deshalb wird in *Listing 2* abschließend geprüft, ob die neue Version erreichbar ist. Dazu muss die Anwendung den Versions-Namen bereitstellen. Wie man dies mit Maven und REST erledigt, zeigt zum Beispiel dieser Blog-Post *[11]*. In *Listing 2* ist zu sehen, wie man den Versions-Namen abfragt und vergleicht, ob die Version mit der gewünschten übereinstimmt. Taucht sie nach einer bestimmten Zeit nicht auf, schlägt der Build fehl und die Entwickler werden von Jenkins informiert. In einem solchen Fall zahlt sich die Verwendung von K8s aus: Durch dessen Rolling-Update-Strategie bleibt die Anwendung weiterhin eingeschränkt verfügbar. In *Listing 2* ist der Hostname hart codiert. Alternativ kann man die externe IP-Adresse des Service mittels „kubectl" abfragen, siehe *[1]*. Die Umsetzung des Deployments in „declarative"-Syntax erfolgt wie in *Listing 2*, bis auf folgende Ausnahmen:

- „createVersion()" muss in einem „steps"-Block (etwa innerhalb der „build"-Stage) aufgerufen werden und schreibt sein Ergebnis nach „env.versionName", da keine Variablen im „steps"-Block möglich sind
- Die Prüfung, ob der Build noch stabil ist, kann in einer „when"-Directive erfolgen (siehe Artikel in der vorletzten Ausgabe)
- Die Prüfung des Branch in der „Deploy"-Stage muss innerhalb eines „steps"- und „script"-Blocks oder in einem eigenen Step stattfinden

Das komplette Beispiel findet man bei GitHub *[1]*. Selbstverständlich kann man Jenkins Pipelines nicht nur auf K8s einrichten; beispielsweise ist das Deployment auf die Container Orchestration Platform Docker Swarm dank des eingebauter Docker-Supports einfach *[12]*. Docker selbst kann man außerdem nutzen, um einfache Staging-Umgebungen aufzubauen, indem man einen Rechner mit Docker-Host als Jenkins-Worker einbindet und darauf aus der Pipeline die Container des Staging-Systems startet. Auch Deployments auf PaaS-Plattformen sind möglich, so gibt es für CloudFoundry ein Plug-in mit Pipeline-Unterstützung *[13]*. Außer Web-Anwendungen kann man auch andere Arten von Anwendungen kontinuierlich ausliefern. Zum Abschluss einige Anregungen aus der Praxis:

- Java-Libraries lassen sich mit wenigen Zeilen Pipeline-Code nach Maven Central einrichten. Ein Beispiel ist der „test-data-loader" [14], der die Shared Library „ces-build-lib" [6] verwendet.
- **Docs-as-Code**
  Eine in einer Markup-Sprache verfasste und im SCM gespeicherte Dokumentation kann automatisiert in ein fertiges Dokument überführt werden. Dieses Beispiel [15] zeigt, wie man aus Markdown mit Pandoc verschiedene Dokumenten-Formate

```
kubectl create namespace jenkins-ns
kubectl create serviceaccount jenkins-sa --namespace=jenkins-ns
kubectl create rolebinding jenkins-ns-admin --clusterrole=admin --namespace=jenkins-ns --serviceaccount=jenkins-
ns:jenkins-sa
./create-kubeconfig jenkins-sa --namespace=jenkins-ns > kubeconfig
```

Listing 4

wie PDF oder ODT erzeugt. Über den Pandocs-Template-Mechanismus können die Dokumente im Corporate Design gerendert werden. In diesem Beispiel wird das Build-Tool Gulp verwendet, damit man den Build auch lokal durchführen kann. Im Jenkinsfile wird die für Gulp notwendige Umgebung (node.js, yarn) in einem „yarn"-Container bereitgestellt. Darin nutzt Gulp den „cloudogu/pandoc"-Container [16] zur Dokumenten-Erstellung. Daher ergibt sich hier die im letzten Teil beschriebene „Docker in Docker"-Herausforderung. Um weitere Container aus dem „yarn"-Container zu starten, werden der Docker Socket durchgereicht und der Docker-Client installiert. Auch dies kann in wenigen Zeilen durch „ces-build-lib" [6] gelöst werden. Wenn man das Markup in einem Git-basierten Wiki wie Gollum [17] oder Smeagol [18] editiert, wird direkt bei Speicherung im Wiki durch die CD-Pipeline ein PDF ausgeliefert. Dadurch ist Docs-as-Code für Nicht-Entwickler besser zugänglich. Das funktioniert auch für Präsentationen. Dieses Beispiel [19] zeigt, wie man in Markdown Präsentationen mit „reveal.js" erstellen und in einer Maven Site (Nexus Repository) oder per Kubernetes (NGINX-Container) im Web bereitstellen kann.

- **Infrastructure as Code**
  Auch ganze virtuelle Maschinen können in der Cloud provisioniert werden, etwa mithilfe des Tools „Terraform". Damit wird beispielsweise die öffentliche Demo-Instanz des Cloudogu EcoSystem [20] mittels Blue-Green Deployment [21] täglich aktualisiert.

Fazit und Ausblick

Diese Artikelserie zeigt einige der Möglichkeiten, die das Jenkins-Pipeline-Plug-in bietet. Es kombiniert die bereits vorhandene große Auswahl an Jenkins-Plug-ins mit einer DSL zur Beschreibung von Build Jobs. So kann man diese als Code formulieren. Sie sind schneller verständlich, können im SCM verwaltet, einfacher wiederverwendet (beispielsweise durch Shared Libraries) und automatisch getestet werden. Durch Parallelisierung kann man außerdem vorhandene Ressourcen nutzen, um mit geringem Aufwand die Laufzeiten der Pipelines zu verkürzen. Die Docker-Integration ermöglicht ohne weitere Konfiguration die Benutzung von weiteren Werkzeugen und bietet die Möglichkeit, Images während des Deployments in eine Registry bereitzustellen.

Ob man Pipelines in „scripted"- oder „declarative"-Syntax beschreibt, bleibt Geschmackssache. Beim Schreiben der Beispiele für den Artikel fiel auf, dass gerade bei komplexeren Aufgaben die „declarative"-Lösung oft umständlicher, aber generell machbar war. Der Vorteil von „declarative"-Pipelines ist, dass sie besser in das „Blue Ocean"-Theme integriert sind und dort visuell editiert werden können.

Die finale Pipeline *(siehe Abbildung 1)* umfasst in beiden Varianten ungefähr 150 Zeilen. Dabei entspricht sie von der Komplexität her durchaus der eines echten Projekts.

An wenigen Stellen (wie Nightly Builds, Möglichkeiten für Unit- und Integrationstests der Pipeline) zeigt sich zwar, dass noch nicht alles perfekt ist. Dennoch ist das Pipeline-Plug-in die wichtigste Neuerung der letzten Jahre für Jenkins und sorgt dafür, dass wir den altgedienten Butler auch weiterhin für moderne Software-Entwicklung einsetzen können.

Weitere Informationen

[1] *https://github.com/triologygmbh/jenkinsfile*
[2] *https://opensource.triology.de/jenkins/job/triologygmbh-github/job/jenkinsfile/*
[3] J. v. Gizycki, Statische Code-Analyse mit SonarQube, Java aktuell 04/2017
[4] *https://docs.sonarqube.org/display/SCAN/Analyzing+with+SonarQube+Scanner+for+Jenkins*
[5] *https://jenkins.io/doc/pipeline/steps/sonar*
[6] *https://github.com/cloudogu/ces-build-lib*
[7] *https://cloud.google.com/container-registry/docs/advanced-authentication*
[8] *https://maven.apache.org/maven-ci-friendly.html*
[9] *https://plugins.jenkins.io/kubernetes-cd*
[10] *https://github.com/zlabjp/kubernetes-scripts/blob/master/create-kubeconfig*
[11] *https://www.triology.de/blog/versionsnamen-mit-maven-auslesen-des-versionsnamens*
[12] *https://jenkins.io/doc/book/pipeline/docker/#using-a-remote-docker-server*
[13] *https://plugins.jenkins.io/cloudfoundry*
[14] *https://github.com/triologygmbh/test-data-loader*
[15] *https://github.com/cloudogu/continuous-delivery-docs-example*
[16] *https://github.com/cloudogu/docker-pandoc*
[17] *https://github.com/gollum/gollum*
[18] *https://github.com/cloudogu/smeagol*
[19] *https://github.com/cloudogu/continuous-delivery-slides-example*
[20] *https://demo.cloudogu.net*
[21] *https://www.martinfowler.com/bliki/BlueGreenDeployment.html*

Johannes Schnatterer
johannes.schnatterer@triology.de

Johannes ist Software-Entwickler und Solution Architect bei der Cloudogu GmbH und dort Teil des DevOps-Teams, um das Backend des Cloudogu EcoSystem zu betreiben. Er ist Continuous-Delivery-Fan, fokussiert auf Software-Qualität, hat einen ausgeprägten Open-Source-Enthusiasmus und ist überzeugt, dass prägnante Dokumentation entscheidend sein kann.