JAVA Mag

Java magazin

Java | Architektur | Software-Innovation

JAVA EE 8

DIE MACHT DER ACHT

Dabarti CGI/Shutterstock.com

Pipeline as Code mit Jenkins 2

Der Praxischeck

Das Major-Release von Jenkins im April 2016 machte die Entwicklung von Delivery-Pipelines zur Kernfunktion. Pipelines lassen sich jetzt als Code implementieren, um die Automatisierung der Strecke vom Commit bis ins Produktionssystem besser zu unterstützen [1]. Anfang dieses Jahres sind weitere Neuerungen erschienen, die die Funktionalität ausbauen: eine deklarative Pipelinesyntax und eine moderne Oberfläche, das Blue Ocean UI. Wir beleuchten die neuen Möglichkeiten zur Pipelineentwicklung genauer und berichten, wie sie sich in der Praxis bewähren.

von Hendrik Brinkmann und Philip Stroh

Als CI/CD-Werkzeug der ersten Generation hat Jenkins seit einiger Zeit Konkurrenz durch neuere Produkte wie Travis CI oder Circle CI erhalten. Diese neue Toolgeneration setzt auf Konzepte wie Convention over Configuration und Auto-Discovery, um den Konfigurationsaufwand für Integrations- und Delivery-Strecken zu verringern. Meist sehen diese Produkte eine Konfigurationsdatei vor, die im Sourcecode-Repository der Anwendung abgelegt wird und die Integrationspipeline steuert. Jenkins 2 greift einige dieser Konzepte auf, um sie mit seinen Stärken zu verbinden. Hierzu zählen Flexibilität, Erweiterbarkeit und vielfältige Integrationsmöglichkeiten durch die mehr als 1 000 Plug-ins [2].

Durch das Major-Update auf 2.0 wurde Pipeline as Code fester Bestandteil von Jenkins. Zuvor als Workflow-Plug-in bekannt, ist das Pipeline-Plug-in nun standardmäßig mit erweiterter Funktionalität im Jenkins enthalten. Gleichzeitig wurde mit der neuen Version die Philosophie verfolgt, das initiale Set-up zu vereinfachen, indem ein Paket von Standard-Plug-ins direkt bei der Installation dabei ist. Dadurch soll dem User ein Werkzeug an die Hand geben werden, das schon out of the Box viele gängige Anwendungsfälle abdeckt.

Jenkins ist als Open-Source-Produkt kostenlos erhältlich. CloudBees, die Hauptentwickler von Jenkins, bieten des Weiteren kostenpflichtig gehostete Jenkins-Instanzen sowie Enterprise-Support an. Das Projekt wird von CloudBees und einer großen Community stark vorangetrieben, sodass auch für die Zukunft eine aktive Weiterentwicklung zu erwarten ist. Jenkins wird in zwei Versionszweigen angeboten: eine Long-Term-Support-Version (LTS), die auf Stabilität ausgelegt ist, und einen Weekly Release, der die neuesten Funktionen enthält. Für den Betrieb eines produktiven Systems empfiehlt sich der LTS-Zweig. Dieser hängt dem Weekly Release vom Funktionsumfang nur unwesentlich hinterher und enthält alle wichtigen Hotfixes. Neben den klassischen Installationsmöglichkeiten über einen Paketmanager (apt und yum) oder das Deployment als WAR existiert ein offizieller Docker-Container. Damit ist es einfach, eine Jenkins-Instanz zu Testzwecken in einer lokalen Umgebung zu installieren.

Pipeline as Code im Überblick

Bevor wir uns mit dem Konzept Pipeline as Code und dessen Umsetzung in Jenkins 2 beschäftigen, wollen wir uns ins Gedächtnis rufen, wie man eine Integrations- oder Delivery-Pipeline bisher mit Jenkins abbilden konnte. Klassische Jenkins-Jobs, sogenannte Freestyle Jobs, werden über das Webinterface konfiguriert. Oft erstellt man viele einzelne Jobs und verkettet diese miteinander, um eine Pipeline abzubilden. Ein erfolgreich durchgelaufener Job stößt dabei weitere Jobs zur Verarbeitung an. Um beispielsweise alle Integrationsschritte wie den Build-Prozess, Unit- und Integrationstests sowie GUI-Tests abzudecken, sind in der klassischen Variante viele einzelne Jobs notwendig. Die Konfiguration ist über diese Jobs verteilt, und eine visuelle Darstellung der Pipeline lässt sich nur über Umwege wie einen Dependency-Graph erreichen.

Genau hier setzt das Pipeline-Plug-in an. Pipeline as Code bedeutet, dass die Jobs nicht mehr über das User Interface konfiguriert, sondern als Code hinterlegt werden. Dazu haben die Jenkins-Entwickler eine eigene Groovy-DSL (Domain Specific Language) entworfen. Diese bietet bereits ein Set an Standardfunktionen und lässt sich durch Plug-ins und eigene Bibliotheken erweitern.

Die wesentlichen Vorteile von Pipeline as Code in Jenkins 2 sind, dass die gesamte Pipeline übersichtlich in einem Skript abgebildet werden kann, anstelle vieler einzelner Jobs. Die DSL lässt sich um zusätzliche Groovy-Logik erweitern, z. B. Schleifen, Fallunterscheidungen oder Variablen. Und Groovy ist für Java-Entwickler leicht zu erlernen. Die Konfiguration wird mit dem Quellcode im SCM eingecheckt und ist somit versioniert. Eine Übersichtsmatrix bietet eine visuelle Darstellung der einzelnen Pipelineschritte. Des Weiteren lassen sich Pipelines pausieren, z. B. bis zur Verarbeitung einer Usereingabe, und sind robust gegen geplante oder ungeplante Restarts. Möglich wird das durch die Groovy CPS Engine von Jenkins. Diese interpretiert den Code der Pipeline und sorgt dafür, dass bei der Step-Ausführung der Zustand der Pipeline serialisiert wird.

Groovy DSL aka Scripted Pipeline

Pipelineskripte werden mit der Groovy DSL erstellt und in einem so genannten Jenkinsfile abgelegt. Sie können neben den speziellen Pipelinefunktionen auch normalen Groovy-Code enthalten. Mit der Einführung der Syntax für die Declarative Pipeline wurde für die Groovy DSL der Begriff Scripted Pipeline eingeführt. Der Pipelinecode läuft standardmäßig in der Groovy Sandbox von Jenkins. Diese unterbindet die Ausführung von möglicherweise sicherheitskritischen Groovy-Befehlen. Damit die Nutzer bei Bedarf dennoch auf diese Befehle zurückgreifen können, gibt es das Feature Script Approval [3].

An dieser Stelle wollen wir einen kurzen Überblick über die wichtigsten DSL-Begrifflichkeiten geben: Node, Stage und Step. Ein Node belegt einen Executor-Slot in Jenkins und stellt einen Workspace zur Verfügung. Viele DSL-Befehle lassen sich nur innerhalb eines Nodes ausführen. Nodes ermöglichen die Lastenverteilung, wenn mehrere Jenkins-Instanzen in einem Master-Agent-Verbund agieren. Beispielsweise werden ressourcenintensive Teile des Build-Prozesses nicht auf dem Master ausgeführt, sondern auf einen anderen Jenkins-Knoten ausgelagert, und blockieren damit nicht dauerhaft Exekutoren auf dem Master. Stages ermöglichen die logische Segmentierung der Pipeline. Eine Stage besitzt ein Label und umschließt einen zusammengehörigen Teil der Pipeline. Stages werden in der Übersichtsmatrix (Pipeline Stage View) visuell hervorgehoben. Eine Integrationspipeline kann beispielsweise aus den folgenden Stages bestehen: Build → Integrationstests → Deployment auf einen Entwicklungsserver → GUI-Tests. Step ist der Oberbegriff für eine Vielzahl von Pipelineschritten, die innerhalb einer Stage ausgeführt werden. Standardmäßig werden bereits Steps wie *sh* (Ausführen von Shell-Kommandos), *git* (Auschecken eines Git-Repos) oder *dir* (Wechseln des Verzeichnisses) mitgeliefert. Weitere Steps stellen Plug-ins zur Verfügung. Dazu müssen die Plug-in-Entwickler das Jenkins-Pipeline-API in ihren Projekten implementieren. Die meisten populären Plug-ins unterstützen bereits die neue Pipeline.

Die erste Pipeline erstellen

Ein Pipelinejob wird initial, wie bei klassischen Freestyle-Jobs, über das Webinterface angelegt. Im Dialog für das Erstellen neuer Jobs sind zwei Einträge neu hinzugekommen: Pipeline und Multibranch Pipeline. Wählt man Pipeline aus, kann man seinen Pipelinecode direkt im Job hinterlegen. Dies eignet sich besonders zum Testen neuer Pipelineskripte oder -Snippets. Dauerhaft sollten Pipelineskripte zusammen mit den Projektdateien im SCM abgelegt und aus

Abb 1: Die Beispielpipeline in der Pipeline Stage View

diesem Repository ausgecheckt werden. Im Pipelinejob wird dazu das entsprechende Repository mit dem entsprechenden Jenkinsfile konfiguriert. Somit liegt die Pipelinekonfiguration zusammen mit dem Quellcode unter Versionskontrolle, und Änderungen lassen sich einfach nachvollziehen. Die Jenkins-Dokumentation bietet eine detaillierte Anleitung für das Erstellen der ersten Pipeline [4].

Bei der Auswahl von Multibranch Pipeline muss im Konfigurationsdialog lediglich ein SCM-Repository angegeben werden. Jenkins scannt dann automatisch alle Branches nach einer Datei namens Jenkinsfile. Wird ein Jenkinsfile gefunden, wird der entsprechende Branch automatisch anhand der Anweisungen im Jenkinsfile gebaut. Es wird somit für jeden Branch, in dem ein Jenkinsfile vorhanden ist, ein Pipelinejob angelegt.

Eine sehr vereinfachte Pipeline könnte aussehen wie in Listing 1. In dieser Pipeline wird der Code einer Spring-Boot-Anwendung ausgecheckt, die Anwendung per Maven gebaut und auf einem Entwicklungsserver deployt. Das Deployment auf den Server erfolgt via SCP, und die Anwendung wird via SSH gestartet.

Um diese Pipeline umzusetzen, definieren wir zu Beginn globale Variablen, z. B. den Git-URL unseres Projekts. Das erhöht die Lesbarkeit des Codes und vereinfacht Wartung und Anpassungen. Im Verlauf der Pipeline können wir jederzeit auf diese Variablen zugreifen. Als Nächstes wird ein *node*-Block angelegt. Der *node*-Block sorgt bei der Ausführung für die Anlage eines Workspace und das Belegen eines Executor-Slots. Nun zerlegen wir unsere Pipeline in sinnvolle Bestandteile und verpacken diese in Stages. Die Stages werden beim Start des Jobs in der Pipeline Stage View visualisiert. **Abbildung 1** zeigt einen Screenshot unserer Beispielpipeline in dieser Ansicht. Innerhalb einer Stage können wir beliebige Steps ausführen. So können wir beispielsweise Kommandos auf der Shell anstoßen oder per SSH Verbindung zu einem Remote-Server aufbauen.

Um eine Pipeline zu entwickeln, bietet die Jenkins-Oberfläche neben einer Syntaxhilfe einen Snippet-Generator an. Über diesen lassen sich einfach von allen

Listing 1: Eine einfache Pipeline

```
def branch = 'master'
def scmUrl = 'ssh://git@myScmServer.com/repos/myRepo.git'
def devServer = 'dev-myproject.mycompany.com'
def devServerPort = '8080'

node {
   stage('checkout git') {
     git branch: branch, credentialsId: 'GitCredentials', url: scmUrl
   }

   stage('build') {
     sh 'mvn clean package'
   }

   stage('deploy dev'){
     sshagent(['RemoteCredentials']) {
       sh "scp target/*.jar root@${devServer}:/opt/jenkins-demo.jar"
       sh "ssh root@${devServer} nohup java -Dserver.
                        port=${devServerPort} -jar /opt/jenkins-demo.jar &"
     }
   }
}
```

Listing 2: Stages parallel ausführen

```
stage ("test") {
  parallel (
    "unit tests": { sh 'mvn test' },
    "integration tests": { sh 'mvn integration-test' }
  )
}
```

Listing 3: Stage mit Testumgebung

```
stage ("deploy staging") {
  timeout (time: 14, unit: 'DAYS') {
    input 'Deploy to staging environment?'
    node {
      // execute our previous deploy steps here
      ...
    }
  }
}
```

verfügbaren Pipeline-Steps vorbelegte Codestücke erzeugen. Um die Dauer der einzelnen Verarbeitungsschritte zu analysieren, gibt es die Pipeline Step View. Hier wird für jeden Step die Ausführungszeit protokolliert. Eine vollständige Auflistung aller Steps ist der Dokumentation zu entnehmen [5]. Um einen Teil der Möglichkeiten zu veranschaulichen, haben wir einige Codebeispiele ausgewählt, mit denen wir unsere initiale Pipeline um gängige Anwendungsfälle erweitern können. Es sollte auch noch erwähnt werden, dass es über die Groovy DSL möglich ist, Optionen, Trigger und Parameter des Pipelinejobs zu konfigurieren und so die manuellen Schritte bei der Jobkonfiguration stark zu reduzieren.

Eine wichtige DSL-Funktion ist *parallel*, über die sich Verarbeitungsschritte zeitgleich abarbeiten lassen. Listing 2 zeigt die Nutzung der Funktion innerhalb einer Stage. Es ist aber auch möglich, Stages parallel auszuführen. Leider gibt es in diesem Fall eine Einschränkung der Stage View. Diese kann Stages nicht parallel darstellen, sondern zeigt sie hintereinander an.

Außerdem lassen sich in der Pipeline Userabfragen über einen *input*-Step umsetzen. Dieser führt dazu, dass die Pipeline bis zur Ausführung der Useraktion pausiert. Listing 3 zeigt eine Stage, in der auf eine Testumgebung deployt wird. Vor Durchführung des Deployments soll eine Bestätigung durch einen Benutzer erfolgen. Wichtig hierbei ist, dass der *input*-Step nicht innerhalb eines *node*-Blocks verwendet wird. Ansonsten wird bis zur Bestätigung durch den User ein Executor-Slot blockiert. Zudem sollte die Bestätigung eines Inputs über einen Time-out notfalls abgebrochen werden, wenn lange Zeit kein Userinput ankommt.

Ein weiterer häufiger Anwendungsfall ist der Versand einer Benachrichtigung bei einer fehlgeschlagenen Pipelineausführung. Das können Slack- oder Hipchat-Benachrichtigungen sein oder ganz klassisch der Versand einer Mail, wie in unserem Beispiel in Listing 4. Hierbei sieht die Scripted-Pipeline-Syntax die Behandlung von Fehlern in der Build-Ausführung mit try/catch-Blöcken vor.

Erweiterung mit Shared Libraries

Sobald mehrere Projekte auf Pipeline as Code umgestellt werden, wird man feststellen, dass in den verschiedenen Pipelines gleiche Probleme und Aufgaben anfallen. Es entsteht daher schnell der Wunsch, bestimmte Hilfsfunktionen und Prozessschritte zentral bereitzustellen. Jenkins stellt dazu das Pipeline-Plug-in Shared Groovy Libraries [6] zur Verfügung. Dieses Plug-in ermöglicht das Erstellen eigener Methoden, die innerhalb einer Pipeline ähnlich wie die Built-in Steps verwendet werden können.

Eine Shared Library wird in Form eines Git-Repos – andere SCMs sind auch möglich – hinterlegt und lässt sich global oder auf Folder-Ebene in Jenkins einbinden. Dabei können beliebig viele Repositorys konfiguriert werden. Die Struktur der Libraries ist vorgegeben und in der Dokumentation des Plug-ins genauer beschrieben. Je nachdem, ob das Repository in der globalen Jenkins-Verwaltung oder in einem Folder verknüpft ist, sind die Funktionen der Bibliothek für alle Jobs oder nur für die Jobs innerhalb eines Folders sichtbar.

Die Versionierung der Bibliotheken erfolgt über Branches oder Tags im Repository. Im Jenkinsfile kann danach optional per *@Library*-Annotation eine konkrete Version einer Bibliothek referenziert werden. Somit kann der Entwickler steuern, wann welche Änderungen in einer Shared Library in welchem Pipelinejob sichtbar werden. Des Weiteren besteht die Möglichkeit, die globalen Funktionen über ein bestimmtes Schema zu dokumentieren. Die Dokumentation wird automatisch geladen und in der Jenkins-Oberfläche über die Syntaxhilfe bereitgestellt.

Exemplarisch werden in Listing 5 die Deployment-Befehle aus der Beispielpipeline in eine globale Funktion ausgelagert und erweitert. Da unsere Anwendung per Spring Boot Actuator eine Shutdown-Funktion und einen Healthcheck anbietet, soll eine bereits laufende Anwendungsversion vor dem Deployment gestoppt werden. Nach dem Start der Anwendung soll per REST-Schnittstelle der Status der Anwendung validiert

Listing 4: Benachrichtigungen per Mail

```
...
try {
  node {
    stage ("checkout git") {
      git branch: branch, credentialsId: 'GitCredentials', url: scmUrl
    }
    // include other stages from our pipeline here
    ...
    }
} catch (err) {
  mail to: 'team@example.com', subject: 'Pipeline failed', body:
                                                    "${env.BUILD_URL}"
  throw err
}
```

Listing 5: „deploy.groovy“

```
def call(def server, def port) {
  httpRequest httpMode: 'POST', url: "http://${server}:${port}/shutdown",
                                                    validResponseCodes: '200,408'
  sshagent(['RemoteCredentials']) {
    sh "scp target/*.jar root@${server}:/opt/jenkins-demo.jar"
    sh "ssh root@${server} nohup java -Dserver.port=${port} -jar /opt/
                                                    jenkins-demo.jar &"
  }
  retry (3) {
    sleep 5
    httpRequest url:"http://${server}:${port}/health",
          validResponseCodes: '200', validResponseContent: '"status":"UP"'
  }
}
```

werden. Das Beispiel zeigt durch Verwendung von *retry*, *sleep* oder *httprequest*, dass sich in einer Shared Library auch alle regulären Pipeline-Steps verwenden lassen.

Wird die Library-Funktion als sogenannte globale Variable in einer Datei *deploy.groovy* hinterlegt, kann sie im Jenkinsfile über den *deploy*-Aufruf genutzt werden. Der Dateiname definiert also in diesem Fall, unter welchem Namen die Funktion bereitgestellt wird. Listing 6 zeigt, wie wir die neue Version der Bibliothek in unsere Pipeline einbinden können. Es ist nun sehr komfortabel, das Deployment-Vorgehen auf weitere Umgebungen auszuweiten.

In Shared Librarys lassen sich nicht nur einfache Hilfsfunktionen auslagern, sondern auch das Erzeugen von ganzen Stages einer Pipeline. Das ist sinnvoll, wenn Pipelines unterschiedlicher Projekte immer dieselben Stages durchlaufen. Es ist sogar möglich, alle Schritte einer Pipeline in eine globale Funktion auszulagern und hierfür eine eigene kleine DSL zu erstellen. Diese Möglichkeit wird in der Plug-in-Dokumentation ebenfalls genauer beschrieben.

Declarative Pipeline nutzen

Obwohl die Nutzung der Scripted Syntax keine tieferen Groovy-Kenntnisse voraussetzt, kann die Groovy DSL eine Einstiegshürde für Nutzer mit wenig Programmiererfahrung sein. Auch aus diesem Grund bietet Jenkins seit Februar 2017 für das Erstellen einer Pipeline eine neue Syntaxerweiterung an: die Declarative Pipeline Syntax. Diese soll die Pipelineerstellung durch eine vordefinierte Struktur erleichtern. Die bisher im Artikel gezeigten Code-Snippets basierten auf der Scripted-Pipeline-Syntax. Listing 7 zeigt eine Pipeline in der neuen deklarativen Syntax, die sich an unseren bisherigen Beispielen orientiert.

Eine Declarative Pipeline beginnt mit einem *pipeline*-Block. Innerhalb dieses Blocks ist auf oberster Ebene nur die Verwendung so genannter Sektionen und Direktiven möglich. Hauptelement der Pipeline ist die *stages*-Sektion. Hier werden die Stages der Pipeline deklariert. Über Direktiven wie *agent* oder *environment* lassen sich die Ausführungsumgebung einer Pipeline sowie Optionen, Trigger und Parameter des Pipelinejobs konfigurieren. Der *agent* dient dabei zur Steuerung, auf welchem Node die Pipeline ausgeführt wird. Gibt es Aktionen, die nach dem Durchlauf aller Stages ausgeführt werden sollen, lassen sich diese in der *post*-Sektion definieren. Hier kann komfortabler als über eine try/catch-Anweisung auf das Ergebnis des Jobs reagiert werden. Eine Stage enthält die *steps*-Sektion. Dort und im Condition-Block der *post*-Sektion können die Steps der Groovy DSL verwendet werden. Auch der Aufruf eigener per Shared Library bereitgestellter Funktionen ist hier möglich. Eine Stage kann außerdem ebenfalls über Direktiven konfiguriert werden und eine *post*-Sektion enthalten.

Im Vergleich mit der Scripted Pipeline fällt auf, dass durch die feste Struktur die Verwendung von Groovy-Code in deklarativen Pipelines nicht direkt vorgesehen ist. Über einen extra *script*-Step ist es aber möglich, mit

Listing 6: Jenkinsfile

```
@Library('my-shared-library@1.0') _

node {
  ...

  stage('deploy dev'){
    deploy(devServer, devServerPort)
  }

  stage('deploy staging'){
    deploy(stagingServer, stagingServerPort)
  }
}
```

Listing 7: Deklarative Pipeline

```
pipeline {
  agent any
  environment {
    branch = 'master'
    scmUrl = 'ssh://git@myScmServer.com/repos/myRepo.git'
    devServer = 'dev-myproject.mycompany.com'
    devServerPort = '8080'
  }
  stages {
    stage('checkout git') {
      steps {
        git branch: branch, credentialsId: 'GitCredentials', url: scmUrl
      }
    }

    stage('build') {
      steps {
        sh 'mvn clean package'
      }
    }

    stage('deploy dev'){
      steps {
        deploy(devServer, devServerPort)
      }
    }
  }
  post {
    failure {
      mail to: 'team@example.com', subject: 'Pipeline failed', body:
                                                    "${env.BUILD_URL}"
    }
  }
}
```

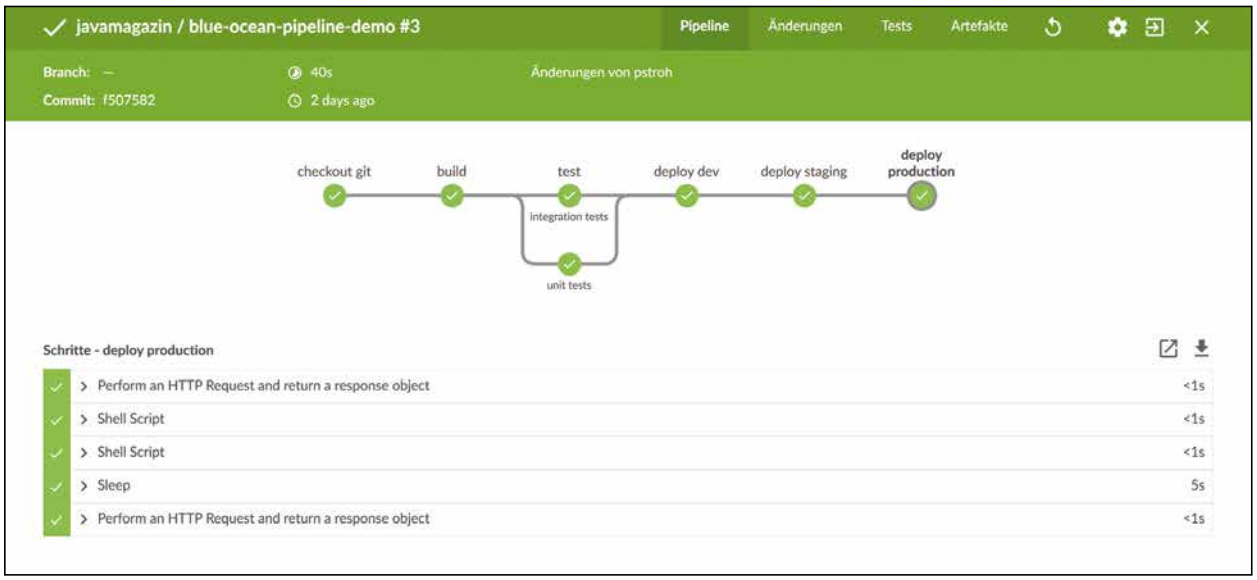

Abb. 2: Die Pipelinevisualisierung in Blue Ocean

den Strukturvorgaben in begrenzter Form zu brechen und innerhalb einer Stage Groovy-Code zu nutzen. Der restriktivere Aufbau der Pipelines hat, neben der vereinfachten Nutzung durch die Anwender, einen weiteren positiven Aspekt: Es wird den Jenkins-Entwicklern möglich, weitere Features anzubieten, die auf Basis der Scripted Pipeline, aufgrund der hohen Flexibilität und Erweiterbarkeit, nicht in der Form umsetzbar wären. So wird für die Declarative Pipeline zum einen eine Validierung bei der Erstellung angeboten, der Pipeline Linter [7], und zum anderen kann ein visueller Pipelineeditor bereitgestellt werden.

Es stellt sich aufgrund der neuen Syntax natürlich die Frage, ob die Scripted Pipeline somit überholt ist und in Zukunft nur noch der deklarative Ansatz weiterentwickelt wird. Das ist jedoch nicht zu befürchten, denn die deklarative Pipeline ist technisch eine Erweiterung auf Basis der bestehenden Groovy Engine. Die Jenkins-Macher weisen darauf hin, dass beide Stile in Zukunft weiter voll unterstützt werden. Für den Standardanwendungsfall ist die komfortablere und einfachere Declarative Pipeline vorgesehen, für fortgeschrittene Anwender und komplexere Anwendungsfälle hingegen ist die Scripted Pipeline die erste Wahl. Mehr zu den Unterschieden der beiden Syntaxvarianten und Hintergründe zur Einführung der Declarative Pipeline sind unter [8] aufgeführt.

Blue Ocean: Mehr als ein neues UI

Blue Ocean wurde Anfang April als zusätzliches optionales Plug-in in Version 1.0 veröffentlicht. Blue Ocean ist keine 1:1-Modernisierung der bestehenden Jenkins-Oberflächen. Stattdessen führt es ein neues pipelinezentriertes Bedienkonzept und entsprechende neue Dialoge ein. Das neue User Interface bietet aktuell ein personalisierbares Dashboard zur Darstellung aller Pipelines, eine Übersicht über die Pipelineläufe (Pipeline Activity View) sowie eine Detailansicht zu jedem einzelnen Durchlauf. Des Weiteren führt das Plug-in einen neuen Pipelineeditor ein. Dieser bietet eine grafische Oberfläche für das Erstellen von Pipelines in der Declarative Syntax. **Abbildung 2** zeigt eine Beispielpipeline in der neuen Oberfläche.

Das Plug-in ist in der ersten Version stark auf die Integration mit GitHub optimiert. Funktionen wie der Pipelineeditor stehen daher nur in der Verbindung mit einem GitHub-Repository zur Verfügung. Auch deckt die Oberfläche noch längst nicht alle Funktionen der klassischen Ansicht ab. Zur Jenkins-Verwaltung oder der Jobkonfiguration wird auf die bestehenden Dialoge umgeleitet.

Es wird noch eine Zeit dauern, bis sich Blue Ocean als echte Alternative zur klassischen Ansicht etabliert hat. Version 1.0 bietet Usern aber jetzt schon die Möglichkeit, Blue Ocean zu testen, da es parallel zum bestehenden User Interface funktioniert. Das Plug-in lässt sich daher ohne Einschränkungen auf bestehenden Systemen aktivieren. Der Benutzer kann jederzeit zwischen der neuen Ansicht und der klassischen Oberfläche umschalten. Dadurch können Nutzer sich bereits ein Bild vom neuen User Interface machen, ohne bekannte Funktionen und Nutzungspatterns aufgeben zu müssen.

Aus der Praxis

Bei TimoCom haben wir in einem Migrationsprojekt mehrere hundert klassische Freestyle-Jobs auf Pipeline as Code umgestellt. Mit dem Ziel, möglichst konsequent auf die Pipelinesyntax umzusteigen, wurde dazu eine Jenkins-2-Installation neben dem bestehenden Jenkins-System neu aufgesetzt. Ein Parallelbetrieb bietet den Vorteil, dass im Vorfeld der Umstellung die Konfiguration hinsichtlich der verwendeten Plug-ins konsolidiert werden kann. Außerdem wird eine sukzessive und störungsarme Umstellung für die einzelnen Entwicklungsteams möglich.

Für Entwicklungs- und DevOps-Teams ergeben sich beim Aufbau einer Delivery-Pipeline viele neue Möglichkeiten.

Im Umgang mit der Groovy DSL waren bei den Vorbereitungen zusätzlich zur Jenkins-Dokumentation folgende Quellen besonders hilfreich: [9], [10], [11]. Hier finden sich allgemeine Best Practices sowie wichtige Hinweise zur Entwicklung performanter, ressourcensparender Pipelines. Es werden beispielsweise die verschiedenen Möglichkeiten beim Einsatz von *parallel* erläutert und die Empfehlung zur Absicherung von Netzwerkaufrufen und Userinteraktionen mit entsprechenden Time-outs gegeben.

Im Vorfeld wurde außerdem ein Set von kleinen Convenient-Funktionen in einer unternehmensinternen Global Shared Library erarbeitet. Die Umstellung in den Teams erfolgte dann schrittweise. Dadurch konnten wir mit den Erfahrungen des jeweiligen Teams unsere Shared-Library-Funktionen entsprechend anpassen und erweitern. Kollegen mit Java-Erfahrung kamen schnell mit der Pipelinesyntax zurecht. Die Vorbereitung der Shared Library sowie das iterative Vorgehen waren zusätzliche Erfolgsfaktoren bei der Umstellung, da die bereitgestellten Funktionen die Einarbeitung weiter erleichterten. Auch teilweise vorhandene Skepsis in Bezug auf die Entwicklung mit Groovy hat sich schnell aufgelöst.

Da zu Beginn unserer Umstellung die Declarative Syntax noch im Betastadium war, sind die meisten unserer Pipelines mit der Groovy DSL entwickelt. Bei einigen neueren Pipelines wird auch die deklarative Syntax verwendet. Unsere Erfahrungen mit beiden Stilen im täglichen Einsatz sind sehr gut. Es gibt zwar, wahrscheinlich auch auf aufgrund des großen Umfangs der Änderungen, im Pipelineumfeld noch kleinere Bugs und Probleme. Hier kann man aber von der großen Community und vielen hilfreichen Einträgen in bekannten Entwicklerforen profitieren, sodass meist schnell eine Lösung gefunden wird. Einzig das UI trübt aktuell das Bild, da die Stage View verschiedene Limitierungen hat, z. B. bei der Darstellung der parallelen Abarbeitung und der zugehörigen Logs. Gleichzeitig brachte Blue Ocean zum Zeitpunkt unserer Evaluierung noch nicht die notwendige Reife mit. Es bleibt somit spannend, die weitere Entwicklung des neuen User Interface zu verfolgen.

Fazit

Der Umstieg auf die Pipelineentwicklung hat sich bei unseren Projekten bewährt, und der positive Eindruck überwiegt deutlich. Die Pipelines direkt mit dem Anwendungscode versionieren zu können, ist hinsichtlich Lokalität und Änderungshistorie ein großer Mehrwert. Sollten durch Änderungen in den Anwendungssourcen bspw. Anpassungen in der Integrationspipeline notwendig sein, wird auch das Jenkinsfile im entsprechenden Feature Branch angepasst. Code- und Pipelineänderungen gehen somit Hand in Hand.

Überzeugt hat uns auch die Möglichkeit, gemeinsame Pipelinefunktionen und -schritte als Shared Library global zur Verfügung zu stellen. Durch die Wiederverwendung kann teamübergreifend von Verbesserungen profitiert werden. Gleichzeitig lassen sich Integrations- und Deployment-Prozesse, wo gewünscht, vereinheitlichen und standardisieren.

Mit den angebotenen Basisfunktionen und durch das große Plug-in-Ökosystem wird es einfacher, Integration und Deployment zu automatisieren. Für Entwicklungs- und DevOps-Teams ergeben sich dadurch beim Aufbau einer Delivery-Pipeline viele neue Möglichkeiten. Mit Pipeline as Code hat es Jenkins somit geschafft, sich von seinem Ursprung als Continuous-Integration-Server zu einem vollwertigen Continuous-Delivery-Werkzeug zu wandeln.

Hendrik Brinkmann arbeitet als Werkstudent bei der TimoCom Soft- und Hardware GmbH. Er studiert Medieninformatik an der Hochschule Düsseldorf mit den Schwerpunkten Web- und Anwendungsentwicklung. Aktuell beschäftigt er sich zudem mit DevOps-Themen wie Docker, Continuous Integration und Continuous Delivery.

Philip Stroh arbeitet als Softwarearchitekt bei der TimoCom Soft- und Hardware GmbH. Er hat langjährige Erfahrung in der Entwicklung und Konzeption von Java-basierten Webanwendungen. Zurzeit sind seine weiteren Schwerpunkte NoSQL-Datenbanken und Delivery-Automatisierung.

Links & Literatur

[1] Kawaguchi, Kohsuke: „Jenkins 2.0 is here!“: https://jenkins.io/blog/2016/04/26/jenkins-20-is-here/

[2] Farcic, Viktor: „The DevOps 2.0 Toolkit“, Leanpub

[3] Script Security Plug-in: https://wiki.jenkins-ci.org/display/JENKINS/Script+Security+Plugin

[4] Getting Started – Create your first Pipeline: https://jenkins.io/doc/pipeline/tour/hello-world/

[5] Pipeline Steps Reference: https://jenkins.io/doc/pipeline/steps/

[6] Extending with Shared Libraries: https://jenkins.io/doc/book/pipeline/shared-libraries/

[7] Pipeline Development Tools: https://jenkins.io/doc/book/pipeline/development/

[8] Pipeline Syntax: https://jenkins.io/doc/book/pipeline/syntax

[9] „Top 10 Best Practices for Jenkins Pipeline Plugin“: https://www.cloudbees.com/blog/top-10-best-practices-jenkins-pipeline-plugin

[10] Best Practices for Scalable Pipeline Code: https://jenkins.io/blog/2017/02/01/pipeline-scalability-best-practice/

[11] Van Oort, Sam: „The Need For Speed: Building Pipelines To Be Faster“: https://www.cloudbees.com/need-speed-building-Pipelines-be-faster