Jenkins, Jenkins: Don't Repeat Yourself (DRY)!

Lukas Pradel, Conciso GmbH

In modernen Microservice-Architekturen wird Software vollautomatisch über CI/CD-Pipelines ausgeliefert. Wie lassen sich diese Pipelines bei wachsender Komplexität und steigender Anzahl von Services beherrschen, zumal sie ja mitunter das Deployment in die Produktion steuern? Man tut gut daran, sich einen alten Bekannten unter den Prinzipien der Softwareentwicklung in Erinnerung zu rufen.

Microservices sind nach wie vor in aller Munde. Oftmals werden sie über eine eigene Continuous Integration Pipeline gebaut. Im Falle von Continuous Deployment werden sie auch automatisch ausgeliefert und deployt. Ein wichtiger Bestandteil von Microservice-Architekturen ist die Unabhängigkeit der Services. Auch wenn die Services in der Praxis immer in Kombination eingesetzt werden, soll jeder Microservice eine abgeschlossene Einheit sein, die vollkommen unabhängig von allen anderen gebaut, deployt und betrieben werden kann. Daher wird jeder Microservice mit einer eigenen, separaten Datenbankinstanz oder Message-Broker-Instanz versehen. So ist sichergestellt, dass bei einem Ausfall einer Datenbank- oder Message-Broker-Instanz nicht automatisch alle Systemkomponenten ausfallen. Aus demselben Grund verfügt jeder Microservice auch über eine eigene separate CI/CD-Pipeline. Die separate Pipeline ist auch deshalb wichtig, weil sich Pipelines von Service zu Service unterscheiden können, wenn die Servicelandschaft insgesamt eher heterogen ist.

Stages sind Quality Gates

CI/CD-Pipelines bestehen aus mehreren Schritten (sogenannten „stages"), die sequenziell oder parallel durchlaufen werden können *(siehe Abbildung 1)*. Dabei ist jeder Schritt ein „Quality Gate". Wenn beispielsweise ein Kompilieren des Codes nicht möglich ist oder ein Integrationstest scheitert, wird die Software nicht ausgeliefert. Ein Software-Artefakt, das alle Pipeline-Stufen durchlaufen hat, kann potenziell in Produktion ausgeliefert werden. Pipelines können sowohl grafisch-konfigurativ als auch deklarativ über Code definiert werden. So kommt zum Beispiel bei GitLab CI die Programmiersprache Ruby zum Einsatz, während Jenkins-Pipelines mit Groovy formuliert werden.

Komplexität vs. Duplizierung

Bei komplexen Softwaresystemen mit konsequentem Domänenschnitt kann die Anzahl der Microservices schnell zweistellig werden. Damit einhergehend wird oft auch die Anzahl der Pipeline-Schritte groß. Je nachdem, wie homogen die Microservices sind, schleichen sich dann schnell Redundanzen und Code-Duplizierungen ein, die man im Anwendungscode (vollkommen zu Recht) peinlichst zu vermeiden versucht. Im Großen und Ganzen funktionieren die Pipelines in allen Services ähnlich: Code auschecken, kompilieren, testen, ausliefern und deployen. Daher wird häufig der Code eines Microservice für einen neuen Microservice übernommen und – wo nötig – angepasst.

Deklarative Jenkins-Pipelines

Ein Beispiel für eine simple deklarative Jenkins-Pipeline zeigt das `Jenkinsfile` in *Listing 1*. Diese Pipeline ist denkbar einfach und besteht aus lediglich zwei Stages: in der ersten wird der Code aus dem Source Control Management System (in diesem Fall Git) ausgecheckt und in der zweiten Stage die Software mit Maven gebaut.

Stellen wir uns vor, dass diese Pipeline nicht in einem Microservice, sondern in zehn Microservices zum Einsatz kommt. In diesem Szenario bereitet es den Engineers genau wie bei Code-Duplizierungen im Anwendungscode Kopfzerbrechen, wenn nun beispielsweise eine querschneidende neue Anforderung eine Änderung am Build-Prozess erforderlich macht. Beispiele für solche Änderungen finden sich schnell und sehen erst einmal sehr unschuldig aus:

- Vor einem neuen Release des Microservice soll die Softwareversion über den Befehl `sh "mvn -e -DnewVersion=${version} versions:set"` gesetzt werden
- In manchen Services soll eine andere Maven-Settings-Datei verwendet werden
- In den Maven-Build-Schritt muss eine neue Umgebungsvariable aufgenommen werden

In all diesen Fällen bleibt einem nichts anderes übrig, als in allen Microservices das Jenkinsfile zu öffnen, nach der entsprechenden Stage und der jeweiligen Stelle zu suchen und dort die gewünschten Änderungen einzupflegen. Eine mühsame Arbeit, die man sich gerne ersparen würde. Die Ursache des Problems besteht darin, dass wir mit dem Code unserer CI/CD-Pipeline gegen ein bewährtes Prinzip der Softwareentwicklung verstoßen haben.

Das DRY-Prinzip

Vor ziemlich genau 20 Jahren haben Andy Hunt und Dave Thomas in ihrem legendären Buch „The Pragmatic Programmer" unter der Überschrift „The Evils of Duplication" das bekannte DRY-Prinzip („Don't repeat yourself" – „Wiederhole dich nicht") formuliert, das besagt, dass **jedes Wissensfragment genau eine eindeutige, unmissverständliche und maßgebliche Repräsentation in einem System haben muss** *[1]*. Die Autoren schlagen vor, das Prinzip in allen Bereichen der Softwareentwicklung anzuwenden: bei Datenbankschemata, Testplänen, Dokumentation und eben auch beim Build-System. Es gibt in der Softwareentwicklung kaum ein Prinzip, das so unumstritten und verbreitet ist wie das DRY-Prinzip. Aus gutem Grund, denn die Folgen bei Verstößen sind äußerst schmerzhaft und kostspielig.

Sobald Informationen redundant vorgehalten werden, werden Anpassungen aufwendig, da diese immer an mehreren Stellen vorgenommen werden müssen. Darüber hinaus wird das Softwaresystem

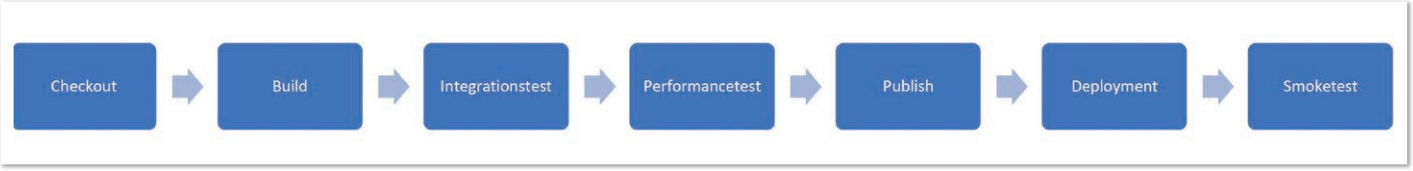

Abbildung 1: Eine einfache CI/CD-Pipeline (© Lukas Pradel)

```
#!/usr/bin/env groovy

pipeline {
  agent any

  stages {
    stage('Checkout') {
      steps {
        git branch: pipelineParams.branch, credentialsId: 'GitCredentials', url: pipelineParams.scmUrl
      }
    }

    stage('Build') {
      steps {
        withMaven(mavenSettingsFilePath: '/var/jenkins_home/settings.xml') {
          sh "mvn -e -ff clean install"
        }
      }
    }
  }
}
```

Listing 1: Ein Jenkinsfile einer einfachen Pipeline

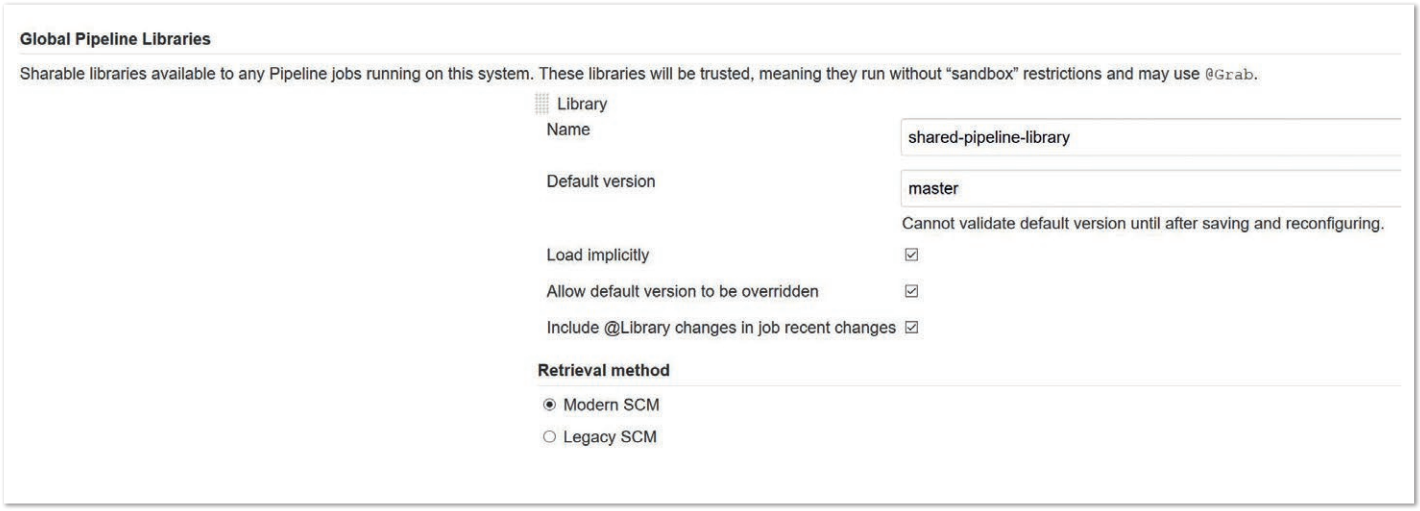

Abbildung 2: Die Jenkins-Konfiguration für eine Shared Pipeline Library (Quelle: Lukas Pradel)

```
#!/usr/bin/env groovy

@Library('shared-pipeline-library@2.1.5')

pipeline {
  agent any

  stages {
    stage('Checkout') {
      steps {
        git branch: pipelineParams.branch, credentialsId: 'GitCredentials', url: pipelineParams.scmUrl
      }
    }

    stage('Build') {
      steps {
        buildService(service: 'FooService', version: '1.2.7')
      }
    }
  }
}
```

Listing 2: Die vereinfachte, DRY-konforme Jenkins-Pipeline.

anfällig für Bugs, weil mit wachsender Anzahl von Redundanzen auch die Wahrscheinlichkeit steigt, dass Stellen übersehen oder fehlerhafte Anpassungen vorgenommen werden. Häufig finden sich bei Code-Duplikaten auch geringfügige Abwandlungen, sodass es auch in modernen Softwareentwicklungsumgebungen immer schwieriger wird, alle Stellen zu identifizieren.

Umgekehrt sind Softwaresysteme, bei denen das Prinzip erfolgreich angewendet wird, leicht zu warten, denn eine Änderung an einer einzelnen Komponente des Systems erfordert keine zusätzlichen Anpassungen an anderen Komponenten.

Shared Pipeline Groovy Libraries

Bezogen auf unsere Pipeline würde sich der geübte Engineer wünschen, für jede der einzelnen Stages eine parametrisierbare Funktion oder Methode extrahieren und diese dann einfach in jeder Pipeline mit den jeweils passenden Argumenten aufrufen zu können.

An genau dieser Stelle schafft das „Pipeline Shared Groovy Libraries Plug-in" für Jenkins Abhilfe *[2,3]*. Den Jenkins-Maintainern scheint der Wert des Plug-ins bewusst zu sein, da es mittlerweile fester Bestandteil des Pipeline-Plug-ins und somit in neueren Jenkins-Installationen standardmäßig installiert ist. Mit dem Plug-in ist es möglich, Pipeline-Code versioniert in eine Bibliothek auszulagern, die man dann in den Pipelines einzelner Microservices einbinden und verwenden kann.

Dazu muss man lediglich im SCM die Bibliothek ablegen. Dann kann man diese in der globalen Jenkins-Konfiguration unter Angabe der SCM-Koordinaten referenzieren. *Abbildung 2* zeigt die entsprechende Konfiguration in Jenkins. In den Pipeline-Skripten der einzelnen Microservices kann die Bibliothek nun sehr einfach verwendet werden. Unsere ursprüngliche Pipeline aus *Listing 1* wird deutlich vereinfacht, wie man *Listing 2* entnehmen kann.

Die Bibliothek wird über die Zeile `@Library('shared-pipeline-library@2.1.5')` in die Version 2.1.5 eingebunden. In unserer Beispielversion ist die Bibliothek in Git versioniert. Das Jenkins-Plug-in klont die Bibliothek mit dem Tag "2.1.5". So ist es uns auch möglich, „breaking changes" an der Bibliothek vorzunehmen, ohne die Pipelines der einzelnen Services alle sofort umstellen zu müssen. Diese können so lange eine beliebige Version der Bibliothek

```
import de.conciso.ServiceMavenSettings

def call(args) {

  assert args.service: "Service name must be provided."
  assert args.version: "Service version must be provided."

  def serviceMavenSettings = new ServiceMavenSettings()
  def mavenSettings = serviceMavenSettings.getMavenSettingsForService(args.service)

  withMaven(mavenSettingsFilePath: mavenSettings) {
    sh "mvn -e -DnewVersion=${args.version} versions:set"
    sh "mvn -e -ff clean install -Dbuild.environment=ci"
  }
}
```

Listing 3: Die extrahierte Build-Stage in der Datei buildService.groovy

verwenden, bis die Pipeline-Skripte entsprechend für die neue Version angepasst wurden – und sie können unabhängig weiterentwickelt werden.

Die gesamte Build-Stage ist nun im Aufruf der Funktion `buildService` verschwunden, wie man es aus der Java-Entwicklung gewohnt ist. Wir wollen uns nun natürlich noch ansehen, wie die eigentliche Bibliothek im Detail aussieht.

Shared Pipeline Library

Die Struktur der Bibliothek ist in *Abbildung 3* dargestellt. Im Ordner `src` liegen Groovy-Klassen, im Ordner `test` die entsprechenden Unit-Tests. Funktionen, die im Pipeline-Skript aufgerufen werden können, liegen unter exakt dem Namen der Funktion als Groovy-Datei im Ordner `vars`. Dementsprechend gibt es dort eine Datei `buildService.groovy`. In dieser findet sich nun der extrahierte und parametrisierte Code der Build-Stage, wie man *Listing 3* entnehmen kann.

Funktionen im Ordner `vars` beginnen immer mit `def call(args)` und können über das `args`-Feld auf ihre Aufrufargumente zugreifen. In diesem Fall haben wir im Vergleich zur ursprünglichen Pipeline in *Listing 1* somit den Namen und die Version des Service parametrisiert und das Setzen der Service-Version sowie eine Maven-Variable ergänzt. Das Ermitteln der korrekten Maven-Settings-Datei haben wir in eine separate Groovy-Klasse mit dem Namen `ServiceMavenSettings` ausgelagert, die über ein einfaches `import`-Statement verwendet wird.

In allen Microservices wird nunmehr lediglich die Funktion `buildService` aufgerufen. Wenn nun die Pipeline-Bibliothek weiterentwickelt wird, muss in den Pipelines der einzelnen Microservices im Idealfall nur noch die Versionsnummer angepasst werden. Alle übrigen Änderungen sind in der Bibliothek weggekapselt.

Objektorientierte Bibliotheken

Abgesehen von den Groovy-Skript-Funktionen kann man, wie bereits erwähnt, auch objektorientiert in der Bibliothek programmieren. Dazu muss man nur mit entsprechenden Groovy-Klassen im Ordner `src` arbeiten. Diese können dann wiederum in den Skripten im Verzeichnis `vars` über ein einfaches `import`-Statement verwendet werden, wie unser Beispiel zeigt. Bezogen auf unsere Pipeline könnte die Klasse `ServiceMavenSettings` dann so aussehen wie in *Listing 4*.

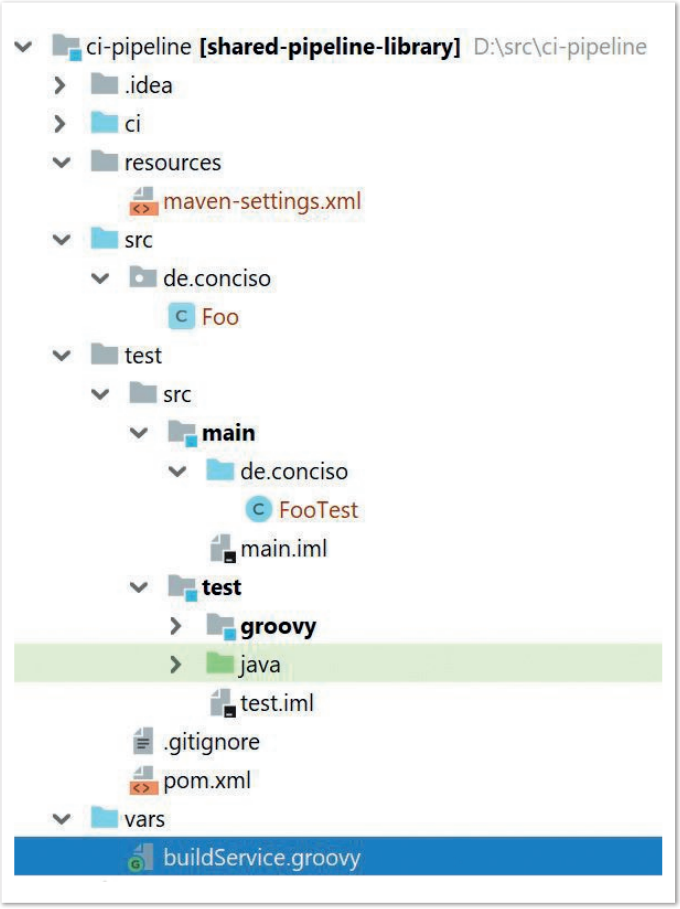

Abbildung 3: Die Struktur einer Shared Pipeline Library (Quelle: Lukas Pradel)

Wir haben hier natürlich ein absolutes Minimalbeispiel betrachtet. Bei realistischen Pipelines mit 20-30 Stages für 15 oder mehr Microservices besteht der erste Schritt in Richtung DRY-Konformität daher darin, für jede Stage, die in einer der Pipelines vorkommt, eine entsprechende Funktion in der Pipeline Library bereitzustellen und diese in den Pipelines zu referenzieren.

Das Auslagern von Pipeline-Schritten in die Bibliothek kann sogar so weit getrieben werden, dass ein komplettes Pipeline-Skript in der Bibliothek abgelegt werden kann (ebenfalls im Verzeichnis `vars`). Somit wandert dann ein komplettes parametrisiertes Jenkinsfile in eine entsprechend benannte Groovy-Datei. Das eigentliche Jenkinsfile sieht dann nur noch wie in *Listing 5* aus.

```
package de.conciso

class ServiceMavenSettings implements Serializable {

  ServiceMavenSettings() {
  }

  getMavenSettingsForService(String serviceName) {
    if (serviceName == 'FooService') {
      return '/var/jenkins_home/settings.xml'
    } else {
      return '/var/jenkins_home/other_settings.xml'
    }
  }
}
```

Listing 4: Eine Groovy-Klasse für Pipeline Libraries.

```
#!/usr/bin/env groovy

@Library('shared-pipeline-library@3.0.0')
servicePipeline(service: 'foo', ..)
```

Listing 5: Die extreme Variante: Die Pipeline verschwindet komplett in der Bibliothek

In diesem Beispiel liegt der gesamte Pipeline-Code in der Bibliothek in der Datei `vars/servicePipeline.groovy`. Diese Variante sollte man allerdings nur nach sorgfältiger Überlegung und mit Vorsicht einsetzen. Sie bietet sich nur an, wenn die Microservices vollkommen homogen sind (identische Programmiersprache, identische Frameworks, identische Prozesse, identische Organisationen etc.). Anderenfalls führt dieses Vorgehen dazu, dass die externalisierte Pipeline so sehr parametrisiert werden muss, dass sie selbst unwartbar wird.

Fazit

Wir beobachten in unseren Projekten immer wieder, dass Pipeline-Code stiefmütterlich behandelt wird, da er kein Teil des Anwendungscodes ist und sich gerade Engineers im Bereich Java-Backend oftmals nicht für Infrastruktur-Themen begeistern.

Diese Vernachlässigung ist jedoch überaus gefährlich: Je mehr Continuous Delivery und DevOps Einzug halten, desto wichtiger werden CI/CD-Pipelines im Softwareentwicklungsprozess. Bei gelebtem Continuous Delivery umfassen sie das Deployment in die Produktion und manuelle Schritte entfallen teilweise komplett – der gesamte Prozess ist vollkommen automatisiert.

Wir plädieren daher dafür, dem Pipeline-Code mit denselben Qualitätsansprüchen zu begegnen, die man auch an den Anwendungscode hat. Dabei ist das DRY-Prinzip nur eines der wichtigen Prinzipien der Softwareentwicklung. So kann auch Pipeline-Code automatisiert getestet und sogar testgetrieben entwickelt werden, auch wenn wir es in diesem Beitrag nur andeuten.

Quellen

[1] Andrew Hunt, David Thomas (1999): The Pragmatic Programmer: From Journeyman to Master. Addison-Wesley, USA.

[2] *https://wiki.jenkins.io/display/JENKINS/Pipeline+Shared+Groovy+Libraries+Plugin*

[3] *https://jenkins.io/doc/book/pipeline/shared-libraries/*

Lukas Pradel

lukas.pradel@conciso.de

Lukas Pradel arbeitet als Senior Consultant bei der Conciso GmbH. Die Zeit, die er mit der Beachtung des DRY-Prinzips einspart, nutzt Lukas zum Motorradfahren.