Ausgabe 06/2018

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 06

Java aktuell

iJUG Verbund
www.ijug.eu

Tipps von Experten
Der richtige Umgang mit Zeitzonen

Im Interview
Mike Milinkovich, Executive Director Eclipse Foundation

Praktische Erfahrungen
Security innerhalb der agilen Software-Entwicklung

Die Reise nach Jakarta

Data Analytics 2019

Die Datenexplosion meistern

26. & 27. März | Phantasialand bei Köln

analytics.doag.org

ORACLE Cloud

DOAG

Containerisierte CI/CD-Pipelines mit OpenShift

Tobias Schneck, Loodse

Container erfreuen sich unter Entwicklern immer größerer Beliebtheit. Kein Wunder, denn sie federn Umgebungs-Unterschiede ab, verbessern die Vorhersehbarkeit und sind skalierbar. Dadurch vereinfachen und beschleunigen sie die Entwicklung und das Deployment neuer Features und verringern zusätzlich die Kosten für komplexe Umgebungen.

Um die Vorteile voll auszuschöpfen, braucht es spätestens im zweiten Schritt auch eine stabile und skalierbare Continuous-Integration/Continuous-Delivery-Umgebung (CI/CD). Diese sind seit jeher schwer aufzusetzen und zu pflegen; beim Aufbau einer containerisierten Build-Pipeline kommen zusätzlich neue Herausforderungen, aber auch Chancen hinzu. Dieser Artikel stellt einen der bislang gängigsten Ansätze bei der Nutzung von Jenkins-CI-Servern in Kubernetes-Clustern vor, die OpenShift-Build-Pipelines.

Für den Aufbau von flexibel skalierbaren CI/CD-Pipelines bietet sowohl Kubernetes mit dem Jenkins-Kubernetes-Plug-in als auch OpenShift mit den integrierten Jenkins-Build-Pipelines eine echte Open-Source-Option für einen containerisierten CI/CD-Server an. OpenShift ist eine Open-Source-Platform-as-a-Service-Lösung (PaaS) von Red Hat auf Kubernetes-Basis, die es Entwicklern ermöglicht, ihre Anwendungen zu bauen, zu testen und sie in private oder öffentliche Clouds zu deployen. Sowohl bei Plain Kubernetes als auch bei OpenShift kann der CI/CD-Server innerhalb oder außerhalb des Clusters betrieben werden.

Dem modernen „Infrastructure-as-a-Code"-Ansatz folgend, fokussiert sich dieser Artikel auf den Betrieb im Cluster selbst. Die größte Herausforderung dabei ist es, ein lauffähiges Container-Image dynamisch zu erstellen, zu testen und zu releasen, ohne die eigene Cluster-Infrastruktur zu verlassen. Dies ermöglicht einen durchgängig gelebten DevOps-Ansatz.

OpenShift integriert zwei Arten, wie Images gebaut werden können: native Docker-Builds mit Dockerfiles und sogenannte „Source-2-Image-Builds" (S2I, siehe *„https://github.com/openshift/source-to-image").* Zur Steuerung der beiden Build-Typen wurde in OpenShift ein auf Kubernetes-Anforderungen konfigurierter Jenkins-CI-Server als „Out-of-the-box"-Komponente hinzugefügt. Die durch das OpenShift-Client-Plug-in erweiterte Jenkins DSL unterstützt neben Java auch „Node.js"-Builds, die zur Laufzeit in spontan gestarteten Pods durchgeführt werden.

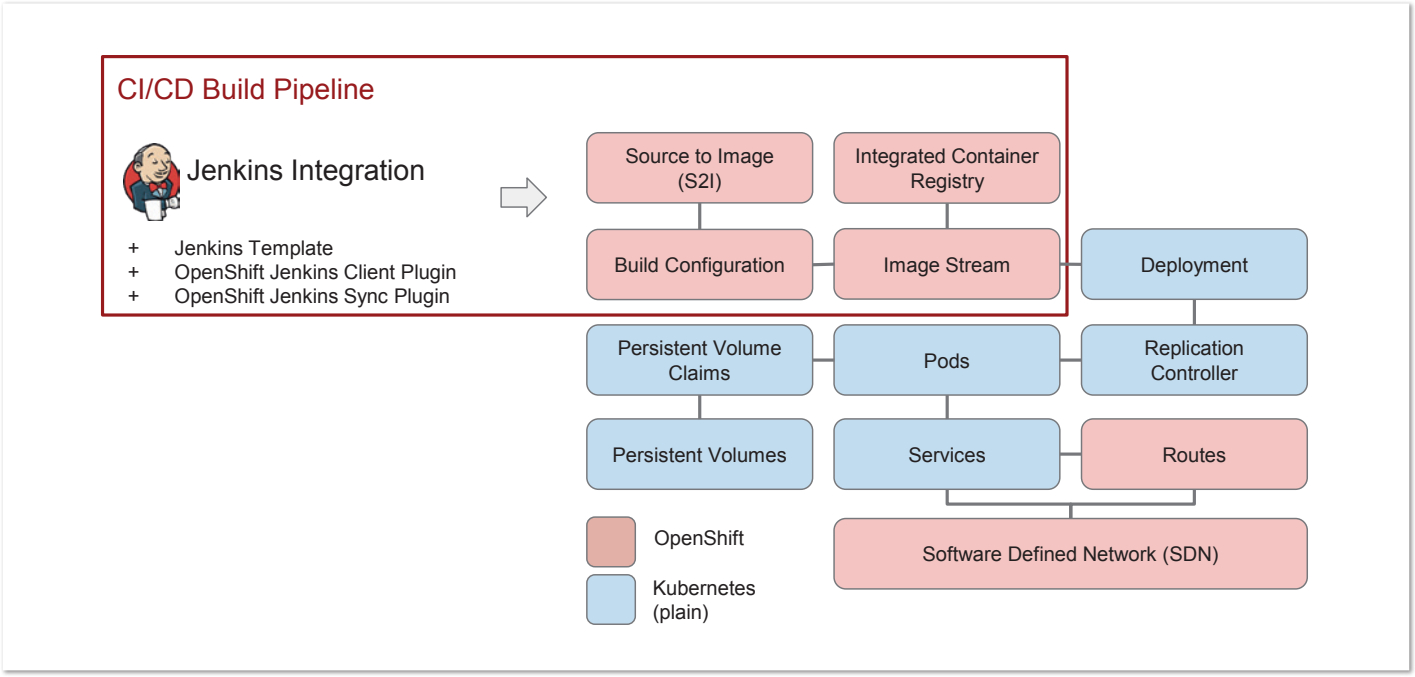


Abbildung 1: Die OpenShift-Komponenten

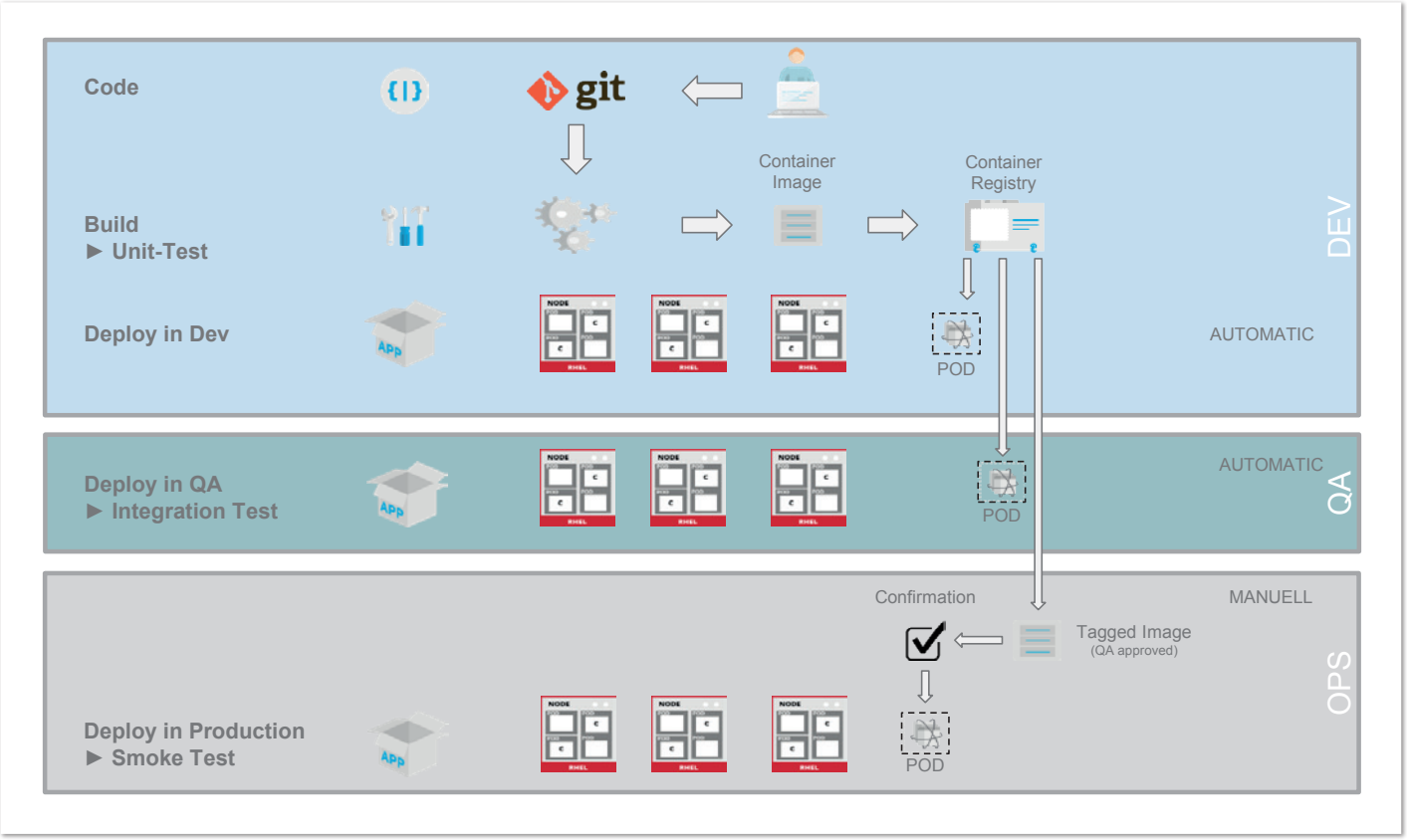


Abbildung 2: Build Workflow und Deployment Stages

Dieser Mechanismus ist erweiterbar, sodass bei Bedarf Unterstützung für weitere Sprachen hinzugefügt werden kann. Dadurch, dass Kubernetes-Pods als Ad-hoc-Build-Ressource genutzt werden, liegen bei hoher Nachfrage die Build-Limits allein bei den verfügbaren Cluster-Ressourcen. Die Jenkins-Slave-Nodes selbst skalieren automatisch, womit das manuelle Ressourcen-Management entfällt.

Die OpenShift-Komponenten

OpenShift ergänzt Kubernetes um einige hilfreiche Komponenten, die sowohl das Deployment selbst als auch den Aufbau einer Build-Pipeline vereinfachen:

- Routes, eine dynamische DNS-Komponente, die einfache Aufrufe der Applikation über eine fest definierbare URL erlaubt.

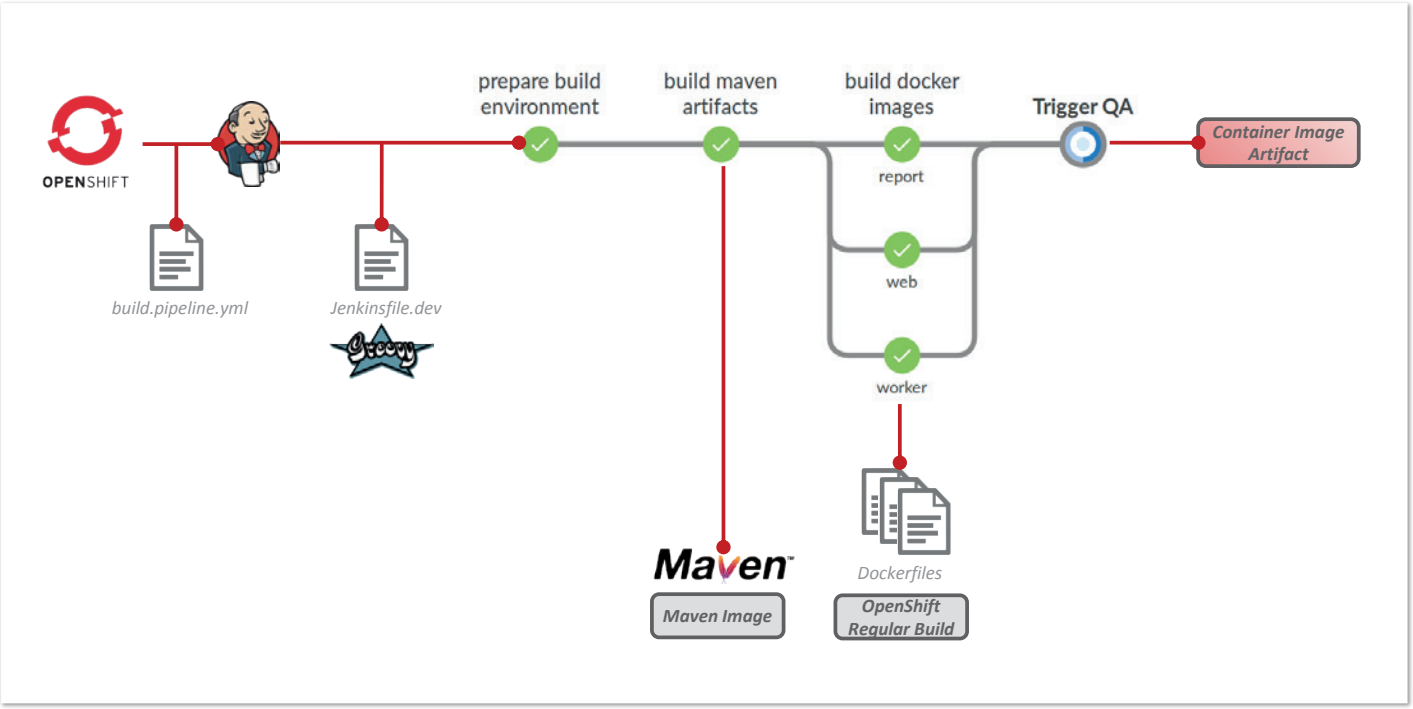


Abbildung 3: Jenkins-Build-Pipeline – DEV-Stage

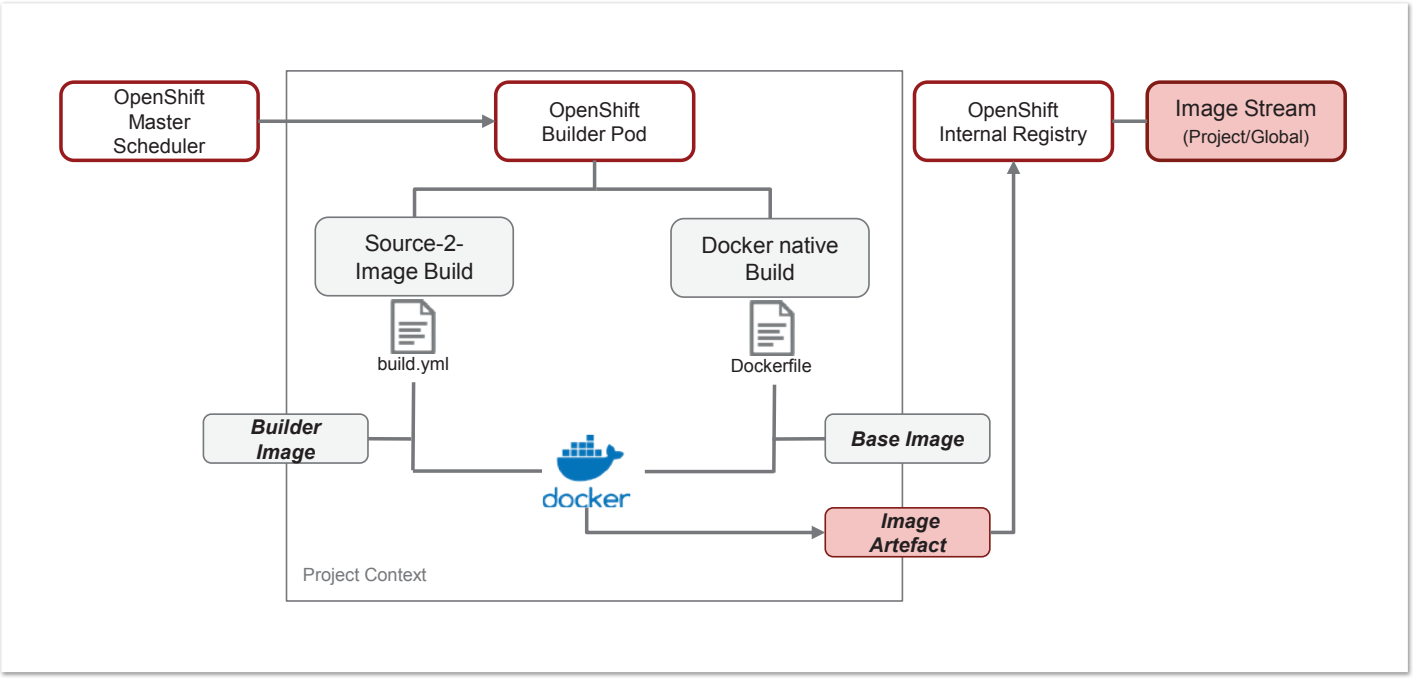


Abbildung 4: OpenShift-Regular-Builds (Source- und Docker-Strategie)

- Software Defined Network (SDN) als Netzwerk-Schicht zwischen OpenShift-Clustern und der Außenwelt.
- Image Stream als interne Objekt-Definition eines Docker-Image.
- Integrierte Container-Registry zum Speichern und Verteilen gebauter Container-Images.
- Build Configuration, die festlegt, welcher Build-Typ (Docker native, S2I, Jenkins-Pipeline) durchgeführt wird. Es definiert zudem die benötigte Build-Umgebung (wie Source Repository, Secrets und Webhook).
- „Source to Image"-Objekt, das einen fixen, vorkonfigurierten Build-Ablauf definiert, der lediglich das Sourcecode-Repository als Übergabe-Parameter benötigt. Die S2I-Builder-Images können selbst erstellt werden oder es werden die mitgelieferten Images wie für Node.js genutzt *(siehe „https://docs.openshift.com/container-platform/latest/using_images/s2i_images/index.html")*.

Damit verfügt OpenShift selbst noch nicht über ausführende CI/CD-Komponenten; sie werden erst mit der Jenkins-Integration ermöglicht. Diese besteht aus den folgenden Komponenten:

- Das Jenkins-Template legt fest, mit welchen Parametern ein Jenkins im OpenShift-Cluster eingerichtet wird. Am Template können Cluster-Admins Anpassungen durchführen und beispielsweise CPU, Memory, Secrets, Umgebungsvariablen oder Persistent Storage konfigurieren.

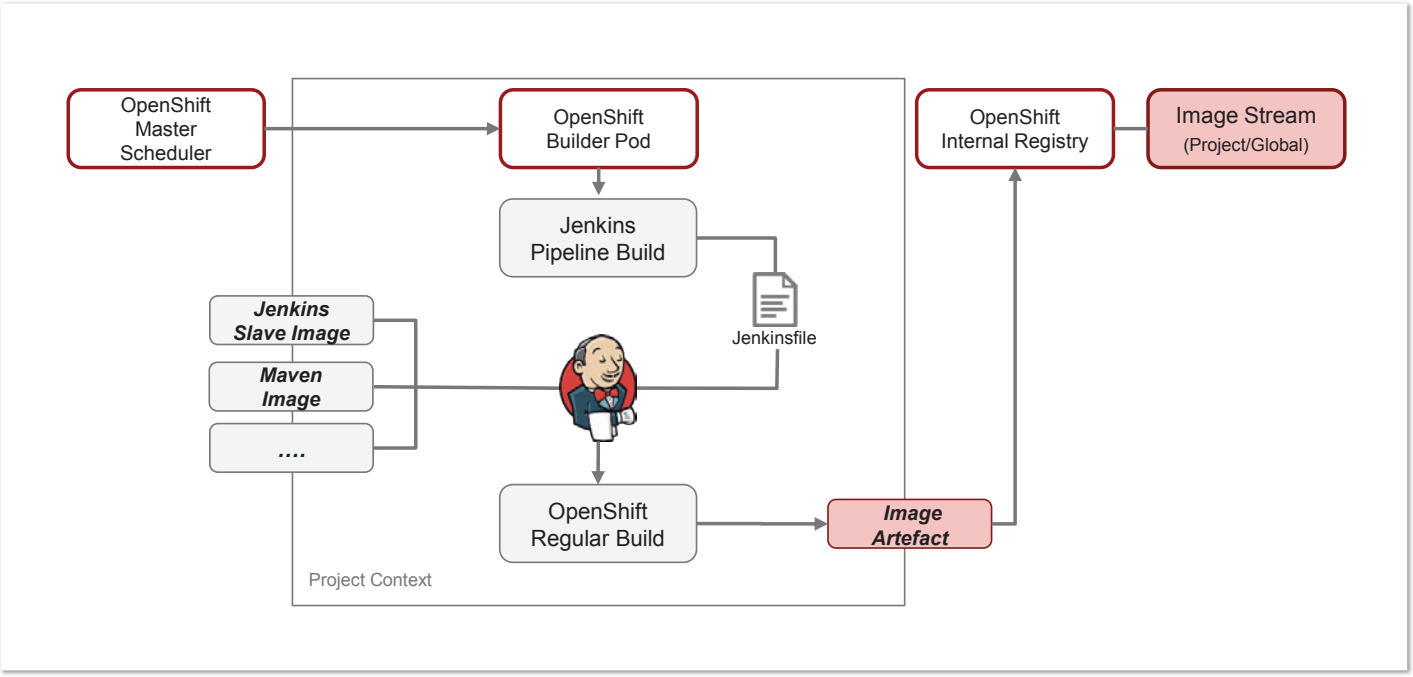


Abbildung 5: OpenShift-CI-Pipeline-Build (Jenkins-Pipeline)

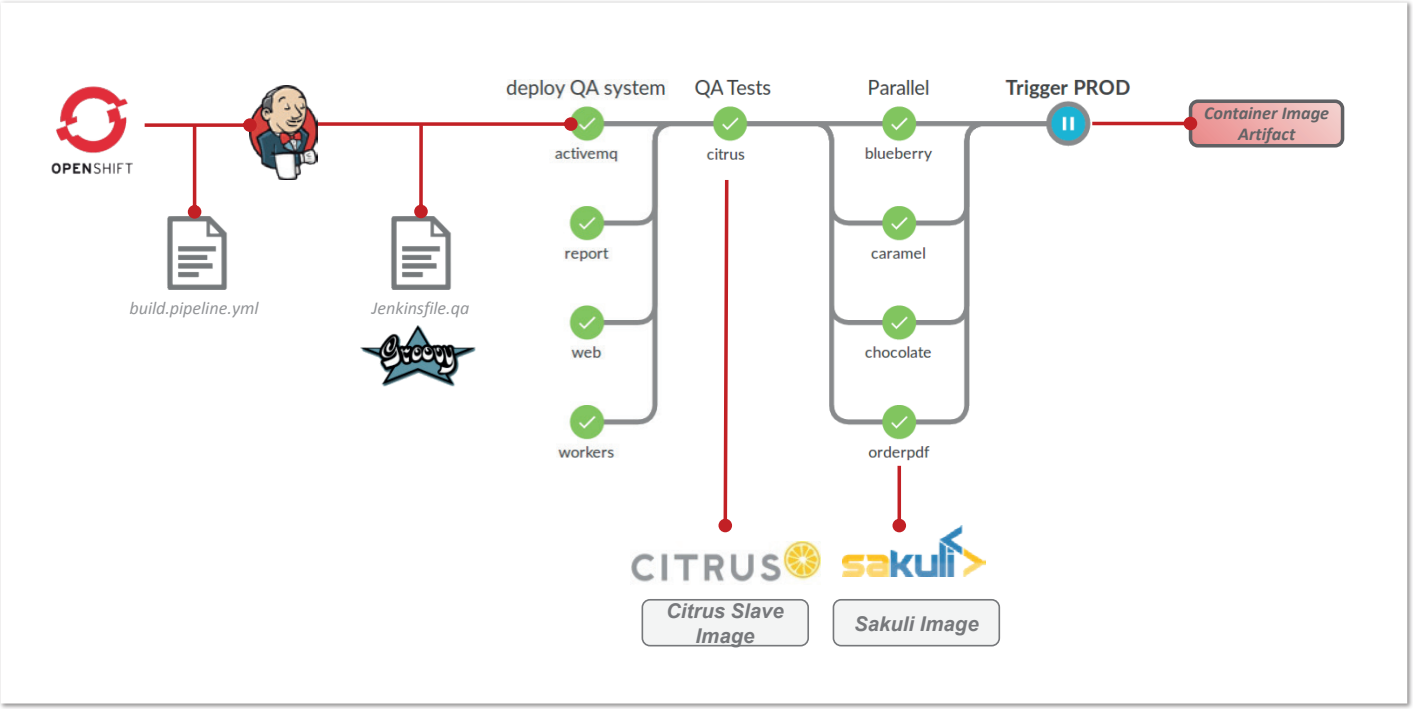


Abbildung 6: Jenkins-Build-Pipeline – QA Stage

- Das OpenShift-Jenkins-Client-Plug-in gewährleistet, dass innerhalb eines Jenkins-Builds direkt mit den OpenShift- und Kubernetes-Ressourcen kommuniziert werden kann, etwa um einen nativen OpenShift-Docker-Build anzustoßen oder ein Deployment auszulösen.
- Das OpenShift-Jenkins-Sync-Plug-in stellt sicher, dass das OpenShift-Dashboard und das entsprechende Build-Objekt über die Ergebnisse eines angestoßenen Jenkins-Jobs informiert werden.

Build Workflow und Deployment-Stages

Im OpenShift wird ein Build im Cluster als Objekt vom Typ „BuildConfig" gespeichert, im YAML-Format beschrieben und per Commandline oder UI im Cluster erzeugt. Das Objekt erwartet im Anschluss, um mit dem Build zu beginnen, ein Start-Event, das durch den sogenannten „Hook" oder einen CLI-Aufruf erzeugt wird. Der erzeugte Hook ist per Web-URL erreichbar und kann in Drittsystemen wie dem Versions-Verwaltungssystem (GitHub, GitLab oder ähnliche) hinterlegt sein. Dadurch wird bei jeder Änderung des Sourcecodes ein neuer Build angestoßen. Eine beispielhafte Konfiguration ist in diesem Projekt auf GitHub zu finden (siehe „toschneck/openshift-example-bakery-ci-pipeline-openshift.build.bakery.generic.yaml"). Ziel des automatisieren Build-Prozesses ist es, in drei Stages ein produktionsreifes Artefakt zu deployen *(siehe Abbildung 2)*:

- **Development Stage**
  Bau und Bereitstellung eines unveränderbaren Container-Artefakts
- **QA Stage**
  Automatisiertes Testen in einer produktionsnahen Umgebung durch geeignete Integration-Tests
- **Production Stage**
  Deployment auf das Produktivsystem und Smoke-Testing

Development Stage

Zum Bau eines unveränderbaren Artefakts wird zuerst der zugehörige Sourcecode ausgecheckt. Dann wird dieser mit einem Build-Tool wie Maven bei Java-Projekten in ein lauffähiges Applikations-Artefakt umgewandelt. Zu diesem Zeitpunkt sollten auch Unit-Tests durchgeführt werden, um auf Funktionsebene sicherzustellen, dass der Code keine Fehler enthält. Schlagen die Tests fehl, wird kein Artefakt erstellt und der Build abgebrochen. Das ist wichtig, um dem Entwickler möglichst schnell Feedback zu geben.

Ist der Test erfolgreich, wird die Applikation in ein Container-Image gepackt und mit einer eindeutigen ID in eine zentrale Container-Registry gepusht, um für die weiteren Stages zur Verfügung zu stehen. *Abbildung 3* zeigt, wie das in unserer Bakery-Beispiel-Applikation erfolgt.

Die „build.pipeline.yml"-Datei beschreibt, dass ein Build vom Typ „JenkinsPipeline" durchgeführt werden soll, der im Jenkinsfile „Jenkinsfile.dev" genauer beschrieben ist. Die zugehörigen Dateien sind auf GitHub unter „toschneck/openshift-example-bakery-ci-pipeline/openshift" einsehbar. Im „Jenkinsfile.dev" ist definiert, dass ein Node vom Typ „maven" das angegebene Java-Projekt kompiliert, testet und ein JAR-Artefakt erzeugt.

Da der Build innerhalb eines OpenShift-Clusters läuft, wird nun durch das OpenShift-Jenkins-Client-Plug-in ad hoc ein vorkonfiguriertes Maven-Container-Image als Pod gestartet. Im Vergleich zur alten Welt mit statischen Jenkins-Slaves ist es nun möglich, dynamische Workloads auf den Clustern auszuführen. Das ist vor allem bei großen Enterprise-Umgebungen interessant, da keine ungenutzten Compute-Ressourcen vorgehalten werden müssen.

Ist dieser Schritt erfolgreich, wird als Nächstes ein „Regular" OpenShift-Build-Objekt erzeugt, da Jenkins selbst keinen Zugang zum Docker-Socket hat und somit auch kein Docker-Image bauen kann. Hierbei ist es wichtig, die Unterschiede zwischen Regular-Builds und Pipeline-Builds zu verstehen.

Abbildung 4 zeigt, wie bei den Regular-Builds S2I- oder Docker-native Builds durchgeführt werden können. Diese haben als einzige Objekte im OpenShift-Cluster Zugriff auf den Docker-Daemon, können ein Container-Image erstellen und an die interne Container-Registry pushen. Dadurch wird ein neues Image-Stream-Objekt erzeugt beziehungsweise ein vorhandenes aktualisiert, was die weiteren Schritte in der Build-Pipeline triggert.

Bei den Pipeline-Builds wird hingegen anhand des angegebenen Jenkinsfiles ein normaler Jenkins-Build getriggert, der per Default keine Container-Images erzeugen kann. Um dies zu erreichen, ist ein weiteres Build-Objekt zu erzeugen, nämlich eines der beiden Regular-Builds. Die durch das OpenShift-Jenkins-Client-Plug-in bereitgestellten Credentials können dabei „out of the box" im Jenkinsfile genutzt werden, um per CLI oder Groovy-API den Regular-Build zu erzeugen. Technisch gesehen wird dafür ein angepasstes Jenkins-Slave-Image genutzt, das per Umgebungsvariablen die benötigten Cluster-Credentials gesetzt bekommt. Jenkins dient dabei mehr oder weniger als Wrapper, um das Java-Artefakt zu bauen und danach den eigentlichen Container-Build anzustoßen.

Aufgrund der derzeit genutzten Docker-Version in OpenShift sind keine Docker-Multi-Stage-Builds möglich, die diesen zusätzlichen Wrapper vermeiden könnten. Alternativ kann der Entwickler auch ein eigenes S2I-Builder-Image schreiben, bauen und in OpenShift einbinden, was allerdings zusätzlichen Pflegeaufwand bedeutet. Aus den genannten Gründen haben derzeit alle Wege eine gewisse Komplexität, die der Entwickler selbst lösen muss *(siehe Abbildung 5)*.

Zudem kann es nützlich sein, die gebauten Container-Artefakte nach jedem Build eines bestimmten Branch (etwa bei Feature-Branches) automatisch zu deployen. Wichtig ist, dass die genutzte Deployment-Konfiguration identisch mit der Produktions-Konfiguration ist. Dafür können bei OpenShift native YAML-Templates genutzt werden, die für die jeweilige Stage per Parameter angepasst werden. Beispielsweise kann das Feature-Branch-Deployment ein Präfix benutzen und steht danach unter einer eigenen URL *„feature-X.dev.bakery.mycluster"* zur Verfügung.

QA Stage

Nach dem erfolgreichen Durchlaufen der Development-Stage sind die erzeugten Artefakte nun ausgiebig zu testen. Je nach Projekt werden dafür manuelle oder automatische Aktionen notwendig. Im einfachsten Fall wird die Anwendung bei einer Code-Änderung im Haupt-Entwicklungspfad auf die QA-Umgebung eingerichtet und manuell getestet. Das sollte allerdings nicht das Ziel sein, da neue Features oder ein Bugfix möglichst schnell in Produktion gehen sollten. Besser ist es, abgeschlossene Features, die zum Beispiel durch einen Pull-Request gekennzeichnet sind, automatisiert mit einer Vielzahl von Integration-Tests zu testen. Im Idealfall ist die aufgebaute Testsuite so mächtig, dass auf manuelle Tests verzichtet werden kann.

Unsere Beispiel-Applikation beinhaltet sowohl API-Schnittstellen als auch eine grafische Web-Oberfläche. Beides sollte ausreichend getestet sein, um sicherzustellen, dass die Code-Änderung keine ungewollten Nebeneffekte hat. Da die Applikation in der Dev Stage durch Unit-Tests auf tieferer Ebene getestet wurde, können jetzt verschiedene Black-Box-Tests auf einem produktionsnahen System folgen. Der in *Abbildung 6* dargestellte Ablauf wurde dabei wie folgt umgesetzt:

- Deployment des Zielsystems mithilfe eines OpenShift-Deployment-Templates
- Testen des REST-API mithilfe von Citrus-Integration-Tests
- Testen der Web-UI und des PDF-Reports mit Sakuli-E2E-Tests

Zunächst muss das sogenannte „System Under Test" (SUT) eingerichtet sein. Wie bereits beschrieben, ist es wichtig, in ein nahezu produktionsnahes System zu deployen, das sich lediglich durch Parameter wie Verbindungsdaten (HTTP-Endpoints, Message-Broker-Host, Credentials) unterscheidet. Um die Durchlaufzeit der Pipeline zu reduzieren, ist das Deployment so zu optimieren, dass die Komponenten parallel gestartet werden können.

Besteht eine Abhängigkeit zu anderen Services, ist es hilfreich, das sogenannte „Init Containers"-Konstrukt in der Pod-Definition zu

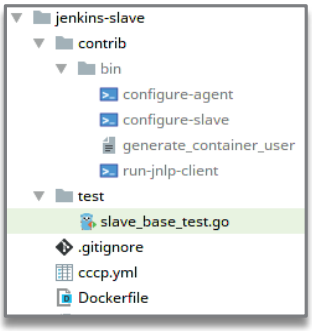
```
# Dockerfile
FROM consol/citrus:2.7.5

### sourced are copied from:
https://github.com/openshift/jenkins

# Copy the jenkins-slave entrypoint
ADD contrib/bin/* /usr/local/bin/

# Run the Jenkins JNLP client
ENTRYPOINT ["/usr/local/bin/run-jnlp-client"]
```

Used by



Jenkinsfile

```
podTemplate(label: "citrus",
    cloud: "openshift",
    inheritFrom: "maven",
    containers: [
        containerTemplate(name: "jnlp",
            image: "citrusframework//jenkins-slave",
        )
])
{
    node('citrus') {
        sh "echo execute oc citrus build"
        checkout scm
        sh "mvn install"
        junit 'citrus-tests/target/citrus-reports/**/*.xml'
        archiveArtifacts "citrus-tests/target/citrus-*/**/*"
    }
}
```

Abbildung 7: Custom-Jenkins-Slave-Image

nutzen. Dort kann mit einfachen Skripten überprüft werden, ob die konfigurierte URL oder der Host bereits erreichbar ist, ohne dass der eigentliche Container startet. Das vermeidet, dass beim Starten der eigentlichen Anwendungscontainer Ressourcen-hungrige Crash-Loops entstehen, da gerade Java-Anwendungen einen hohen Ressourcenverbrauch beim Start haben.

Sind alle Komponenten erreichbar, kann das API getestet werden. Hierfür wird im Beispiel das Citrus Integration Testing Framework *(siehe „https://citrusframework.org")* benutzt, das es ermöglicht, Unit-Test-artige Java-Tests zu schreiben, die sämtliche gängigen Message-Protokolle (HTTP REST, SOAP, JMS etc.) und Nachrichtenformate (XML, JSON, XSD etc.) beherrschen. Zudem bietet Citrus die Möglichkeit, nicht nur einfache Request-Response-Szenarien zu testen, sondern auch komplexe, Nachrichten-basierte Message-Workflows abzubilden, bei denen beispielsweise Message-Payloads über mehrere Nachrichten und Endpoints hinweg verarbeitet werden.

Um den von Citrus bereitgestellten Container möglichst nahtlos zu integrieren, kann die Groovy-DSL mit einem eigenen Jenkins-Slave-Image erweitert werden. Dieses Vorgehen bietet sich an, da die Tests wie die Unit-Tests selbst per Maven getriggert werden. Eine kurze Beschreibung steht in *Abbildung 7*.

Zunächst wird ein Jenkins-kompatibles Image benötigt. Um ein eigenes Image zu bauen, empfiehlt es sich, den Sourcecode des Maven-Slave-Image *(siehe „https://github.com/openshift/jenkins/tree/master/slave-maven")* zu kopieren und seine eigene Implementierung darauf aufzusetzen. Das Image lässt sich anschließend als neuer Node „citrus" ansprechen. Die darin aufgeführten Befehle wie „mvn clean install" werden dann innerhalb des zur Laufzeit gestarteten Containers ausgeführt. Sind alle Integration-Tests erfolgreich, wird im Anschluss die UI getestet.

Für die UI-Tests wurde im Beispiel Sakuli *(siehe „http://www.sakuli.org")* genutzt. Sakuli erlaubt es, sowohl Web- als auch native Oberflächen innerhalb eines Kontextes zu testen. Konkret wurden vier parallel laufende Testfälle definiert. In den drei Web-basierten Tests wurden unterschiedliche Produkte über die Web-Oberfläche bestellt und im Anschluss die erfolgreiche Abarbeitung auf dem Report-Server überprüft. Im vierten Fall wurde über die native Browser-Funktion „Als PDF speichern" eine PDF-Version des Produktions-Reports erzeugt und im Anschluss in einem nativen PDF-Viewer validiert.

Die Kombination von Web und OpenCV-basierter, visueller Testmethodik macht es möglich, komplexe End-to-End-Workflows zu testen. Damit ist sichergestellt, dass die Anwendung aus End-User-Perspektive wie gewünscht funktioniert. Diese High-Level-UI-Tests sind notwendig, da bei der Vielzahl der Technologien und Frameworks, die ein modernes Entwicklungsprojekt einsetzt, immer ein unvorhersehbarer Seiteneffekt auftreten kann, der nur beim finalen End-to-End-Szenario sichtbar wird. *Abbildung 8* zeigt, wie ein solcher Test mithilfe der von Sakuli bereitgestellten Container innerhalb eines OpenShift-Clusters ausgeführt wird und der Entwickler per Web-VNC den Testlauf beobachten kann.

Damit die Laufzeit optimiert ist, werden die Testfälle parallel ausgeführt. Bei größeren Testsuites lassen sich unter anderem verschiedene Browser in unterschiedlichen Versionen testen. Technisch gesehen wird innerhalb der Build-Pipeline dafür für jede Test-Instanz ein Pod im Cluster gestartet. Anschließend wird mit dem kleinen Helper-Skript „validate_pod-state.sh" überprüft, ob der Pod mit Status „Succeeded" oder „Failed" bereits fertig durchlaufen wurde. Der daraus resultierende Exitcode des Containers wird validiert und Logs sowie Screenshots zur eventuellen Fehler-Analyse in einen persistenten Speicher kopiert. Ist einer der parallelen Testläufe nicht erfolgreich, wird die QA-Build-Pipeline als fehlerhaft markiert, was im „Jenkinsfile.qa" konfiguriert wurde.

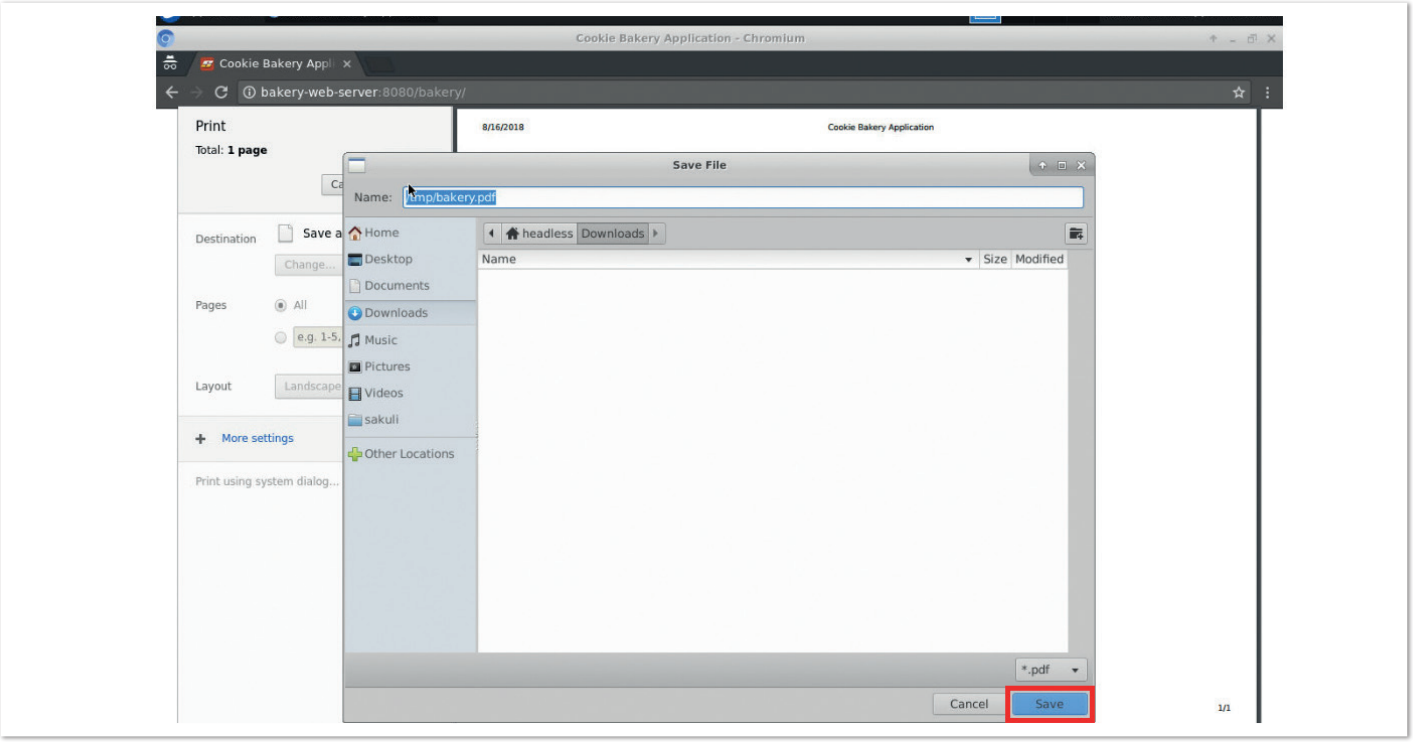


Abbildung 8: Sakuli-UI-Test in OpenShift

Die umfassenden automatisierten Tests können den Reifegrad des gebauten Image verifizieren. Ist alles in Ordnung, wird es für die nächste Stage freigeben. Dies kann entweder mit einem Update eines ImageStream-Objekts erfolgt oder mit dem Aufruf einer weiteren Build-Pipeline.

Production Stage

Wurde die QA Stage erfolgreich durchlaufen, kann das Deployment in das produktive System erfolgen. Meist wird dafür ein eigener Namespace (abgebildet als separates Projekt in OpenShift) oder ein separates Cluster genutzt, um die Umgebung entsprechend der Firmen-Policies abzusichern. Um Audit-Vorgaben für das produktive Deployment gerecht zu werden, lässt sich in die Jenkins-Pipeline eine User-Input-Aktion einbauen *(siehe Listing 1)*. Im Jenkins-Log ist dann zu sehen, welcher User das „OK" gesetzt hat. So kann ein hoher Automatisierungsgrad erreicht werden, ohne die Möglichkeit des manuellen Eingreifens zu nehmen.

Es ist ratsam, die Pipeline so zu gestalten, dass man das gleiche Deployment-Template wie in der QA Stage benutzen kann und nur Parameter anzupassen sind. Nach dem produktiven Deployment sollte ein sogenannter „Smoke-Test" durchgeführt werden, der nochmals sicherstellt, dass alles erfolgreich war. Je nach Komplexität können hier ein Subset der Integration-Tests oder ein kleiner Connection-Test wie ein sogenannter „Wait-for-it"-Container *(siehe „https://github.com/toschneck/wait-for-it")* benutzt werden. Das Ergebnis des gesamten Builds beziehungsweise des Deployments sollte per E-Mail oder Chat Notification dem verantwortlichen Team mitgeteilt werden, um im Fehlerfall schnell reagieren zu können.

Fazit

Beim Aufbau der Build-Pipelines mit OpenShift wird deutlich, dass Jenkins als statische Java-Anwendung nicht für den Bau und das Verwalten von Pods und Containern entwickelt wurde. Selbst mit den von OpenShift gelieferten Plug-ins ist es mühsam, das OpenShift-API direkt anzusprechen. Im Unterschied zum klassischen Jenkins kann bei der OpenShift-Integration das Jenkinsfile derzeit nicht einfach angepasst und ein neuer Build gestartet werden. Bei jeder Änderung am Jenkinsfile ist ein neuer Commit zu erzeugen, was wiederum bedeutet, dass die komplette Pipeline erneut bis zur eigentlichen Änderung ausgeführt wird. Das Resultat ist, dass das Schreiben und Pflegen der Pipeline sehr viel Aufwand verursacht. Statt das Groovy-API zu nutzen, empfiehlt sich als Workaround, direkt über Skripte mit dem „oc"-Client den API-Befehl für das Cluster abzusetzen. Der Vorteil ist, dass die einzelnen Schritte der Pipeline in Skripte verpackt sind und unabhängig voneinander ausgeführt beziehungsweise entwickelt werden können. Zudem ist der Entwickler unabhängiger vom Release-Zyklus des OpenShift-Jenkins-Client-Plug-ins und kann dieselbe OpenShift-Client-Version benutzen, in der der Cluster selbst betrieben wird.

Beim Arbeiten mit den Pipelines entsteht schnell der Eindruck, dass es zu viele Wrapper gibt, um letztendlich den eigentlichen Befehl für die Erzeugung der OpenShift-Objekte auszuführen. Zudem gibt es per Default kein einheitliches Vorgehen, wie auf die während des Builds erzeugten Laufzeitdaten wie Logs oder Screenshots zugegriffen werden kann. Hier ist deutlich sichtbar, dass Jenkins nicht dafür entwickelt wurde, auf dynamisch verteilten System-Landschaften wie OpenShift eingesetzt zu werden.

```
stage('Trigger PROD') {
  timeout(time: 2, unit: 'DAYS') {
    input message: '==> deploy PROD system?'
  }
 //… next steps
}
```

Listing 1

Das Hinzufügen eigener Node-Container ist zwar technisch möglich, aber eher als Workaround anzusehen als ein wirkliches durchdachtes Konzept, da es zu dem Thema keine Dokumentation gibt. Die Dokumente sind generell über viele Quellen verteilt: OpenShift, Kubernetes, Jenkins, Jenkins–Plug-in, Docker.

Als Gesamteindruck bleibt eher eine gefühlte Zwangsheirat als eine glückliche Ehe hängen. Mit viel Mühe und eigenen Workarounds lässt sich viel erreichen. Wer allerdings ein ready-to-use, Cloud-native denkendes Build-Tool erwartet, wird enttäuscht sein. Unabhängig von der eher trägen Performance des CI-Systems gibt es vor allem für das Handling der Build-Skripte im Entwickler-Alltag noch einiges an Verbesserungspotenzial.

Ziel muss es im Cloud-native-Kontext sein, Build-Pipelines schnell aufzubauen und zu verändern, was die Jenkins-OpenShift-Integration nur in einfachen Use Cases zulässt. Derzeit gibt es allerdings auch kein anderes Tool, das komplexe Use Cases standardmäßig abdecken kann. Als alternativ zu beobachtende Lösungen sind unter anderem das GitKube-Projekt *(siehe „https://github.com/hasura/gitkube")*, Skaffold von Google *(siehe „https://github.com/GoogleContainerTools/skaffold")*, GitLab CI *(siehe „https://gitlab.com")*, DroneCI/KubeCI *(siehe „https://www.kubeci.io")* und JenkinsX *(„https://github.com/jenkins-x/jx")* zu erwähnen. Es kann davon ausgegangen werden, dass sich beim CI/CD-Tooling für verteilte Systeme in den kommenden Monaten viel Spannendes tun wird und wir erst am Anfang einer langen Reise Richtung containerisierte CI/CD-Pipelines stehen.

Tobias Schneck
tobias.schneck@loodse.com

Tobias Schneck ist Senior Software Engineer bei Loodse. In seiner langjährigen Laufbahn hat sich Tobias auf Test-Automatisierungs- und CI-Projekte spezialisiert. Aktuell widmet er sich mit Leidenschaft den vielfältigen Anwendungsmöglichkeiten von Container-Technologien sowie Kubernetes und möchte diesen zum breiten Durchbruch verhelfen, um Entwicklern das Leben zu erleichtern. Tobias Schneck wurde von verschiedenen Fachkonferenzen als Sprecher eingeladen und organisiert das „Agile Testing @Munich"-Meetup.

20. - 23. November in Nürnberg

2018.doag.org

PROGRAMM ONLINE
mit rund 450 Vorträgen

Eventpartner: AOUG SOUG swiss oracle user group iJUG Verbund ORACLE

JavaLand

Early Bird
bis 15. Jan. 2019

19. - 21. März 2019 in Brühl bei Köln

Ab sofort Ticket & Hotel buchen!

www.javaland.eu

Präsentiert von: DOAG Heise Medien

Community Partner: iJUG Verbund

www.javaland.eu