Continuous Delivery Praktisch für Enterprise

Dr. Halil-Cem Gürsoy
adesso AG
Dortmund

Schlüsselworte

Continuous Delivery, Continuous Integration, Hudson, Jenkins, Maven, Puppet

Einleitung

"Continuous Delivery" ist zur Zeit „in“. In diversen Fachartikeln und Büchern wurden die zugrunde liegenden Konzepte tiefgehend im Detail erläutert. Die Frage, die sich dann aber den geneigten Lesern stellt ist wie nun solch ein Konzept in einem Enterprise-Umfeld in die Wirklichkeit umgesetzt werden kann. Und welche Werkzeuge notwendig sind und was beachtet werden muss.

In dieser Session implementieren wir gemeinsam eine Build und Delivery Pipeline und setzen dabei Werkzeuge wie Hudson/Jenkins, Puppet, Vagrant und Oracle Virtualbox ein. Dieses Thema kann natürlich in so einer beschränkten Zeit nicht umfassend behandelt werden, aber diese Session sollte die Teilnehmer inspirieren eine Implementierung selbst anzugehen und auszuprobieren.

Ein Szenario

Um eine Umsetzung der Konzepte, auf die gleich noch einmal ganz kurz eingegangen wird, durchzuführen definieren wir erst einmal ein sehr einfaches Szenario:

Es liegt eine Web-Applikation (Java, WAR-Datei) vor. Diese muss gebaut und auf einem Test-Server (in unserem Fall eine lokale Virtuelle Maschine), dann später auf einem Vorproduktions- und einem Produktionsserver deployt werden (wir deployen hier erst einmal in nur einen Server). Diese Builds und Auslieferungen müssen regelmäßig erfolgen. Ideal wäre nach jeder Änderung in dem Projekt. Zudem sollte es so sein, dass jeder Build potentiell auch produktiv gesetzt kann aber hierfür keine manuellen Eingriffe durchgeführt werden sollten, maximal abgesehen von einer Freigabe-Aktion.

Continuous Delivery in wenigen Sätzen

Nachdem wir unser Szenario definiert haben sind sicherlich einige Fragen aufgekommen. „Warum jedes Build potentiell Produktiv setzen?“ oder „Warum keine manuellen Eingriffe?“ und „Wo sind meine Server?“ sind solche die dem Autor spontan einfallen. Daher sollte, wenn auch bereits in der breiten Literatur ausführlich besprochen, das Konzept von Continuous Delivery kurz besprochen werden.

Als erstes nehmen wir das „*Agile Manifesto*“ zu Hand und lesen als ersten Punkt:

> *Our highest priority is to satisfy the customer through early and **continuous delivery** of valuable software*

Demzufolge wird hier auf eine kontinuierliche Auslieferungen von Software abgestellt. Kontinuierlich bedeutet aber nicht „regelmäßig einmal pro Quartal“ sondern tatsächlich gegebenenfalls mehrmals an einem Tag. Aus dieser Forderung heraus ergeben sich Konsequenzen für das Buildsystem eines Projektes:

- es müssen kontinuierlich Builds erfolgen
- die erzeugten Artefakte müssen vollständig durchgetestet werden (nicht nur technisch sondern auch fachlich)
- Definitionsgemäß sind solche Artefakte nach erfolgreichem Bestehen der Tests produktionsreif und können ausgeliefert werden
- Der Build, alle Tests und die Auslieferung müssen vollautomatisiert erfolgen
- Die Bereitstellung von Ressourcen wie Server und Datenbanken muss ebenfalls vollautomatisch erfolgen.

Ziel ist es, unter Berücksichtigung dieser Bedingungen ein System zu schaffen, das schnell und fehlerfrei automatisiert arbeitet.

Werkzeugauswahl

Um diese Ziele zu erreichen benötigen wir einige Werkzeuge. Da der Teilnehmer dieses Verfahren gegebenenfalls auf einem lokalen System austesten möchte muss dies auch beachtet werden. Aus diesem Grunde fällt die Entscheidung auf folgende Werkzeuge die wir kurz näher betrachten möchten.

Vagrant

Bei Vagrant handelt es sich um ein Werkzeug zur Provisionierung von Virtuellen Maschinen oder Cloud-Ressourcen. Initial bot dieses Werkzeug nur einen Support für Oracle VM VirtualBox, inzwischen wurde der Support durch diverse Plugins auf verschiedene Virtualisierungsplattformen ausgedehnt. Vereinfacht gesagt kann mit Hilfe von Vagrant das Anlegen einer Virtuellen Maschine in VirtualBox in Dateien konfiguriert werden. Diese Konfigurationsdateien bestehen aus reinem Ruby-Code, so dass sich Entwickler dort schnell wieder finden können. Ein Beispiel:

```
config.vm.define :vm1 do |vm1_config|
    vm1_config.vm.hostname = 'simplebox1'
    vm1_config.vm.network :private_network, ip: '192.168.56.11'
    vm1_config.hostmanager.aliases = %w(simplebox1.localdomain)
    vm1_config.vm.provider :virtualbox do |vb|
      vb.gui = true
      vb.name = "simplebox1"
      vb.customize ["modifyvm", :id, "--memory", "1024"]
      vb.customize ["modifyvm", :id, "--cpus", 1]
    end
    vm1_config.vm.provision :puppet do |puppet|
      puppet.manifests_path = "manifests"
      puppet.module_path = "modules”
      puppet.manifest_file  = "simple-shard-1.pp"
      puppet.options = "--verbose"
    end
  end
```

In diesem Beispiel wird Vagrant mitgeteilt, dass eine VM mit dem Namen „simplebox1“ angelegt werden soll. Dabei soll der VM 1024Mb Hauptspeicher und ein CPU-Kern zugewiesen werden. Wenn diese Virtuelle Maschine angelegt wurde soll diese bitte mit Hilfe des Provisioners *Puppet* fertig erstellt werden. Wird nun Vagrant unter Verwendung dieser Konfigurationsdatei gestartet so wird eine VM mit den entsprechenden Parametern neu angelegt und zur weiteren Verarbeitung wird Puppet gestartet. Was es sich dabei mit Puppet auf sich hat, schauen wir uns nun im Weiteren an:

Puppet

Bei Puppet handelt es sich um ein Werkzeug das den Sollzustand eines Systems (z.B. Server) definiert und dieses System in diesen Zustand überführt. Die Definition erfolgt dabei mit Hilfe einer DSL. Wird solch eine Definition auf ein System angewandt so wird ein Katalog aus der Differenz des Ist- und des Soll-Zustandes berechnet in dem alle notwendigen Aktionen aufgelistet sind um den Soll-Zustand zu erreichen. Dabei abstrahiert Puppet betriebssystemsspezifische Aspekte. Soll z.B. *Apache httpd* auf einem Linux-Server installiert werden so teilt man dies in der Definition mit, unabhängig davon, welche Distribution zur Verfügung steht. Das dafür zuständige Modul bestimmt dann selbst ob z.B. ein DEB oder RPM-Paket installiert werden muss und wie diese heißen. Alle Aktionen die definiert werden müssen idempotent sein, also eine wiederholte Anwendung sollte jedes Mal zu dem gleichen Ergebnis führen.

Zusammengefasst kann auch gesagt werden, dass mit Hilfe von Puppet Infrastruktur als Code ausgedrückt wird („Infrastructure as Code“), was zu einer Wiederholbarkeit, Nachvollziehbarkeit und Versionierbarkeit von Infrastruktur-Konfigurationen führt.

Hudson/Jenkins

Bei Hudson bzw. Jenkins, dem „Geschwister-Produkt“ handelt es sich um einen erweiterbaren *Continuous Integration*-Server. Dieser Server ist in der Lage, beispielsweise Quelldateien von einem Versionsverwaltungssystem wie Git zu beziehen und bei Änderungen mit Hilfe von *Apache Maven* oder anderen Build-Werkzeugen Artefakte zu erzeugen. Die Funktionalität von Hudson/Jenkins kann sehr stark durch die Nutzung von Plugins erweitert werden die in diesem Szenario auch zum Einsatz kommen werden (Details später im Rahmen der Implementierung).

Weitere Werkzeuge

Es werden natürlich auch weitere Werkzeuge benötigt. Zu nennen wären da ein Versionsverwaltungssystem wie Git oder Subversion und ein Build Werkzeug, wie z.B. Gradle oder Apache Machen. Zudem benötigen wir einen Repository Server wie Sonatype Nexus oder Apache Archiva um die erstellten Artefakte zu verwalten.

Die Umsetzung

Nach dem wir nun unsere Werkzeugkiste und unser Szenario definiert haben geht es an den praktischen Teil der Umsetzung.

Als Prozess betrachtet haben wir ganz vereinfacht vier Schritte vorliegen:

Im ersten Schritt wird das Artefakt „gebaut“, danach erfolgt die Provisionierung eines lokalen Servers. In diesem Server wird mit Hilfe von Puppet ein Application Server installiert und in diesen Application Server die Applikation deployt. Im letzten Schritt erfolgt ein Test der Applikation.

Diese einzelnen Schritte werden in dem Build Server als „Job“ hinterlegt und diese werden miteinander verbunden so dass am Ende eine durchgehende Kette, eine sogenannte „Build-Pipeline“ entsteht.

Der Build

Für den Build-Job werden in dem Build-Server und dem Build-Werkzeug einige zusätzliche Plugins verwendet. Da jeder Build potentiell produktiv gesetzt werden kann werden keine Snapshots gebaut, statt dessen werden fortlaufende Versionsnummern (z.B. bestehend aus der Build-Nummer) vergeben. Hierzu verwenden wir die Maven-Plugins „build-helper-maven-plugin“ (stellt einen Versions-Parser für Maven-POMs zur Verfügung), „versions-maven-plugin“ (kann neue Projektversionen setzen) sowie das Jenkins-Plugin „Parameterized Trigger Plugin“, mit dessen Hilfe Folge-Jobs angestoßen und diesen Parameter übergeben werden können.

Der Job wird nun so konfiguriert dass die Sourcen des Projektes regelmäßig mit wenigen Minuten Abstand auf dem Versionsverwaltungsserver überprüft werden. Der Build erfolgt mit Maven, so dass der Job entsprechend konfiguriert wird.

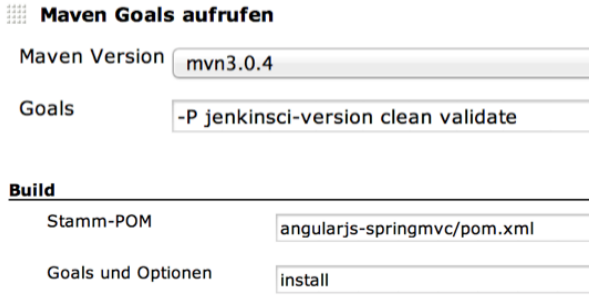

Abb. 1: Konfiguration des Build-Jobs in Jenkins. In einem ersten Schritt werden die neuen Versionen mit Hilfe von Maven gesetzt, im nächsten Schritt erfolgt der eigentliche Build mit dem Maven Goal „install“

Ist das Artefakt fertig gebaut wird dieses in einem Repository Server deployt. Hierzu bedienen wir uns ebenfalls Jenkins, alternativ wäre es auch möglich, direkt aus Maven in einen Repository Server die Artefakte zu laden.

Nun muss der nächste Schritt eingeleitet werden. Dies geschieht mit Hilfe des Jenkins „Parameterized Trigger Plugin“ da auch einige Parameter wie Artefakt-Version oder Revsionsnummer in der Versionsverwaltung mitgeteilt werden muss.

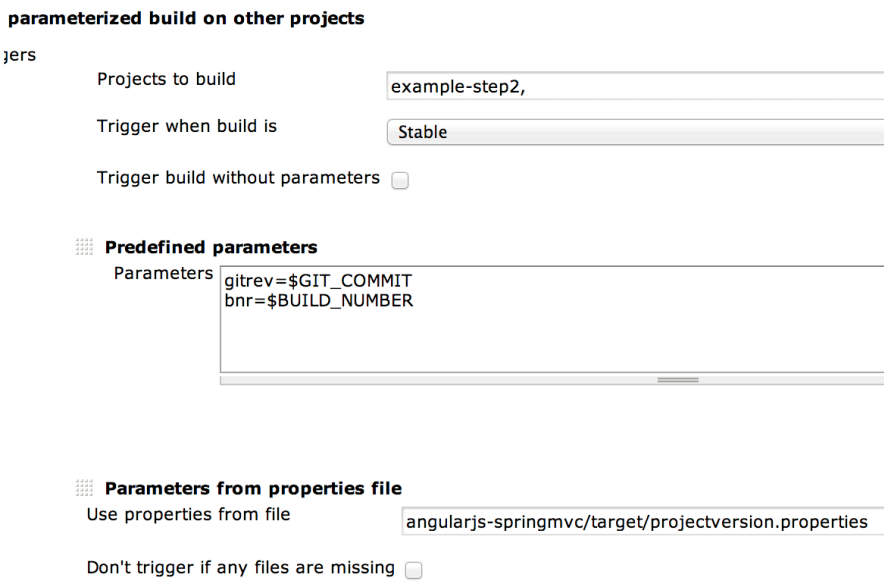

Abb. 2: Aufruf des Folge-Jobs mit übergebenen Parametern

Server-Provisionierung

Das Deployment möchten wir gerne in eine lokale VM durchführen, die hierzu neu angelegt und gestartet werden muss. Zudem muss in diesem Schritt in dem Server der Application Server Apache Tomcat 7 installiert und entsprechend konfiguriert werden. Dies alles geschieht mit Hilfe von Vagrant und Puppet. Vagrant startet eine VM auf Basis einer sogenannten Base-Box, die für Vagrant vorbereitet wurde. Ist diese VM konfiguriert und gestartet so wird durch Vagrant Puppet auf diesem Server ausgeführt. Puppet sorgt nun dafür, dass alle notwendigen Schritte unternommen werden um im nächsten Schritt die Applikation zu deployen. Ein Auszug aus der Definition für Puppet:

```
include tc7
```

Diese einfache Anweisung sorgt dafür, dass durch Puppet Tomcat 7 installiert und nach den Standardwerten konfiguriert wird.

Wichtig ist, dass in diesem Schritt aus dem Versionsverwaltungssystem die Revision bezogen wird, die zu dem davorliegenden Build-Schritt geführt hat denn auch die Infrastruktur-Aspekte wie z.B. die Konfiguration von Tomcat ist nun versioniert.

Der Aufruf von Vagrant erfolgt aus Jenkins heraus.

Das Deployment der Applikation

Nun liegt der Application Server einsatzbereit vor, das Artefakt ist im ersten Schritt gebaut worden und muss „nur noch“ deployt werden. Da es eine unnötige Ressourcenverschwendung[1] wäre das Artefakt erneut zu bauen wird das Artefakt aus dem Repository Server bezogen. Dies erledigen wir mit dem Jenkins-Plugin „Repository Linker“. Danach erfolgt das Deployment in den vorbereiteten Tomcat-Server, ebenfalls mit einem Jenkins-Plugin, dem „Deployment Plugin“, welches auf dem Cargo-Plugin beruht. Wenn dies erfolgreich war liegt nun die Applikation einsatzbereit vor.

---

[1] Um ganz korrekt zu sein wäre dies nicht nur eine Ressourcenverschwendung sondern ist auch häufig die Quelle von Fehlern wenn z.B. beim zweiten Build das Artefakt sich von der ersten Version unterscheidet

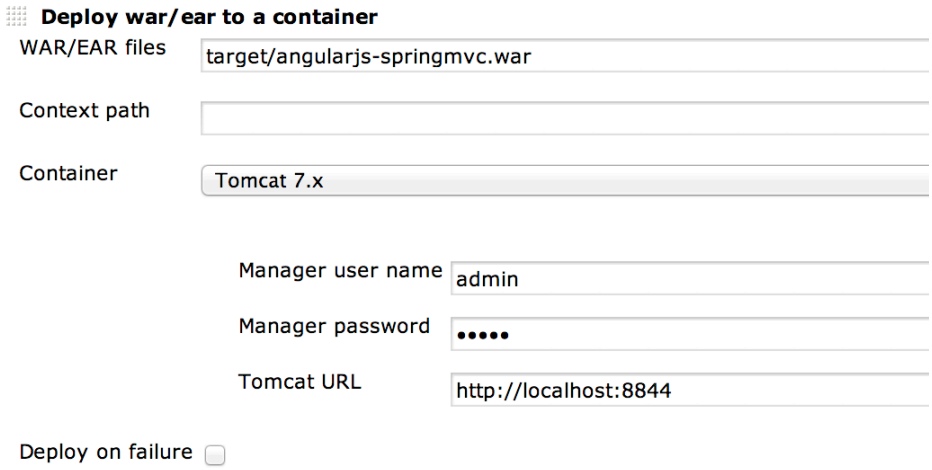

Abb. 3: Konfiguration des Jenkins Deployment Plugins

Testen

Als letzter Schritt kommt nun das Testen der Applikation. Ideal wäre es, Testwerkzeuge wie Selenium (für Oberflächen) und weitere Testwerkzeuge für z.B. Integrations- und Schnittstellen-Tests einzusetzen. Da dies den Rahmen dieser kleinen Session sprengen würde beschränken wir uns auf ein einfaches WGET in dem Job welches auf das Deployment folgt. Dies wird aus dem Plugin „Execute Shell“ heraus ausgeführt.

Die komplette Build-Pipeline

Nach diesen wenigen Schritten haben wir schon eine sehr einfache, wenn auch effektive Build-Pipeline aufgebaut. Diese beinhaltet natürlich noch nicht Aspekte wie parallele Testausführungen, Provisionierung von Cloud-Ressourcen oder die Versionierung von Datenbank-Schemata in einer RDBMS. Grundsätzlich würden auch diesen Schritten konzeptionell das gleiche Verfahren zu Grunde liegen. Mit dieser Pipeline sind wir jetzt schon in der Lage, ein kleines Freischalt-Verfahren einzuführen (mit Hilfe des dann einzusetzenden „Promoted Build Plugins“) und auch im Fehlerfall (z.B. Fachlichkeit ist nachträglich als falsch umgesetzt bewertet worden) auf eine ältere Version der Applikation zurück zu fallen – dies kann z.B. mit dem Jenkins „Build Pipeline Plugin“ erledigt werden, welches zudem eine nette Ansicht über den aktuellen Status der Pipeline ermöglicht.

Abb. 4: Unsere Build Pipeline. Der letzte Build war vollständig erfolgreich, während im Build #32 der Test nicht erfolgreich war; daher wurde auch der folgende Schritt nicht mehr ausgeführt.

Zusammenfassung

Mit sehr einfachen Mitteln und unter Einsatz von freien (Oracle) Werkzeugen kann bereits eine simple aber effektive Build Pipeline für Continuous Delivery implementiert werden. Diese kann sukzessive ausgebaut werden. Wichtig ist dabei zu beachten, dass auch die Infrastruktur als „Code“ behandelt wird.

Kontaktadresse:
Dr. Halil-Cem Gürsoy
adesso AG
Stockholmer Allee 24
D-44269 Dortmund

Telefon: +49 231 930-9330
Mobil: +49 178 2808148
E-Mail halil-cem.guersoy@adesso.de
Internet: www.adesso.de
Twitter: @hgutwit
G+: http://goo.gl/hljRS