Continuous Deployment mit GIT, Jenkins und Docker

Gamze Uysal

Universität Stuttgart, Pfaffenwaldring 7, 70563 Stuttgart-Vaihingen, Deutschland
st142819@stud.uni-stuttgart.de

Abstract. Diese Seminararbeit befasst sich mit der kontinuierlichen Qualitätssicherung, mithilfe von Versionskontrollsystemen (z.B. GIT) und Integrationssystemen (z.B. Jenkins), im Rahmen des Studienprojekts „Mobilitätsplattform zur Integration verschiedener Mobilitätsanbieter“. Die Kernfrage lautet: Wie können Entwicklerteams derart kurze Release-Zyklen erreichen, ohne dabei auf die Qualitätssicherung (QA) zu verzichten? Die Antwort liegt in einem umfassenden, automatisierten Qualitätssicherungsprozess. In dessen Zentrum steht ein Feedback-Mechanismus, der den gesamten Weg einer Code-Änderung begleitet: von der Entwicklungsumgebung bis zum produktiven System. Es wird jede Code-Änderung von ihrer Entstehung in der Entwicklungsumgebung bis zu ihrem Einsatz im produktiven System automatisiert auf Einhaltung der Qualitätskriterien überprüft. *Continuous Deployment*, kurz CD, beschreibt solche Mikro-Release-Zyklen, bei denen jede geänderte Programmzeile zur Auslieferung einer neuen, vollständigen und qualitätsgesicherten Version führt. CD bildet einen komplett automatisierten Entwicklungsprozess. Jedoch gibt es noch zwei andere automatisierte Entwicklungsprozesse, die ein Teil von CD bilden: *Continuous Integration* und *Continuous Delivery*. Anlehnend zu diesen Entwicklungsprozessen, werden Versionskontrollsysteme, wie Subversion und Git, sowie ein Integrationssystem, Jenkins, vorgestellt.

Keywords: Continuous Deployment, Continuous Integration, Continuous Delivery, Subversion, SVN, Git, Jenkins, Docker, Integrationsserver, Versionskontrollsystem, Versionsverwaltung.

1 Grundlagen (Pipeline)

Im Allgemeinen werden drei Begriffe unterschieden: Continuous Integration, Continuous Delivery und Continuous Deployment. Diese Begriffe stützen sich jeweils auf Teile desselben Prozesses. Der Grad der Automatisierung ist der Hauptunterschied. Continuous Integration bildet die Grundlage indem es die ersten Schritte des Prozesses beschreibt. Continuous Delivery erweitert Continuous Integration um den Akzeptanztest. Mit Continuous Deployment lässt sich der vollständige automatisierte Prozess beschreiben. Die Unterschiede werden graphisch in Abbildung 1 dargestellt.

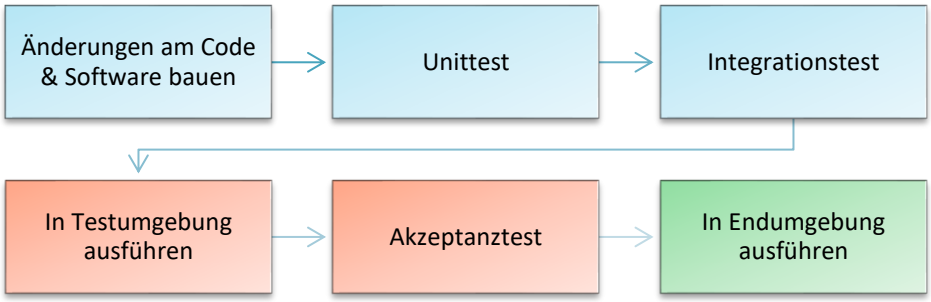

Abb. 1. Abgrenzung der Begriffe Continuous Integration, Continuous Delivery und Continuous. Continuous Integration besteht aus den Aufgaben in Blau, Continuous Delivery aus den blau und rot gefärbten und Continuous Deployment aus allen sechs Aufgaben.

2 Continuous Integration

Als Continious Integration, dessen Abkürzung auch als "CI" bekannt ist, bezeichnet man einen Prozess, in dem regelmäßig die Basis des Codes vollständig erstellt und auch getestet wird. Dies wird gemacht um Feedback zur Integration neuer Anwendungskomponenten zu bekommen. Versehentlich integrierte Fehler können dadurch schneller erkannt und auch behoben werden. Es ist ein unverzichtbarer Bestandteil von CI, dass die Entwickler ihre Code-Änderungen frühzeitig und regelmäßig in ein Versionskontrollsystem einpflegen. Typischerweise wird die Software nach jedem Commit, oder bei sehr zeitintensiven Builds zumindest regelmäßig mehrmals täglich, gebaut.
Im Anschluss werden Tests und Codeanalysen durchgeführt. Abschließend wird die Software beispielsweise auf einem Stage-System bereitgestellt. Auf Wunsch können die Entwickler über den Build-Vorgang informiert werden. Ebenfalls ist es für die Entwickler möglich, sich Logs, Analysen und Resultate anzeigen zu lassen.
Durch die zeitnahen und kontinuierlichen Rückmeldungen der Continuous Integration Server (Jenkins, GIT), sind fehlerhafte Commits für die Entwickler sehr schnell wahrzunehmen. Ebenfalls ist dies ein Vorteil für die Kunden der Entwickler. Sie können sich bei Bedarf jederzeit über den aktuellen Entwicklungstand des Projektes informieren und ein aktuelles Testsytem in Anspruch nehmen. [1]

2.1 Best Practices

Bei CI werden drei Schritte automatisiert durchgeführt (siehe Abb. 1.).

- Zu Beginn steht das Anstoßen der Pipeline. Dies geschieht in aller Regel durch den Commit von neuem Quellcode, kann aber auch manuell durch einen Entwickler durchgeführt werden. Dabei werden Jobs im Continuous Integration-Server gestartet, welche die weiteren Aufgaben ausführen. Außerdem sollten die Entwickler kontinuierlich Feedbacks geben. Builds sollten direkt ausgeführt werden.

- Im zweiten Schritt steht der Unittest. Bei Unittests werden kleine Einheiten – beispielsweise einzelne Klassen oder Komponenten – einer Software getestet. Die Tests sollten in einer herunterskalierten Version der Produktionsumgebung ausgeführt werden. Außerdem sollten nach jedem erfolgreichen Build, Build-Artefakte automatisch auf eine Testumgebung deployt werden.
- Der Integrationstest fasst einige kleine Tests zusammen. Außerdem stellt er eine Zusammenarbeit der einzelnen Komponenten des komplexen Systems und allen Schnittstellen sicher. Im Continuous Integration-Server werden dann die Ergebnisse der beiden Testarten Unittest und Integrationstest zusammengefasst und dargestellt. [2]

Wenn all dies umgesetzt wurde, ist man auf einem reifen CI-Level und bereit für den nächsten Schritt: Continuous Delivery (siehe Kapitel 3).

2.2 Vorteile

Fehler werden zu einem frühen Zeitpunkt im Entwicklungsprozess erkannt. Dadurch können die erforderlichen Änderungen schnell und effizient durchgeführt werden. So werden nicht nur die Kosten gesenkt, sondern durch das Schließen von zeitlichen Lücken zwischen den Integrationen wird auch der gesamte Release-Lebenszyklus beschleunigt und somit die zeitgerechte Abwicklung des Projekts sichergestellt. Außerdem verstärkt CI die Zusammenarbeit zwischen verschiedenen Entwicklungsteams und sorgt für eine verbesserte Übersicht. [3]

3 Continuous Delivery

Continuous Delivery (CD) ist eine Erweiterung der Continuous Integration (CI).
Der bestehende Code, kann nur durch einen völlig automatisierten Entwicklungsprozess, in eine funktionsfähige und auslieferbare Softwareversion übersetzt werden. Bereitgestellt, wird die Version, oft in einer produktionsähnlichen Umgebung. Somit kann die Software direkt in die Produktion genommen werden, da der getestete Code des CI Prozesses –möglichst automatisiert – eine reife und produktionsnahe Version erzeugt. Diese Prozesskette in Verbindung mit den verwendeten Tools wird auch als Delivery Pipeline bezeichnet. [4]

3.1 Best Practices

Der Vorsprung von Continuous Delivery zu Continuos Integration, ist der, dass die Funktion der Software zusätzlich getestet wird (siehe Abb. 1).

- Dazu wird die Software zuerst in eine Test-Umgebung überführt, wo festgestellt werden soll, ob die Software ihre Funktion erfüllt. Je nach Umgebung verhält sich die Software unterschiedlich. Daher ist es wichtig, dass eine Software beim Funktionstest in einer zur Einsatz-Umgebung ähnlichen Umgebung ausgeführt wird.

Übereinstimmen sollten die Betriebssysteme und Bibliotheksversionen sowie eingesetzte Technologien mit der zukünftigen Umgebung.
- Diese Funktionstests werden Akzeptanztests genannt. Die Unit- und Integrationstests legen einen starken Fokus auf die Code- beziehungsweise Architekturebene. Auf die Sicht des Kunden und späteren Nutzers, fokussiert sich der Akzeptanztest. Dafür werden Szenarien und Nutzer-Stories definiert, welche typische Nutzerinteraktionen mit dem System abbilden. Die volle Funktionalität des Systems, wird nur festgestellt, wenn die Testfälle des Akzeptanztests das System gut abbilden [5]

3.2 Vorteile

Ein Vorteil ist eine Reduzierung des Risikos von Deployments in Produktion. Da (fast) alles automatisiert ist (Test, Deployment, Infrastruktursetup und –deployment), Eine langfristige Kostenreduzierung, kann somit belegt werden. Außerdem entfallen lange Installationslisten und stundenlange Downtimes.
Auch werden agile Projektvorgehensmethoden durch Continuous Delivery unterstützt. Mit Continuous Delivery können regelmäßige Lieferungen in den Sprints vereinfacht eingehalten werden, da sich die Lieferung, Qualitätssicherung und Integration von Bugfixes extrem verschnellert. Sprints mit wenig Business-Value (Bugfix-Sprints, Liefer-Sprints, Refactoring-Sprints) werden vermieden und es gibt eine echte Definition of Done. Manuelle Tests sind zeitaufwändig und werden in Projekten auf Grund von Zeitmangel oft vernachlässigt. Die Qualität der Software steigt durch automatische Modul-, Integrations- und Akzeptanztests.

Das Risiko, dass eine Idee für die Kunden nicht funktioniert, ist reduziert, da man schnell Feedback bekommt, anstatt erst Monate oder Jahre später. Aus diesem Grund auch schnellere Time-to-Market: ein schneller und agiler Entwicklungsprozess, der einzelne Features schneller produziert. [6]

3.3 Unterschied CI und CD

Neue Features und Verbesserungen werden schneller auf den Markt gebracht, durch die Erweiterung von CI auf CD. Man nimmt inkrementellen Änderungen an der Software vor und verfolgt diese mit einem messbaren Feedback. Somit ist eine Leichtigkeit, Änderungen durchzuführen oder vorherige Zustände wiederherzustellen.
Auch sind nichtfunktionale (z. B. wirtschaftliche) Anforderungen erfüllt, da das Einfügen von „Quality Gates“ in wichtigen Stadien sichergestellt ist.

Eine Nachvollziehbarkeit der Artefakte (wer hat was gemacht), sogar vom Quellcode bis zur Lieferung der Software- Applikation, ist ermöglicht, da CD die gleichen Softwareprinzipien wie CI anwendet.

Mitarbeiter und Tools spielen beide eine ebenso wichtige Rolle, aber die Implementierung einer Software-Delivery-Pipeline an sich, erzeugt eine gemeinsame Sicht und eine gemeinschaftliche Antriebskraft für den gesamten Prozess und alle daran beteiligten Teams. [4]

4 Continuous Deployment

Continuous Deployment und Continuous Delivery teilen sich zwar den selben Kürzel (CD), doch Continuous Deployment ist aber eigentlich noch ein weiterer Schritt im Entwicklungsprozess. Eine produktionsreife Software, die nicht automatisiert in Produktion genommen wird bei Continuous Delivery erstellt. Der Ort und eine mögliche Bereitstellung der Software im Produktionssystem wird auch hier auch entschieden.
Continuous Deployment sieht es die erstellte Software automatisch in die Produktionsumgebung zu überführen. Durch einen komplett automatisierten Demployment Prozess, gibt es die Möglichkeit, Änderungen, welche bestimmte Tests erfolgreich durchlaufen haben, mit sofortiger Wirkung in der Produktion zu verwenden.
„Deployment Pipeline“ ist ein Begriff der gerade in diesem Zusammenhang nicht selten ist. Man kann es als das wichtigste Stück bei der agilen und automatisierten Softwareentwicklung bezeichnen. [7]

4.1 Best Practices

Die Abrundung der Pipeline, aus Abb. 1, bildet Continuous Deployment. Die bei Continuous Integration und Continuous Delivery getestete Software, wird zusätzlich, in die Ausführungsumgebung überführt und dadurch für den Benutzer erreichbar gemacht. Gerade wegen den häufigen Testausführungen und auch den klein gehaltenen Änderungen pro Pipelinedurchlauf ist es eine Seltenheit, dass Fehler in diesem Zustand vorkommen. [7]

4.2 Vorteile und Rahmenbedingungen

Das Zeitfenster für eine verlorene Gelegenheit ist nicht groß, da es beim Continuous Deployment so ist, dass jede Änderung direkt in die Produktion geht. Dadurch wird es möglich das Feedback schneller erfolgen kann. Der Einsatz von Feature Toggles macht es möglich, nach Produktion zu deployen, ohne die neuen Funktionalitäten zu nutzen. Es sind also Teile, die noch unfertig sind, bereits in Produktion deployt. So kann bereits Feedback über das Deployment der neuen Teile gesammelt werden.

Um frühzeitiges Feedback zu bekommen, kann man auch eine neue Funktionalität von einer kleinen Nutzergruppe bereits testen lassen. Auch wenn Continuous Deployment viele Vorteile wie z.B. (verkürzte Release-Zyklen, Qualitätsprüfung, schnelle Auslieferung) bietet, müssen bestimmte Rahmenbedingungen erfüllt werden. Gerade Versionskontrollen und Deployment Scripts sind neben dem CI Server erforderlich. Die Deployment Pipeline braucht auch eine flächendeckende Testumgebung, welche Unit-, Integrations- und GUI-Tests beinhaltet. Es können bei der Deployment Pipeline weiterhin, zur Verzögerungen führende, manuelle Schritte erforderlich sein. Obwohl die automatisierten Tests existieren. [8]

4.3 Continuous Delivery vs. Continuous Deployment

Durch Continuous Deployment wird nochmals eine Steigerung des Bereitstellungsprozesses erreicht. Beim Continous Delivery ist es so, dass Entscheider festlegen können, wann Versionen und neue Funktionen in die Produktionsumgebung überführt werden. Dieser Vorgang ist beim Continous Delivery komplett automatisiert. Die manuelle Entscheidung über eine neue Version oder auch über einem neuen Feature, die in die Produktionsumgebung überführt werden sollen, fällt damit weg, da diese Automatisch bereitgestellt werden. Die benötigten Prozesse um eine neue Version der Anwendung in der Produktionsumgebung bereitzustellen, starten, wenn eine neue Version hinzugefügt wird. [9]

5 Übersicht: CI vs. CD vs. CD

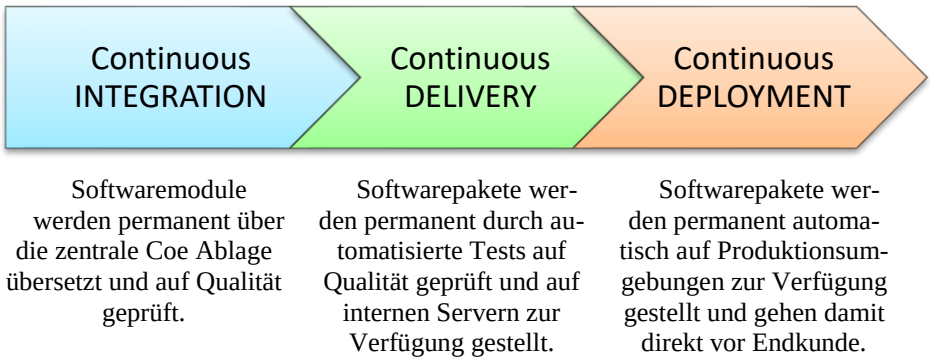

Abb. 2. Gegenüberstellung der drei automatisierten Entwicklungsprozesse.

6 Versionsverwaltung

In diesem Kapitel werden zwei Versionskontrollsysteme gegenübergestellt. Git vs. SVN. Beide Versionskontrollsysteme werden unter den Entwicklern diskutiert und beurteilt und schließlich wird für einer der Versionskontrollsystemen entschieden, denn die Entscheidung hängt auch stark von den Bedürfnissen des Teams und des Unternehmens ab. Git ist ein verteiltes Versionskontrollsystem (DVCS) und SVN ein Programm zur zentralen Versionskontrolle. Während in verteilten Versionskontrollsystemen jeder Entwickler seine eigene lokale Kopie des gesamten Versionsverlaufs besitzt, wird in zentralen Systemen alles nur in einem serverseitigen Projektarchiv gespeichert. [10]

6.1 Subversion

Subversion ist ein freies Open-Source Versionskontrollsystem, welches die Verwaltung von Dateien und Verzeichnissen und Änderungen erlaubt. Somit können Änderungen verfolgt werden und alte Versionen, sowie Daten wiederhergestellt werden. Durch den

netzwerkübergreifenden Mechanismus, ermöglicht das Subversion die Verwendung an verschiedenen Computern. Eines der größten Vorteile ist es, das Daten parallel von unterschiedlichen Personen bearbeitet und verwaltet werden können, ohne auf einen einzigen Kanal beschränkt zu sein, über den alle anderen Änderungen abgewickelt werden. Diesbezüglich kann das Vorankommen und die Entwicklung beschleunigt werden. Ebenfalls wird die Arbeit versioniert, d.h. man braucht sich keine Gedanken über den Verlust zu machen, denn falsche Änderungen an Daten können, können wieder rückgängig gemacht werden. [11]

6.2 GIT

Git ist ein dezentrales „Version Control System“, welches ursprünglich für die Sourcecode-Verwaltung des Linux-Kernels von Linus Torvald entwickelt wurde.
Grundsätzlich arbeitet ein „Version Control System“ gemeinschaftlich an einer Liste an Dateien. Ein VCS zeichnet sämtliche Veränderungen aller Dateien auf. Damit ermöglicht es eine gewisse Transparenz, welche Person, zu welchem Zeitpunkt, welche Änderung an einer oder mehrerer Dateien vorgenommen hat. [12]

Arbeiten mit Git. Jeder Entwickler eines Projektes, hat bei Git eine Kopie des ganzen Repositories auf seinem lokalen Rechner. Bei Git werden zwischen drei Bereichen unterschieden: Dem Arbeitsverzeichnis (working directory), dem überwachten Bereich (staging area, auch Index genannt) und dem Repository selbst.

- **Repository und Working Copy:** In ein Repository bzw. einem Repo befinden sich alle Dateien inklusive derer vorangegangenen Versionen. Alle Änderungen aller Dateien, die ins Repo gespielt werden – also wer, wann, welche Änderung durchgeführt hat – stehen zur Verfügung. Das besondere an Git ist, dass jede lokale Working Copy eines Users (ein "Klon" - via "git clone") wieder ein vollständiges, eigenes, lokales Repo darstellt. Es existieren somit mehrere Kopien der Repos, der, der einen Klon besitzt, kann daran arbeiten - inklusive kompletter History, auch offline und ohne Abhängigkeit von einem zentralen Server. Wenn die Änderungen aus dem eigenen Repo/der Working Copy als public angesehen werden, können sie wieder in das Remote-Repo "gepusht" werden (git push).
- **Branches:** Branches dienen dazu, einen separaten Arbeitszweig zu erstellen. Dieser kann dann auch als eine Kopie gesehen werden, in der gearbeitet wird. So kann z.B. die Programmierung eines Sicherheits-Patches in einem eigenen Branch erfolgen (im Kontext des Patches), der bei Fertigstellung und nach dem Testen zurück in den Master-Zweig eingearbeitet wird.
- **Versionierung:** Git protokolliert alle getätigten Änderungen, die in einer Working Copy gemacht werden, mit. Änderungen zu einem Repository können mittels „commit“ hinzugefügt werden, im Repo befindet sich dann eine neue Version der Datei(en). Es können verschiedene Versionen miteinander verglichen, Änderungen rückgängig oder zu einer früheren Version zurückgekehrt werden. Mit „git log“ können die Log-Informationen, die von Git mit aufgezeichnet werden, ausgegeben

werden. Mit „git status“ können die noch nicht ins Repo gespielten Änderungen der Working Copy aufgelistet werden. [13]

7 CI Build Server

7.1 Jenkins

Nachfolgend wird grundlegendes Wissen zum kontinuierlichen Integrationsserver Jenkins beschrieben. Bei Jenkins ist ein webbasiertes System zur kontinuierlichen Integration. Allerdings wurde die Software ursprünglich unter dem Namen Hudson entwickelt und bekannt, zu einer Umbenennung kam es aufgrund von Namensrechtunstimmigkeiten.
In Java ist es geschrieben und plattformunabhängig. Jenkins hat eine Große Basis, die zahlreiche Werkzeuge unterstützt, darunter SVN, Ant, Maven sowie JUnit. Auch können weitere Funktionen mit Hilfe von Plugins problemlos durch die Community hinzugefügt werden. Somit lässt sich Jenkins für jedes Projekt individuell anpassen. Andere Sprachen/Technologien wie z. B. PHP, Ruby oder .NET sind mit Jenkins in verschiedenen Projekten funktionell.
Über die intuitive Benutzeroberfläche lassen sich Testwerkzeuge über Plugins integrieren. Durch verschiedene Auslöser können Builds gestartet werden: z. B. Änderung des CVS oder Zeitplan (z. B. Nightly Builds). Nightly Builds, eine in der Nacht gebaut und getestete Applikation, sind besonders bei Open Source Projekten zu finden. [14]

Im Folgenden wird anhand der Jenkins-Konfigurationsseite (Abb. 3) die Hauptfunktionalität des Jenkins-Servers beschrieben: Source-Code-Management Der Abschnitt Source-Code-Management bietet die Möglichkeit, aktuelle Quelldaten zu erhalten. Als Standard werden hierfür Interfaces zu Versionierungstools wie Subversion oder Git bereitgestellt.

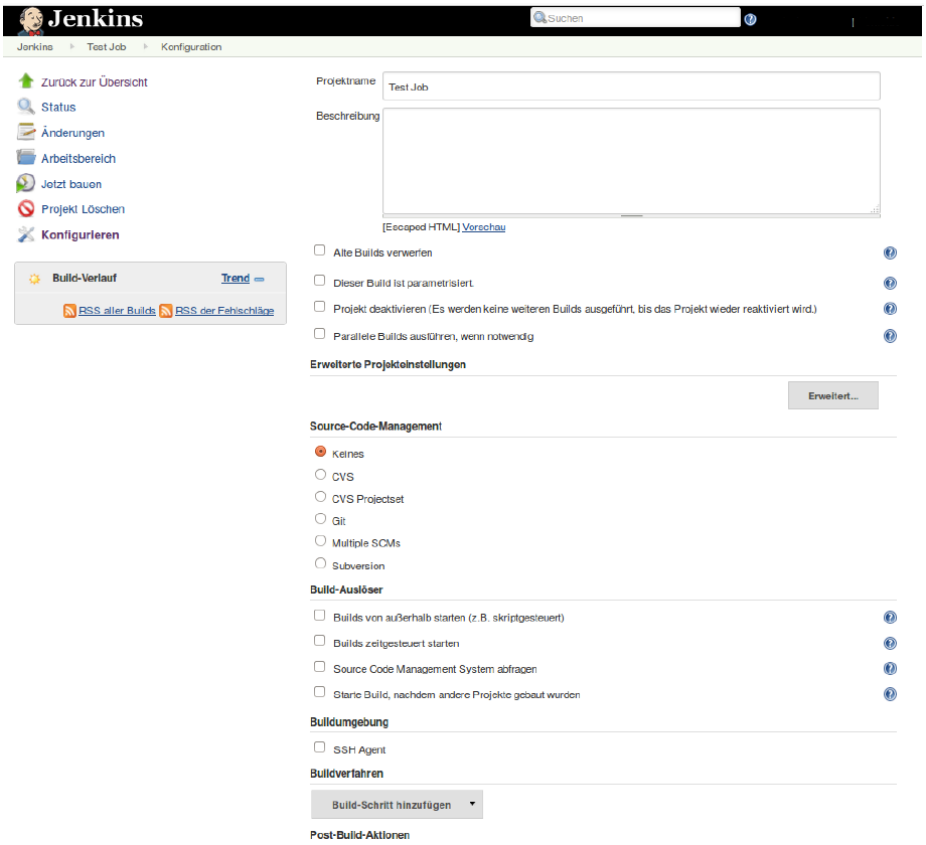

Abb. 3. Abb. Die Konfigurationsseite des Jenkins Integration-Servers. Sie zeigt einen Überblick über die Einstellmöglichkeiten eines Jenkins-Jobs.

Build-Auslöser. Durch Auswahl verschiedener Ereignisse, kann ein Job gestartet werden. Wiederkehrende Ereignisse können im Stil von Cron- Jobs festgelegt werden. Mittels eines Remote- Aufrufs ist es möglich API Jobs zu starten. Auch durchsucht und prüft Jenkins aktiv Source-Code-Management-Systeme nach Änderungen. Abhängigkeiten zwischen Jobs werden möglicherweise definiert, wodurch bestimmte Jobs dann gebaut werden, wenn andere erfolgreich waren.

Buildverfahren. Dieser Abschnitt legt fest, was im Build-Schritt gemacht werden soll. Skripte wie Shell oder Batch können in der Standardkonfiguration auszuführt werden. Dadurch hat man eine flexible Gestaltung der Ausführung der Jobs.Neben dem Aufrufen von einfachen Build-Befehlen können hier Tests gestartet werden, aber auch das Ausführen von Aufgaben anderer Art ist hier möglich.

Post-Build-Aktionen. In diesem Schritt können Aufgaben, die den Buildprozess abschließen, durchgeführt werden. Zum Beispiel können das Zusammenfassungen der

Compiler-Warnungen oder Testergebnissen sein. Die Erstellung von Dokumentationen ist eine weitere Möglichkeit, Benachrichtigungen an Entwickler zu schicken oder Artefakte zu speichern, die beim Buildprozess entstanden sind. [14]

3 Best Practices.

Sicherheit hat oberste Priorität. Es ist von großer Wichtigkeit, zuallererst die Zugriffsrechte entsprechend einzustellen, damit nicht jeder Anwender auf eine neu installierte Jenkins-Instanz zugreifen kann und Jenkins konfiguriert, Jobs erstellt/editiert/löscht oder Builds durchführt et cetera.

Versionsverwaltung für saubere Builds verwenden. Es wird empfohlen ein Versionsverwaltungsystem (z.B. Git) mit Jenkins einzusetzen.

Auch wenn der Einsatz eines Versionsverwaltungssystems (z.B. Git) mit Jenkins nicht zwingend ist, empfehlen wir ihn trotzdem. Denn der Source Code, welcher für saubere Builds verwendet wird, sollte immer von einem Versionsverwaltungssystem kommen. Nur so lässt sich ein durchgeführter Build reproduzieren. In der Praxis hat es sich bewährt, dass der komplette Code (auch Build-Skripte, Release-Notes etc.) in VCS (Version Control System) eingecheckt werden.

Zusammenarbeit mit Ticketsystem gewährleisten. Wenn möglich, sollte man dafür Sorge tragen, dass das verwendete Ticketsystem (z.B. Redmine oder Jira) mit Jenkins zusammen arbeitet um durchgeführte Anpassungen zentraler zu protokollieren. Für Redmine beispielsweise gibt es Plugins, die Redmine in Jenkins integrieren und umgekehrt genauso. [15]

7.2 Docker meets Jenkins

Eine automatisierte Bereitstellung von Applikationen, die in einem Container organisiert sind, wird dank Docker, einem Open-Source-Projekt, ermöglicht. Es nutzt hierzu die Eigenschaften des Linux-Kernel: Eine Isolation, ohne den Start einer einzigen virtuellen Maschine, von Prozessor, RAM, Netzwerk oder Block-Speicher. Weiterhin können Applikationen vollständig von der jeweiligen Umgebung inklusive der Prozesse, Dateisysteme oder des Netzwerks getrennt und damit autonom betrieben werden. Ein autonomes Verschieben von Applikationen über Systeme hinweg, können mit dem Aufheben von externen Abhängigkeiten ermöglicht werden. Docker kapselt dafür die eigentliche Anwendung und ihre notwendigen Abhängigkeiten wie Bibliotheken in einen virtuellen Container, welcher dann auf jedem beliebigen Linux- und Windows-System ausführbar ist. Dadurch wird die Flexibilität und der Portabilitätsgrad erhöht. Während des Betriebs einer Applikation auf einer virtuellen Maschine wird neben der Applikation selbst eine Reihe weiterer Ressourcen benötigt, was zu einem enormen Overhead und großen Abhängigkeiten führt. Dazu gehören die notwendigen Bibliotheken, ein vollwertiges Betriebssystem und gegebenenfalls weitere Dienste und

Applikationen. Ein Docker-Container hingegen umfasst nur die eigentliche Applikation sowie die dazugehörigen Abhängigkeiten. Dieser wird als isolierter Prozess auf dem Host-Betriebssystem ausgeführt und teilt sich den Linux Kernel mit anderen Docker-Containern. Durch die strikte Isolation können mehrere Container auf dieselben Kernel-Ressourcen zugreifen. Für jeden Container lässt sich dabei exakt definieren, wie viele Ressourcen (Prozessor, RAM, Bandbreite) ihm zur Verfügung stehen. [16]

Ein einfach und flexibel gestaltetes Handling von Build-, Test- und auch Produktions-Umgebungen, durch benutzung von Docker-Containern ist heutzutage sehr oft zu sehen. Der Grund dafür ist ein Trend der nach der Einführung von Continuous Delivery im Kontext DevOps entstand.

Docker-Image. Ein Docker-Image dient als Read-only-Template für Container. Es kann per "docker build" aus einem Dockerfile erstellt werden oder aus einem Repository bezogen werden. Sie haben die eigenschaft "stapelbar" zu sein, das heißt, das sie erben und erweitert werden können. Die ist ein Unterschied zu den VM-Images.

Docker-Container. Aus einem Docker-Image werden per "docker run" lauffähige Docker-Container instanziiert und gestartet, welche die Laufzeitumgebung und die Anwendung betreiben.

Repository. Docker-Images werden zwar in einem lokalen Repository gespeichert, doch sie können in ein Remote-Repository (z.B. Docker Hub Registry) kopiert werden und somit auch anderen zur Verfügung gestellt werden. [17]

Referenzen

1. entwickler.de, Torsten Kühn, Eine Einführung in die Continuous Integration mit Jenkins, https://entwickler.de/online/eine-einfuehrung-in-die-continuous-integration-mit-jenkins-158667.html
2. heise.de, Björn Feustel und Steffen Schluff Continuous Integration in Zeiten agiler Programmierung
   https://www.heise.de/developer/artikel/Continuous-Integration-in-Zeiten-agiler-Programmierung-1427092.html
3. automatic.com, Ron Gidron, Was ist Continuous Integration? Und was ist es nicht?
   https://automic.com/de/blog/was-ist-continuous-integration-und-was-ist-es-nicht
4. dev-insider.de, Thomas Joos und Stephan Augsten, Continuous Integration und Continuous Delivery,
   https://www.dev-insider.de/continuous-integration-und-continuous-delivery-a-618454/
5. Nikhil Patania, Pro Continuous Delivery with Jenkins 2.0, Apress India 2017, Chapter 1: Elements of Continuous Delivery
6. torsten-horn.de, Torsten Horn, Aachen 2015,
   http://www.torsten-horn.de/techdocs/ContinuousDelivery.html

7. infos.seibert-media.net, Video: Continuous Deployment – Schneller entwicklen, https://infos.seibert-media.net/display/Productivity/Continuous+Deployment-Schneller+entwickeln
8. http://www.scaledagileframework.com/continuous-deployment/
9. devops.com, Contributor, Continuous Delivery vs. Continuous Deployment, https://devops.com/continuous-delivery-vs-continuous-deployment/
10. http://www.itwissen.info/Versionskontrolle-version-control-VCS.html
11. https://subversion.apache.org/
12. andreas-schrade.de, Andreas Schrade Schnelle Einführung in Git, http://www.andreas-schrade.de/2015/03/11/git-kompakte-einfuehrung/
13. Rudolf Zimmermann, Artur Klos, Institut für SoftwareArchitektur. Technische Hochschule Mittelhessen, Campus Giessen, Kurzanleitung Git
14. Jenkins als CI Werkzeug, http://home.edvsz.fh-osnabrueck.de/skleuker/CSI/Werkzeuge/Jenkins/
15. Jenkins, Claudia Meindl, https://alphanodes.com/de/jenkins
16. t3n.de, Rene Büst, Docker: Was du über die Container-Technologie wissen musst, http://t3n.de/magazin/ueber-container-technologie-wissen-musst-docker-gehts-240047/
17. jaxenter.de, Tobias Gesellchen, Docker-Container als flexible Buildumgebung, https://jaxenter.de/docker-container-als-flexible-buildumgebung-237