Automatisch ans Ziel – so oft Sie wollen

Einführung in Continuous Delivery

Marcel Birkner

Continuous Delivery beschreibt die vollständige Automatisierung des Softwareentwicklungsprozesses und verkürzt damit die Lieferzeiten bei gleichzeitig verbesserter Qualität. Dieser Beitrag führt in das Thema ein und zeigt auf, wie Continuous Delivery sich in Unternehmen umsetzen lässt.

Was ist Continuous Delivery?

Continuous Delivery beschreibt die vollständige Automatisierung des Softwareentwicklungsprozesses. Dabei werden alle Schritte, von der Planung, über die Entwicklung, bis hin zum Release neuer Anforderungen, gesamtheitlich betrachtet. Um dies zu erreichen, kombiniert Continuous Delivery drei Kerndisziplinen:

- *Continuous Integration* (CI) hat in den letzten Jahren Einzug in viele Entwicklungsabteilungen gehalten. Als Teil des Entwicklungsprozesses kümmert sich CI um das Bauen der Software-Artefakte. Zusätzlich beinhaltet CI auch die Versionskontrolle aller Artefakte und Auflösung aller Abhängigkeiten (Dependency-Management).
- *Continuous Testing* ist ein weiterer Schritt, der durch bessere Integration mit CI-Servern wie Jenkins/Hudson, CruiseControl und Bamboo folgte. Dabei werden Unit-, Akzeptanz- und Performance-Tests nach jedem Build ausgeführt und die Ergebnisse im CI-Server automatisch ausgewertet und aufbereitet.
- *Continuous Deployment* wurde in den letzten Jahren immer mehr zum Flaschenhals für Software-Releases, da Entwicklungs- und Betriebsteams in den meisten Unternehmen, aus historischen Gründen, getrennt agieren. Die DevOps-Bewegung [Hütt13] hat sich dieser Problematik angenommen und versucht, eine Brücke zwischen den Lagern zu schlagen.

Welche Ziele werden mit Continuous Delivery verfolgt?

Continuous Delivery vereint die Interessen von Fachbereichen, Entwicklung, Testteam und Betrieb. Fachbereichen wird Raum für Innovationen gegeben, sodass sie auch neue Ideen leicht ausprobieren können. Somit wird auch ein direktes Kunden-Feedback ermöglicht. Den Entwicklern und Testern wird die Angst vor dem nächsten Release genommen, da Software-Releases mit einheitlichen Skripten und Prozessen automatisiert in allen Umgebungen durchgeführt werden. Zudem bekommen die Test-/QA-Teams fortwährend aktuelle Testumgebungen. Dadurch kann eine sehr hohe Qualität des Softwaresystems garantiert werden. Der Betrieb gewinnt durch Continuous Delivery mehr an Stabilität und Kontrolle, trotz vermehrter Releases. Die Angst vor Anrufen in der Nacht und Systemausfällen wird dem Betrieb genommen.

Die wesentlichen Vorteile von Continuous Delivery sind:

- Software-Releases mit geringerem Risiko,
- fachliche Innovationen kommen schneller auf den Markt,
- Verbesserung des „Return on Invest" durch häufigere Releases,
- Verbesserung der Konkurrenzfähigkeit und Reaktionszeit,
- Verbesserung der Qualität neuer Softwareversionen,
- Bestimmung der Release-Planung durch die Fachbereiche und nicht durch die IT,
- Software ist immer bereit für ein Produktions-Release,
- vollständige Automatisierung des Release-Prozesses,
- keine oder sehr kurze Downtime.

Ohne Continuous Delivery schaffen viele Unternehmen nur drei bis vier große Releases im Jahr. Das Release eines neuen Features erfolgt dadurch erst mehrere Monate nach der Programmierung. Zudem müssen aufwendige Abnahmetests durchgeführt werden, bei denen immer alle Features und Bugfixes seit dem letzten Release getestet werden, egal ob sie schon einmal getestet und abgenommen wurden. Des Weiteren ist die nachträgliche Fehlersuche bei Software-Releases mit vielen Änderungen aufwendiger.

Der Release-Prozess bei Continuous Delivery löst viele dieser Probleme. Durch die vollständige Automatisierung des Release-Prozesses und durch kleinere Release-Pakete können Änderungen in sehr kurzen Intervallen ausgeliefert werden (z. B. täglich/wöchentlich). Sollten dann Fehler in der Produktion auftreten, beschränkt sich die Fehlersuche auf eine kleinere Menge an Quellcodeänderungen. Die Kosten der Fehlersuche reduzieren sich dadurch um ein Vielfaches. Features, die schneller in die Produktion gelangen, tragen frühzeitiger zum Wertbeitrag der Software bei. Die Reduzierung der Liegezeit von Features resultiert daher direkt in einer Steigerung des Umsatzes, der durch die Software generiert wird.

Was sind organisatorische Voraussetzungen?

Um Continuous Delivery in einem Unternehmen einführen zu können, müssen gewisse Voraussetzungen erfüllt sein. Alle an der Wertschöpfung der Software Beteiligten sind einzubinden. Crossfunktionale Teams bestehen in der Regel aus Mitarbeitern aus dem Fachbereich, Entwicklern, Testern und dem Betrieb. Dadurch wird gewährleistet, dass bei Unstimmigkeiten immer eine enge Abstimmung innerhalb des Teams möglich ist. Ohne die Unterstützung der Fachbereiche bei der Abnahme gewünschter Features können keine häufigeren Releases durchgeführt werden. Ohne die enge Zusammenarbeit mit dem Betrieb, ist es auch nicht möglich, den Release-Prozess zu automatisieren und Änderungen an der Systemlandschaft durchzuführen.

Dieser Herausforderung nimmt sich die DevOps-Bewegung seit mehreren Jahren an. Für viele Firmen bedeutet dies im ers-

ten Schritt eine Überwindung von organisatorischen Grenzen. Firmen mit Scrum- und Kanban-Teams dürften viele dieser Voraussetzungen schon umgesetzt haben.

Was sind kulturelle Voraussetzungen?

Oft stehen Zielvereinbarungen im Weg und sorgen für Interessenkonflikte zwischen den unterschiedlichen Abteilungen. In den Fachbereichen werden Bonuszahlungen nach generierten Umsätzen ausgezahlt und nach geschaffenen Innovationen. Ein Fachbereichsmitarbeiter möchte daher viele seiner Ideen, die zur Steigerung des Umsatzes beitragen, so schnell wie möglich im nächsten Release sehen. Entwickler wollen an technischen Innovationen arbeiten und werden daran gemessen, wie gut sie diese umsetzen. Die Fachleute im IT-Betrieb werden dafür bezahlt, dass die Software reibungslos läuft und keine Störungen auftreten. Jedes große manuelle Release birgt die Gefahr einer Störung und aufwendiger und langwieriger Fehlersuche. Nur, wenn alle Beteiligten die Vorteile von Continuous Delivery erkennen, sind die Voraussetzungen für eine Einführung geschaffen.

Was sind die technischen Voraussetzungen?

Von allen Teammitgliedern wird ein hohes technisches und methodisches Wissen verlangt, das für eine erfolgreiche Umsetzung notwendig ist. Diese Fertigkeiten lassen sich, wie in Abbildung 1 zu sehen, in die Bereiche Handwerkszeug (Analyse, Architektur, Code) und Automatisierung (Build, Test, Deployment) einteilen.

Analyse

Bevor eine neue Anforderung vom Entwicklungsteam in einem Scrum-Sprint umgesetzt werden kann, muss diese die „Definition of Ready" erfüllen. Eine wichtige Voraussetzung ist die vollständige Beschreibung aller Anforderungen in Form von User Stories. Diese müssen in einem Backlog priorisiert und gepflegt werden, das sogenannte „Backlog grooming". Abhängigkeiten zu anderen User Stories müssen ebenfalls dokumentiert werden, damit Stories in der richtigen Reihenfolge umgesetzt werden können. Für die Entwicklung von grafischen Oberflächen ist es zudem hilfreich, wenn mit den Fachbereichen Präsentationsprototypen (Mock-ups) erstellt werden, die dem Entwickler bei der Programmierung der Benutzeroberfläche helfen.

Architektur

Die Architektur eines Softwaresystems sollte immer nur in Verbindung mit neuen Anforderungen entstehen. Die Funktionalität validiert somit gleichzeitig die Architekturentscheidungen. Architektur sollte nie für sich alleine entstehen. Sie ist nie Selbstzweck. Während des Projektverlaufs entwickelt sich die Architektur somit zu einem Gesamtbild, das alle Anforderungen perfekt unterstützt.

Für Projekte, die auf der „grünen Wiese" entstehen, ist dieser Ansatz leichter umzusetzen. Bei bestehenden Projekten und Legacy-Anwendungen bedarf es intensiver Auseinandersetzungen mit der Systemlandschaft und den Entwicklungsprozessen, bevor die Anwendung für Continuous Delivery bereit ist. Softwaremodule müssen eventuell restrukturiert und neu geschnitten werden, komplexe manuelle Deployment-Prozesse müssen automatisiert, Teamstrukturen aufgebrochen und Aufgaben eventuell neu verteilt werden. Trotzdem lohnt sich der Aufwand, da sich die Vorteile im weiteren Lebenszyklus der Anwendung auszahlen werden.

Nicht nur die Architektur, sondern auch die Dokumentation sollte während des Projektverlaufs entstehen. Zu Beginn des Projektes müssen alle Stakeholder ihre Wünsche an eine ausreichende Dokumentation formulieren und diese dann auch vom Team einfordern. Die Dokumentation sollte immer, in Verbindung mit neuen User Stories, vom Team aktualisiert werden. Dabei haben sich Wiki-Systeme für die Dokumentation als sehr nützlich erwiesen. Das Team kann gemeinsam an der Dokumentation arbeiten, ohne sich gegenseitig zu blockieren. Zum Release-Zeitpunkt kann aus den Wikiseiten automatisch ein Word- oder PDF-Dokument generiert werden, sollte dies notwendig sein. Nach Projektende können alle weiteren Änderungen an der Software im Wiki dokumentiert werden. Die folgenden Dokumente sollten für die meisten Projekte ausreichen:

- Betriebsleitfaden (Betrieb),
- Administrator-Guide (Fachlicher Administrator),
- Schnittstellen-Guide (Externe/Interne Partner/Entwickler),
- Architektur-Guide (Entwickler).

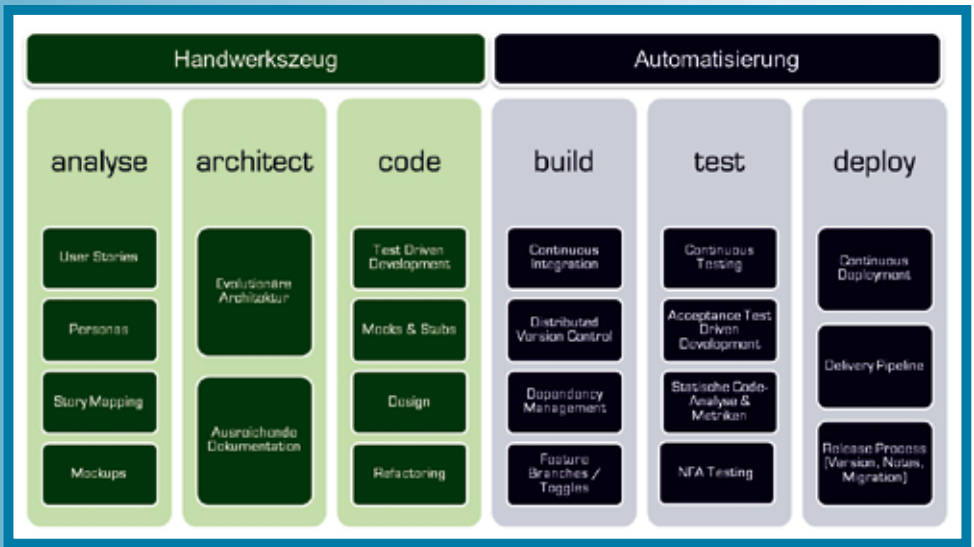


Abb. 1: Technische Voraussetzungen für Continuous Delivery

Code

Für Continuous Delivery ist es sehr wichtig, dass alles in ein Versionskontrollsystem (VCS) gehört. Nur wenn der Quellcode, die Build-Skripte, Datenbankskripte und sämtliche Konfigurationen der Systemlandschaft in der Versionskontrolle gehalten werden (Infrastructure-as-Code), ist ein vollautomatischer und reproduzierbarer Release-Prozess realisierbar. Dass der Quellcode im VCS gepflegt wird, ist seit Langem üblich. Dass auch Datenbankskripte, Anwendungs- und Serverkonfigurationen im VCS gespeichert werden, ist bisher nicht weit verbreitet. Dies hat jedoch einen entscheidenden Vorteil. Nur wenn alle Skripte versioniert im VCS

gespeichert und für die Deployments verwendet werden, wird eine schnelle Fehlersuche ermöglicht. Sollten nach einem Release Fehler mit der Konfiguration eines Servers auftreten, kann sofort in der VCS-Historie nachgeschaut werden, welche Skripte geändert wurden und von wem. Somit ist auch immer ein Rollback zu einem funktionsfähigen Stand möglich.

Für das Konfigurationsmanagement von Serverlandschaften empfehlen sich Skriptsprachen wie Puppet, CFEngine und Chef. Dabei wird die Konfiguration von Systemen mit einer deklarativen Sprache beschrieben und die so entstehenden Spezifikationen werden an zentraler Stelle gehalten. Speziell im Kontext von Continuous Delivery sind Konfigurationsmanagement-Tools unerlässlich. Rollt man permanent neue Features aus, muss auch der Prozess zur Erweiterung und Anpassung der Infrastruktur und deren Konfiguration möglichst vollständig automatisiert werden. Die Verwendung von Konfigurationsmanagement-Tools fördert in diesem Bereich die Transparenz zwischen Entwicklung und Betrieb.

Für die Versionierung von Datenbanken empfehlen sich Tools wie Liquibase und DbMaintain. Schema-Änderungen werden dabei in SQL- oder XML-Dateien mit einer Version gespeichert. Somit können die Tools auf den unterschiedlichen Umgebungen immer die aktuell vorhandene Version der Datenbank mit der aktuellsten Version aus dem VCS abgleichen. Für komplexere Datenmigrationen empfiehlt es sich, eigene DB-Updater zu schreiben, die effektiver große und komplexe Datenmengen migrieren können.

Build

Für den Build einer Softwareanwendung sind moderne Build-Tools wie Maven und Gradle zu empfehlen. Abhängigkeiten zu anderen Bibliotheken und Frameworks werden in der Build-Konfiguration beschrieben. Zudem existiert eine Vielzahl nützlicher Plug-ins, die für die Umsetzung von Continuous Delivery notwendig sind (Versionierung, Release-Management, Test, Quellcodeanalyse, Repository-Integration).

Alle gebauten Artefakte müssen versioniert in ein Artefakt-Repository ausgeliefert werden. Nexus und Artifactory sind zwei populäre Open-Source-Artefakt-Repositories für die Verwaltung verschiedenster Software-Artefakte. Beide bieten die gleiche Basisfunktionalität, die es erlaubt, versionierte Software-Artefakte zu speichern, beispielsweise via Maven, Gradle, Ant, Ivy oder manuell über eine Benutzeroberfläche. Zudem kann ein Artefakt-Repository auch als Proxy und Cache für andere Repositories im Internet dienen (z. B. Maven Central, Apache, Codehaus). Dadurch müssen Artefakte nicht von allen Entwicklern oder CI-Servern erneut aus dem Internet geladen werden. Zudem verringert sich die Abhängigkeit zu externen Repositories, falls diese durch Wartung oder technische Probleme nicht erreichbar sein sollten.

Ein weiterer wichtiger Faktor ist die Kontrolle über den Zugriff und die Verwendung von Software-Artefakten. Durch ein zentrales Repository kann kontrolliert werden, welche externen, teilweise auch sicherheitsrelevanten Artefakte verwendet werden dürfen und wer Zugriff darauf erhält. Interne Artefakte können auch leichter zwischen verschiedenen Entwicklungsteams geteilt werden. Durch die Trennung von Release- und Snapshot-Repositories wird zusätzlich gewährleistet, dass nur getestete Bibliotheken im Release-Repository landen.

Test

Bei der Umsetzung von Continuous Delivery ist es zwingend erforderlich, alle Komponenten bei jedem neuen Build vollautomatisiert zu testen. Da jede Quellcodeänderung automatisch einen neuen Build erzeugt und somit auch ein neues Release-Artefakt, ist es wichtig, dass dieses Artefakt vollständig getestet wird. Dies gewährleistet einen hohen Grad an Sicherheit. Dazu gibt es eine Reihe von Tools und Frameworks, die in einer Continuous-Delivery-Pipeline eingesetzt werden können. Generell lassen sich diese Tests in folgende Kategorien unterteilen: *Unit-Tests*, *Akzeptanz-Tests*, *Smoke-Tests* und *Last-/Performance-Tests*.

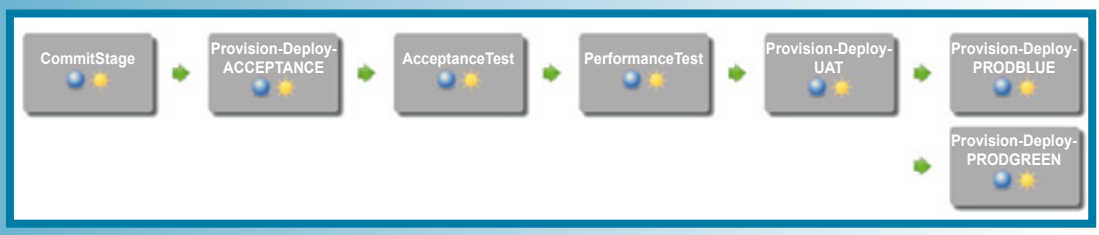
Die Umsetzung der Tests sollte nach dem Prinzip „Fail-Fast" erfolgen. Nach jedem neuen Quellcode-Check-in wird der gesamte Code kompiliert und es werden alle Tests ausgeführt. Sollte die Build-Pipeline erfolgreich durchlaufen, wird sichergestellt, dass die fertigen Software-Artefakte vollständig getestet wurden. Dieser Vorgang läuft vollautomatisch auf einem CI-Server.

Folgendes Vorgehen hat sich in der Vergangenheit bewährt:

- Nachdem die Software gebaut wurde, werden die *Unit-Tests* ausgeführt. Sollte ein Softwaremodul Fehler aufweisen, bricht der Build sofort ab und meldet einen Fehler an das Entwicklungsteam. Für Unit-Tests haben sich JUnit und TestNG etabliert.
- Im zweiten Schritt wird die Anwendung auf einer Akzeptanz-Testumgebung ausgeliefert. Für *Smoke-Tests* bieten sich Scripting-Sprachen wie Perl, Shell, Groovy und Ruby an, die prüfen, ob die Anwendung erfolgreich deployt wurde.
- Im nächsten Schritt werden alle *Akzeptanz-Tests* ausgeführt, die während der Anforderungsanalyse mit Hilfe des jeweiligen Fachbereichs dokumentiert und später vom Entwickler umgesetzt wurden. Cucumber, jBehave und das Robot-Framework sind weit verbreitete Akzeptanz-Test-Frameworks.
- Als letzter Testschritt sollte immer sichergestellt werden, dass neue Features die Performance einer Anwendung nicht negativ beeinträchtigen. Dazu werden auch *Last-/Performance-Tests* durchgeführt. Durch das Definieren von Schwellwerten können Probleme bei der Performance schnell erkannt und frühzeitig von den Entwicklern behoben werden.

Commit Stage	Provision Acceptance Stage	Acceptance Test	Performance Test	Provision Deployment UAT	Provision Deployment PROD
Unit-Tests JUnit TestNG	Smoke-Tests Shell Perl Ruby Groovy	Akzeptanz-Tests jBehave Fitness Selenium Robot Cucumber Concordion	Performance-Tests jMeter loadUI	Smoke-Tests Shell Perl Ruby Groovy Manuelle Tests	Smoke-Tests Shell Perl Ruby Groovy Manuelle Tests

Tabelle 1: Tests in einer Continuous-Delivery-Pipeline



In Tabelle 1 sehen sie eine Continuous-Delivery-Pipeline und die Stellen, an denen die verschiedenen Tests durchgeführt werden.

Deployment

Das Deployment besteht aus einer vollautomatisierten Deployment-Pipeline (s. Abb. 2). Sobald Änderungen in das VCS eingecheckt werden, startet der Durchlauf der *Commit Stage*. Zuerst wird der Quellcode gebaut und alle Unit-Tests werden ausgeführt. Danach wird eine statische Code-Analyse gestartet. Nur, wenn keine Qualitätsregeln verletzt werden, wird der Code getaggt und das gebaute Artefakt versioniert ins Artefakt-Repository ausgeliefert.

Im nächsten Schritt wird eine Testumgebung *provisioniert* und die aktuelle Anwendung *ausgeliefert*. Mit Smoke-Tests wird sichergestellt, dass dieser Schritt erfolgreich verlaufen ist.

In der *Akzeptanztest-Phase* werden alle Akzeptanz-Tests auf der frisch provisionierten Testumgebung ausgeführt. Wenn diese erfolgreich durchgelaufen sind, ist sichergestellt, dass alle fachlichen Anforderungen erfolgreich umgesetzt wurden. Im nächsten Schritt werden *Last-/Performance-Tests* auf der Testumgebung ausgeführt.

Sollten auch hier alle Grenzwerte eingehalten werden (z. B. Antwortzeiten) und sich das Verhalten der Anwendung unter Last nicht verschlechtert haben, wird die *User-Akzeptanztest-umgebung* (UAT) provisioniert. Auf dieser Umgebung kann der Fachbereich die letzten Änderungen noch einmal manuell gegenprüfen, bevor er das Release für ein Deployment in die Produktion freigibt. Sollte ein sofortiges Release gewünscht sein, so kann dies über das *Blue/Green-Konzept* mit einem Knopfdruck umgesetzt werden. In diesem Fall existieren zwei identische Produktivumgebungen:

- Eine ist das aktuelle Livesystem,
- die zweite ist im Hot-Standby und kann für neue Deployments verwendet werden.

Sobald die neue Version einer Anwendung auf der Hot-Standby-Produktivumgebung erfolgreich ausgeliefert wurde, kann durch einfaches Umschalten des Loadbalancers ein Teil der eingehenden Requests auf die zweite Produktivumgebung geleitet werden. Sollten keine Probleme auftreten, werden alle Requests in die zweite Umgebung geleitet, und das ehemals aktive Produktivsystem wechselt in den Standby-Modus.

Sollte während der Deployments ein Fehler auftreten, kann jederzeit ein Rollback der Umgebung auf eine der vorherigen funktionierenden Versionen durchgeführt werden.

Fazit

Continuous Delivery automatisiert und beschleunigt alle für die Softwareentwicklung notwendigen Prozesse. Die Lieferzeiten werden verkürzt, potenzielle Fehlerquellen reduziert und die Anzahl der Releases steigt. Dadurch ist eine bessere Kundenorientierung möglich und die Steigerung der Innovationsfähigkeit Ihrer IT-Systeme. Um dies in einem Unternehmen umzusetzen, sind organisatorische, kulturelle und technische Voraussetzungen notwendig. Zusammenfassend bietet Ihnen Continous Delivery eine höhere Wertschöpfung der IT, eine bessere Release-Qualität, reduzierte Entwicklungs-/Wartungskosten und eine höhere Kundenzufriedenheit. So bringen Sie Innovationen schneller voran und Deployments werden zur Routine.

Wenn ich Ihr Interesse für das Thema geweckt habe, dann können Sie sich auf die nächste Ausgabe von JavaSPEKTRUM freuen. Dann werden wir eine Continuous-Delivery-Pipeline, mit Hilfe von Open-Source-Tools, Schritt für Schritt aufbauen.

Links

[Artifactory] Artifactory Repository, `http://www.jfrog.com/`
[Bamboo] Atlassian Bamboo, `http://www.atlassian.com/software/bamboo/overview`
[Birk12] M. Birkner, Continuous Delivery in the Cloud – Part 1: Overview, `http://blog.codecentric.de/en/2012/04/continuous-delivery-in-the-cloud-part1-overview/`
[Chef] OPSCODE Chef, `http://www.opscode.com/chef/`
[Confluence] Atlassian Confluence, `http://www.atlassian.com/software/confluence/`
[Cucumber] `http://cukes.info/`
[DbMaintain] `http://dbmaintain.sourceforge.net/`
[DevOps] DevOps-Bewegung, `http://en.wikipedia.org/wiki/DevOps`
[DevOps Days] `http://www.devopsdays.org`
[Gradle] Gradle Build Tool, `http://www.gradle.org/`
[Hütt13] M. Hüttermann, Agiler Baukasten für DevOps, in: JavaSPEKTRUM, 1/2013
[jBehave] `http://jbehave.org`
[JenkinsCI] Jenkins CI Server, `http://jenkins-ci.org/`
[JUnit] `https://github.com/KentBeck/junit`
[Liquibase] `http://www.liquibase.org/`
[Maven] Maven Build Tool, `http://maven.apache.org/`
[Nexus] Nexus Repository, `http://www.sonatype.org/nexus/`
[Puppet] `http://www.puppetlabs.com`
[Robot] Robot Framework, `http://code.google.com/p/robotframework/`
[Sonar] `http://www.sonatype.com/`

Marcel Birkner ist Software Consultant bei der codecentric AG. Sein Fokus liegt auf Java, Spring, Cloud-Technologien und allem rund um die Automatisierung von Softwareentwicklungsprozessen. E-Mail: marcel.birkner@codecentric.de