Nicht ohne Prophylaxe

Continuous Delivery – Welche Anforderungen muss meine Anwendung erfüllen?

Marcel Birkner

Continuous Delivery beschreibt die vollständige Automatisierung des Softwareentwicklungsprozesses und verkürzt damit die Lieferzeiten bei gleichzeitig verbesserter Qualität. Dieser Beitrag zeigt, wie Entwickler ihre Anwendungen fit für Continuous Delivery machen können.

Im ersten Artikel dieser Serie [Birk13a] haben wir die Grundlagen und Vorteile von Continuous Delivery vorgestellt. Des Weiteren haben wir die notwendigen organisatorischen und kulturellen Voraussetzungen betrachtet, um Continuous Delivery erfolgreich in einem Unternehmen umzusetzen. Im zweiten Artikel [Birk13b] wurde eine vollständige Continuous-Delivery-Pipeline modelliert und mit Hilfe von Open-Source-Werkzeugen umgesetzt. In diesem Artikel werde ich Ihnen zeigen, welche Dinge man bei der Entwicklung von Anwendungen beachten sollte, um sie fit für Continuous Delivery zu machen. Dabei versuche ich, das Thema nicht nur aus Sicht von neuen „Grüne Wiese"-Projekten zu beschreiben, sondern auch auf Probleme von älteren Anwendungen einzugehen.

Automatisierter Build

Eine der wichtigsten Voraussetzungen für die Umsetzung von Continuous Delivery ist die Automatisierung des Build-Prozesses. Die Anwendung muss sich ohne manuelle Schritte auf einem zentralen Build-Server bauen lassen. Dabei helfen Standard-Build-Tools wie Maven [Maven] oder Gradle [Gradle]. Das Ziel sollte es sein, mit einem Skript aus dem Quellcode ein fertiges Build-Artefakt zu generieren.

Entwicklerteams, die bei der Anwendungsentwicklung Standard-Werkzeuge und -Methoden einsetzen, werden es in der Regel leichter haben, ihre Anwendung auch in Zukunft problemlos weiter zu entwickeln. Das Layout ihrer Java-Anwendungen sollte den Maven-Projektstruktur-Konventionen entsprechen. Diese haben sich seit Jahren bewährt und werden auch von neuen Build-Werkzeugen wie Gradle verwendet. Der Vorteil von *Convention over Configuration* zeigt sich sehr schnell im Konfigurationsaufwand der Maven- und Gradle-Plug-ins. In vielen Fällen reicht die Standard-Konfiguration aus, da die Plug-ins genau wissen, in welchen Ordnern sich der produktive Quellcode, die Testklassen und die Konfigurationsdateien befinden. Durch die standardmäßige Trennung des Produktiv- und Test-Codes in Maven wird automatisch gewährleistet, dass Test-Code nicht mit in die Produktion gelangt.

Viele ältere Anwendungen werden noch mit Ant gebaut und haben ihre eigene Projektstruktur. Meist wurde diese in einem Projekt vorgegeben und dann für alle weiteren Projekte im Unternehmen kopiert. Die Erzeugung der Klassen und das Erstellen der Software-Artefakte (JAR-, WAR- oder EAR-Dateien) wird dann meistens von verschiedenen Ant-Tasks erledigt. Beim Einsatz von Standard-Plug-ins müssen immer wieder eine Reihe von Einstellungen gesetzt werden. Bei unachtsamen Entwicklern kann es auch vorkommen, dass Test-Code und -Bibliotheken mit in die Produktion gelangen [Ant].

Ein weiterer Vorteil von modernen Build-Werkzeugen ist die gute Unterstützung in integrierten Entwicklungsumgebungen (Eclipse, IntelliJ, NetBeans). Mit wenigen Schritten kann ein Projekt importiert werden, ohne nachträglichen Konfigurationsaufwand für die Entwickler zu erfordern. Neue Entwickler im Team finden sich in der Regel schneller im Quellcode zurecht, wenn das Maven-Projekt-Layout (s. Abb. 1) verwendet wird.

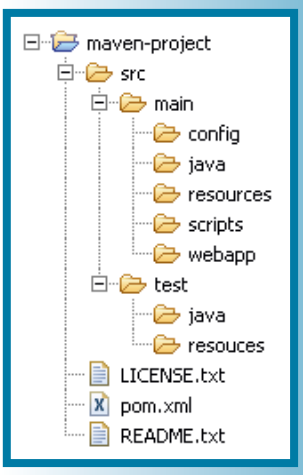

Abb. 1: Maven-Projekt-Layout

Wenn Sie noch ältere Build-Werkzeuge wie Apache Ant verwenden, sollten Sie sich die Mühe machen und einen Wechsel zu Apache Maven oder Gradle evaluieren. Je nach Größe und Komplexität Ihrer Anwendungen lohnt es sich, den Build-Prozess mit modernen Build-Werkzeugen zu vereinfachen. Auf längere Sicht wird sich der einmalige Aufwand rentieren, da Sie von einer Vielzahl an Open-Source-Werkzeugen und Plug-ins profitieren werden, die Ihren Entwicklungsprozess verbessern. Das Maven-Eclipse-Plug-in löst nicht nur alle Abhängigkeiten zu Third-Party-Bibliotheken auf, sondern lädt die Javadoc-Dokumentation und Sources-Bibliotheken herunter, sollten diese für die Bibliotheken vorhanden sein. Dadurch ist es für den Entwickler wesentlich einfacher, Third-Party-Bibliotheken korrekt zu verwenden und mögliche Fehler zu beheben. In Ant hingegen ist dies nur mit sehr viel manuellem Aufwand möglich, was in Maven mit einer Zeile erledigt ist:

```
mvn eclipse:eclipse -DdownloadSources=true -DdownloadJavadocs=true
```

Am Ende eines jeden Builds sollte automatisch eine statische Quellcode-Analyse mit dem Plug-in für SonarQube (ehemals Sonar) durchgeführt werden. Dadurch wird der Quellcode ständig auf Regelverletzungen und kritische Bugs geprüft. Sonar verwendet dabei bekannte Quellcode-Analysewerkzeuge wie FindBugs, PMD, Checkstyle und Macker [SonarQube]:

```
mvn sonar:sonar -Ddetail=true
```

Verwaltung von Abhängigkeiten

Abhängigkeiten zu Third-Party- oder eigenen Bibliotheken sollten nie fest mit der Anwendung eingecheckt sein, sondern im Build-Skript deklarativ beschrieben werden. Zum einen wird es so leichter, den Überblick über die verwendeten Versionen zu behalten. Zum anderen kann durch einfaches Hochsetzen der Version im Build-Skript eine aktuellere Version der Bibliothek verwendet werden. Bibliotheken existieren zudem nicht mehr dupliziert und eingecheckt in allen Anwendungen, sondern werden zentral in einem Artefakt-Repository gelagert und verwaltet, zum Beispiel [Nexus] oder [Artifactory].

Wenn eine Anwendung gebaut wird, lädt das Build-Werkzeug alle notwendigen Bibliotheken und deren transitiven Abhängigkeiten aus dem konfigurierten Artefakt-Repository herunter. Maven und Gradle verwenden von Hause aus Artefakt-Repositories für das Dependency-Management. Für Ant gibt es

mit IVY eine Erweiterung, die diese Aufgabe übernimmt. Artefakt-Repositories helfen dabei, einen genauen Überblick über alle verwendeten Artefakte zu behalten. Plug-ins verschaffen zudem einen Überblick über die verwendeten Lizenzen (Open Source vs. Closed Source) und bieten die Möglichkeit, Lizenzen zu überwachen. Sollten unerwünschte Lizenzen von einer Anwendung im Unternehmen eingesetzt werden (z. B. Copy-Left), wird eine E-Mail an eine Empfängerliste mit den wichtigsten Informationen gesendet [LicenseControl].

Mit der Professional Version von Nexus ist es sogar möglich, einen Health Check aller verwendeten Bibliotheken zu starten. Als Ergebnis erhält man einen Bericht mit einer Klassifizierung aller Bugs. Dadurch erhält man für alle Anwendungen einen guten Überblick, welche davon eine veraltete Bibliothek mit einer kritischen Sicherheitslücke verwendet [HealthCheck].

Umgebungsunabhängige Artefakte erstellen

Im nächsten Schritt sollten alle umgebungsspezifischen Konfigurationen einer Anwendung in eigene Konfigurationsdateien ausgelagert werden. Das Spring Framework bietet mit dem `PropertyPlaceholderConfigurer` eine sehr einfache Möglichkeit, dies umzusetzen:

```
<bean class=
"org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
  <property name="locations"
            value="classpath:application.properties" />
</bean>
```

Alle Properties aus der konfigurierten *.properties-Datei sind mit ${propertyname} innerhalb der Spring-Konfiguration verwendbar.

Durch das Herauslösen der umgebungsspezifischen Konfigurationen wird die Anwendung von außen konfigurierbar. Dadurch kann die identische JAR-, WAR- oder EAR-Datei auf allen Test- und Produktionsumgebungen eingesetzt werden. Für jede Zielumgebung sollte es eine eigene Konfigurationsdatei geben, die erst zum Deployment-Zeitpunkt mit ausgeliefert wird. Zum einen erspart dies unnötige Compile- und Testzyklen, da für alle Umgebungen nur ein Build-Artefakt erzeugt wird. Zum anderen ist gewährleistet, dass absolut identische Artefakte auf den unterschiedlichen Umgebungen deployt sind und Fehler vermieden werden.

Oft werden Anwendungen pro Zielumgebung mit den jeweiligen Konfigurationen von einem VCS-Tag gebaut. Dabei kann es dennoch zu schwer identifizierbaren Problemen kommen. Selbst wenn zwei Artefakte vom gleichen VCS-Tag gebaut wurden, kann es durch unterschiedliche Build-Zeiten zu unterschiedlichen Ergebnissen führen. Ein Entwickler könnte zum Beispiel noch schnell einen Bug-Fix eingecheckt und den VCS-Tag überschrieben haben (Moving a Tag). Schon kleinste Änderungen führen so zu einem unterschiedlichen MD5-Hash des gebauten Artefaktes und erhöhen den Aufwand einer späteren Fehlersuche. Alle Konfigurationsdateien müssen im Versionskontrollsystem (VCS) eingecheckt und versioniert werden. Dadurch können diese während des Deployment-Prozesses den jeweiligen Artefakt-Versionen zugeordnet werden.

Testabdeckung der Anwendung

Es gibt verschiedene Arten, eine Anwendung zu testen. Lisa Crispin und Janet Gregory beschreiben in ihrem Buch *Agile Testing: A Practical Guide for Testers and Agile Teams* [AgileTesting] verschiedene Arten von Tests und teilen diese in Testquadranten (s. Abb. 2) ein. Auf den ersten Blick sehen die verschiedenen Testmethoden nach viel Aufwand aus. Der Testquadrant zeigt meiner Meinung nach jedoch sehr schön die Probleme und Herausforderungen, die mit den jeweiligen Testmethoden gelöst werden können. Viele Anwendungen werden immer noch mit sehr hohem manuellem Aufwand getestet. Für einen Großteil der hier aufgelisteten Tests gibt es Open-Source-Frameworks, damit Tests voll automatisiert ablaufen können.

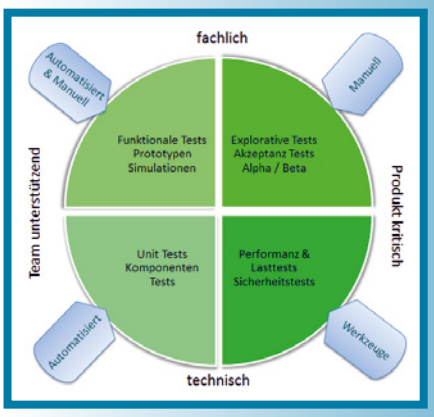

Abb. 2: Agile Testing – A Practical Guide for Testers and Agile Teams

Unit-Tests

Die schnellsten Erfolge im Bereich des automatischen Testens kann man mit Unit-Tests erzielen. Die zwei bekanntesten Unit-Test-Frameworks im Java-Umfeld sind jUnit und TestNG. Da ein Unit-Test immer nur die Funktionalität einer Klasse testet, werden Mocking-Frameworks wie Mockito, JMock oder EasyMock eingesetzt, um referenzierte Klassen aus dem Test zu entfernen. Das Ziel von Unit-Tests ist es, alle Klassen und Funktionalitäten einzeln zu testen. Unit-Tests sollten bei jedem CI-Build automatisch mit ausgeführt werden. Daher ist es wichtig, dass sie schnell durchlaufen und dem Entwickler ein schnelles Feedback liefern.

Zusammen mit Tools wie Cobertura wird automatisch ein Bericht erstellt, der aufzeigt, welche Zeilen des Quellcodes von einem Unit-Test abgedeckt sind. Ziel ist es, eine möglichst hohe Testabdeckung zu erreichen, um eine hohe Sicherheit bezüglich der Korrektheit der Anwendung zu bekommen. Haben Sie noch keine Unit-Tests in Ihrer Anwendung, ist es dennoch nie zu spät, damit anzufangen.

Unit-Tests werden oft erst im Nachhinein entwickelt. Wenn Bugs für eine Anwendung gemeldet werden, ist es sinnvoll, erst einen Unit-Test zu schreiben, der fehlschlägt und dadurch den Bug nachvollziehbar macht. Nachdem der Bug gefixt wurde, sollte auch der Unit-Test wieder grün sein. Dadurch wird sichergestellt, dass der gleiche Bug für diese getestete Klasse nie wieder auftreten kann.

Test-Driven Development (TDD) ist eine Methode für das Programmieren mit Hilfe von Unit-Tests. Dabei wird zuerst ein einfacher Unit-Test geschrieben und im zweiten Schritt die konkrete Implementierung, die dafür sorgt, dass der Unit-Test grün wird. Dies wird schrittweise wiederholt, bis die Funktionalität vollständig implementiert wurde. Durch die sehr gute Integration von Unit-Test-Frameworks in Entwicklungsumgebungen, wie zum Beispiel Eclipse, NetBeans und IntelliJ, bekommt der Entwickler ein sofortiges Feedback, ob seine Implementierung korrekt ist oder nicht.

Unit-Tests dienen Entwicklern auch zur Dokumentation, wie eine Klasse und deren Methoden zu verwenden sind. Zusätzlich geben sie Entwicklern mehr Sicherheit, wenn Teile der Anwendung refaktoriert werden müssen. Wenn alle Unit-Tests nach einem Refactoring erfolgreich durchlaufen, funktioniert die Anwendung mit sehr hoher Wahrscheinlichkeit weiterhin und die Angst vor größeren Refactorings wird dem Entwickler genommen.

Akzeptanztests

Akzeptanztests testen kritische Funktionalitäten und Prozesse einer Anwendung. Bei einem Web-Shop kann ein Akzeptanztest zum Beispiel den kompletten Einkaufsprozess eines Kunden nachstellen und verifizieren, dass alle ausgewählten Produkte im Warenkorb liegen und die Rechnungssumme korrekt berechnet wurde. Dies sind klassische Tests, die in langen Testbögen von Fachbereichen manuell bei jedem Release aufs Neue ausgeführt werden müssen. Je komplexer die Anwendung wird, desto länger und aufwendiger werden solche manuellen Tests und es können sich leicht Fehler einschleichen.

Abhilfe schaffen dabei Frameworks wie JBehave (Behaviour Driven Development) und das Robot-Framework (Keyword Driven Development). In JBehave werden Tests in Story-Dateien geschrieben. Dabei sollte eine JBehave-Story einer User-Story zugeordnet werden können. Dann wird ein Szenario in der Form "Given, When, Then" formuliert. Danach werden in einer Tabelle Testdaten hinterlegt, die beim Ausführen von JBehave verwendet werden sollen. Durch die einfache Lesbarkeit der Akzeptanz-Tests soll es auch Nicht-Entwicklern ermöglicht werden, die Tests zu verstehen und weitere Testdaten zu erstellen.

Im ersten Schritt wird eine Szenario-Story-Datei erstellt:

```
Given an empty Shopping Cart
When a 20 Euro book is added to the shopping cart
Then the shopping cart bill should be 20 Euro
```

Im zweiten Schritt wird der Szenario-Text mit Hilfe von Annotationen zu Java-Methoden gemappt:

```
@Given("an empty Shopping Cart")
public void emptyShoppingCart() {
  cart = new ShoppingCart();
}
@When(a $price Euro book is added to the shopping cart ")
public void addBookToCart(double price) {
  cart.addItemWith(price);
}
@Then("the shopping cart bill should be $sum Euro")
public void theShoppingCartBillShouldBe(double sum) {
  ensureThat(cart.getTotalBill(), equalTo(sum));
}
```

Zusätzlich muss JBehave für die Anwendung noch konfiguriert werden. Danach können die JBehave-Tests innerhalb der Entwicklungsumgebung ausgeführt werden oder vollständig automatisiert auf dem Build-Server. JBehave bietet eine sehr gute Unterstützung für JUnit, Ant, Maven, Eclipse und IntelliJ an. Die Ergebnisse werden in einem lesbaren Format in einer HTML-Datei abgelegt, die auch vom Fachbereich verwendet werden kann.

Last-/Performance-Tests

Ein weiterer kritischer Punkt für die Akzeptanz von Anwendungen ist das Testen von nicht-funktionalen Anforderungen, wie zum Beispiel Last- und Performance-Tests. Mit Apache jMeter können Testpläne über eine grafische Oberfläche erstellt werden, mit denen verschiedenste Komponenten einer Anwendung auf die Performance hin getestet werden. jMeter unterstützt dazu verschiedene Protokolle, die für die Aufrufe verwendet werden können, zum Beispiel HTTP, HTTPS, SOAP, IMAP, POP, JDBC, LDAP. Bei jedem Aufruf protokolliert jMeter die Anfrage-/Antwortzeiten. Zudem ist es möglich, mehrere jMeter-Clients koordiniert zu starten, damit eine größere Last auf den Test-Servern erzeugt wird. Sollten definierte Schwellenwerte erreicht werden oder Fehler bei den Anfragen auftreten, würde der Test fehlschlagen.

Sicherheits-Tests

Webanwendungen manuell auf Sicherheitslücken zu testen, kann sehr aufwendig sein, vor allem wenn man nicht alle aktuellen Angriffsszenarien kennt und nicht weiß, wo man anfangen soll. Daher gibt es vom Open Web Application Security Project (OWASP) den Zed Attack Proxy (ZAP). Dieser wird als Proxy zwischen die Webanwendung und den aufrufenden Browser/Client gesetzt und schneidet alle Requests mit. Im zweiten Schritt startet ZAP unterschiedliche Angriffsszenarien gegen die Anwendung, indem es die Parameter in den aufgezeichneten Requests manipuliert. Wenn man existierende jMeter- oder JBehave-Tests hat, kann man diese zum Aufzeichnen der Anwendungsaufrufe verwenden.

Beim Einsatz von ZAP sollte bedacht werden, dass die Analyse sehr lange dauern kann und es sinnvoll ist, die Analyse auf einer stabilen Umgebung durchzuführen. Zusätzlich bietet ZAP weitere manuelle Tools für die Sicherheitsanalyse von Webanwendungen.

Versionierung

Jeder Build erzeugt immer ein potenziell release-fähiges Artefakt. Daher muss die Versionierung automatisiert geschehen. Im CI-Server kann dazu die Build-Nummer des Build-Jobs verwendet werden. Diese wird, wie in Abbildung 3 zu sehen, als Parameter mit an das Maven-Build-Skript übergeben. Das Skript verwendet die Version für das erzeugte Artefakt und schreibt die Versionsnummer zusätzlich mit in die Manifest-Datei innerhalb des Artefaktes.

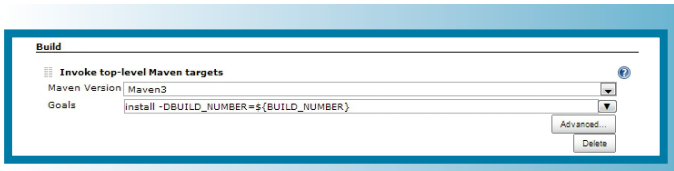

Abb. 3: Maven-Konfiguration in einem Maven-Build

Für interne Webanwendungen kann die Versionsnummer aus der Manifest-Datei ausgelesen und in der Anwendung dargestellt werden. Somit sehen Tester und Entwickler auf Anhieb, welche Version der Anwendung auf einer Test-/Staging- oder Produktionsumgebung deployt ist:

```
<groupId>de.codecentric</groupId>
<artifactId>sample-webapp</artifactId>
<version>${BUILD_NUMBER}</version>
<packaging>war</packaging>
...
<!-- Adding build information to MANIFEST -->
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-war-plugin</artifactId>
  <version>${war.plugin.version}</version>
  <configuration>
    <manifest>
      <addDefaultImplementationEntries>true
      </addDefaultImplementationEntries>
    </manifest>
    <archive>
      <manifestEntries>
        <Version>${BUILD_NUMBER}</Version>
      </manifestEntries>
    </archive>
  </configuration>
</plugin>
```

Für Anwendungen, die im Internet uneingeschränkt sichtbar sind, sollte man sich gut überlegen, ob die Version mit angezeigt wird.

Wenn eine WAR- oder EAR-Datei nur Webservices zur Verfügung stellt, kann es für Smoke-Tests hilfreich sein, wenn es

einen `GetVersion`-Aufruf gibt, der die Version aus der Manifest-Datei zurückliefert. Nach einem automatisierten Deployment kann so zum Beispiel via curl geprüft werden, ob die gewünschte Version des Webservice deployt wurde. Ein gutes Beispiel ist die REST-Programmierschnittstelle von Artifactory. Mit einem einfachen curl-Kommando bekommt man die Version, Revision und alle installierten Add-ons des Artifactory-Servers angezeigt.

Anfrage:

```
curl -sL
  --user username:password http://server/artifactory/api/system/version
```

Antwort (JSON):

```
{
  "version" : "2.6.1",
  "revision" : "13124",
  "addons" : [ "build", "ldap", "properties", "rest",
    "search", "sso", "watch", "webstart" ]
}
```

Automatisiertes Deployment

In vielen Unternehmen ist der Deployment-Prozess noch nicht vollständig automatisiert. Das hat oft mit der Trennung von Entwicklungs- und Betriebsteams zu tun. Das Deployment einer Anwendung sollte mit nur einem Skript vollständig automatisiert ablaufen. Dabei sollte das gleiche Skript für alle Umgebungen eingesetzt werden. Dadurch wird sichergestellt, dass Fehler im Deployment-Prozess auf den Testsystemen auffallen und nicht erst beim Produktions-Deployment. Alle umgebungsabhängigen Konfigurationen werden erst beim Deployment mit dem Software-Artefakt zusammen ausgeliefert.

Für die Automatisierung von Deployments gibt es verschiedene Möglichkeiten. Für Linux-Systeme könnte die Anwendung in einem speziellen Release-Job als RPM-Paket verpackt werden, welches über den yum-Paketmanager auf den jeweiligen Servern installiert wird. Für die automatisierte Provisionierung von Servern werden in der Regel Konfigurationsmanagement-Tools wie Puppet, Chef und CFEngine eingesetzt.

Bei Puppet werden das Setup und die Konfiguration von Systemen mit einer deklarativen Sprache beschrieben und die so entstehenden Spezifikationen an zentraler Stelle gehalten [Puppet]. Beteiligte Systeme führen im Hintergrund den sogenannten Puppet-Agenten aus, der sich kontinuierlich beim zentralen Puppet-Master erkundigt, ob für den jeweiligen Host neue Konfigurationen vorliegen. Im Falle neuer Daten übernimmt der Agent das Ausführen der übertragenen Spezifikation und aktualisiert das System und seine Konfiguration.

Kombiniert man Puppet mit einer verteilten Versionsverwaltung wie Git, erhält man leicht die Möglichkeit, auch System-Konfigurationen zuerst in abgeschotteten Umgebungen zu testen, bevor man sie auf Test-, Staging- und Produktionsumgebungen deployt.

Für komplexe IT-Landschaften mit vielen Abhängigkeiten zwischen den einzelnen Systemen empfiehlt es sich, eine klare Trennung der Systeme zu definieren und anhand dieser Trennung für jede Anwendung eine eigene Continuous-Delivery-Pipeline mit automatisierten Deployments einzurichten. Somit kann jede Komponente unabhängig deployt werden. Auf die gleiche Art und Weise können einzelne Komponenten schrittweise fit für Continuous Delivery gemacht werden, bis alle Anwendungen vollständig automatisiert getestet und deployt werden.

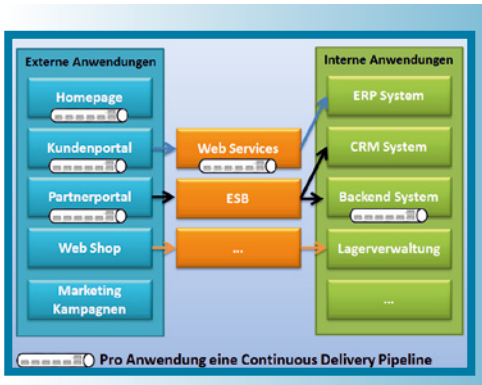

Abb. 5: Beispiel einer komplexen IT-Landschaft

Fazit

Ich hoffe, ich konnte Ihnen mit den Artikeln einen guten Einblick in das Thema Continuous Delivery geben. Vielleicht habe ich ja auch Ihr Interesse wecken können, Continuous Delivery als Einstieg zur Verbesserung des Software-Entwicklungsprozesses in Ihrem Unternehmen zu verwenden.

Literatur und Links

[AgileTesting] L. Crispin, J. Gregory, Agile Testing: A Practical Guide for Testers and Agile Teams, Addison-Wesley Professional, 2009, s. a. `http://lisacrispin.com/`

[Ant] Build-Werkzeug, `http://ant.apache.org/`

[Artifactory] Artefakt-Repository, `http://www.jfrog.com/`

[Birk13a] M. Birkner, Einführung in Continuous Delivery, in: JavaSPEKTRUM, 02/2013, s. a. `http://www.sigs-datacom.de/fileadmin/user_upload/zeitschriften/js/2013/02/birkner`

[Birk13b] M. Birkner, Technischer Aufbau einer Continuous-Delivery-Plattform in der Cloud, in: JavaSPEKTRUM, 03/2013

[Gradle] Build Tool, `http://www.gradle.org/`

[HealthCheck] Nexus Health Check, `http://blog.sonatype.com/people/2012/02/gainsomeinsightwithanexusrepositoryhealthcheck/`

[LicenseControl] Artifactory License Control, `http://www.jfrog.com/confluence/display/RTF/License+Control`

[Maven] Build-Werkzeug, `http://maven.apache.org/`

[Nexus] Artefakt-Repository, `http://www.sonatype.org/nexus/`

[Puppet] Konfigurationsmanagement-Werkzeug, `https://puppetlabs.com/`

[SonarQube] Source Code Quality Management, `http://www.sonarqube.org/`

Abb. 4: Compile-, Package- und Deployment-Prozess

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