Structr: Neues CMS auf Basis der Graphdatenbank Neo4j ▸ 48

S&S

Deutschland €9,80 Österreich €10,80 Schweiz sFr 19,50 Luxemburg €11,15 12.2014

JAVA Mag

Java™ magazin

Java • Architekturen • Web • Agile

www.javamagazin.de

Elasticsearch & Apache Lucene

4 194586 709801 12

Teil 2: Preproduction

Production-ready?

Im ersten Teil dieser Serie (siehe Java Magazin 11.2014) ging es um das Zusammenspiel zwischen Entwicklern und Betreibern rund um das Themengebiet des Loggings, und wir konnten aufzeigen, wie wichtig es bereits während der Entwicklung ist, gemeinsam zu arbeiten und sich abzustimmen. Dieses Mal werden wir uns mit der Phase der Produktionsvorbereitung beschäftigen. Die Entwicklung hat die benötigen Features und Storys umgesetzt und bereitet sich nun auf die Produktion vor. Was muss ein Team in dieser Situation leisten, um sicherstellen zu können, dass alle relevanten Schrauben korrekt sitzen? Welche Themen müssen berücksichtigt werden, und welche Tools kann man zur Unterstützung einsetzen?

von Sebastian Heib und Joachim Arrasz

Stellen wir uns folgendes Szenario vor: Ein Team hat mehrere Wochen/Monate an einem neuen, auf der grünen Wiese entstandenen Softwareprojekt gearbeitet und dabei jede für den Livegang relevante und abnahmepflichtige Story umgesetzt (**Abb. 1**). Was kommt nun? Was sollte passieren und was passiert üblicherweise?

Zunächst wird die komplette Umgebung des Systems mit der zu erwartenden Last konfrontiert. Wichtig ist hierbei, alle Szenarien, die an einem Tag vorkommen können, abzudecken. Wie aber macht man das? Man erstellt Lasttests für alle Herausforderungen, die sich aus verschiedenen Szenarien ergeben (Beispiele s. u.). Hierbei entstehen, anders als bei menschlicher Interaktion, komplett andere Lastszenarien, die es sehr genau zu beobachten gilt.

Was hilft einem ein nächtlicher Batch-ETL-Job, der irgendwann so lange benötigt, dass die Nacht schlicht nicht mehr ausreicht, um alle Aufgaben zu verarbeiten? Die einzelnen Lasttests kann man einfach mit Tools wie JMeter erstellen (**Abb. 2**), automatisieren und pflegen. Fassen wir also die verschiedenen abzudeckenden Szenarien zusammen:

- Durchschnittliche Last, über den Tag verteilt
- Durchschnittliche Last, über den Tag verteilt, an Wochenenden
- Feierabendlastspitzen
- Guten-Morgen-Lastspitzen
- Lastspitzen durch Werbung
- Verhalten der Applikation bei Überlastung (Slashdot-Effekt)
- Session Handling

Nach den erfolgreich durchgeführten Tests sollte dieses Team die Sicherheit haben, dass das System die zu erwartenden Lastanforderungen im Betrieb erfüllt, inklusive eines Puffers.

Im nächsten Schritt werden wir nun die Betriebsbedingungen genauer betrachten, um die Failover- und Load-Balancing-Mechanismen zu testen. Das Team sollte bereits seit einigen Wochen nächtliche Builds auf eine Stage-Umgebung installieren. Diese Stage sollte zwingend gleich der Produktion sein. Hier ist aber nicht die exakt gleiche Hardwareausstattung gemeint, sondern dass

- die Konfiguration,
- die Versionen der Systeme,

Tipp

Im Team sollten mittlerweile auch dringend Systemadministratoren oder DevOps vorhanden sein. Idealerweise ist ein Team von Beginn an cross-funktional aufgestellt (vgl. Teil 1 der Serie).

Tipp

Sessions sind eine althergebrachte Technik zur Identifikation eines Users und seiner applikationsrelevanten Daten. Viele moderne Anwendungen werden auf Basis von Frameworks entwickelt, die „das Session Handling auch erledigen“. Braucht man sie, erzeugt man sie pro Anwender und erst, wenn es unbedingt nötig ist, nicht mit dem ersten Aufruf des Systems, wie es oft üblich ist. Weiterhin ist es unabdingbar, die Größe einer einzelnen Session im Auge zu behalten, da Sessions leider viel zu oft als Ablage für beliebige Daten verwendet werden. Somit kann eine Session schnell ein paar Megabyte an Daten enthalten. Ein klarer Fall für Feedback an das Entwicklungsteam.

- die aktivierten Module,
- die Patchlevel,
- die Firewalls und DMZs

gleich sind. Nachdem sich ein Team exakt das Produktionssystem auf dem erklärten Niveau erarbeitet hat, kann es nun gezielt daran gehen, Störungen zu erzeugen und das System dabei genau zu beobachten (**Abb. 3**):

- Funktionieren alle Failover-Mechanismen wie erwartet?
- Reagieren die Replikationsmechanismen wie erwartet?
- Wie verhält sich die Last der Systeme bei Ausfall einzelner Systeme bzw. Verbindungen?
- Erreichen wir den gewünschten Durchsatz des Systems (wie viele gleichzeitige fachliche Transaktionen hält ein System innerhalb der SLA*-Grenzen aus)?
- Wurde eine Kapazitätsberechnung des Systems gemacht?

Hat das Team auch diese Hürde genommen und sich ein sauberes „Nach-dem-Livegang"-Set-up erarbeitet, kann es damit beginnen, die weiteren Schritte der Preproduction in Angriff zu nehmen.

Bisher haben wir uns um die Verwaltung und die Produktionsvorbereitung in Bezug auf den Applikationscode gekümmert. In den meisten Fällen ist es jedoch so, dass die Anwendung ihre Daten in einer Datenbank persistiert. Wird hierfür eine schemalose NoSQL-Datenbank verwendet, so liegt auch hier die Verantwortung ganz bei der Anwendung.

Im klassischen Fall einer relationalen Datenbank (**Abb. 4**) ist die Anwendung jedoch meist sehr hart an das vorhandene Datenbankschema gekoppelt. Wir wollen hier betrachten, was in Bezug auf ein initiales Deployment, die Weiterentwicklung sowie die fortlaufende Pflege der (relationalen) Datenbank zu beachten ist.

Grundsätzlich sollte das Ziel sein, immer genau zu wissen, auf welchem Stand die Datenbank sein muss (**Abb. 5**), um mit der Applikation kompatibel zu sein. Im Idealfall muss der Entwickler hierfür nichts tun, sondern die Anwendung selbst oder das Deployment sorgt automatisch dafür, dass die Datenbank sich im entsprechenden Zustand befindet.

Während der Entwicklung, bzw. insbesondere zu Beginn eines Projekts, erledigt dies meist der verwendete O/R Mapper (z. B. JPA über den Provider Hibernate). Dieser erzeugt beim Starten der Anwendung, anhand der vorhandenen Domainklassen, automatisch das passende DB-Schema. Für die Entwicklung ist dies eine bequeme, für Test- oder Produktivumgebungen aber keine praktikable Lösung, da hier meist eine Migration des Schemas geschehen muss. Die Grundidee besteht nun darin, die Datenbank zusammen mit der Anwendung iterativ weiterzuentwickeln bzw. anzupassen.

Der einfachste Ansatz wäre es, eine Reihe von SQL-Skripten zu sammeln, die während des Deployments in der entsprechenden Reihenfolge in die Datenbank eingespielt werden müssen. Diese Lösung ist zwar im ersten

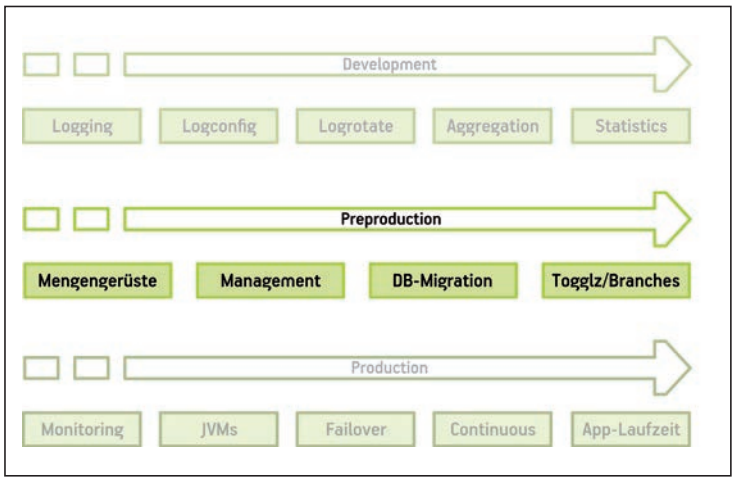

Abb. 1: Reise durch die Entstehung (das Development) über die Stabilisierungsphase der Preproduction bis hin zur Produktion

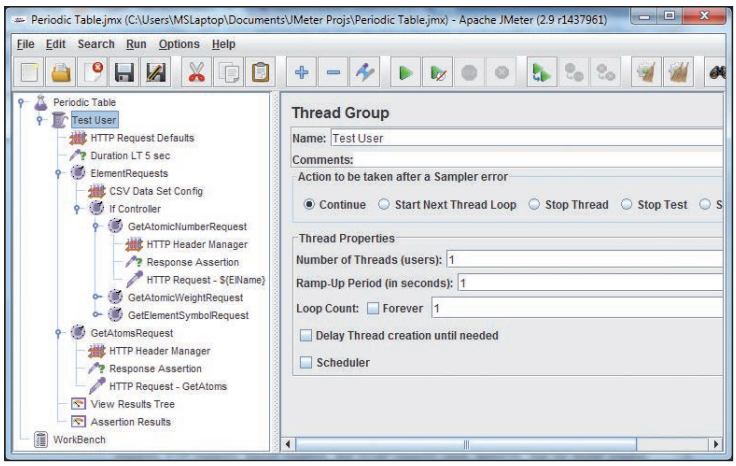

Abb. 2: Beispielhaftes Apache-JMeter-Set-up

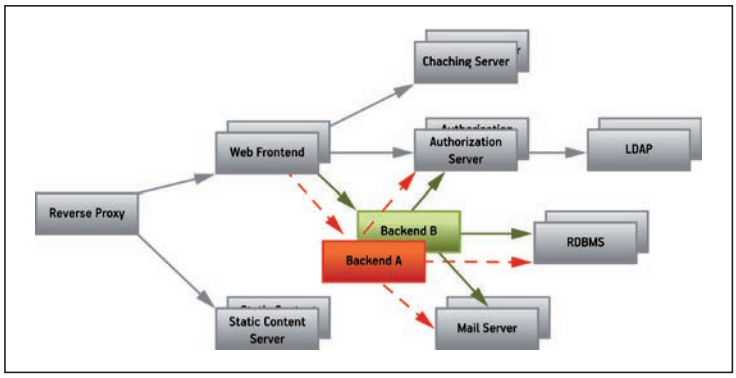

Abb. 3: Beispielhafter Failover-Verlauf

Moment sehr praktikabel; jedoch wird es hiermit sehr schnell unübersichtlich, in welchem Zustand sich gerade welche Datenbank befindet.

Wir benötigen also ein Tool, das sowohl den aktuellen Zustand unserer Datenbank verwaltet als auch die Datenbank entsprechend migriert, sodass dieser zu unserer Anwendung passt. Als bekannte Tools erfüllen sowohl Liquibase als auch Flyway diese Aufgabe. Die Funktionsweise ist dabei bei beiden sehr ähnlich: Die Datenbank wird über Change-Skripte bzw. Migrations erstellt bzw. erweitert.

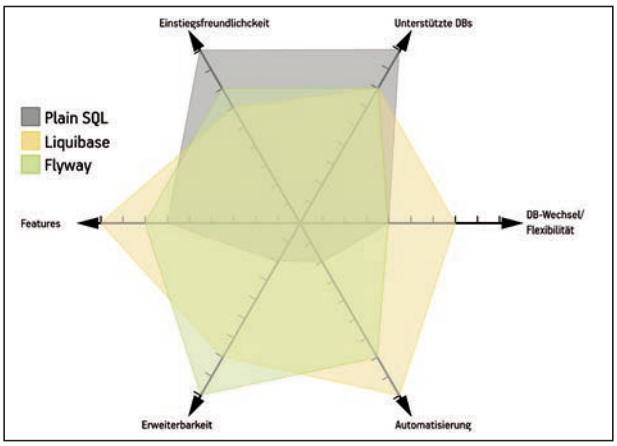

Abb. 4: Vor- und Nachteile von Liquibase, Flyway und Plain SQL

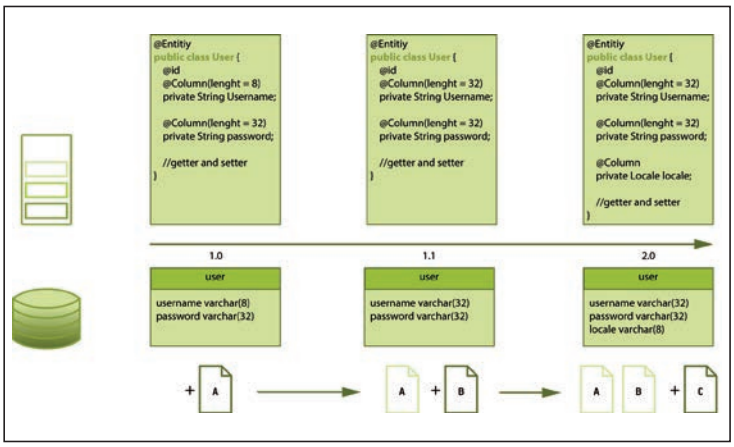

Abb. 5: Grafik zur Visualisierung der Versionen der DB

Dabei wird in einer Metatabelle innerhalb der DB mitverfolgt, welche Änderungen schon eingespielt wurden und in welchem Zustand sich dadurch die Datenbank befindet. Die Änderungen werden in definierter Reihenfolge eingespielt – jeweils die Änderungen, die zuvor noch nicht angewendet wurden. Dabei ist zu beachten, dass beiden Tools jeweils nur der Zustand der Datenbank bekannt ist, der durch das jeweilige Tool erstellt wurde. Dagegen werden manuelle (Schema-)Änderungen nicht registriert und sollten daher auf jeden Fall unterlassen werden. Beide Tools können in den Build- bzw. Deployment-Prozess integriert werden. Dadurch wird sichergestellt, dass sich die Datenbank immer genau in dem Zustand befindet, der von der Anwendung erwartet wird.

Vergleicht man beide Tools, so besteht der wesentliche Unterschied darin, wie die Change-Skripte aufgebaut sind. Flyway fährt hier den Ansatz, dass die Migrationen aus SQL-Skripten bestehen. Das bietet den Vorteil, dass vorhandene Skripte einfach übernommen werden können. Außerdem muss der Entwickler nichts Neues lernen – von der Organisation der Skripte einmal abgesehen. Ein Nachteil, der sich daraus ergibt, ist, dass man sich mit den Skripten an eine bestimmte Datenbank bindet. Für komplexere Änderungen können außerdem auch Migrationen in Java bereitgestellt werden.

Liquibase wählt dagegen einen etwas anderen Ansatz. Die einzelnen Changesets werden datenbankneutral in XML, JSON, oder YAML definiert. Das auszuführende SQL wird dann zur Laufzeit von Liquibase für die entsprechende Zieldatenbank generiert. Der Vorteil der Datenbankunabhängigkeit steht hier allerdings dem Nachteil gegenüber, dass der jeweilige Entwickler zunächst lernen muss, wie solche Changesets zu schreiben und im Dateisystem zu organisieren sind. Allerdings bietet diese Metaebene auch den Vorteil, dass neben den eigentlichen Änderungen noch weitere Angaben gemacht werden können. So ist es z. B. auch möglich, Vorbedingungen anzugeben. Nur, wenn diese erfüllt sind, wird

Info

Wie schreibt man eigentlich gute Produktionstests?
Ein guter Test beginnt damit, dass man sich die Akzeptanz des Kunden einholt und sich sicher ist, das Richtige zu implementieren. Der Kunde muss nicht zwingend menschlich sein, auch eine Maschine ist hier ein guter Kunde. (Eigentlich sogar ein besserer, da Maschinen einfacher zu interpretieren sind.) Wir befinden uns mit der Artikelserie allerdings in der Phase der Produktionsvorbereitung, daher müssen wir auch hier einmal näher betrachten, was wirklich gute und sinnvolle Testszenarien sind.
Ein Test sollte nicht prüfen, ob etwas funktioniert, sondern ob es das nicht tut. Wenn eine Anwendung erwartet, dass ein Kunde Cookies aktiviert hat, sollte auf das Ankommen der Response nicht geachtet werden. Das passiert dauerhaft während der Implementierung. Das Augenmerk muss darauf liegen, ob sie nicht ankommt. Ein Beispiel für dann aussagekräftigere Tests:

- Cookies/JavaScript disabled
- Ständige Reloads und Abbrüche durch Wechsel der Masken (oder Tabs in One Page Applications)
- Connection Pools über alle Maße strapazieren
- Massiv parallele Zugriffe produzieren und das OS und insbesondere die Anzahl von FileHandles und andere systemrelevante Ressourcen dabei genau im Auge behalten
- Parallel hierzu einmal verschiedene Nodes des verteilten Systems deaktivieren, I/O-Strecken ausfallen lassen und währenddessen Monitoring und Profiling im Auge behalten
- Sessiongröße im Verhältnis zum verfügbaren Speicher im Auge behalten, insbesondere Session-Timeouts bedenken (die Session stirbt nicht mit dem Schließen des Browserfensters, sondern nachdem serverseitig der Timeout erreicht wurde, oder aber wenn der Kunde es explizit über die Anwendung anfordert)

die Änderung auch ausgeführt. Ein weiterer Vorteil von Liquibase besteht darin, dass es für (viele) Operationen auch die Möglichkeit bietet, diese automatisiert zurückzurollen.

Ein Problem, das beide Tools nicht lösen, ist die Versionskontrolle komplexer Strukturen, wie z. B. *Stored Procedures*. Auch bei kleinen Änderungen werden diese komplett neu eingespielt. Die erfolgte Änderung ist dabei nur schwer nachzuvollziehen.

Wichtig ist auch, sich rechtzeitig Gedanken zur Konfiguration der Anwendung zu machen, insbesondere, wenn man sich für eine automatische Lösung entschieden hat. Da meistens jede Umgebung eine andere Konfiguration benötigt, sollten folgende Fragen geklärt werden:

- Woher bezieht die Anwendung zur Laufzeit ihre Konfiguration?
- Wo und wie werden diese Konfigurationsdaten verwaltet bzw. versioniert?

Im Detail hängt dies auch eng mit den verwendeten Frameworks zusammen, weshalb wir hier nicht näher darauf eingehen werden, sondern auf geeignete Artikel verweisen. Es ist aber absolut notwendig, sich Gedanken rund um das Management von Konfigurationen zu machen.

Sind alle Tests abgeschlossen, und die Software wurde als produktionsreif eingestuft, so ist es nun an der Zeit, ein Release der Software zu erstellen. Dies sollte immer geschehen, bevor ein bestimmter Softwarestand auf ein Produktivsystem deployt wird. Alleine schon, um dort nachvollziehen zu können, welcher genaue Codestand momentan am Laufen ist. Wichtig ist dabei, dass alle Abhängigkeiten auf feste Versionen verweisen; nur so kann sichergestellt werden, dass ein Build – auch zu einem späteren Zeitpunkt ausgeführt – immer das gleiche Ergebnis liefert. Ein solches Release sollte immer enthalten:

- Setzen einer festen Versionsnummer
- Erstellen eines Tags im SCM

Weiterhin sind folgende Punkte sehr hilfreich:

- Dokumentation – wann, von wem, und auf welchem System wurde dieses Release deployt?
- Gepflegte Projektdokumentation (z. B. Javadoc, Markdown).
- Erstellen (und deployen) von *sources-jars* und *javadoc-jars*.
- Festes Versionierungsschema (Stichwort Semantic Versioning).

Wird im Projekt Maven als Build-Tool eingesetzt, so ist es jetzt an der Zeit, das Maven-Release-Plug-in einzusetzen. Wurde dieses initial einmal richtig konfiguriert (Angabe des Deployment-Repositories, Setzen der *scm*-Parameter, evtl. Anpassungen der Plug-in-Konfiguration), so lässt sich ein Release mit zwei einfachen Kommandos erstellen.

maven-release:prepare nimmt notwendige Prüfungen vor (Stehen alle Abhängigkeiten auf festen Versionsnummern? Baut das Projekt? Laufen alle Tests?), setzt die Versionsnummern für das Release und die weitere Entwicklung und erstellt einen Tag für das Release.

War dieser Schritt erfolgreich, so wird mit *maven-release:perform* das eigentliche Release gebaut: Das zuvor erstellte Tag wird ausgecheckt und gebaut, die erstellten Artefakte werden in das konfigurierte Repository deployt. Zusätzlich erstellt das Plug-in auch sekundäre Artefakte mit Javadoc und Sourcen, die ebenfalls in das Artefakt-Repository geladen werden und beispielsweise in der IDE genutzt werden können und so den Entwickler unterstützen.

Was passiert nach dem ersten Livegang der Anwendung? Leider ist es oft so, dass die Endanwender nun zum ersten Mal einen Blick auf die bereits fertige Anwendung werfen (dürfen). Somit beginnt die Zeit der Evolution der Anwendung. Jetzt wird sich herausstellen, ob alle Entscheidungen, die während der Entwicklung getroffen wurden (siehe unter anderem Teil 1 der Serie), korrekt waren und die Anwendung evolutionierbar ist. Um die Weiterentwicklung korrekt zu organisieren, brauchen die Teams nun weitere Unterstützung in den Fragen:

Tipp

Verwendet man das Maven-Release-Plug-in zusammen mit Git als Versionsverwaltung, so sind folgende Properties für das Plug-in sehr hilfreich:

- *localCheckout=true*: Für das Deployment wird das Tag aus dem lokalen Repository verwendet; das spart an Bandbreite und Zeit.
- *pushChanges=false*: Alle Änderungen am Repository werden nur commitet, aber noch nicht gepusht. Fehlerhafte Releases können einfach lokal zurückgerollt werden – die Historie bleibt dadurch sauber und lesbar. Nach dem erfolgreichen Release muss der Entwickler dann natürlich seine Änderungen selbst pushen (inkl. des erstellten Tags).

Info

Feature Toggles (Implementation beispielsweise Togglz [1]) helfen enorm bei der Weiterentwicklung einer Anwendung, sind jedoch in ihrer Komplexität auf keinen Fall zu unterschätzen. Man bekommt die Möglichkeit, zur Produktionszeit mit Features und Versionen von Features zu arbeiten und diese auch schnell wieder zurückzurollen. Diese Flexibilität erkauft man sich durch Aufwand in der Konfiguration, Pflege, Komplexität und im Umfang von Tests.
Verschiedene Arten von Feature Toggles, die man einsetzt, müssen dabei akkurat unterschieden und verwaltet werden. Eine gute Vertiefung des Themas bietet der Blog von Martin Fowler [2], der eine sehr gute Aufarbeitung dieses sehr spannenden, aber auch sehr komplexen Themas geschrieben hat.

- Wie entwickle ich Anpassungen an Features?
- Wie entwickle ich neue Features?
- Wann spiele ich diese Veränderungen ein?

Dazu gibt es verschiedene Möglichkeiten:

- Master Development
- Feature und Release Toggles
- Feature und Release Branches

Um entscheiden zu können, welche der Möglichkeiten das Team wählen sollte, muss es folgende Fragen beantworten können:

- Habe ich Releases zu geplanten Zeitpunkten oder möchte ich möglichst nahtlos von der Entwicklung in die Produktion gehen?
- Gibt es ausreichend Versions- und Releaseverwaltungserfahrung im Team?
- Ist die Anwendung in ihrer Struktur und Architektur überhaupt für Feature Toggles geeignet (Stichwort: SOLID Principles)?

Feature Toggles (nahezu egal in welcher Ausführung) erhöhen die Komplexität, mit der sich das Entwicklungs- und das Produktionsteam beschäftigen muss, enorm. Daher muss in diesem Fall immer gut überlegt werden, ob der gewonnene Nutzen größer ist als die gesteigerte Komplexität.

Nachdem wir nun gesehen haben, wie Teams die Anwendung weiterentwickeln können, sollten wir uns noch Gedanken dazu machen, wie aktuellere Versionen der Anwendung installiert werden können. Auch der Fall, eine ältere Version wieder ausrollen zu können, wenn eine neue Version fehlerhaft ist, sollte nicht außer Acht gelassen werden. Aber passen denn alle Delivery-Strategien zu allen Betriebsszenarien? Schauen wir ein wenig genauer hin.

Im ersten Schritt will man die Laufzeit der Software und die erwartete Evolutionsgeschwindigkeit betrachten. Es ist wenig sinnvoll, eine komplett automatisierte Delivery-Pipeline zu erstellen, wenn die Software eine Lebenserwartung von einem Jahr oder weniger hat.

In allen anderen Fällen sollte man sich jedoch sehr genau überlegen, ob es sinnvoll ist, eine voll automatisierte Delivery-Pipeline aufzubauen. Weiterhin sollte das Team noch darauf achten, ob es eine Downtime der Anwendung(en) geben wird, und wenn ja:

- Ist dies für alle anderen Abhängigkeiten des Systems tragbar?
- Kann zu jeder Zeit deployt werden?
- Werden vertraglich zugesicherte SLAs dennoch eingehalten?

Es muss auch berücksichtigt werden, dass die Automatisierungsprozesse und deren Konfiguration verwaltet und am besten versioniert werden müssen. Das Stichwort hier: *Infrastructure as Code*.

Tipp

Unterschiedliche Versionskontrollsysteme begünstigen verschiedene Entwicklungsmodelle. In verteilten Versionierungssystemen wie Git sind Branches sehr leichtgewichtig, wodurch recht einfach mit Feature und Release Branches gearbeitet werden kann. Mittels Cherry Picking und Patches können Features auch über Branches hinweg gepflegt werden. In zentralen Versionierungssystemen wie SVN oder CVS sind Branches hingegen sehr schwergewichtig. Arbeiten mit vielen (Feature) Branches wird dadurch schwieriger und somit ein Toggle-basierter Ansatz attraktiver.

Ausblick

Dieser Teil der Serie zeigte auf, welche Schritte mindestens nötig sind, um eine Entwicklungsleistung für die Produktion vorzubereiten. Im nächsten Teil widmen wir uns der Produktion selbst und dem Feedback, das die Produktion in die weitere Entwicklung liefern kann. Zudem werden wir uns detaillierter mit dem Aufbau einer automatisierten Delivery-Pipeline auseinandersetzen.

Sebastian Heib ist als Softwareentwickler in Karlsruhe bei der synyx GmbH & Co. KG tätig und beschäftigt sich dort mit der Entwicklung verteilter Backend-Systeme und CodeClinic-Aufgaben.

Joachim Arrasz ist als Software- und Systemarchitekt in Karlsruhe bei der synyx GmbH & Co. KG als Leiter der CodeClinic tätig. Darüber hinaus twittert und bloggt er gerne.

@arrasz **http://blog.synyx.de**

Links & Literatur

[1] http://www.togglz.org

[2] http://martinfowler.com/bliki/FeatureToggle.html

[3] http://testautomationnoob.blogspot.de/2013/02/jmeter-overview.html

[4] http://media.ccc.de/browse/conferences/froscon/2014/froscon2014_-_1304_-_de_-_hs3_-_201408231515_-_datenanalyse_mit_r_fur_administratoren_-_stefan_moding.html

[5] http://www.liquibase.org

[6] http://flywaydb.org

[7] http://www.baeldung.com/2012/03/12/project-configuration-with-spring

[8] https://www.atlassian.com/git/tutorials/comparing-workflows

[9] http://martinfowler.com/delivery.html

[10] http://sdarchitect.wordpress.com/2012/12/13/infrastructure-as-code