Don't break the build!

Advanced Continuous Integration in der Praxis

Richard Attermeyer

In der Literatur werden „Broken Builds" als ein Übel dargestellt, das zu vermeiden ist. Aber ein „Broken Build" ist nur schlecht, wenn er den Arbeitsfluss des Teams aufhält. Er ist aber ein sehr gutes Feedback-Instrument für einen einzelnen Entwickler. Im Artikel wird aufgezeigt, wie man mittels Git und Jenkins durch geeignete Branching- und CI-Strategien stets einen stabilen Hauptentwicklungszweig sicherstellen und gleichzeitig private Entwickler-Builds durch den CI-Server ausführen lassen kann.

Die Herausforderung

In agilen Projekten arbeiten wir in kurzen Iterationen. Iterationslängen von zwei Wochen sind hier häufig die Norm. Am Ende der Iteration soll ein potenziell auslieferfähiges Release stehen. Einige Teams gehen sogar zu Continuous Delivery (CD) mit einem oder gar mehreren Releases pro Tag über.

Solch kurze Iterationszyklen haben den Vorteil, dass man bereits nach kurzer Zeit Feedback erhält. Gleichzeitig erfordern sie, die unproduktiven Zeiten für Entwickler möglichst klein zu halten.

Ein etabliertes Verfahren für kontinuierliches Feedback in der Entwicklung ist der Einsatz von Continuous Integration (CI). Dabei wird häufig gefordert, dass die gesamte Entwicklung direkt auf dem Hauptentwicklungszweig erfolgen soll, damit die Änderungen aller Entwickler möglichst direkt integriert werden. Allerdings hat das den Nachteil, dass sich der Entwicklungszweig dann möglicherweise nicht immer in einem releasefähigen Zustand befindet, insbesondere wenn Entwickler Code in den Zweig integrieren, der dazu führt, dass dieser Zweig nicht mehr sauber baut, oder er nicht alle aufgestellten Qualitätskriterien erfüllt.

Wenn ein CI-Build nicht erfolgreich ist, so bedeutet dies, dass sich im Hauptentwicklungszweig „schlechter" Code befindet. Jeder Entwickler, der einen Merge seines lokalen Entwicklungszweigs mit diesem Code durchführt, holt sich Code in seinen Arbeitsbereich, der verhindert, dass er weiter erfolgreich die Applikation bauen kann. Er muss also seine Arbeit unterbrechen. Das Mindeste, das nun auf ihn zukommt, ist, diesen Merge zurückzurollen. Eventuell stellt der Programmierer das Problem auch erst nach einer umfangreicheren Merge-Sitzung fest. Ein ärgerlicher Verbrauch an Zeit, ohne einen Wert erzeugt zu haben. Kommt dies häufiger vor, frustriert das verständlicherweise.

Doch es geht ja auch anders! Im Folgenden möchte ich zeigen, wie Entwicklungsprofis den CI-Prozess ohne Zeitverschwendung und Nervenstrapazen gestalten können.

Der klassische Lösungsvorschlag

Die traditionelle Lösungsmöglichkeit für das geschilderte Problem wäre, dem Team die folgende Verantwortung zu übertragen:

- Keinen fehlerhaften und ungetesteten Code einchecken!
- Nicht einchecken, solange der Build kaputt ist!
- Wenn man eingecheckt hat, nicht nach Hause gehen, bis das System erfolgreich baut!
- Verantwortung für fehlgeschlagene Builds übernehmen: Ein fehlgeschlagener Build ist unmittelbar zu beheben, ansonsten ist die Änderung rückgängig zu machen!

Dazu käme die Empfehlung, Build und Testskripte auf der eigenen Maschine laufen zu lassen (oder aber den CI-Server zu veranlassen, dies zu tun) [HuFa10].

Damit das Ausführen eines Builds auf einem Entwicklerrechner praktikabel ist, muss dieser allerdings schnell ablaufen. Darüber, was „schnell" heißt, gibt es unterschiedliche Meinungen. Wenn man davon ausgeht, dass der Build zweimal laufen muss (lokal und auf dem Integrationsserver), bevor man entsprechend den oben aufgestellten Regeln weiter entwickeln darf, dann sollte der gesamte Prozess nicht länger als fünf bis zehn Minuten dauern, also so lange, wie es dauert, sich eine neue Tasse Kaffee zu holen und kurz ein paar Worte mit den Kollegen oder dem Pairing-Partner zu wechseln. Das heißt, ein einzelner Build darf nicht mehr als fünf Minuten Zeit in Anspruch nehmen. Das geht aber nur, wenn die große Mehrzahl der Tests (schätzungsweise mehr als 80-90 %) echte Unit-Tests sind. Und je mehr Tests es insgesamt gibt, umso höher muss die Quote der echten Unit-Tests sein.

Eine weiter gedachte Alternative

Gerade bei größeren Projekten nimmt auch die Zahl der Integrationstests zu. Als Integrationstests zählen dabei alle Tests, bei denen die Dauer für das Herstellen der Testumgebung signifikant ist. Meiner Ansicht nach liegt die Grenze bei ca. zwei Sekunden. Tests brauchen schnell länger, wenn sie mittels ORM-Mapper Teile des Datenbankzugriffs testen oder mittels Shrinkwrap/Arquillian ein Archiv bauen, welches dann auf einem Applikationsserver ausgeführt wird.

Die häufigste Art von Tests, die auf den Code angewendet werden, sind Unit-Tests beziehungsweise Komponententests. Man braucht sie, um zügig Informationen über einen großen Teil der Programmfunktionen zu erhalten und zu erfahren, ob

sie in Isolation wie erwartet funktionieren. Für sich genommen reichen Unit-Tests aber häufig nicht aus, um sicherzustellen, dass der zu integrierende Code den Qualitätskriterien für den Hauptentwicklungszweig genügt. Eine gewisse Menge an weiteren Tests (wie z. B. Integrations-, Akzeptanz- und ggf. weitere Tests und statische Codeanalyse) ist notwendig, um Teams davon zu überzeugen, dass Code „gut genug" für eine Integration ist. Werden die Tests dabei alle lokal auf dem Rechner des Entwicklers ausgeführt, blockieren diese seine Arbeit.

Aber warum sollte ein Entwickler überhaupt lokal „bauen"? Häufig sind genügend freie Server-Ressourcen vorhanden, um die Builds aller Entwickler, die selten gleichzeitig gestartet werden, problemlos und häufig sogar schneller durchzuführen, als auf dem lokalen Entwicklerrechner. Dies gilt insbesondere, wenn dem CI-System mehrere Build-Knoten zur Verfügung stehen.

Und wenn wir schon Entwickler sind, wollen wir das Problem der Broken Builds im Hauptentwicklungszweig nicht durch organisatorische Regelungen beheben, sondern technisch durch einen mehrstufigen CI-Build. Das CI-System soll schließlich für den Entwickler arbeiten, und nicht umgekehrt. Ein Entwickler sollte zudem seinen Code zentral einchecken können, um seine Arbeit zu sichern, ohne dass andere Teammitglieder durch unfertigen Code gestört werden.

Wir bräuchten also einen privaten Bereich im Versionskontrollsystem, in dem jeder Entwickler seine Arbeit sichern kann, und Jobs auf dem CI-Server, die seine persönlichen Builds ausführen.

Wenn wir weiterhin von dem Ziel ausgehen, eigentlich auf dem Hauptentwicklungszweig zu arbeiten, dann sollte das CI-System noch mehr leisten: Einen Merge des persönlichen Codes mit dem Hauptentwicklungszweig, ein Build und nach erfolgreichem Build ein Commit des zusammengeführten und erfolgreich gebauten Codes in den Hauptentwicklungszweig.

Das Vorgehen hat einige Vorteile:
- Jeder Entwickler kann durch persönliche Builds seinen mit dem Hauptentwicklungszweig zusammengeführten Code zentral bauen und testen lassen.
- Tests auf dem CI-System können umfangreicher sein als lokale Tests und der Entwickler kann zwischenzeitlich weiterarbeiten.
- Es findet ein regelmäßiger, kontrollierter Abgleich mit dem Hauptentwicklungszweig statt.
- Der Hauptentwicklungszweig wird stabiler.

Von der Lösungsidee zur Umsetzung

Die von mir skizzierte Lösung stammt aus realen Projekten. Die Umsetzung erfolgte mit jeweils einem Entwicklungsteam, mit Jenkins als CI-Server und mit Git als Versionskontrollsystem. Die Teams setzten zusätzlich noch Repository-Management-Systeme ein (GitLab und Atlassian Stash), die weitere Möglichkeiten zum Schutz des Integrationszweigs bieten.

Es stellte sich also die Frage, wie man mittels Jenkins und Git private Builds abbilden kann. Einige CI-Systeme bieten dieses Feature gut dokumentiert an, Jenkins stellt aber keine direkte Unterstützung zur Verfügung. Und wahrscheinlich ist das auch der Grund, warum viele Teams die Möglichkeit eines privaten Builds nicht nutzen.

Da Git gesetzt war, wurde das Jenkins-Git-Plug-in verwendet, das eine einfache Umsetzungsmöglichkeit für private Builds anbietet. Die Entwicklung erfolgte auf einem Hauptentwicklungszweig „development". Zusätzlich verfügte jeder Entwickler über einen privaten Entwicklungszweig, der einer Namenskonvention entsprach. Für die Entwicklerin Alice hieß dieser Zweig „alice/for-development".

Auf dem Jenkins-Server wurden zwei Jobs eingerichtet: Der erste Job war für die Personal Builds verantwortlich. Er überwachte alle Zweige mit dem Muster „**/for-development". Bei einer Änderung checkt dieser Job den aktuellen Stand des Zweigs „development" aus, führt den geänderten „for-development"-Zweig mit „development" zusammen und baut diesen Stand. Der Build arbeitet also auf dem Hauptentwicklungszweig mit den aktuellen Änderungen. Waren Mergen, Bauen und die Tests erfolgreich, wird dieser neue Stand von Jenkins ins zentrale Repository gepusht (s. Abb. 1).

Aus Entwicklersicht stellt sich der Workflow wie in Abbildung 2 beschrieben dar. Jeder Entwickler holt sich die Änderungen über den Hauptentwicklungszweig „development" in seinen privaten Entwicklungszweig. Es erfolgt niemals ein Merge mit dem Zweig eines anderen Entwicklers, sondern nur mit dem Integrationszweig „development" (außer man weiß, was man tut). Gleichzeitig pusht ein Entwickler nie in den Integrationszweig.

Damit wirklich kein Entwickler direkt in den Integrationszweig pusht, kann man ein Feature von GitLab beziehungsweise Stash nutzen, welches es erlaubt, Berechtigungen auf Ent-

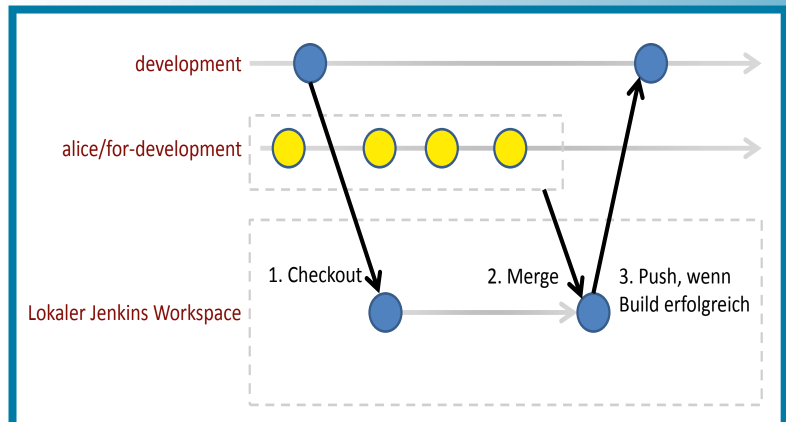

Abb. 1: Workflow während des Build

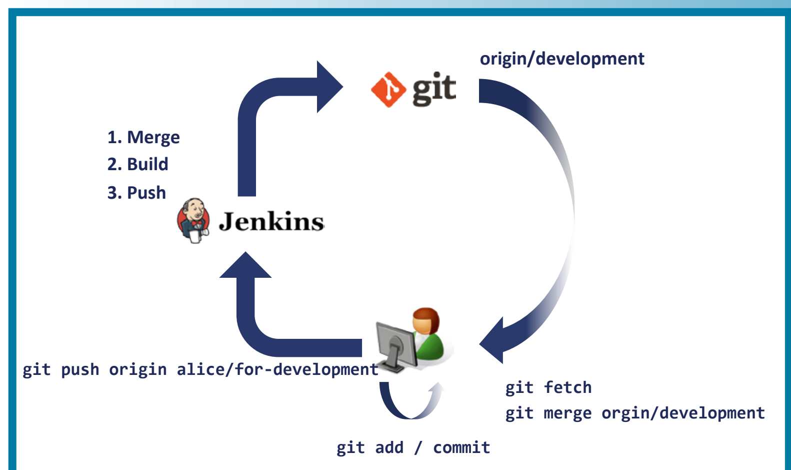

Abb. 2: Workflow aus Entwicklersicht

wicklungszweige zu vergeben. Nur der betreffende Jenkins-Benutzer darf in den Hauptentwicklungszweig pushen. Dies hat uns schon vor einigen Missgeschicken bewahrt, bei denen ein Entwickler aus Versehen Änderungen am Integrationszweig vornahm und dann ein „git push" ausführte, welches alle getrackten Zweige an das Ursprungsrepository überträgt.

Best Practices

Wir haben seit 2012 in verschiedenen Projekten sehr gute Erfahrung mit dieser Vorgehensweise sammeln können. Gerade in einem Neuentwicklungsprojekt, bei dem viel Code in kurzer Zeit produziert wurde und es noch häufig zu Code-Umorganisationen kam, verringerte diese Methode schnell die Anzahl der Frustrationsmomente der Entwickler. Sie konnten nun sicher sein, dass ihr privater Entwicklungszweig auch nach einem Merge mit dem Hauptentwicklungszweig noch erfolgreich bauen würde.

Über die Nutzung von privaten Builds kann also das Vertrauen in den Integrationszweig erhöht werden, indem vor einem Commit in diesen Zweig eine Reihe automatischer Tests (etwa Komponententests oder auch statische Codeanalyse) durchgeführt werden.

Die Verlagerung der Verantwortung für die Integration seines Code in den Hauptentwicklungszweig auf den Entwickler offenbarte aber auch Schwächen. So hörten wir manchmal Aussagen wie „Git hat meine Änderungen verschlampt". Die im Klartext nichts anderes bedeuten als „Ich habe entsprechende Nachrichten ignoriert, die mir via Chat und Mail mitteilten, dass mein privater Build fehlgeschlagen ist."

Herausforderungen

Damit jeder Commit in den privaten Zweig zu einem Build führt und möglichst nicht mehrere Änderungen zusammengebaut werden, sollte man die Möglichkeit von Git Hooks beziehungsweise Web Hooks von GitLab/Stash nutzen, um einen Build möglichst direkt zu veranlassen.

Wie sich der aufmerksame Leser vielleicht schon denken kann, gibt es noch ein anderes Problem: Trotz eines erfolgreichen privaten Builds kann es zu einem Fehlschlag kommen, wenn der Push auf den Hauptentwicklungszweig kein „Fast Forward" ist. Das tritt auf, wenn der CI-Server einen anderen privaten Build in den Hauptentwicklungszweig pusht, während der eigene private Build noch läuft.

In unserem siebenköpfigen Team ist dieser Fall so selten aufgetreten, dass sich niemand sicher erinnern konnte, ob er überhaupt vorgekommen ist. Sollte das Problem tatsächlich einmal auftreten, kann es durch einen neu angestoßenen Build recht einfach behoben werden. Die Zeiten, zu denen unterschiedliche Entwickler einen privaten Build anstoßen, verteilen sich erfahrungsgemäß recht gut über den Tag. Anders sieht es allerdings aus, wenn sehr viele Entwickler auf dem Hauptentwicklungszweig arbeiten. Dann aber wäre die Repository- und gegebenenfalls die Branching-Strategie ohnehin zu überdenken.

Bei diesem mehrstufigen Vorgehen müssen zwei unterschiedliche Anforderungen ausbalanciert werden: Zum einen die Sicherheit, dass der Hauptentwicklungszweig immer releasefähig ist. Zum anderen die Bereitstellung von schnellem Feedback sowie die Verteilung von Bug Fixes und neuen Features an die Kollegen über den Hauptentwicklungszweig.

Wichtig ist hierbei, dass das Team definiert, welche Kriterien es braucht, um die Integration eines privaten Zweigs in den Hauptentwicklungszweig zuzulassen. Über diese Kriterien sollte das Team regelmäßig reflektieren. In einem Projekt startete ein Team zum Beispiel mit der Voraussetzung, dass alle Komponenten- und Integrationstests laufen müssen. Das Team hatte zu dieser Zeit ein hohes Sicherheitsbedürfnis. Im Laufe der Zeit wurde das Projekt stabiler, und dem Team reichten Komponententests und wenige kritische Integrationstests. Dabei kam es dann durchaus vor, dass der Build des Hauptentwicklungszweigs einmal innerhalb von zwei Wochen kaputt ging. Da mehrmals täglich ein vollständiger Build mit allen Tests lief, reichte es dem Team aus, dies mit maximal vier Stunden Verspätung zu erfahren. Die Fehler waren dann häufig sehr schnell zu lokalisieren und zu beheben. Das Bedürfnis nach schnellerer Bereitstellung von Bug Fixes und Features für das ganze Team überwog hier gegenüber dem Sicherheitsbedürfnis.

Alternativ könnte man auch über einen generellen Code-Review-Prozess beispielsweise mit Unterstützung von Gerrit nachdenken. Meiner Erfahrung nach ist die Akzeptanz dafür jedoch bei vielen Entwicklungsteams recht gering, insbesondere bei Neuentwicklungen. Ein automatischer Review-Prozess, der ins CI integriert ist, wird hingegen eher angenommen.

Jenkins-Konfiguration

Damit das Vorgehen wie beschrieben gelingt, muss das Git-Plug-in wie in Abbildung 3 konfiguriert werden. Dies führt dazu, dass die Aktionen jedes Mal wie oben beschrieben ausgelöst werden, wenn ein Zweig mit der Namenskonvention „**/for-development" gepusht wird.

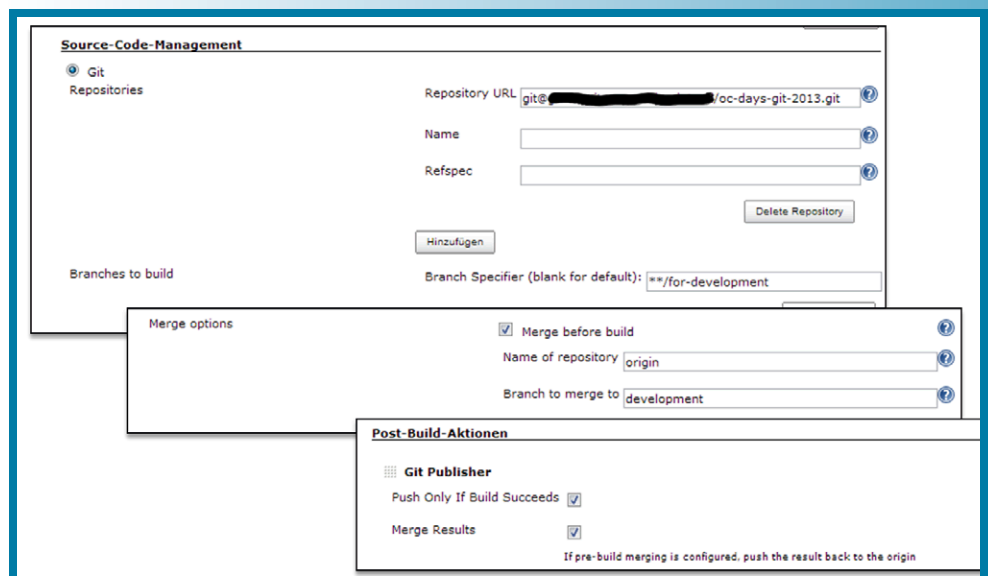

Abb. 3: Git-Plug-in-Konfiguration

Andere Umsetzungen

Nicht allen Teams stehen Git und Jenkins zur Verfügung. Das genannte Vorgehen lässt sich aber auch in anderen Szenarien umsetzen. Hier sind einige Alternativen, von denen wir wissen:

- Atlassian Bamboo: unterstützt das gezeigte Vorgehen unter dem Namen Gatekeeper. Das dort beschriebene „Branch Updater"-Muster ist ebenfalls mit dem Git-Plug-in umsetzbar [Bamboo].
- Jenkins Subversion Merge Plugin: erlaubt die einfache Erstellung von privaten Feature-Branches und CI-Jobs und kann auch so konfiguriert werden, dass ein automatisches Rebase und eine Integration wie beschrieben stattfindet [JenSVNMP].
- TeamCity Pre-Tested Commits [TeamCity]: war unser eigentlicher Motivator für die Umsetzung auf Jenkins. Pre-Tested Commits gibt es in TeamCity schon lange. Da aber in der Zwischenzeit unsere Kunden überwiegend Jenkins einsetzen, suchten wir nach Alternativen.

Wahrscheinlich gibt es auch noch weitere Lösungen, da es sich um ein Muster handelt, das sich förmlich aufdrängt. Unter Entwicklungsteams scheint das Vorgehen jedoch leider immer noch wenig verbreitet zu sein.

Literatur und Links

[Bamboo] Atlassian Bamboo, Gatekeeper Feature, https://confluence.atlassian.com/pages/viewpage.action?pageId=398393354

[HuFa10] J. Humble, D. Farley, Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation, Addison-Wesley, 2010

[JenSVNMP] Jenkins Subversion Merge Plugin, https://wiki.jenkins-ci.org/display/JENKINS/Subversion+Merge+Plugin

[PhSiWa11] S. Phillips, J. Sillito, R. Walker, Branching and merging: an investigation into current version control practices, in: Proc. of the 4th Int. Workshop on Cooperative and Human Aspects of Software Engineering, CHASE '11, pp. 9–15, ACM, 2011

[TeamCity] JetBrains TeamCity, Pre-Tested Commits, http://www.jetbrains.com/teamcity/features/delayed_commit.html

Richard Attermeyer arbeitet als Senior Solution Architect bei der OPITZ CONSULTING Deutschland GmbH. Er ist seit vielen Jahren als Entwickler, Architekt und Coach für die Themen Java EE und agile Projekte tätig und hilft Unternehmen, mit motivierten Teams erfolgreiche Projekte zu realisieren.
E-Mail: richard.attermeyer@opitz-consulting.com