

5 Hudson im Überblick

- Wer steckt hinter Hudson?
- Wie funktioniert Hudson?
- Was sind die wichtigsten »Hudson-Highlights«?
- Welche anderen CI-Systeme gibt es?

Dieses Kapitel gibt Ihnen einen schnellen und kompakten Überblick über das Projekt Hudson. Es ist somit der ideale Startpunkt, wenn Sie wenig Zeit haben oder als Projektleiter nur das »große Bild« ohne tiefgehende technische Details benötigen.

Sie erfahren dabei zunächst, wie Hudson entstanden ist und welches Team dahintersteckt. Gerade bei Open-Source-Projekten ist dieses Wissen nicht unerheblich, denn jeder Werkzeugwechsel ist eine Investition, die sich auch dauerhaft lohnen soll. Anschließend lernen Sie Hudsons Kernkonzepte, seine Architektur sowie zehn ausgewählte Highlights kennen. Zum Schluss vergleichen wir Hudson mit Alternativen, kostenlosen wie kommerziellen.

5.1 Die Hudson-Story

Hudsons Gründungslegende beginnt 2006 mit Kohsuke Kawaguchi, einem Softwareentwickler, der damals bei Sun Microsystems in Kalifornien arbeitete. Kawaguchi charakterisiert sich selbst gerne als »faul und Informatiker«. Er begann deshalb in seiner Freizeit mit der Programmierung eines Werkzeugs, das ihm langweilige und sich wiederholende Aufgaben in der Softwareerstellung abnehmen sollte – gewissermaßen ein virtueller Butler für Entwickler. Die Namenswahl für den Assistenten fiel auf »Hudson«.

Kollegen meldeten rasch Interesse an so einem persönlichen Assistenten an, und Kawaguchi war selbstverständlich alles andere als faul: So veröffentlichte er Hudson im April 2007 in der Version 1.100 auf Suns Entwicklerplattform `dev.java.net`. Seitdem erscheint im Schnitt

April 2007: Version 1.100

einmal pro Woche eine neue Version, ganz im Sinne des Prinzips »*release early, release often*«. Die Versionsnummern werden dabei einfach hochgezählt. Dieses Buch bezieht sich auf Version 1.360 (August 2010).

April 2008:
Duke's Choice Award

Einen besonderen Schub erfuhr das Projekt im April 2008 durch den »Duke's Choice Award«, den Sun Microsystems auf seiner Leitkonferenz JavaOne in der Kategorie »Entwicklerwerkzeuge« vergab. Die Auszeichnung erhöhte nicht nur die Sichtbarkeit des Projektes in der Entwicklergemeinde. Sie führte auch dazu, dass Kawaguchi sich im Rahmen seiner Arbeitszeit nun voll um Hudson kümmern konnte. Im Umfeld der Übernahme von Sun Microsystems durch Oracle Anfang 2010 beschloss Kawaguchi, seine Aktivitäten rund um Hudson in einem eigenen Unternehmen weiterzuführen, und bietet nun kommerzielle Unterstützung und Anpassungen im Rahmen seines Unternehmens InfraDNA (<http://www.infradna.com>) an. Hudson selbst bleibt weiterhin ein freies Open-Source-Projekt.

April 2010: 200 Committer

Inzwischen arbeiten über 200 Committer an Hudson, davon ungefähr 15 am Anwendungskern. Der Rest entwickelt Plugins für die unterschiedlichsten Anwendungsfälle. Im Schnitt entstehen so 1–2 neue Plugins pro Woche. Dies entspricht Hudsons Design-Philosophie, einen möglichst kleinen und kompakten Kern mit CI-Grundfunktionen bereitzustellen und die Integration von Versionsmanagementsystemen, Build-Mechanismen und Drittsystemen spezialisierten Plugins zu überlassen. Somit kann jede Hudson-Instanz durch Installation passender Plugins auf die umgebende Infrastruktur abgestimmt werden, ohne überflüssigen, ungenutzten Funktionsballast mitzuschleppen.

Inzwischen hat Hudson weite Verbreitung gefunden, von kleinen Entwicklungsteams bis hin zu den großen Namen wie etwa eBay, Hewlett-Packard, SAP, Goldman Sachs, Yahoo, Xerox, Allianz und – wenig überraschend – Sun bzw. inzwischen Oracle selbst.

MIT-Lizenz

Hudson ist kostenlos, quelloffen und steht unter der äußerst liberalen MIT-Lizenz, die eine kommerzielle Nutzung ausdrücklich gestattet. So bietet beispielsweise Sun bzw. inzwischen Oracle seit Mitte 2009 für Unternehmensanwender Supportverträge für Hudson an. Dies unterstreicht dreierlei: den fortgeschrittenen Reifegrad des Produkts, die strategische Bedeutung Hudsons bei Sun sowie die Bedeutung, die Hudson bei Unternehmensanwendern inzwischen zukommt.

5.2 Architektur und Konzepte

In diesem Abschnitt erfahren Sie, mit welchen Systemen Hudson typischerweise »nach außen« kommuniziert (Systemlandschaft) und wie Hudson mit Blick »nach innen« funktioniert (Datenmodell). Abschlie-

ßend betrachten wir, über welche Benutzerschnittstellen Sie mit Hudson arbeiten können.

5.2.1 Systemlandschaft

Zunächst aber: »Was ist Hudson? Ganz konkret, aus technischer Sicht?«

Hudson ist ein CI-Server, der als Java-Webapplikation realisiert ist. Hudson muss also immer innerhalb eines Webcontainers wie Jetty, Tomcat, JBoss, WebSphere o.Ä. betrieben werden. Wer keinen großen Webcontainer installieren möchte: Hudson bringt einen »Mini«-Webcontainer mit (Winstone), der selbst für Produktionssysteme völlig ausreichend ist.

Ein CI-System umfasst aber neben dem eigentlichen CI-Server (hier: Hudson) zahlreiche weitere Systeme. Abbildung 5-1 zeigt exemplarisch eine solche Systemlandschaft, in deren Mittelpunkt der CI-Server steht.

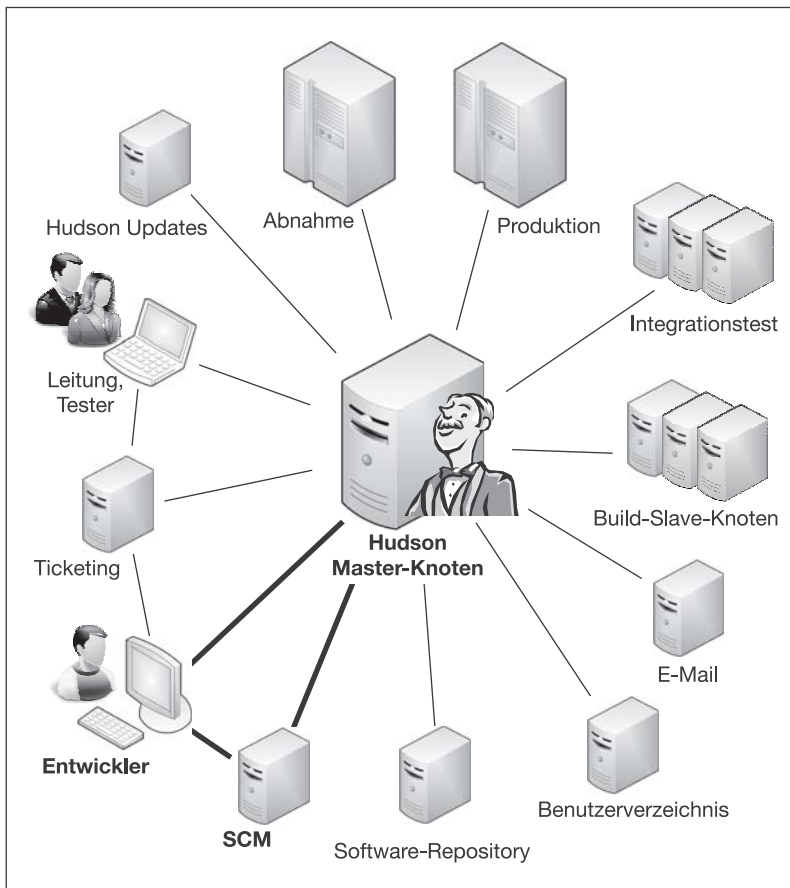


Abb. 5-1

Umgebung eines
CI-Systems

Ein minimales CI-System umfasst nur drei Komponenten (in Abb. 5–1 fett hervorgehoben):

- Einen *Entwicklerrechner*, auf dem neue Codeänderungen entwickelt und anschließend in ein Software-Configuration-Management-System (SCM-System) übertragen werden.
- Das *SCM-System* enthält alle Quelltexte, die notwendig sind, um ein Projekt zu bauen, und ordnet ihnen eindeutige Revisionen zu. Hudson verwendet den Begriff »SCM« synonym zu »Versionskontrolle«. Ein ausgewachsenes SCM-System kann aber deutlich mehr leisten als nur das Versionieren von Quelltexten und beispielsweise auch andere Artefakte eines Softwareprodukts kontrolliert verwalten. Dazu können etwa Anforderungen, Problemlberichte oder archivierte Endprodukte gehören. Hudson versteht unter dem Begriff »SCM« aber – wie eingangs erwähnt – nur den Aspekt der Versionierung. Um den Bezug zu Hudsons Benutzeroberfläche zu erleichtern, folgt dieses Buch dem abweichenden Sprachgebrauch. Wenn Sie also SCM lesen, denken Sie einfach an Subversion, CVS, Git, Perforce usw.
- Der *CI-Server*, also hier Hudson, holt sich vom SCM-System die kompletten Quelltexte eines Projekts ab und baut es ohne manuelles Zutun eines Benutzers. Die Ergebnisse sind über eine Weboberfläche abrufbar.

*Ein CI-System kommt
selten allein.*

In der Praxis besteht ein CI-System jedoch aus weit mehr Komponenten. Abbildung 5–1 zeigt ein solches »Build-Ökosystem« mit seinen typischen Bestandteilen. In der Abbildung sind alle Dienste als eigenständige Server dargestellt. In vielen kleineren Teams werden diese Dienste jedoch auch auf wenigen Rechnern zusammengefasst.

- Der *E-Mail-Server* verschickt Benachrichtigungen an die Projektmitglieder, um über den Ausgang neuer Builds zu informieren. Trotz reger Entwicklung neuer Informationskanäle (Instant Messenger, Twitter, mobile Endgeräte) ist E-Mail immer noch in den meisten Teams der Standard für Benachrichtigungsaufgaben.
- Das *Benutzerverzeichnis* dient der Authentifizierung der Benutzer und unterstützt die Verwaltung von Zugriffsrechten.
- Das *Ticketing-System* (*issue tracker*, *bug tracker*) enthält Änderungswünsche (*feature requests*) und Problemlberichte (*bug reports*). Durch Verzahnung von SCM- und Ticketing-System mit dem CI-Server wird transparent, welche Arbeiten (Ticket) wie umgesetzt wurden (SCM-System) – und ob diese Änderungen erfolgreich waren (CI-Build).

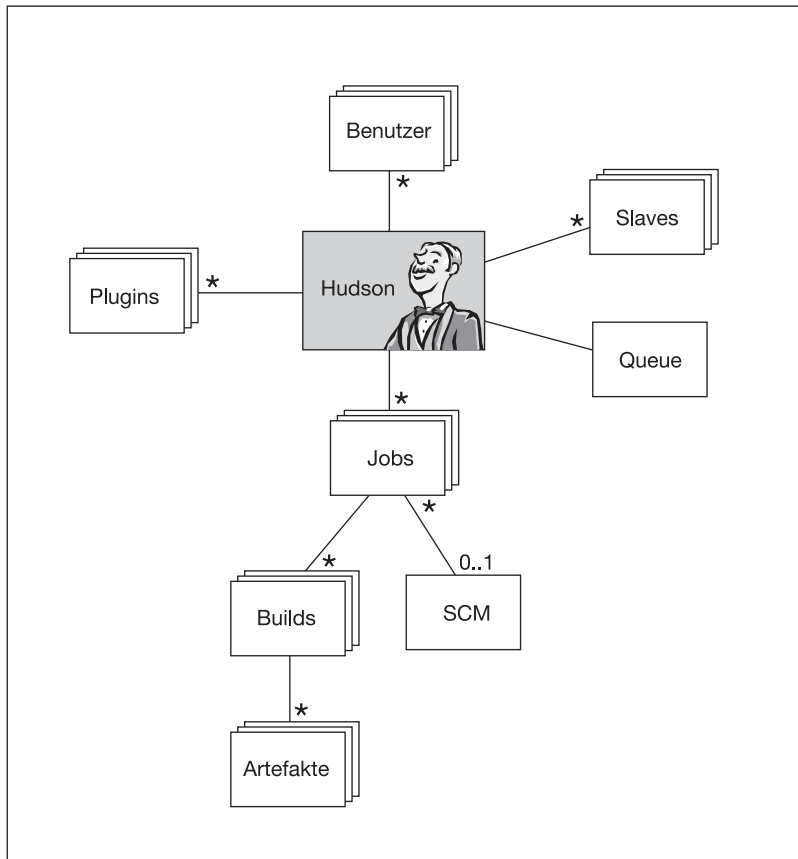
- An den Arbeitsplätzen der *Projektleitung und der Testabteilung* kann der aktuelle Status eines Softwareprojekts eingesehen werden.
- Die *Testinfrastruktur* wird für Integrationstests benötigt, bei denen die spätere Einsatzumgebung so realistisch wie möglich simuliert wird. Dazu gehören etwa Datenbanken, Applikationsserver und spezialisierte Drittsysteme.
- Auf *Demo- und Abnahmesystemen (staging server, stager)* werden die gebauten Produkte ausgebracht, um beispielsweise neue Funktionen dem Management vorzustellen oder eine Kundenfreigabe einzuholen.
- Auf dem *Produktivsystem (production system, live server)* läuft ein Softwareprodukt im Wirkbetrieb. Denkt man bei CI zunächst an die Phase der Produktentwicklung, so kann sie auch bei der Ausbringung (*deployment*) wertvolle Dienste leisten: Zum einen vermeidet die Vollautomatisierung Flüchtigkeitsfehler einer manuellen Installation. Zum anderen sorgt die Protokollierung aller Aktivitäten für eine lückenlose Nachvollziehbarkeit (»Wann haben wir das letzte Update auf dem Produktionssystem eingespielt? In welcher Version?«).
- *Build-Slave-Knoten (slave nodes)* erlauben es, den Build-Prozess auf mehrere Rechner zu verteilen. Dies ist besonders attraktiv, wenn viele Projekte gebaut werden müssen, die einzelnen Builds lange laufen oder das Produkt in heterogenen Umgebungen (z.B. auf unterschiedlichen Betriebssystemen) getestet werden muss.
- In *Software-Repositories* befinden sich Bibliotheken, die während des Build-Vorgangs in das Endprodukt integriert werden. Dabei kann es sich um Produkte von Drittherstellern handeln (z.B. Open-Source-Frameworks), aber auch um Eigenentwicklungen, die der besseren Handhabung halber als eigenständige Module verwaltet werden. Bekannte Vertreter dieser Gattung sind Maven-Repositories wie Nexus oder Artifactory. Ein CI-Server kann Software-Repositories nicht nur auslesen, sondern auch mit neuen Versionen füllen, die während eines Build-Laufs entstanden sind.
- Der öffentliche *Update-Server* des Hudson-Projekts hält Aktualisierungen für die Hudson-Webapplikation und zugehörige Plugins bereit und unterstützt bei der automatischen Installation von JDKs, Ant und Maven. Dies funktioniert allerdings nur, wenn Hudson einen Zugang zum Internet besitzt. Große Unternehmen setzen durchaus auch interne Update-Server auf, um die Auswahl an installierbaren Plugins besser zu kontrollieren und firmeneigene Plugins einfacher verteilen zu können.

5.2.2 Datenmodell

Hudsons Datenmodell stellt eine »Innensicht« auf den gesamten CI-Vorgang dar. Abbildung 5–2 zeigt die wichtigsten Objekte des Modells, die wir im Folgenden kurz erörtern.

Abb. 5–2

Hudsons Datenmodell



Hudson

Im Zentrum des Datenmodells steht das Hudson-Objekt, von dem es pro Hudson-Installation nur eine Instanz gibt. Das Hudson-Objekt erfüllt zweierlei Funktionen: Zum einen enthält es alle systemweiten Konfigurationen und Zustandsinformationen, beispielsweise Einstellungen zum E-Mail-Versand oder zur Absicherung des Systems. Zum anderen nimmt das Hudson-Objekt als Master-Knoten (*master node*) an der Ausführung von Builds teil.

Jobs

Wichtigste Aufgabe einer Hudson-Installation ist die Ausführung von Jobs. Hudson unterscheidet zwei Kategorien von Jobs: Projekte und externe Jobs.

Projekte bilden den typischen Anwendungsfall eines CI-Systems ab, also Auschecken von Quelltexten, Bauen und Testen der Artefakte, Archivieren der Ergebnisse und Versenden von Benachrichtigungen. Die weit überwiegende Mehrheit aller Jobs dürfte vom Typ »Projekt« sein. Um ausgewählte Szenarien noch besser zu unterstützen, bietet Hudson momentan Projekte in drei Ausprägungen an:

Projekte

- Ein *Free-Style-Projekt* bietet die meisten Freiheitsgrade. Dem Auschecken der Quelltexte schließen sich ein oder mehrere Build-Schritte an, z. B. mit Ant, Maven, Batchdateien oder Shell-Skripten. Abschließend werden die Build-Ergebnisse ausgewertet, gegebenenfalls archiviert und Benachrichtigungen verschickt.
- Ein *Maven-2-Projekt* ist da schon sehr viel mehr auf Mavens besondere Art und Weise spezialisiert, Software zu bauen. Durch Auswertung der Informationen in POM-Dateien erspart man sich einige manuelle Angaben in der Projektkonfiguration und erschließt sich zusätzliche Funktionalität wie automatisches Abhängigkeitsmanagement zwischen weiteren in Hudson angelegten Maven-2-Projekten.
- Ein *Multikonfigurationsprojekt* (auch als Matrix-Build bezeichnet) erlaubt die mehrfache Durchführung eines Builds in unterschiedlichen Konfigurationen. Dies könnten beispielsweise unterschiedliche JDK-Versionen, Datenbanken oder Applikationsserver sein. Hudson baut dabei alle spezifizierten Konfigurationen und stellt die Ergebnisse in zusammengefasster, tabellarischer Form dar.

Externe Jobs dienen der Überwachung von Abläufen, die nicht innerhalb Hudsons stattfinden, sondern – wie es der Name ja andeutet – extern. Bei externen Jobs fungiert Hudson also nicht als aktiver Ausführer eines Jobs, sondern eher als passiver Buchhalter, der den Ausgang eines anderen Prozesses zur Archivierung und Visualisierung mitgeteilt bekommt.

Externe Jobs

Builds

Der Begriff »Build« ist – je nach Kontext – unterschiedlich belegt, so dass wir die Verwendung in Zusammenhang mit Hudsons Datenmodell festlegen müssen:

- Zum einen kann darunter der Build-*Prozess* verstanden werden, der in einer langen Kette Schritt für Schritt des Build-Vorgangs ausführt: Auschecken, Kompilieren, Testen, Dokumentation erstellen, Archivieren, Verteilen, Benachrichtigen usw. Diese Bedeutung ist im folgenden Text *nicht* gemeint.
- Zum anderen kann ein »Build« auch das Ergebnis einer Ausführung eines Jobs sein. In dieser Bedeutung wird der Begriff in Hudsons Datenmodell und auch im folgenden Text verwendet.

Bei jedem Lauf eines Jobs entsteht also ein neuer, weiterer Build. Hudson nummeriert diese Builds pro Job streng monoton steigend durch, so dass beispielsweise Build »foobar #5« den fünften Lauf des Jobs »foobar« darstellt und es innerhalb einer Hudson-Installation nur genau einen Build mit dieser Bezeichnung geben kann.

Builds können auf unterschiedliche Weise ausgelöst werden: manuell, nach Zeitplan (z.B. einmal pro Nacht), bei Veränderungen im SCM oder durch Abhängigkeiten zu anderen Projekten.

Der Ablauf eines Build-Vorgangs erfolgt nach einem festen Schema. Zunächst werden die Quelltexte aus dem SCM abgerufen. Dann erfolgt eine Reihe von Build-Schritten, typischerweise Ant-, Maven- oder Skript-Aufrufe, welche die Quelltexte in Artefakte umwandeln. In einer letzten Phase werden abschließende Aktionen (*post-build actions*) ausgeführt, etwa das Verschicken von E-Mail-Benachrichtigungen, Auswerten von Testberichten, Archivieren von Dateien und ggf. Auslösen weiterer Builds.

Da Hudson die Ergebnisse jedes Builds archivieren kann, sind später nicht nur die Daten eines einzelnen Builds abrufbar (»Wie lange dauerte eigentlich Build foobar #5?«), sondern auch Trends über den Verlauf aller Builds (»Werden unsere Builds wirklich langsamer – oder wir nur ungeduldiger?«).

Artefakte

Das Ergebnis eines Builds wird als Artefakt bezeichnet. Dabei handelt es sich immer um Dateien, beispielsweise das Installationsprogramm einer Software, eine Webapplikation oder eine JAR-Datei, die als Bibliothek in andere Projekte eingebunden werden soll. Im Gegensatz zu allen anderen Dateien, die im Verlauf eines Builds entstehen und beim nächsten Build wieder gelöscht oder überschrieben werden können, werden Artefakte dauerhaft archiviert.

Ein Artefakt besteht technisch gesehen immer aus genau einer Datei. Ein Build kann jedoch mehrere Artefakte erzeugen. Für Projekte vom Typ »Free Style« gibt man Hudson mithilfe eines regulären Aus-

drucks in der Notation eines Ant-FileSets die Namen der Dateien an, die archiviert werden sollen. Bei Projekten vom Typ »Maven« ist durch die POM-Datei ja bereits beschrieben, welches Endergebnis der Maven-Build liefern soll. Hier müssen Sie nicht explizit Ihr Artefakt konfigurieren – Hudson findet dies durch Auswertung der POM-Datei selbst heraus.

SCM

Kontinuierliche Integration setzt voraus, dass alle Quelltexte eines Builds aus einem Versionsmanagementsystem bezogen werden. Hudson verwendet hierzu synonym den Begriff SCM (*Software Configuration Management*). Sie können pro Projekt ein oder mehrere SCM-Quellen angeben. Wird in einer dieser Quellen eine Änderung festgestellt, wird aus allen SCMs der aktuelle Stand bezogen und das Projekt gebaut.

Benutzer

Hudson verwaltet eine Liste aller ihm bekannten Personen. Wie baut Hudson diese Liste auf? Personen finden auf zwei unterschiedlichen Wegen Zugang:

- Personen werden in der Benutzerverwaltung explizit angelegt. Dieser Personenkreis benötigt in der Regel interaktiven Zugriff auf die Oberfläche von Hudson.
- Personen tragen zu einer SCM-Revision als Committer bei und werden von Hudson während des Builds implizit angelegt. Im Laufe eines Projektes lernt Hudson also alle Committer automatisch durch Auswertung der SCM-Revisionen kennen. Dieser Personenkreis muss nicht zwingenderweise Zugriff auf die Hudson-Oberfläche benötigen.

Neben Namen und Beschreibungstext können für Personen weitere Angaben hinterlegt werden, z.B. deren E-Mail-Adresse.

Plugins

Eine der größten Stärken Hudsons ist seine Plugin-Architektur. Hudson wird mit einer Handvoll Plugins ausgeliefert, die vor allem die Subversion-Anbindung und die Maven-Unterstützung beinhalten. In den weitaus meisten Fällen wird man über den eingebauten Plugin-Manager aus dem öffentlichen Plugin-Verzeichnis des Hudson-Projekts individuell ausgewählte Plugins hinzuininstallieren.

Hudson verwendet übrigens kein bekanntes Plugin-Framework mit Isolationsmöglichkeiten der Plugins untereinander wie etwa OSGi. Die theoretisch vorstellbaren Konflikte bereiten in der Praxis glücklicherweise (noch) kein allzu großes Kopfzerbrechen. Alle Plugins tragen eine Versionsnummer und sind mit Metadaten ausgestattet, die angeben, für welche Hudson-Version das Plugin entwickelt wurde.

Queue

Alle Build-Aufträge werden zunächst in einer Warteschlange (*build queue*) aufgenommen und deren Ausführung geplant. Hudson verteilt diese Build-Aufträge dann an freie Build-Prozessoren (*builders*) aller angeschlossener Knoten. Ein Job kann nicht mehrfach in der Queue geplant werden. Maximal kann ein Job momentan gebaut werden und dessen nächste Ausführung bereits in der Queue geplant sein. Dies vermeidet lange »Rückstaus«, die entstehen würden, wenn für ein Projekt mit vielen Änderungen in kurzer Zeit immer weitere neue Jobs geplant würden.

Slaves

In verteilten Builds werden Build-Aufträge nicht nur auf dem Master-Knoten ausgeführt, sondern auch an sogenannte Slave-Knoten (*slave nodes*) delegiert. Welche Vorteile bringt diese Verteilung? Die wichtigsten drei Motive aus der Praxis:

■ *Parallelisierung der Gesamtlast des CI-Systems.*

Durch das Verteilen der Jobs auf mehrere Rechner soll die Last parallelisiert und somit schneller abgearbeitet werden. Dies funktioniert am besten mit vielen, aber tendenziell kurzen Builds, da die Parallelisierung auf Job-Ebene stattfindet. Ein einzelner, stundenlanger »Bandwurm«-Build kann also nicht von Hudson auf magische Weise zerlegt und auf mehrere Knoten verteilt werden.

■ *Unterstützung unterschiedlicher Plattformen.*

Es soll auf mehr als einer Plattform gebaut und getestet werden. Slave-Knoten müssen nicht identisch in Hard- und Software sein. Sie können für jeden Job spezifizieren, welche Umgebung zur Ausführung benötigt wird. Hudson delegiert den Job dann an einen passenden Knoten.

■ *Entlastung des Master-Knotens.*

In manchen Hudson-Installationen werden Builds sogar ausschließlich auf Slave-Knoten gebaut, um auf dem Master-Knoten mehr Ressourcen für andere Aufgaben zur Verfügung zu haben.

Auf diese Weise bleibt beispielsweise die webbasierte Benutzeroberfläche auch während äußerst rechenintensiver Builds immer flüssig bedienbar und wird nicht von parallelen Prozessen »an die Wand gedrückt«.

5.2.3 Benutzerschnittstellen

Nachdem Sie das Umfeld eines Hudson-Servers und dessen Innenleben als Datenmodell kennengelernt haben, betrachten wir abschließend, über welche Schnittstellen Sie mit Hudson in Kontakt treten können. Dazu stehen Ihnen gleich drei Möglichkeiten zur Auswahl (Abb. 5–3):

- die webbasierte grafische Oberfläche (GUI)
- die REST-ähnliche Anwendungsschnittstelle (API)
- die Kommandozeilenanwendung (CLI)

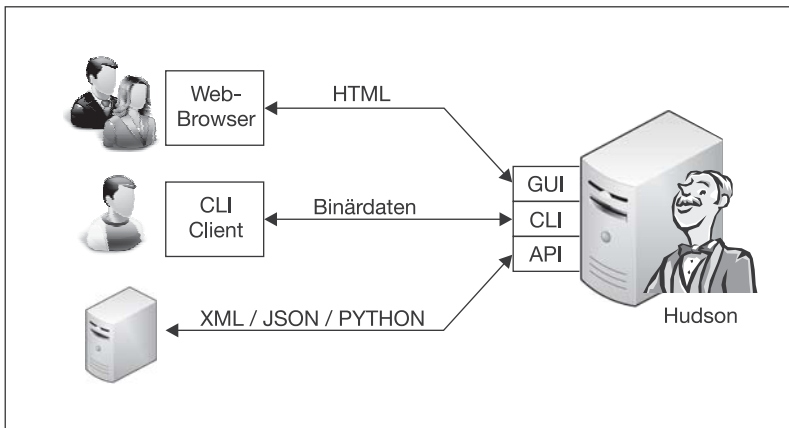


Abb. 5–3
Benutzerschnittstellen

Die webbasierte grafische Oberfläche (GUI)

Die meisten Anwender kommunizieren mit Hudson ausschließlich über die webbasierte Oberfläche. Mit Kenntnis der Konzepte aus dem vorausgegangenen Abschnitt kann man sich schon sehr viel bei einem Rundgang durch die Oberfläche erschließen. Folgende »Bonbons« sollten Sie jedoch keinesfalls verpassen:

Zu den meisten Konfigurationsmöglichkeiten ist eine umfangreiche Online-Hilfe hinterlegt. Nutzen Sie sie! Der Umfang dieser Hilfen übersteigt die im Hudson-Wiki verfügbare Dokumentation bei Weitem. In der Hilfe finden Sie zudem in vielen Fällen gebrauchsfertige Beispiele, die Sie herauskopieren und in Ihre Einstellungen einfügen können.

Online-Hilfe

Abb. 5-4
Beispiel einer
ausführlichen Online-Hilfe

Source Code Management System abfragen

Zeitplan `# Alle volle Viertelstunde auf Änderungen prüfen`
`*15 * * * *`

Dieses Feld verwendet die Cron-Syntax (mit kleinen Unterschieden). Jede Zeile besteht dabei aus 5 Feldern, getrennt durch Tabulator oder Leerzeichen:

MINUTE STUNDE TAG MONAT WOCHENTAG

MINUTE Die Minute der Stunde (0-59)
STUNDE Die Stunde des Tages (0-23)
TAG Der Tag des Monats (1-31)
MONAT Der Monat (1-12)
WOCHENTAG Der Wochentag (0-7), 0 und 7 entsprechen dem Sonntag.

Um mehrere Werte pro Feld anzugeben, können folgende Operatoren verwendet werden. In absteigender Priorität sind dies:

- '*' entspricht allen gültigen Werten
- 'M-N' gibt einen Bereich an, z.B. "1-5"
- 'M-N/X' oder '* /X' gibt Schritte in X-er Schritten an durch den Bereich an, z.B. "*/15" im Feld MINUTE für "0,15,30,45" und "1-6/2" für "1,3,5"
- 'A,B,...,Z' entspricht direkt den angegebenen Werten, z.B. "0,30" oder "1,3,5"

Leere Zeilen und Zeilen, die mit '#' beginnen, werden als Kommentarzeilen ignoriert.

Zusätzlich werden '@yearly', '@annually', '@monthly', '@weekly', '@daily', '@midnight' und '@hourly' unterstützt.

Beispiele # Jede Minute
* * * * *
Immer 5 Minuten nach der vollen Stunde
5 * * * *

Online-Hilfe mit Beispielen im Funktionskontext

Sprechende URLs

Der Aufbau der URLs der Weboberfläche erfolgt einer klaren Systematik, die Hudsons Datenmodell widerspiegelt. So lassen sich auf einfache Weise »Einsprung-URLs« bilden, die Sie etwa aus einem Wiki oder einem Ticketing-System direkt an die gewünschte Stelle innerhalb der Hudson-Oberfläche führen. Zusätzlich existieren nützliche Permalinks, also konstante URLs, die zu besonders interessanten Stellen führen, etwa `/job/foobar/lastSuccessfulBuild` zum letzten erfolgreichen Build des Projektes foobar. Tabelle 5-1 zeigt weitere Beispiele dieser Systematik.

Tab. 5-1
Beispiele für Hudsons
sprechende URLs

URL	Bedeutung
/	Übersicht (Dashboard) anzeigen
/jobs/foobar	Übersicht zu Projekt »foobar« anzeigen
/job/foobar/5	Build Nr. 5 des Projekts »foobar« anzeigen
/job/foobar/5/testReport	Testergebnisse aus Build Nr. 5 des Projekts »foobar« anzeigen
/job/foobar/lastBuild	Letzten Build des Projekts »foobar« anzeigen
/job/foobar/build?delay=0sec	Neuen Build des Projekts »foobar« ohne Verzögerung sofort starten
/user/swiest	Benutzer »swiest« anzeigen

URL	Bedeutung
/pluginManager/installed	Liste aller installierten Plugins anzeigen
/view/MeineProjekte	Listenansicht »MeineProjekte« anzeigen

Meist übersehen: In der oberen rechten Ecke befindet sich eine Volltextsuche, mit der Sie sich einige Klicks ersparen können. Beispiel: Sie haben ein Projekt »foobar« angelegt. Eine Suche nach »foo 2 co« führt Sie dann – durch automatische Vervollständigung – direkt zur Konsoleausgabe des Builds Nr. 5 des Projekts »foobar«.

Volltextsuche



Abb. 5-5

Schnelle Navigation über Volltextsuche

Zu vielen Objekten (z.B. Knoten, Jobs, Builds, Personen) können Sie Beschreibungen hinterlegen, die dann in der Oberfläche angezeigt werden. Dieser Beschreibungstext kann HTML-Auszeichnungen enthalten. Damit lassen sich nicht nur Formatierungen einfügen, sondern auch Tabellen aufbauen und Verknüpfungen zu externen Systemen herstellen. Abbildung 5-2 enthält Anregungen, was Sie in den Beschreibungen hinterlegen könnten.

Ressource	Beispiele für Beschreibungen
Knoten	Maßgebliche Hardwarekomponenten des Knotens, installierte Software, Kontaktinformationen des Administrators, regelmäßige Wartungszeiten
Jobs	URLs auf Staging- und Produktivsysteme oder korrespondierende Übersichten im Ticketing-System
Builds	SCM-Revisionsnummer, Modulversion (Maven)
Personen	URL auf Blog

Tab. 5-2

Einsatzbeispiele für Beschreibungen

Die REST-ähnliche Anwendungsschnittstelle (API)

Mit der Anwendungsschnittstelle (API) können Sie Hudson fernsteuern – zum Beispiel aus einem Skript heraus. Im Prinzip rufen Sie per HTTP genau dieselben URLs auf, die Sie interaktiv in der GUI anklicken würden. Die URLs sind REST¹-ähnlich nach folgendem Schema aufgebaut:

1. Representational State Transfer (REST) ist ein Softwarearchitekturstil, bei dem jede Ressource einer Anwendung durch eine eigene Adresse, z.B. eine URL, angesprochen werden kann [Fielding 2000]. Auf jede dieser Ressourcen können dann bestimmte Operationen angewendet werden, etwa Anlegen (PUT), Verändern (POST), Abrufen (GET) oder Löschen (DELETE).

```
http://hudson/pfad/zur/ressource/aktion?param1=wert1&param2=wert2...
```

Beispiel: Sie möchten einen Build des Jobs »foobar« starten. Dazu rufen Sie mit dem Linux-Kommando `wget` folgende URL auf:

```
wget http://hudson/jobs/foobar/build
```

Benötigt Ihr Aufruf Parameter oder erwarten Sie strukturierte Daten in der Rückantwort, so bietet Hudson eine praktische, eingebaute Dokumentation aller möglichen API-Funktionen an: Dazu navigieren Sie in der GUI auf die Seite der Ressource, die Sie fernsteuern möchten, zum Beispiel die Übersichtsseite eines Projekts. Dann hängen Sie an die URL ein `/api` an und bekommen die unterstützten API-Aufrufe angezeigt:

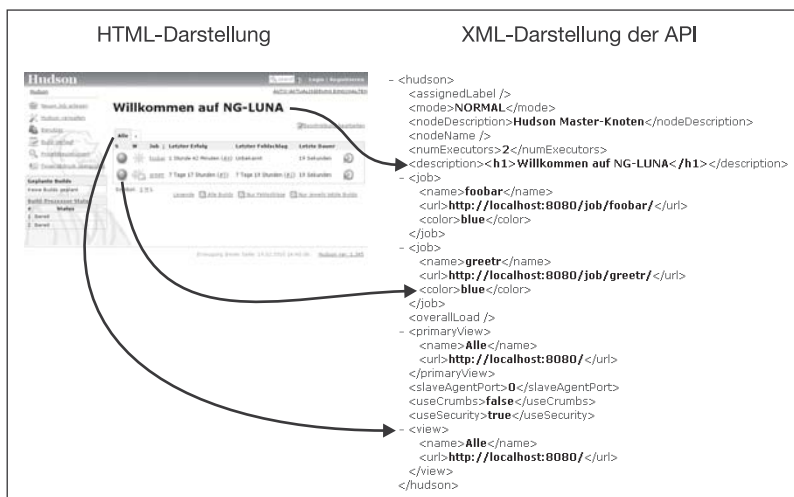
```
wget http://hudson/jobs/foobar/api
```

Rückgabeformate

Je nach gewünschtem Rückgabeformat hängen Sie zusätzlich `/xml`, `/json` oder `/python` an ihren API-Aufruf. Hudson liefert Ihnen dann die Ergebnisse in diesen »maschinenlesbaren« Formaten. Abbildung 5–6 zeigt Ihnen links die Übersichtsseite eines Hudson-Systems (`http://hudson/`) und rechts daneben die entsprechende XML-Version der REST-Schnittstelle (`http://hudson/api/xml`).

Abb. 5–6

Darstellung der
Startseite als XML



Für die wichtigsten Ansichten stehen API-Funktionen zur Verfügung. Bei Plugins ist dies bisher leider die Ausnahme. Benötigen Sie Daten aus einem Hudson-System, die nicht über die REST-Schnittstelle abgefragt werden, könnte Ihnen die Kommandozeilenanwendung weiterhelfen.

Die Kommandozeilenanwendung (CLI)

Als dritte Benutzerschnittstelle bietet Hudson eine Kommandozeilenanwendung an, das *command line interface*, kurz: CLI. Das CLI schließt eine Lücke zwischen Weboberfläche und REST-API, da es besser automatisierbar ist als die Weboberfläche, sich gleichzeitig aber besser interaktiv nutzen lässt als die REST-API.

Woher erhalten Sie die passende CLI-Anwendung für Ihre Hudson-Instanz? Ganz einfach: Hudson bringt sie bereits mit. Unter der URL <http://hudson/cli> können Sie das CLI in Form eines Java-Archivs herunterladen.

Der Aufruf, der einen Build des Projektes »foobar« startet, muss lediglich den anzusprechenden Hudson-Server, das Kommando `build` und den Projektnamen enthalten:

```
java -jar hudson-cli.jar -s http://hudson build foobar
```

Eine Übersicht aller verfügbaren CLI-Kommandos erhalten Sie mit:

```
java -jar hudson-cli.jar -s http://hudson help
```

Beachten Sie, dass Sie auch für das Kommando `help` die Server-URL Ihres Hudson-Servers angeben müssen. Die verfügbaren Kommandos sind nämlich nicht im CLI implementiert, sondern hängen vom Server ab, den Sie ansprechen. Aktuelle Hudson-Versionen bieten über 30 Kommandos an, etwa zum Ein- oder Ausschalten eines Wartungsfensters, zur Build-Queue-Steuerung und zur Fernkonfiguration.

Ein wahres »Schweizer Taschenmesser« für Administratoren stellt die Möglichkeit dar, Groovy-Kommandos per CLI auf dem Hudson-Server ausführen zu lassen. Beispiel: Sie möchten per Skript abfragen, welche Plugins in welcher Version auf Ihrer Hudson-Instanz installiert wurden. Dies können Sie folgendermaßen realisieren:

Groovy über das CLI

```
$ java -jar hudson-cli.jar -s http://localhost:8080 groovysh =>
' Hudson.model.Hudson.instance.pluginManager.plugins.each { =>
println("${it.longName} - ${it.version}") };'
Static Analysis Utilities - 1.3
Hudson batch task plugin - 1.13
Checkstyle Plug-in - 3.2
Hudson CVS Plug-in - 1.0
Maven Integration plugin - 1.345
Hudson Support Subscription Notification Plugin - 1.2
Hudson SSH Slaves plugin - 0.9
Hudson Subversion Plug-in - 1.8
```

5.3 Die Top-10-Highlights

In diesem Abschnitt lernen Sie zehn ausgewählte »Hudson-Highlights« kennen: von der Installation über die täglichen Handgriffe bis hin zu Erweiterungsmöglichkeiten durch Plugins. Eine Auswahl an Highlights hat zugegebenermaßen immer etwas Subjektives, erfolgte hier aber aus der Perspektive eines Softwareentwicklers bzw. IT-Projektleiters.

5.3.1 Schnelle Installation

Wenn Sie bereits andere Serveranwendungen installiert und getestet haben, wissen Sie, dass dies mitunter eine abendfüllende Veranstaltung werden kann: Zunächst muss das passende Installationspaket für den eigenen Rechner gefunden werden. Im Anschluss sind dann Abhängigkeiten und Vorbedingungen zu erfüllen (leider auch die undokumentierten). Und speziell unter Windows startet man nicht selten ein Installationsprogramm, das ungefragt Dateien und Registry-Einträge auf der Festplatte verteilt...

*»Hudson komplett«
in einer WAR-Datei*

Hudson ist in dieser Hinsicht erfrischend einfach gestrickt: Es gibt nur eine Datei zum Herunterladen (unter der konstanten URL <http://hudson-ci.org/latest/hudson.war>), in der die komplette Software enthalten ist. Diese kann in einen Servlet-Container (z.B. Jetty, Tomcat, JBoss) ausgebracht werden oder wird ganz einfach direkt in der Kommandozeile gestartet. Ein spezielles Installationsprogramm ist also nicht notwendig. Darüber hinaus legt Hudson alle seine Daten kompakt unterhalb eines einzigen Datenverzeichnisses ab, dem sog. <HUDSON_HOME>. Möchte man sich von Hudson wieder trennen, löscht man dieses Verzeichnis und hat ein rückstandsfrei gesäubertes System. Die Datenhaltung in Dateien macht außerdem die Installation oder Anbindung eines Datenbanksystems überflüssig.

5.3.2 Effiziente Konfiguration

Der Installation schließt sich die Konfiguration an. Je nach Philosophie des Softwareherstellers geschieht dies typischerweise entweder durch Anpassung von Textdateien oder aber über eine grafische Benutzeroberfläche. Beide Ansätze haben ihre Stärken und Schwächen: Textbasierte Konfigurationen lassen sich einfach sichern, versionieren und automatisch erstellen, erfordern aber Lernaufwand oder – Himmel bewahre! – einen Blick ins Handbuch. Gute grafische Benutzeroberflächen sind hingegen intuitiver zu verstehen. Es ist aber unklar, wie und wo die Konfigurationsdaten intern gespeichert werden.

Hudsons Philosophie verbindet beide Ansätze elegant, indem alle Einstellungen über die Browseroberfläche veränderbar sind, hingegen serverseitig als XML-Dateien abgespeichert werden. Somit kann der Administrator je nach vorliegender Aufgabe und persönlichen Vorlieben entscheiden, ob er seine Arbeit schneller im Browser oder im Texteditor erledigen kann.

*Grafische Konfiguration
im Frontend, XML im
Backend*

In der Praxis verwenden die meisten Administratoren das Webinterface – selbst hartgesottene Kommandozeilen-Liebhaber. Sie schätzen gleichzeitig aber die einfache Sicherheits- und Wiederherstellungsmöglichkeit der XML-Konfigurationsdateien.

5.3.3 Unterstützung zahlreicher Build-Werkzeuge

Bei aller Standardisierung: Jedes Build-System ist ein kleiner Mikrokosmos, der individuell an die Infrastruktur einer Arbeitsgruppe angepasst ist. Hudson ist es weitestgehend egal, wie Sie Ihre Software bauen. Selbstverständlich werden die wichtigsten Werkzeuge aus der Java-Welt, Ant und Maven, direkt unterstützt. Sie können aber auch Shell-Skripte bzw. Batch-Dateien starten und somit ausgefallene oder proprietäre Build-Werkzeuge einbinden. Über Plugins erhalten Sie weitergehende Unterstützung für jüngere oder stärker spezialisierte Build-Werkzeuge wie etwa gant oder rake.

*Direkte Unterstützung
von Ant, Maven, Shell und
Batch*

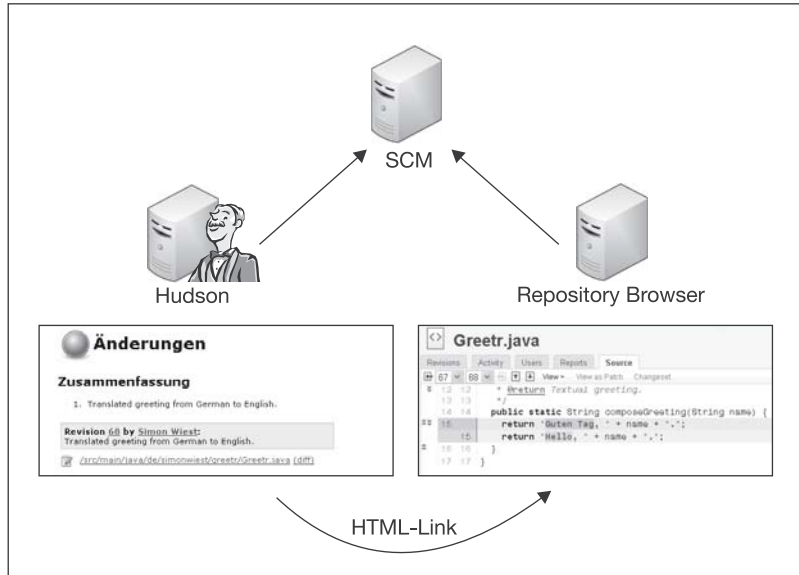
5.3.4 Anbindung von Versionsmanagementsystemen

Kontinuierliche Integration setzt voraus, dass alle für einen Build benötigten Daten in einem Versionsmanagementsystem verwaltet werden. Hudson unterstützt die »üblichen Verdächtigen«, CVS und Subversion, direkt. Kommerzielle Vertreter wie Perforce oder Open-Source-Projekte wie Git oder Mercurial können per Plugin angebunden werden. Hudson kann Versionsmanagementsysteme nicht nur nach Änderungen befragen, sondern auch detaillierte Informationen zu einzelnen *change sets* anzeigen, also wer wann was wo geändert hat.

*Direkte Unterstützung von
Subversion und CVS*

Wenn Sie einen spezialisierten Repository-Browser verwenden, um in Ihren versionierten Daten zu navigieren (z.B. FishEye, Sventon oder ViewSVN), so können Sie direkt aus Hudson über Links in entsprechende Übersichten im Repository-Browser springen. Abbildung 5–7 zeigt dies aus der Anwendersicht: Im linken Teil der Abbildung sehen Sie einen Ausschnitt aus Hudsons Weboberfläche. Dateinamen sind mit Links hinterlegt, die direkt zum Repository-Browser führen, der auf der rechten Seite dargestellt ist (hier: Atlassian FishEye).

Abb. 5-7
Verknüpfung zwischen
Hudson und einem
Repository-Browser



5.3.5 Testberichte

*Direkte Unterstützung
von JUnit*

Obwohl »kontinuierliche Integration« im strengen Wortsinn nur das regelmäßige Zusammenführen aller Produktbestandteile bezeichnet, wird in der Praxis auch das Testen mit eingeschlossen. Typische Test-Frameworks in der Java-Welt sind JUnit, TestNG oder Selenium. Für viele dieser Testwerkzeuge kann Hudson deren Ergebnisprotokolle verstehen und grafisch aufbereiten. Somit haben Sie in Hudson nicht nur alle Codeänderungen im Blick, sondern Seite an Seite auch die zugehörigen Testergebnisse. Abbildung 5-8 zeigt exemplarisch eine navigierbare Auswertung eines JUnit-Laufs auf Ebene eines Java-Packages.

Testergebnis : de.acme.controller

Fehlgeschläge(+5) Teste(40)
Took 4 Minuten 36 Sekunden
@pschreibrubu.hinsufugen

Alle fehlgeschlagenen Tests

Testname	Dauer	Alter
>>> de.acme.controller.ValueItemListPropertiesTabControllerTest.testGetListValuesEditorProperty	5.097	1
>>> de.acme.controller.ValueItemListPropertiesTabControllerTest.testGetPropertySheet	6.179	1
>>> de.acme.controller.ValueItemListPropertiesTabControllerTest.testGetPropertySheetForNewValueItemList	7.109	1
>>> de.acme.controller.ValueItemListPropertiesTabControllerTest.testSavePropertiesForListValues	5.022	1
>>> de.acme.controller.ValueItemListPropertiesTabControllerTest.testUpdateListValues	5.01	1

Alle Tests

Klasse	Dauer	Fehlgeschlagen	(Veränderung)	Skip	(Veränderung)	Summe	(Veränderung)
AbstractPropertiesTabControllerTest	1 Minute 29 Sekunden	0		0		18	
FlagPropertiesTabControllerTest	5 Sekunden	0		0		1	
FeaturePropertiesTabControllerTest	16 Sekunden	0		0		5	
FilePropertiesTabControllerTest	26 Sekunden	0		0		5	
LocalPropertiesTabControllerTest	16 Sekunden	0		0		3	
ProductPropertiesTabControllerTest	1 Minute 29 Sekunden	0		0		15	
RuleBasedProductPropertiesTabControllerTest	9,1 Sekunden	0		0		1	
ValueItemListPropertiesTabControllerTest	20 Sekunden	5	+5	0		5	

Abb. 5-8

JUnit-Testergebnisse
eines Java-Packages

5.3.6 Benachrichtigungen

Die wenigsten von uns werden den ganzen Tag vor einem Hudson-Browserfenster sitzen wollen, um Build-Probleme frühzeitig zu erkennen. Hudson kann Sie daher über eine Vielzahl von Kommunikationskanälen auf dem Laufenden halten, u.a. per E-Mail, Instant-Messenger, Twitter, RSS-Feed. Auch sogenannte *eXtreme Feedback Devices* (XFD) lassen sich mit minimalem Aufwand anschließen. So können Sie Erfolg und Misserfolg Ihrer Builds auf intuitive und originelle Weise signalisieren lassen. Wie wäre es beispielsweise mit einer Ampel aus Leuchtbären (Abb. 5-9)?

Direkte Unterstützung von
E-Mails und RSS-Feeds

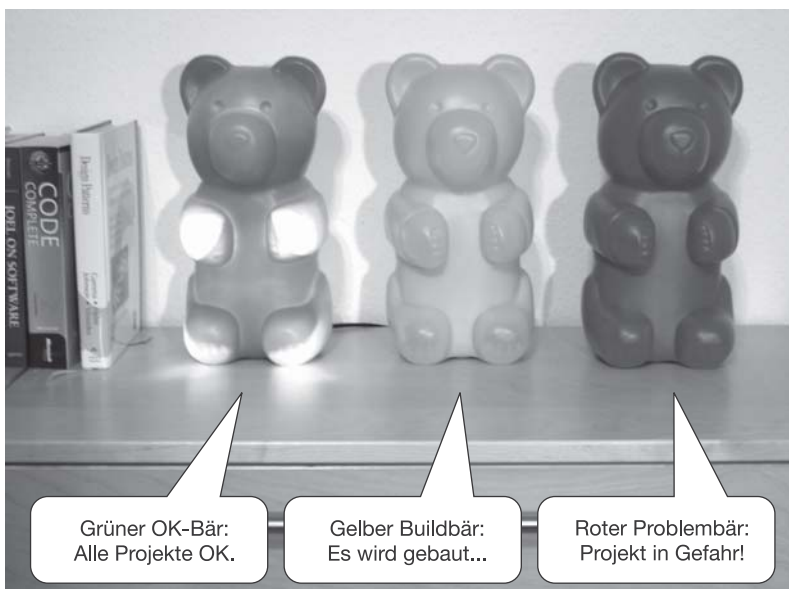


Abb. 5-9

»Bärenampel« als eXtreme
Feedback Device

5.3.7 Remoting-Schnittstelle

Effizienz durch Remoting

In den meisten Fällen wird man Hudson interaktiv über die Weboberfläche benutzen. Für bestimmte Aufgaben ist dies jedoch mühsam: Stellen Sie sich vor, Ihr Chef hat sich an Ihrer Begeisterung für Hudson angesteckt und weist Sie an, alle 236 Projekte Ihrer Abteilung in Hudson anzulegen. Manuelles »Durchklicken« wäre hier viel zu langsam und fehleranfällig. Stattdessen lässt sich Hudson mit einfachen REST-Kommandos fernsteuern, z.B. aus einer Batch-Datei oder einem Perl-Skript heraus. Die REST-Schnittstelle nimmt nicht nur Kommandos entgegen, sondern erlaubt auch den Abruf von Informationen über die Hudson-Instanz, ein bestimmtes Projekt, einen bestimmten Build usw. Um die maschinelle Weiterverarbeitung zu erleichtern, stellt Hudson diese Informationen in XML-, JSON- sowie Python-Notation bereit.

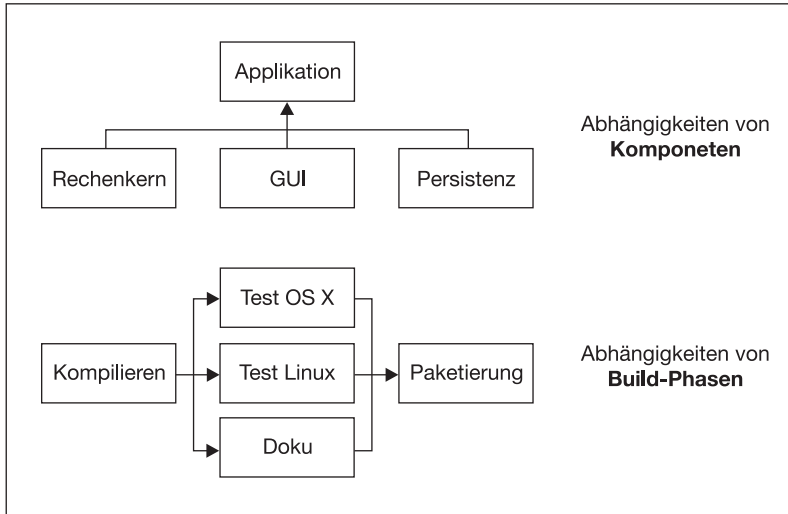
Zusätzlich können Sie mit einem Kommandozeilen-Client (CLI) die wichtigsten Funktionen einer Hudson-Instanz aus der Entfernung ausführen.

5.3.8 Abhängigkeiten zwischen Jobs

*Automatisches
Management von
Abhängigkeiten*

Die wenigsten Softwareprojekte werden völlig isoliert voneinander entwickelt, sondern stehen untereinander in Beziehung. Ein Beispiel: Eine Applikation setzt sich aus den drei Modulen Rechenkern, GUI (Benutzeroberfläche) und Persistenz (Datenhaltung) zusammen (Abb. 5–10, oben). Bei einer Änderung des Rechenkerns sollte nicht nur dieses Modul, sondern auch die resultierende Applikation neu gebaut und getestet werden, um Probleme im Zusammenspiel mit der GUI oder der Persistenzschicht zu entdecken. Hudson bietet dazu die Möglichkeit, Beziehungen zwischen vor- und nachgelagerten Projekten zu definieren. Der erfolgreiche Build des Rechenkerns stößt dann automatisch die Integration und den Test der Applikation an. Wünschenswerterweise werden dabei nur diejenigen Teile des Produkts neu gebaut und getestet, die von den Änderungen betroffen sein könnten. Das aufwendige Bauen *aller* Module nach *jeder* Änderung kann entfallen. Dies spart Ressourcen und verkürzt die Build-Zeit.

Abhängigkeiten können auch genutzt werden, um einen langen Build-Ablauf in Phasen zu unterteilen. Für jede Phase wird in Hudson ein eigenes Projekt mit entsprechenden Abhängigkeiten angelegt. Im Idealfall kann Hudson dann die Ausführung bestimmter Phasen parallelisieren und die Build-Zeit verringern (Abb. 5–10, unten).

**Abb. 5-10**

Abhängigkeiten zwischen
Projekten

5.3.9 Multikonfigurationsbuilds (»Matrix-Builds«)

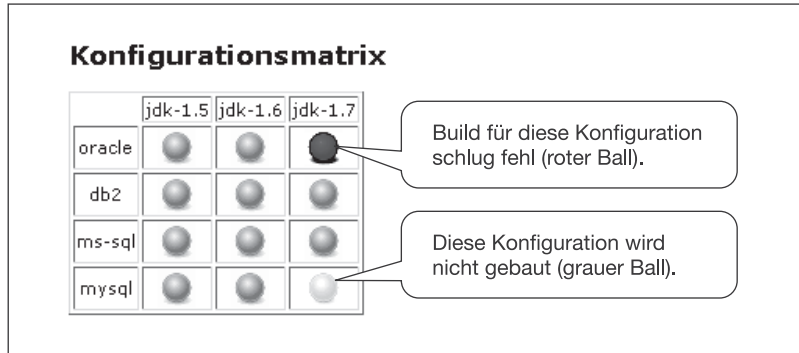
Stellen Sie sich vor, Sie entwickeln eine Software, die für drei Java-Versionen und vier Datenbankanbieter gebaut und getestet werden soll. Kommt Ihnen das bekannt vor? Sie könnten natürlich für jede dieser zwölf Kombinationen einen Job in Hudson anlegen. Übersichtlich und einfach zu verwalten wäre das allerdings nicht mehr.

Verwandte

*Konfigurationen in einem
Projekt verwalten*

Abhilfe schaffen hier Hudsons Multikonfigurationsbuilds, oft auch als Matrix-Builds bezeichnet: Dazu verändern Sie zunächst Ihr Build-Skript so, dass es mit zwei Parametern für Java-Version und Datenbankanbieter aufgerufen werden kann, etwa durch Setzen von Umgebungsvariablen vor dem Start des Build-Skripts. Dann legen Sie in Hudson einen Job vom Typ »Multikonfigurationsbuild« an. Sie definieren zwei Achsen für die Java-Version und den Datenbankanbieter mit den jeweiligen Ausprägungen. Wird dieser Job nun gestartet, baut und testet Hudson für Sie vollautomatisch alle 12 Kombinationen durch und stellt die Ergebnisse übersichtlich in einer Tabelle dar (Abb. 5-11). Sind bestimmte Kombinationen aus technischen oder fachlichen Gründen nicht möglich, können Sie diese gezielt vom Build ausschließen. Sie können sogar Einfluss auf die Reihenfolge der Abarbeitung der Kombinationen nehmen, um kurz laufende Builds zuerst ausführen zu lassen. Somit erfahren Sie von neu aufgetretenen Problemen noch schneller und können früher mit deren Behebung beginnen.

Abb. 5-11
Effiziente Verwaltung
von Konfigurationen
durch Hudson



5.3.10 Verteilte Builds

*Schnellere Builds
durch Verteilung*

Buildzeiten haben die unliebsame Angewohnheit, stetig zu wachsen. Zum einen entsteht im Laufe eines Projekts mehr und mehr Quelltext, der kompiliert und getestet sein möchte. Zum anderen verlockt die Vollautomatisierung dazu, ein immer größeres Arsenal an Inspektionen und Analyseverfahren in den Build aufzunehmen. Fast jedes Entwicklerteam stellt sich daher irgendwann die Frage: »Wie schrumpfen wir unsere Build-Zeiten?«

Eine ebenso naheliegende wie mächtige Idee ist die Verteilung des Builds auf mehrere Rechner über das Netzwerk. Doch Vorsicht, der Teufel steckt hier im Detail:

- Was passiert, wenn ein Rechner zeitweilig vom Netz geht?
- Wie werden die verteilten Build-Ergebnisse wieder zentral zusammengeführt und dargestellt?
- Wie kann ein Rechner während des Builds auf Teilergebnisse eines anderen Rechners zugreifen?
- Wer verteilt die benötigten Build-Werkzeuge auf alle beteiligten Rechner?
- Wie können Rechner mit heterogenen Betriebssystemen gleichförmig angesprochen werden?

Hudson bietet hier zahlreiche Funktionen und Konzepte an, welche die Realisierung eines verteilten Builds immens erleichtern, z.B. Überwachung der Verfügbarkeit der beteiligten Rechner oder automatische Verteilung benötigter Build-Werkzeuge.

5.3.11 Plugins

Ihr Arbeitgeber verwendet aus historischen Gründen ein ausgefallenes Versionsmanagementsystem, der Senior-Architekt möchte seine Lieblingsmetriken in Hudson sehen, Sie möchten ein brandneues Werkzeug in Ihren Build-Prozess integrieren: In der Praxis werden Sie immer wieder vor unvorhersehbare Herausforderungen gestellt. Hudson hält dazu über 200 Plugins aus den unterschiedlichsten Anwendungsgebieten bereit. Sollte das passende Plugin fehlen, so ist zumindest die Chance groß, dass sich ein bestehendes Plugin für den eigenen Fall anpassen lässt. Die liberale MIT-Open-Source-Lizenz lädt ein, bestehenden Quelltext als Sprungbrett für eigene Entwicklungen zu verwenden. Falls Sie ein eigenes Plugin realisieren möchten, finden Sie in Kapitel 9 eine Einführung in die Plugin-Entwicklung für Hudson.

*Infrastruktur-Chamäleon
dank Plugins*

5.4 Hudson im Vergleich zu Mitbewerbern

Wie in der Einleitung des Buches bereits angesprochen, mag Hudson nicht für alle Arbeitsgruppen der ideale CI-Server sein. Bei aller Euphorie, die Sie von einem Hudson-Committer erwarten dürfen, sollen an dieser Stelle auch die wichtigsten Alternativen angesprochen werden.

Die Auswahl an CI-Servern ist zahlenmäßig groß. Eine Vergleichsmatrix mit den dreißig bekanntesten Vertretern finden Sie unter [ThoughtWorks10]. Trotzdem lässt sich eine gewisse Konzentration auf wenige Produkte feststellen. Im Folgenden sollen diese Alternativen zunächst vor- und dann Hudson gegenübergestellt werden.

Zuvor noch ein kurzes Wort der Warnung: Die vorliegende Auswahl der Alternativen in diesem Abschnitt stellt keine objektive Erhebung oder Empfehlung dar, sondern basiert auf der Häufigkeit der Nennung in persönlichen Gesprächen des Autors mit Anwendern. Vor- und Nachteile einzelner Produkte können sich zwischenzeitlich geändert haben, so dass Sie für Ihre Vorhaben stets die aktuellen Versionsstände miteinander vergleichen sollten. Alle hier genannten Produkte sind als kostenlose Testversionen (zeitlich beschränkt) oder Einsteigerversionen (leistungsbeschränkt) verfügbar, so dass Sie mit überschaubarem Aufwand Ihre Projekte in unterschiedlichen CI-Servern testen können.

5.4.1 Proprietäre Eigenentwicklungen

Jeder Vergleich wäre unvollständig, wenn er nicht zunächst auf die unzähligen, selbst entwickelten »Hauslösungen« zur Automatisierung des Build-Prozesses einginge.

Architektur und besondere Merkmale

In der Regel handelt es sich bei diesen Hauslösungen um einen bunten Mix von sich gegenseitig aufrufenden Programmen, die als Shell-Skripte, in Perl und einem halben Dutzend weiterer Skriptingsprachen im Lauf der Jahre gewachsen sind. Speziell in stark kontrollierten Branchen (etwa im Finanzsektor oder der Medizintechnik) werden gesetzliche Auflagen zur Dokumentationspflicht und Nachvollziehbarkeit oftmals durch proprietäre Erweiterungen realisiert, die eine spätere Migration auf verbreitetere Standardprodukte erschweren. Darüber hinaus kann der Wartungsaufwand nicht unerhebliche Dimensionen erreichen: Ein Build-System benötigt zahlreiche Schnittstellen zu weiteren Systemen (u. a. Versionsverwaltung, Repositories, Testsysteme, Staging-/Produktionsserver, Benutzerverwaltung). Da diese Systeme sich laufend weiterentwickeln, müssen auch die Schnittstellen eines hauseigenen Build-Systems angepasst werden. Durch den proprietären Charakter können dafür aber nur eigene Mitarbeiter eingesetzt werden – von weit verfügbaren und breit getesteten Lösungen kann in diesem Falle nicht profitiert werden.

Typischerweise wird in diesen Umgebungen nicht kontinuierlich nach Änderungen gebaut, sondern in festen Zeitintervallen, etwa einmal pro Nacht oder Woche. Öfter könne ja auch nicht gebaut werden, so die häufig zu hörende Begründung, da der Build über mehrere Stunden laufe.

Muss bereits großer Aufwand in die Erstellung und laufende Pflege des Build-Systems gesteckt werden, ist es nicht verwunderlich, dass Auswertungen über mehrere Builds hinweg eher stiefmütterlich implementiert werden. Nicht selten beschränkt sich das »Reporting« auf E-Mails, die am Ende des Build-Prozesses abgesetzt werden, oder auf unstrukturierte Protokolldateien im zweistelligen Megabyte-Bereich.

Vergleich zu Hudson

Glücklicherweise weisen die genannten Probleme bereits auf ihre Lösung: Umstellung auf verbreitete Build-Werkzeuge, Zerlegen des Build-Prozesses in kürzere Teilmodule sowie Visualisierung der Ergebnisse durch eine darauf spezialisierte Anwendung, also durch einen CI-Server wie Hudson.

In der Praxis gestaltet sich dieser Wandel jedoch schwieriger als erwartet: Es bremsen mangelnde Ressourcen (»Der Kunde zahlt schließlich nur das Produkt, nicht den Prozess.«), Angst vor Risiken (»Da kennt sich keiner aus – besser nichts ändern ...«), aber auch persönliche Befindlichkeiten (»Werden ich oder meine Skripte verzichtbar?«). Realistisch betrachtet wird man in diesen Fällen eine Verbesserung nur durch viele kleine, inkrementelle Verbesserungen herbeiführen können, beginnend mit der Kapselung der Schnittstellen zu Drittsystemen und Modularisierung von Build-Phasen. Diese Module werden dann sukzessive auf gängige Build-Werkzeuge migriert und dienen als natürliche Trennstellen zum Zerlegen monolithischer »Bandwurm-Builds«.

Parallel dazu können bestehende Build-Prozesse bereits als *black box* durch CI-Server angestoßen werden. Oft lässt sich schon in dieser Phase das Ausgabeformat eines Builds so ändern, dass die Visualisierung durch den CI-Server übernommen werden kann. Beispiel: Werden Testergebnisse nicht unstrukturiert in eine Protokolldatei geschrieben, sondern als JUnit-kompatible XML-Datei ausgegeben, können praktisch alle gängigen CI-Server diese Ergebnisse aus dem Stand visualisieren.

Fazit

Hauseigene CI-Systeme starten zwar klein und schnell. Sie tendieren aber dazu, zu hochkomplexen Monstern zu wachsen, die nur noch von Mitarbeitern mit Kopfmonopol überschaut werden können. Möchte man sich das Nachentwickeln von CI-Grundfunktionen sparen, gleichzeitig aber eine weitgehende Integration mit hauseigenen Systemen erreichen, drängt sich als Alternative ein quelloffenes Produkt mit guten Erweiterungsmöglichkeiten auf – etwa Hudson.

5.4.2 CruiseControl

CruiseControl (<http://cruisecontrol.sourceforge.net>) darf zu Recht als der Altvater der CI-Systeme bezeichnet werden. Seit fast einer Dekade leistet CruiseControl unauffällig, aber zuverlässig seine Dienste und stellte für viele Entwickler die Eintrittskarte in die Welt der CI dar. Ursprünglich von der Firma ThoughtWorks initiiert, wurde das Projekt bereits früh als Open Source veröffentlicht und ist kostenlos auf der Sourceforge-Plattform verfügbar.

Altvater der CI-Systeme

Da zahlreiche Artikel und Anleitungen im Netz auf dieses Projekt verweisen, steht der Name CruiseControl in vielen Arbeitsgruppen sogar synonym für Continuous Integration.

CruiseControl.NET

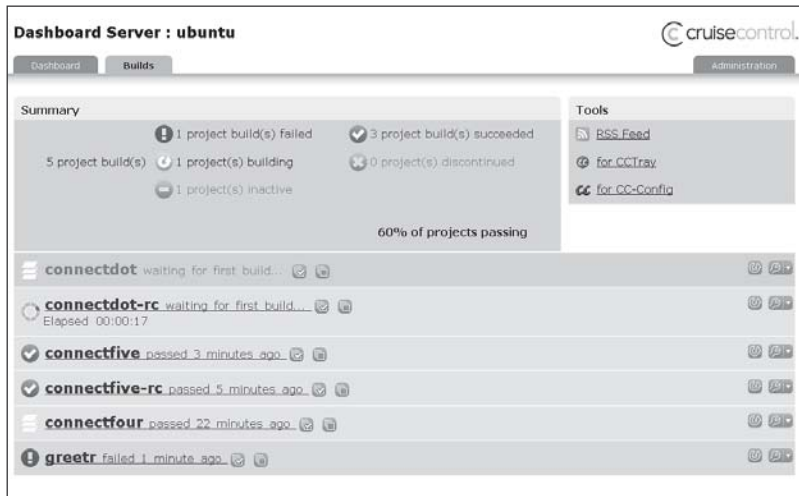
Für weitere Bekanntheit sorgte das Schwesterprojekt CruiseControl.NET (<http://ccnet.thoughtworks.com>), das vornehmlich die Werkzeugwelt der Microsoft-Landschaft bedient, etwa NAnt, MSBuild, NUnit, NDepend.

Abb. 5-12

Build-Übersicht in

CruiseControls

»Dashboard«-Oberfläche



Architektur und besondere Merkmale

Zwei Komponenten:
»Die Schleife« und die
Weboberfläche

Ein CruiseControl-Server besteht aus einem Prozess, der zeitgesteuert nach Änderungen in Versionsmanagementsystemen horcht und gegebenenfalls neue Builds anstößt. Nach Beendigung des Builds werden Benachrichtigungen abgesetzt. Da dieser Prozess endlos läuft, wird er als »die Schleife« (*the loop*) bezeichnet. Abgesehen von einer Konsolenausgabe hat dieser Prozess keine Oberfläche, über die ein Benutzer den Fortschritt beobachten könnte. Stattdessen liefert CruiseControl gleich zwei webbasierte GUIs mit: die »klassische« Ansicht und das *dashboard* (Abb. 5-12, Abb. 5-13). Beide visualisieren den Status der Schleife und zeigen Informationen aus vergangenen Builds an. Technisch betrachtet handelt es sich bei beiden GUIs um Java-Webapplikationen, die in einem schlanken Jetty-Container betrieben werden.

CruiseControl fokussiert sich auf die wichtigsten CI-Schritte »Änderungen erkennen, Projekt bauen, Team benachrichtigen«. Umfangreiche Visualisierungen oder ein ausgefeiltes Rechte- und Rollenmodell sucht man hier vergebens.

Wenige Möglichkeiten
zur Interaktion

Ohne sein großes Verdienst um die Popularisierung des CI-Gedankens schmälern zu wollen, merkt man CruiseControl sein Alter an: Die Oberflächen dienen ausschließlich der passiven Betrachtung des Fortschritts. Außer dem manuellen Auslösen eines Builds stehen hier keine

Aktionen zur Verfügung. Auch lässt sich CruiseControl nicht über die Oberfläche konfigurieren. Dies erfolgt ausschließlich über Änderungen an einer XML-Datei. Gerade an diesem Umstand entzündete sich bei vielen Anwendern der Unmut. Wie schwer dieser Nachteil in der Praxis wiegt, kommt auf den Umfang der Änderungsaktivitäten an. Gerade bei sporadisch durchgeführten Wartungsarbeiten wünscht man sich eine geführte Konfigurationsmöglichkeit herbei, die Benutzereingaben automatisch validiert und kontextsensitive Hilfen bereithält.

CruiseControl unterstützt in der aktuellen Version auch verteilte Builds über mehrere Rechner in einer Master-Slave-Architektur durch ein optionales Erweiterungspaket. Master und Slaves müssen dabei individuell eingerichtet, konfiguriert und gestartet werden. Die Zuteilung von Build-Aufträgen und das Einsammeln der dezentral entstandenen Build-Ergebnisse auf den Master erfolgt dann automatisch.

Verteilte Builds



Abb. 5-13

*Ansicht eines Builds in
CruiseControls*

»Dashboard«-Oberfläche

Vergleich mit Hudson

Im Vergleich zu Hudson fällt zunächst auf, dass CruiseControl – der Name ist Programm – nur ein Minimum an Benutzerinteraktion benötigt und zulässt. Hudson hingegen kann zur aktiven Schaltzentrale eines Entwicklungsteams ausgebaut werden, weil es neben deutlich überlegenen Analyse- und Visualisierungsmöglichkeiten zahlreiche Schnittstellen zu Drittsystemen bietet.

*Engerer Aufgabenfokus
bei CruiseControl*

Des Weiteren ist CruiseControl weitestgehend agnostisch gegenüber den im Build eingesetzten Werkzeugen. Grundsätzlich ist dies kein Nachteil, da es maximale Flexibilität in der Auswahl der Werkzeuge

*Bessere
Werkzeugunterstützung
bei Hudson*

ermöglicht. Versteht aber ein CI-System auch den internen Aufbau eines Builds, etwa durch Auswertung einer Maven-POM-Datei, kann es eine wesentlich bessere Werkzeugunterstützung anbieten. Beispielsweise kann Hudsons Maven-Projekttyp durch Analyse von POM-Dateien automatisch Build-Abhängigkeiten zwischen Projekten aufbauen und die richtigen Artefakte im Build-Arbeitsverzeichnis auffinden.

*Verteilte Builds möglich,
aber in Hudson
komfortabler*

Sowohl CruiseControl als auch Hudson unterstützen grundsätzlich verteilte Builds. Der Unterschied zeigt sich jedoch im täglichen Betrieb. Hier fehlen CruiseControl die Funktionen zur zentralen Überwachung und dem Ausbringen der benötigten Build-Infrastruktur auf die Slave-Knoten.

*Keine Rechte und Rollen
bei CruiseControl*

Im Unternehmensumfeld dürften vor allem CruiseControls fehlendes Rechte- und Rollensystem Probleme bereiten. Zwar lässt sich der Zugriff auf die Weboberfläche mittels der Sicherheitsmechanismen des ausführenden Webcontainers einschränken. Ein feingranulares Rechtemodell ist hingegen nicht vorgesehen, etwa um die Sichtbarkeit von Projekten für einzelne Benutzer einzuschränken. Auf der anderen Seite existieren ja auch nur wenige »aktive« Funktionen, die es abzusichern gälte ...

Zukunft unklar

Der gravierendste Unterschied dürfte jedoch die deutlich abklingende Weiterentwicklung CruiseControls sein, die einem dynamischen Wachstum der Entwicklergemeinde Hudsons gegenübersteht. Aktualisierungen erscheinen für CruiseControl nur noch in monatelangen Abständen und überwiegend zur Fehlerkorrektur. Große Versionsprünge dürften bei CruiseControl in naher Zukunft nicht zu erwarten sein.

Fazit

Obwohl CruiseControl sicherlich auch weiterhin in vielen Abteilungen zuverlässig und robust seine Dienste leisten wird, sollten bei Neueinführungen dringend Alternativen geprüft werden. Soll eine Lösung aus dem Open-Source-Lager zum Einsatz kommen, wäre Hudson hier sicher ein vielversprechender Kandidat, der alles kann, was CruiseControl anbietet – und noch mehr.

5.4.3 ThoughtWorks Cruise

Viele Anwender erhofften sich 2008 mit der ersten öffentlichen Version von Cruise einen zeitgemäßen (wenngleich auch kommerziellen) Nachfolger von CruiseControl. Zwar suggeriert die sicher gewollte Namensähnlichkeit eine Verwandtschaft zwischen beiden Produkten. Der Hersteller ThoughtWorks betont jedoch ausdrücklich, dass sich

beide Produkte nur in geringem Umfang Code teilen würden. Verständlich, denn schließlich müssen bisherige CruiseControl-Anwender erst einmal überzeugt werden, von einem kostenlosen Produkt auf dessen kommerzielles Pendant zu wechseln. Dementsprechend gering sind die Marktanteile für Cruise bisher.

ThoughtWorks positioniert Cruise als ein System für Continuous Integration, das darüber hinaus Software-Deployment und Release-Management abdeckt. In der Praxis kann man sich Cruise als ein CruiseControl-System vorstellen, dem ein Workflow-System zur schrittweisen Freigabe von Builds in modernem Web-2.0-Gewand übergestülpt wurde. Die folgende Betrachtung bezieht sich auf die Version 1.3.2 (September 2009).

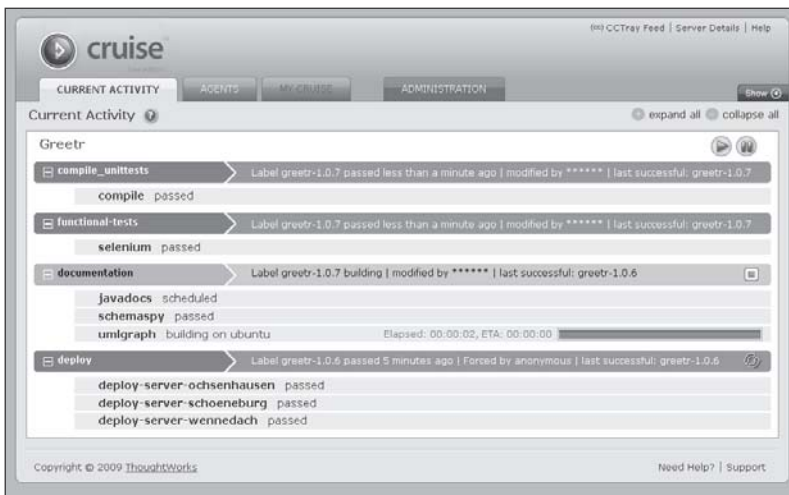


Abb. 5-14
Aktivitätsübersicht
in Cruise

Architektur und besondere Merkmale

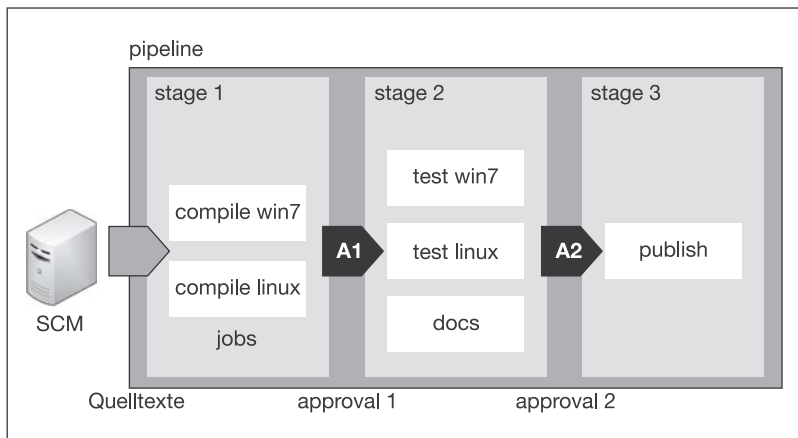
Cruise ist erhältlich für Windows, Mac OS X und Linux. Als Versionsmanagementsysteme werden Subversion, Perforce, Git und Mercurial direkt unterstützt, wenige weitere können über separat zu installierende Erweiterungen angebunden werden.

Cruise arbeitet mit einer Master/Slave-Architektur, wobei die Slave-Knoten bei Cruise als Agenten (*agents*) bezeichnet werden. Für jeden Agenten lassen sich seine charakteristischen Eigenschaften angeben, z.B. Betriebssystem, JDK-Version oder installierte Datenbanken. Werden den Build-Jobs dazu korrespondierende Anforderungen hinterlegt, führt Cruise diese auf passenden Agenten aus.

Verteilte Builds

Pipeline-Modell

Das Alleinstellungsmerkmal von Cruise dürfte das sogenannte *Pipelining*-Modell sein. Hierbei wird der Build-, Release- und Deployment-Prozess in mehrere Stufen (*stages*) unterteilt (siehe Abb. 5–15). Innerhalb dieser Stufen werden Arbeitspakete in Jobs aufgeteilt, die parallel und auf unterschiedlichen Agenten ausgeführt werden können. Die Pipeline kann durch Änderungen im Versionsmanagementsystem, durch den Abschluss anderer Pipelines oder manuell gestartet werden. Dabei werden die aktuellen Quelltexte aus dem Versionsmanagementsystem den Jobs der ersten Stufe zugeführt. Wurden alle Jobs einer Stufe abgearbeitet, muss eine Freigabe (*approval*) für die nächste Stufe erteilt werden. Dies geschieht automatisch, in Abhängigkeit von Testergebnissen oder durch manuelle Freigabe, z.B. durch den QA-Manager. Als letzte Stufe kann die Veröffentlichung bzw. die Ausbringung des Produkts in den Produktionsbetrieb stehen. Dieser mehrstufige Freigabeprozess, der von einer Änderung im Versionsmanagementsystem bis hin zum Produktionsbetrieb durchläuft, stellt den größten Mehrwert von Cruise gegenüber anderen CI-Servern dar.

Abb. 5–15*Pipelining in Cruise**XML-Konfiguration*

Cruise bietet eine aufgeräumte Weboberfläche mit zeitgemäßer Optik (Abb. 5–14). Leider beschränkt sie sich auf die Darstellung der Pipelines und Stufen. Die Konfiguration des Systems erfolgt ausschließlich über XML-Fragmente, die in großen Textfeldern editiert werden. Echtzeitvalidierung der Eingaben, Syntaxhervorhebung oder automatische Codevervollständigung fehlen.

Rechte und Rollen

Über eine Anbindung an LDAP bzw. Active Directory lässt sich der Zugriff auf Cruise gezielt einschränken.

Lizenzmodell

Cruise ist in zwei Editionen erhältlich: Cruise Free und Cruise Professional. Für Erstere kann eine kostenlose Jahreslizenz bei ThoughtWorks angefordert werden. Cruise Free ist auf 10 Benutzer (Personen,

die im Versionsmanagement Änderungen einchecken) und 10 lokale Agenten eingeschränkt. An verschiedenen Stellen der Produkt-Website wird diese Edition auch als Cruise Trial bezeichnet, was die Intention des Herstellers wohl besser darlegen dürfte. Für höhere Anforderungen ist Cruise Professional erforderlich. Lizenzen kosten je nach Anzahl der Benutzer und Agenten ab 3.200 USD pro Jahr.

Vergleich mit Hudson

Dem Anwender stellt sich Cruise als CruiseControl plus Freigabe-Workflow mit Weboberfläche dar. Unverständlich ist dabei, dass in der Teildisziplin »Continuous Integration« Cruise leider sogar dem »CI-Opa« CruiseControl unterliegt. So fehlen beispielsweise grundlegende Dinge wie eine Möglichkeit zur Zeitplanung von Builds: Momentan werden alle überwachten Versionsmanagementsysteme minütlich abgefragt. Für den Unternehmenseinsatz ist dies nicht ausreichend flexibel.

*CI-Bereich noch
schwächer als
CruiseControl*

Ist man im ersten Moment von der aufgeräumten Weboberfläche angetan, erstaunt die Beibehaltung des gusseisernen XML-Konfigurationskonzeptes aus CruiseControl-Tagen. Es verwundert, dass ThoughtWorks auf der einen Seite die Oberfläche als »usability-driven« bezeichnet, auf der anderen Seite aber den Hauptkritikpunkt an CruiseControl, die ausschließliche Konfiguration per XML ohne wirkliche Hilfsmittel, ins Herz des kommerziellen Nachfolgers pflanzt.

XML-Konfiguration

Die Dokumentation zu Cruise behandelt zwar alle Masken und Eingabefelder, bietet aber wenig technische Hintergrundinformation. Diese finden sich im – schwach frequentierten – Onlineforum. Ein Entwicklerprogramm oder eine Plugin-API existieren momentan nicht, was den Ausbau und die Weiterentwicklung des gesamten Produktes exklusiv auf den Schultern von ThoughtWorks lasten lässt. Es ist fraglich, ob sich mit diesem Modell die Werkzeugvielfalt in Build-Ketten abdecken lässt. Sogar für bereits etablierte Build-Werkzeuge, wie etwa Maven, fehlen intelligente Integrationen. So kann etwa ein Maven-Build momentan lediglich als externe Kommandozeilenanwendung aufgerufen werden. Weitergehende Funktionen, wie etwa die automatische Konfiguration eines Jobs durch Auswertung einer POM-Datei, sind nicht verfügbar.

Keine Plugin-API

Fazit

Ist Cruise also der legitime Nachfolger von CruiseControl? Nein. So interessant der Pipeline-Workflow in Cruise sein mag: Die CI-Grundfunktionen sind momentan sogar noch schwächer als die des wesentlich älteren CruiseControls. Wer bei CruiseControl Wert auf die kostenlose Verfügbarkeit gelegt hat, wird sich daher eher Hudson

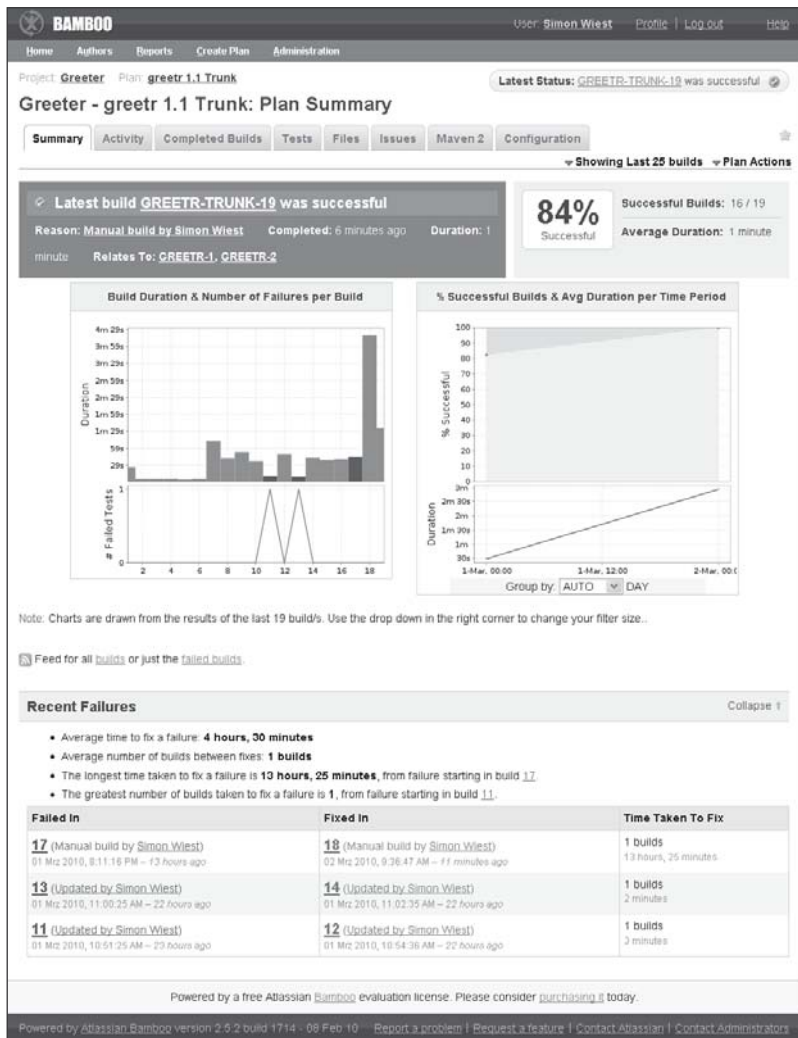
zuwenden. Wenn kommerzielle Vertreter ebenfalls in Betracht gezogen werden dürfen, gibt es aber auch gute Gründe, die Produkte Atlassian Bamboo und JetBrains TeamCity näher zu betrachten. Dies werden wir in den folgenden zwei Abschnitten tun.

5.4.4 Atlassian Bamboo

Die Firma Atlassian ist vielen Anwendern der IT-Branche als Hersteller des Enterprise-Wikis Confluence und des Issue-Trackers JIRA ein Begriff. In den letzten Jahren konnte Atlassian sein Portfolio rund um Softwareentwicklung und webbasierte Zusammenarbeit stetig erweitern, so auch 2007 mit dem CI-Server Bamboo. Die folgende Betrachtung bezieht sich auf Version 2.5.2 (Februar 2010).

Abb. 5-16

Ansicht eines Build-Plans
in Bamboo



Architektur und besondere Merkmale

Bamboo ist eine Java-Webapplikation, die entweder im mitgelieferten Jetty-Server betrieben (*standalone distribution*) oder in einen existierenden Tomcat-Server ausgebracht (*EAR-WAR distribution*) wird. Als Serverbetriebssysteme werden Microsoft Windows, Linux/Solaris und Apple Mac OS X offiziell unterstützt. Bamboo benötigt ein Java Development Kit (kein JRE) ab Version 1.5 und setzt eine Datenbank zur Speicherung seiner Daten voraus. Offiziell werden MySQL 5.x, PostgreSQL ab Version 8.2, Microsoft SQL Server 2005/2008, Oracle 10g/11g und HSQLDB (zu Testzwecken) unterstützt.

Java-Webapplikation

Bamboo arbeitet mit einer verteilten Architektur, bei der ein zentraler Server angelegte Projekte mit Build-Plänen (*build plans*) verwaltet und Build-Aufträge zur Ausführung an Build-Agenten (*build agents*) delegiert. Abschließend werden die Build-Ergebnisse der Agenten wieder eingesammelt, auf dem Server archiviert und über die Weboberfläche visualisiert. Für alle Agenten werden deren Fähigkeiten (*capabilities*) konfiguriert, etwa installierte JDKs, Betriebssystemversionen oder Build-Werkzeuge (z.B. Ant, Maven, Bash-Shell). Pro Build-Plan werden Anforderungen (*requirements*) an die Agenten deklariert, etwa »Dieser Plan benötigt JDK 1.6 auf einem Windows-Rechner mit Oracle 11g mindestens 2 CPUs«. Durch einen Abgleich der Fähigkeiten der Agenten und den Anforderungen eines Build-Plans entscheidet der Server, auf welchem Agenten ein Build zur Ausführung kommt.

Master-Slave-Architektur

Bei den Build-Agenten wird zwischen lokalen Agenten (*local agents*) unterschieden, die zusammen mit dem Server in der gleichen JVM ausgeführt werden, und entfernten Agenten (*remote agents*), die autonom laufen, in der Regel auf einer anderen Hardware als der Server. Entfernte Agenten sind Java-Anwendungen, die als JAR-Dateien entweder manuell gestartet werden oder durch einen betriebssystemabhängigen Java Service Wrapper gestartet und überwacht werden (*remote agent supervisor*).

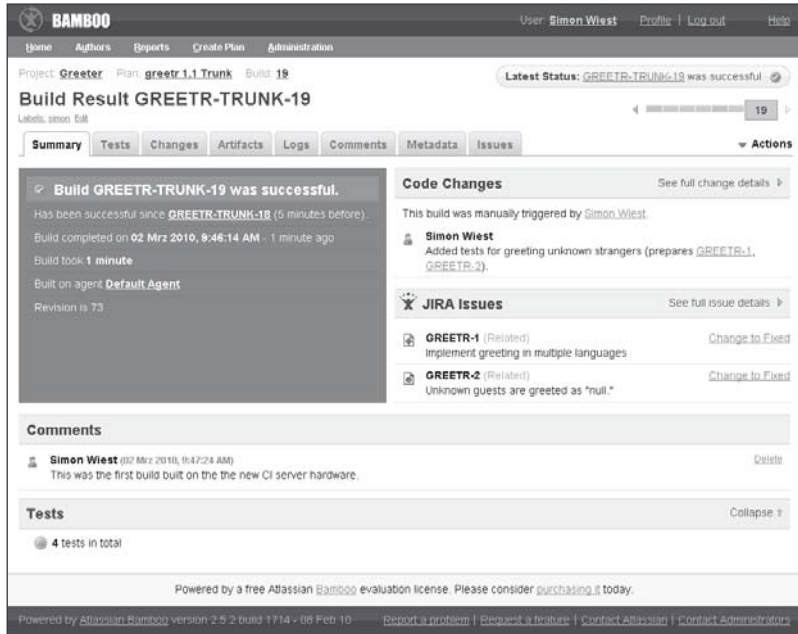
Bamboo kann über Plugins erweitert werden. Zum Zeitpunkt der Drucklegung dieses Buches waren rund 35 Plugins verfügbar, von denen der überwiegende Teil allerdings (noch) nicht mit der aktuellen Bamboo-Version kompatibel war.

Plugin-Konzept

Die Benutzeroberfläche folgt weitestgehend den Atlassian-typischen Konventionen, die aus den anderen Produkten des Herstellers bekannt sind (Abb. 5-16, Abb. 5-17). Zusätzlich zur interaktiven Oberfläche steht auch eine REST-Schnittstelle zur Verfügung, mit der die wichtigsten Informationen abgefragt werden können, z.B. eine Liste aller angelegten Projekte, der Ausgang eines Builds oder die erzeugten Artefakte eines Builds.

Benutzeroberfläche

Abb. 5-17
Ansicht eines Builds in
Bamboo



Benutzerverwaltung

Bamboo kommt mit einer eingebauten, autarken Benutzerverwaltung, kann jedoch auch an einen externen LDAP-Server angebunden werden. Bei komplizierteren Anforderungen lassen sich über das Single-Sign-On-Produkt Atlassian Crowd weitere Benutzerverzeichnisse und Technologien verwenden, z.B. Microsofts Active Directory. Rechte und Rollen lassen sich feingranular und übersichtlich zuordnen.

Atlassian sieht Bamboos Stärken in folgenden vier Bereichen liegen:

- *Schnelles Einrichten neuer Build-Pläne:*

Gängige Infrastrukturen (etwa Subversion in Kombination mit Maven, JUnit und E-Mail-Benachrichtigungen) werden direkt unterstützt und können assistentengeführt schnell und einfach angelegt werden. Build-Pläne für unterschiedliche Entwicklungszweige (z.B. trunk, release candidate, maintenance) werden in Projekten zusammengefasst, was die Konfiguration beschleunigt und die Übersichtlichkeit erhöht.

- *Enge Integration mit anderen Atlassian-Produkten:*

Bamboo fühlt sich nicht nur so an wie die anderen Produkte aus der Atlassian-Familie, es verwendet auch intern die gleichen Technologien, wie zum Beispiel Schnittstellen zu Atlassian Crowd. Darüber hinaus lassen sich die Produkte auch untereinander verknüpfen, so dass beispielsweise ein JIRA-Ticket automatisch einen

Verweis auf zugehörige Bamboo-Builds erhält und von dort wiederum auf eine Darstellung der zugehörigen Codeänderungen in Fisheye gesprungen werden kann.

■ *Hohe Skalierbarkeit:*

Bamboos verteilte Architektur bewältigt auch umfangreiche Build-Aufkommen einer Abteilung oder eines ganzen Unternehmens. Zur Abdeckung von großen Spitzenlasten können entfernte Agenten auch in elastischen Instanzen (*elastic instance*) in der Amazon EC2 Cloud nach Bedarf gestartet werden. Bei diesen Instanzen handelt es sich um von Atlassian vorkonfigurierte Systeme, die jeweils einen entfernten Agenten enthalten. Dieser Agent wird nach Hochfahren der Instanz aktiv und steht dann dem Bamboo-Server für Builds zur Verfügung.

■ *Berichte und Visualisierungen:*

Bamboo bietet zahlreiche Kennzahlen und Auswertungen auf unterschiedlichsten Ebenen, etwa Laufzeiten per Build, JUnit-Testergebnisse per Java-Package, Builds per Committer usw. Die Berichte werden als Text oder als Diagramm dargestellt.

Bamboos Lizenzmodell orientiert sich an der Anzahl der Build-Agenten. Werden keine entfernten Agenten und nur bis zu 10 Build-Pläne benötigt, ist Bamboo mit Lizenzgebühren von 10 USD praktisch kostenlos. Kommen hingegen entfernte Agenten zum Einsatz, steigen die Gebühren (ab 2.000 USD). Die Anzahl der Build-Pläne ist in diesen Fällen unbegrenzt. Open-Source-Projekte können kostenlose Lizenzen beantragen.

Lizenzmodell

Vergleich mit Hudson

Arbeitet man in einer Mainstream-Infrastruktur (z.B. mit Subversion, Ant/Maven, JUnit und E-Mail-Benachrichtigungen) und benötigt vor allem die grundlegenden CI-Funktionen wie automatisches Bauen von Projekten nach Codeänderungen, so ist Bamboo ein unaufgeregter und zuverlässiger Begleiter, der sich gut in eine bestehende Atlassian-Landschaft integriert.

Sollen jedoch auch neuere, noch weniger verbreitete Werkzeuge angebunden oder Berichte von spezielleren Testframeworks visualisiert werden, hat Hudson dank seiner Plugin-Vielfalt klar die Nase vorn. Bamboo verfügt zwar ebenfalls über eine Plugin-Schnittstelle, die es erlaubt, eigene Plugins zu entwickeln – Hudsons Plugin-Szene erscheint aber um ein Vielfaches vitaler. Der überwiegende Teil der Bamboo-Plugins war zudem zum Zeitpunkt der Drucklegung dieses Buches mit der aktuellen Version inkompatibel und wirkte verwaist. Ein Grund dafür

Plugins stiefmütterlich behandelt

dürfte das unübersichtliche Geflecht aus Wiki-Seiten der Online-Dokumentation zur Bamboo und dessen Plugins sein, bei denen ein Einsteiger leicht den Überblick verliert. Hier fehlt Bamboo (noch) eine eingebaute Plugin-Verwaltung nach dem Vorbild des »großen Bruders« Confluence, in der verfügbare und installierte Plugins inklusive Versionsnummern angezeigt und konfiguriert werden können.

*Maven-Integration gut,
in Hudson besser*

Die Maven-Integration ist mit Bamboos Version 2.5 wesentlich verbessert worden. So können neue Build-Pläne durch Auswerten einer vorhandenen POM-Datei deutlich schneller angelegt werden. Hudsons Maven-Unterstützung ist in dieser Hinsicht jedoch noch einen Schritt weiter entwickelt, z.B. durch die Multi-Modul-Darstellung oder dem automatischen Erkennen abhängiger Maven-Projekte.

Verteilte Builds

Bamboos Konzept des Abgleichs von Fähigkeiten der Agenten einerseits und Anforderungen der Build-Pläne andererseits ist wesentlich eingängiger als Hudsons Label-Konzept. Ähnlich wie Hudson unterstützt auch Bamboo Builds in der Amazon EC2 Cloud. Bamboos vorkonfigurierte Instanzen funktionieren tatsächlich ohne viel Konfigurationsaufwand. Allerdings stellen sich im Alltag prompt die kleinen, aber lästigen praktischen Probleme ein: So muss beispielsweise der Agent in der elastischen Instanz über das Internet Zugriff auf das Versionsverwaltungssystem haben, von dem er Code auschecken soll. Hier zögern Administratoren, diese firmeninterne Schatzkiste zum Internet hin zu öffnen. Sicherlich sind alle diese Probleme technisch lösbar. Die Euphorie um Build-Skalierung in der Wolke »mit ein paar Mausclicks« wird dadurch allerdings deutlich gedämpft.

*Verteilung der
Build-Werkzeuge*

Eine weitere Herausforderung verteilter Builds stellt die Verteilung der benötigten Build-Werkzeuge dar. Schließlich werden zum Bauen eines Projekts nicht nur die Quelltexte, sondern auch die passenden Werkzeuge benötigt, etwa Compiler und spezialisierte Development Kits. Bamboo setzt voraus, dass diese durch einen Administrator bereits auf den Systemen der entfernten Agenten eingerichtet und in Bamboo eingetragen wurden. Hudson kann hingegen aktiv die zentrale Verteilung von Build-Werkzeugen an Slave-Knoten übernehmen und sogar JDKs sowie Ant- und Maven-Installationen in unterschiedlichen Versionen direkt aus dem Internet auf Slave-Knoten installieren. Wie stark man davon profitieren kann, hängt natürlich davon ab, wie viele Knoten man verwalten muss und wie häufig sich Änderungen in der Konfiguration ergeben.

Fazit

Arbeitet man in einer Infrastruktur, die ausschließlich gängige Werkzeuge einsetzt, und hat man vielleicht sogar schon das eine oder andere Atlassian-Produkt im Haus, dann kann Bamboo der ideale fehlende Mosaikstein für das Aufgabengebiet »Continuous Integration« sein. Möchte man hingegen auch neueste Technologien anbinden oder gar eigene Plugins für hausinterne Werkzeuge schreiben, so punktet Hudson hier mit ungleich mehr Beispielen und Material im Internet und vor allem einer deutlich vitaleren Entwicklergemeinschaft.

5.4.5 JetBrains TeamCity

Das vierte Produkt, auf das hier eingegangen werden soll, ist TeamCity von JetBrains. Es hat einen ähnlichen Hintergrund wie Atlassians Bamboo: Auch TeamCity wird von einem Werkzeugspezialisten hergestellt, der sich durch seine Produkte einen sehr guten Namen unter Entwicklern erarbeitet hat – in diesem Falle durch die Entwicklungsumgebung IntelliJ IDEA und das Visual Studio Add-In ReSharper. Analog zu den Mehrwerten, die Atlassian durch die Integration von Bamboo mit seinen Flaggschiff-Produkten zu schaffen versucht, sind bei JetBrains interessante Ansätze im Zusammenspiel von TeamCity mit interaktiven Entwicklungsumgebungen (IDEs) zu erwarten. Die folgende Betrachtung bezieht sich auf die Version 5.1 (April 2010).

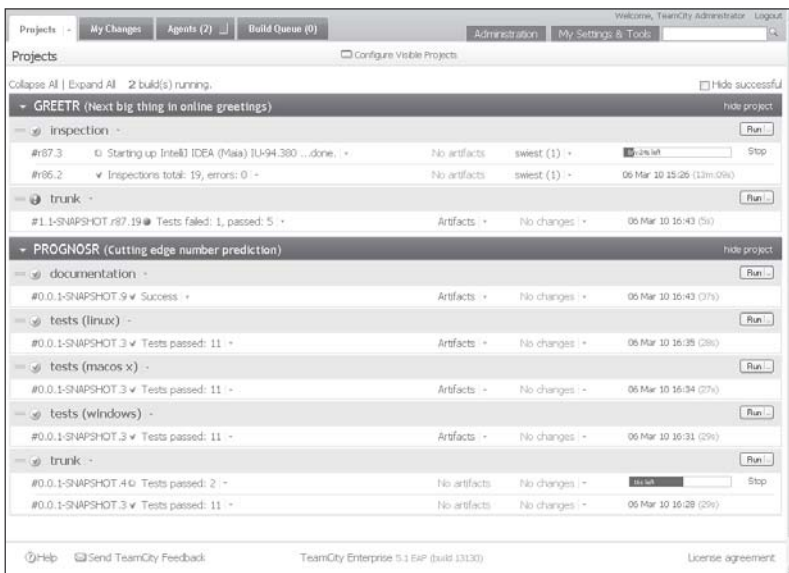


Abb. 5-18

Projektübersicht
in TeamCity

Architektur und besondere Merkmale

Java-Webapplikation

Wie alle Produkte, die wir bisher betrachtet haben, ist auch TeamCity als Java-Webapplikation realisiert. TeamCity kann in einen bestehenden J2EE-Container ausgebracht oder aber über den mitgelieferten Tomcat-Server betrieben werden. Die Betriebssysteme Windows, Mac OS X und Linux werden explizit unterstützt. TeamCity verwendet zur Datenspeicherung neben einem Datenverzeichnis auch eine relationale Datenbank. Für Testzwecke ist die mitgelieferte HSQLDB-Datenbank ausreichend, für Produktionseinsatz sollte aber besser eine externe Datenbank eingesetzt werden. Unterstützt werden hier zurzeit MySQL, PostgreSQL, Oracle, Microsoft SQL Server und Sybase.

Projekte und Build-Konfigurationen

TeamCity erlaubt die Gruppierung von sogenannten Build-Konfigurationen (dies entspricht den Jobs in Hudson bzw. den Build-Plänen in Bamboo) zu Projekten. Build-Konfigurationen eines Projektes teilen sich Pfade ins Versionsmanagementsystem und die Zuweisung von Benutzerrechten.

Verteiltes Bauen

Auch TeamCity arbeitet mit einer Master-Slave-Architektur, bei der ein zentraler Server Build-Aufträge an Agenten verteilt, die auf unterschiedlicher Hardware betrieben werden können. Zur Überwachung eines solchen *build grids* bietet TeamCity spezielle Ansichten, auf denen sich die Auslastung der einzelnen Agenten im Zeitverlauf anzeigen lassen. Zusätzlich ermittelt TeamCity für jeden Agenten einen Geschwindigkeitsindex. Eilige Builds lassen sich beim Start interaktiv besonders schnellen Knoten zuordnen. Für umfangreiche Builds bietet TeamCity eine Integration mit Amazons EC2 Cloud. Anders als bei Bamboo wird allerdings kein vorkonfigurierter Rechner in Form eines Amazon-EC2-Rechnerabbilds (*Amazon Machine Image, AMI*) bereitgestellt. Dieses muss durch den Build-Manager vorbereitet und im Vorfeld auf den Amazon-Servern abgelegt werden. In der Praxis wird man dieses Merkmal aber sowieso nur mit individuell angepassten Rechnerabbildern nutzen wollen, so dass dieser Umstand in langfristigen betriebenen Installationen keinen wirklichen Nachteil darstellt.

Private Builds

TeamCitys wichtigste Alleinstellungsmerkmale dürften die privaten Builds (*private builds* bzw. *remote run*) und der vorgetestete Commit (*pre-tested commit*) sein. Private Builds erlauben es dem einzelnen Entwickler, seine Codeänderungen auf dem CI-Server probeweise zu bauen, und zwar ohne diese zuvor ins Versionsmanagement übernehmen zu müssen. Dies hat dreierlei Vorteile: Erstens wird dadurch der Arbeitsplatzrechner des Entwicklers entlastet, da der Build auf einer anderen Hardware ausgeführt wird. Der Entwickler kann also parallel seinen Rechner weiter nutzen. Zum zweiten werden die Änderungen in der Infrastruktur des CI-Servers getestet, die umfangreicher sein kann

als die eines Arbeitsplatzrechners (z.B. weitere Datenbanksysteme oder Applikationsserver). Drittens hat ein privater Build aus psychologischer Sicht einen unverbindlicheren Charakter, was zu häufigeren Builds und damit früherer Fehlererkennung animieren kann.

Technisch ist ein privater Build dadurch realisiert, dass vom Arbeitsplatzrechner nur die Codedifferenzen zum letzten versionierten Stand auf den CI-Server übertragen und dort vor dem Build auf eine ausgecheckte Kopie gepatcht werden. Dies erfordert ein gutes Zusammenspiel von IDE, Versionsmanagementsystem und CI-Server und erklärt, warum dieses Merkmal nur für ausgewählte Versionsmanagementsysteme (CVS, Subversion, Perforce) und ausgewählte IDEs (IntelliJ IDEA, Eclipse, Visual Studio) verfügbar ist. Mit TeamCity 5.1 besteht darüber hinaus erstmals die Möglichkeit, private Builds per Kommandozeile, also außerhalb der genannten IDEs, anzustoßen.

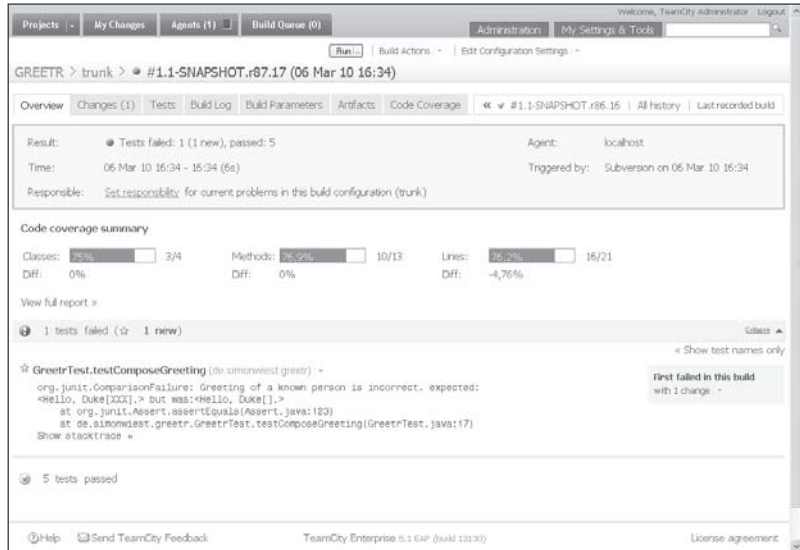
Vorgetestete Commits erweitern private Builds, indem Codeänderungen erst nach einem erfolgreichen Build automatisch ins Versionsmanagementsystem übernommen werden. Dadurch soll sichergestellt werden, dass nur fehlerfreier Code (im Sinne der automatischen Testmöglichkeiten) eingchecked werden kann und es zu keinen Blockaden ganzer Arbeitsgruppen kommt, nur weil ein Kollege versehentlich unfertigen Code eingchecked hatte (kurz vor Abreise in einen 14-tägigen Urlaub auf einer abgelegenen Berghütte – Sie kennen das).

Vorgetesteter Commit

Wie bereits erwähnt, ist von JetBrains als Hersteller einer bekannten Entwicklungsumgebung eine besonders enge Integration zwischen CI-Server und IDE zu erwarten. JetBrains berücksichtigt hier nicht nur das Produkt aus dem eigenen Hause, IntelliJ IDEA, sondern auch Eclipse und Visual Studio. Die Integration umfasst beispielsweise das Starten der bereits angesprochenen privaten Builds und vorgetesteten Commits, das Abfragen des Zustands überwachter Projekte oder das Verfolgen von Builds in Echtzeit. Darüber hinaus kann nicht nur von der IDE auf die Weboberfläche des CI-Servers navigiert werden. Auch umgekehrt kann in der Weboberfläche eine Quelltextdatei oder ein fehlgeschlagener JUnit-Test ausgewählt und dadurch die korrespondierende Stelle in der IDE geöffnet werden. Technisch wird dies übrigens durch einen schlanken Server ermöglicht, der in den IDE-Plugins eingebettet läuft und Kommandos vom CI-Server entgegennimmt.

IDE-Integration

Abb. 5-19
Ansicht eines Builds in
TeamCity



JUnit-Unterstützung

JUnit-Tests werden von TeamCity auf besondere Weise unterstützt: Zum einen werden JUnit-Tests bereits während des Builds in Echtzeit überwacht und ausgewertet. Der Entwickler kann somit bereits auf erste fehlgeschlagene Tests reagieren, während der Build noch läuft. Des Weiteren kann TeamCity die Ausführungsreihenfolge der JUnit-Tests beeinflussen und somit beispielsweise fehlgeschlagene Tests des vorausgegangenen Builds zuerst starten.

Inspektionen und Coverage-Ermittlung

Zwei interessante Technologien, die JetBrains vom Produkt IDEA übernommen hat, sind zum einen die sogenannten Inspektionen, die mögliche Schwachstellen im Code aufgrund statischer Analyse finden sollen (vergleichbar zu Checkstyle, PMD und FindBugs). Zum anderen bringt TeamCity eine Funktion zur Ermittlung und Visualisierung der Codeabdeckung (*code coverage*) mit (vergleichbar zu Cobertura, EMMA oder Clover).

Rollen und Rechte

Das Rechte-Management ist sehr umfangreich implementiert, inklusive einer LDAP-Anbindung. Globale und projektbezogene Rechte lassen sich einzelnen Personen und Gruppen zuweisen. Manche Arbeitsgruppen würden sich jedoch die Möglichkeit wünschen, noch feiner, also auf Build-Konfigurationsebene, Rechte vergeben zu können.

Plugins

TeamCity lässt sich über Plugins erweitern. Zurzeit sind rund 25 Plugins verfügbar, davon etwa die Hälfte vom Hersteller JetBrains entwickelt und als Open Source gestiftet. Ähnlich wie bei Atlassian's Bamboo scheint die Plugin-Szene keine maßgebliche Säule im Produktkonzept zu sein. Vielleicht sind aber auch die meisten Anwender mit dem Leistungsumfang »aus der Box« einfach zufrieden.

JetBrains bietet zwei unterschiedliche Editionen an: Die kostenlose Professional-Edition ist auf 3 Build-Agenten, 20 Benutzer und 20 Build-Konfigurationen beschränkt. Außerdem steht nur die eingebaute Benutzerverwaltung zur Verfügung, die LDAP-Anbindung fehlt. Umfangreichere Installationen erfordern eine Enterprise-Lizenz (1720 EUR). Weitere Build-Agenten können für 258 EUR pro Agent hinzugekauft werden. Open-Source-Projekte können kostenlose Lizenzen beantragen.

Vergleich mit Hudson

Mit privaten Builds und den vorgetesteten Commits verfügt TeamCity über ein klares Alleinstellungsmerkmal gegenüber anderen Mitbewerbern, auch gegenüber Hudson. In der Hudson-Entwicklergemeinde wird schon seit Längerem über die Implementierung eines vergleichbaren Features nachgedacht. Erste Ansätze unterstützen aber bisher nur verteilte Versionsmanagementsysteme wie Git. Hier hat JetBrains ganz klar die Nase vorn, indem es über Kompetenz für die komplette Kette von der IDE über das Versionsmanagement bis hin zum CI-Server verfügt.

Die IDE-Integration ist erwartungsgemäß sehr viel weitgehender als die der momentan verfügbaren Hudson-Plugins für Eclipse oder IntelliJ IDEA. Sieht man allerdings von den privaten Builds ab, haben viele Entwickler in der Praxis auch kein Problem damit, im ohnehin geöffneten Webbrowser die Oberfläche des CI-Servers parallel mitlaufen zu lassen.

TeamCitys Weboberfläche wirkt auf den ersten Blick nüchtern monochrom und »tabellenlastig«, erweist sich im Detail aber als gut durchdacht (Abb. 5–18, Abb. 5–19). Viel »Intelligenz« ist beispielsweise in dynamisch per AJAX aufgebauten Menüs eingebaut. Eine einfache Visualisierung der wichtigsten Werkzeuge wie Checkstyle, PMD oder FindBugs ist vorhanden. Momentan reicht diese im Umfang jedoch nicht an die entsprechenden Visualisierungsplugins heran, die für Hudson verfügbar sind.

Die Verwaltung der Agenten für verteilte Builds wird bei TeamCity durch sinnvolle und übersichtliche Visualisierungen unterstützt. Werkzeuge zum zentralen Ausbringen einer Build-Umgebung (JDK, Ant, Maven) wie bei Hudson fehlen aber. Es wird bei TeamCity vorausgesetzt, dass alle beteiligten Rechner zuvor durch einen Administrator eingerichtet wurden und der Build-Agent gestartet wurde.

Obwohl TeamCity durchaus ein Plugin-Konzept und sogar brauchbare Entwicklerdokumentation mitbringt, scheint die Plugin-Szene leider (noch) nicht recht in Schwung gekommen zu sein.

Fazit

TeamCity hat in der aktuellen Version einen Reifegrad erreicht, der für die meisten Mainstream-Umgebungen »aus der Box« sehr gut funktionieren sollte. Die kostenlose Professional-Edition ist im Leistungsumfang zwar eingeschränkt, ist aber deutlich mehr als nur eine »Demo-version«. Sie dürfte für kleinere Arbeitsgruppen durchaus produktiv einsetzbar sein.

Soll das CI-System hingegen mit den neuesten Werkzeugen oder proprietärer Infrastruktur kombiniert werden, bietet Hudson durch sein gelebtes Plugin-Konzept dafür zahlreiche einsatzbereite Erweiterungen. Es liefert so auch eine Fülle an Vorlagen für eigene Entwicklungen. Dies spricht überraschenderweise nicht nur notorisch bastelfreudige Informatikstudierende an. Vielmehr weckt dies auch das Interesse gerade großer Unternehmen: Hier wird Hudsons Offenheit geschätzt, wenn der Build-Prozess aus technischen, rechtlichen oder organisatorischen Gründen um individuelle Funktionalität erweitert werden muss. Hudson wird in diesem Umfeld weniger als Produkt mit festgelegtem Funktionsumfang verstanden, sondern als eine Plattform für Build-Automatisierung gesehen.

5.5 Zusammenfassung

In diesem Kapitel haben Sie die prinzipielle Funktionsweise von Hudson sowie ausgewählte Highlights kennengelernt. Darüber hinaus haben wir die prominenteren Alternativen beleuchtet, um den Überblick abzurunden.

Wenn Sie inzwischen Appetit auf den konkreten Einsatz von Hudson bekommen haben: Ausgezeichnet! Im nächsten Kapitel beginnen wir mit der Installation.