trivadis
Part of Accenture

Test-Coverage für Grails

AUTOR

Sven Lange
Orientation in Objects GmbH

Veröffentlicht am: 31.1.2009

TEST-COVERAGE FÜR GRAILS

Die Ermittlung der Testabdeckung kann ein wichtiges Instrument sein, um die Qualität einer Anwendung zu überwachen und zu steigern. In diesem Artikel wird gezeigt, wie die Testabdeckung einer Grails Anwendungen ermittelt und über einen Continuous Integration Server bereitgestellt werden kann. Die Abdeckung soll aus Komponenten-, Integrations- und funktionalen Tests resultieren.

) Schulung)

) Beratung)

) Entwicklung)

) Artikel)

Trivadis Germany GmbH

Weinheimer Str. 68
D-68309 Mannheim

Tel. +49 (0) 6 21 - 7 18 39 - 0
Fax +49 (0) 6 21 - 7 18 39 - 50

www.oio.dekontakt@trivadis.com

EINLEITUNG

Grails ist ein Web-Framework, welches den Entwickler unterstützt in kurzer Zeit produktive Webanwendungen zu erstellen. Dies wird dadurch erreicht, dass Grails alle Architekturelemente und Konfigurationen einer Java EE-Webanwendung bereits mitbringt und der Entwickler sich sofort der Entwicklung der eigentlichen Anwendungsfunktionalität, ohne aufwendige Konfigurationen oder Installationen vornehmen zu müssen, widmen kann. Neben dem leichtgewichtigen und standardisierten Architekturansatz kommt mit Groovy, eine dynamische und für die Java Virtual Machine (JVM) entwickelte Skriptsprache, zum Einsatz die das Entwickeln fachlicher Funktionalität erleichtert.

Damit neben der hohen Produktivität die Qualität der Anwendung nicht auf der Strecke bleibt, ist es wichtig die Anwendung ausgiebig zu testen. Auch fordert der dynamische Charakter von Groovy ausführlicheres Testen [1]. Eine große Hilfe um zu sehen wie umfangreich Tests eine Anwendung prüfen, ist die Ermittlung der Testabdeckung. Anhand der Berichte ist ein Entwickler in der Lage zu sehen welche Teile des Quellcodes ungetestet oder nicht ausreichend getestet sind.

In diesem Artikel werden die nötigen Arbeitsschritte gezeigt, um auf Basis von Grails 1.0.4 die Testabdeckung einer Grails Anwendung zu ermitteln, bei der Integrations-, Komponenten- und funktionale Tests ausgeführt werden. Weiterhin sollen die Ergebnisse über einen Continuous Integration Server (CI Server) bereitgestellt werden.

GRAILS UND GROOVY

Inspiriert von Ruby on Rails [2] stellt Grails ein Framework dar, dass es ermöglicht komplette Web-Anwendungen in kurzer Zeit zu erstellen. Der große Vorteil von Grails ist, dass es auf der Java Plattform basiert und somit auf einen großen Schatz von ausgereiften Java EE Frameworks und Java Technologien zurückgreifen kann.

Grails kommt mit einer Vielzahl von Gant-Skripten [3], die viele Vorgänge, wie bspw. das Deployment oder das Ausführen der Anwendung, automatisieren und somit dem Entwickler helfen, sich auf die Entwicklung der eigentlichen Anwendung zu fokussieren. Sie werden meist von der Konsole aus gestartet, können allerdings auch über einen CI Server oder aus einer IDE heraus aufgerufen werden. Man spricht in diesem Zusammenhang auch von Grails Kommandos.

Weiterhin verfügt Grails über ein sehr flexibles Plug-In System. Damit ist man in der Lage die eigene Anwendung zu modularisieren, oder benötigte Funktionalität über das breite Angebot von freien Plug-ins zu erweitern.

Produktivität wird bei Grails groß geschrieben. Daher kommt auch die dynamische Skriptsprache Groovy zum Einsatz. Sie erweitert die Sprache Java um viele nützliche Funktionen und versucht das Entwickeln durch eine mächtigere Ausdrucksmöglichkeit zu vereinfachen. Auch dient die Sprache dazu die verschiedenen Frameworks die Grails beinhaltet zu vereinen und den Umgang mit ihnen zu vereinfachen.

Allerdings haben dynamische Sprachen den Nachteil, dass beim Kompilieren weniger Fehler entdeckt werden können und somit erst zur Laufzeit auffallen. Daher ist es sehr wichtig, dass deren Code ausgiebig getestet wird.

TESTEN EINER GRAILS ANWENDUNG

Grails bietet von Haus aus die Möglichkeit eine Anwendung mittels Komponenten- und Integrationstests zu testen. Dabei kommt JUnit zum Einsatz, welches dank Groovy um weitere nützliche Funktionen erweitert wurde. Über die Konsole können mittels der Grails Kommandos grails *create-integration-test* und *grails create-unit-test* neue Tests angelegt werden. Ausgeführt werden können diese über das Kommando *grails test-app*. Die Ergebnisse des Testdurchlaufes werden automatisch in einem Bericht zusammengefasst.

Komponenten- und Integrationstests unterscheiden sich darin, dass nur die Integrationstests auf die Funktionen des Grails-Frameworks zurückgreifen können. Damit lassen sich Integrationstests einfacher schreiben und schränken den Entwickler weniger ein. Allerdings laufen sie erheblich langsamer als einfache Komponententest. Daher sollte man das Erstellen von Komponententests bevorzugen.

Den Nachteil der Eingeschränktheit der Komponententests soll das zukünftige Testing Plugin beheben, welches Bestandteil des Grails Cores ab Version 1.1 sein wird.

Möchte man funktionale Tests ausführen muss man eines der dazu zur Verfügung stehenden Plug-Ins installieren. In diesem Artikel werden wir das Webtest Plugin in der Version 0.5.1 verwenden, welches auf Canoo Webtest basiert. Damit ist es möglich die Anwendung über das Frontend browserunabhängig zu testen. Auch Webtest produziert einen Bericht der die Testergebnisse nachvollziehbar macht.

In diesem Artikel werden die drei soeben aufgeführten Testarten genutzt, um die Anwendung zu testen.

TESTABDECKUNG

Der Bericht einer Testabdeckungsmessung gibt aufschlussreiche Informationen darüber, welche Zeilen des Quellcodes durch die Tests nicht erfasst wurden. Damit ist man in der Lage ausführliche Tests zu schreiben, was der Qualität der Anwendung zu Gute kommt. Je höher die Testabdeckung, desto besser. Allerdings muss man sich im Klaren sein, dass auch eine 100% Abdeckung aller Quellcodezeilen nicht für Fehlerfreiheit garantiert. Sie sagt nur aus, dass alle Zeilen durchlaufen wurden.

Die Testabdeckung kann in Grails mit dem Cobertura Plug-In ermittelt werden [4]. Cobertura ist ein Werkzeug das dazu dient die Abdeckung zu messen. Dies geschieht indem der kompilierte Quellcode durch Cobertura instrumentiert wird. Die anschließende Ausführung von Tests wird dabei erfasst und für die Generierung eines Berichtes genutzt.

Das Plug-In ist nach der Installation in der Lage die Abdeckung von Komponenten- und Integrationstests zu erfassen, leider aber nicht die von funktionalen Tests. Ein Weg wie die Abdeckung aus allen drei Testarten ermittelt werden kann soll in diesem Artikel vorgestellt werden.

CONTINUOUS INTEGRATION

Ein Continuous Integration Server (CI-Server) ist ein dedizierter Server mit der Aufgabe eine Anwendung regelmäßig oder bei Code-Änderungen komplett neu zu bauen, zu testen und alle dabei entstehenden Berichte zur Verfügung zu stellen. Darüber hinaus sollte er auch immer den aktuellsten lauffähigen Entwicklungsstand auf einen Demo-Server deployen.

In diesem Artikel wird Jetbrains TeamCity CI-Server in Version 3.1.2 [5] verwendet, um eine Grails Anwendung zu bauen und alle dabei entstehenden Berichte bereit zu stellen.

DEMO ANWENDUNG

Um das Vorhaben zu Veranschaulichen wird eine einfache Demo Anwendung erstellt. Sie wird alle benötigten Plugins und eine Domänenklasse mit zugehörigem Controller und Views beinhalten. Zudem wird sie einen Komponenten-, einen Integrations- und einen funktionalen Test beinhalten. Abbildung 1 gibt einen Überblick über die Demo Anwendung die im Folgenden erstellt wird.

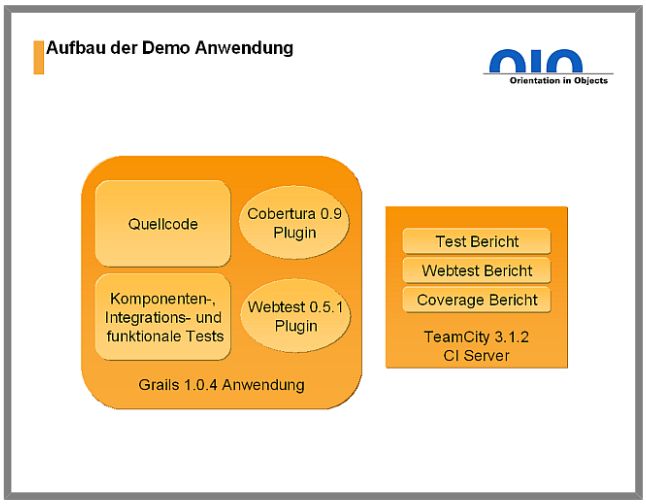

Abbildung 1: Aufbau der Demo Anwendung

Vorausgesetzt man hat Grails bereits installiert, kann man mit folgenden Konsolenkommandos alle nötigen Plugins und Anwendungsbestandteile erstellen.

```
grails create-app abdeckung
cd abdeckung
grails install-plugin code-coverage 0.9
grails install-plugin webtest 0.5.1
grails create-domain-class de.oio.grails.Buch
```

Beispiel 1: Erstellung einer Demo Anwendung

Der Inhalt der generierten Domänenklasse soll der folgende sein.

```
package de.oio.grails
class Buch {
  String title
  Integer isbn
  static constraints = {
    isbn(nullable: true)
  }
}
```

Beispiel 2: Inhalt der Domänenklasse Buch

Nachdem die Domänenklasse angepasst wurde, folgen drei weitere Kommandos.

```
grails generate-all de.oio.grails.Buch
grails create-unit-test de.oio.grails.BuchUnit
grails create-webtest Buch
```

Beispiel 3: Generierung von Views, Controller und Tests

Die soeben erstellte Demo-Anwendung ist nun fertig und kann bereits getestet werden. Auch bekommen wir unsere ersten Berichte. Allerdings ist es noch nicht möglich einen Testabdeckungs-Bericht zu bekommen, der aus allen existierenden Tests resultiert. Bisher ist es nur möglich die Abdeckung die aus Komponenten- und Integrationstests resultiert mit folgendem Kommando zu ermitteln.

```
grails test-app-cobertura
```

Beispiel 4: Grails Coverage Plugin Kommando zur Ermittlung der Testabdeckung aus Komponenten- und Integrationstests

Darüberhinaus möchte man wahrscheinlich die Config.groovy der Anwendung um folgende Zeilen erweitern, damit keine Bestandteile des Webtest Plugins in den Abdeckungs-Bericht gelangen.

```
//cobertura exclusions
coverage {
  exclusions = ['grails/util/**']
}
```

Beispiel 5: Eintrag in Config.groovy um Webtest Klassen vom Abdeckungsbericht auszuschließen

Würde man den Befehl zur Ermittlung der Testabdeckung jetzt ausführen, sollte man einen Testbericht erhalten, wie er in Abbildung 2 zu sehen ist.

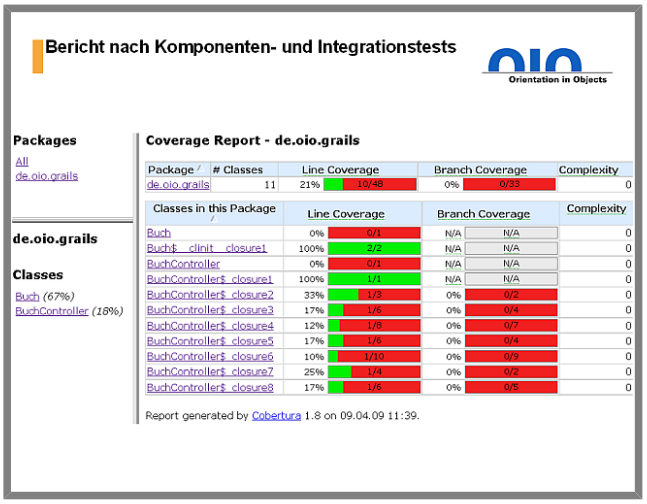

Abbildung 2: Cobertura Bericht nach Komponenten- und Integrationstest

TESTABDECKUNG ERMITTELN

Ein Weg die Testabdeckung aus allen Tests zu ermitteln, ist das Erstellen und Ausführen eines eigenen Gant-Skriptes. Dieses kann mit dem Kommando *grails create-script TestCoverage* generiert werden. Das neue Skript wurde im Ordner *scripts* der Anwendung abgelegt und soll nun die Skripte des Cobertura und Webtest Plugins so kombinieren, dass wir unser Ziel erreichen.

Folgend ist der Inhalt eines Gant-Skriptes aufgelistet, mit dem es möglich ist die Gesamtabdeckung zu ermitteln.

```
includeTargets << grailsScript ( "Clean" )
includeTargets << new File
   ("${webtestPluginDir}/scripts/RunWebtest.groovy")
includeTargets << new File
   ("${codeCoveragePluginDir}/scripts/TestAppCobertura.groovy")
coverageReportDir = "${testDir}/cobertura"
target('default': "Run all tests with Cobertura") {
  clean()
  cleanup()
  // Add custom exclusions.
  // packageApp is needed to load the config packageApp()
  if (config.coverage.exclusions) {
    codeCoverageExclusionList += config.coverage.exclusions
  }
  compileTests()
  packageTests()
  Ant.path(id: "cobertura.classpath") {
    fileset(dir: "${codeCoveragePluginDir}/lib/cobertura") {
      include(name: "*.jar")
    }
  }
  instrumentTests()
  loadApp()
  runWebTest()
  stopServer()
  runUnitTests()
  runIntegrationTests()
  produceReports()
  try {
    def saveClass =
      Class.forName('net.sourceforge.cobertura.coveragedata.ProjectData')
    def saveMethod =
      saveClass.getDeclaredMethod("saveGlobalProjectData",
      new Class[0])
    saveMethod.invoke(null, new Object[0]);
  }
  catch (Throwable t) {
    t.printStackTrace()
    event("StatusFinal",
        ["Unable to flush Cobertura code coverage data."])
    exit(1)
  }
  coberturaReport()
  Ant.delete(dir: testDirPath)
}
```

Beispiel 6: Erstelltes Skript zur Ermittlung der Testabdeckung aus Komponenten-, Integrations und funktionalen Tests

Im vorangegangenen Listing werden zuerst alle existierenden Kompilate gelöscht. Wird dies nicht gemacht, kann es zu Problemen bei der Ermittlung der Abdeckung kommen.

Anschließend wird *packageApp()* aufgerufen, damit auf das Config-Objekt zugegriffen werden kann. Dies ist nötig um alle uninteressanten Klassen vor der Instrumentierung durch Cobertura auszuschließen.

Im nächsten Schritt wird *compileTests()* aufgerufen, wodurch alle Klassen, ausgenommen die der Plugins, kompiliert werden. Nun ist es möglich mit dem Aufruf von instrumentTests die soeben erstellten Klassen zu instrumentieren.

Anschließend können die Webtests mit dem Befehl *runWebTest()* ausgeführt werden. Sie müssen vor den Integrationstests ausgeführt werden, da es sonst zu Problemen mit dem ApplicationContext kommen kann. Zuvor muss allerdings noch *loadApp()* aufgerufen werden, da nach dem Aufruf von *compileTests()* der Classpath verändert wurde, so dass die Plugins nicht mehr sichtbar sind, wie in Abbildung 3 zu sehen ist. Der Webtest-Bericht wird ohne weiteres Zutun erstellt.

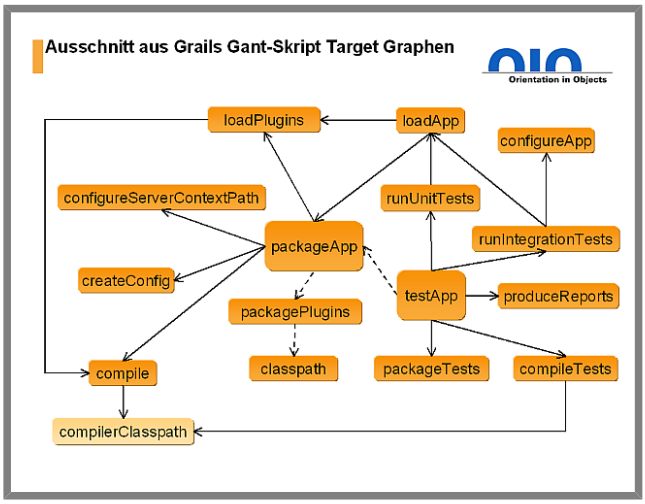

Abbildung 3: Ausschnitt aus Grails Gant-Skript Target Graphen

Nach dem Aufruf von *stopServer()* können die Komponenten- und Integrationstests ausgeführt werden. Sind alle Tests durchlaufen, wird mit dem Aufruf von *produceReports()* der dazugehörige Bericht erstellt.

Anschließend werden die Aufzeichnungsdaten von Cobertura gesichert und mit dem Aufruf von *reportCobertura()* in einem Testabdeckungsbericht zusammengefasst. Der erstellte Bericht für die Demo Anwendung sollte wie in Abbildung 4 aussehen.

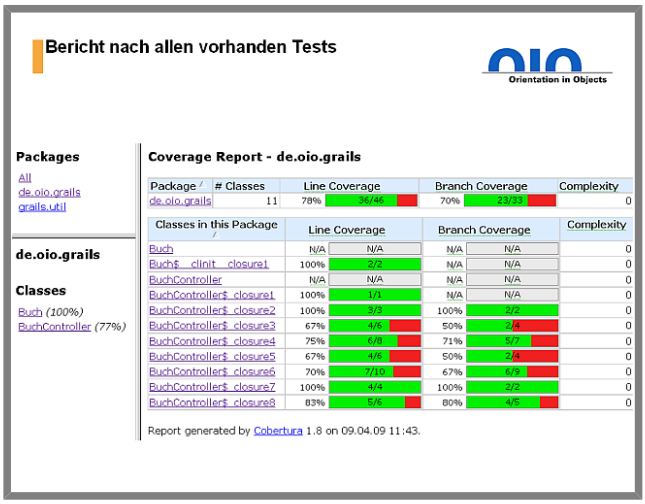

Abbildung 4: Cobertura Bericht der Demo Anwendung nach allen vorhanden Tests

BEREITSTELLEN DER BERICHTE

Damit die Ergebnisse der Testabdeckungsmessung über den CI Server erreichbar sind, passen wir als erstes die Datei *build.xml* im Wurzelverzeichnis der Anwendung an. Wir fügen einen neuen Eintrag hinzu, der im folgenden Listing zu sehen ist.

```
<target name="codecoverage" description="Run all tests with
Cobertura">
  <exec executable="${grails}" failonerror="false"
  resultproperty="resultCode">
    <arg value="code-coverage"/>
  </exec>
  <zip basedir="test/reports/cobertura"
  destfile="test/reports/codeCoverage.zip"/>
  <fail status="1" message="Some unit tests failed">
    <condition>
      <equals arg1="${resultCode}" arg2="1"/>
    </condition>
  </fail>
</target>
```

Beispiel 7: Zusätzliches Target für die build.xml zur Erstellung des Testabdeckungsberichtes

Zu beachten ist, dass *failonerror* auf *false* gesetzt ist, so dass ein Bericht generiert wird, auch wenn Tests fehlschlagen. Damit der CI Server das Fehlschlagen der Tests registrieren und melden kann, folgt der fail-Task. Dieser wird zurückgegeben, wenn *resultCode* auf 1 gesetzt wurde.

Nun muss noch die Datei *.BuildServer/config/main-config.xml* angepasst werden, damit der erstellte Abdeckungsbericht auch im TeamCity CI Server komfortabel über dessen Web-Interface erreichbar ist. Dazu braucht man einfach nur die folgende Zeile dem Teamcity XML-Dokument hinzufügen.

```
<report-tab title="Coverage" basepath="codeCoverage.zip"
startpage="index.html"/>
```

Beispiel 8: Zusätzlicher Eintrag in die Konfiguration des TeamCity CI Servers

Abschließend muss noch der Build Runner des TeamCitys angepasst werden, so dass das neu hinzugefügte Target der *build.xml* ausgeführt wird.

Möchte man weitere Berichte einbinden, muss man für diese ganz genauso vorgehen. Eine gute Hilfestellung beim Einrichten des TeamCity CI Servers bietet die Artikelserie von Václav Pech [6].

ZUSAMMENFASSUNG

Wie zu sehen, ist es möglich Komponenten-, Integrations- und funktionale Tests ablaufen zu lassen und die resultierende Abdeckung aus allen existierenden Tests aufzuzeichnen.

Allerdings ist es relativ aufwendig zu durchschauen welche Skripte welchen Einfluss auf den Anwendungszustand haben. Grails besitzt insgesamt 40 Skripte mit über 160 Targets. Da kann es leicht zu unerwarteten Nebeneffekten kommen, wenn man sich nicht intensiv mit der Materie auseinander gesetzt hat.

Das Bereitstellen der Ergebnisse ist jedoch wieder sehr einfach. Die ZIP-Dateien mit den Berichten können einem CI Server übergeben werden und dieser kann sie dann z.B. im Falle von TeamCity über das Web-Interface einfach zugänglich machen.

REFERENZEN

- The Definitive Guide to Grails
  Rocher, Graeme , SpringSource; Brown, Jeff , SpringSource
  Apress; 2009-01-21;Seiten: 384
- Ruby on Rails
  (http://rubyonrails.org/)
- Gant
  (http://gant.codehaus.org/)
- Test Code Coverage Plugin
  Hugo, Mike
  (http://www.grails.org/Test+Code+Coverage+Plugin)
- Jetbrains TeamCity
  (http://www.jetbrains.com/teamcity/)
- Automate Grails App Builds with TeamCity
  Pech, Václav , Jetbrains, Inc.
  (http://jetbrains.dzone.com/news/automate-grails-app-builds-tea/)