
Protokoll SEW Continuous Integration

Continuous Integration with Jenkins

SEW
5bHITT 2016/17

Philipp Kogler

Betreuer: Prof. Rafeiner, Prof. Dolezal

Begonnen am 5. März 2017
Beendet am 5. März 2017

Inhaltsverzeichnis

1	Einführung	1
1.1	Ziele	1
1.2	Voraussetzungen	1
1.3	Aufgabenstellung	1
2	Allgemein - Continuous Integration	2
2.1	Continuous Integration mit Jenkins	2
3	Ergebnisse	3
3.1	Installieren/Konfigurieren von Jenkins	3
3.1.1	Konfigurieren und Installieren der Plugins	3
3.2	Erstellen eines neuen Jobs	4
3.3	Durchführen eines Builds	7
3.4	Build Report und Auswertung	8
3.4.1	Übersicht des Reports	8
3.4.2	Coverage Report	9
3.4.3	Test Reports	9
3.4.4	Pylint Reports	9
3.5	Aufgetretene Probleme	10
3.6	GUI - Testing mit Jenkins	11
3.6.1	QF-Test Plugin [1]	11
3.6.2	Python GUI Testing mit Selenium [2]	11
4	Zeitaufwand	11

1 Einführung

Diese Übung soll mithilfe von Jenkins einen Einblick in das umfangreiche Thema Continuous Integration bieten. Hierbei wird auf einem bereits erstellten Python Programm aufgebaut und mittels Git (Github) eine automatische Testumgebung aufgebaut.

1.1 Ziele

Das Ziel dieser Übung ist die Einführung in das umfangreiche Thema Continuous Integration.

Zusätzlich soll ein Protokoll erstellt werden, welches diesen Vorgang protokolliert und entsprechende Merkmale hervorhebt

1.2 Voraussetzungen

- Grundlagen in Python
- Grundlagen Testing (JUnit)
- Grundlagen Continuous Integration

1.3 Aufgabenstellung

- Installiere auf deinem Rechner bzw. einer virtuellen Instanz das Continuous Integration System Jenkins
- Installiere die notwendigen Plugins für Jenkins (Violations, Cobertura) und alle weiteren benötigten Plugins
- Installiere Nose und Pylint (mithilfe von pip) um das Modul Bruch ausführlich zu testen
- Integriere dein Bruch-Projekt in Jenkins, indem du es mit Git verbindest
- Konfiguriere Jenkins so, dass deine Unit Tests automatisch bei jedem Build durchgeführt werden
- Protokolliere deine Vorgehensweise (inkl. Zeitaufwand, Konfiguration, Probleme) und die Ergebnisse (viele Screenshots!)
- Schreibe fünf weitere Testfälle für die Bruch-Klasse
- Können GUI-Tests mit Jenkins automatisch durchgeführt werden?

2 Allgemein - Continuous Integration

Kontinuierliche Integration (Continuous Integration) ist eine Praxis in der Softwareentwicklung. Dabei werden isolierte Änderungen sofort geprüft und anschließend zur Gesamtcodebasis einer Software hinzugefügt. Ziel der kontinuierlichen Integration ist es, ein unmittelbares Feedback bieten zu können, so dass ein versehentlich integrierter Fehler so schnell wie möglich identifiziert und korrigiert wird. Tools für kontinuierliche Integration lassen sich dazu verwenden, Tests zu automatisieren und um eine fortlaufende Dokumentation zu erstellen.

Kontinuierliche Integration wurde in den letzten Jahren permanent weiterentwickelt. Ursprünglich war ein täglicher Build (täglicher Erstellungsprozess) Standard. Mittlerweile hat sich der Ansatz etabliert, regelmäßige Builds zu erstellen - zum Beispiel so bald ein bestimmter Arbeitsschritt fertiggestellt ist.

Wird kontinuierliche Integration richtig angewandt, bietet der Ansatz verschiedene Vorteile, wie zum Beispiel ständige Feedbacks zum Status einer Software. Da sich mit kontinuierlicher Integration Mängel bereits in einem frühen Entwicklungsstadium erkennen lassen, fallen Softwarefehler kleiner sowie weniger komplex aus und lassen sich einfacher beseitigen.

2.1 Continuous Integration mit Jenkins

Jenkins ist eine Software, welche das Prinzip Continuous Integration ermöglicht.

Jenkins wird auf einem Server (localhost auch möglich) installiert, wo der Build Zentral stattfindet

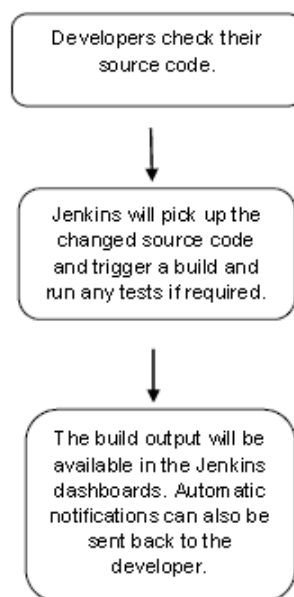


Abbildung 1: Jenkins

3 Ergebnisse

3.1 Installieren/Konfigurieren von Jenkins

Jenkins ist für fast alle Plattformen erhältlich. Nach der Installation von Jenkins wird auf localhost unter 8080 ein Webserver gestartet, über welchen Jenkins konfiguriert werden kann.

Zu erst muss ein automatisch generiertes Passwort, welches sich in dem erstellten jenkins Logfile befindet, eingegeben werden. In meinem Fall befand sich das Logfile unter `/var/log/jenkins`

3.1.1 Konfigurieren und Installieren der Plugins

Manage Jenkins/ Github Support

Unter *Manage Jenkins* > *Configure System* unter dem Punkt **Git plugin** kann der Username sowie die email Adresse global gesetzt werden.




Abbildung 2: Username setzen / Git

Benötigte Plugins

Um alle Anforderungen zu erfüllen müssen mindestens folgende Plugins installiert werden.

- Violations
- GitHub Integration Plugin
- Cobertura Plugin

Um ein Plugin zu installieren, muss zu folgender Seite navigiert werden

Manage Jenkins > *Manage Plugin* > *Available*

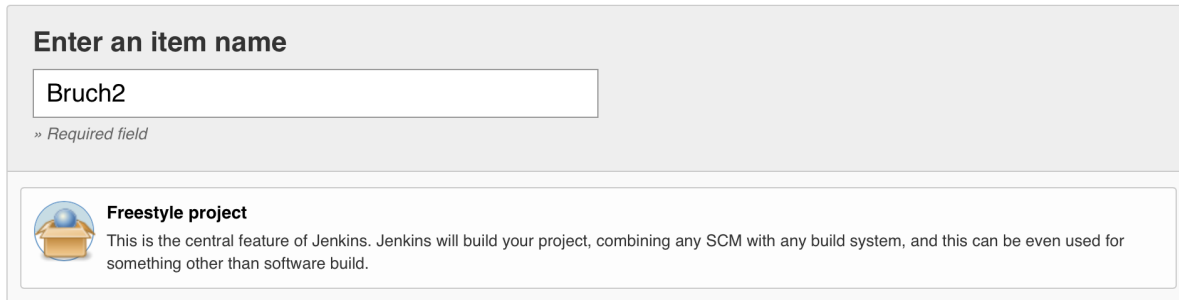
Nach der Installation eines Plugins sollte Jenkins neugestartet werden.

3.2 Erstellen eines neuen Jobs

Nach der Konfiguration kann ein neuer Job/Item erstellt werden. Folgende Schritte sind hierbei zu beachten.

- **Projekt Typ auswählen**

Eingabe des Projektnamen, sowie Auswahl des Typs



Enter an item name

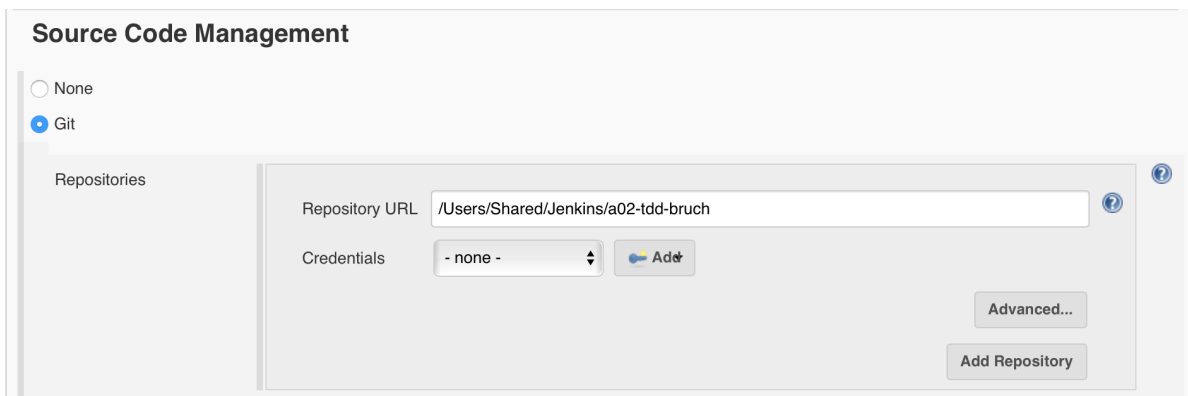
Bruch2

» Required field

Freestyle project
This is the central feature of Jenkins. Jenkins will build your project, combining any SCM with any build system, and this can be even used for something other than software build.

- **Source Code Management**

Angabe des Projekt Pfaded hier GitHub Integration In meinem Fall wird ein lokales Repository



Source Code Management

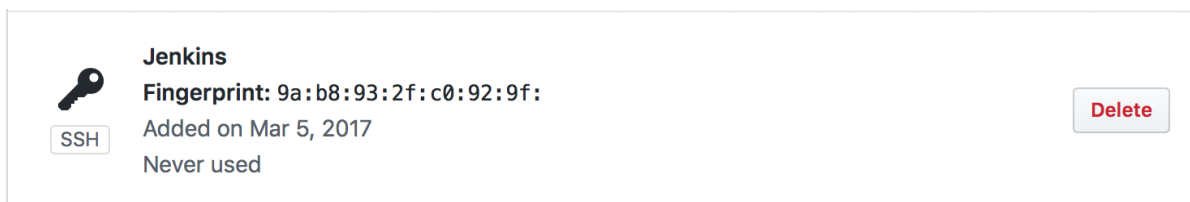
☐ None
☒ Git

Repositories

Repository URL

Credentials

genutzt. Damit das erfolgreich eingerichtet werden kann und Jenkins somit Zugriff auf dieses Repo hat, kann ein neuer ssh Key generiert werden und auf Github registriert werden.



Jenkins
Fingerprint: 9a:b8:93:2f:c0:92:9f:
Added on Mar 5, 2017
Never used

• Build Triggers

Anschließend können ein oder mehrere Build Trigger gesetzt werden. Also nach welcher Aktion/Aktionen ein Build gestartet werden soll. In unserem Fall wurde ein SCM Trigger gesetzt, welcher alle 5 min einen automatischen Build startet.

Build Triggers

☐ Trigger builds remotely (e.g., from scripts)

☐ Build after other projects are built

☐ Build periodically

☐ GitHub hook trigger for GITScm polling

☒ Poll SCM

Schedule

⚠ Do you really mean "every minute" when you say "*****"? Perhaps you meant "H *****" to poll once per hour

Would last have run at Sunday, March 5, 2017 6:54:20 PM CET; would next run at Sunday, March 5, 2017 6:54:20 PM CET.

Ignore post-commit hooks ☐

• Build Configuration

Der eigentliche Build kann nun in als *Execute Shell* beschrieben und definiert werden. Alle auszuführenden Kommandos werden in dem angegeben Verzeichnis angegeben.

Build

Execute shell

Command

```
/Library/Frameworks/Python.framework/Versions/3.5/bin/nosetests bruch/test/*.py --with-xunit --a
/usr/local/bin/python3 -m coverage xml
/Library/Frameworks/Python.framework/Versions/3.5/bin/pylint -f parseable -d I0011,R0801 bruch |
```

See [the list of available environment variables](#)

Advanced...

Add build step

.../nosetests	bruch/test/*.py erstellt ein xml file welches später interpretiert werden kann
./python cov	erstellt einen coverage report als xml
.pylint parse	analysiert das modul bruch

- **Post Build Configuration**

Anschließend können diese Build Ergebnisse als Post Build Script ausgewertet werden. Also nach einem erfolgreichem Build können bestimmte Befehle ausgeführt werden.

Auswerten der erstellten Reports. Alle ausgewerteten Berichte sind im XML Format und können so ausgewertet werden. Ebenfalls soll das Ergebnis von pylint dargestellt werden. Dies kann mithilfe

Post-build Actions

Publish Cobertura Coverage Report

X

Cobertura xml report pattern

coverage.xml

This is a file name pattern that can be used to locate the cobertura xml report files (for example with Maven2 use ****/target/site/cobertura/coverage.xml**). The path is relative to the module root unless you have configured your SCM with multiple modules, in which case it is relative to the workspace root. Note that the module root is SCM-specific, and may not be the same as the workspace root.

Cobertura must be configured to generate XML reports for this plugin to function.

NOTE: If concurrent builds are enabled for this job, and a later build finishes before an earlier build, the later build will reduce or skip trend analysis/charting.

Advanced...

Publish JUnit test result report

X

?

Test report XMLs

nosetests.xml

[Fileset 'includes'](#) setting that specifies the generated raw XML report files, such as 'myproject/target/test-reports/*.xml'. Basedir of the fileset is [the workspace root](#).

☐ Retain long standard output/error

Health report amplification factor

1,0

?

1% failing tests scores as 99% health. 5% failing tests scores as 95% health

Allow empty results

☒ Do not fail the build on empty test results

?

des generierten *pylint.out* umgesetzt werden.

pylint

10


999

999

**pylint.out

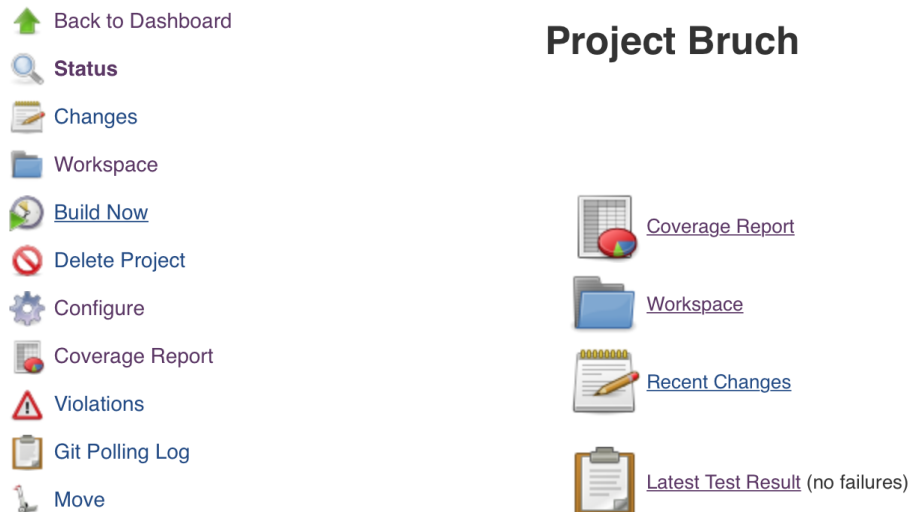
5bHITT 2016/17

6 of 12

 Philipp Kogler

3.3 Durchführen eines Builds

Ein Build kann auch manuell durchgeführt werden. Um diesen einzuleiten muss wie folgt vorgegangen werden.

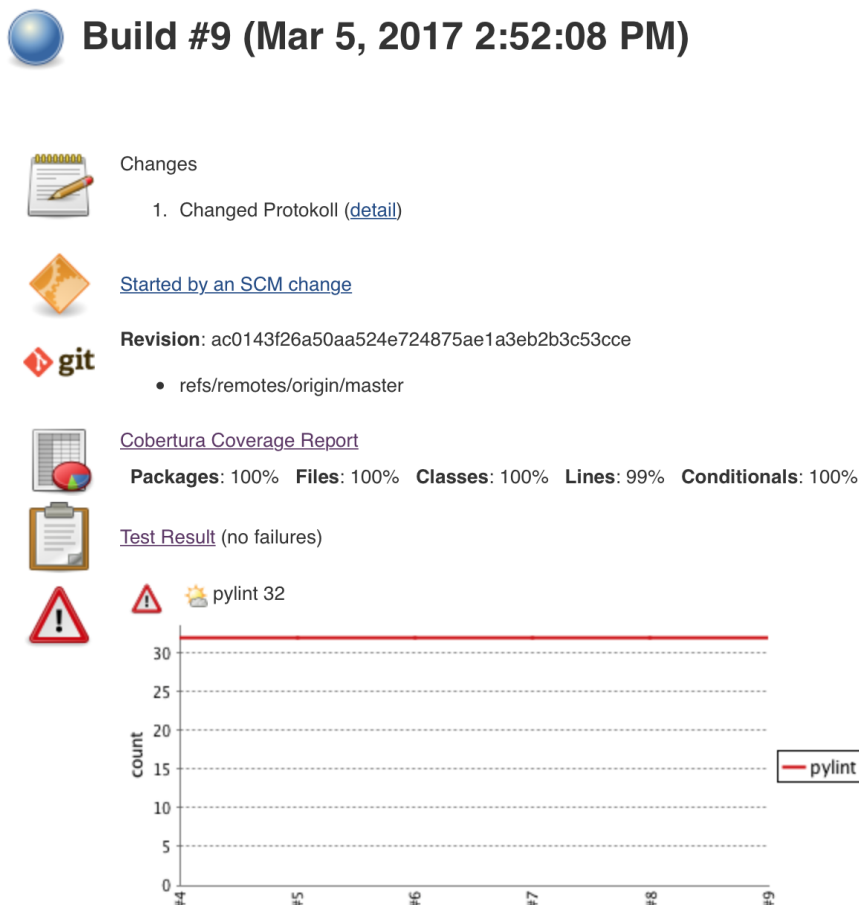


3.4 Build Report und Auswertung

Nach jedem erfolgreichem Build (gekennzeichnet mit einem Blauen Punkt) wird das dazugehörige Post Build Script ausgeführt. In unserem Fall werden sowohl die Testberichte als auch der Coverage Bericht ausgewertet. Ebenfalls wird mittels pylint eine kurze Code Analyse durchgeführt.

3.4.1 Übersicht des Reports

Aufgeteilt in Änderungen, Git Report, Coverage Report und Test Results



Test Result

0 failures (±0)

140 tests (±0)

[Took 0 ms.](#)[add description](#)

All Tests

Package	Duration	Fail	(diff) Skip	(diff) Pass	(diff) Total	(diff)
Testall	0 ms	0	0	69	69	
Unit_Addition	0 ms	0	0	8	8	
Unit_Allgemein	0 ms	0	0	18	18	
Unit_Division	0 ms	0	0	12	12	
Unit_Erweitert	0 ms	0	0	5	5	
Unit_Multiplikation	0 ms	0	0	8	8	
Unit_String	0 ms	0	0	2	2	
Unit_Subtraktion	0 ms	0	0	8	8	
Unit_Vergleich	0 ms	0	0	7	7	
Unit_Zusatz	0 ms	0	0	3	3	

3.4.2 Coverage Report

3.4.3 Test Reports

Alle durchgeführten Tests werden ebenfalls dargestellt.

Die Navigation erfolgt ähnlich wie bei normalen (html) exportierten Unit Tests

Test Result

0 failures (±0)

140 tests (±0)

[Took 0 ms.](#)[add description](#)

All Tests

Package	Duration	Fail	(diff) Skip	(diff) Pass	(diff) Total	(diff)
Testall	0 ms	0	0	69	69	
Unit_Addition	0 ms	0	0	8	8	
Unit_Allgemein	0 ms	0	0	18	18	
Unit_Division	0 ms	0	0	12	12	
Unit_Erweitert	0 ms	0	0	5	5	
Unit_Multiplikation	0 ms	0	0	8	8	
Unit_String	0 ms	0	0	2	2	
Unit_Subtraktion	0 ms	0	0	8	8	
Unit_Vergleich	0 ms	0	0	7	7	
Unit_Zusatz	0 ms	0	0	3	3	

3.4.4 Pylint Reports

Die Violation Reports können mit pylint durchgeführt werden. Hierbei werden vor allem Die Richtlinien nach dem Styleguide von Python **PEP 8** beachtet.

Violations Report /job/Bruch/9 for build 9

Type	Violations	Files in violation
pylint	32	1

pylint



filename	l	m	h	number ↑
bruch/Bruch.py	23	5	4	32

3.5 Aufgetretene Probleme

Das Installieren von Jenkins war sehr einfach und aufgrund der gut ausgeführten Dokumentation sowie der relativ großen Community unkompliziert umzusetzen.

Auch das Erstellen des Items war nicht weiter schwierig. Jedoch sind Probleme beim Erstellen der Build Konfiguration aufgetreten. Da alle Befehle direkt von dem erstellten jenkins User durchgeführt werden, muss darauf geachtet werden, dass dieser die Berechtigungen für alle notwendigen Befehle hat.

Beispielsweise wurden exportierte Aliases nicht für den User übernommen. Als Lösung wurde der Pfad absolut angegeben und auf diese benötigten Scripts dementsprechende Berechtigungen vergeben.

Das Installieren von nose und pylint ist ebenfalls sehr einfach verlaufen. Jedoch wurde unter OSX nose-test nicht in die PATH Variable aufgenommen. Nach Lokation des bin Verzeichnisses von Python, war die Verlinkung jedoch kein Problem mehr.

3.6 GUI - Testing mit Jenkins

Nach einer ausführlichen Recherche über das oben genannte Thema bin ich auf folgende Ergebnisse gekommen.

GUI - Testing an sich ist mit Jenkins selbstverständlich möglich, jedoch sind die Möglichkeiten sehr eingeschränkt. Beispielsweise kann unabhängig von OS und zu testender Programmiersprache (sofern solche Tools für die Programmiersprache vorhanden sind), ein Script ausgeführt werden welches das Programm startet und beispielsweise Deskription und Initialisierungswerte der GUI abfragt und dementsprechend in einem XML File ablegt. Dieses kann anschließend von Jenkins ausgewertet werden.

3.6.1 QF-Test Plugin [1]

Diese Plugin für Jenkins ermöglicht das Testen von GUI - Applikationen. Hierbei können jedoch nur Java Applikationen getestet werden. (JavaFX, Swing) und Java EE Applikationen.

3.6.2 Python GUI Testing mit Selenium [2]

Um eine Python Applikation kann beispielsweise Selenium verwendet werden. Ein ausführliches Tutorial wie so ein Test Job durchgeführt werden kann wird in diesem Blog Post beschrieben. [2]

4 Zeitaufwand

Fertigstellung	Beschreibung	Geschätzte Zeit	Benötigte Zeit
05.03..2017	Installation von Jenkins	50 min	30 min
05.03..2017	Konfiguration von Jenkins	50 min	40 min
05.03..2017	Erstellen eines Jenkins Jobs	130 min	100 min
05.03..2017	Konfiguration des Builds	40 min	50 min
05.03..2017	Erweiterte Kompetenzen	40 min	50 min
05.03..2017	Protokoll	100 min	100 min
	Insgesamt	290 min	310 min

Tabelle 1: Zeitschätzung

Literatur

[1] <https://www.qfs.de/en.html>.

[2] <https://www.blazemeter.com/blog/how-automate-testing-using-selenium-webdriver>

Tabellenverzeichnis

1	Zeitschätzung	11
---	-------------------------	----

Abbildungsverzeichnis

1	Jenkins	2
2	Username setzen / Git	3