
Übungsprotokoll A04

Continuous Integration mit Jenkins

Softwareentwicklung
5BHIT 2017/18

Martin Wölfer

Note:
Betreuer: DOLD & RAFW

Version 0.1
Begonnen am 5. Oktober 2017
Beendet am 19. Oktober 2017

Inhaltsverzeichnis

1	Aufgabenstellung	1
1.1	Grundanforderungen (70%)	1
1.2	Erweiterungen (30%)	1
2	Installation und Konfiguration	2
2.1	Erstes Problem mit VM	2
2.2	Installation in Windows	2
2.3	Github-User Konfiguration	2
3	Ersten Job erstellen	3
3.1	Job konfigurieren	3
3.1.1	Woher Job Quelle bezieht	3
3.1.2	Wann Job ausgeführt wird	4
3.1.3	Was ausgeführt wird	4
3.1.4	Was passiert mit Ergebnissen	5
3.2	Job ausführen	5
3.2.1	Code Coverage	6
3.2.2	Test Results	6
3.2.3	Violations report	7

1 Aufgabenstellung

"Continuous Integration is a software development practice where members of a team integrate their work frequently, usually each person integrates at least daily - leading to multiple integrations per day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible. Many teams find that this approach leads to significantly reduced integration problems and allows a team to develop cohesive software more rapidly. This article is a quick overview of Continuous Integration summarizing the technique and its current usage." M.Fowler

Lass das Bruch-Projekt mithilfe von Jenkins automatisch bei jedem Build testen!

1.1 Grundanforderungen (70%)

- Installiere auf deinem Rechner bzw. einer virtuellen Instanz das Continuous Integration System Jenkins
- Installiere die notwendigen Plugins für Jenkins (Violations, Cobertura)
- Installiere Nose, Coverage und Pylint (mithilfe von pip)
- Integriere dein Bruch-Projekt in Jenkins, indem du es mit Git verbindest
- Überlege dir und argumentiere eine sinnvolle Pull-Strategie
- Konfiguriere Jenkins so, dass deine Unit Tests automatisch bei jedem Build durchgeführt werden inkl. Berichte über erfolgreiche / fehlgeschlagene Tests und Coverage
- Protokolliere deine Vorgehensweise (inkl. Zeitaufwand, Konfiguration, Probleme) und die Ergebnisse (viele Screenshots!)

1.2 Erweiterungen (30%)

- Konfiguriere und teste eine Git-Hook, sodass Änderungen auf GitHub automatisch einen Build auslösen! Dokumentiere deine Vorgangsweise (mit Screenshots)!
- Recherchiere, wie mithilfe von Jenkins GUI-Tests durchgeführt werden können und baue selbstständig einen Beispiel-GUI-Test ein! Dokumentiere deine Vorgangsweise (mit Screenshots)!
- Lass deine Sphinx-Dokumentation automatisch mitbuilden und veröffentlichen! Dokumentiere deine Vorgangsweise (mit Screenshots)!

2 Installation und Konfiguration

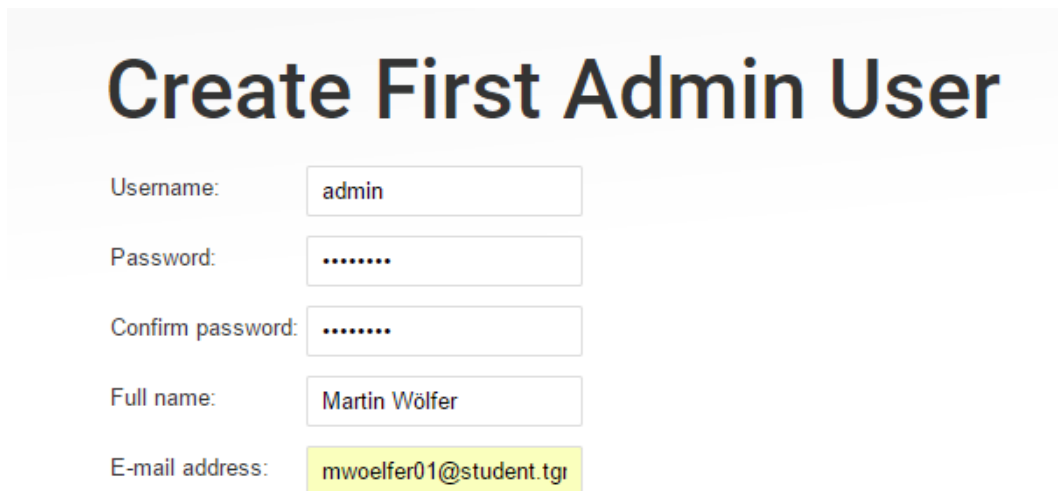
Der erste Schritt war es Jenkins zu installieren.

2.1 Erstes Problem mit VM

Bei der Installation ist mein erstes Problem aufgetreten, da ich dachte dass es einfacher wäre Jenkins in einer virtuellen Maschine laufen zu lassen. Dies bedeutet es wurde Jenkins installiert, was recht einfach lief dadurch dass Jenkins im apt-get Repository liegt, und die Konfiguration funktionierte dank Tutorial auch flüssig. Nun ist mir aufgefallen, vor allem beim Plugins installieren, dass die VM sehr langsam ist bzw. bei manchen Plugins garnicht funktioniert - noch dazu kommt dass ich das gesamte Git-Repository auf die virtuelle Maschine kopieren musste da Jenkins die Dateien lokal bezieht. Nach längerer Frustration bin ich schlussendlich auf Windows umgestiegen.

2.2 Installation in Windows

Die Installation in Windows läuft auch einfach ab, man installiert das .msi file, öffnet localhost:8080 und geht den Installations-Wizard durch. Es wurden die richtigen Plugins angewählt bei der Erstkonfiguration (Außer Jenkins Violations da dieses erst im Dashboard installiert werden kann) und ein Admin user wurde angelegt:



The screenshot shows the 'Create First Admin User' form in Jenkins. The form has the following fields and values:

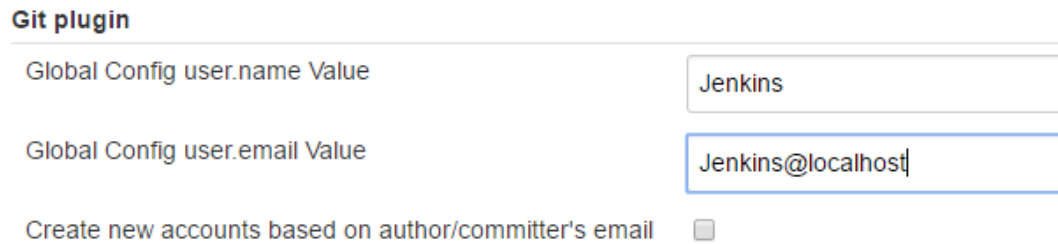
Field	Value
Username:	admin
Password:
Confirm password:
Full name:	Martin Wölfer
E-mail address:	mwoelfer01@student.tgr

Abbildung 1: Ein Admin User wurde erstellt

Anschließend wurde unter Manage Jenkins → Manage plugins → Available auch noch das letzte fehlende Plugin installiert (Jenkins Violations).

2.3 Github-User Konfiguration

Unter Manage Jenkins → Configure System → Git Plugin werden die Daten eingegeben.



Git plugin

Global Config user.name Value

Global Config user.email Value

Create new accounts based on author/commmitter's email ☐

Abbildung 2: Daten werden eingegeben für Git

3 Ersten Job erstellen

Im Dashboard wird einem gleich angeboten einen Job zu erstellen, danach wird ein Name für den Job eingegeben und es wird **Freestyle Project** ausgewählt



Enter an item name

[» Required field](#)

 **Freestyle project**
This is the central feature of Jenkins. Jenkins will build your project, co

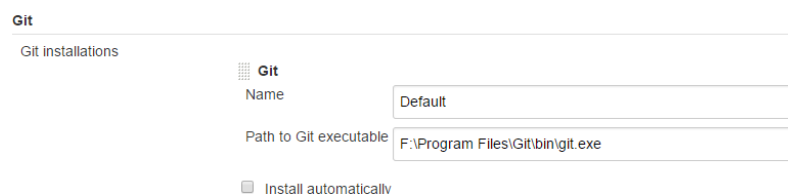
Abbildung 3: Neuen Job erstellen

3.1 Job konfigurieren

3.1.1 Woher Job Quelle bezieht


Zuerst muss Jenkins mitgeteilt werden wo die Quelle des Projektes überhaupt liegt, es wird zuerst unter der Section **Source Code Management** der Button **Git** angewählt, und anschließend der lokale Repository URL angegeben werden.

Hier bin ich auf ein weiteres Problem gestoßen, und zwar wurde die Fehlermeldung ausgegeben: **jenkins failed to connect to repository**. Dies lag daran dass Jenkins nicht wusste wo sich lokal mein **git.exe** befindet, daher musste ich in **Manage Jenkins** → **Global Tool Configuration** → **Git** → **Git Installations** den Pfad zu **git.exe** angeben:



Git

Git installations

 **Git**

Name

Path to Git executable

☐ Install automatically

Nun konnte der lokale Pfad zum Repository angegeben werden ohne dass ein Problem besteht:

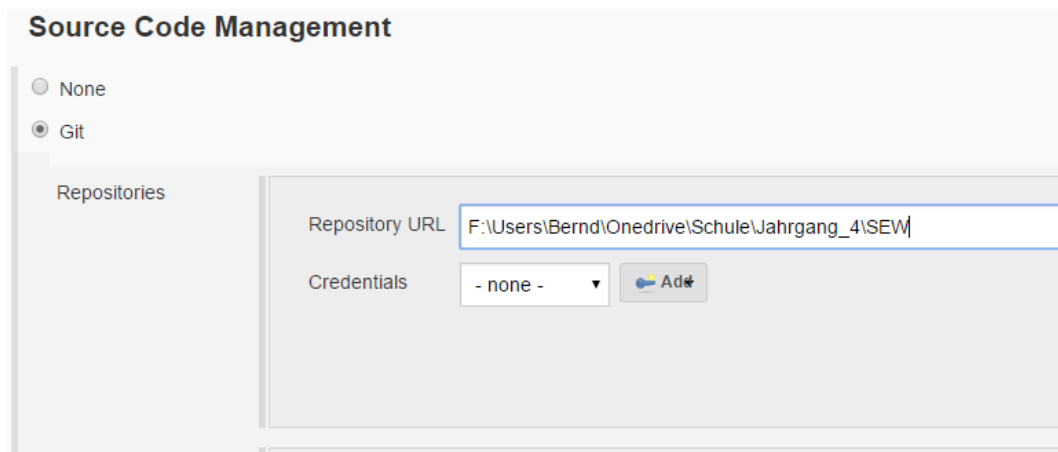
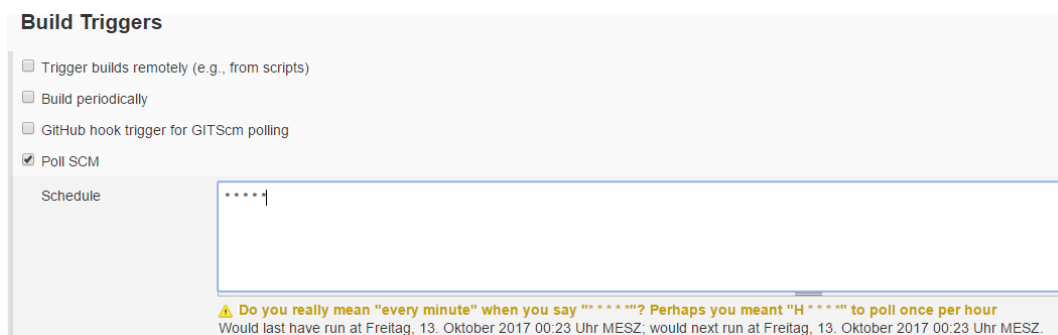


Abbildung 4: Pfad konnte angegeben werden ohne Fehlermeldung

3.1.2 Wann Job ausgeführt wird

Nun wird unter Build Triggers → Poll SCM angegeben wann der Job ausgeführt wird, indem 5 * angegeben werden:



3.1.3 Was ausgeführt wird

Unter Build → Add build step → Execute Shell wird nun folgender Code angegeben:

```
1 nosetests --with-xunit --all-modules --traverse-namespace --with-coverage --cover-inclusive
  coverage xml Testall.py
3 pylint -f parseable -d I0011,R0801 testall.py > pylint.out | type pylint.out
```

Der erste Befehl erstellt das `nosetests.xml` File. Es werden in dem momentanen Arbeitspfad nach Tests gesucht, welche anschließend ausgeführt werden und in dem File gespeichert.

Der zweite command erstellt ein xml coverage report file vom File `Testall.py`, in welchem alle Tests definiert sind.

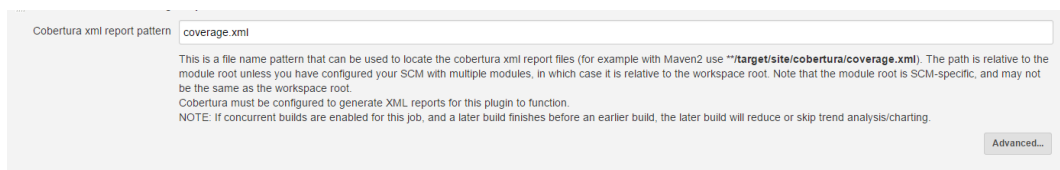
Mit dem dritten Befehl wird das `pylint.out` File erzeugt, in welchem violations erkannt werden.

3.1.4 Was passiert mit Ergebnissen

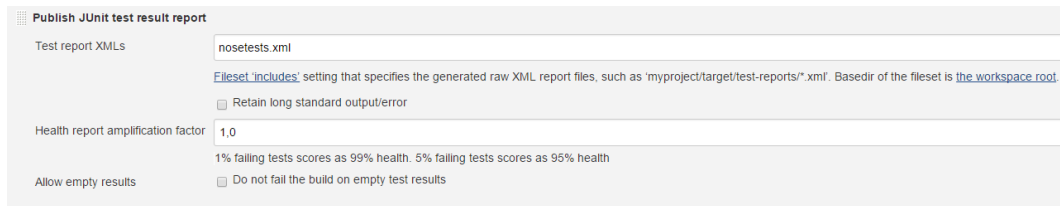
In diesem Schritt wird nun entschieden wie die Ergebnisse analysiert werden. In diesem Fall werden sie interpretiert durch

- Coverage
- JUnit testing
- Report Violations

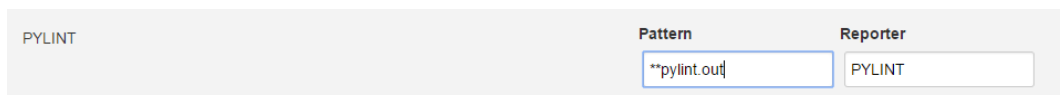
Unter **Post-build Actions** → **Publish Cobertura Coverage Report** wird der Link zum Coverage File angegeben. Es wird lediglich `coverage.xml` angegeben, weil dieses durch den Code im Schritt davor automatisch erzeugt wird.



Als nächstes werden die JUnit Test results hinzugefügt. Dazu unter **Post-build Actions** → **Publish JUnit test result report** wieder den Filenamen angegeben vom Code erzeugten File, `nosetest.xml`.



Der letzte Schritt ist es Report Violations zur Interpretation hinzuzufügen. Unter **Post-build Actions** → **Report Violations to GitHub** → **PYLINT** im Feld **Pattern** folgendes eingeben: `**/pylint.out`. Dieses File wird auch durch den bereits erwähnten Code erstellt.



Der Job muss nun nur mehr gespeichert werden.

3.2 Job ausführen

Dazu wird im Dashboard auf **Build Now** gedrückt. Falls alles funktioniert sollte nun der Punkt Blau aufscheinen:



Man kann nun den Statusbericht einsehen, indem man auf den build drückt:

Build #18 (18-Oct-2017 11:25:50)



No changes.



Started by user [Martin Wölfer](#)



Revision: 735e11afe9a73d8a99592def024d4fcb23d65879

• refs/remotes/origin/master



[Cobertura Coverage Report](#)

Packages: 100% **Files:** 100% **Classes:** 100% **Lines:** 100% **Conditionals:** 100%



[Test Result](#) (no failures)

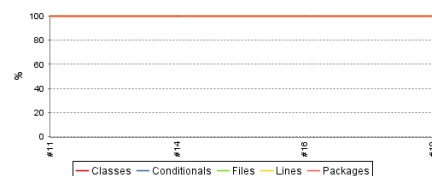
3.2.1 Code Coverage

Durch cobertura kann das erstellte `coverage.xml` file eingelesen werden und verarbeitet werden. Das Ergebnis sollte folgendermaßen aussehen:

Code Coverage

Cobertura Coverage Report

Trend



Project Coverage summary

Name	Packages	Files	Classes	Lines	Conditionals
Cobertura Coverage Report	100% 1/1	100% 1/1	100% 1/1	100% 249/250	100% 0/0

Coverage Breakdown by Package

Name	Files	Classes	Lines	Conditionals
-	100% 1/1	100% 1/1	100% 249/250	N/A

Man sieht nun das vollkommene Coverage herrscht

3.2.2 Test Results

Man kann sich auch alle Tests im genauen ansehen. Diese werden in ihre einzelnen Kategorien gegliedert um eine leichte Navigation zu gewährleisten:

Test Result : Testall

0 failures (±0)

64 tests (±0)

[Took 14 ms.](#)[add description](#)**All Tests**

Class	Duration	Fail (diff)	Skip (diff)	Pass (diff)	Total (diff)
TestAddition	14 ms	0	0	8	8
TestAllgemein	14 ms	0	0	17	17
TestDivision	14 ms	0	0	12	12
TestIteration	14 ms	0	0	3	3
TestMultiplikation	14 ms	0	0	8	8
TestString	14 ms	0	0	2	2
TestSubtraktion	14 ms	0	0	8	8
TestVergleich	14 ms	0	0	6	6

3.2.3 Violations report

Es wurde versucht einen Violations report zu erzeugen, welcher auch erzeugt wird im `pylint.out` File.

Dieser wird jedoch im Build nicht angezeigt aus einem unerklärlichem Grund.

Abbildungsverzeichnis

1	Ein Admin User wurde erstellt	2
2	Daten werden eingegeben für Git	3
3	Neuen Job erstellen	3
4	Pfad konnte angegeben werden ohne Fehlermeldung	4