

**Laborprotokoll**

**Continuous Integration**

**Softwareentwicklung**

**5BHITT 2015/16**

**Manuel Reiländer**

|  |  |
| --- | --- |
|  | **Version 0.1** |
| **Note:** | **Begonnen am 9. Februar 2016** |
| **Betreuer: D. Dolezal, L. Vittori** | **Beendet am 16. Februar 2016** |

Inhalt

[1 Einführung 3](#_Toc443399911)

[1.1 Ziele 3](#_Toc443399912)

[1.2 Voraussetzungen 3](#_Toc443399913)

[1.3 Aufgabenstellung 4](#_Toc443399914)

[2 Ergebnisse 5](#_Toc443399915)

[2.1 Installieren von jenkins auf einer Debian 8 (jessie) Distribution 5](#_Toc443399916)

[2.2 Installieren von Plugins 6](#_Toc443399917)

[2.3 Integrieren eines Github Projektes 8](#_Toc443399918)

[2.4 Automatisches Testen 10](#_Toc443399919)

[2.5 Darstellen/Interpretieren der Testergebnisse 11](#_Toc443399920)

[2.6 Der erste Build 13](#_Toc443399921)

[3 Literaturvezeichnis 14](#_Toc443399922)

[3.1 Zeitaufwand 15](#_Toc443399923)

# Einführung

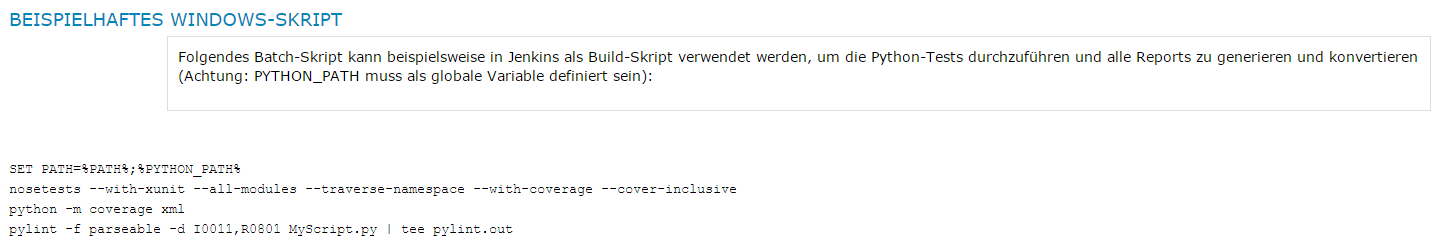
Dieses Protokoll dient als Hilfestellung zur Implementierung/Verwendung von Jenkins.

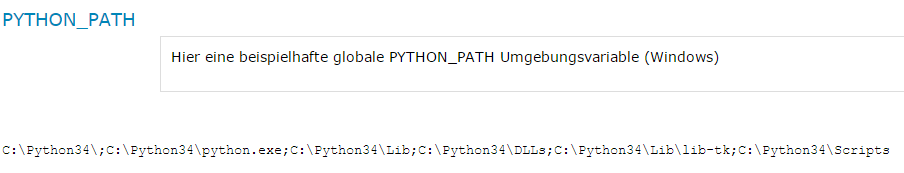
## Ziele

Erwerben der Kompetenz „Implementierung/Verwendung eines Continuous Integration Systems“.

## Voraussetzungen

* <http://docs.python-guide.org/en/latest/scenarios/ci/>
* <https://jenkins-ci.org>
* <http://bhfsteve.blogspot.co.at/2012/04/automated-python-unit-testing-code.html>
* <http://bhfsteve.blogspot.co.at/2012/04/automated-python-unit-testing-code_20.html>
* <http://bhfsteve.blogspot.co.at/2012/04/automated-python-unit-testing-code_27.html>





## 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*

Schreibe fünf Testfälle für dein CSV-Projekt und lass diese mithilfe von Jenkins automatisch bei jedem Build testen!

* Installiere auf deinem Rechner bzw. einer virtuellen Instanz das Continuous Integration System Jenkins
* Installiere die notwendigen Plugins für Jenkins (Git Plugin, Violations, Cobertura)
* Installiere Nose und Pylint (mithilfe von pip)
* Integriere dein CSV-Projekt in Jenkins, indem du es mit Git verbindest
* Schreibe fünf Unit Tests für dein CSV-Projekt
* 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!)

# Ergebnisse

## Installieren von jenkins auf einer Debian 8 (jessie) Distribution

Alle Befehle, wie sie dort beschrieben sind ausführen

* wget -q -O - http://pkg.jenkins-ci.org/debian/jenkins-ci.org.key | sudo apt-key add –

Neues repository zur sources.list hinzufügen oder neues list-File für jenkins erstellen

* sudo vim /etc/apt/sources.list.d/jenkins.list

Folgendes hinzufügen

* deb http://pkg.jenkins-ci.org/debian binary/

Local Package index updaten

* sudo apt-get update

Jenkins (und dessen Dependencies) installieren

* sudo apt-get -f install Jenkins

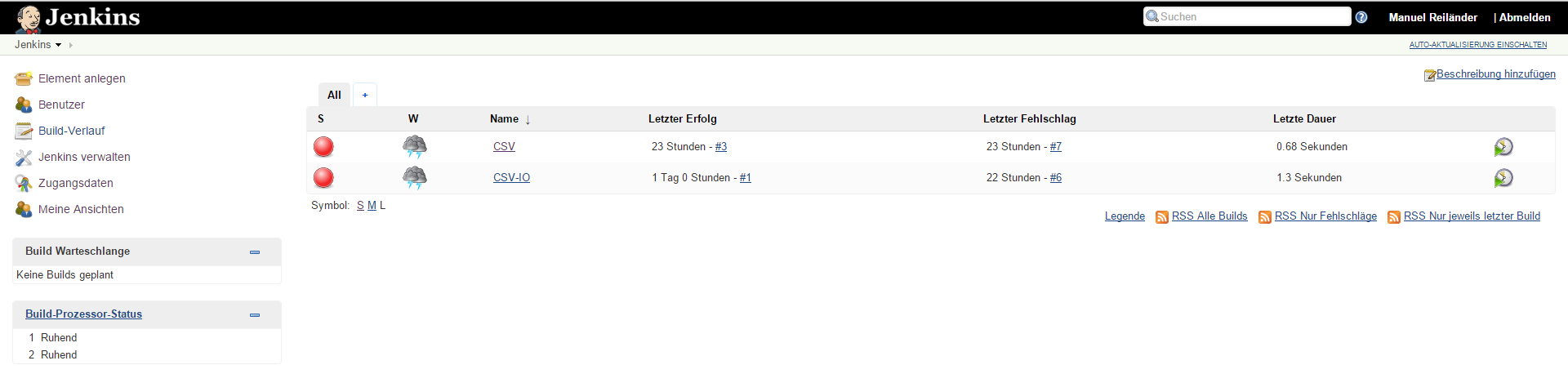
Jenkins starten, falls noch nicht gestartet wurde

* sudo service jenkins start

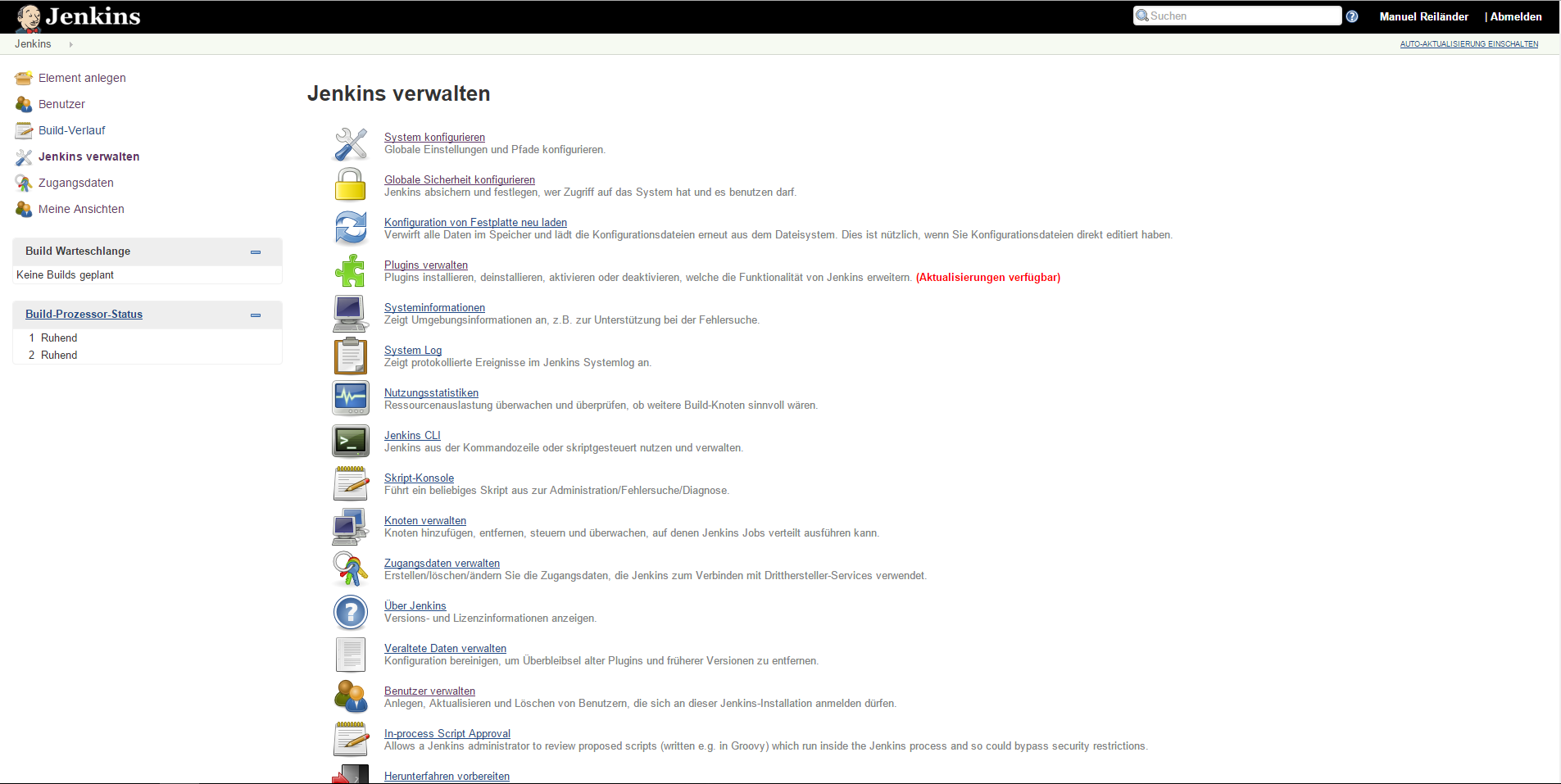
Das admin Interface von jenkins läuft nun auf port 8080 (je nach manueller Konfiguration unterschiedlich)

## Installieren von Plugins

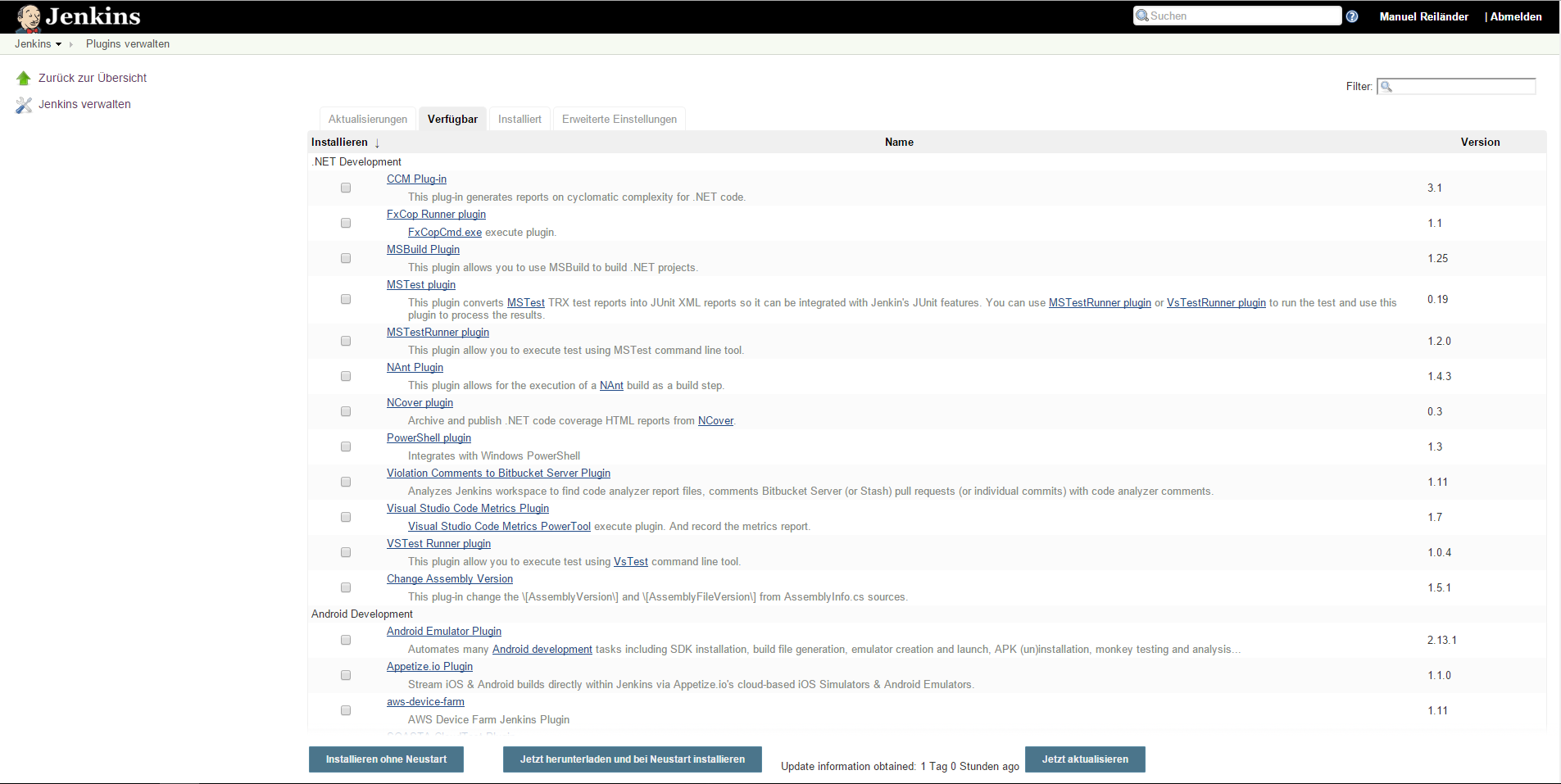
Auf der Weboberfläche auf *„Jenkins verwalten“*



*„Plugins verwalten“*



Im Tab *„Verfügbar“* kann man nun die Plugins auswählen die man installieren möchte.

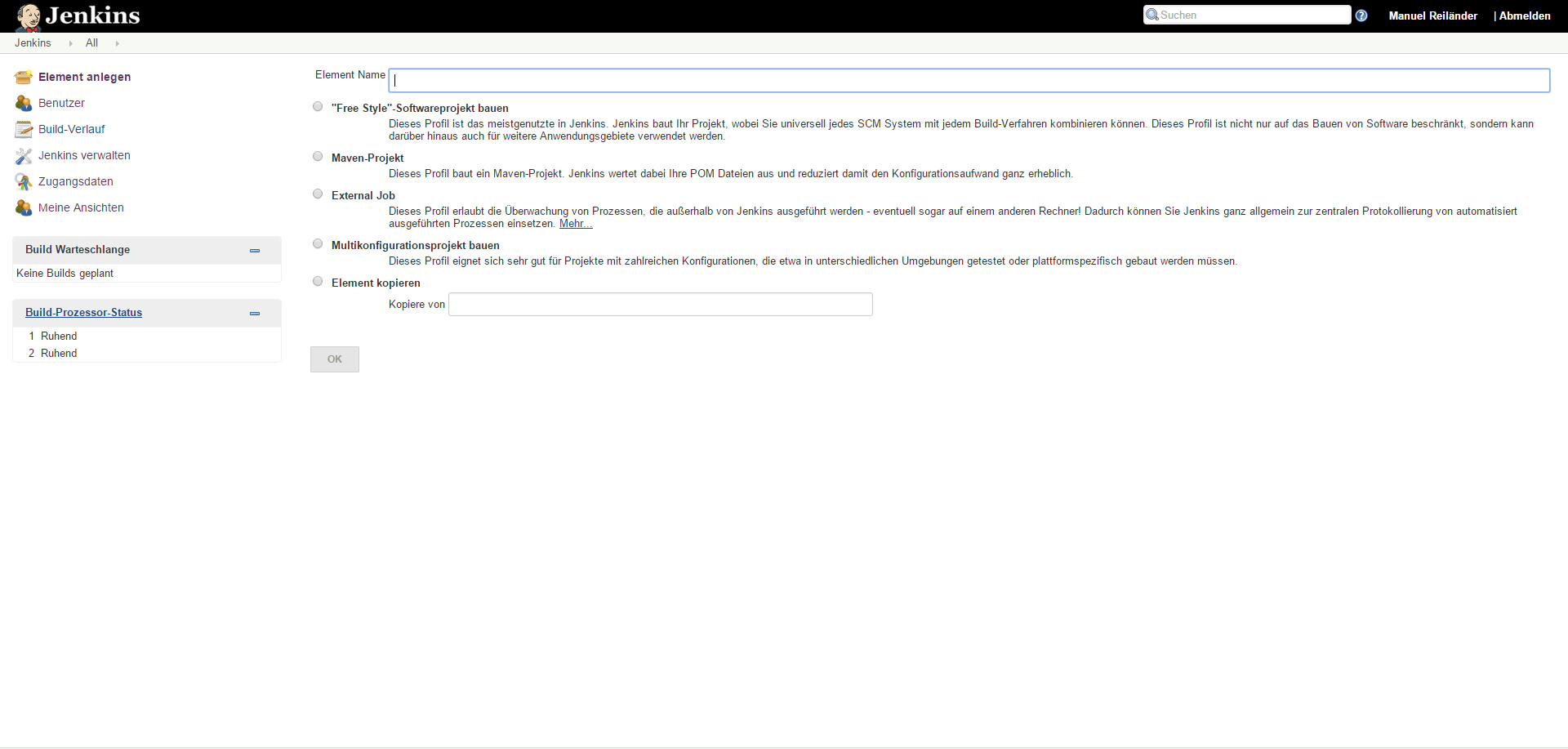


In meinem Fall sind das folgende

* + Git Plugin
  + Violations
  + Cobertura

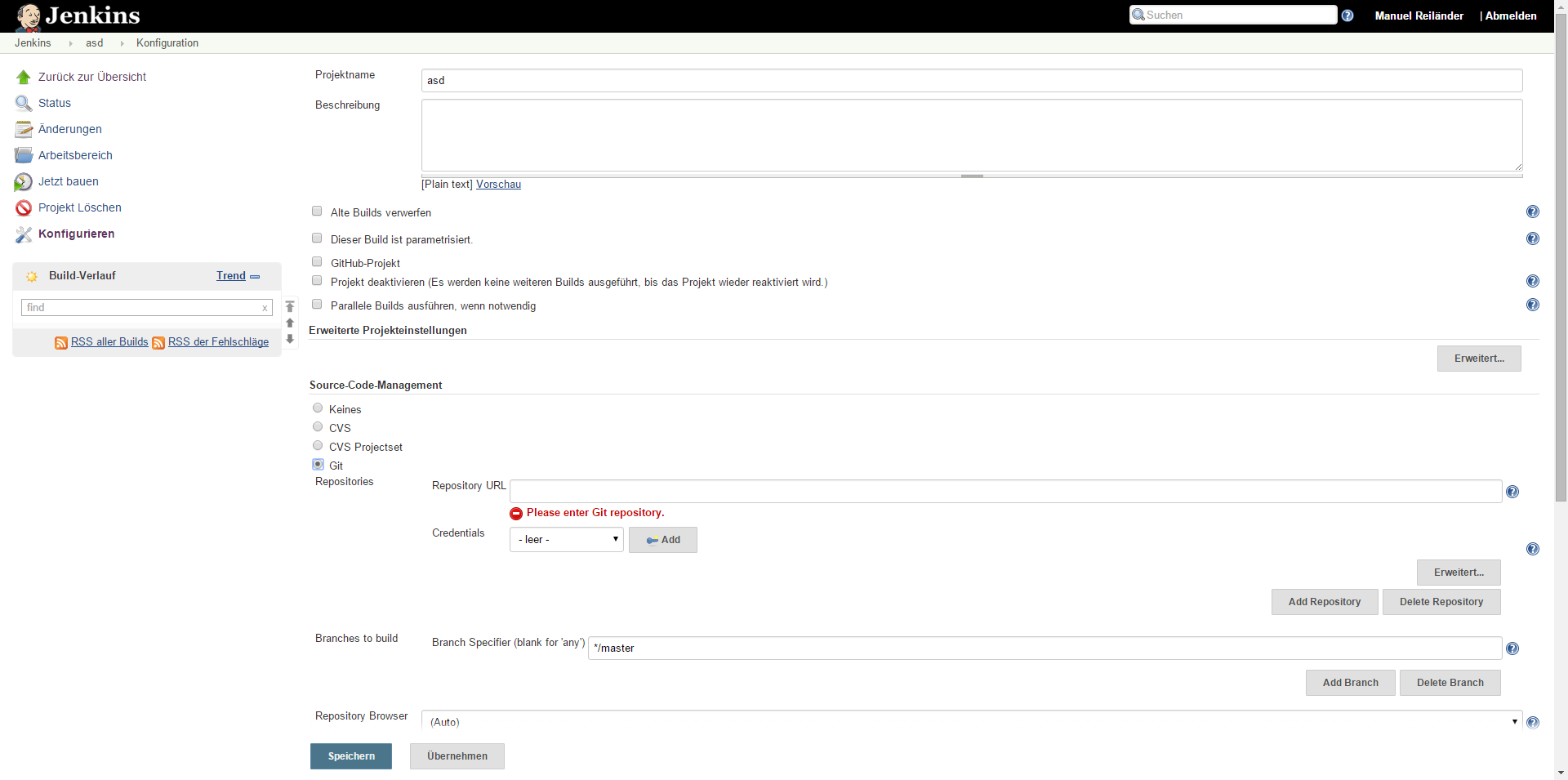
## Integrieren eines Github Projektes

In der Hauptansicht von Jenkins-Admin, klickt man auf *Element anlegen*

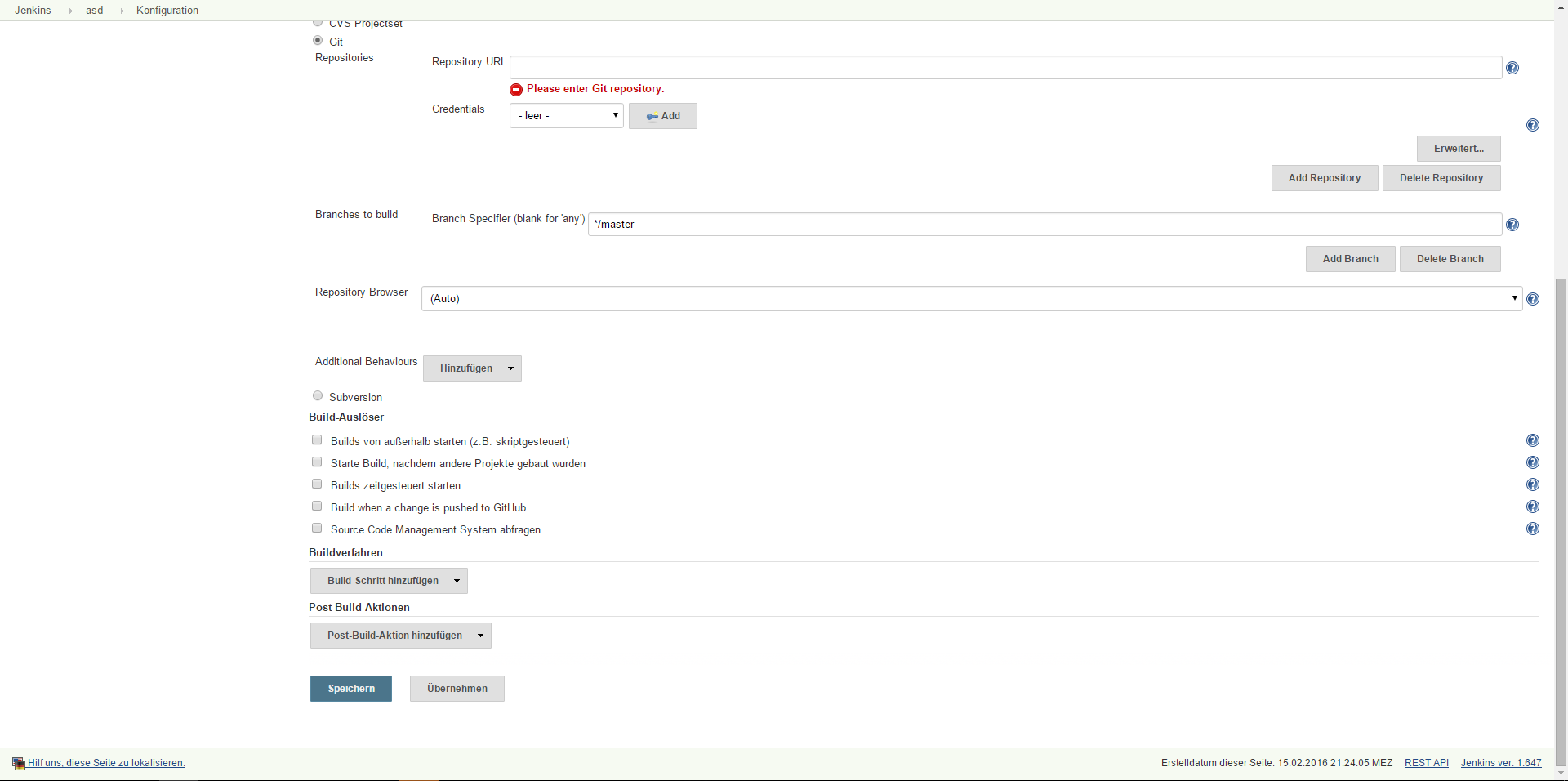


wählt *Free-Style-Softwareprojekt* aus und gibt dem Element einen Namen.

In der darauffolgenden Ansicht, wählt man unter dem Punkt *Source-Code Management*, *git* aus sofern man im vorherigen Schritt das entsprechende *Git Plugin* installiert hat.

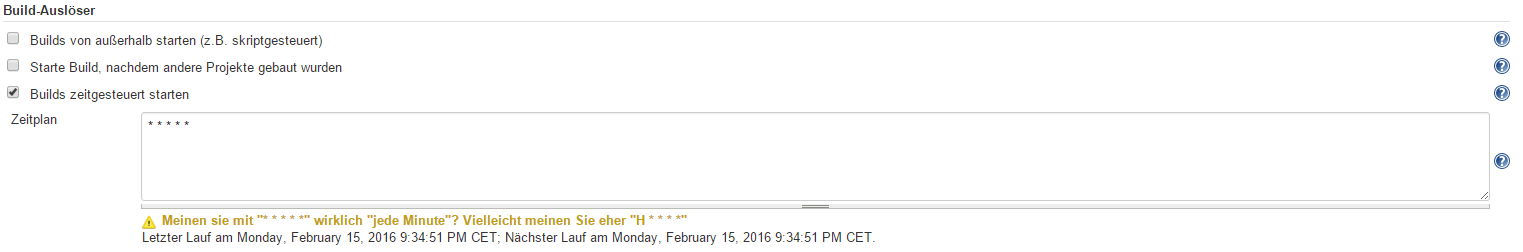
Nun kann man die URL des Repositories im entsprechenden Feld angeben und unter *Branches to build*, den Branch angeben, der gebuildet werden soll.

Unter credentials, kann man nun seinen Account auswählen, dazu muss dieser jedoch erst hinzugefügt werden. Dies passiert mit dem Button *Add*. Im nachfolgenden Fenster kann man nun seine Git-Daten eingeben, hierbei hat man viele verschiedene Möglichkeiten, unter anderen z.B. SSH-Keys oder Benutzername und Passwort.



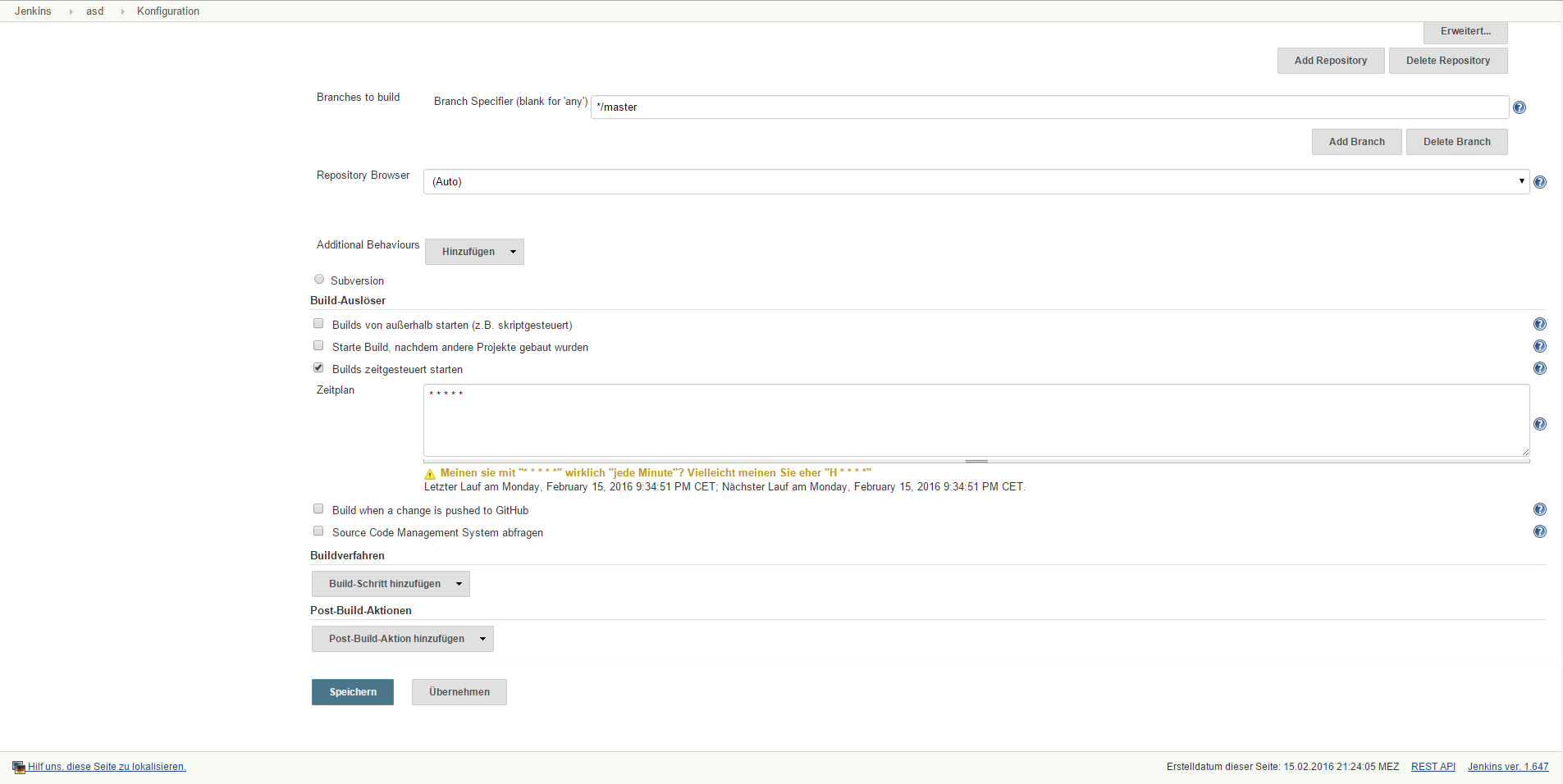
## Automatisches Testen

Bei *Build-Auslöser*, kann man nun einen Auslöser definieren, der Jenkins dazu bewegt das Projekt zu builden. Zu Testzwecken habe ich hier unter *Build zeitgesteuert starten*, jede Minute eingestellt, was man mit 5 Sternen kennzeichnet, wie im Bild zu sehen ist.



Hierbei gilt es zu beachten, die Abstände nach jedem \* einzuhalten.

Als nächsten Schritt, wird ein *Build-Schritt* hinzugefügt, welcher in meinem Fall die Ausführung eines Kommandos in der Shell ist.



Hierzu klickt man im Bereich *Buildverfahren* auf *Build-Schritt hinzufügen* und dann auf *Shell ausführen*. Hier kann man nun die Befehle eintragen, die in der Shell ausgeführt werden sollen, wie im folgenden Bild zu sehen ist.

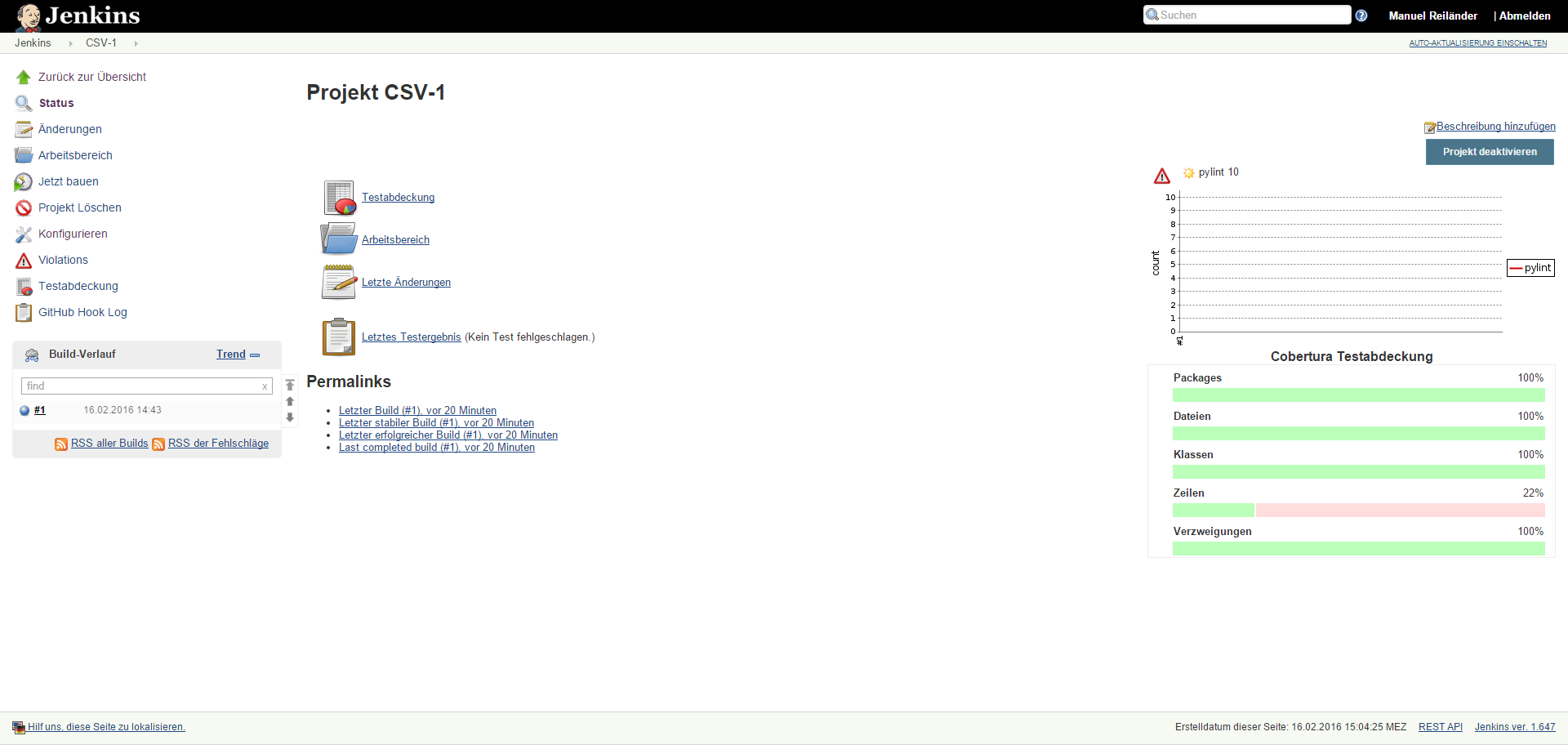


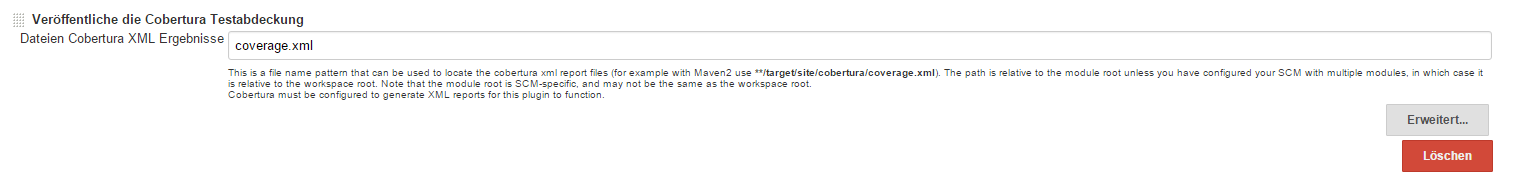
In meinem Fall sind das folgende

* + *nosetests-3.4 --with-xunit --all-modules --traverse-namespace --with-coverage --cover-inclusive*
    - nosetests testet alle test files in einem directory
    - generiert eine *nosetests.xml* und *.coverage*
  + *python3.4 -m coverage xml*
    - hierzu muss folgendes modul via pip installiert werden
      * pip3 install coverage
    - verwendet das zuvor erstellte *.coverage* file und generiert daraus ein *coverage.xml* file
  + *pylint -f parseable -d I0011,R0801 src/\* | tee pylint.out*
    - testet die code qualität eines bestimmten directories welche hier *src/\** ist.

## Darstellen/Interpretieren der Testergebnisse

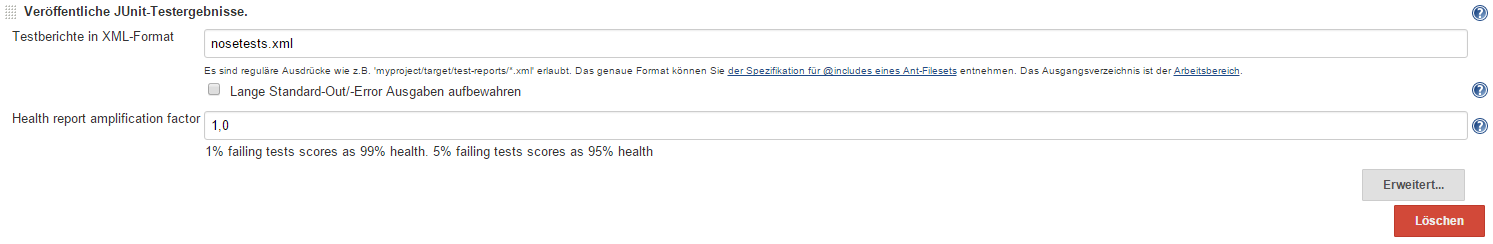
Nun kann jenkins die Ergebnisse des testens auch grafisch darstellen/interpretieren. Dies kann man im Bereich *Post-Build-Aktion hinzufügen* konfigurieren. Hierzu klickt man auf den *Job -> Konfigurieren* und wählt dann unter *Post-Build-Aktion hinzufügen*, *Veröffentliche die Cobertura Testabdeckung* aus.



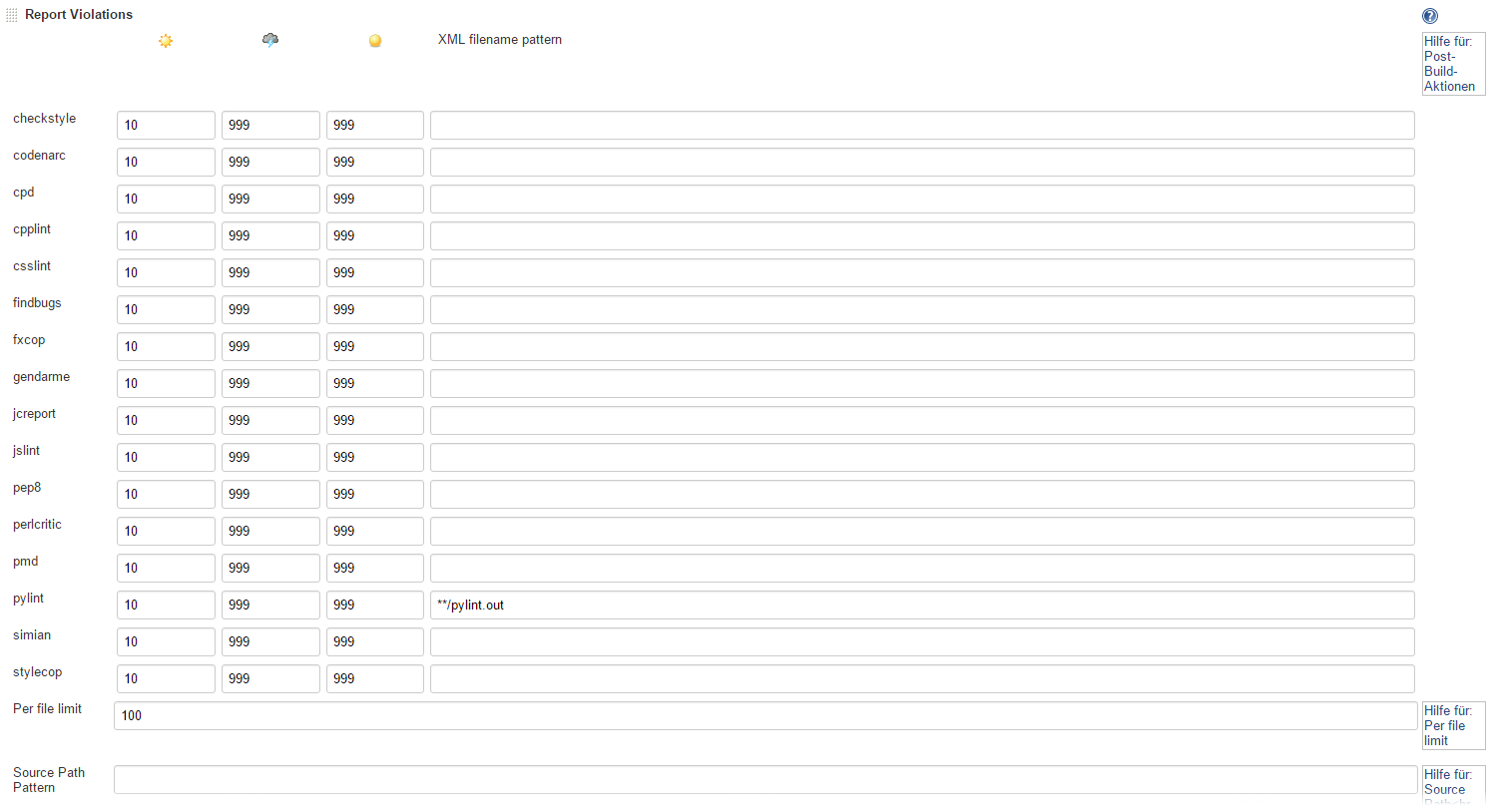


Hier gibt man nun die zuvor erstellte Datei an, indem man einfach den Dateinamen angibt, wie in Abbildung xx zu sehen.

Als nächstes fügen wir eine weitere Post-Build Aktion hinzu, *Veröffentliche J-Unit Testergebnisse* und geben hier das zuvor erstellte *nosetests.xml* an.



Eine letzte Post-Build Aktion fügen wir nun hinzu, *Report Violations*. Hier gibt man dann bei der Zeile pylint, das zuvor erstellte pylint.out file an.

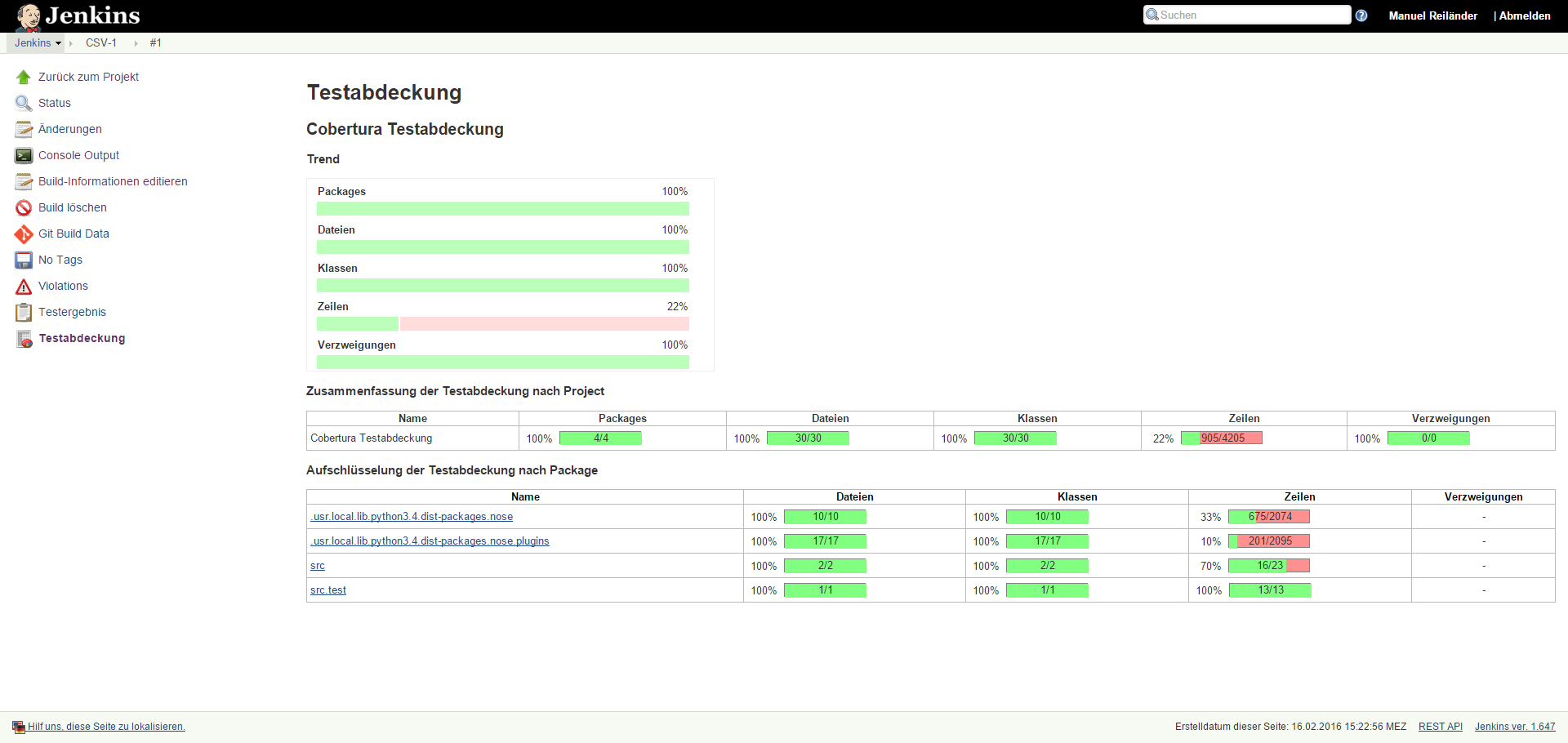


## Der erste Build

Nun ist die Konfiguration abgeschlossen und wir sind bereit zum builden. Hierzu klickt man im Hauptfenster des Jobs, auf *Jetzt bauen*.



Ist alles ohne Probleme erfolgt, wird die Ansicht die im Bild rechts zu sehen grün. Ansonsten kann man sich den *Console-Output* anschauen, indem man auf *Testabdeckung -> Console-Output* klickt.



# Literaturvezeichnis

[1] bhfsteve. *Automated Python Unit Testing with* Jenkins Part 1. Available at: <http://bhfsteve.blogspot.co.at/2012/04/automated-python-unit-testing-code.html>

[2] bhfsteve. *Automated Python Unit Testing with* Jenkins Part 2. Available at: <http://bhfsteve.blogspot.co.at/2012/04/automated-python-unit-testing-code_20.html>

[3] bhfsteve. *Automated Python Unit Testing with* Jenkins Part 3. Available at: <http://bhfsteve.blogspot.co.at/2012/04/automated-python-unit-testing-code_27.html>

## Zeitaufwand

Protokoll ~2h

Durchführung ~4h

Probleme ~2h