Kontinuierliches Bauen und Testen von Software mit Jenkins

Hochschule München
Benjamin Keeser, 31. Mai 2013

Erklärung der Selbstständigkeit

Hiermit versichere ich, die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie die Zitate deutlich kenntlich gemacht zu haben.

(München, den 31. Mai 2013) (Benjamin Keeser)

Inhaltsverzeichnis

1 Einleitung 1
1.1 Continuous Integration (CI) . . . 1
1.2 Apache Maven . . . 3
1.3 Jenkins . . . 3

2 Apache Maven 4
2.1 Grundlagen . . . 4
2.2 Verwaltung von Abhängigkeiten . . . 7
2.3 Erweiterte Konzepte . . . 9

3 Jenkins 12
3.1 Grundlagen . . . 13
3.2 Builds verwalten . . . 15
3.3 Plugins . . . 16

4 Continuous Integration in der Praxis 17
4.1 Tapestry Beispiel Anwendung . . . 17
4.2 Maven als zentrales Build-Tool . . . 18
4.3 Die Integration in Jenkins . . . 23

5 Zusammenfassung und Ausblick 30

Literaturverzeichnis 31

Abbildungsverzeichnis 32

Tabellenverzeichnis 33

1 Einleitung

Um die Softwarequalität von Projekten zu steigern und deren Stabilität dauerhaft zu garantieren haben sich in den letzten Jahren verschiedene Buildsystem-Lösungen unter dem Schlagwort „Continuous Intergration“ etabliert. Was sich genau hinter diesem Begriff verbirgt, ist im einführenden Kapitel dieser Arbeit nachzulesen. Dabei werden die wesentlichen Eigenschaften von Jenkins näher beschrieben, um anschließend eine mögliche Umsetzung an einem konkreten Projekt zu demonstrieren. Hierfür wird es im Kapitel 2 zunächst noch eine kurze Einführung in das Build Management Tool Apache Maven geben.

1.1 Continuous Integration (CI)

Nach Martin Fowler wird das Kernkonzept von Continuous Integration wie folgt beschrieben:

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 (vgl. [Fow06]).

Dieser Beitrag war für CI wegweisend und das Thema erlangte in den darauf folgenden Jahren eine immer größere Aufmerksamkeit. Vereinfachend kann gesagt werden, dass unter einer kontinuierlichen Integration das Einfügen von neuem oder geändertem Programmcode in ein Ursprungsprojekt zu verstehen ist. Dies geschieht normalerweise über ein Versionsverwaltungssystem wie Git oder Subversion. Die Kontinuität dieser Integration erfolgt dabei in kurzen Abständen (in der Regel mindestens einmal am Tag). Dadurch können Fehler rechtzeitig im Programmcode erkannt werden. Zudem soll damit verhindert werden, dass Softwareprojekte durch eine zu späte Zusammenstellung der verschiedenen Komponenten in Gefahr geraten zu scheitern. Durch eine frühzeitige Automatisierung des Integration-Prozesses lässt sich ein Scheitern von Projekten oft vermeiden. Hierzu zählen Tests, Dokumentationen und die Bereitstellung der Software-Lösung. Umso komplexer eine Software-Anwendung wird, umso wichtiger ist es das Ergebnis anhand messbarer Größen wie Metriken, der Testabdeckung oder Ähnlichem selbst überprüfen zu können.

Die Vorteile einer häufigen Integration lassen sich wie folgt zusammenfassen:

- Der Integrationsaufwand sinkt durch frühzeitige Warnungen bei auseinander laufenden Projekt-Bestandteilen. Damit wird auch die Fehlersuche vereinfacht.
- Die Risiken werden durch häufige, reproduzierbare Integrationen verringert.
- Dokumentationen können in regelmäßigen Abständen automatisch erstellt werden. Hierzu zählen z.B. Berichte zu Code-Analysen, Junit-Testergebnisse oder eine Api-Dokumentation durch Javadocs.
- Die Teammoral steigt durch häufige Rückmeldungen an die Entwickler.

Martin Fowler zählt, in dem Eingangs bereits erwähnten Artikel, 10 Praktiken als Voraussetzung für ein sinnvolles CI auf (vgl. [Wie11, S.16-18]):

- Gemeinsame Codebasis - alle Projekt-Daten müssen an einen zentralen Ort gehalten werden. Im Normalfall geschieht dies über ein Versionverwaltungssystem wie Git oder Subversion.
- Automatisierte Builds - die Anwendung muss automatisch übersetzt und generiert werden.
- Testen der Builds - die Software Komponenten werden während des Builds auf korrekte Funktionsweise überprüft.
- Häufige Integration - mindestens einmal am Tag sollte Code der unterschiedlichen Entwickler in ein Versionskontrollsystem übermittelt werden.
- Builds nach Codeänderung - nach jeder Code-Änderung sollte automatisch getestet und gebaut werden.
- Schnelle Build-Zyklen - zwischen dem Einchecken einer Änderung und dem Rückmelden des CI-System sollte möglichst wenig Zeit vergehen.
- Tests - Tests sollten in einer realitätsnahen Umgebung ausgeführt werden.
- Einfacher Zugriff - der aktuelle Stand des Software Projekts muss für alle daran Beteiligten leicht einsehbar sein.
- Automatische Berichte - als Build-Ergebnisse sollten automatische Berichte erstellt und an die Entwickler weitergeleitet werden.
- Automatisierte Verteilung - Verteilung der jeweils erstellten Software Komponenten (z.B. durch ein Deployment auf einen Testserver).

1.2 Apache Maven

Maven[1] bietet bereits ein reiches Repertoire an Funktionalität, welches dem Grundgedanken von „Convention over Configuration“ nahe kommt. Über wenige Konfigurationsschritte lässt sich der komplette Lebenszyklus einer Software-Anwendung bei der Erstellung mit Maven bereits automatisieren. Die wichtigste Funktion ist dabei die Auflösung von Abhängigkeiten zu anderen Softwarepaketen, die automatisiert vom Repository-Servern im Internet geholt werden (sollten sie lokal nicht vorhanden sein). Unterstützt werden neben dem Testen und Kompilieren des Quellcodes auch Analyse-Tools wie Checkstyle[2] oder PMD[3] (siehe Kapitel 3.3). Die Definitionen müssen in der Konfigurationsdatei *pom.xml*, welche sich im Root-Verzeichnis eines Projektes befinden sollte, festgelegt werden (siehe Kapitel 2.1).

1.3 Jenkins

Jenkins[4] wurde durch einen Mitarbeiter von Sun Microsystems 2004 ins leben gerufen, und war damals noch unter dem Namen Hudson bekannt. Es ist eine auf Java basierender, leicht erweiterbarer Continuous-Integration Server.

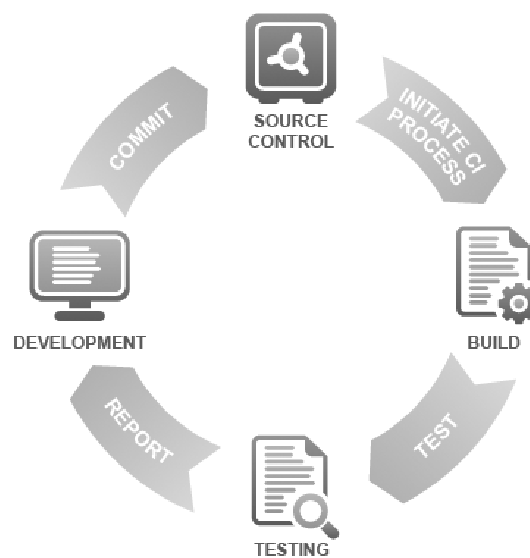

Abbildung 1.1: Continuous Integration Prozess

1 http://maven.apache.org - zuletzt besucht: 31.05.2013
2 http://checkstyle.sourceforge.net - zuletzt besucht: 31.05.2013
3 http://pmd.sourceforge.net - zuletzt besucht: 31.05.2013
4 http://jenkins-ci.org - zuletzt besucht: 31.05.2013

2 Apache Maven

Mit Hilfe eines Build (Management) Tools wird das Zusammenstellen einer Software-Anwendung automatisiert. Apache Maven bietet hierfür Basisfunktionalität zum Kompilieren aller nötigen Dateien und zur Auflösung von Software Abhängigkeiten an (Dependency Management). Auch die Zusammenstellung kompletter Softwarepakete in komprimierter Form zählt, neben der Möglichkeit von Umfangreichen Tests, zu seiner Kernfunktionalität. Dadurch lassen sich mit Maven die verschiedenen Schritte eines Build-Prozesses zusammenfassen. Apache Maven existiert als reine Konsolen-Anwendung und muss über eine Shell ausgeführt werden.
Für alle Builds gilt dabei die Regel, dass sie immer Reproduzierbar bleiben müssen. Auch wenn sie später noch einmal auf einem anderen System ausgeführt werden.

2.1 Grundlagen

Der Buildprozess unter Maven ist fest vorgegeben und wird über eine Konfigurationsdatei gesteuert (*pom.xml*). Er durchläuft verschiedene Stadien, wobei Standardkonfigurationen bereits durch „Convention over Configuration“ festgelegt sind. Abhängigkeiten zu anderen Software-Komponenten werden während dem Prozess automatisch aus dem Internet geladen.

Das Project Object Model (POM) dient während dem Build-Prozess der eindeutigen Identifizierung von Projekten und fungiert als zentrales Steuerelement. Darin festgelegt sind Format, Ort der Quelldateien, Aufbau des Projekts sowie sämtliche Abhängigkeiten. Die eindeutige Identifizierung findet über sogenannte *Projektkoordinaten* statt.

- **groupId** - Name der Organisation (oft Domainname)
- **artifactId** - Name des Projekts (Modul etc.)
- **version** - Versionsnummer des Projekts
- **scope** - Sichtbarkeit der Abhängigkeit (z.B. compile, runtime, test).
- **packaging** - Auslieferungszustand (jar, war, zip, apk, ...)

Um Maven zu starten muss in ein Verzeichnis mit einer vorhandenen Maven-Konfigurationsdatei (*pom.xml*) gewechselt werden.

```
<project>
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.mycompany.app</groupId>
    <artifactId>my-app</artifactId>
    <version>1.0</version>
    <dependencies>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>3.8.1</version>
            <scope>test</scope>
        </dependency>
    </dependencies>
</project>
```

Abbildung 2.1: Grundgerüst eines minimalen POM

```
usage: mvn [options] [<goal(s)>] [<phase(s)>]
```

Abbildung 2.2: mvn in einer Shell - mvn –help > Anzeige der verfügbaren Optionen

Die Build-Modularisierung findet durch unterschiedliche Phasen statt, was den allgemeinen Lifecycle eines Maven-Builds entspricht.

Dabei gliedern sich die **Lifecycles** wie folgt auf:

- **Clean Lifecycl**e - pre-clean, clean, post-clean
- **Default Lifecycle** (Build-Phase) - validate, compile, test, package, integration-test, verify, install, deploy
- **Site Lifecycle** - pre-site, site, post-site, site-deploy

Sämtliche Beschreibungen zu den Lifecycle-Phasen können unter folgender URL eingesehen werden: http://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html

Eine Lifecycle-Phase wird wie folgt aufgerufen:

mvn install

Beim Ausführen der unterschiedlichen Build-Phasen sind auch Kombinationen möglich.

Z.B. **mvn clean compile**

was einem clean mit einem anschließendem Start der Kompilierungs-Phase entspricht.

Um ein Goal aufzurufen kann dieser an eine Livecycle-Phase gehängt werden. Folgender Befehl würde z.B. eine Anpassung des Codes für den Import unter Eclipse bewirken (Import > Existing Project into Workspace):

mvn eclipse:eclipse

Plugins können Aktionen an bestimmte Phasen hängen. Mit dem Befehl

mvn jar:jar

würde z.B. eine entsprechende JAR-Datei aus dem Quellcode erstellt werden.
Plug-ins bestehen damit aus Goals und sind eindeutig einer Phase zugeordnet.

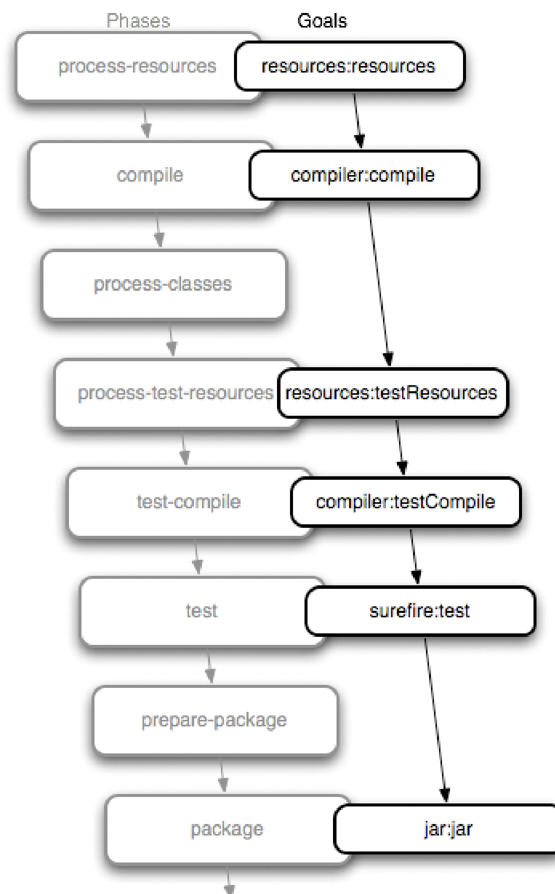

Abbildung 2.3: Apache Maven: Phasen und Goals

2.2 Verwaltung von Abhängigkeiten

Jedes Java-Projekt enthält Abhängigkeiten in Form von externen Bibliotheken oder Submodulen (Open-Source/Kommerziell). Unter Apache Maven werden abhängige Module im POM deklariert. Eine Referenzierung erfolgt hier über die *Projektkoordinaten* (siehe Kapitel 2.1). Falls sich die abhängigen Komponenten nicht im lokalen Repository befinden (~/.m2) werden sie automatisch aus dem Internet geladen.

```
<dependency>
    <groupId>org.apache.tapestry</groupId>
    <artifactId>tapestry-hibernate</artifactId>
    <version>${tapestry-release-version}</version>
</dependency>

<dependency>
    <groupId>hsqldb</groupId>
    <artifactId>hsqldb</artifactId>
    <version>1.8.0.7</version>
</dependency>
```

Abbildung 2.4: Abhängigkeit zu einer Software-Komponente

Je Modul wird immer bestimmt ob es

- für eine Auslieferung bestimmt ist > **scope:** compile, runtime
- in der Zielumgebung bereits vorhanden ist > **scope:** system.
- nur für Tests bestimmt ist > **scope:** test.

```
<dependency>
    <groupId>org.testng</groupId>
    <artifactId>testng</artifactId>
    <version>5.12.1</version>
    <scope>test</scope>
</dependency>

<dependency>
    <groupId>org.easymock</groupId>
    <artifactId>easymock</artifactId>
    <version>2.5.2</version>
    <scope>test</scope>
</dependency>
```

Abbildung 2.5: Beispiel für POM Abhängigkeiten

Automatisches Einbinden von Abhängigkeiten

Vorteile

- Durch einfaches Einbinden von Abhängigkeiten.
- Das Management transitiver Abhängigkeiten entfällt.
- Die Bibliotheken liegen nicht im Versionskontrollsystem.

Nachteile

- U.U. sehr langes laden aus dem Internet beim ersten Build.
- Der Überblick kann verloren gehen („Dependency Hell“).

Die POM Abhängigkeiten werden immer in 3 Stufen geladen: Lokales Repository (wichtig für Offline-Builds), Firmen-Repository (Verteilungsplattform für Firmen-Bibliotheken), Internet (Maven Central)[1].

1 http://search.maven.org - zuletzt besucht: 31.05.2013

2.3 Erweiterte Konzepte

Durch die Definition eines Parent-POM unter Maven kann sämtliche dort festgelegte Funktionalität vererbt werden. Zur Laufzeit werden diese POMs in ein *effective POM* übersetzt.

```
<project xmlns="http://maven.apache.org/POM/4.0.0
  <parent>
    <artifactId>magnolia-project</artifactId>
    <groupId>info.magnolia</groupId>
    <version>4.5.8</version>
    <relativePath>../pom.xml</relativePath>
  </parent>
  ...
```

Abbildung 2.6: Beispiel für eine POM Vererbung

Auch eine Aggregation durch Submodule ist möglich. Maven ruft diese dann rekursiv auf.

```
  ...
  <modules>
    <module>serviceplan-events</module>
    <module>serviceplan-events-webapp</module>
  </modules>
</project>
```

Abbildung 2.7: Beispiel für eine POM Aggregation

Darüber hinaus können vordefinierte Grundgerüste für Projekte automatisch über Maven generiert werden. Dies geschieht durch sogenannte *Archetypes* (vordefinierte Templates). Für komplizierte Projekte kann das durchaus sinnvoll sein, da damit bereits funktionsfähige Demo-Anwendungen ausgeliefert werden können. Hier ein Beispiel eines Archetypes, dass ein komplexes CMS erstellt (hierfür wird Maven 2.2.1 benötigt):

mvn archetype:generate
-DarchetypeCatalog=http://nexus.magnolia-cms.com/content/groups/public

Durch ein anschließendes Wechseln in die *artifactId* (magnolia-webapp) und dem Ausführen von folgendem Maven Befehl wird ein komplexes, funktionsfähiges CMS generiert (zu finden im Projekt-Verzeichnis unter *target*: magnolia-webapp-1.0-SNAPSHOT.war):

mvn install

```
Choose archetype:
1: http://nexus.magnolia-cms.com/content/groups/public/ -> info.magnolia:maven-archetype-magnolia-webapp (-)
2: http://nexus.magnolia-cms.com/content/groups/public/ -> info.magnolia:maven-archetype-magnolia-module (-)
3: http://nexus.magnolia-cms.com/content/groups/public/ -> info.magnolia.maven.archetypes:magnolia-module-archetype (An ar
4: http://nexus.magnolia-cms.com/content/groups/public/ -> info.magnolia.maven.archetypes:magnolia-forge-module-archetype
 the Magnolia Forge)
5: http://nexus.magnolia-cms.com/content/groups/public/ -> info.magnolia.maven.archetypes:magnolia-theme-archetype (An arc
6: http://nexus.magnolia-cms.com/content/groups/public/ -> info.magnolia.maven.archetypes:magnolia-project-archetype (An a
 webapp))
Choose a number or apply filter (format: [groupId:]artifactId, case sensitive contains): : 6
Downloading: http://nexus.magnolia-cms.com/content/groups/public/info/magnolia/maven/archetypes/magnolia-project-archetype
Downloaded: http://nexus.magnolia-cms.com/content/groups/public/info/magnolia/maven/archetypes/magnolia-project-archetype/
Downloading: http://nexus.magnolia-cms.com/content/groups/public/info/magnolia/maven/archetypes/magnolia-project-archetype
Downloaded: http://nexus.magnolia-cms.com/content/groups/public/info/magnolia/maven/archetypes/magnolia-project-archetype/
Define value for property 'groupId': : net.tatura
Define value for property 'artifactId': : magnolia
Define value for property 'version':  1.0-SNAPSHOT: :
Define value for property 'package':  net.tatura: :
Define value for property 'magnolia-version': : 4.5.8
```

Abbildung 2.8: Beispiel für ein POM Archetype

Das generierte WAR muss nur noch in einen entsprechenden Servlet Container wie Tomcat gelegt werden und nach einem Installationsprozess kann das CMS bereits genutzt werden (http://localhost:8080/magnolia-webapp-1.0-SNAPSHOT).

Vorbelegte Werte können in Maven innerhalb von Properties definiert und an verschiedenen Stellen wiederverwendet werden.

```
<description>Serviceplan Events Modul</description>
<properties>
  <magnoliaVersion>4.5.8</magnoliaVersion>
  <javaVersion>1.6</javaVersion>
</properties>

<dependencies>
  <dependency>
    <groupId>info.magnolia</groupId>
    <artifactId>magnolia-core</artifactId>
    <version>${magnoliaVersion}</version>
  </dependency>
  ...
```

Abbildung 2.9: *magnoliaVersion* als festgelegtes Beispiel-Property

Unter Maven existieren bereits vorbelegte Variablen, die innerhalb von Konfigurationsdateien benutzt werden können:

- **${project.basedir}** - Pfad zum Verzeichnis in dem sich die *pom.xml* befindet
- **${project.build.outputDirectory}** - Zielverzeichnis (*target*)
- **${project.name}** - Name des Projekts
- **${project.version}** - Version des Projekts

Mit Maven können zudem Plugins eingebunden werden, die z.B. Code-Analysen Übersichtlich in Berichten erstellen. Auch die Möglichkeit, Quellcode-Dokumentationen mit UML-Diagrammen zu erstellen, ist machbar (siehe Kapitel 4.2).

```
<plugins>
    <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-pmd-plugin</artifactId>
        <version>3.0.1</version>
        <configuration>
            <minimumTokens>100</minimumTokens>
            <verbose>true</verbose>
            <rulesets>
                <ruleset>${project.basedir}/configs/pmd.xml</ruleset>
            </rulesets>
        </configuration>
    </plugin>
    ...
```

Abbildung 2.10: Code Analyse Plugin PMD

3 Jenkins

Laut einer von CloudBees durchgeführten repräsentativen Umfrage vom Oktober 2012[1], an der 700 Jenkins-Anwender teilnahmen sind 87% von ihnen zufrieden mit Jenkins. Nicht nur bei Java-Anwendern scheint die Begeisterung über das Projekt zu bestehen, welches 2010 aus einem Fork von Hudson entstanden ist (nachdem Kohsuke Kawaguchi Oracle verließ)[2]. Mittlerweile soll es laut einer Meldung von Heise Developer[3], eine nicht zu unterschätzende Rolle in der C/C++-, JavaScript-, Python- und C#-Entwicklung spielen. Der auf Java basierende Continuous Integration Server Jenkins lässt sich zudem mit seinen über 600 Plugins nach belieben erweitern.

Die grundlegenden Eigenschaften setzen sich wie folgt zusammen (vgl. [Beh12]):

- Einfache Installation.
- Intuitive Bedienung.
- REST-Schnittstelle.
- Simpler und stabiler Aktualisierungsprozess.
- Kompatibilität (fast jede Programmiersprache kann eingerichtet werden).
- Flexibilität und Erweiterbarkeit (es existieren hunderte Plugins).
- Verteilte Build-Vorgänge über mehrere Server.
- Kostenlos, frei und Open Source (MIT-Lizenz).
- Eine hochaktive und professionelle Community.

1 http://www.cloudbees.com/sites/default/files/documents/2012%20Jenkin%20Survey%20Results.pdf - zuletzt besucht: 31.05.2013

2 http://www.heise.de/developer/meldung/Hudson-Chefentwickler-verlaesst-Oracle-971849.html - zuletzt besucht: 31.05.2013

3 http://www.heise.de/developer/meldung/Jenkins-nicht-nur-bei-Java-Entwicklern-beliebt-1780638.html - zuletzt besucht: 31.05.2013

Aus einer weiteren im Januar 2013 von Rebellabs erstellten Studie wird deutlich, dass Jenkins das Projekt mit der größeren und aktiveren Community ist (vgl. [Zer13, S. 3]).

APPROXIMATE NUMBER OF COMMITS OVER THE PAST 12 MONTHS

Jenkins: 2500 Core, 5450 with plugins

Hudson: 500

Abbildung 3.1: Jenkins/Hudson commits

3.1 Grundlagen

Die Installationsverzeichnisse von Jenkins befinden sich je nach Betriebssystem an unterschiedlichen Orten:

- **Windows** - C:\Programme\Jenkins
- **Linux** - /var/lib/jenkins
- **Mac OS X** - /Users/Shared/Jenkins/Home

Innerhalb dieser Pfade existieren weitere Ordner, die für das Verständnis von Jenkins eine wichtige Rolle spielen:

- **jobs** - hier befinden sich sämtliche Konfigurationsdateien, die eigentlichen Builds, sowie der Workspace der dazu gehörenden Jobs.
- **userContent** - alle Dateien die hier abgelegt werden, können über die Weboberfläche zugänglich gemacht werden (http://domain.tld/userContent/datei).
- **users** - Konfiguration und Konten der einzelnen Benutzer (bei Jenkins internen Benutzerverzeichnissen)

Benutzerverwaltung

Eine Zugriffskontrolle kann durch folgende Technologien bereit gestellt werden:

- LDAP - Lightweight Directory Access Protocol[1]
- Unix-Benutzer/Gruppen - bereits im Betriebssystem angelegte Benutzer und Gruppen können verwendet werden.
- Jenkins interne Benutzerverzeichnisse - über Jenkins verwaltete Benutzer.

Die Rechte können über eine Zugriffsmatrix allgemein oder Projektbezogen eingeschränkt werden.

Rechtevergabe

- ○ Angemeldete Benutzer dürfen alle Aktionen ausführen
- ○ Jeder darf alle Aktionen ausführen
- ○ Legacy-Autorisierung
- ○ Matrix-basierte Sicherheit
- ● Projektbasierte Matrix-Zugriffssteuerung

Benutzer/Gruppe	Allgemein				
	Administer	Read	RunScripts	UploadPlugins	ConfigureUpdateCenter
tatura	☑	☐	☐	☐	☐
Anonym	☐	☐	☐	☐	☐

Weitere Benutzer/Gruppe: [ ] Hinzufügen

Abbildung 3.2: Sicherheitsmatrix unter Jenkins

Da Jenkins immer unter einen bestimmten Port läuft lässt sich dies über das a2enmod-Tool z.B. auf den Standard Webport 80 ändern[2].

1 http://www.openldap.org - zuletzt besucht: 31.05.2013
2 https://wiki.jenkins-ci.org/display/JENKINS/Running+Jenkins+behind+Apache zuletzt besucht: 31.05.2013

3.2 Builds verwalten

Projektarten
Mit Jenkins lassen sich 5 unterschiedliche Projektarten über sogenannte Jobs erstellen.

»Free Style«-Softwareprojekt
Hiermit kann ein völlig neues Projekt, ohne vordefinierte Parameter generiert werden. Auch wenn dies nach mehr Arbeit klingt ist es oft der beste und sauberste Schritt auf dem Weg zu einem lauffähigen CI-Prozess. Einmal erstellt, kann es auch als Template für weitere gleichartige Projekte dienen. Freestyle Projekte sind allgemein am flexibelsten und einfachsten an die eigene Bedürfnisse anpassbar.

Maven 2/3-Projekt
Mit Jenkins können Mavenbuilds komfortabel eingebunden werden. Es reicht dafür aus den Pfad zur der jeweiligen Konfigurationsdatei anzugeben (*pom.xml*). Über das Goal-Feld können die gewünschten Optionen für den Build ergänzt werden. Unter den erweiterten Maven Einstellungen befindet sich in Differenz zum „Free-Style“-Projekt die Option „Baue dieses Projekt, wenn eine SNAPSHOT-Abhängigkeit gebaut wurde“ (im Bereich Build-Auslöser). Bei Nutzung dieser Auswahl durchsucht Jenkins die POM-Datei nach Abhängigkeiten zu anderen Projekten, die unter Jenkins-Kontrolle stehen. Falls einer dieser Jobs angestoßen wurde, wird das aktuelle Projekt ebenfalls neu generiert.

Externen Job überwachen
Hiermit lassen sich z.B. EMails, Cronjobs oder Services überwachen. Damit kann festgelegt werden, ob Jobs nur dann ausgeführt werden, wenn dafür benötigte Dienste aktiv sind. Dies kann z.B. ein Selenium-Server sein, der für bestimmte Testergebnisse benötigt wird oder die schlichte Prüfung auf Anwesenheit einer Testdatenbank. Es ist auch möglich eine installierte Version eines Versionkontrollsystem wie Git zur Voraussetzung für einen Jobstart zu machen.

Multikonfigurationsprojekt bauen
Unter dieser Projektart handelt es sich um die komplexeste Möglichkeit einen Jenkins Job zu definieren. Denkbar wäre hier z.B. ein Kompatibilitäts-Test von unterschiedlichen Versionen einer Software. Auch ein Test einer Webanwendung durch alle relevanten Browser ist damit relativ einfach zu verwirklichen (z.B. durch Selenium).

Kopiere bestehenden Job
Durch diese Projektart können bereits angelegte Jobs als Ausgangsbasis wiederverwendet werden. Dadurch lässt sich ein Großteil des Konfigurationsaufwands sparen.

3.3 Plugins

Durch die zahlreich zu Verfügung stehenden Plugins unter Jenkins, lässt sich dessen Funktionsumfang deutlich erweitern. Z. B. ist es möglich dadurch statistische Code Analyse Tools wie Checkstyle einzubinden und sich Berichte zur Codequalität der einzelnen Projekte generieren zu lassen. Es können auch Compiler für Programmiersprachen wie Ruby oder C++ integriert werden.

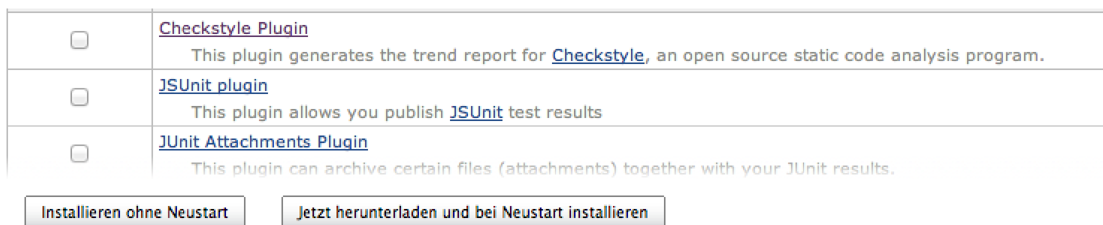

Abbildung 3.3: Plugin-Verwaltung unter Jenkins

Zudem besteht die Möglichkeit ganze Test-Frameworks wie Sonar[1] oder Selenium[2] über Jenkins zu nutzen.

1 http://www.sonarsource.org - zuletzt besucht: 31.05.2013
2 http://docs.seleniumhq.org - zuletzt besucht: 31.05.2013

4 Continuous Integration in der Praxis

Contionuous Integration mit Jenkins scheint immer mehr zur gängigen Praxis von Software-Entwicklungszyklen zu werden. Vieles spricht dafür Software unter eine Kontinuierliche Kontrolle zu stellen. Wie dies genau aussehen kann wird auf den folgenden Seiten anhand einer im Wintersemester 2011/2012 entwickelten Tapestry[1] Anwendung demonstriert.

Die folgende Tabelle ist eine Praxis-Empfehlung über einen Zeitplan von Build-Phasen (vgl. [Zer13, S. 19]):

Job Typ	Zeitplan	Beschreibung	Notwendig
Main-Build	Alle 15 Minuten oder jede halbe Stunde	Nur Kompilierung und Unit-Tests. Sollte nach 15-20 Minuten beendet sein.	Ja
Integration-Build	Alle 24 Stunden (normalerweise nachts)	Es laufen Integrations-Tests, die 2-3 Stunden in Anspruch nehmen können.	Ja
Test-Build	Alle 2-3 Tage (auch möglich bei Nachfrage)	Deployment des gesamten Test-System (alle zum Projekt gehörenden Komponenten).	Nein
Qualitäts-Build	1 x Woche	Es laufen Analyse-Tools zum Testen der Code-Qualität (z.B. Sonar o.ä.).	Nein

Tabelle 4.1: Tabelle mit Build-Phasen

4.1 Tapestry Beispiel Anwendung

SecureFileStorage ist eine Anwendung, über die Daten serverseitig verschlüsselt innerhalb von Truecrypt Containern abgelegt werden können. Das Tapestry Projekt wird über Apache Maven zusammengestellt und fungiert dabei als Dateibrowser und sichere Dateiablage. Es dient hier als Beispiel für eine Überwachung des Build-Prozesses durch Jenkins. Dafür wurde die POM entsprechend angepasst.

Zum testen der Befehle wurde Maven in Version 2.2.1 verwendet.

1 http://tapestry.apache.org - zuletzt besucht: 31.05.2013

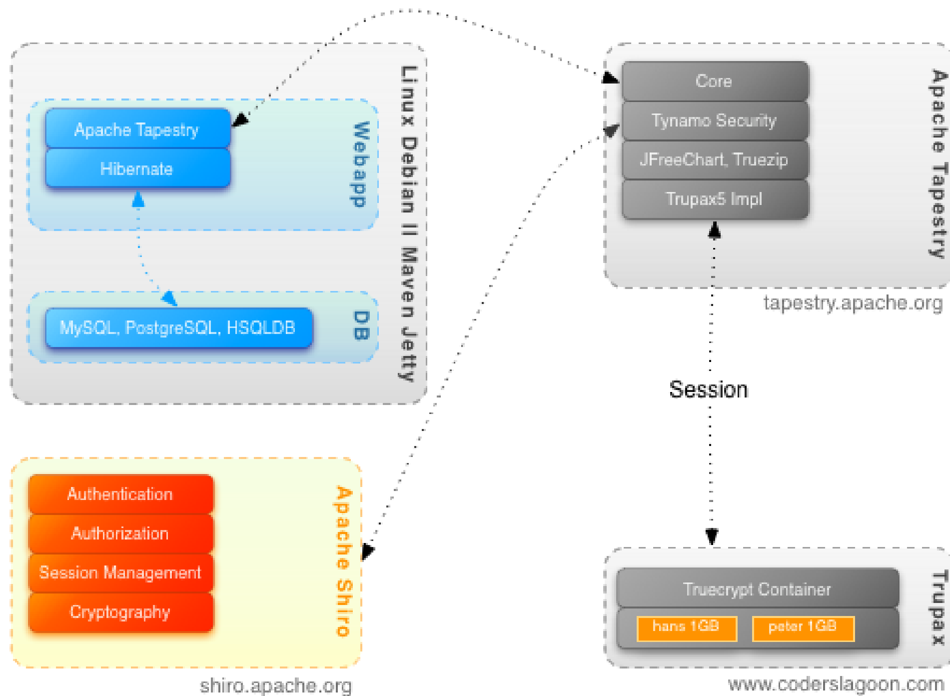

Abbildung 4.1: Tapestry Anwendung - *SecureFileStorage* (WS 11/12)

4.2 Maven als zentrales Build-Tool

Da der Code der erwähnten Tapestry Anwendung in einem Versionskontrollsystem (Git) liegt, wechseln wir in ein von uns dafür angelegtes Verzeichnis und holen uns den Code mit folgendem Befehl[1].

git clone ssh://root@jenkins/var/local/git/fileStorage

Anschließend bietet uns Maven verschiedene Möglichkeiten den Code für ein Eclipse Projekt anzupassen oder direkt über Jetty zu starten. Auch eine Dokumentierung des Quellcodes oder das erstellen einer lauffähigen WAR-Datei ist mit dafür bestimmten Maven Parameter möglich (mvn [options] [<goal(s)>] [<phase(s)>]). Hierfür können *goals* und *phasen* nach belieben gemischt werden (siehe Kapitel 2.1).

1 git - the simple guide: http://rogerdudler.github.io/git-guide/ - zuletzt besucht: 31.05.2013

Die Tapestry Anwendung über Jetty starten

Dazu muss in das Hauptverzeichnis der Anwendung gewechselt werden (wo sich die *pom.xml*-Datei befindet) und folgender Befehl in der Shell eingegeben werden (mvn [goal]):

mvn jetty:run

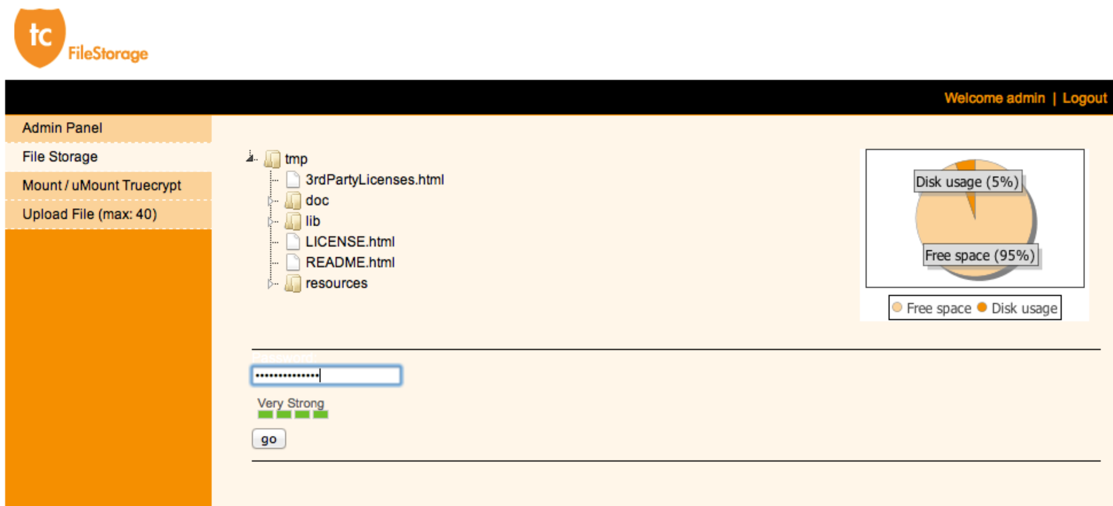

Abbildung 4.2: Tapestry Anwendung - http://127.0.0.1:8080/fileStorage

Unter *http://127.0.0.1:8080/fileStorage* kann nun die Anwendung begutachtet werden (user/user, admin/admin [user/pass]). Das Testen der Webanwendung ermöglicht uns u.a. die dateibasierte DB HSQLDB sowie das Maven-Jetty-Plugin, die in der Konfigurationsdatei dafür eingetragen wurden (*pom.xml*).

```
<dependencies>
    ...
    <dependency>
        <groupId>hsqldb</groupId>
        <artifactId>hsqldb</artifactId>
        <version>1.8.0.7</version>
    </dependency>
    ...
</dependencies>
```

Abbildung 4.3: *pom.xml* Ausschnitt - HSQLDB

Die Tapestry Anwendung als WAR erstellen

Eine lauffähige WAR-Datei wird über folgenden Befehl erstellt (mvn [option] [phase]):

mvn clean install

```
<build>
    <finalName>fileStorage</finalName>
    <plugins>
        ....
        <plugin>
            <groupId>org.mortbay.jetty</groupId>
            <artifactId>maven-jetty-plugin</artifactId>
            <version>6.1.16</version>
            <configuration>
                <!-- Log to the console. -->
                <requestLog implementation="org.mortbay.jetty.NCSARequestLog">
                    <!-- This doesn't do anything for Jetty, but is a workaround for a
                        Maven bug that prevents the requestLog from being set. -->
                    <append>true</append>
                </requestLog>
            </configuration>
        </plugin>
        ...
    </plugins>
</build>
```

Abbildung 4.4: *pom.xml* Ausschnitt - Maven Jetty Plugin

Die WAR-Datei ist nach erfolgreicher Generierung unter dem *target*-Verzeichnis im Projekt-Ordner zu finden. Die Option clean bewirkt ein löschen der bereits erzeugten Artefakte und des *target*-Verzeichnisses (die Kompilierung übernimmt dabei das Maven-Compiler-Plugin). Wollen wir zusätzlich zum erstellten WAR-Datei eine Javadoc-Dokumentation mit UML-Diagrammen, ist dies durch den Zusatz eines *goals* ohne Probleme möglich:

mvn clean install **javadoc:aggregate**

Unter *target/site* finden wir nun die generierten Apidocs mit UML-Diagrammen.
Dies wurde durch folgende Einstellungen im Maven-Javadoc-Plugins erreicht (siehe Abbildung 4.5).
Die benötigten Jar- (lib/styleed.jar, lib/ydoc.jar) -und Konfigurationsdateien (resources/*) wurden dafür direkt im doc-Verzeichnis des Projekts hinterlegt[1].

1 http://www.yworks.com/de/products_ydoc.html - zuletzt besucht: 31.05.2013

```
...
</dependencies>
<build>
    <finalName>fileStorage</finalName>
    <plugins>
        ...
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-javadoc-plugin</artifactId>
            <version>2.9</version>
            <configuration>
                <encoding>UTF-8</encoding>
                <doclet>ydoc.doclets.YStandard</doclet>
                <docletPath>${basedir}/doc/lib/ydoc.jar:$basedir}/doc/lib/styled.jar:${basedir}/doc/resources</doclet
                <additionalparam>-umlautogen</additionalparam>
                <doctitle>${project.name} (${project.version})</doctitle>
                <show>private</show>
                <additionalJOption>-J-Xmx512m</additionalJOption>
                <stylesheetfile>${basedir}/doc/css/api/stylesheet.css</stylesheetfile>
                <javadocDirectory>${basedir}/doc/css/api</javadocDirectory>
                <docfilessubdirs>true</docfilessubdirs>
            </configuration>
        </plugin>
        ...
    </plugins>
</build>
```

Abbildung 4.5: *pom.xml* Ausschnitt - Maven Javadoc Plugin

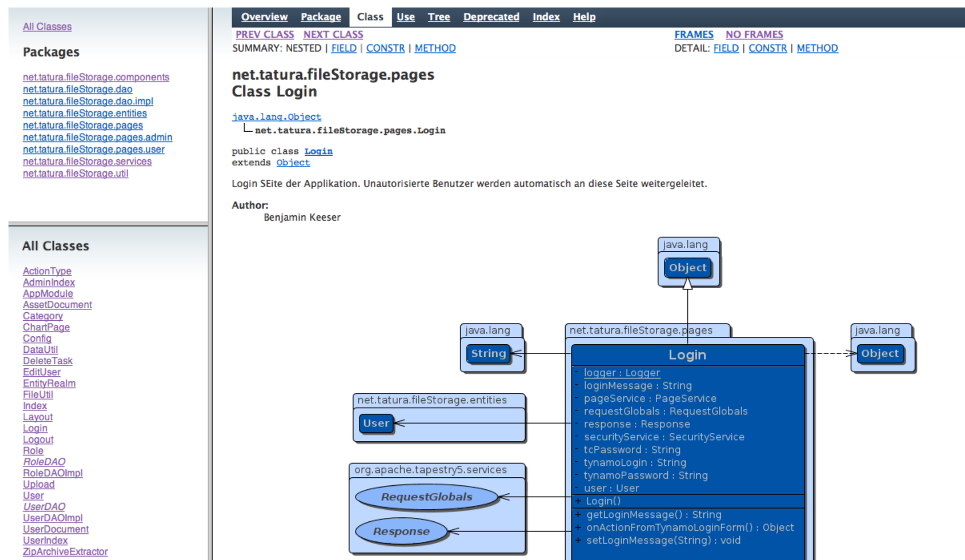

Abbildung 4.6: Yworks UML Doclet - Javadocs mit UML-Diagrammen

Die Tapestry Anwendung über Code-Analyse Tools prüfen

Um wöchentlich Code-Analysen zu generieren wurden unterschiedliche Tools zum Messen der Metriken eingebunden. Dazu zählen in diesem Projekt FindBugs[1] und PMD.

```
<reporting>
    <plugins>
        <plugin>
            <groupId>org.codehaus.mojo</groupId>
            <artifactId>findbugs-maven-plugin</artifactId>
            <version>2.5.2</version>
            <configuration>
                <xmlOutput>true</xmlOutput>
                <threshold>Exp</threshold>
                <effort>Max</effort>
                <includeFilterFile>${project.basedir}/configs/findbugs.xml</includeFilterFile>
            </configuration>
        </plugin>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-pmd-plugin</artifactId>
            <version>3.0.1</version>
            <configuration>
                <minimumTokens>100</minimumTokens>
                <verbose>true</verbose>
                <rulesets>
                    <ruleset>${project.basedir}/configs/pmd.xml</ruleset>
                </rulesets>
            </configuration>
        </plugin>
    </plugins>
</reporting>
```

Abbildung 4.7: *pom.xml* Ausschnitt - Maven Cody-Analyse Plugins

Die Erstellung der Dokumentation(en) wird durch folgenden Befehl angestoßen:

mvn site javadoc:aggregate

Nach erfolgreichem Durchlauf können die Berichte zur Code-Qualität betrachtet werden.

1 http://findbugs.sourceforge.net - zuletzt besucht: 31.05.2013

PMD Results

The following document contains the results of PMD 5.0.2.

Files

net/tatura/fileStorage/components/Layout.java

Violation	Line
Avoid unused private fields such as 'title'.	20

net/tatura/fileStorage/pages/Login.java

Violation	Line
The class 'Login' has a Cyclomatic Complexity of 4 (Highest = 9).	35 - 126
Avoid unused private fields such as 'user'.	39
Avoid unused private fields such as 'tcPassword'.	42
The method 'onActionFromTynamoLoginForm' has a Cyclomatic Complexity of 9.	70 - 113

net/tatura/fileStorage/pages/Logout.java

Violation	Line
The class 'Logout' has a Cyclomatic Complexity of 4 (Highest = 5).	23 - 66
The method 'onActivate' has a Cyclomatic Complexity of 5.	33 - 55

Abbildung 4.8: PMD Bericht

4.3 Die Integration in Jenkins

Zunächst müssen wir unter „Jenkins verwalten > Plugins verwalten > Verfügbar“, die benötigten Plugins installieren. Für unser Projekt brauchen wir das:

- Credentials Plugin
- HTML Publisher plugin
- Javadoc Plugin
- Jenkins GIT client plugin, Jenkins GIT plugin
- Jenkins Mailer Plugin
- Maven Integration plugin
- Static Analysis Collector Plug-in, Static Analysis Utilities, Warnings Plug-in

Schritt 1

Nach erfolgreicher Installation der Plugins muss ein neuer Job eingerichtet werden (vgl. [Sma11, S.78-133]). Da unserer Projekt über Maven aufgebaut wurde, fällt die Wahl auf „Maven 2/3 Projekt bauen“. Der Job-Titel ist frei wählbar. In unserem Falle wollen wir alte Builds verwerfen und bis zu 50 neue, maximal 3 Tage lang aufheben.

Schritt 2

Da sich unsere Jenkins-Anwendung auf einer Virtual Maschine mit Debian befindet können wir für das Git-Repository den (lokalen) Pfad angeben, unter dem wir unseren Code abgelegt haben: z.B. /var/local/git/fileStorage. Unter den erweiterten Einstellungen geben wir als Branch noch den Master-Zweig an: */master*.

Abbildung 4.9: Jenkins - Maven Job-Einrichtung: Schritt 1

Abbildung 4.10: Jenkins - Maven Job-Einrichtung: Schritt 2

Schritt 3

Dem Build-Auslöser teilen wir mit, dass er jede Minute Änderungen im Versionsverwaltungssystem Git prüfen soll. In einer realen Umgebung würden hier alle 15 Minuten völlig ausreichen (**/15 * * * **).

Im Stamm POM-Feld ändern wir den Eintrag auf „fileStorage/pom.xml“ und unter „Goals und Optionen“ geben wir noch „install“ an.

Build-Auslöser

- [ ] Baue dieses Projekt, wenn eine SNAPSHOT-Abhängigkeit gebaut wurde
- [ ] Starte Build, nachdem andere Projekte gebaut wurden.
- [ ] Auslöser, um entfernte Builds zu starten (z.B. skriptgesteuert)
- [ ] Builds zeitgesteuert starten
- [x] Source Code Management System abfragen

Zeitplan: * * * * *

Do you really mean "every minute" when you say "* * * * *"? Perhaps you meant "H * * * *"

Ignore post-commit hooks

Pre Steps

Add pre-build step

Build

Stamm-POM: fileStorage/pom.xml

Goals und Optionen: install

Post Steps

Run only if build succeeds · Run only if build succeeds or is unstable · Run regardless of build result

Should the post-build steps run only for successful builds, etc.

Abbildung 4.11: Jenkins - Maven Job-Einrichtung: Schritt 3

Schritt 4

Daran anschließend fügen wir über das Drop-Down Menü „Add post-build step“ eine Shell hinzu (Shell ausführen). Im Shell-Body tragen wir nun noch ein paar Zeilen Bash-Code ein, welche die WAR-Datei in unserem Apache Tomcat auswechselt. Zudem wollen wir per Mail über Fehlgeschlagene Builds benachrichtigt werden.

Shell ausführen

Befehl

```
if [ -f /home/jenkins/apache-tomcat-7.0.40/webapps/fileStorage.war ]; then
    rm /home/jenkins/apache-tomcat-7.0.40/webapps/fileStorage.war
fi
sleep 5

mv ${WORKSPACE}/fileStorage/target/fileStorage.war /home/jenkins/apache-tomcat-7.0.40/webapps
```

Liste der verfügbaren Umgebungsvariablen

Add post-build step

Build-Einstellungen

☑ E-Mail-Benachrichtigung

Empfänger: info@tatura.net

Liste der Empfängeradressen, jeweils durch Leerzeichen getrennt. E-Mails werden versandt, wenn ein Build fehlschlägt.

☑ E-Mails bei jedem instabilen Build senden

☑ Getrennte E-Mails an diejenigen Anwender senden, welche den Build fehlschlagen ließen

Abbildung 4.12: Jenkins - Maven Job-Einrichtung: Schritt 4

Schritt 5

Nun fügen wir in einem letzten Schritt die Suche nach Compiler-Warnungen hinzu („Add post-build action“). Dort geben wir als Parser Maven an und binden über den „Add post-build action“ das „Nachgelagerte Test-Ergebnisse zusammenfassen“ ein, welches z.B. Junit-Tests zusammenfasst.

Da wir nur wöchentlich Analyse-Tools laufen lassen wollen, um die Code-Qualität zu prüfen und eine Javadoc-Dokumentation mit UML-Diagrammen zu erstellen, springen wir in das Hauptmenü von Jenkins zurück.

Post-Build-Aktionen

Suche nach Compiler Warnungen

Konsole durchsuchen

Parser Maven

Passt keiner der vorhandenen Parser, so kann in der System Konfiguration ein eigener Parser entwickelt werden, d
abgestimmt ist.

Hinzufügen

Diese Parser werden zum Analysieren der Konsolenausgabe verwendet. Falls kein Parser ausgewählt wird, wird die Konsolenaus

Dateien durchsuchen Hinzufügen

Dateien aus dem Arbeitsbereich, die mit einem festdefiniertem Parser durchsucht werden sollen.

Nachgelagerte Testergebnisse zusammenfassen

- [x] Alle nachgelagerten Tests zusammenfassen
- [ ] Fehlgeschlagene Builds in Ergebnis erfassen

Abbildung 4.13: Jenkins - Maven Job-Einrichtung: Schritt 5

Report Job - Schritt 1

Dort angekommen legen wir einen neuen Job an und wählen diesmal „Kopiere bestehenden Job“ aus. Damit können wir die Konfiguration aus dem zuletzt erstellten Job übernehmen und nach unseren Wünschen anpassen. Statt dem bisherigen Punkt „Source Code Management System abfragen“ wählen wir nun „Builds zeitgesteuert starten“ aus. Im Zeitplan-Feld geben wir nun noch @weekly für einen wöchentlichen Aufruf des Jobs an. Unter den Maven Goals und Optionen ändern wir die Parameter auf folgende Werte um:

site javadoc:aggregate

Build-Auslöser

- [ ] Baue dieses Projekt, wenn eine SNAPSHOT-Abhängigkeit gebaut wurde
- [ ] Starte Build, nachdem andere Projekte gebaut wurden.
- [ ] Auslöser, um entfernte Builds zu starten (z.B. skriptgesteuert)
- [x] Builds zeitgesteuert starten

Zeitplan: @weekly

- [ ] Source Code Management System abfragen

Pre Steps

Add pre-build step

Build

Stamm-POM: fileStorage/pom.xml

Goals und Optionen: site javadoc:aggregate

Abbildung 4.14: Jenkins - Maven Report Job-Einrichtung: Schritt 1

Report Job - Schritt 2

Nun muss noch die Shell mit Bash-Code, der die Reports und Javadoc-Api an die richtige Stelle schiebt, erweitert werden.

Shell ausführen

Befehl

```
if [ -f /var/www/reports/* ]; then
    rm -r /var/www/reports/*
fi
if [ -f /var/www/documentation/* ]; then
    rm -r /var/www/documentation/*
fi
sleep 1

if [ -f ${WORKSPACE}/fileStorage/target/site/apidocs/index.html ]; then
    cp -r ${WORKSPACE}/fileStorage/target/site/apidocs/* /var/www/documentation
fi

if [ -f ${WORKSPACE}/fileStorage/target/site/index.html ]; then
    cp -r ${WORKSPACE}/fileStorage/target/site/* /var/www/reports
fi
```

Liste der verfügbaren Umgebungsvariablen

Abbildung 4.15: Jenkins - Maven Report Job-Einrichtung: Schritt 2

Report Job - Schritt 3

Unter den „Post-build Aktionen“ fügen wir als Parser noch das JavaDoc Tool hinzu und unser wöchentlicher Report-Job ist fertig.

Post-Build-Aktionen

Suche nach Compiler Warnungen

Konsole durchsuchen

Parser Maven

Passt keiner der vorhandenen Parser, so kann in der System Konfiguration ein eigener Parser entwickelt werd abgestimmt ist.

Parser JavaDoc Tool

Passt keiner der vorhandenen Parser, so kann in der System Konfiguration ein eigener Parser entwickelt werd abgestimmt ist.

Abbildung 4.16: Jenkins - Maven Report Job-Einrichtung: Schritt 3

5 Zusammenfassung und Ausblick

In der vorliegender Semesterarbeit wurde der Versuch unternommen Maven und Jenkins nach einer kurzen Einführung anhand einer Beispiel-Anwendung vorzustellen. Aufgrund der Komplexität des Themenfeldes konnten dabei nur kleine Einblicke in die Welt von Jenkins und Maven ermöglicht werden. Maven ist in dieser Arbeit von zentraler Bedeutung, da es bereits fest an der vorgestellten Beispiel-Anwendung gekoppelt war und darüber hinaus auch sinnvoll mit Jenkins zusammen eingesetzt werden kann.
Im letzten Abschnitt wurde kurz auf die statischen Analyse-Tools eines Jenkins Plugins eingegangen (Static Code Analysis Plugins). Wie könnte nun eine Erweiterung in Richtung „Continuous Delivery aussehen“? Denkbar wäre z.B. die durch dieses Tool generierten Metriken für eine Zustimmung oder Ablehnung eines Live-Deployments zu nutzen. Zudem müssten hierfür noch simulierte Browser-Tests über Selenium, mit Kriterien für eine abschließende Auslieferung einer Software-Anwendung bestimmt werden. Für eine Automatisierung von finalen Software-Releases sollten neben dem Themenfeld „kontinuierliche Installationen“ sicher noch weitere Aspekte, wie die Automatisierung von Lieferprozessen heraus gearbeitet werden.
Die Richtung einer Testautomatisierung als ersten Schritt, wurde in dieser Arbeit bereits aufgezeigt.

Literaturverzeichnis

[Beh12] Behrendt, Mario: *Jenkins - kurz und gut*, O'Reilly Verlag (2012)

[Fow06] Fowler, Martin: Continuous Integration (2006), URL `http://martinfowler.com/articles/continuousIntegration.html`

[Sma11] Smart, J.F.: *Jenkins: The Definitive Guide*, O'Reilly Media (2011), URL `http://books.google.de/books?id=pQ-DG7aT0IgC`

[Wie11] Wiest, S.G.: *Continuous Integration mit Hudson: Grundlagen und Praxiswissen für Einsteiger und Umsteiger*, dpunkt-Verlag (2011)

[Zer13] ZeroTurnaround: Jenkins CI: The Origins of Butlers, Build Masters and Bowties. *Rebellabs Journal* (2013), (1)

Abbildungsverzeichnis

1.1 Continuous Integration Prozess . . . 3

2.1 Grundgerüst eines minimalen POM . . . 5
2.2 mvn in einer Shell - mvn –help > Anzeige der verfügbaren Optionen . . . 5
2.3 Apache Maven: Phasen und Goals . . . 6
2.4 Abhängigkeit zu einer Software-Komponente . . . 7
2.5 Beispiel für POM Abhängigkeiten . . . 8
2.6 Beispiel für eine POM Vererbung . . . 9
2.7 Beispiel für eine POM Aggregation . . . 9
2.8 Beispiel für ein POM Archetype . . . 10
2.9 *magnoliaVersion* als festgelegtes Beispiel-Property . . . 10
2.10 Code Analyse Plugin PMD . . . 11

3.1 Jenkins/Hudson commits . . . 13
3.2 Sicherheitsmatrix unter Jenkins . . . 14
3.3 Plugin-Verwaltung unter Jenkins . . . 16

4.1 Tapestry Anwendung - *SecureFileStorage* (WS 11/12) . . . 18
4.2 Tapestry Anwendung - http://127.0.0.1:8080/fileStorage . . . 19
4.3 *pom.xml* Ausschnitt - HSQLDB . . . 19
4.4 *pom.xml* Ausschnitt - Maven Jetty Plugin . . . 20
4.5 *pom.xml* Ausschnitt - Maven Javadoc Plugin . . . 21
4.6 Yworks UML Doclet - Javadocs mit UML-Diagrammen . . . 21
4.7 *pom.xml* Ausschnitt - Maven Cody-Analyse Plugins . . . 22
4.8 PMD Bericht . . . 23
4.9 Jenkins - Maven Job-Einrichtung: Schritt 1 . . . 24
4.10 Jenkins - Maven Job-Einrichtung: Schritt 2 . . . 24
4.11 Jenkins - Maven Job-Einrichtung: Schritt 3 . . . 25
4.12 Jenkins - Maven Job-Einrichtung: Schritt 4 . . . 26
4.13 Jenkins - Maven Job-Einrichtung: Schritt 5 . . . 27
4.14 Jenkins - Maven Report Job-Einrichtung: Schritt 1 . . . 28
4.15 Jenkins - Maven Report Job-Einrichtung: Schritt 2 . . . 28
4.16 Jenkins - Maven Report Job-Einrichtung: Schritt 3 . . . 29

Tabellenverzeichnis

4.1 Tabelle mit Build-Phasen 17