Fachhochschule
Braunschweig/Wolfenbüttel
– University of Applied Sciences –

Entwicklung einer Testumgebung für das WebDAV-Modul Catacomb

Diplomarbeit

20118723 - Rudolf Steger

5. August 2009

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, die vorliegende Arbeit selbstständig und ohne fremde Hilfe angefertigt zu haben. Die verwendete Literatur und sonstige Hilfsmittel sind vollständig angegeben.

Inhaltsverzeichnis

Abbildungsverzeichnis

Nomenclature

ACL Access Control List

DASL DAV Searching and Locating

DBMS Datenbankmanagementsystem

Delta-V Web Versionierungs und Konfigurations Management Protokol spezifiziert in der RFC 3253 für WebDAV

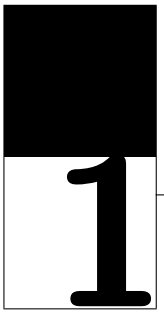
HTTP Hypertext Transfer Protocol

SVN Subversion ist eine Open-Source-Software zur Versionsverwaltung von Dateien und Verzeichnissen.

URI Uniform Resource Identifier

URL Uniform Resource Locator

WebDAV Web-based Distributed Auhoring and Versioning



Kapitel 1

Einleitung

1.1 Umgebung

1.1.1 DLR e. V. (SISTEC)

Der praktische Teil dieser Arbeit wurde in der Abteilung Verteilte Systeme und Komponentensoftware der Einrichtung Simulations- und Softwaretechnik (SISTEC) des Deutschen Zentrums für Luft- und Raumfahrt e. V. (DLR) erstellt. Die Aufgaben der DLR-Einrichtung SISTEC sind Forschung und Entwicklung auf dem Gebiet innovativer Software-Engineering-Technologien und die Bereitstellung und Anwendung dieses Software-Know-hows. Die derzeitigen Themenschwerpunkte sind die komponentenbasierte Softwareentwicklung für verteilte Systeme, Software für eingebettete Systeme und Software-Qualitätssicherung.

Die Schwerpunkte der Abteilung Verteilte Systeme und Komponentensoftware liegen in den Bereichen Grid-Computing und der komponentenbasierten Software-Entwicklung für verteilte Systeme. Zu den Aufgaben gehören die Entwicklung moderner ingenieurwissenschaftlicher Software-Anwendungen und Software-Integrations-Werkzeuge unter Nutzung Service-Orientierter-Architekturen sowie die Beratung und Entwicklung im Bereich Automatisierung und praktischer Nutzung von Software-Engineering-Prozessen.

Abb. 1.1: Logo DLR

1.1.1.0.1 Aktuelle Schwerpunkte in Forschung und Entwicklung

Grid-Computing

- Entwicklung Grid-fähiger Anwendungen
- Grid-Security (Sicherer Zugriff auf Grid-Ressourcen durch Firewalls)
- Nachvollziehbarkeit von Prozessen in verteilten Systemen (Provenance)

Integrationstechnologie

- Entwicklung der Integrationssysteme TENT und RCE
- Wrappen beliebiger Applikation in Software-Integrations-Systeme
- Einheitliche Datenschnittstellen für DLR-Codes und Vernetzung unterschiedlicher Fachdisziplinen
- Objektorientierte Schnittstellen für numerische Software

Wissenschaftliches Daten- und Informationsmanagement

- Datenmanagement-Software DataFinder
- Best-Practices-Tools / Expertensysteme für numerische Strömungsmechanik
- Herkunft und Verlässlichkeit von Daten (Data Provenance)

Graphische Benutzeroberflächen

- Entwicklung von Benutzeroberflächen für ingenieurwissenschaftliche Anwendungen
- Grafische Test-Tools für numerische Applikationen (Strömungslöser etc.)

Software Engineering

- Automatisierung von Software-Engineering-Verfahren
- Code-Generierung in Applikationen für die Test-Automatisierung
- Customizing von Software-Engineering-Prozessen und -Tools

1.1.2 Motivation

Das DLR entwickelt im eigenen Hause das Produkt DataFinder. Der Datafinder ist eine Software zur Verwaltung technisch-wissenschaftlicher Daten. Ein wesentlicher Bestandteil des Backends dieser Software ist das Opensource Apache WebDAV Modul Catacomb.

Das DLR ist maßgeblich an der Entwicklung von Catacomb beteiligt. Diese Diplomarbeit soll helfen die Qualität dieses Produkts zu sichern.

Das DLR ist maßgeblich an der Entwicklung von Catacomb beteiligt. Diese Diplomarbeit soll helfen die Qualität dieses Produkts zu sichern. Software Tests sind ein grundlegendes Mittel zur Qualitätssicherung in der Softwareentwicklung. Um also eine gleichbleibende Qualität, oder gar eine Qualitätssteigerung zu erreichen, ist es sinnvoll Tests mit einer gewissen Regelmäßigkeit durchzuführen. In dem naturgemäßen etwas chaotischen Entwicklungumfeld einer Opensource Software kommen gut geplante und genau definierte Softwaretests meistens etwas zu kurz. An diesem Punkt soll eine automatisierte Testumgebung Abhilfe schaffen.

Was ist überhaupt ein Software Test? In der Literatur gibt es viele unterschiedliche Definitionen und Ansichten darüber, was ein Software Test denn nun genau ist.

Das *IEEE standard glossary of software engineering terminology* definiert einen Test ganz allgemein:

"test. (1) An activity in which a system or component is executed under specified conditions, the results are observed or recorded, and an evaluation is made of some aspect of the system or component"[20, S. 74]

Wozu die genannte Evaluation nun dient, bleibt offen. Viele Entwickler sehen Tests als eine Art Beweis, dass Ihre Software genau das tut, was sie soll. Dieser Ansatz ist etwas bedenklich, kann er doch zu einer gewissen Blindheit führen.

Myers beschreibt den Sinn von Software Test in *The art of software testing* wie folgt:

"Testing is the process of executing a program with the intent of finding errors."[25, S. 6]

Der eigentliche Ansatz sollte also die gezielte Suche nach Fehlern sein.

In diesem Sinne soll beim DLR für das Apache Modul Catacomb eine Testumgebung geschaffen werden, die es den Entwicklern ermöglicht die Qualität des Moduls bestmöglich zu gewährleisten.

1.1.3 Kapitelübersicht

In Kapitel 2 werde ich Anforderungen die an die Testumgebung gestellt werden genauer erläutern.

In Kapital 3 werde ich einige Grundlagen, Tools und Frameworks vorstellen, die zum Umsetzen der Umgebung Evaluiert wurden.

In Kapital 4 werde ich dann den endgültigen Aufbau der Testumgebung erläutern.

In Kapitel 5 werde ich ein Resümee ziehen und weitergehende Möglichkeiten für die Testumgeben erörtern.

Kapitel 2

2 Anforderungen

Einer der ersten Schritte zur Realisierung der Applikation war die Anforderungsanalyse. In ihr wurden alle funktionalen und nicht-funktionalen Anforderungen zusammengetragen. Einige wurden von der Abteilung festgelegt, andere ergaben sich aus dem technischen Umfeld. Die Anforderungen wurden in einem Wiki zusammengetragen und in ein Lastenheft übertragen. Balzert beschreibt diese Art der Vorgehensweise als "traditionelle Beschreibungsform einer Produkt-Definition"[22, S. 109ff]. Die nachfolgende Gliederung ist in der DIN ISO 9126 definiert. Dieser Standard legt wichtige Qualitätsmerkmale einer Software fest. Im Rahmen dieser Arbeit wird nur auf die wesentlichen Anforderungen eingegangen[22, S. 1102].

2.1 Funktionalität

Wie in Kapitel 3 beschrieben benötigt Catacomb eine gewisse Infrastruktur von Programmen. Da es sich bei Catacomb um ein Apache Modul handelt, ist zum Testen ein lauffähiger Apache zwingende Vorraussetzung. Zusätzlich wird zum Persistieren der abgelegten Daten ein DBMS vorrausgesetzt. Im Moment werden MySQL und PostgreSQL Datenbanken unterstützt.

Die Testumgebung soll nun in der Lage sein Catacomb gegen verschiedene Kombinationen dieser Programme zu testen.

Um das Testen für den Entwickler so komfortabel wie möglich zu machen, soll die Testumgebung komplett automatisiert funktionieren. Der Sourcecode von Catacomb wird in einem SVN repository verwaltet. Ein guter Ansatzpunkt um einen Test zu starten ist also, wenn ein Entwickler den Sourcecode angefasst hat, bzw. eine veränderte Datei eingecheckt hat. Es muss also eine Möglichkeit gefunden werden das Repository zu überwachen und im Fall einer Veränderung die Testumgebung anzustarten.

Im Verlauf der Diplomarbeit wurden verschiedene Möglichkeiten untersucht, um Tests zu entwerfen und auszuführen.

2.2 Zuverlässigkeit

Gerade bei einer Testumgebung ist die Zuverlässigkeit extrem wichtig. Es muss sichergestellt sein, dass Tests so weit wie möglich deterministisch und eben zuverlässig ablaufen. Tests müssen miteinander vergleichbar sein, um nicht nur einen objektiven Ist-Zustand über die Software zu erhalten, sondern auch Rückschlüsse über Tendenzen erlangen zu können.

2.3 Benutzbarkeit und Änderbarkeit

Die Testumgebung soll so einfach wie möglich zu bedienen sein. Dieser Teil der Anforderung wird hauptsächlich durch die Automatisierung abgedeckt. Im Idealfall ist seitens des Nutzer keine Interaktion notwendig.

Die Umgebung soll zusätzlich auch einfach wartbar sein, so dass Tests neuer Funktionen des Catacombs einfach und schnell implementiert werden können.

2.4 Effizienz

Um die Testumgebung möglichst effizient umzusetzen, soll so weit möglich, auf bereits vorhandene Software zurückgegriffen werden. So ist die Umsetzung schnell realisierbar und letztendlich auch besser zu warten.

2.5 Interoperabilität

Die Interoperabilität ist bei so einer Testumgebung nahezu vernachlässigbar. Trotzdem ergibt sich durch die Vorgabe der ausschließlichen Verwendung von Open Source Produkten von selbst die Möglichkeit, die Testumgebung auf jedem aktuellen Betriebssystem aufzusetzen.

Weiterhin soll als Programmiersprache Python[7] verwendet werden. Der Python Interpreter ist ebenfalls für alle gängigen Betriebssysteme verfügbar und gewährleistet so eine hervorragende Portierbarkeit des Sourcecodes.

Kapitel 3

3 Evaluierung geigneter Werkzeuge zum Entwurf der Testumgebung

Beim Entwurf der Testumgebung durften einige Aspekte nicht aus dem Auge verloren werden. Das Hauptziel der Implementierung der Umgebung ist es funktionierende Installationsvarianten der Kombination Apache, Mysql und Catacomb zu finden. Anhand dieser gefundenen funktionierenden Kombinationen werden dann die Funktionen des Catacomb Moduls im Einzelnen untersucht. Der Entwicklerstab von Catacomb ist nicht besonders groß, deshalb war es wichtig, so weit wie möglich auf verschiedene fertige Lösungen zurückzugreifen, diese zu Evaluieren und die am einfachsten wartbare und effizienteste Lösung zu wählen.

Die Mindestanforderung an die Umgebung ist ein automatisierter Test des Catacomb Moduls auf seine Konformität mit den Spezifizierungen, die in der RFC2518[12] festgehalten sind. Das heißt. Es muss irgendwie realisiert werden, das möglichst komfortabel ein automatisierter Test des Moduls nach dem Einchecken in das SVN-Repository angestoßen werden kann, um es auf Regressionen und andere Fehler zu überprüfen.

Im Folgenden werden einige Grundlagen erläutert und einige Frameworks, Tools und Applikationen besprochen, die zum Realisieren der Testumgebung in Betracht

gezogen und evaluiert wurden.

3.1 Grundlagen

3.1.1 Datafinder

Der DataFinder ist eine leichtgewichtige Anwendungssoftware zur Verwaltung technisch-wissenschaftlicher Daten. Er wurde entwickelt, um große Datenmengen zu verwalten und ermöglicht die Speicherung von Daten über eine Reihe verschiedener Speicher-Schnittstellen (z.B. WebDAV, FTP, GridFTP, OpenAFS oder TSM). Die Struktur der Daten und beschreibende Metadaten werden in XML-Format auf einem zentralen Server gespeichert und können über das standardisierte Protokoll WebDAV bearbeitet werden.

Abb. 3.1: Logo Datafinder

Der DataFinder besitzt folgende Hauptfunktionalitäten:

- Up- und Download von Daten.
- Versehen der Daten mit standardisierter und benutzerdefinierter Meta-Information.
- Eine Suchfunktion, in der mehrere Suchterme verknüpft werden können.
- Abarbeitung von Skripten zur Automatisierung von Arbeitsabläufen (z.B. automatisches Up- und Download oder Durchführung von Rechnungen).

Die Verwaltung der Datenstrukturen an einer zentralen Stelle und die Beschreibung der Daten mit standardisierter Meta-Information erleichtern das Auffinden von Daten wesentlich und vermeiden damit Doppelarbeit. Auch kann ein Anwender Teilergebnisse oder Eingabedaten anderer Berechnungen, die schon auf Servern

vorhanden sind, für seine aktuelle Arbeit nutzen, ohne die gleichen Daten noch einmal erzeugen zu müssen. Das flexible Metadaten-Konzept erlaubt dem Anwender ferner, die auf den Servern vorhandenen Daten mit zusätzlicher Meta-Information zu versehen.

Eine Datenmanagement-Lösung mit dem DataFinder setzt serverseitig auf offene und flexible Standards wie das WebDAV-Protokoll und bleibt dadurch leicht erweiterbar und flexibel. Aktuell unterstützte WebDAV-Server sind zum einen der Tamino XML Server der Software AG als kommerzielle Lösung und der aus einem Open-Source-Projekt hervorgegangene WebDAV-Server Catacomb. Diese Server erlauben serverseitige Suche sowie die automatische Versionisierung von Dokumenten und Daten.

Der DataFinder ist als Open-Source-Software unter der BSD-Lizenz verfügbar. Die Client-Applikation des Datafinders ist komplett in Python geschrieben. Die Graphische Oberfläche wurde durch QT-bindings zu Python realisiert.

3.1.2 Python

Python[7] ist eine universell einsetzbare Hochsprache. Bei Ihrem Entwurf wurde besonderer Wert auf die Lesbarkeit des Codes gelegt.

Abb. 3.2: Logo Python

“Python is an interpreted, object-oriented, high-level programming language with dynamic semantics. Its high-level built in data structures, combined with dynamic

typing and dynamic binding, make it very attractive for Rapid Application Development, as well as for use as a scripting or glue language to connect existing components together. Python's simple, easy to learn syntax emphasizes readability and therefore reduces the cost of program maintenance. Python supports modules and packages, which encourages program modularity and code reuse. The Python interpreter and the extensive standard library are available in source or binary form without charge for all major platforms, and can be freely distributed.

Often, programmers fall in love with Python because of the increased productivity it provides. Since there is no compilation step, the edit-test-debug cycle is incredibly fast. Debugging Python programs is easy: a bug or bad input will never cause a segmentation fault. Instead, when the interpreter discovers an error, it raises an exception. When the program doesn't catch the exception, the interpreter prints a stack trace. A source level debugger allows inspection of local and global variables, evaluation of arbitrary expressions, setting breakpoints, stepping through the code a line at a time, and so on. The debugger is written in Python itself, testifying to Python's introspective power. On the other hand, often the quickest way to debug a program is to add a few print statements to the source: the fast edit-test-debug cycle makes this simple approach very effective." [19]

3.1.3 Apache httpd

Der Apache HTTP Server ist ein Projekt der Apache Software Foundation und der meistbenutzte Webserver im Internet [18]. Der Server unterstützt eine Vielzahl von Betriebssystemen. Unter anderem Linux, Unix und Mac OS X. Ziel des Projektes ist es einen sicheren, effizienten und erweiterbaren Server zu entwickeln, der den aktuellen HTTP Standards entspricht.

Abb. 3.3: Logo Apache

Der Apache Webserver ist als Open-Source-Software unter der Apache-Lizenz [1] verfügbar.

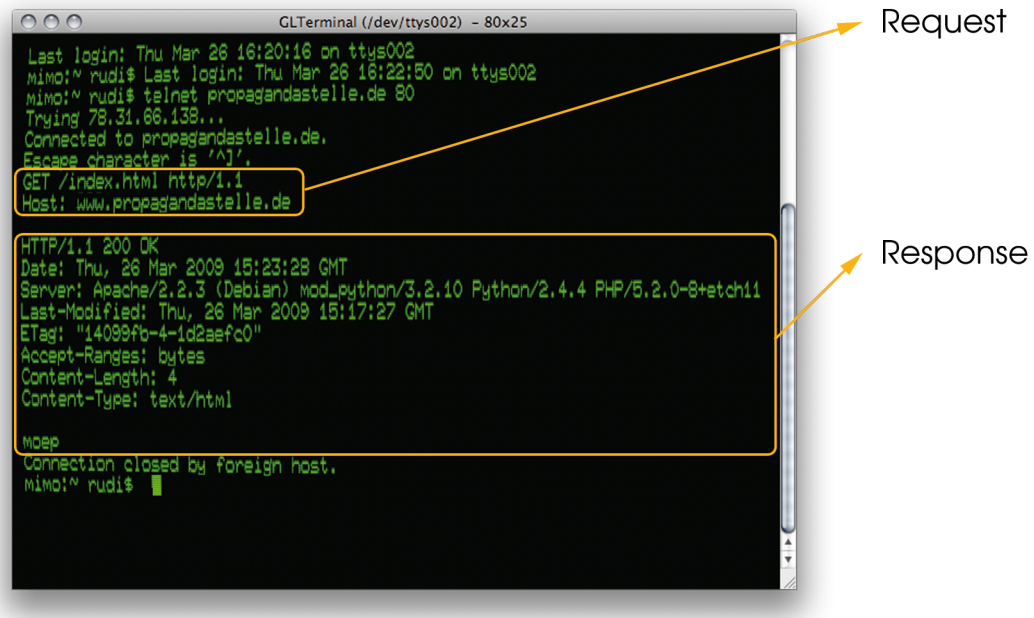
3.1.3.1 HTTP

Das Hypertext Transfer Protocol (HTTP, dt. Hypertext-Übertragungsprotokoll) ist ein Protokoll zur Übertragung von Daten über Netzwerke. Der hauptsächliche Einsatzzweck ist das WWW. Dort wird es benutzt um Webseiten vom Server zum Client (Webbrowser) zu übertragen und darzustellen.

HTTP ist in der Anwendungsschicht des ISO/OSI-Schichtenmodells angesiedelt.

Durch Erweiterung seiner Anfragemethoden, Header-Informationen und Statuscodes ist das HTTP nicht auf Hypertext beschränkt, sondern wird zunehmend zum Austausch beliebiger Daten verwendet. Zur Kommunikation ist HTTP auf ein zuverlässiges Transportprotokoll angewiesen. In nahezu allen Fällen wird hierfür TCP verwendet. Eine solche Erweiterung ist WebDAV, siehe Kapitel3.1.5.

Abb. 3.4: HTTP Request



3.1.3.2 Module

Der Apache Webserver ist einfach durch sogenannte Module erweiterbar. Zu diesem Zweck existiert die Apache API. Module werden in der Konfigurationsdatei des Servers aktiviert und kümmern sich von nun an, um spezielle Anfragen. Beispielsweise WebDAV Anfragen.

3.1.4 DBMS

Ein Datenbanksystem (DBS) ist ein System zur elektronischen Datenverwaltung. Die wesentliche Aufgabe eines DBS ist es, große Datenmengen effizient, widerspruchsfrei und dauerhaft zu speichern und benötigte Teilmengen in unterschiedlichen, bedarfsgerechten Darstellungsformen für Benutzer und Anwendungsprogramme bereitzustellen.

3.1.4.1 MySQL

MySQL ist ein relationales Datenbank Management System. MySQL läuft als Server und erlaubt den multi-user Zugriff auf mehrere Datenbaken die es verwaltet.

Abb. 3.5: Logo Mysql

Der Sourcecode des Projektes ist unter anderem unter den Bedingungen der GNU General Public License[21] erhältich.

Dieses DBMS ist dank seines Reifegrades und dank der Opensource Lizenz sehr beliebt und verbreitet.

3.1.4.2 PostgreSQL

PostgresSQL ist ein objekt-relationales Datenbank Management System und unter einer BSD[2] ähnlichen Lizenz erhältlich. PostgresSQL ist nicht ganz so beliebt wie MySQL hat aber doch einen treuen Nutzerstamm.

Abb. 3.6: Logo PostgreSQL

3.1.5 WebDAV

WebDAV (Web-based Distributed Authoring and Versioning) ist ein offener Standard um Dateien auf Internet Servern zu verwalten und editieren. Es handelt sich technisch um eine Erweiterung des HTTP/1.1 Standards. Anders als bei HTTP/1.1 stehen wesentlich mehr Methoden zum verwalten von Dateien bereit.

Abb. 3.7: Logo WebDAV

Bei HTTP/1.1 wurden Parameter Informationen ausschließlich in den HTTP Headern kodiert. Im Gegensatz dazu beschreibt WebDAV Methoden Parameter Informationen entweder in einem XML (Extensible Markup Language) Request Entity Body, oder ebenfalls im HTTP Header. Die Motivation XML zum Beschreiben der Parameter-Informationen zu wählen liegt in der Möglichkeit leicht zusätzliche XML Elemente zu bestehenden Strukturen hinzufügen zu können. So ist eine hervorragende Erweiterbarkeit gegeben. Zusätzlich war die Möglichkeit entscheidend, Informationen im ISO 10646 [16] Character Set kodieren zu können. Dieses Character Set unterstützt die Internationalisierung. Als grober Anhaltspunkt gilt, dass Parameter in XML content bodies kodiert werden, wenn sie eine unbestimmte Länge haben. Auch wenn sie später von Menschen betrachtet werden sollen, bietet sich XML an, da sie dort im ISO 10646 Character Set kodiert werden können. Andernfalls werden die Parameter im HTTP Header kodiert.

3.1.5.1 WebDAV Terminologie

URI/URL UniForm Resource Identifier und Uniform Resource Locator. Diese Begriffe und unterschiede zwischen Ihnen werden in der RFC 2396[11] beschrieben.

Identity Grady Booch, Object-Oriented Design with Applications: “Identity is the property of an object that distinguishes it from all other objects.” [23, S.141]

Resource Eine Resource kann alles sein das eine Identität (Identity) hat. Zum Beispiel eine Textdatei, eine Bilddatei oder eine Collection aus anderen Resourcen.

Collection Eine Ressource die ein Set aus URIs (member URIs), die sogenannten member Ressourcen identifiziert.

Member-URI - Ein URI der member eines sets von URIs in eine collection ist.

Property Ein name/value Paar das beschreibende Informationen über eine Ressource enthält.

Live-Property Eine Property dessen Semantik und Syntax vom Server erzeugt wird. Der Wert des "getcontentlength" Live Properties z.b. wird automatisch vom Server erzeugt, wenn es im Zuge eines GET Requests abgefragt wird und liefert die Länge der Resource in Bytes.

Dead-Property Eine Property dessen Semantik und Syntax nicht vom Server erzeugt wird. Der Server speichert nur den Wert einer Dead Property. Der Client ist für die Wahrung der Konsistenz der Syntax und Semantik einer Dead Property verantwortlich.

3.1.5.2 WebDAV Methoden

Die Spezifikation des WebDAV-Protokolls ist in sogenannten RFCs (Request for Comments) festgehalten. Die grundlegende Spezifikation ist in der RFC 2518[12]festgehalten. Das RFC2518 spezifiziert ein Set von Methoden, Headern und Content-Typen zur Erweiterung des HTTP/1.1 Standards. Diese Erweiterung dient der Verwaltung von Resource-Properties, der Verwaltung und Erzeugung von Resource-Collections, der Manipulation des Name spaces und der Möglichkeit des Resource lockings zur Vermeidung von Kollisionen.

3.1.5.2.1 Umfang der RFC2518

Spezifikationen für Properties Die Möglichkeit Informationen zu Webseiten (Resourcen) zu erstellen, zu löschen und abzufragen. Beispielsweise Autoren, Erstellungsdaten, usw. zusätzlich die Möglichkeit Seiten jeglichen Medientyps zu anderen verwandten Seiten zu verlinken.

Spezifikationen für Collections Die Möglichkeit Sets von Dokumenten (Resourcen) zu erstellen und ein hierarchisches membership listing, wie ein directory listing bei normalen Dateisystemen, zu erhalten.

Spezifikationen für Locking Die Möglichkeit zu kontrollieren, dass nicht mehr als eine Person an einem Dokument (Resource) arbeitet. Das verhindert, dass Aktualisierungen verloren gehen, wenn bei geöffneter Resource, zuerst ein Autor Veränderungen in das Dokument schreibt und dann ein zweiter, ohne dass die Veränderungen zusammengeführt werden.

Spezifikationen für Namespace Operationen Die Möglichkeit den Server zu instruieren Web Resourcen zu kopieren und zu verschieben. Voraussetzungen und Grundvorraussetzungen werden in dem begleitenden Dokument "Requirements for a Distributed Authoring and Versioning Protocol for the World Wide Web" beschrieben[10].

Die Erweiterungen zur Versionierung (Delta-V) werden in der RFC 3253 [13]spezifiziert.

Die Benutzerverwaltung (ACL) wird in der RFC 3744 [14]spezifiziert.

Die Methoden zur Suche (DASL) auf einem WebDAV Server werden in der RFC 5323[15]spezifiziert.

3.1.5.3 Catacomb

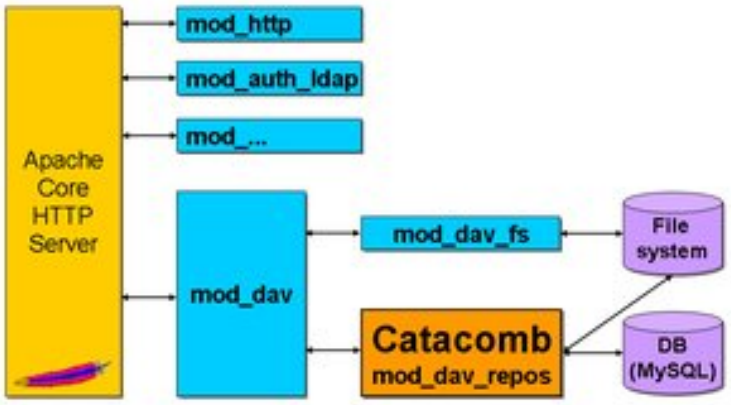
Der Catacomb WebDAV Server ist ein WebDAV-Modul für den Apache Webserver, der das Standardmodul mod_dav um einige Zusatzfunktionen des WebDAV-Protokolls erweitert. mod_dav bringt normalerweise eine eigene Modulerweiterung mit, mod_dav_fs, bei der sowohl Inhalt als auch die Metadaten der Ressource im lokalen Dateisystem gespeichert werden.

Abb. 3.8: Logo Catacomb

Catacomb ersetzt in diesem Zusammenhang mod_dav_fs mit einem Modul, das mod_dav_repos heißt, und speichert Ressourcen und Metadaten in einer relationalen Datenbank. Durch die Datenbankabstraktion mit mod_dbd des Apache Projekts werden eine Vielzahl von Datenbanken unterstützt. Der Hauptvorteil dabei ist das dadurch stark verbesserte Suchverhalten des Servers, das bei der Umsetzung des Protokolls für die serverseitige Suche (DASL) benötigt wird. Auch die Erweiterung zur Versionisierung von Ressourcen (DeltaV) wird mit Hilfe des Datenbankkonzepts erreicht.

Durch die Aufnahme des Prinzip der relationalen Datenbanken ist Catacomb in der Lage wichtige Aspekte eines typischen Dokumentenmanagement-Systems zu übernehmen: Die Fähigkeit, eine große Anzahl von Dokumenten zu speichern und über deren Metadaten zu suchen.

Abb. 3.9: Schema Catacomb



Das Catacomb WebDAV-Modul ist, wie alle Produkte der Apache Software Foundation, als Open Source unter der Apache-Lizenz 2.0 verfügbar und damit kostenlos. Die aktuelle Version ist 0.9.6.

3.1.6 Arten von Tests

3.1.6.1 Black-Box Testing

Black-Box-Test bezeichnet eine Methode des Softwaretests, bei der die Tests ohne Kenntnisse über die innere Funktionsweise des zu testenden Systems entwickelt werden. Er beschränkt sich auf funktionsorientiertes Testen, d. h. für die Ermittlung der Testfälle werden nur die Anforderungen, aber nicht die Implementierung des Testobjekts herangezogen. Die genaue Beschaffenheit des Programms wird nicht betrachtet, sondern vielmehr als Black Box behandelt. Nur nach außen sichtbares Verhalten fließt in den Test ein.

Ziel ist es, die Übereinstimmung eines Softwaresystems mit seiner Spezifikation zu überprüfen. Ausgehend von formalen oder informalen Spezifikationen werden Testfälle erarbeitet, die sicherstellen, dass der geforderte Funktionsumfang eingehalten wird. Das zu testende System wird dabei als Ganzes betrachtet, nur sein Außenverhalten wird bei der Bewertung der Testergebnisse herangezogen.

3.1.6.2 Integration Testing

"Integration testing" (manchmal auch Integration and Testing genannt, I&T) wird der Bereich des Software Testens genannt, in dem einzelne Teile/Module einer Softwarelösung kombiniert und als zusammenhängende Gruppe getestet werden. Üblicherweise werden Integration Tests nach dem Unit Test und vor dem Systemtest durchgeführt.

Der Sinn eines Integrationstestes ist es nun die Funktionalen, Geschwindigkeits- und Zuverlässigkeits-Anforderungen zu überprüfen, die an die Teile des Systems gestellt werden. Diese Teile des Systems werden anhand Ihrer Interfaces mithilfe von Black-Box Tests untersucht. Wobei erfolgreiche und nicht erfolgreiche Test cases anhand der übergebenen Parameter und Dateneingaben simuliert werden. Die Verwendung von geteilten Speicherbereichen und Interprozesskommunikation wird ebenfalls getestet und individuelle Subsysteme werden anhand Ihrer Eingabe Interfaces untersucht.

Es werden Testfälle definiert, die sicherstellen sollen, dass alle Komponenten eines Subsystems korrekt miteinander interagieren. Z.B. procedure calls oder process activations. Dies geschieht nachdem alle Einzelteile/Module bereits durch Unit-Tests gelaufen sind.

Der Grundgedanke der sich hinter dieser Technik verbirgt ist die building block Herangehensweise. Dabei werden überprüfte Komponenten einer bereits getestenen Basis hinzugefügt um weitere Baugruppen dem Integrations Test zu unterziehen.

Andere Arten der Herangehensweisen beim Integrations Testen sind: big bang, top-down und bottom-up.

3.1.6.3 System Testing

System Tests von Software oder Hardware sind Tests die auf einem kompleten integrierten System durchgeführt werden, um die Übereinstimmung mit den spezifizierten Vorgaben zu evaluieren. System Tests fallen in den Bereich des Black Box Testings und sollten als solche kein Wissen über das innere Design des Codes und der Logik benötigen.

In aller Regel beinhaltet ein Systemtest alle integrierten Softwarekomponenten, die vorher erfolgreich den Integrationstest bestanden haben und ein geeignetes Hardwaresystem oder mehrere geeignete Systeme auf denen das Softwaresystem selbst integriert werden soll. Wie schon erwähnt beschäftigt sich das Integrationstesten mehr mit Ungereimtheiten zwischen den einzelnen Komponenten die in einem System zusammengefügt werden, oder um Fehler zwischen einzelnen Softwareeinheiten und dem Zielhardwaresystem. Systemtest sind eine limitierendere Art von Tests. Hier wird nach Fehlern zwischen den Komponenten aber auch nach Fehlern im System als ganzes gesucht.

System Tests werden in Abstimmung mit den Functional Requirement Specifications (FRS) und/oder der System Requirement Specification (SRS)

über das ganze System durchgeführt. System Tests sind eine investigative Phase der Softwareentwicklung, bei der der Fokus fast schon in den destruktiven Bereich gehen kann. Es wird nicht nur das Design überprüft, sondern auch das verhalten des Systems, bis hin zu den angenommen Erwartungen des Kunden eines solchen Systems. Es ist auch vorgesehen über die Grenzen die in den Software/Hardware Requirements Specifications definiert sind, hinaus zu testen.

Die folgenden Beispiele sind unterschiedliche Arten des Testens, die bei einem System Test in Erwägung ziehen sollte:

- GUI software testing - Ein Prozess bei dem die Graphische Oberfläche eines Produktes (sofern vorhanden) anhand unterschiedlicher Usecases gegen die festgehaltenen Spezifikation getestet wird.
- Usability testing - Eine Technik die anhand von User Tests das Produkt evaluiert. Sehr wichtig bei Endnutzer Systemen, um einen überblick zu bekommen, wie gut ein Benutzer das System benutzen kann und so die wirkliche usability eines Produktes feststellen zu können. Im Gegensatz zu usability Prüfungen die von Experten durchgeführt werden und keine echten Nutzer erfordern.
- Performance testing - deckt eine Fülle an technischen und funktionalen Evaluierungen ab. Es wird nicht so sehr auf einzelne Spezifikationen eingegangen, viel mehr steht die messbare Leistungsfähigkeit im Fordergrund. Beispielsweise die Geschwindigkeit.

- Compatibility testing - Ist Teil der non-funktionalen Tests. Die Tests werden an der applikation durchgeführt um etwaige Kompatibilitätsprobleme in der vorgegebenen Einsatzumgebung aufzudecken. Die einsatzumgabung kann z.B. durch bestimmte Peripheriegeräte, Betriebssysteme, Datenbanken, Dateisysteme, Browserkompatibilität, Bandbreite die zur Verfügung steht, u.a.
- Error handling testing - hier wird gezielt getestet wie die Anwendung mit Fehlern umgeht.
- Load testing - Hier wird gezielt auf das Lastverhalten der Applikation eingegangen und die Reaktion auf eben jene gemessen, um Aussagen über das lastverhalten treffen zu können, oder eventuelle Spezifikationnen in dem bereich zu erfüllen.
- Volume testing - bezeichnet das Testen einer Applikation mit einer bestimmten Menge an Daten. Diese Menge kann beispielsweise die grösse einer Datenbank bezeichnen oder die grösse einer Interfacedatei die beim Volume Testing untersucht wird.
- Stress testing - eine Form des Testens eines Systems oder Teil eines Systems, bei dem dessen Stabilität überprüft wird. Dabei wird gezielt die normale Betriebskapazität dieses (Teil-) Systems überschritten. Oft bis zu einem Punkt an dem das System überlastet ist, um zu sehen wie sich das System an diesem Punkt verhält.
- Security testing - soll sicherstellen, das ein IS (Information System) Daten korrekt schützt und die Funktionsfähigkeit, wie vorgesehen bereitstellt.
- Scalability testing - hier wird getestet wie gut eine Softwareapplikation skaliert.
- Capacity testing - Capacity Testing beschäftigt sich mit der Kapazität an Usern die ein laufendes System bedienen kann.
- Sanity testing - ein schneller Test um die Richtigkeit einer Anname zu überprüfen.
- Smoke testing - ein smoke test ist üblicherweise eine ansammlung von fertigen Tests, die durchgeführt werden, bevor das System überhaupt für weitere Tests zugelassen wird. Diese Tests sind meist automatisiert und teilweise auch als Build Verification Test bekannt.

- Exploratory testing - ist eine manuelle Test Maethodik mit einem gewissen Fokus auf der Erfahrung und Kreativität des Testers. Ist jedoch sonst mit dem Ad hoc testing vergleichbar.
- Ad hoc testing - beschreibt das Testen einer Software ohne jegliche Planung und Dokumentation. Es wird einfach ausprobiert.
- Regression testing - das Regression Testing versucht regressions (Rückschritte) aufzudecken. Wenn also bereits funktionierende Teile, oder schon verbesserte Teile eines System plötzlich nicht mehr so arbeiten wie vorgesehen.
- Reliability testing - versucht Fehlerraten zu bestimmen. Also einen Prozentsatz zu berechnen, mit dem man sich auf ein System verlassen kann.
- Recovery testing - es wird getestet wie gut sich eine Application von Abstürzen, Hardwarefehlern und ähnlichen Problemen erholt.
- Installation testing - beschäftigt sich mit der Installation einer Applikation. Hier wird getestet in welcher Reihenfolge und was ein Kunde auf seinem System installieren muss um die Applikation korrekt zum laufen zu bringen. Sehr wichtig werden solche Tests bei der Bestimmung von Upgrade- und Updatepfaden, um beispielsweise eine akkurate Datenmigration zu gewährleisten.
- Maintenance testing - eine Art von Testen die durchgeführt wird um Probleme an teilen des Hardwareequipments zu identifiezieren, zu diagnostizieren oder um zu bestätigen das Reperaturarbeiten erfolgreich waren.

3.1.6.4 Unit Testing

In der Softwareentwicklung bilden Unit Tests eine Methode zur Verifikation und Validation einzelnener Einheiten von Sourcecode. So wird gewährleistet das diese Einheiten an Sourcecode für den Einsatz bereit sind. So eine Einheit ist das kleinste testbare Teil einer Applikation. Bei der Prozedualen Programmierung kann diese kleinste Einheit ein einzelnes Programm, eine Funktion, ein Prozedur usw. sein. Die kleinste Einheit beim objektorientierten Programmieren ist die Methode, die zu einer Basisklasse, Superklasse, einer abstrakten Klasse oder einer abgeleiteten Klasse gehören kann.

Unit Testing kann auf so simpler Basis wie dem schrittweisen durchgehen eines Stück Codes in einem debugger sein. Heutzutage wird häufig ein test framework für solche Zwecke genutzt. Beispielsweise jUnit in Java-Applikationen.

Idealerweise sollte jeder Testfall unabhängig vom anderen sein. Einzelne Module müssen demnach isoliert werden in dem bei Abhängigkeiten geieignete Methoden stubs, mock Objekte, fakes und Test Harnische benutzt werden. Unit Test werden üblicherweise direkt vom Entwickler der Software geschrieben um sicherzustelen das der Code den ansprüchen genügt und sich wie erwartet verhält. Die Umsetzung kann unterschiedlich geschehen. Von manuell ausgeführt bis formal festgehalten und als Teil eines automatischen Build Prozesses.

3.2 Vorhandene Infrastruktur

3.2.1 Subversion

Subversion ist ein Version Control System. Es wird benutzt um aktuelle und historische Version von Dateien, wie beispielsweise Sourcecode, Web Pages und Dokumentationen zu verwalten. Die Versionierung erfolgt in einem zentralen Projektarchiv (engl. repository) in Form einer einfachen Revisionszählung. Wenn Änderungen an Inhalten verteilt auf den Computern der Bearbeiter ausgeführt werden, werden zwischen dem Projektarchiv und einem Arbeitsplatz jeweils nur die Unterschiede zu bereits vorhandenen Ständen übertragen; anfangs das gesamte Projekt, später nur Änderungen.

3.2.2 VMware Server

VVMware stellt Virtualisierungsprodukte her, die es ermöglichen auf einer Gastmaschine mehrere virtuelle Wirtmaschinen laufen zu lassen.

Für den Betrieb von Servern in virtuellen Maschinen gibt es das kostenlose Einstiegsprodukt VMware Server und die Programmsuite VMware Infrastructure. Letztere beinhaltet den ESX Server und das VirtualCenter sowie weitere optionale Komponenten (HA, DRS).

VMware Server ist der Nachfolger von VMware GSX. VMware GSX und VMware Server sind Hosted-Produkte, das heißt, sie benötigen ein Wirtsbetriebssystem (Windows oder Linux).

Der VMware Server unterstützt Multiprozessorsysteme und Intel64 / AMD64 und ist kostenlos (allerdings fordert der Hersteller eine Registrierung der Kunden).

VMware ESX Server basiert auf einem eigenem Kernel, der um Linux Treiber erweitert wurde und daher kein Wirtsbetriebssystem benötigt. Diese Version ist für den Einsatz in Rechenzentren gedacht. Ein Verkaufsargument ist die Platz- und Kosteneinsparung, die durch Konsolidierung der Server erreicht werden kann. VMware Infrastructure in der Version 3 bietet als Option eine Hochverfügbarkeitslösung (HA) und eine automatische Lastverteilung (DRS). Seit Ende Juli 2008 ist der VMware ESX Server in der Version 3i auch in einer kostenlosen Version verfügbar. Generell arbeitet VMware ESX Server nur auf spezieller, von VMware offiziell unterstützter, Server-Hardware. Allerdings gibt es auch Non-Server-Hardware, auf der das System läuft.

Der vorhandene VMware Server wird interessant, wenn zur Implementierung der Testumgebung so eine virtuelle Maschine gewählt werden würde. Das hätte den Vorteil einer genau definierten Testumgebung, die nach Bedarf gestartet und erweitert werden kann. Die komplette Abkapselung der Umgebung hätte auch den Vorteil, das die virtuelle Maschine notfalls auch von einem Entwickler heruntergeladen werden kann um lokal in einer fertigen Infrastruktur Tests fahren zu können.

3.2.3 Hudson

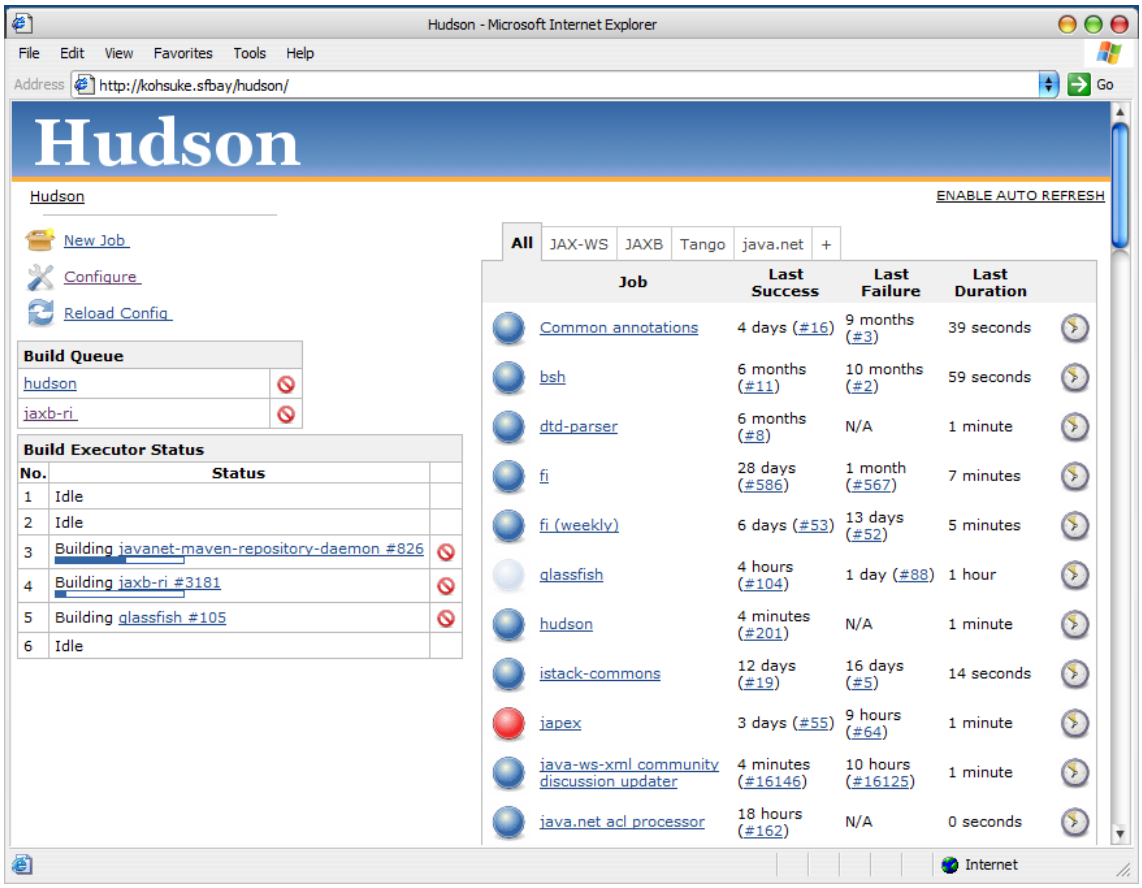
Hudson ist eine Applikation das die Ausführung von wiederkehrenden Aufgaben überwacht. Zum Beispiel das kompilieren von Software Projekten oder Prozesse die von cron ausgeführt werden. Hudson konzentriert sein Augenmerk auf folgende 2 Punkte:

1. Softwareprojekte kontinuierlich kompilieren oder auch testen. Ähnlich anderen Applikationen wie zum Beispiel CruiseControl[3] oder dem im Moment nicht unter aktiver Entwicklung stehendem DamageControl[5]. Hudson stellt ein einfach zu bedienendes sogenanntes Continuous Integration System zur Verfügung.

Es vereinfacht dem Entwickler Änderungen an einem Projekt vorzunehmen und vereinfacht es dem User einen aktuellen Build eines Projektes zu bekommen. Der kontinuierlich automatische Build-Prozess steigert die Produktivität, in dem er den Entwickler entlastet.

2. Das Überwachen von Ausführungen extern laufendender Jobs, wie zum Beispiel cron Jobs und procmail Jobs. Diese Jobs können auch auf externen Maschinen laufen. Bei cron Jobs Beispielsweise, von denen man nur eine E-Mail erhält in der die Ausgabe des Jobs vorhanden ist, ist es Sache des Empfängers die Mails sorgfältig zu beobachten und auf eventuelle Fehler zu achten. Hudson speichert alle diese Ausgaben und macht es so einfacher festzustellen, wenn etwas falsch läuft.

Abb. 3.10: Screenshot Hudson



3.2.3.1 Features

Hudson bietet die folgenden Features:

- Einfache Installation: Kann einfach mit "java -jar hudson.war" ausgeführt werden, oder in einem vorhanden Servlet Container installiert werden. Sonst ist nichts weiter notwendig. Auch keine Datenbank.
- Einfache Konfiguration: Hudson kann komplett über die Weboberfläche konfiguriert werden. Die Weboberfläche bietet on-the-fly error checks und inline Hilfen. Es ist nicht notwendig die Konfigurationsdateien (XML) per Hand anzufassen. Was allerdings möglich ist, wenn man möchte.
- Change set Unterstützung: Hudson kann eine Liste der Veränderungen generieren, die an dem Build aus dem CVS oder Subversion Repository vorgenommen wurden. Das passiert relativ effizient und schont die Belastung des Repositories.
- Permanente Links: Hudson erzeugt für einzelne Build- oder Testjobs gut lesbare und permante Links, so das man sie leicht von anderer Stelle verlinken kann. Fest eingebaute Links sind beispielsweise Links zu dem letzen erfolgreichen Build und dem letzen Build.
- RSS, E-Mail und IM Integration: Es ist einfach möglich Build Prozesse und Ergebnisse per RSS, E-Mail oder IM zu verfolgen. Bzw. Benachrichtigungen über erfolgreiche oder fehlgeschlagene Prozesse zu erhalten.
- After-the-fact tagging: Builds können lang nachdem Sie abgschlossen sind getagged werden. Interessant für releases.
- JUnit und TestNG test Benachrichtigungen: JUnit Test Ergebnisse können in Form von Tabellen und Zusammengefasst dargestellt werden. Mit zusätzlichen historischen Informationen. Zum Beispiel Informationen darüber, wann etwas angefangen hat nicht mehr Korrekt zu funktionieren. Der Trend wird in einen Graph geplottet.
- Verteilte Builds: Hudson kann build und Test Aufgaben an mehrere Maschinen verteilen. So ist es Möglich die Belastung der einzelnen Rechner in einer Entwicklungs und Testumgebung bestmöglich auszunutzen.

- Datei fingerprinting: Hudson kann sich darum kümmern mitzuloggen welcher Build welche jars erzeugt hat und welcher Build welche Versionen von jars benutzt und so weiter. Das funktioniert sogar für jars die ausserhalb von Hudson erzeugt wurden. Somit ist diese Funktionalität ideal, um Projektabhängigkeiten zu verfolgen.
- Plugin Support: Zusätzlich ist es Möglich Hudson durch Plugins zu erweitern und weitere Funktionalitäten hinzuzufügen. Ein sehr interessantes Plugin ist beispielsweise das Twitter plugin.

3.3 Test-Generierung

3.3.1 Apache::Test framework

Das Apache::Test framework wurde speziell entwickelt, um Testsuites für Produkte zu entwickeln, die auf dem Apache httpd Webserver laufen. Ursprünglich für das mod_perl modul entwickelt ist es in perl geschrieben und ist nun an einem Punkt seiner Entwicklungsgeschichte angelangt, an dem man es für jegliche Apache Module verwenden kann. Im Zuge des Entwurfs der Testumgebung wurde ein genauerer Blick auf dieses framework geworfen, da es sich förmlich anbietet, um Tests für eine Umgebung mit einem Apache Webserver, zu entwickeln.

3.3.1.1 Basics

Aus der Entwicklungsgeschichte ergibt sich natürlich die Konsequenz, das die Tests selber auch in Perl geschrieben werden. Das Framework bietet eine recht großen Umfang an Funktionalität zum testen von Modulen. Das erleichtert das Schreiben von Tests. Die Testkonzepte lehnen sich an die üblichen Herangehensweisen zum Testen von Perlmodulen an. Das Script “t/TEST” führt sämtliche Dateien mit der Endung “.t” aus, das es im Verzeichnis “t” findet. Wenn das Script ausgeführt wird, erzeugt es einen Standardisierten Output der folgenderweise aussieht:

```
1..3 # going to run 3 tests
ok 1 # the first test has passed
```

```
ok 2 # the second test has passed
not ok 3 # the third test has failed
```

Wie man sieht, folgt jedem “ok” oder “not ok” die Nummer des Entsprechenden Sub-Tests, der erfolgreich oder nicht erfolgreich abgeschlossen wurde. “t/TEST” nutzt das “Test::Harness” Modul, welches den STDOUT stream abfängt, parst und nach Abschluss aller Tests die Resultate ausgibt, also wie viele Tests und Sub-Tests durchgelaufen sind und wie viele davon erfolgreich bzw. nicht erfolgreich waren oder übersprungen wurden.

Einige Test können übersprungen werden:

```
1..0 # all tests in this file are going to be skipped.
```

Zum Überspringen von Tests kann es kommen, wenn ein Feature des Moduls optional ist und/oder eine Vorraussetzung für einen Test nicht erfüllt wurde, aber nicht kritisch für die Gesamtaussage eines Tests ist. Das heißt, wenn einmal herausgefunden wurde, dass man mit einem Test nicht fortfahren kann, es aber nicht notwendig ist, dass er erfolgreich abschließt, kann er einfach übersprungen werden.

Standardmäßig werden print statements von Test::Harness herausgefiltert. Wenn man sehen möchte, was ein Test genau macht, um ihn beispielsweise zu debuggen, muss die -verbose option gewählt werden. Wenn also ein Test folgendes beinhaltet:

```
print "# testing : feature foo\n";
print "# expected: $expected\n";
print "# received: $received\n";
ok $expected eq $received;
```

sieht man im normalen Modus keine dieser prints. Wenn man allerdings t/TEST mit der -verbose option ausführt, wird folgendes ausgegeben.

```
# testing : feature foo
# expected: 2
# received: 2
ok 2
```

Beim Entwickeln eines Tests ist es also immer möglich eine Reihe von debug statements einzufügen und diese, sobald der Test funktioniert, nicht auszukommentieren oder zu löschen. Durch diesen Aufbau des frameworks ist es eine gute Herangehensweise die debug statements im Test zu belassen, denn wenn beispielsweise ein andere Nutzer des Tests einen Fehler in einem Test hat, kann man ihn bitten, diesen im verbose Modus durchzuführen und kann nachher die Ausgabe analysieren. Mit solchen debug Ausgaben wird es verständlicherweise einfacher Fehler von Nutzern zu verstehen. Eine einfachere Methode ist es das Test::More Modul zu verwenden. Dieses Modul stellt eine Vielzahl von nützlichen Test Funktionen zur Verfügung. Unter anderem diag. Dies ist eine Funktion die automatisch strings escaped und sie vorbei an Test::Harness an print weitergibt:

```
use Test::More;
diag "testing : feature foo\n";
diag "expected: $expected\n";
diag "received: $received\n";
ok $expected eq $received;
```

Bei einem solchen Beispiel kann man auch einfach die “is”-Funktion aus Test::More verwenden. Sie gibt die notwendigen Informationen im Falle eines Testfehlers aus.

```
is $received, $expected;
```

Die Ausgabe des Tests würde dann in etwa so aussehen:

```
not ok 1 # Failed test (-e at line 1) # got: '1' # expected: '2'
```

For more details about the Test::Harness module please refer to its manpage. Also see the Test and Test::More manpages for documentation of Perl’s test suite.

3.3.1.2 Ausführen von Tests

Test lassen sich ganz einfach wie auch bei CPAN Perl Modulen üblich starten. Zuerst wird ein Makefile generiert und kompilieren die Tests mit:

```
% perl Makefile.PL [options]
% make
```

Jetzt können die Tests gefahren werden. Es gibt 2 Arten einen Test zu starten. Der erste ist das übliche:

```
% make test
```

Allerdings erzeugt dies eine Menge overhead, da alle Sourcen auf ihren aktuellen Stand überprüft werden (make source change control). Deshalb sollte man dies nur nach dem ersten make nutzen. Um Tests nochmals auszuführen empfiehlt es sich den Test direkt auszuführen:

```
% t/TEST
```

Dies ist um einiges schneller. Beim ausführen von “make test” oder “t/TEST” werden alle Tests die im Verzeichnis “t” gefunden werden ausgeführt (alle Dateien die auf .t enden).

3.3.1.3 Individuelles Testen

Um nur einen einzigen Test auszuführen, muss man das nur in der Kommandozeile angeben. Um Beispielsweise den vorhandenen Test t/protocol/echo.t auszuführen, müsste man nur folgendes eingeben:

```
% t/TEST protocol/echo
```

Das prefix “t” und die Endung “.t” sind optional zum angeben der Dateinamen, wenn man sie explizit anstartet. Folgende eingaben sind also alle möglich:

```
% t/TEST    protocol/echo.t
% t/TEST t/protocol/echo
% t/TEST t/protocol/echo.t
```

Der Server wird gestoppt, falls er gerade läuft und ein neuer wird gestartet bevor der Test “t/protocol/echo.t” ausgeführt wird. Nachdem der Test abgeschlossen wurde, wird er wieder heruntergefahren. Wie bereits erwähnt, kann ein Test abhängig davon, wie er geschrieben wurde auch mit der -verbose Option ausgeführt werden um weitere debugging Informationen zu erhalten. Das geht natürlich auch mit einzelnen Tests. Die Kommandozeile müsset dann so aussehen:

```
% t/TEST -verbose protocol/echo
```

Es ist auch Möglich einzelne Gruppen von Tests zu starten. Dieses Kommando:

```
% ./t/TEST modules
```

würde alle Tests im Verzeichnis “t/modules” starten.

Kombinationen sind auch Möglich:

```
% ./t/TEST modules protocol/echo
```

Dies startet alle Test in modules und den protocol/echo Test. Wenn Test explizit angegeben werden, werden si in der angegebenen Reihenfolge ausgeführt. Wenn die Test nicht explizit angegeben werden, werden sie in alphabetischer Reihenfolge abgearbeitet.

3.3.1.4 Wiederholtes Testen

Wenn man Tests ohne die -run-tests option ausführt wird der server standardmässig vor den Tests gestartet und danach wieder gestoppt. Wenn man während des debuggings einen Test nochmals ausführen möchte ohne auf einen Server-Neustart warten zu wollen, kann man den Server einmal starten:

```
% t/TEST -start-httpd
```

und dann Tests mit der -run-tests option mehrfach ausführen ohne auf einen Neustart warten zu müssen:

```
% t/TEST -run-tests
```

Nachdem alle Tests durchgeführt wurden, kann der Server wieder gestoppt werden:

```
% t/TEST -stop-httpd
```

Während der Server läuft können .t Dateien modifiziert werden und Tests ohne einen Neustart nochmals ausgeführt werden. Wenn man allerdings Antwort-Handler in dem zu testenden Modul ändert, muss der Server neugestartet werden, damit die Änderungen aktiv werden. Bei Perl Modulen wäre es theoretisch Möglich mit Hilfe

von Apache::Test den code ohne einen Neustart neuzuladen. Das ist aber in Hinblick auf das zu testende Modul zu vernachlässigen.

Wichtig zu wissen ist, das bei dem Aufruf mit der -start-httpd option immer zuerst der Server gestoppt wird, sofern einer läuft.

3.3.1.5 Paralleles Testen

Es könnte Notwendig werden mehrere Apache-Test framework Instanzen nebeneinander laufen zu lassen. In diesem Fall muss jeder Instanz ein anderer Port zugewiesen werden. Man kann beim starten der Tests mit der -port option explizit einen port angeben, auf dem der server laufen soll. Um den server beispielsweise auf port 8888 laufen zu lassen. Gibt man folgendes ein:

```
% t/TEST -start-httpd -port=8888
```

Es ist auch Möglich den Port durch die Umgebungsvariable "APACHE_TEST_PORT" festzulegen und auf den gewünschten Wert zu setzen, bevor der server gestertet wird. Den Port per hand festzulegen ist aber unter Umständen nicht die komfortabelste Methode, wenn es notwendig werden sollte eine grössere Anzahl von servern gleichzeitig laufen zu lassen. Dafür gibt es die -port=select option. Diese option wählt automatisch den nächsten freien Port. Wenn man zum Beispiel:

```
% t/TEST -start-httpd -port=select
```

ausführt und bereits eine Instanz einer anderen laufenden Testsuite die den Standard-Port 8529 nutzt, wird die neue Instanz versuchen den Server auf einem höheren Port zu starten. Momentan gibt es dort allerdings ein kleines Problem, wenn 2 Instanzen gleichzeitig gestartet werden, da nur überprüft wird, ob der Port frei ist und nicht sofort benutzt wird. So kann es passieren, dass zwei Testsuites probieren den Server auf dem selben Port zu starten. Um dieses Problem zu umgehen sollte man darauf achten so gestartete Instanzen mit einem gewissen Mindestabstand zu starten. Als Richtwert sollte man die Zeit nehmen, die das Testsystem zum konfigurieren des Servers braucht. Danach ist der Port belegt und eine neue Instanz wird nicht mehr versuchen den Server darauf zu starten.

3.3.1.6 Änderung der Konfiguration

Auch nachdem der Server mit “make test” oder “t/TEST -config” konfiguriert wurde, ist es immer noch möglich bestimmte Konfigurationsparameter zu überschreiben. Die änderbaren Parameter werden aufgelistet, wenn man den Befehl:

```
% t/TEST -help
```

ausführt.

Die vermutlich wichtigsten Parameter sind die -preamble Option zum Hinzufügen von Konfigurations-Direktiven am Anfang der httpd.conf, um beispielsweise das tracing anzustellen:

```
% t/TEST -preamble "PerlTrace all"
```

und die -postamble option um Konfigurations Direktiven am Ende der httpd.conf einzufügen, um zum Beispiel ein bestimmtes Modul zu laden.

```
% t/TEST -postamble "LoadModule dav_repos_module modules/libmod_dav_
```

Eine weitere Möglichkeit ist den Test als user “nobody” laufen zu lassen:

```
% t/TEST -user nobody
```

Und natürlich wie bereits erwähnt die -port option um den Server auf einem anderen Port laufen zu lassen:

```
% t/TEST -port 8889
```

Die -servername option um einen test auf einem anderen Server laufen zu lassen:

```
% t/TEST -servername test.server.de
```

Die -httpd option um einen anderen httpd daemon zu benutzen als den, den apxs gefunden hat.

```
% t/TEST -httpd ~/httpd-2.0/httpd
```

Die -apxs option um ein anderes apxs zu nutzen:

```
% t/TEST -apxs ~/httpd-2.0-prefork/bin/apxs
```

Eine volle Liste aller optionen und änderbaren Konfigurationsparameter erhält man mit “t/TEST -help”:

```
usage: TEST [options ...]
   where options include:
-breakpoint=bp set breakpoints (multiply bp can be set)
-bugreport     print the hint how to report problems
-clean         remove all generated test files
-configure     force regeneration of httpd.conf (tests will
    not be run)
-debug[=name]  start server under debugger name (gdb, ddd,
   etc.)
-get           GET url
-head          HEAD url
-header        add headers to (get|post|head) request
-help          display this message
-http11        run all tests with HTTP/1.1 (keep alive)
   requests
-no-httpd      run the tests without configuring or starting
    httpd
-one-process   run the server in single process mode
-order=mode    run the tests in one of the modes: (repeat|
   rotate|random|SEED)
-ping[=block]  test if server is running or port in use
-post          POST url  -postamble     config to add at
   the end of httpd.conf
-preamble      config to add at the beginning of httpd.conf
-proxy         proxy requests (default proxy is localhost)
-run-tests     run the tests
-save          save test paramaters into Apache::
   TestConfigData
-ssl           run tests through ssl
-start-httpd   start the test server
```

```
-stop-httpd     stop the test server
-times=N        repeat the tests N times
-trace=T        change tracing default to: warning, notice,
   info, debug, ...
-verbose[=1]    verbose output

   configuration options:
-access_module_name access module name
-apxs           location of apxs (default is from
   Apache2::BuildConfig)
-auth_module_name   auth module name
-bindir         Apache bin/ dir (default is apxs -q
   BINDIR)
-cgi_module_name    cgi module name
-defines        values to add as -D defines (for example
   , "VAR1 VAR2")
-documentroot       DocumentRoot (default is $ServerRoot/
   htdocs
-group          Group to run test server as (default is
   $GROUP)
-httpd          server to use for testing (default is
   $bindir/httpd)
-httpd_conf         inherit config from this file (default
   is apxs derived)
-httpd_conf_extra   inherit additional config from this file
-libmodperl         path to mod_perl's .so (full or relative
    to LIBEXECDIR)
-maxclients         maximum number of concurrent clients (
   default is minclients+1)
-minclients         minimum number of concurrent clients (
   default is 1)
-perlpod        location of perl pod documents (for
   testing downloads)
```

```
-php_module_name     php module name
-port                Port [port_number|select] (default 8529)
-proxyssl_url        url for testing ProxyPass / https (
   default is localhost)
-sbindir             Apache sbin/ dir (default is apxs -q
   SBINDIR)
-servername          ServerName (default is localhost)
-serverroot          ServerRoot (default is $t_dir)
-src_dir             source directory to look for mod_foos.so
-ssl_module_name     ssl module name
-sslca               location of SSL CA (default is $t_conf/
   ssl/ca)
-sslcaorg            SSL CA organization to use for tests (
   default is asf)
-startup_timeout     seconds to wait for the server to start
   (default is 60)
-t_conf              the conf/ test directory (default is
   $t_dir/conf)
-t_conf_file         test httpd.conf file (default is $t_conf
   /httpd.conf)
-t_dir               the t/ test directory (default is
   $top_dir/t)
-t_logs              the logs/ test directory (default is
   $t_dir/logs)
-t_pid_file          location of the pid file (default is
   $t_logs/httpd.pid)
-target              name of server binary (default is apxs -
   q TARGET)
-thread_module_name thread module name
-top_dir             top-level directory (default is $PWD)
-user                User to run test server as (default is
   $USER)
```

3.3.1.7 Schreiben von Tests

Jegliche Kommunikation zwischen Tests und dem ausführenden Test::Harness findet durch den STDOUT statt. Das heißt: was auch immer ein Test ausgibt gibt er aus, in dem er es auf STDOUT ausgibt. Wenn ein Test einige debug Informationen ausgeben soll, sollte er das tun, in dem er eine neue Zeile anfängt. Zusätzlich sollte die Zeile mit einer Raute anfangen. Am besten benutzt man zu diesem Zweck die t_debug() Funktion aus dem Apache::TestUtil package.

3.3.1.7.1 Festlegen der Anzahl der Subtests Bevor ein Subtest eines bestimmten Tests gestartet werden kann, muss er deklarieren, wie viele Subtests er enthalten wird. Wie bereits beschrieben, kann es vorkommen, das ein Test entscheidet das bestimmte Subtests übersprungen werden oder eventuell gar keine ausgeführt werden. Deshalb ist das erste was ein Test auszugeben hat:

```
1..M\n
```

M ist ein positiver Integer Wert. Wenn also der Test 5 Subtest starten wird, sollte er:

```
print "1..5\n";
```

enthalten. Mit APache::Test kann man das folgenderweise umsetzen:

```
use Apache::Test;
plan tests => 5;
```

3.3.1.7.2 Überspringen von Tests Es kann vorkommen, das ein Test nicht ausgeführt werden kann, weil bestimmte Voraussetzungen nicht erfüllt sind. Das teilt man Test::Harness durch folgende Ausgabe mit:

```
print "1..0 # skipped because of foo is missing\n";
```

Der optionale Kommentar nach der Raute wird als Grund für das Überspringen des Tests genutzt. Das optionale letzte Argument in der plan()-Funktion von Apache::Test kann zum Definieren der Voraussetzungen genutzt werden, die nötig sind um einen Test zu starten und gegebenenfalls zu überspringen:

```
use Apache::Test;
plan tests => 5, $test_skipping_prerequisites;
```

Optionen für dieses letzte Argument:

- **ein SKALAR**
  der Test wird übersprungen, wenn das Skalar einen falschen Wert hat. Zum Beispiel:

  ```
  plan tests => 5, 0;
  ```

  So bekommt man allerdings keine Gründe über den Grund für das Überspringen. Deshalb ist es besser “have()” zu benutzen.

  ```
  plan tests => 5,
          have 'LWP',
                  { "not Win32" => sub { $^O eq 'MSWin32'} };
  ```

- **eine ARRAY Referenz**
  Die Funktion have_module() wird für jeden Wert im Array aufgerufen. Der Test wird übersprungen, wenn hav_module() ein false zurück gibt. Das passiert, wenn mindestens ein C oder Perl Modul aus der Liste nicht gefunden wird. Ein Beispiel:

  ```
  plan tests => 5, [qw(mod_index mod_mime)];
  ```

- **eine CODE Referenz**
  Ein Test wird übersprungen, wenn die Funktion den Rückgabewert false hat. Zum Beispiel:

  ```
  plan tests => 5, \&have_lwp;
  ```

  Hier wird der Test übersprungen, wenn LWP nicht verfügbar ist.

Es gibt eine Reihe von nützlichen Funktionen, deren Rückgabewert sich als das letzte Argument für die Funktion “plan()” eignen:

- **have_module()**
  Die Funktion have_module() überprüft das Vorhandensein von Perl- und C-Modulen. Die Funktion akzeptiert als Eingabe eine Liste von Modulen, oder eine Referenz auf eine Liste. Wenn eins der Module nicht gefunden wird gibt die Funktion ein "false" zurück, andernfalls ein "true". Zum Beispiel:

  ```
  plan tests => 5, have_module qw(Chatbot::Eliza CGI mod_proxy);
  ```

  Der ganze Test wird übersprungen, wenn nicht beide Module vorhanden sind. In diesem Fall das Perl Modul Chatbot::Eliza oder das C Modul mod_proxy.c.

- **have_min_module_version()**
  Diese Funktion kann benutzt werden um eine kleinst erlaubte Version eines Moduls vorrauszusetzen:

  ```
  plan tests => 5, have_min_module_version(CGI => 2.81);
  ```

  Hier wird für das Modul CGI.pm die Version 2.81 oder grösser vorrausgesetzt. Diese Funktion funktioniert im Moment nur für Perl Module.

- **have()**
  Die have() Funktion kann als letztes Argument von plan() gleich mehrer Voraussetzungen erzwingen bzw. überprüfen. Have() kann als Argumente Skalare beinhalten, die an die have_module() Funktion übergeben werden, und hash Referenzen. Wenn hash Referenzen genutzt werden, sind die Keys des Hashes strings, die einen Grund für den Misserfolg dieses Eintrags enthalten. Die Values sind der Zustand für ein Gelingen, wenn sie "true" sind. Wenn der Wert ein Skalar ist, wird er benutzt wie er ist. Wenn der Wert eine ode Referenz ist, wird der Code zum Zeitpunkt des Überprüfens ausgeführt und der Rückgabewert wird herangezogen um den Zustand zu überprüfen. Wenn der Zustand stets fehlschlägt, wird der im Schlüssel hinterlegte Grund ausgegeben um dem Benutzer zu sagen, warum der Test übersprungen wurde. Zum Beispiel:

  ```
  plan tests => 5,
          have 'LWP',
          { "perl >= 5.8.0 is required" => ($] >= 5.008) },
          { "not Mac OS X" => sub { $^O eq 'darwin' },
  ```

```
"foo is disabled" => \&is_foo_enabled,
},
'cgid';
```

In diesem Beispiel wird vorausgesetzt, dass das LWP Perl Modul vorhanden ist, das die Perlversion auf dem System größer oder gleich 5.8.0 ist, das Betriebssystem Mac OS X ist, das is_foo_enabled ein "true" zurückgibt und das Modul mod_cgid vorhanden ist. Wenn nur eine von diesen Voraussetzungen nicht erfüllt wird, wird der Test übersprungen und jede unerfüllte Voraussetzung wird einen Grund für das Versagen ausgeben.

- **have_perl()**
  Diese Funktion überprüft, ob der Wert von $Config{foo} oder $Config{usefoo} gleich dem definierten Wert ist. Zum Beispiel:

  ```
  plan tests => 2, have_perl 'ithreads';
  ```

  Wenn in diesem Beispiel Perl nicht mit der option -Duseithreads kompiliert wurde, wird der Test übersprungen. Man kann auch gezielt Perl Erweiterungen abfragen:

  ```
  plan tests => 5, have_perl 'iolayers';
  ```

  Hier wird überprüft ob PerlIO verfügbar ist.

- **have_min_perl_version()**
  Wird benutzt um eine kleinst erlaubte Version von Perl vorauszusetzen. Zum Beispiel:

  ```
  plan tests => 5, have_min_perl_version("5.008001");
  ```

  Hier wird mindestens Perl 5.8.1 oder grösser vorausgesetzt.

- **have_threads()**
  Die Funktion have_threads überprüft, ob threads unterstützt sind. Das gilt für Perl als auch Apache.

  ```
  plan tests => 2, have_threads;
  ```

- **under_construction()**
  Dies ist nur eine eine nützliche Platzhalterfunktion. Sie wird den Test überspringen lassen und "skipped: this test is under construction" ausgeben.

  ```
  plan tests => 2, under_construction;
  ```

- **have_lwp()**
  Überprüft ob das Perl Modul LWP installiert ist.

- **have_http11()**
  Versucht der LWP mitzuteilen, das der aktuelle Subtest mit dem HTTP 1.1 protocol gefahren werden soll. Die Funktion gibt ein "false" zurück, falls die installierte Version von LWP dazu nich in der Lage ist.

- **have_cgi()**
  Überprüft ob mod_cgi oder mod_cgid verfügbar ist.

- **have_apache()**
  Diese Funktion überprüft ob eine spezifische httpd generation vorhanden ist.

  ```
  plan tests => 2, have_apache 2;
  ```

  Dies wird den Test überspringen, wenn der Test nicht auf einem Apche Server der zweiten Generation läuft (httpd-2.x.xx).

  ```
  plan tests => 2, have_apache 1;
  ```

  Dies wiederum wird den Test überspringen, wenn er nicht auf einem Apache der ersten Generation läuft (httpd-1.3.xx).

- **have_min_apache_version()**
  Kann benutzt werden, um eine kleinst erlaubte Version von Apache festzulegen.

  ```
  plan tests => 5, have_min_apache_version("2.0.40");
  ```

  Dies würde eine pache Version grösser oder gleich Apache 2.0.40 voraussetzen.

- **have_apache_version()**
  Kann benutzt werden um eine genau spezifizierte Version von Apache vorauszusetzen,

```
plan tests => 5, have_apache_version("2.0.40");
```

Hier wird nur die Version 2.0.40 des Apache Servers zugelassen. Ansonsten wird der Test übersprungen.

3.3.1.7.3 Ausgabe von Erfolg oder Fehlschlag eines Subtest Nach dem Ausgeben der Anzahl der geplanten Subtests, angenommen die Tests werden nicht übersprungen, werden die entsprechenden Subtests ausgeführt und jeder Subtest gibt seine Ergebnisse aus. Von einem Subtest wird erwartet, das er seine Ausgabe wie folgt formatiert. Der Erfolg des Tests wird durch ein "ok" oder "not ok" signalisiert gefolgt von einer aufsteigenden Nummer und einem Linefeed.

```
print "ok 1\n";
print "not ok 2\n";
print "ok 3\n";
```

In Apache::Test wird dies durch die ok() Funktion realisiert. Sie gibt ein "ok" aus, wenn ihr Argument ein wahrer Wert ist. Andernfalls gibt sie ein "not ok" aus. Zusätzlich zählt die Funktion ihre Aufrufe und gibt diese inkrementelle Nummer wie erfordert aus. Dadurch ist es möglich Subtests zu verschieben ohne an das Ändern der Nummer denken zu müssen. Dieser Testcode:

```
use Apache::Test;
use Apache::TestUtil;
plan tests => 3;
ok "success";
t_debug("expecting to fail next test");
ok "";
ok 0;
```

wird folgendes ausgeben:

```
1..3
ok 1
# expecting to fail next test
not ok 2
```

```
not ok 3
```

Die meisten Subtest werden sich um folgende Dinge kümmern:

- Testen ob eine Variable definiert ist:

  ```
  ok defined $object;
  ```

- Testen ob eine Variable den boolschen Wert wahr oder falsch hat:
  wahr:

  ```
  ok $value;
  ```

  flasch:

  ```
  ok !$value;
  ```

- Testen ob ein von irgendwoher erhaltener Wert einem erwartetem Wert entspricht:

  ```
  $expected = "correct value";
  $received = get_value();
  ok defined $received && $received eq $expected;
  ```

3.3.1.7.4 Subtest überspringen Wenn die auf die Standardausgabe ausgegebene Zeile den Substring "# Skip" (Variationen bei spacing und case sind erlaubt) nach einem "ok" oder "ok" + Nummer enthält, wird der Test als übersprungen gezählt. Test::Harness gibt den Text nach dem "# Skip" als Grund für das Überspringen aus. Es ist also einfach möglich einen Subtest als übersprungen zu zählen, wenn man folgendes ausgibt:

```
print "ok 3 # Skip for some reason\n";
```

Natürlich stellt auch wieder Apache::Test eine entsprechende Funktion zur Verfügung, um diese Ausgabe komfortabler zu gestalten. Die Funktion skip() erfüllt genau diese Funktionalität. Sie funktioniert ähnlich wie die ok() Funktion:

```
skip $should_skip, $test_me;
```

Wenn also $should_skip wahr ist, wird der Test als übersprungen gemeldet. Das zweite Argument ist das Argument, das an ok() gesendet wird. Wenn $should_skip ein "falsch" liefert wird ein normaler ok() Subtest gefahren. Das folgende Beispiel demonstriert 4 mögliche Ergebnisse der skip() Funktion:

skip_subtest_1.t

```
use Apache::Test;
plan tests => 4;
my $ok = 1;
my $not_ok = 0;
my $should_skip = "foo is missing";
skip $should_skip, $ok;
skip $should_skip, $not_ok;
$should_skip = '';
skip $should_skip, $ok;
skip $should_skip, $not_ok;
```

Wenn der Test ausgeführt wird erhalten wir folgende Ausgabe:

```
% ./t/TEST -run-tests -verbose
skip_subtest_1 skip_subtest_1....1..4
ok 1 # skip foo is missing
ok 2 # skip foo is missing
ok 3
not ok 4
# Failed test 4 in skip_subtest_1.t at line 13
Failed 1/1 test scripts, 0.00% okay. 1/4 subtests failed, 75.00% okay
```

Wie man sieht, wurden die ersten beiden Subtests explizit übersprungen (mit %shoul_skip als Grund), da should_skip() ein "wahr" geliefert hat. Das zweite Argument war also nicht mehr wichtig. In den letzen beiden Subtests liefert $should_skip() den Wert "falsch" zurück, deshalb wurde das zweite Argument an die Funktion on() übergeben. Der Code:

```
$should_skip = '';
```

```
skip $should_skip, $ok;
skip $should_skip, $not_ok;
```

ist äquivalent zu dem Code:

```
ok $ok;
ok $not_ok;
```

Wenn man allerdings t_cmp() oder andere Funktionen die Argumente für die Funktion ok() aufrufen lassen möchte, wird das nicht ganz funktionieren, da die Funktion immer aufgerufen wird, egal ob das erste Argument ein wahr oder falsch zurückgibt. Wenn man Beispielsweise eine Funktion:

```
ok t_cmp($received, $expected, $comment);
```

und man diesen Test nur laufen lassen möchte, wenn das Modul HTTP::Date vefügbar ist, wäre der erste Ansatz:

```
my $should_skip = eval { require HTTP::Date } ? "" : "missing HTTP::I
skip $should_skip, t_cmp($received, $expected, $comment);
```

Das Problem ist, das so t_cmp() auch ausgeführt wird, wenn HTTP::Date nicht vorhanden ist. Eine besserer weg diese Funktionalität umzusetzen ist:

```
if (eval {require HTTP::Date}) {
        ok t_cmp($received, $expected, $comment);
        }
else {
        skip "Skip HTTP::Date not found";
}
```

3.3.2 Codegenerierung mit Python für Unit Tests

Zur weitergehenden Untersuchung des Moduls Catacomb wurde die Möglichkeit erwogen, mittels automatisch generiertem Python Code zum wrappen des Moduls Unittest zu erzeugen. Die Idee Python für diesen Zweck zu nutzen entstand wieder aus dem Gedanken, das die Sprache sehr leicht leserlich und komfortabel ist. In Python

Unittests zu schreiben ginge um ein vielfaches einfacher als in C. Es galt nun also herauszufinden, ob diese Methode für eine spätere Erweiterung der Testumgebung praktikabel ist.

3.3.2.1 ctypes

Code Generation zum Wrappen von c Datentypen, Funktionen, Konstanten, usw. mit der ctypes Bibliothek ist nicht besonders schwer. Es gibt ein Tutorial auf den Wikiseiten des ctypes Projektes, das die Vorgehensweisen und Konzepte sehr gut beschreibt. [4]Bei größeren Projekten kann das aber umständlich werden, wenn es gilt sehr viele Funktionen zu wrappen. Deshalb wurde versucht mit Hilfe der Tools der Bibliothek "ctypeslib" herauszufinden, wie einfach es ist dieses Wrappen zu automatisieren.

3.3.2.2 ctypeslib

Um den Prozess des Wrappens zu automatisieren gibt es die ctypeslib. Die ctypeslib beinhaltet einen Codegenerator für genau diesen Zweck.

3.3.2.2.1 Parsen der Headerdatei Das Skript "h2xml.py" der ctypeslib greift auf GCCXML zurück. Dies ist eine spezielle Version des GCC-Kompilers, der den geparsten Sourcecode als XML-Baumstruktur ausgibt. Es wurde sehr schnell deutlich das diese spezielle Version des Kompilers noch nicht wirklich ausgereift ist. Die Ergebnisse waren weniger als befriedigend.

Trotz der Bemühungen, die Catacomb Header mit einer als statischen Bibliothek kompilierten Version von Apache zum Erfüllen der Speichermanagement Abhängigkeiten zu parsen, ist der Compiler regelmäßig ohne Aussagekräftige Fehlermeldung abgestürzt.

3.3.2.2.2 Erzeugen des C Wrappers Würde das Erzeugen der XML Datei erfolgreich sein, ist für den nächsten Schritt das Erzeugen des Python-Wrapping-Codes geplant. Diese Aufgabe übernimmt das in ctypeslib enthaltene Skript "xml2py.py".

Dieses Skript würde dann richtig ausgeführt ein Python Modul ausgeben, das sämtliche Funktionalitäten des C Codes als Python Methoden dieses Moduls zur Verfügung stellt. Da leider nicht einmal das Parsen der Headerdatei zuverlässig funktioniert hat, konnte dieser Schritt nie ausgeführt werden.

3.4 Verfügbare Tools

Es gibt eine Vielzahl von fertigen Tools die sich bereits mit dem Testen der WebDAV Spezifikationen befassen. Bei dem Entwurf der Testumgebung wurden alle bereits vorhandenen Tools dieser Art untersucht, um herauszufinden, ob diese eventuell ausreichen um alle wichtigen Spezifikationen abzudecken. Das hätte den Vorteil, das möglichst wenig Betreuungsaufwand für neuen Code durch die Tests entsteht, da diese Tools alle unter Opensource Lizenzen stehen und bereits von anderen Entwicklern gepflegt werden.

3.4.1 Cadaver

Cadaver ist ein Kommandozeilen WebDAV Client. Er implementiert nahezu alle in den RFCs festgelegten Protokollerweiterungen für WebDAV und eignet sich somit sehr gut um ein aufgesetztes WebDAV System schnell manuell zu testen.

Cadaver ist als freie Software unter der GPL-Lizenz [21]verfügbar.

3.4.2 Litmus

Litmus ist eine Kommandozeilen Test Suite um einen WebDAV Server gegen die Spezifikation in der RFC 2518 [12] zu testen.

Aktuell implementierte Tests sind:

- OPTIONS für DAV: header
- PUT, GET mit anschliessendem byte Vergleich
- MKCOL

- DELETE (collections, non-collections)
- COPY, MOVE mit Kombinationen aus:
  - überschreiben TRUE/FALSE
  - Ziel existiert/existiert nicht
  - collection/non-collection
- Property Veränderung und Abfrage:
  - setzen, löschen, ersetzen von Properties
  - erhalten von dead properties nach COPY
  - namespace Behandlung
- Locking
  - versuchen geschützte resourcen zu ändern (als lock Besitzer und nicht Besitzer)
  - shared/exclusive locks
  - entdecken von locks
  - collection locking
  - lock refresh

3.4.3 Prestan

Prestan ist eine Kommandozeilen Performance Test Suite für WebDAV Server. Mit Prestan ist es möglich, die Geschwindigkeit von diversen WebDAV Methoden zu überprüfen. Folgende Methoden werden von Prestan auf Ihre Performance hin überprüft:

- PROPPATCH
  - ändern/löschen von Properties. einzeln und mehrere auf einmal.
- PROPFIND

  - finden von dead und live properties.

- PUT
  - ablegen von Resourcen unterschiedlicher Grösse
- GET
  - herunterladen von Resourcen unterschiedler Grösse
- COPY
  - kopieren von Resourcen und Collections
- MOVE
  - verschieben von Reourcen und Collections
- DELETE
  - löschen von Resourcen und Collections
- MKCOL
  - erstellen von Collections
- LOCK
  - schützen von Resourcen und Collections

3.4.4 DAVTool

Davtool ist ein Kommandozeilen Tool das WebDAV Methoden ausführen kann. Davtool funktioniert ähnlich wie das bekannte wget, unterstützt allerdings im Gegensatz zu wget nicht nur die GET Methode sondern sämtliche WebDAV Methoden. DAVTool eignet sich deshalb hervorragend für batch Aufgaben. Es ist gut dazu geeignet um es in eigene Skripte zu integrieren und Beispielsweise eine gezielte Suche auf einem Repository durchzuführen.

So eine Suche würde Beispielsweise so aussehen:

```
% davtool http://localhost/repos -SEARCH -i query.xml
```

Die Suchanfrage selber wird in einer XML Datei formuliert und dann als Requestbody angefügt. Die Grundsemantik von Suchen ist im Anhang zu finden. Als Beispiel könnte hier der Inhalt der Datei query.xml folgenderweise aussehen:

```
<d:searchrequest xmlns:d="DAV:">
 <d:basicsearch>
  <d:select>
   <d:prop><d:getcontentlength/></d:prop>
  </d:select>
  <d:from>
   <d:scope>
    <d:href>/repos/</d:href>
    <d:depth>infinity</d:depth>
   </d:scope>
  </d:from>
  <d:where>
   <d:gt>
    <d:prop><d:getcontentlength/></d:prop>
    <d:literal>10000</d:literal>
   </d:gt>
  </d:where>
  <d:orderby>
   <d:order>
    <d:prop><d:getcontentlength/></d:prop>
    <d:ascending/>
   </d:order>
  </d:orderby>
 </d:basicsearch>
</d:searchrequest>
```

Diese Suchanfrage generiert eine Liste aller Werte von allen Resourcen, die sich in dem Repository “/repos/” befinden und eine Länge von 10000 überschreiten. Die Liste ist aufsteigend nach Länge sortiert.

Die verfügbaren Parameter von DAVTool sind:

-u, –username username

-p, –password password

-o, –outfile outputfile

-i, –infile request_body_file

-d, –depth depth

-m, –method method (Default: GET)

-t, –type content-type (Default: text/xml)

3.5 Automation

Zur Automatisierung mussten nicht viele Tools untersucht werden. Da in der vorhanden Infrastruktur bereits Hudson enthalten ist und Hudson sich hervorragend zur Automatisierung von Build Prozessen und Tests eignet, wurde beschlossen das so auszunutzen und Hudson zu verwenden.

Kapitel 4

4 Implementierung

4.1 Aufbau der Testumgebung

Nach der Evaluierungsphase wurde festgelegt, die Umgebung wie folgt aufzubauen. Dreh- und Angelpunkt der Umgebung soll die Continous Integration Engine Hudson werden. Hudson wird so konfiguriert, das es das auf tigris befindliche Subversion-Repository überwacht und im Falle eines Eincheckens von neuem Sourcecode automatisch einen Build-Prozess anstößt. Nach dem erfolgreichen Abschließen dieses Builds sollen nun automatisiert das Litmus Testutilitiy gestartet werden und deren Ausgabe analysiert werden.

Der Fokus liegt also erst einmal auf dem Blackbox-Testen des Catacomb Moduls, um sicherzustellen, das die Funktionalität nach Sourcecodeveränderungen erhalten bleibt, also keine Regressionen auftreten.

Zusätzlich soll auch Prestan ausgeführt werden. Ebenfalls automatisiert und mit einer folgenden Analyse der Ausgabe, um die Performance des Moduls zu überwachen, bzw. sicherzustellen, dass selbige erhalten bleibt oder durch entsprechende Veränderungen im Code gesteigert wird. Als Skriptsprache wurde wie bereits erwähnt Python gewählt. Python bietet für das Parsen der Ausgabe eine Menge nützlicher Bibliotheken und Hilfen.

Als Betriebssystem für die Testumgebung wurde Ubuntu 8.04 LTS in der Server Version gewählt.

Die Infrastruktur der Testumgebung, also die Installationen von Apache Versionen, MySQL Versionen und sämtlichen anderen Abhängigkeiten sollen statisch vorinstalliert sein, damit gewährleistet ist, das sie so wie sie vorhanden ist auch tatsächlich funktioniert.

Eine weitere wichtige Entscheidung war, dass die Infrastruktur der Tests selber in virtuellen Maschinen laufen soll. Das vermindert den Hardwareaufwand um die Tests laufen zu lassen und vereinfacht die Installation der Infrastruktur, da sie in dem Festplatten-Image der Virtuellen Maschine schon komplett vorhanden ist. So eine VM kann sich ein Entwickler im Notfall auch lokal auf seinem Computer installieren und hat so jederzeit die Möglichkeit ohne großen Installationsaufwand Tests gegen verschiede Apacheversionen zu fahren.

Das Apache::Test Framework wurde zum Erstellen von Tests ausgeschlossen. Es bietet zwar eine Fülle an nützlichen Funktionen rund um das Apache-Zentrische Testen, eignet sich aber letztendlich doch eher dazu es mit dem Sourcecode herauszugeben und es in einem Userumfeld zu testen. Mit einfacheren Test, wie beispielsweise einfach dem Anfordern einer Resource, um nach der Installation festzustellen, ob die Installation erfolgreich war. Die komplette Abbildung der Funktionalitäten der RFC2518 auf das Apache::Test Framework wäre viel zu aufwendig, gerade da mit Litmus bereits ein gut geeignetes Werkzeug zum Testen der Spezifikation vorliegt. Es wurde also beschlossen die Tests eher entwicklerspezifisch zu belassen.

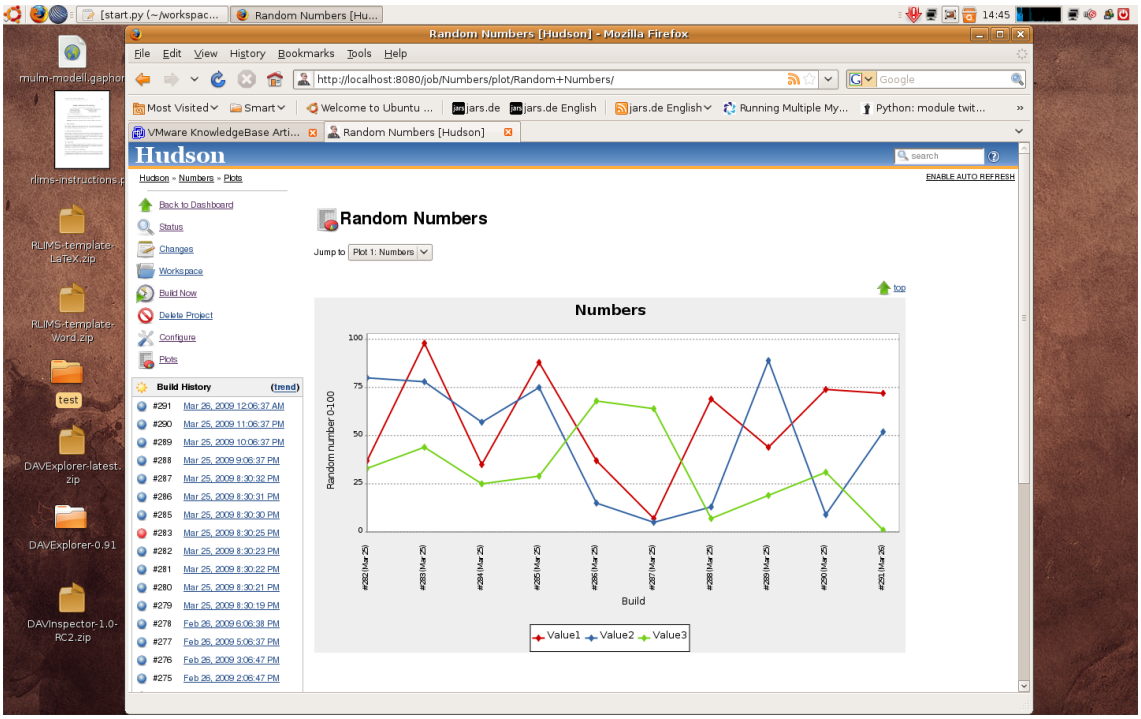
4.1.1 Betriebssystem

Als Betriebssystem für die Testumgebung wurde Ubuntu 8.04 LTS in der Server Version gewählt.

Abb. 4.1: Logo Ubuntu

Ubuntu zeichnet sich durch eine besonders einfache und reibungslose Installation aus. Der Support ist hervorragend, es erscheinen regelmäßig Updates und gerade die LTS (Long Time Support) Version gewährleistet, dass das System lange nutzbar und up-to-date ist ohne ein komplettes Distributionsupgrade fahren zu müssen. Das kann unter Linux häufig zu Problemen führen. Aus eigener Erfahrung musste ich feststellen, das Distributionsupgrades nicht immer reibungslos ablaufen. Für die ersten Tests und Versuche mit Hudson und Testen von Installationsvarianten der Infrastruktur habe ich eine Bereits fertig konfigurierte Ubuntu Desktop VM von jars.de[17] verwandt.

Abb. 4.2: Screenshot Ubuntu Desktop



Nach einem Distributionsupgrade war das System nicht mehr bootbar. Glücklicherweise handelte es sich wie bereits beschrieben um eine VM von der ich auch regelmäßig Backups erstellt habe, so war es ein leichtes war die Fehlerquellen für ein erfolgreiches neues upgrade zu entfernen. Allerdings entstand so der Wunsch für die Produktionsversion der Testumgebung ein möglichst lang aktiv supportetes System zu verwenden.

4.1.2 Vmware

Wie bereits beschrieben soll die Testumgebung in einer Virtuellen Maschine realisiert werden.

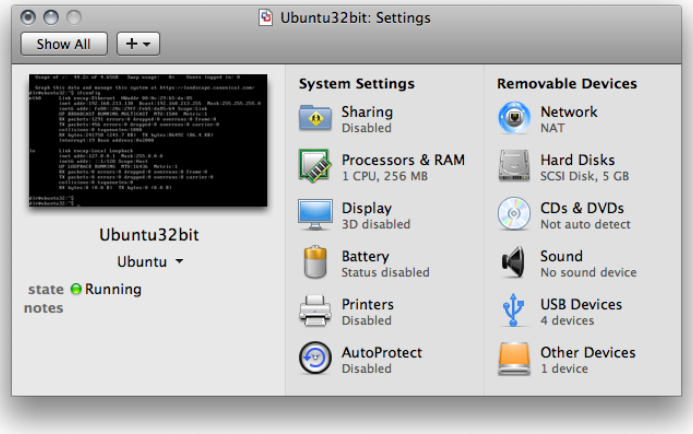
Es wurde beschlossen, zusätzlich zu einer normalen 32-Bit Version der Virtuellen Maschine auch noch eine komplett identische VM in 64-Bit Technik aufzubauen. Nahezu jede aktuell erhältliche Hardware ist 64-Bit basiert. Im Serverbereich mittlerweile ausschließlich 64-Bit. Bisher wurden überhaupt keine Versuche unternommen wie

sich und ob sich Catacomb in einer komplett 64-Bit kompilierten Umgebung anders verhält. Das gros der bisherigen Installationen läuft in 32-Bit Umgebungen, deshalb hat das Testen dort absoluten Vorrang.

Zu diesem Zweck wurden 2 identische Virtuelle Maschinen erzeugt. Jeweils mit einem virtuellen Prozessor und vorläufig 256MB Arbeitsspeicher und einer virtuellen Festplatte mit 5GB Speicher. Die Maschinen sollen identisch sein, damit man später besser Vergleiche zwischen ihnen anstellen kann.

Die VM's wurden der Einfachheit halber erst einmal in VMware Fusion erstellt. Der Mac OS X Version der Desktop Applikation der VMware-Lösungen.

Abb. 4.3: VMware Konfiguration



So war es möglich, die Virtuelle Maschine lokal zu erstellen und jederzeit an ihr arbeiten zu können.

In einer der Maschinen wurde nun die 32-Bit Version von Ubuntu Server 8.04 LTS installiert, in der anderen die 64-Bit Version. Anschließend wurden sie parallel mit unterschiedlichen Apache httpd Versionen, MySQL Versionen, PostgresSQL Versionen und zusätzlich Release Versionen von Catacomb befüllt, um auch unterschiedliche Konfigurationen von Versionen testen zu können.

Um später einfach auf die VM's zugreifen zu können wurde auch der openssh-Server installiert. Er erlaubt es sich per Kommandozeile remote an dieser Maschine

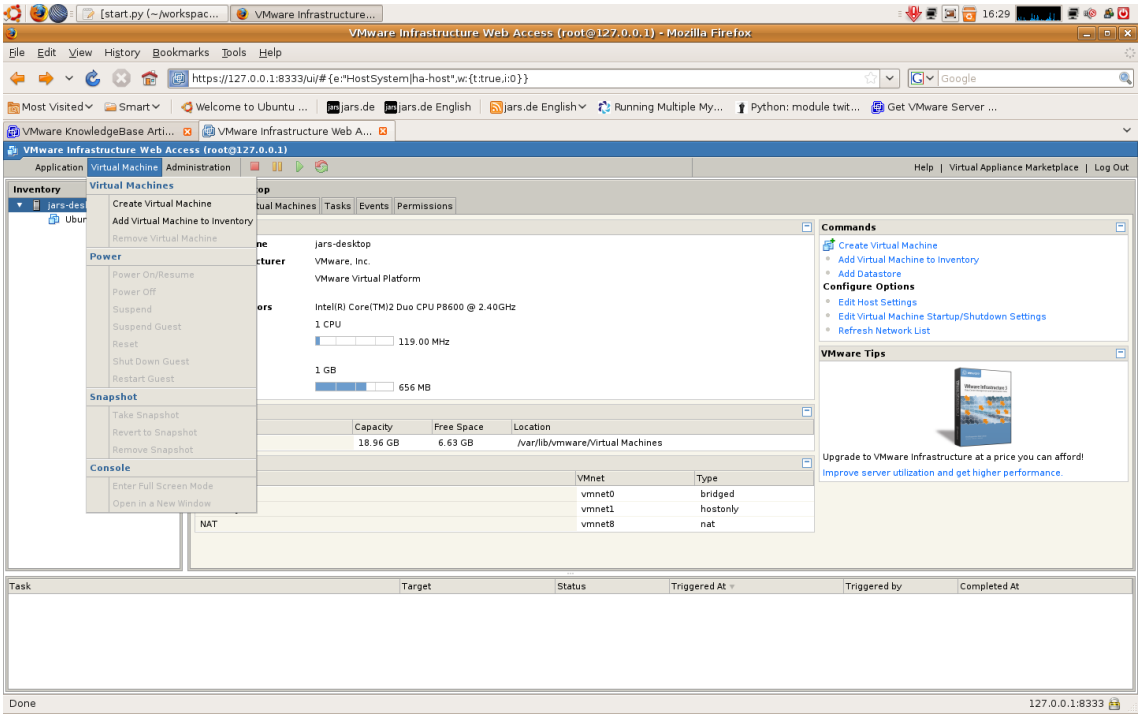
anzumelden und Befehle auszuführen. Das ist wichtig für die spätere Automation mit Hudson.

Da die VM's komplett kompatibel sind, ist es ohne weiteres möglich die Virtuelle Maschine zu einem späteren Zeitpunkt in den Wmware-Server zu integrieren und sie von dort aus nutzbar zu machen. Der VMware-Server hat unter Linux, dem zukünftigen Hostsystem für die virtuellen Maschinen einen festgelegten Ordner:

`/var/lib/vmware/Virtual Machines/`

Dort können beliebig viele Maschinen hinterlegt werden und über die Graphische Benutzeroberfläche des Servers in das allgemeine Inventar an virtuellen Maschinen hinzugefügt werden.

Abb. 4.4: VMware Server Graphische Weboberfläche



Wie auf dem Screenshot ersichtlich, ist in diesem Inventar bereits eine Maschine hinterlegt. Es handelt sich um die 32-Bit Installation von Ubuntu. Über die graphische Weboberfläche können nun leicht weitere Maschinen hinzugefügt werden.

4.1.3 Installation MySQL

Die Installationen von MySQL wurden nach dem Schema:

```
/home/dlr/ajimushkay/mysql/VERSIONNUMBER
```

vorgenommen. So lassen Sich die unterschiedlichen Versionen leicht finden d.h. sehr einfach über die Versionsnummer lokalisieren. Das war wichtig im Hinblick auf die Automatisierung der Testumgebung. So ist es möglich, nur durch die Information das es sich bei der Datenbank um eine MySQL-Datenbank handeln soll und die Versionsnummer den exakten Pfad der Installation herauszubekommen.

Durch diese Anordnung in der Verzeichnisstruktur soll die Installation auch komplett unabhängig vom restlichen System sein und damit komplett abgekapselt. So stören sich unterschiedliche Versionen nicht gegenseitig durch Konfigurationsdateien, PID- oder Lockingdateien.

Zusätzlich zum allgemeinen Installationsort, ist es bei MySQL wichtig anderen bestimmten Dateien einen festen Installations-Ort mitzuteilen, wenn man den Sourcecode per:

```
./configure
```

konfiguriert. Ein einfaches Ändern des Installationspräfixes von dem Standardort "/usr/local/mysql" auf zum Beispiel diesen Installationsort:

```
./configure --prefix=/home/dlr/ajimushkay/mysql/5.0.67
```

führt dazu, das trotzdem noch diverse Teile der MySQL Infrastruktur an Standard Systemorte geschrieben werden. Es müssen also noch mehr configure Optionen genutzt werden um eine komplett gekapselte Installation von MySQL zu erhalten.

Die wichtigsten zusätzlichen configure Optionen in dem Zusammenhang sind:

–exec-prefix installiert Architekturabhängige Dateien in dem übergebenen Verzeichnis

–with-tcp-port ändert den Standard Port

–with-pid_file ändert den Ort der pid Datei des Prozesses

–with-unix-socket-path ändert den Ort der Unix Domänen-Socket

–localstatedir ersetzt die Standardposition der Datenbankverzeichnisse (normalerweise /usr/local/var)

Ein configure Lauf der eine komplett gekapselte Installation erzeugt sieht folgenderwiese aus:

```
./configure
--prefix=/home/dlr/ajimushkay/mysql/5.0.67
--enable-local-infile
--exec-prefix=/home/dlr/ajimushkay/mysql/5.0.67
--with-base_dir=/home/dlr/ajimushkay/mysql/5.0.67
--with-tcp-port=5067
--with-log=/home/dlr/ajimushkay/mysql/5.0.67/mysqld.log
--with-pid_file=/home/dlr/ajimushkay/mysql/5.0.67/mysqld.pid
--with-unix-socket-path=/home/dlr/ajimushkay/mysql/5.0.67/
    mysqld.sock
--localstatedir=/home/dlr/ajimushkay/mysql/5.0.67
--enable-thread-safe-client
```

Danach kann der übliche Build und Installationsprozess, wie unter Linux und Unix üblich angestossen werden:

```
make
make install
```

Notwendige Post-Installations-Schritte sind, die vom Make Prozess erzeugte Konfigurationsdatei aus dem Source Verzeichnis ins Installations Verzeichnis zu kopieren. Leider passiert das nicht automatisch.

```
cp support-files/my–medium.cnf /home/dlr/ajimushkay/mysql
    /5.0.67/my.cnf
```

Zusätzlich mus die Datenbank noch mit folgendem Befehl initialisiert werden:

```
/home/dlr/ajimushkay/mysql/5.0.67/bin/mysql_install_db
```

Nun ist die Installation einsatzbereit.

4.1.4 Installation PostgresSQL

Die Installation von PostgreSQL gestaltet sich ähnlich. Allerdings reicht hier ein einfaches Installationspräfix:

```
./configure --prefix=/home/dlr/ajimushkay/pgsql/8.3.6
```

Der Build und Installationsprozess wird wie folgt gestartet:

```
make
make install
```

Bei PostgreSQL ist es noch notwendig die erforderlichen internen Datenbanken anzulegen. Der Befehl:

```
mkdir /home/dlr/ajimushkay/pgsql/8.3.6/data /home/dlr/
    ajimushkay/pgsql/8.3.6/bin/initdb -D /home/dlr/ajimushkay
    /pgsql/8.3.6/data/
```

initialisiert diese notwendigen Datenbanken.

4.1.5 Installation Apache

Die Installationen von Apache wurden nach dem Schema:

/home/dlr/ajimushkay/apache/VERSIONNUMBER

angelegt. Auch hier ergibt sich wieder der Vorteil, das so die Installationen leicht zu lokalisieren und somit einfach durch Skriptlösungen quasi anzusteuern sind.

Anders als bei MySQL reich hier ein einfaches ändern des Installationpräfixes. die anderen Optionen dienen dem folgenden:

–enable-so erlaubt es C-Module nachzuinstallieren

–enable-dav aktiviert die WebDAV Funktionalitäten. Dies ist wichtig für die Installation von Catacomb

–enable-dav-fs aktiviert die WebDAV Filesystem-Funktionalitäten. Ebenfalls für die Installation von Catacomb vorausgesetzt

–enable-dbd aktiviert die Datenbankunterstützung

–with-included-apr teilt der Installation mit die mitgelieferte Apache Runtime Library zu nutzen

Eine funktionierender configure Lauf der eine gekapselte Installation von Apache liefert wäre beispielsweise:

```
./configure
--prefix=/home/dlr/ajimushkay/apache/2.2.10
--enable-so
--enable-dav
--enable-dav-fs
--enable-dbd
--with-included-apr
```

Aufpassen muss man man bei Apache Versionen >= 2.2.6 hier ist der MySQL-Treiber noch nicht in der Sourcedistribution enthalten. Dort ist es notwendig die Datei "apr_dbd_mysql.c" aus einer neueren Version, die diesen Treiber enthält, beispielsweise Apache 2.2.8, in das Verzeichnis "/home/dlr/ajimushkay/httpd/2.2.6/srclib/apr-util/dbd" zu kopieren. Die nacheinanander ausgeführten Befehle:

```
cd /home/dlr/ajimushkay/httpd/2.2.6/srclib/apr-util/
./buildconf
./configure --with-apr=../apr
```

kompilieren dann den für die MySQL-Anbindung notwendigen Treiber.

Auch hier wird der Build und Installationsprozess Linux üblich durchgeführt:

```
make
make install
```

4.1.6 Installation Catacomb

Die Installationen von Catacomb wurden nach dem Schema:

/home/dlr/ajimushkay/catacomb/VERSIONNUMBER

angelegt. Auch hier ergibt sich wieder der Vorteil, dass so die Installationen leicht zu lokalisieren und somit einfach durch Skriptlösungen quasi anzusteuern sind.

4.1.6.1 Installation mit MySQL

Nachdem nun MySQL, PostgreSQL und Apache installiert sind, kann das Catacomb Modul in eine gewünschte Apache Version installiert werden. Hier ein beispielhafter configure Aufruf um Catacomb als Modul in den apche httpd Version 2.2.8 mit Support für MySQL-Datenbanken zu installieren:

```
./configure
--with-apache=/home/dlr/ajimushkay/apache/2.2.8
--with-mysql=/home/dlr/ajimushkay/mysql/5.0.67
--with-debug
```

–with-apache hier wird die zu verwendenede Apache Installation angegeben

–with-mysql hier wird die MySQL Installation angegeben, die verwendet werden soll

–with-debug stellt zusätzliche debugging Ausgaben des Moduls an

Dann wieder das übliche:

```
make
make install
```

Nach der Installation muss noch die für Catacomb notwendige Datenbank in MySQL angelegt werden. Die dafür notwendigen SQL-Statements sind in der Source Distribution enthalten. Sie werden durch ausführen der folgenden Kommandos für die gewünschte MySQL Version ausgeführt:

```
/home/dlr/ajimushkay/mysql/5.0.67/bin/mysqladmin -u root
    create repos /home/dlr/ajimushkay/mysql/5.0.67/bin/mysql
    -u root repos < table.sql /home/dlr/ajimushkay/mysql
    /5.0.67/bin/mysql -u root repos < data.sql
```

Zusätzlich muss nun noch die Konfiguration des Apache Servers angepasst werden, damit der Apache Server das neu installierte Modul verwendet, um WebDAV Anfragen an eine bestimmte URL abzuwickeln.

```
DavDBMSTmpDir /tmp/
#DavDBMSFileDir /home/dlr/ajimushkay/apache/2.2.8/htdocs/
    repos/
#DavDMBSMaxFileSize 102400

<Location /repos>
        DAV repos ModMimeUsePathInfo on
</Location>

DBDriver mysql DBDParams "host=localhost, port=5067, sock=/
    home/dlr/ajimushkay/mysql/5.0.67/mysqld.sock, user=root,
    dbname=repos"
```

DavDBMSTmpDir spezifiziert ein Temporäres Verzeichnis für Aktionen

DavDBMSFileDir spezifiziert ein Verzeichnis für Dateien, die nicht in der Datenbank gespeichert werden sollen

DavDMBSMaxFileSize legt die Grösse der Dateien fest, ab der Sie im Dateisystem und nicht mehr in der Datenbank gespeichert werden sollen

Location der Location Tag legt den Namespace des DAV Moduls fest. An dieser URL ist dann das DAV-Modul für Anfragen an den Webserver zuständig

DBDriver legt einige Einstellungen für die Datenbank fest.

- **mysql** spezifiziert MySQL als Datenbanktreiber
- **DBDParams** enthält Parameter für diesen Treiber
  - **host** der Host auf dem die Datenbank läuft, in diesem Fall der localhost (127.0.0.1)
  - **port** der Port auf dem die Datenbank auf Verbindungen wartet
  - **sock** der Ort der Unix-Domänen Socket zum verbinden mit der Datenbank

user der Benutzer mit dem der Treiber auf die Datenbank zugreifen soll. Da für die Testumgebung kein root-Passwort für die Datenbank eingegeben wurde, muss hier auch keins spezifiziert werden.

dbname der Datenbankname in dem das Repository verwaltet werden soll

Des weiteren wird in der Konfigurationsdatei des Apache der Port, auf dem er auf Verbindungen wartet, auf "2080" gestellt. Dies ist notwendig, damit Apache auch ohne Superuser Rechte gestartet werden kann. Alle Dienste die Ports unter 1000 verwenden, benötigen unter Linux diese Rechte.

4.1.6.2 Installation mit PostgreSQL

Die Installation mit PostgreSQL gestaltet sich ganz ähnlich der Installation mit MySQL. Ich werde hier allerdings nicht genauer darauf eingehen, da PostgreSQL im Moment eher stiefmütterlich vom Catacomb Projekt behandelt wird. Die mitgelieferten SQL-Statements für PostgrSQL erzeugen eine Reihe von Fehlern beim Ausführen. Die installierte Basis an Catacomb/Datafinder Installationen ist auch sehr MySQL orientiert, so dass dort zwar ein Bugbericht abgegeben wurde, aber sonst erst einmal auf weiteres Nachforschen verzichtet wurde.

4.1.7 Skripte

Um das Starten von Tests möglichst komfortabel zu halten, wurden diverse Skripte erstellt, die sich selbständig, je nach Parameter, um die entsprechende Verwaltung der Infrastruktur aus Apache httpd, mysql und catacomb kümmern.

Die Skipte und verwendete Bibliotheken werden im folgenden Abschnitt genauer erläutert.

4.1.7.1 OptionParser

Zum Parsen von Parametern wurde die OptionParser[8] Bibliothek genutzt, sie bietet hervorragende Möglichkeiten in diesem Bereich. Sie wird einfach benutzt, in dem

man eine Instanz von “Optionparser” erzeugt und diese dann mit der addoption() Methode mit Optionen befüllt. OptionParser erlaubt es dem Entwickler die Optionen in der üblichen GNU/POSIX Syntax zu deklarieren und erzeugt zusätzlich Hilfe Nachrichten anhand der eingegebenen Daten.

4.1.7.2 Parsen der Ausgaben

Hier kommt hauptsächlich der Vorteil von Python als Sprache für die Skripts zum Tragen. Die nötigen Werkzeuge zum Parsen und verarbeiten von Strings sind schon in der mitgelieferten Standard Bibliothek enthalten, diese und eine ganze Reihe von Methoden zum Lesen und Bearbeiten von Strings. Die Bibliothek “commands” erlaubt es Prozesse zu starten. Wenn ein Prozess mit der Methode “getoutput()” gestartet wird, wird die komplette Standard Ausgabe des Prozesses in einem String zurückgegeben.

Eingebaute Standard Methoden für Strings, wie “split()”, “find()” oder “replace()”, erlauben es dann die Ausgabe gezielt nach Informationen zu durchsuchen, die man später in einer ansprechenderen oder generell anderen Form präsentieren möchte.

Zusätzlich werden sämtliche Ausgaben in .keep benannten Dateien gespeichert, um ein späteres Debuggen einfacher zu gestalten, falls ein Schritt im Aufbau des Testlaufs oder beim Durchführen des Tests fehlschlägt.

4.1.7.3 ajimushkay.py

Mit diesem Skript wird ein Test angestartet. Folgende Parameter sind erlaubt und notwendig:

–apache hiermit wird die gewünschte Apache Version angegeben

–mysql hiermit wird die gewünschte MySQL Version angegeben

–catacomb und hiermit letztendlich die gewünschte Catacomb Version. Bei der Eingabe von “svn” als Version wird die aktuelle Version aus dem Subversion Repository ausgechecked

Das Skript startet nacheinander eine Reihe weiter Skripte, die sich letzendlich um folgende Aufgaben kümmern:

1. Das Deployment der Angegebenen Catacomb Version
2. Das Starten des MySQL Servers
3. Das Starten des Apache httpd
4. Das Ausführen des Litmus Tests
5. Das Ausführen des Prestan Tests
6. Das Herunterfahren des Apache httpd
7. Das Herunterfahren des MySQL Servers

4.1.7.4 deploycatacomb.py

Dieses Skript kümmert sich um das Deployment des Catacomb Moduls. Es benötigt als Parameter:

–apache hiermit wird die gewünschte Apache Version angegeben

–mysql hiermit wird die gewünschte MySQL Version angegeben

–catacomb und hiermit letztendlich die gewünschte Catacomb Version. Bei der Eingabe von "svn" als Version wird die aktuelle Version aus dem Subversion Repository ausgechecked

Diese Parameter werden normalerweise von dem Skript "ajimushkay.py" übergeben.

Es ist aber auch Möglich es einzeln anzustarten.

Dieses Skript führt sämtlich Aufgaben durch, die nötig sind um Catacomb lauffähig als Modul in Apache zu installieren. Das heißt:

1. falls die Angegebene Version "svn" entspricht, wird die aktuelle Version aus dem Repository ausgechecked
2. der bereits beschriebene configure Lauf für Catacomb wird für die entsprechenden Apache und MySQL Versionen gerecht durchgeführt
3. Catacomb wird kompiliert
4. Catacomb wird installiert
5. Die Datenbank wird wie erläutert vorbereitet

a) Die spezifizierte Datenbank wird gestartet

b) Falls vorhanden wird die Datenbank “repos” gelöscht und wieder neu angelegt

c) Die in der Catacomb-Distribution enthaltenen SQL-Statements werden ausgeführt

d) Die Datenbank wird wieder angehalten

6. Sämtliche Ausgaben von den einzelnen Schritten werden in Dateien geschrieben, um auch später genau eventuelle Fehler nachvollziehen zu können.

4.1.7.5 mysql_control.py

Dieses Skript stellt eine einfache Möglichkeit zur Verfügung eine spezifizierte MySQL Installation zu kontrollieren.

Die Parameter für mysql_control.py sind:

–version hier wird die MySQL Installation angegeben

–action hier wird die Aktion angegeben die durchgeführt werden soll. Mögliche Aktionen sind “start” und “stop”

4.1.7.6 apache_control.py

Dieses Skript stellt eine einfache Möglichkeit zur Verfügung eine spezifizierte Apache Installation zu kontrollieren.

Die Parameter für mysql_control.py sind:

–version hier wird die Apache Installation angegeben

–action hier wird die Aktion angegeben, die durchgeführt werden soll. Mögliche Aktionen sind “start” und “stop”

4.1.7.7 parse_litmus.py

Dieses Skript startet den Litmus Test und kümmert sich um die Verarbeitung der Ausgabe dieses Testutilities.

Dieses Skript benötigt keine Parameter, da beschlossen wurde Apache immer auf dem selben Port laufen zu lassen und auch Catacomb immer im selben Namespace Kontext laufen zu lassen. Das heißt konkret, wenn erst einmal eine Catacomb Version in der Testumgebung deployed wurde und der dazugehörige Server läuft, dass das Repository immer unter “http://localhost:2080/repos” erreichbar ist.

Hier Beispielhaft die Ausgabe der Grundtests von Litmus:

```
-> running 'basic':
 0. init ..................  pass
 1. begin .................  pass
 2. options ...............  pass
 3. put_get ...............  pass
 4. put_get_utf8_segment ..  pass
 5. mkcol_over_plain ......  pass
 6. delete ................  pass
 7. delete_null ...........  pass
 8. delete_fragment .......  pass
 9. mkcol .................  pass
10. mkcol_again ...........  pass
11. delete_coll ...........  pass
12. mkcol_no_parent .......  pass
13. mkcol_with_body .......  pass
14. finish ................  pass
<- summary for 'basic': of 15 tests run: 15 passed, 0 failed. 100.0%
```

Diese Ausgabe wird nun durch das Skript geparst. Ein Test kann 4 Zustände haben: “pass”, “FAIL”, “WARNING” und “SKIPPED”.

Die Ausgabe wird nun zur besseren Weiterverarbeitung folgenderweise aufgearbeitet:

1. Die Original Ausgabe wird eingelesen und in die Datei “litmus-output.keep”

geschrieben

2. Die 4 Zustände werden gezählt und zu jeweils einer Gesamtsumme addiert.
3. Die einzelnen Summen werden in Java Propertydatei konforme Dateien geschrieben, um sie später in Hudson besser verwenden zu können

Diese Property Dateien enthalten ein Parameter/Wert Paar in der Form: yvalue=ANZAHL.

4.1.7.8 parse_prestan.py

Dieses Skript startet den Prestan Test und kümmert sich um die Verarbeitung der Ausgabe dieses Testutilities.

Auch dieses Skript benötigt keine Parameter, da, wie bereits beschrieben, beschlossen wurde Apache immer auf dem selben Port laufen zu lassen und auch Catacomb immer im selben Namespace Kontext laufen zu lassen.

Hier ein Teil der Ausgabe von Prestan:

```
ProppatchMult ..... Rsp = 978 [us]
ProppatchSingle ..... Rsp = 810 [us]
PropfindDeadMult ..... Rsp = 1184 [us]
PropfindDeadSingle ..... Rsp = 1166 [us]
PropfindLiveMult ..... Rsp = 1618 [us]
PropfindLiveSingle ..... Rsp = 1567 [us]
```

Die einzelnen Subtests des Utilities werden aufgelistet und mit den ermittelten Responsezeiten versehen. Fehlgeschlage Tests ergeben in NaN (Not a Number) und können so auch geparst und gezählt werden.

Die Ausgabe von Prestan wird ebenfalls zur besseren Verarbeitung aufgearbeitet:

1. Die Original Ausgabe wird eingelesen und in die Datei “prestan-output.keep” geschrieben
2. Die 2 Möglichen Zustände werden gezählt und zu jeweils einer Gesamtsumme addiert

3. Die Gesamtzeit aller Responsezeiten wird ausgerechnet
4. Alle diese Summen werden auch wieder in Java Property Dateien abgespeichert, um sie in Hudson komfortabel weiterverwenden zu können

4.1.7.9 dav_query.py

Dieses Skript erlaubt es unterschiedliche Requests ähnlich dem davtool an einen WebDAV Server zu stellen. Primär wurde es für eine Suche über ein Repository geschrieben. Der Grund warum für diese Funktioalität nicht davtol, sondern ein eigenes Skript entwickelt wurde, ist die Möglichkeit in einem eigenen Skript mehr Kontrolle über den HTTP Header zu haben, so können gegebenenfalls zu debugging Zwecken auch falsche Header erzeugt werden.

Benötigte Parameter für davquery:

–server hier wird der Server angegeben, auf den der Request ausgeführt werden soll

–port zum Angeben des Ports des Servers auf dem der Catacomb läuft

–resource spezifiziert die Resource auf die zugegriffen werden soll

–infilespezifiziert die Datei mit dem Inhalt des Content-Headers, die eingelesn werden soll

–method gibt die WebDAV Methode an die verwendet werden soll. Zum Beispiel SEARCH

–login falls auf dem Server ACL aktiviert ist, kann hiermit der Loginname angegeben werden

–pwd hiermit wird das zugehörige Passwort zum Login angegeben

Ein Beispielhafter Aufruf für eine Suche würde so aussehen:

```
% ./dav_query.py -s localhost -p 2280 -r /repos/ -m SEARCH -i testsea
```

4.1.8 Hudson

Um nun die gesamten Testskripte automatisiert auszuführen wird wie beschrieben Hudson eingesetzt. Hudson ermöglicht das Überwachen des Catacomb Subversion

Repositories und erlaubt es im Falle einer Veränderung den Build und Test Prozess zu starten.

4.1.8.1 Konfiguration

Die Konfiguration von Hudson gestaltet sich dank der graphischen WebUI sehr einfach. Im folgenden Beispiel wird ein Test konfiguriert, der bei Veränderung der Sourcen im Repository einen Build der aktuellen Sourcen in der Virtuellen Maschine auslöst und das Ergebnis in Apache 2.2.8 testet.

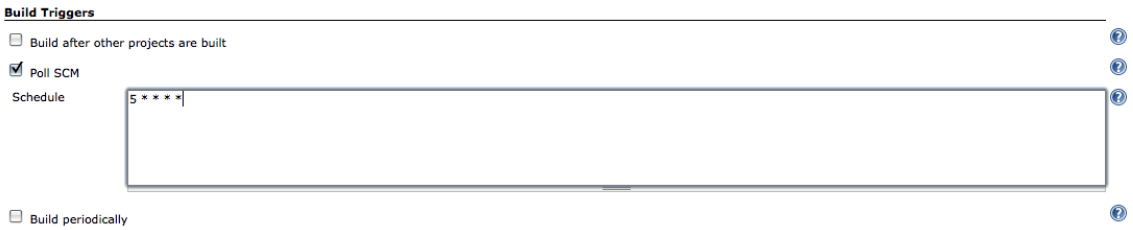
Dazu steuert man in seinem Browser die URL des Hudson Servers an und wählt dann im Hudson Dashboard die Option “New Job”. Dort gibt man als erstes einen Namen für diesen Job an. Danach kommen die wichtigen Konfigurationen.

Im Bereich Sourcecode Management wird “Subversion“ ausgewählt und die URL des gewünschten Repositories eingeben. Wenn man dies das erste Mal macht, kann es passieren, das man nach den Zugangsdaten gefragt wird. Im Falle des Catacomb Repositories handelt es sich um eine Login + Passwort Abfrage. Gibt man die Daten ein, kann Hudson ab diesem Zeitpunkt auf das Repository zugreifen.

Abb. 4.5: Hudson Konfiguration Source Code Management

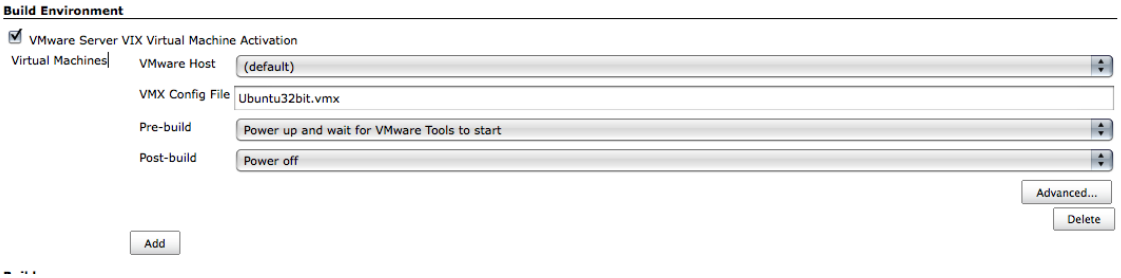
Als nächstes wird ein Build Trigger angegeben. Also ein Vorkommnis, das die hinterlegten Aktionen in Hudson anstößt. Da das Repostitory überwacht werden soll, wird hier “Poll SCM” gewählt. Hudson überprüft nun in festgelegten Abständen, ob sich die Sourcen verändert haben. In diesem Fall wurde der Poll Interval auf 5 Minuten gesetzt.

Abb. 4.6: Hudson Konfiguration Build Trigger



Nun muss noch eine Build Umgebung angegeben werden. Da der Build in einer Virtuellen Maschine ausgeführt werden soll, wird die Option “VMware Server VIX Virtual Machine Activation” ausgewählt. Hier kann nun der standardmäßig für die Hudson Installation angegebene VMware Sever/Host ausgewählt werden und es muss nur eingetragen werden, welche VM gestartet werden soll. Es können auch mehrere VM’s gestartet werden. Hier beispielhaft der Eintrag für die 32-Bit Maschine:

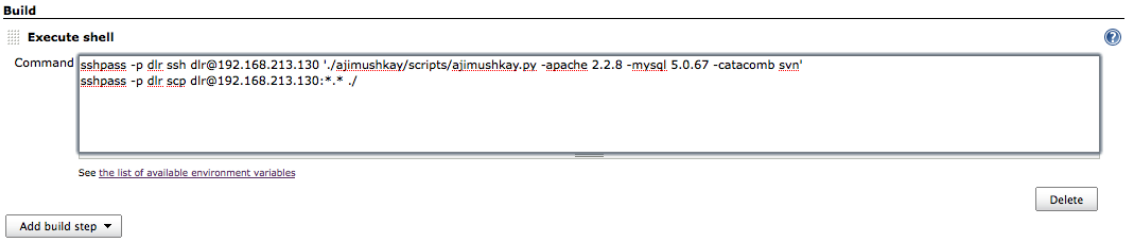
Abb. 4.7: Hudson Konfiguration Build Environment



Das Starten der VM wird so konfiguriert, dass gewartet wird, bis die VMware Tools in der VM geladen sind. Die VM ist also ab diesem Zeitpunkt einsatzbereit und man kann sich per ssh an der gestarteten Maschine anmelden.

Im Bereich “Build” der Konfiguration können nun die zu startenden Befehle eingegeben werden.

Abb. 4.8: Hudson Konfiguration Build



Es werden nur 2 Befehle benötigt. Der erste Befehl startet die Tests. Der 2te Befehl startet eine scp (Secure Copy) Verbindung und kopiert sämtliche erzeugten Dateien in den lokalen Workspace des Hudson Servers, damit sie archiviert werden können.

Die ssh-Verbindungen müssen mit “sshpass” gestartet werden, da es ssh aus Sicherheitsgründen nicht erlaubt das Passwort direkt auf der Kommandozeile zu übergeben. In diesem Fall kann aber auf Sicherheit verzichtet werden, da die Maschinen nur in einem lokalen Netz laufen und die Passwörter jedem Administrator der Zugriff auf diese Maschinen hat sowieso bekannt ist.

Wichtig ist nun noch in den “Post-build Actions” anzugeben, welche Dateien, die während des Testlaufs erzeugt wurden, archiviert werden sollen.

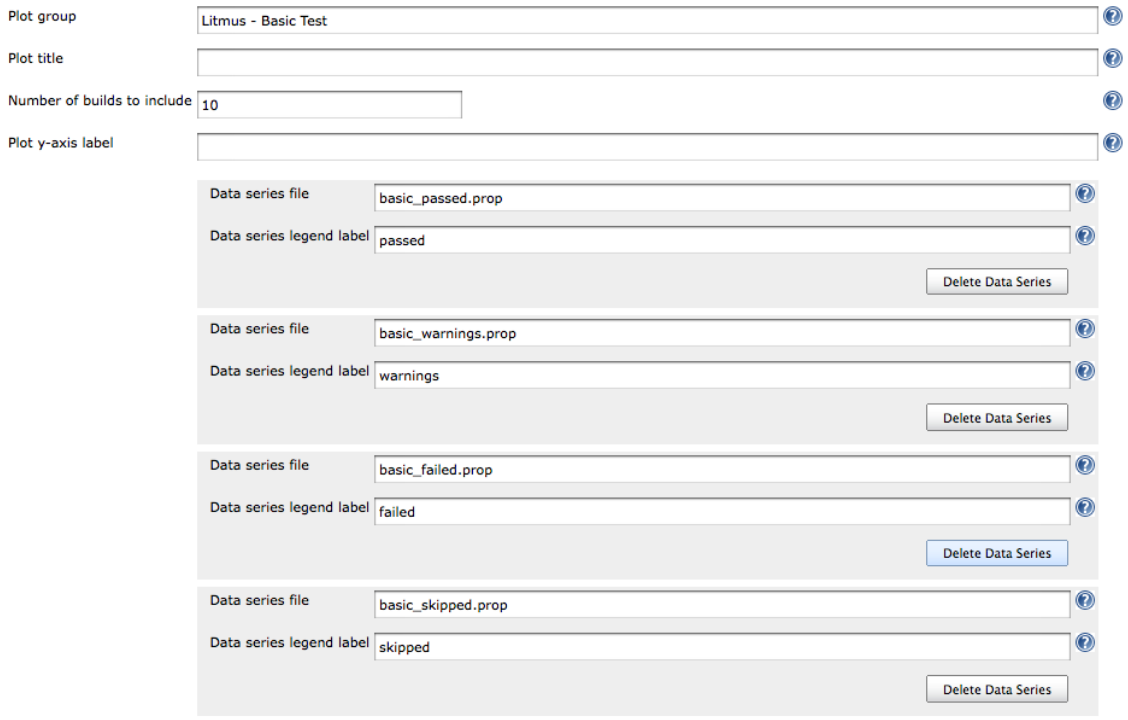
Abb. 4.9: Hudson Konfiguration Post-build Actions

Da durch die Skripte alle Ausgaben in “.keep” Dateien abgespeichert werden und die Java Property Dateien die Endung “.prop” erhalten haben, müssen auch nur diese Dateien archiviert werden.

4.1.8.2 Plots

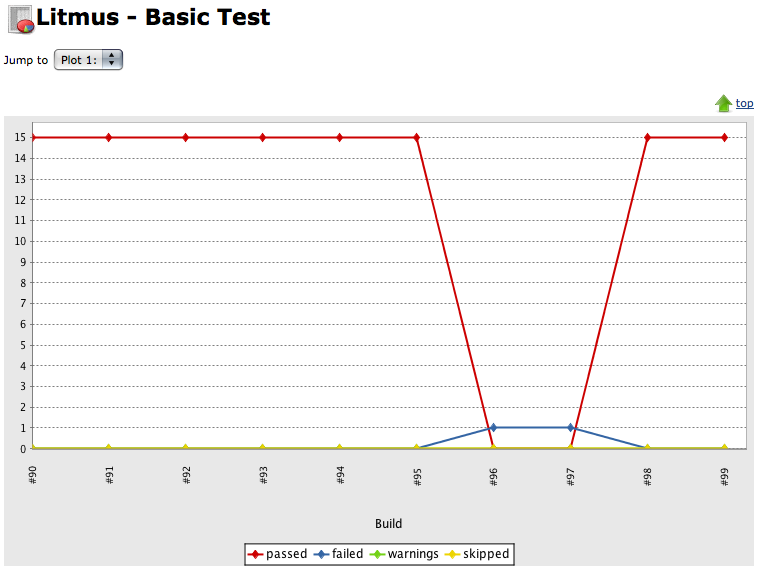
Zu diesen archivierten Dateien gehören auch die generierten Java Property Dateien der Skripte. Mit diesen Dateien und dem Plot Plugin von Hudson ist es nun möglich Werte über mehrere Builds bzw. Tests zu vergleichen. Als Beispiel hier die Konfiguration für das Plotten der Grundtests von Litmus.

Abb. 4.10: Hudson Konfiguration Plot Builddata



Es wird angegeben, dass für den Plot die letzten 10 Tests herangezogen werden sollen. Dann müssen nur noch pro gewünschtem Graphen die entsprechenden Java Property Dateien angegeben werden und eine Bezeichnung angegeben werden. Hier der beispielhafte Plot für die “Basic Tests” von Hudson über die letzten 10 Builds.

Abb. 4.11: Hudson Plot Litmus Basic



Insgesamt wurden 6 verschieden Plots erstellt. Einer für die Gesamtzahl der erfolgreichen, fehlgeschlagenen, übersprungen und mit Warnungen versehenen Tests und 4 für die jeweiligen Subtests von Litmus. Auch Prestan hat einen Plot über erfolgreiche und nicht erfolgreiche Subtest erhalten.

Kapitel 5

5 Resümee

5.1 Zusammenfassung

Mit der Kombination aus Hudson und Images von virtuellen Maschinen ist es nun sehr komfortabel möglich die Grundfunktionen des Apache Moduls Catacomb zu überprüfen und anhand der Plots sogar Trends zu erkennen. Durch das hauptsächliche Verwenden bereits vorhandener Tools ist der Wartungsaufwand für diese Testumgebung minimal, bzw. zu vernachlässigen. Litmus und Prestan sind beides gut ausgereifte Werkzeuge die zuverlässig ihren Dienst verrichten. Durch die Verwendung von Python sind die Skripte, die die Tests starten und Auswerten gut lesbar und ebenfalls gut wartbar.

Sämtliche Anforderungen an den Komfort der Testumgebung sind erfüllt. Der Entwickler hat keine zusätzliche Arbeit, um einen Test anzustoßen. Der Test wird wie erfordert automatisch nach dem einchecken des neuen Sourcecodes in das Subversion Repository gestartet, läuft durch und stellt dem Entwickler sämtliche Ausgaben der Build- und Test-Vorgänge einfach zur Verfügung. Durch die generierten Plots ist es zusätzlich möglich schnell Trends oder Einbrüche in der Funktionalität von Catacomb zu entdecken.

5.2 Ausblick

Für die Zukunft wäre es interessant zu den Blackbox Tests zusätzliche Unit Tests zu integrieren. Der Weg über Python und das Wrappen des C-Codes hat sich als noch nicht ausgereift genug erwiesen. Vielleicht ist die libctypes in naher Zukunft soweit, das man sie besser für diese Zwecke einsetzen kann oder es wird der schmerzhafte Weg über das implementieren der Unit Test direkt in C gewählt. Dies wäre e wünschenswerte Funktionalität für die Testumgebung.

Kapitel 6

6 Anhang

6.1 Grundlegende Semantik für DASL Suchen

```
<!-- "basicsearch" element -->
 <!ELEMENT basicsearch (select, from, where?, orderby?,
    limit?) >
<!-- "select" element -->
 <!ELEMENT select (allprop | prop) >
<!-- "from" element -->
 <!ELEMENT from (scope+) >
 <!ELEMENT scope (href, depth, include-versions?) >
 <!ELEMENT include-versions EMPTY >
<!-- "where" element -->
 <!ENTITY % comp_ops "eq | lt | gt| lte | gte">
 <!ENTITY % log_ops "and | or | not">
 <!ENTITY % special_ops "is-collection | is-defined |
    language-defined | language-matches">
 <!ENTITY % string_ops "like"> <!ENTITY % content_ops "
    contains">
 <!ENTITY % all_ops "%comp_ops; | %log_ops; | %special_ops;
    | %string_ops; | %content_ops;">
 <!ELEMENT where ( %all_ops; ) >
```

```
<!ELEMENT and ( %all_ops; )+ >
<!ELEMENT or ( %all_ops; )+ >
<!ELEMENT not ( %all_ops; ) >
<!ELEMENT lt (prop, (literal|typed-literal)) >
<!ATTLIST lt caseless (yes|no) #IMPLIED>
<!ELEMENT lte (prop, (literal|typed-literal)) >
<!ATTLIST lte caseless (yes|no) #IMPLIED>
<!ELEMENT gt (prop, (literal|typed-literal)) >
<!ATTLIST gt caseless (yes|no) #IMPLIED>
<!ELEMENT gte (prop, (literal|typed-literal)) >
<!ATTLIST gte caseless (yes|no) #IMPLIED>
<!ELEMENT eq (prop, (literal|typed-literal)) >
<!ATTLIST eq caseless (yes|no) #IMPLIED>
<!ELEMENT literal (#PCDATA)> <!ELEMENT typed-literal (#
  PCDATA)>
<!ATTLIST typed-literal xsi:type CDATA #IMPLIED>
<!ELEMENT is-collection EMPTY > <!ELEMENT is-defined (prop)
   >
<!ELEMENT language-defined (prop) > <!ELEMENT language-
  matches (prop, literal) >
<!ELEMENT like (prop, literal) > <!ATTLIST like caseless (
  yes|no) #IMPLIED>
<!ELEMENT contains (#PCDATA)>
<!-- "orderby" element -->
<!ELEMENT orderby (order+) >
<!ELEMENT order ((prop | score), (ascending | descending)?)
  >
<!ATTLIST order caseless (yes|no) #IMPLIED> <!ELEMENT
  ascending EMPTY> <!ELEMENT descending EMPTY>
<!-- "limit" element -->
<!ELEMENT limit (nresults) >
<!ELEMENT nresults (#PCDATA) >
```

Literaturverzeichnis

[1] *Apache License*. http://www.apache.org/licenses/LICENSE-2.0

[2] *BSD License*. http://www.opensource.org/licenses/bsd-license.php

[3] *CruiseControl*. http://cruisecontrol.sourceforge.net/

[4] *ctypes tutorial*. http://python.net/crew/theller/ctypes/tutorial.html

[5] *damagecontrol*. http://damagecontrol.codehaus.org/

[6] *Meet Hudson*. http://wiki.hudson-ci.org/display/HUDSON/Meet+Hudson

[7] *Python*. http://www.python.org/

[8] *Python OptionParser Bibliothek*. http://docs.python.org/library/optparse.html

[9] *Python String Methods*. http://docs.python.org/library/stdtypes.html#string-methods

[10] *RFC 2291*. http://tools.ietf.org/html/rfc2291

[11] *RFC 2396*. http://tools.ietf.org/html/rfc2396

[12] *RFC 2518*. http://tools.ietf.org/html/rfc2518

[13] *RFC 3253*. http://tools.ietf.org/html/rfc3253

[14] *RFC 3744*. http://tools.ietf.org/html/rfc3744

[15] *RFC 5323*. http://tools.ietf.org/html/rfc5323

[16] *A short overview of ISO/IEC 10646 and Unicode*. http://www.nada.kth.se/i18n/ucs/unicode-iso10646-oview.html

[17] *Ubuntu VM bei jars.de*. http://jars.de/english/ubuntu-804-vmware-image-download-english

[18] *Web Server Survey*. http://news.netcraft.com/archives/web_server_survey.html

[19] *What is Python? Executive Summary*. http://www.python.org/doc/essays/blurb/

[20] *IEEE standard glossary of software engineering terminology: approved September 28, 1990, IEEE Standards Board*. Revision and redesignation of IEEE Std 792-1983. New York, NY : Inst. of Electrical and Electronics Engineers, 1990. – ISBN 155937067X

[21] *Die GPL kommentiert und erklärt:*. 1. Aufl., dt. Orig.-Ausg. O'Reilly http://www.oreilly.de/german/freebooks/gplger. – ISBN 3897213893

[22] Balzert, H. : *Lehrbuch der Softwaretechnik: Softwaremanagement*. 2., Aufl. Spektrum Akademischer Verlag http://deposit.d-nb.de/cgi-bin/dokserv?id=3030756&prov=M&dok_var=1&dok_ext=htm. – ISBN 9783827411617 (Gb.)

[23] Booch, G. : *Object-oriented analysis and design: with applications*. 2. ed., 13. print. Reading, Mass. : Addison-Wesley, 1997. – ISBN 0805353402

[24] Martelli, A. : *Python cookbook:*. 2. ed. O'Reilly http://www.loc.gov/catdir/enhancements/fy0715/2005281191-d.html. – ISBN 0596007973 (Kt.)

[25] Myers, G. J. ; Badgett, T. ; Thomas, T. M. ; Sandler, C. : *The art of software testing*. 2nd ed. John Wiley and Sons http://www.netLibrary.com/urlapi.asp?action=summary&v=1&bookid=114566. – ISBN 047167835X (electronic bk.)