Eine Einführung in den Apache Webserver der Apache Software Foundation

Computerlabor KuZeB

09. März 2009

Inhaltsverzeichnis

1 Geschichte....3
2 Ein paar Zahlen....6
3 Technische Grundlagen....7
3.1 Wie arbeitet ein Webserver....7
3.2 HTTP....7
3.2.1 Anfragen....8
3.2.1.1 Argumente....9
3.2.1.1.1 Per GET....9
3.2.1.1.2 Per POST....9
3.2.1.2 Wie muss ich mir das nun vorstellen?....9
3.2.2 Antworten....10
3.2.2.1 Unterteilung Header / Body....11
4 Installation....12
5 Konfiguration....13
5.1 apache2.conf....13
5.2 Modulkonfigurationen....13
6 Host Konfiguration....15
6.1 Host Konfiguration im Detail....15
6.1.1 <VirtualHost *:80>....16
6.1.2 DocumentRoot /home/me/public_html/apache/public....16
6.1.3 ServerName me.localhost....16
6.1.4 <Directory /home/me/public_html/apache/public>....16
6.1.5 AllowOverride all....17
6.1.5.1 AuthConfig....17
6.1.5.2 FileInfo....17
6.1.5.3 Indexes....17
6.1.5.4 Limit....17
6.1.5.5 Options....18
7 Hallo Welt....19
8 PHP Modul....20
8.1 PHP Konfiguration....20
9 .htaccess....21
10 Ausblick....22
11 Quellen....25

1 Geschichte

Im Februar 1995 dominierte der „public domain HTTP daemon“ die Webserver-Welt. Dieser daemon Prozess wurde von Rob McCool an der Universität von Illinois am „National Center for Supercomputing Applications“ entwickelt.

Nachdem Rob die NCSA im Jahre 1994 verliess, wurde die Weiterentwicklung dieses HTTP Servers eingestellt. Viele Webmaster entwickelten daraufhin ihre eigenen Erweiterungen und Fehlerbehebungen, sie alle benötigten eine allgemeine Distribution des Webservers. Eine kleine Gruppe dieser Webmaster schloss sich über E-Mail Kontakt zusammen, um sich über die Koordination Ihrer Änderungen (in Form von „patches“) zu beraten. Brian Behlendorf und Cliff Skolnick formten eine Mailing Liste, einen gemeinsam nutzbaren Speicher und Logins für die Core-Entwickler auf einer Maschine in Kalifornien, unterstützt von HotWired. Bis Ende Februar gruppierten sich acht Leute, welche die originale Apache Gruppe bildeten:

Brian Behlendorf
David Robinson
Robert S. Thau
Roy T. Fielding
Cliff Skolnick
Andrew Wilson
Rob Hartill
Randy Terbush

Zu diesen acht Personen gesellten sich

Eric Hagberg
Frank Peters
Nicolas Pioch

als unabhängige Mitentwickler.
Sie alle nahmen den NCSA httpd 1.3 als Basis und addierten alle publizierten Fehlerbehebungen und Erweiterungen, um im April 1995 die erste öffentliche Version (0.6.2) des Apache Server zu veröffentlichen. Durch „Zufall“ begab es sich, dass die NCSA die Weiterentwicklung ihres httpd Servers zur selben Zeit wieder aufnahm, Brandon Long und Beth Frank des NCSA Server Entwicklungsteams stiessen zur Mailingliste der Apache Foundation hinzu, wodurch weitere Ideen und Fehlerbehebungen geteilt werden konnten.

Die erste Veröffentlichung des Apache Webservers war ein grosser Erfolg, doch die Entwickler waren sich bewusst, dass die Code-Basis ein umfassendes Re-Design benötigte. Zwischen Mai und Juni im Jahre 1995, entwickelte Robert Thau eine neue Server Architektur („Shambhala“), welche eine modulare Struktur und eine besser erweiterbare Programmier-Schnittstelle beinhaltete. Dieser neue Server bildete die Basis für die Veröffentlichung der Version Apache 0.8.8, welche im August 1995 herausgegeben wurde.

Weitere Entwicklungsarbeiten und die Portierung auf unterschiedlichste Plattformen führten schliesslich zur Veröffentlichung des Apache Webserver 1.0

am 01. Dezember 1995.

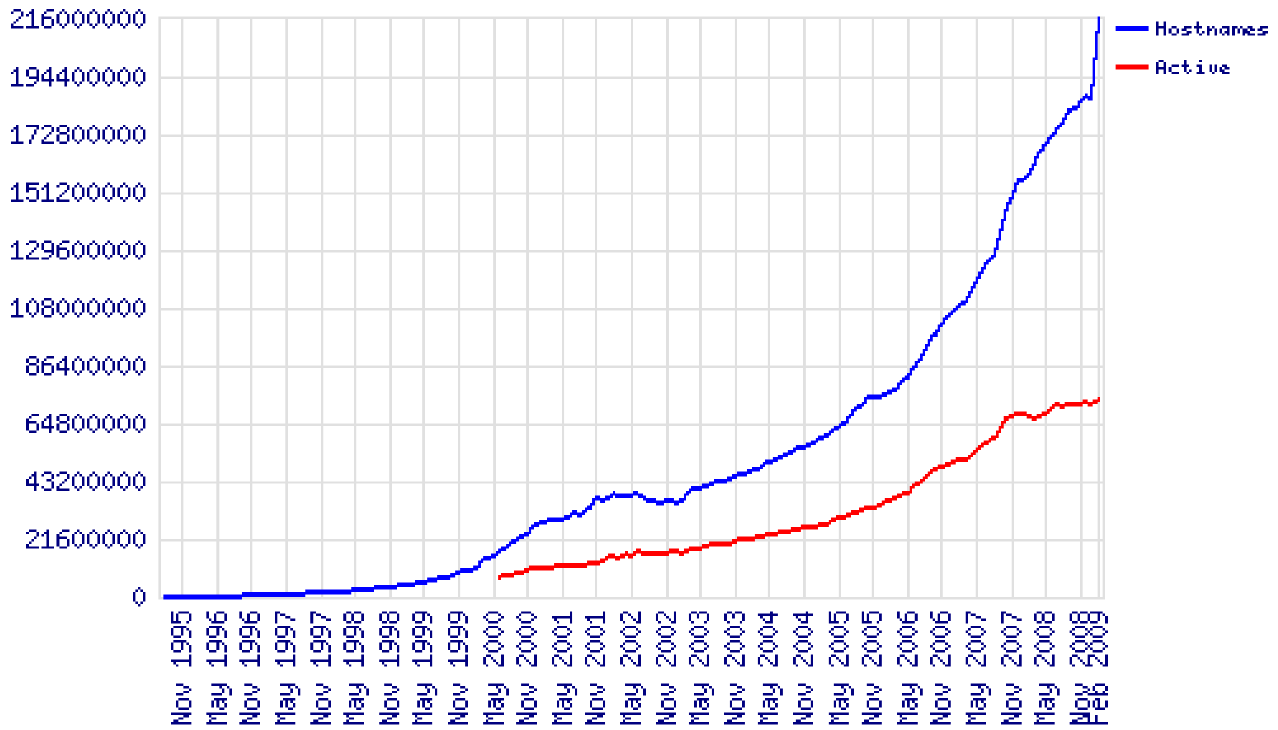
1999 formierte sich die Apache Gruppe zur Apache Software Foundation mit dem Ziel, organisatorische, rechtliche und finanzielle Hilfe für den Apache HTTP Server bereitzustellen. Diese Foundation bildete eine solide Grundlage für zukünftige Entwicklungen und expandierte schliesslich zu einer Foundation, welche die Zahl von Open Source Software Projekten in die Höhe schiessen liessen, längst ist es nicht mehr bloss der Apache Webserver, welcher von dieser Foundation unterstützt wird. Hier eine Liste von zusätzlichen Projekten (unter http://apache.org ersichtlich), welche von der Foundation unterstützt werden:

Abdera
ActiveMQ
Ant
APR
Archiva
Beehive
Buildr
Camel
Cayenne
Cocoon
Commons
Continuum
CouchDB
CXF
DB
Directory
Excalibur
Felix
Forrest
Geronimo
Gump
Hadoop
Harmony
HiveMind
HttpComponents
iBATIS
Incubator
Jackrabbit
Jakarta
James
Labs
Lenya
Logging
Lucene
Maven
Mina
MyFaces

ODE
OFBiz
OpenEJB
OpenJPA
Perl
POI
Portals
Qpid
Roller
Santuario
ServiceMix
Shale
SpamAssassin
STDCXX
Struts
Synapse
Tapestry
TCL
Tiles
Tomcat
Turbine
Tuscany
Velocity
Wicket
Web Services
Xalan
Xerces
XML
XMLBeans
XML Graphics
Attic

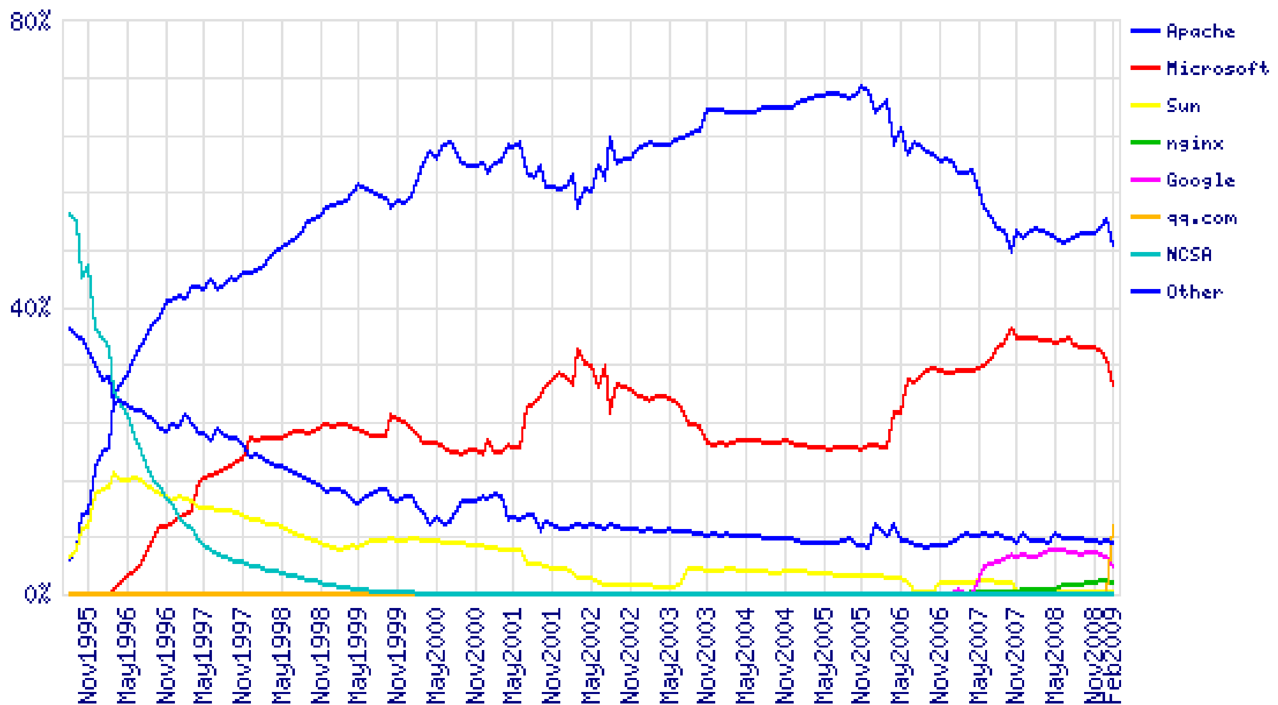
2 Ein paar Zahlen

Total Sites Across All Domains August 1995 - February 2009



Wir zählen insgesamt rund 216 Mio. Hostnamen, davon sind rund 70 Mio. Aktiv.

Market Share for Top Servers Across All Domains August 1995 - February 2009



3 Technische Grundlagen

3.1 Wie arbeitet ein Webserver

Ein Webserver, resp. ein Serverdienst ist ein Prozess, welcher im Hintergrund läuft. Einen Prozess lässt sich unter Linux auch mittels dem &-Operator in den Hintergrund verlegen. Beispiel:

$ firefox &

Wenn bei diesem Befehl das „Ampersand“ nicht eingegeben wird, so „blockiert“ der Prozess, in diesem Fall Firefox, die Konsole. Dies kann einfach nachvollzogen werden:

$ firefox

Wenn nun in der Konsole Ctrl+c gedrückt wird, so wird der Prozess beendet.

Das Ampersand hat den Zweck, den aufgerufenen Prozess vom Elternprozess (in diesem Falle die laufende Konsole) zu „lösen“.

Ein Serverdienst ist also ein Prozess, welcher im Hintergrund läuft, technisch gesehen wird ein sogenannter „Dämon“ Prozess gestartet. Ein „Dämon“ Prozess wird durch ein Ablösen vom Eltern-Prozesses erreicht.

Der Apache Webserver verfolgt nun ein konkretes Ziel: Er soll Anfragen über das HTTP Protokoll beantworten. Ein Browser setzt seine Anfrage über den Port 80 an den Webserver ab, insofern nicht eine SSL verschlüsselte Seite (Port 443) aufgerufen wird oder mit dem Doppelpunkt spezifisch ein anderer Port angezielt wird. Der Webserver soll daher standardmässig auf den Port 80 lauschen, möchte man zusätzlich SSL verschlüsselte Seiten anbieten, so sollte der Webserver daher auch den Port 443 „bedienen“.

Wir können uns den Startprozess des Webservers daher folgendermassen vorstellen:

1. Der Dämon Prozess wird gestartet.
2. Der Dämon Prozess „bindet“ sich an den Port 80 (evtl. Auch Port 443)
3. Jede eingehende Verbindung an die gebundenen Ports werden vom Dämon Prozess behandelt.

3.2 HTTP

Wie bereits angedeutet, ist HTTP eine Abkürzung für „Hypertext Transport Protocol“. Ein Protokoll, welches für den Austausch von Hypertext ausgelegt wurde. Durch die Erweiterung siner Anfragemethoden, Header-Informationen und Statuscodes ist HTTP jedoch nicht auf Hypertext beschränkt, zunehmend werden x-beliebige Daten über dieses Protokoll übermittelt.

Das Protokoll wurde mitsamt der URL und der HTML („Hypertext Markup

Language“) am CERN von Tim Berners-Lee im Jahre 1989 entwickelt, die Grundlagen für das World Wide Web sind damit entstanden.

HTTP wird über TCP abgewickelt, um eine zuverlässige Übermittlung zu gewährleisten. Dennoch ist HTTP ein „statusloses“ Protokoll, es wird nicht eine Verbindung zwischen Client und Server hergestellt, sondern der Client sendet eine Anfrage („Request“) an den Server, welcher mit einer Antwort („Response“) antwortet. Sobald der Client die Antwort erhalten hat, ist die Verbindung zwischen den beiden getrennt.

Schematisch lässt sich dies folgendermassen darstellen:

Ein client sendet eine Anfrage an den betroffenen Server, dieser antwortet und der Client stellt das Ergebnis entsprechend mithilfe eines Browsers dar. Die Kommunikationseinheiten werden daher verständlicherweise als Nachrichten bezeichnet.

3.2.1 Anfragen

Eine Anfrage, kann unterschiedliche Formen annehmen. So kann ein Client den Inhalt einer Webseite abfragen oder auch Daten senden (Formulardaten übertragen). Folgende Methoden wurden definiert:

- **GET** ist die gebräuchlichste Methode. Mit ihr werden Inhalte vom Server angefordert.
- **POST** ähnelt der GET-Methode, nur dass ein zusätzlicher Datenblock übermittelt wird. Dieser besteht üblicherweise aus Name-Wert-Paaren, die aus einem HTML-Formular stammen. Grundsätzlich können Daten auch mittels GET übertragen werden (als Argumente im URI), aber die Übertragung der Argumente erfolgt bei POST diskret (wichtig bei sensiblen Daten), und die zulässige Datenmenge ist deutlich größer.
- **HEAD** weist den Server an, die gleichen HTTP-Header wie bei einem GET oder POST, nicht jedoch den eigentlichen Dokumentinhalt selbst zu senden. So kann zum Beispiel schnell die Gültigkeit einer Datei im Browsercache geprüft werden.
- **PUT** dient dazu, Dateien unter Angabe des Ziel-URIs auf einen Webserver hochzuladen. Heute kaum noch implementiert (vergl. dazu WebDAV), war es in der Anfangszeit des WWW eine tatsächlich genutzte Möglichkeit.
- **DELETE** löscht die angegebene Datei auf dem Server. Dies ist heutzutage ebenso wie der PUT-Befehl kaum noch implementiert bzw. in der Standardkonfiguration aktueller Webserver abgeschaltet.

- **TRACE** liefert die Anfrage so zurück, wie der Server sie empfangen hat. So kann überprüft werden, ob und wie die Anfrage auf dem Weg zum Server verändert worden ist – sinnvoll für das Debugging von Verbindungen.
- **OPTIONS** liefert eine Liste der vom Server unterstützen Methoden und Features.
- **CONNECT** wird von Proxyservern implementiert, die in der Lage sind, SSL-Tunnel zur Verfügung zu stellen.

Erstaunlicherweise existieren eine Hülle und Fülle unterschiedlicher Anfragemethoden, jedoch werden in der Praxis meist bloss GET und POST implementiert. PUT und DELETE werden allerdings bei WebDAV Diensten verwendet (WebDAV implementiert noch zusätzliche Methoden wie PROPFIND, PROPPATCH etc.).

3.2.1.1 Argumente

3.2.1.1.1 Per GET

Eine Anfrage kann auch Argumente enthalten – Name / Werte Paare, um die Anfrage zu spezialisieren. Bei GET Anfragen werden diese Argumente in der URL mitgeliefert. Der URL wird ein Fragezeichen angehängt und die Werte durch ein Gleichheitszeichen dem jeweiligen Argument zugewiesen, wobei die Argumente durch ein Ampersand getrennt werden. Beispiel:

http://www.google.ch/search?hl=de&q=computerlabor&btnG=Google-Suche&meta=

Dies entspricht einer GET Anfrage an die Adresse *„http://www.google.ch/search“* mit den Parametern hl = „de“, q = „computerlabor“, btnG = „Google-Suche“ und meta als leeren Parameter:

btnG	= „Google-Suche“
hl	= „de“
meta	= „“
q	= „computerlabor“

3.2.1.1.2 Per POST

Die Daten, welche über einen POST Request mitgesandt werden, befinden sich nicht in der URL, sondern im Körper der Anfrage. Daher können auch grössere Datenmengen übertragen werden (Bilder etc.).

3.2.1.2 Wie muss ich mir das nun vorstellen?

Was heisst das nun jedoch konkret, resp. wie wird eine solche Anfrage abgesetzt?

Wir erinnern uns an das Schema Request und Response, das Antwortspiel. Wir können uns einen einfachen Nachrichtenaustausch vorstellen. Dies möchten

wir kurz näher betrachten:

```
$ sudo ngrep -d any port 80
GET /apache/ HTTP/1.1..Host: me.meski..User-Agent: Mozilla/5.0 (X11; U; Linux i686; de;
rv:1.9.0.6) Gecko/2009030407 Gentoo Firefox/3.0.6 FirePHP/0.2.4..Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8..Accept-Language: de-
de,de;q=0.8,en-us;q=0.5,en;q=0.3..Accept-Encoding: gzip,deflate..Accept- Charset: ISO-
8859-1,utf-8;q=0.7,*;q=0.7..Keep-Alive: 300..Connection: keep-alive...Pragma: no-
cache..Cache-Control: no-cache...
```

Wir erkennen bei einem einfachen Aufruf einer Webseite, dass es sich um eine GET Anfrage handelt. Diese Anfrage wird als Text an den Webserver übermittelt. Bei der näheren Betrachtung dieser Anfrage sehen wir einen weiteren Parameter, nämlich den Host. Weshalb spielt dieser Parameter nun eine wichtige Rolle?

Wir erinnern uns an den Aufbau des Internets: Jeder Knoten ist über eine eindeutige Adresse erreichbar. Ein Hostname wird über DNS Server auf eine entsprechende IP Adresse aufgelöst. Einer IP Adresse können nun jedoch mehrere Hostnamen zugeordnet werden, wäre dies nicht möglich, so würde jeder Hostname eine eigene IP Adresse benötigen, das wäre eine furchtbare Verschwendung von IP Adressen.

Ein Webserver muss daher erkennen können, für welchen Host er die Anfrage beantworten soll, weswegen dieser Teil der Anfrage so wichtig ist. Selbstverständlich ist auch eine Anfrage über eine IP Adresse möglich, diese wird schliesslich auf den „default“ Host angewandt, der Webserver antwortet in diesem Fall mit dem Host, welcher dem vorkonfigurierten Standard entspricht. Der Host würde in diesem Fall der IP Adresse entsprechen, wir sprechen dann von einem sogenannten IP basierten Host, resp. IP basierten virtual host im Gegensatz zu den namensbasierten virtual hosts.

3.2.2 Antworten

Die Anfrage ist jedoch bloss ein Teil der Kommunikation zwischen Client und Server. Wir gehen nochmals kurz zurück auf die Konsole und schauen uns die Antwort etwas genauer an:

```
$ sudo ngrep -d any port 80
...
T 127.0.0.1:80 -> 127.0.0.1:44350 [AP]
HTTP/1.1 200 OK..Date: Sun, 08 Mar 2009 21:25:19 GMT..Server: Apache..Last-Modified:
Tue, 24 Feb 2009 20:32:
43 GMT..ETag: "3a38b19-13d-463b0056d4412"..Accept-Ranges: bytes..Content-Length:
317..Keep-Alive: timeout=15
, max=100..Connection: Keep-Alive..Content-Type: text/html....<!DOCTYPE html PUBLIC
"-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
strict.dtd">.<html xmlns="http://www.w3.org/1999/xhtml">.
  <head>  .  <meta http-equiv="Content-Type" content="text/html; charset=utf-8" /> .
<title>Hallo Welt!</title>.</head>.<body>.  <p>Hallo Welt!</p>.</body>.</html>.
```

Wir erkennen, dass der Webserver mit der Protokollversion (HTTP/1.1) und einem Statuscode (200 OK) antwortet. Folgende „Familien“ an Statuscodes existieren, wobei die beiden x andeuten, dass noch weitere „Unter-Stati“ möglich sind:

- **1xx**
  Informationen
  Die Bearbeitung der Anfrage dauert trotz der Rückmeldung noch an. Eine solche Zwischenantwort ist manchmal notwendig, da viele Clients nach einer bestimmten Zeitspanne (Timeout) automatisch annehmen, dass ein Fehler bei der Übertragung oder Verarbeitung der Anfrage aufgetreten ist, und mit einer Fehlermeldung abbrechen.
- **2xx**
  Erfolgreiche Operation
  Die Anfrage wurde bearbeitet und die Antwort wird an den Anfragesteller zurückgesendet.
- **3xx**
  Umleitung
  Um eine erfolgreiche Bearbeitung der Anfrage sicherzustellen, sind weitere Schritte seitens des Clients erforderlich. Dies ist zum Beispiel der Fall, wenn eine Webseite vom Betreiber umgestaltet wurde, sodass sich eine gewünschte Datei nun an einem anderen Platz befindet. Mit der Antwort des Servers erfährt der Client im Location-Header, wo sich die Datei jetzt befindet.
- **4xx**
  Client-Fehler
  Bei der Bearbeitung der Anfrage ist ein Fehler aufgetreten, der im Verantwortungsbereich des Clients liegt. Ein 404 tritt beispielsweise ein, wenn ein Dokument angefragt wurde, das auf dem Server nicht existiert. Ein 403 weist den Client darauf hin, dass es ihm nicht erlaubt ist, das jeweilige Dokument abzurufen. Es kann sich zum Beispiel um ein vertrauliches oder nur per HTTPS zugängliches Dokument handeln.
- **5xx**
  Server-Fehler
  Es ist ein Fehler aufgetreten, dessen Ursache beim Server liegt. Zum Beispiel bedeutet 501, dass der Server nicht über die erforderlichen Funktionen (d.h. zum Beispiel Programme oder andere Dateien) verfügt, um die Anfrage zu bearbeiten.

3.2.2.1 Unterteilung Header / Body

Dass wir in unserem Browser keine der Informationen über die Protokollversion und den Statuscode sehen, macht durchaus Sinn, wenn wir eine Webseite betrachten, interessiert uns eigentlich herzlich wenig davon, wir sind schliesslich am Inhalt der Webseite interessiert. Die Interpretation einer

Antwort und einer Anfrage bedarf schliesslich einer Unterscheidung zwischen dem Kopf und dem Körper der Anfrage oder der Antwort. Was wir in unserem Browser sehen, entspricht bloss dem Körper der Antwort. Der Browser interpretiert schliesslich den Quelltext des Antwort-Körpers und stellt diesen dar.

Wir veranschaulichen dies anhand eines Hilfsmittels, einem Plugin für den Firefox, dem Firebug:

Wir sehen die verschiedenen Header, sowohl der Anfrage als auch der Antwort. Was wir zuvor mit ngrep betrachtet haben, stellt uns das Firebug Plugin lesbar dar, somit entspricht der Teil des Headers „Content-Type: text/html“ einem Name-Werte Paar, wobei Content-type dem Namen des betreffenden Headers und text/html dem Wert dazu entspricht.

Der Körper der Antwort entspricht schliesslich dem Quelltext der angefragten Webseite:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
  <title>Hallo Welt!</title>
</head>
<body>
  <p>Hallo Welt!</p>
</body>
</html>
```

Unser Browser stellt dies mit einem einfachen Absatz dar (p = paragraph = Absatz), in welchem „Hallo Welt“ zu lesen ist.

4 Installation

Die Installation eines Softwarepaketes ist unter Ubuntu wie gewohnt sehr einfach:

```
$ sudo aptitude install apache2
```

Dies installiert bereits alle wünschenswerten Pakete, welche wir für den Start mit dem Apache Webserver benötigen.

5 Konfiguration

Die grundsätzliche Konfiguration finden wir (wie alle anderen Konfigurationen des Systems) unter /etc/, resp. in diesem Falle unter /etc/apache2:

```
$ ls -lh /etc/apache2
-rw-r--r-- 1 root root 9.9K 2008-09-19 15:41 apache2.conf
drwxr-xr-x 2 root root 4.0K 2009-03-08 21:36 conf.d
-rw-r--r-- 1 root root  378 2008-09-19 15:41 envvars
-rw-r--r-- 1 root root    0 2009-03-08 21:36 httpd.conf
drwxr-xr-x 2 root root 4.0K 2009-03-08 21:36 mods-available
drwxr-xr-x 2 root root 4.0K 2009-03-08 21:37 mods-enabled
-rw-r--r-- 1 root root  351 2008-09-19 15:41 ports.conf
drwxr-xr-x 2 root root 4.0K 2009-03-08 21:36 sites-available
drwxr-xr-x 2 root root 4.0K 2009-03-08 21:36 sites-enabled
```

5.1 apache2.conf

In dieser Datei befindet sich die grundlegende Konfiguration des Webservers. Die Datei ist sehr gut dokumentiert, zu jeder Direktive finden wir eine Erklärung und eine Erläuterung. Die Datei wird bereits mit sinnvollen Werten belegt ausgeliefert, wir brauchen darin grundsätzlich keine Änderungen vorzunehmen. Dennoch möchten wir uns einige Zeilen näher betrachten:

```
ServerRoot "/etc/apache2"
```

Dieses Verzeichnis stellt den Ausgangsort für weitere Konfigurationsdirektiven dar. ServerRoot bezeichnet nicht wie vielleicht zu vermuten, das Verzeichnis unter welchem Dateien ausgeliefert werden! Wie wir bereits festgestellt haben, befinden sich in diesem Verzeichnis all unsere Konfigurationen, wir belassen dies daher so wie es ist.

5.2 Modulkonfigurationen

Der Apache Webserver lässt sich modular erweitern. Sprich, statt alle Fähigkeiten des Webservers fix einzukompilieren, kann der Webserver dynamisch Module laden. Diese Module wiederum bedürfen unter Umständen weiterer Konfigurationen. Wir möchten jedoch vermeiden, eine Konfigurationsdirektive zu setzen, welche nicht für ein bestimmtes Modul bestimmt ist, daher möchten wir solche Direktiven nur bedingt setzen. Die Konfiguration des Apache Webservers weist eine eigene Struktur auf, welche auf den ersten Blick an eine HTML oder XML Datei erinnert. Am Beispiel der Modulkonfiguration sieht dies folgendermassen aus:

```
<IfModule {Name des Moduls}>
  {Name der Direktive} {Wert für die Direktive}
</IfModule>
```

Für den Apache bedeutet dies nun, dass die entsprechenden Direktiven nur dann eingelesen werden, wenn das entsprechende Modul geladen ist.

Interessant sind in Bezug auf Module nun vor allem folgende Zeilen:

```
# Include module configuration:
Include /etc/apache2/mods-enabled/*.load
Include /etc/apache2/mods-enabled/*.conf
```

(include = einbinden)

Diese Direktiven bewegen den Apache dazu, zusätzliche Dateien in die Konfiguration miteinzubeziehen. Das ermöglicht es uns, Konfigurationen auszulagern, sodass nicht die gesamte Serverkonfiguration in einer einzigen Datei untergebracht werden muss.

Diese beiden Include Direktiven binden sämtliche Dateien mit den Endungen .load und .conf im Verzeichnis /etc/apache2/mods-enabled ein (* = Wildcard).

Wie die Endungen andeuten, stehen die Dateien mit den Endungen .load für das Einbinden, das Laden eines Modules, während die Dateien mit den Endungen .conf für die Konfiguration des entsprechenden Modules stehen. Diese Trennung, resp. dieses Layout ermöglicht eine sehr flexible Handhabung und übersichtliche Konfiguration... Richtig, übersichtlich! Man stelle sich vor, sämtliche Inhalte all dieser Dateien stehe in einer einzelnen Datei, sollte jemand nun auf die Idee kommen, ein zusätzliches Modul zu laden, so müsste dieser jemand mühsam in einer riesigen Datei nach dem entsprechenden Modul suchen und den Inhalt auskommentieren, bei Bedarf, das Modul wieder aus der Konfiguration zu nehmen, müsste dieses mühseligst wiederholt werden, diesmal müssten die betreffenden Zeilen wieder kommentiert werden.

Ubuntu vereinfacht daher die Handhabung der Module mittels zwei scripts:

- a2enmod
- a2dismod

Wenn wir nun den Webserver dazu bewegen möchten, ein Modul einzubinden, so tippen wir ganz einfach:

```
$ sudo a2enmod {Name des Moduls}
```

Möchten wir das Modul wieder beseite lassen, so genügt eine einzelne Befehlszeile:

```
$ sudo a2dismod {Name des Moduls}
```

Im Hintergrund geschieht nichts anderes, als dass symlinks von den entsprechenden Modulkonfigurationen aus dem Verzeichnis

/etc/apache2/mods-available nach /etc/apache2/mods-enabled gesetzt resp. gelöscht werden. Betrachten wir den Inhalt des Verzeichnisses /etc/apache2/mods-enabled, sehen wir dies:

```
$  ls -lh /etc/apache2/mods-enabled/
insgesamt 0
lrwxrwxrwx 1 root root 28 2009-03-08 21:36 alias.conf -> ../mods-available/alias.conf
lrwxrwxrwx 1 root root 28 2009-03-08 21:36 alias.load -> ../mods-available/alias.load
...
```

6 Host Konfiguration

```
# Include the virtual host configurations:
Include /etc/apache2/sites-enabled/
```

Betrachten wir als erstes den Inhalt dieses Verzeichnisses:

```
$ ls -lh /etc/apache2/sites-enabled/
insgesamt 0
lrwxrwxrwx 1 root root 26 2009-03-08 21:36 000-default -> ../sites-available/default
```

Auch hier sehen wir, dass das Verzeichnis lediglich symlinks enthält. Dieser bereits vorhandene symlink bedeutet folglich, dass der virtuelle Host mit dem Namen „default“ aktiv ist. Dieser Host entspricht nun dem Standard Host, welcher über den Hostnamen der Maschine und auch über dessen IP Adresse erreichbar ist.

6.1 Host Konfiguration im Detail

Springen wir ins kalte Wasser und erstellen uns einen eigenen Host. Hierzu erstellen wir eine entsprechende Konfiguration und platzieren den Inhalt in das Verzeichnis der verfügbaren Hosts, also in das Verzeichnis /etc/apache2/sites-available. Der Inhalt dieser Datei sieht folgendermassen aus (ersetze einfach me mit Deinem Benutzernamen):

```
<VirtualHost *:80>
  DocumentRoot /home/me/public_html/apache/public
  ServerName me.localhost
  <Directory /home/me/public_html/apache/public>
    AllowOverride all
    Order allow,deny
    Allow from all
  </Directory>
</VirtualHost>
```

Dies ist eine minimalistische Konfiguration für einen virtuellen Host, welche fürs erste jedoch vollkommen ausreicht. Wie wir bereits in der Datei /etc/apache2/apache2.conf gesehen haben, wird die Konfiguration des Webservers in einer etwas eigenen Form vorgenommen.

6.1.1 <VirtualHost *:80>

Die erste Zeile besagt, dass eine Konfiguration für einen virtuellen Host vorgenommen wird. Jede Direktive startet mit einem öffnenden „tag“, also einem HTML ähnlichen tag. Jede Direktive, welche mit einem startenden tag begonnen wird, muss zwingend mit einem schliessenden Tag beendet werden. In diesem Fall finden wir den schliessenden Tag zuunterst (</VirtualHost>). Ein schliessender Tag beendet eine geöffnete Direktive, indem der Name der Direktive mit einem führenden slash wiederholt wird. Ganz wie in HTML. Die Wildcard (*) im öffnenden Tag (dem Beginn des „containers“ VirtualHost) besagt, dass sämtliche Anfragen an den Port 80 aller lokalen IP Adressen berücksichtigt werden. Statt einer Wildcard könnten wir auch die IP Adresse der Maschine eintragen. Wir könnten auch einen anderen Port wählen, müssten dann jedoch den Port zusätzlich in die Datei /etc/apache2/ports.conf eintragen, sodass der Server auch an diesem Port auf Anfragen lauscht. Der Port 80 entspricht jedoch dem Standardport für HTTP Anfragen, wir belassen dies daher dabei.

6.1.2 DocumentRoot /home/me/public_html/apache/public

Ganz wichtig nun die zweite Direktive: „DocumentRoot“. Diese Direktive besagt, wo auf dem Dateisystem die „Webseite“ zu finden ist. Sprich, alle vom Webserver auszuliefernden Dateien müssen sich in diesem Verzeichnis oder in einem untergeordneten Verzeichnis des document roots befinden. Eine Datei, welche nun irgendwo unter /home/me/public_html/blah liegt, wird durch den Webserver in diesem Fall nicht gefunden, resp. nicht der Öffentlichkeit preisgegeben.

6.1.3 ServerName me.localhost

Der Server Name bedeutet nun für den Apache Webserver, dass diese Webseite dem Host me.localhost zugeordnet wird. Wir erinnern uns an den Request Header „Host: ...“, der Server Name entspricht exakt diesem Hostnamen und muss der Anfrage entsprechen. Wenn wir nun die IP Adresse des Webservers anfragen, so landen wir nie auf dieser Webseite, stattdessen würde der „default host“ antworten, resp. wir würden im Dateisystem unseres Webservers unter /var/www landen, da dieser dem „default host“ entspricht.

6.1.4 <Directory /home/me/public_html/apache/public>

Die Directory Direktive weist den Apache Webserver an, spezifische Einstellungen für das angegebene Verzeichnis gelten zu lassen. Auch dieser container wird am Schluss mit dem schliessenden tag </Directory> abgeschlossen.

6.1.5 AllowOverride all

Das AllowOverride besagt, dass der Betreiber dieser Seite (in diesem Falle me, da das document root im Verzeichnis des Benutzers me liegt) mit eigenen Konfigurationen gewisse Einstellungen selbst vornehmen kann. Hierbei stossen wir auf ein weiteres Konfigurationskonzept des Apache Webservers, denn diese Direktive besagt, dass der Benutzer .htaccess Dateien verwenden kann, um das Verhalten des Webservers selbst beeinflussen zu können. Statt alle (all) möglichen .htaccess Direktiven zu erlauben, können auch spezifische Eigenschaften genannt werden, welche der Betreiber „überschreiben“ darf. Dies kann einer oder mehreren der folgenden Optionen entsprechen, welche direkt in einer .htaccess Datei innerhalb des document roots gesetzt werden können:

6.1.5.1 AuthConfig

Erlaubt es dem Betreiber, eine Authorisation zu erwirken (AuthName, AuthType, AuthUserFile, Require etc.)

6.1.5.2 FileInfo

Erlaubt es dem Betreiber, spezifische Dokument-Typen zu kontrollieren (ErrorDocument etc.)

6.1.5.3 Indexes

Erlaubt es dem Betreiber, Verzeichnis-Indexierung zu kontrollieren (DirectoryIndex etc.)

6.1.5.4 Limit

Durch diese Option kann der Betreiber den Zugriff beschränken (Allow, Deny und Order).
Beispiel:

```
Order deny,allow
Deny from all
Allow from me.localhost
```

Dies würde ein Zugriff grundsätzlich verbieten, resp. erst werden alle deny Direktiven ausgewertet, hernach die allow Direktiven. Hierbei gilt darauf zu achten, dass der standardmässige Zugriff der zweiten Direktive entspricht, sollte also deny,allow stehen, so ist der standardmässige Zugriff erlaubt. Im Gegensatz dazu verbietet ein allow,deny standardmässig den Zugriff, wenn nicht mindestens eine allow Direktive den Zugriff explizit erlaubt. Betrachten wir folgendes Beispiel:

```
Order allow,deny
Allow from me.localhost
Deny from blah.me.localhost
```

In diesem Beispiel haben lediglich und ausschliesslich Zugriffe von me.localhost Zugriff, sämtliche anderen Zugriffe werden untersagt, da der standardmässige Zugriff verboten ist.

6.1.5.5 Options

Dies erlaubt es dem Betreiber, spezifische Verzeichnis-Optionen zu setzen (Vergleiche <Directory ...>). Diese Optionen können durch mehrfaches Vorkommen derselben <Directory ...> Sektion überschrieben werden. Daher gibt es zusätzlich die Möglichkeit, explizit eine Option zu setzen, ohne die anderen Optionen zu überschreiben.

Beispiel:

```
<Directory /home/me/public_html/apache/public>
  Options Indexes FollowSymLinks
</Directory>
<Directory /home/me/public_html/apache/public>
  Options Includes
</Directory>
```

In diesem Fall wäre lediglich die Option Includes gesetzt, da die Einstellungen des Verzeichnisses durch die zweite Sektion überschrieben wurde. Um dies zu verhindern, kann mittels einem + oder einem – explizit eine Option hinzugefügt oder entfernt werden:

```
<Directory /home/me/public_html/apache/public>
  Options Indexes FollowSymLinks
</Directory>
<Directory /home/me/public_html/apache/public>
  Options +Includes -Indexes
</Directory>
```

Dies resultiert in den gesetzten Optionen „FollowSymLinks“ und „Includes“. In einer .htaccess Datei würde allerdings das Directory tag entfallen, was im übrigen für alle Direktiven in einer .htaccess Datei gilt (da unmissverständlich klar ist, dass die Direktiven bereits für das entsprechende Verzeichnis gelten, in welchem sich die .htaccess Datei befindet).

7 Hallo Welt

Wir möchten nun endlich unsere Hallo Welt Seite in Betrieb nehmen, hierfür müssen wir allerdings berücksichtigen, dass wir einen host gewählt haben, welcher noch nirgends registriert ist: me.localhost – resp. wir haben den Webserver angewiesen, auf diesen hostnamen zu reagieren, doch unser DNS resolver muss erst mal wissen, wo denn dieser host zu finden ist. Für unseren Fall, in welchem wir die Webseite auf dem lokalen Rechner betreiben und diesen über den lokalen Rechner betrachten, reicht es, den hostnamen in der Datei /etc/hosts unterzubringen:

/etc/hosts:

```
127.0.0.1  me.localhost me
```

Damit weisen wir unseren Rechner an, bei einem Aufruf des hosts me.localhost auf das „Loopback“ device, den lokalen host zuzugreifen.

Als nächstes benötigen wir unsere Webseite, resp. natürlich muss das angegebene Verzeichnis erst mal existieren:

```
~ $ mkdir -p public_html/apache/public
```

Nun existiert wohl das Verzeichnis, doch ohne Inhalt hat der Webserver herzlich wenig zu präsentieren. Wir erstellen daher unsere erste Hallo Welt Webseite und platzieren ein einfaches HTML Dokument in das document root. Wir erhalten schliesslich folgende Struktur:

/home/me/public_html/apache/public/index.html

Nachdem wir sichergestellt haben, dass das Verzeichnis, welches wir für das document root konfiguriert haben existiert und darin eine HTML Seite existiert, weisen wir den Webserver an, seine Konfiguration neu einzulesen:

```
$ sudo /etc/init.d/apache2 reload
```

Wir können nun mit unserem Browser unsere Hallo Welt Seite betrachten, indem wir folgende URL aufrufen:

http://me.localhost/

Der Grund, weshalb wir nicht index.html an die Adresse anhängen müssen, liegt darin, dass die Datei mit dem Namen index.html der Standard-Index Seite entspricht. Dies sehen wir in unserer Serverkonfiguration, genau genommen in der Konfiguration des geladenen Moduls, welches hierfür zuständig ist:

```
$ cat /etc/apache2/mods-enabled/dir.conf
<IfModule mod_dir.c>
        DirectoryIndex index.html index.cgi index.pl index.php index.xhtml index.htm
</IfModule>
```

Wir sehen, dass etwas weiteres vorbereitet ist: index.php...

8 PHP Modul

Wir möchten als nächstes, dass unser Webserver die Skriptsprache PHP interpretiert. Wie gewohnt, ist die Installation ein Klacks:

```
$ sudo aptitude install php5
```

Dies installiert uns allerdings bloss das PHP Modul für den Webserver, wenn wir zusätzlich eine Kommandozeilenversion für PHP möchten, so können wir dies mittels

```
$ sudo aptitude install php5-cli
```

installieren – dadurch können wir PHP Skripte von der Konsole aus ausführen.

Dass unser Webserver ein neues Modul erhalten hat und dieses auch gleich automatisch aktiviert hat, sehen wir in unserem Konfigurationsverzeichnis:

```
$ ls -lh /etc/apache2/mods-enabled/php5*
rwxrwxrwx 1 root root 27 2009-03-09 11:09 /etc/apache2/mods-enabled/php5.conf -> ../
mods-available/php5.conf
lrwxrwxrwx 1 root root 27 2009-03-09 11:09 /etc/apache2/mods-enabled/php5.load -> ../
mods-available/php5.load
```

Testen wir dies doch gleich einmal:

```
$ sudo /etc/init.d/apache2 restart
$ cd public_html/apache/public
$ mv index.html hallo.html
$ vim index.php
<?php
phpinfo();
```

8.1 PHP Konfiguration

Die PHP Konfiguration erfolgt in einer Datei namens php.ini, welche in Ubuntu unter /etc/apache2/php5/apache2/php.ini für das Apache Modul zu finden ist.

Eine .ini Datei weist eine spezifische Struktur auf. Ein Kommentar beginnt nicht wie gewohnt mit einem #, sondern wird durch ein Semikolon eingeleitet.

Wir möchten bloss auf einige Punkte eingehen:

- short_open_tag
  Ermöglicht es, in PHP Dateien (insbesondere interessant für .phtml Dateien), eine Kurzschreibweise zu verwenden: Statt <?php reicht ein einfaches <?. Sehr interessant: <?= entspricht einem <?php echo
- memory_limit = 16M ; Maximum amount of memory a script may consume (16MB)
  Sollte grundsätzlich erhöht werden (viele Pakete funktionieren nicht mit dieser Beschränkung). Ein guter Startwert wäre 32MB

- register_globals = Off
  NIEMALS auf On stellen! Diese Option wird spätestens ab PHP 5.3 abgeschafft.
- post_max_size = 8M
  Sollte erhöht werden, wenn es möglich sein sollte, insgesamt mehr als 8MB zu senden.
- upload_max_filesize = 2M
  Gibt die maximale Grösse einer einzelnen gesendeten Datei an (ein Bild mit einer Grösse von 4MB würde nicht akzeptiert).
  (-> post_max_size sollte grösser sein als upload_max_filesize)
- allow_url_fopen = On
  Eine nette Sache, dies erlaubt es PHP scripts, Dateien übers Web wie lokale Dateien zu öffnen (Bsp: file_get_contents('http://kire.ch/linux/computerlabor.htm');)
  Hierbei sollte allerdings beachtet werden, dass dies unter Verwendung von „unsicheren“ scripten zu einem Sicherheitsrisiko werden kann. Grundsätzlich jedoch empfehlenswert.
- allow_url_include = Off
  Ähnlich wie allow_url_fopen, gilt jedoch für Aufrufe von include() und require() (resp. include_once() und require_once()).
  Hiervon ist eher abzusehen, das Einbinden einer Datei auf diese Weise sollte nicht über URLs geschehen (erhebliches Sicherheitsrisko).

9 .htaccess

Die .htaccess Dateien spielen eine wichtige Rolle, in welcher der Betreiber eine gewisse Kontrolle über das Verhalten des Webservers für seinen eigenen host erhält. So wäre es beispielsweise möglich, sein Verzeichnis vor ungewolltem Zugriff zu schützen, beispielsweise mit einem Passwort Schutz. Wir möchten nun einen geschützten Bereich erstellen, in welchen bloss mit einem gültigen Benutzernamen – Passwort Paar zugegriffen werden kann. Hierzu muss natürlich der Webserver erst mal wissen, wer mit welchem Passwort Zugriff erhält, dies wird mittels einer spezifischen Passwortdatei erreicht. Die Apache Software bringt ein Werkzeug mit, solche Dateien anzulegen und zu verändern: htpasswd. Betrachten wir uns kurz die Optionen, welche dieses Werkzeug mit sich bringt:

```
$ htpasswd –help
Usage:
        htpasswd [-cmdpsD] passwordfile username
        htpasswd -b[cmdpsD] passwordfile username password
        htpasswd -n[mdps] username
        htpasswd -nb[mdps] username password
 -c  Create a new file.
 -n  Don't update file; display results on stdout.
 -m  Force MD5 encryption of the password.
 -d  Force CRYPT encryption of the password (default).
```

```
 -p  Do not encrypt the password (plaintext).
 -s  Force SHA encryption of the password.
 -b  Use the password from the command line rather than prompting for it.
 -D  Delete the specified user.
On Windows, NetWare and TPF systems the '-m' flag is used by default.
On all other systems, the '-p' flag will probably not work.
```

Die wichtigsten Optionen sind sicherlich -c um eine neue Datei zu erstellen, -D um einen vorhandenen Benutzer zu entfernen und keine Option, um einen neuen Benutzer hinzuzufügen. Spielen wir doch mal damit:

```
$ mkdir htpasswd
$ cd htpasswd
$ htpasswd -c .htpasswd me
New password:
Re-type new password:
Adding password for user me
$ cat .htpasswd
me:3juHPcz9nZBTQ
```

Möchten wir einen neuen Benutzer hinzufügen:

```
$ htpasswd .htpasswd her
New password:
Re-type new password:
Adding password for user her
$ cat .htpasswd
me:3juHPcz9nZBTQ
her:6UTv6vJA.PEug
```

Damit diese Datei nun Verwendung findet, resp. damit wir damit ein Verzeichnis vor ungewolltem Zugriff schützen können, müssen wir dem Webserver mitteilen, welches Verzeichnis er mit welcher Passwort-Datei schützen soll:

```
$ cd public_html/apache/public
$ vim .htaccess
AuthType Basic
AuthName "Melde Dich erst mo an!"
AuthUserFile /home/me/htpasswd/.htpasswd
Require valid-user
```

Ein erneuter Aufruf unserer Webseite gelingt uns nun bloss noch über eine Authentifikation.

10 Ausblick

Durch das Verwenden des PHP Modules wird ein PHP script vom Benutzer des Webservers ausgeführt (www-data). Wenn mehrere Webseiten von unterschiedlichen Benutzern auf einem Server betrieben werden, so besteht die Notwendigkeit, dass jeder Benutzer auf die Dateien aller anderer Benutzer Zugriff erhält, da der www-data Benutzer auf sämtliche Dateien Zugriff haben

muss. Wenn nun der www-data Benutzer Zugriff auf sämtliche Dateien erhält, erhält folglich auch jeder andere Benutzer Zugriff auf diese Dateien.

Man stelle sich vor, man betreibt eine Webseite, welche einen Zugriff auf eine Datenbank benötigt. Die Zugangsdaten auf die Datenbank sind folglich in einer Datei (einer Konfigurationsdatei) untergebracht. Wenn nun jeder Zugriff auf diese Datei erhält, so ist es auch jedem anderen Benutzer möglich, diese Angaben auszulesen und erhält folglich auch Zugriff auf die Datenbank. Autsch!

Eine Möglichkeit dies zu umgehen ist die Verwendung von PHP als CGI Komponente (CGI = Common Gateway Interface). CGI dient als Schnittstelle zwischen dem Webserver und einem separaten Prozess, welcher das PHP (oder eine beliebig andere Skriptsprache) Skript ausführt. Der Prozess, welcher das Skript ausführt, kann schliesslich unter dem Benutzer betrieben werden, welcher die Webseite betreibt. So lässt sich jede „heikle" Datei vor unliebsamen Zugriff schützen ($ chmod 600 db_access.ini), da nicht mehr der Webserver, sondern der ausführende Prozess, welcher schliesslich unter dem Benutzer des Webseitenbetreibers ausgeführt wird, Zugriff auf diese Datei benötigt.

Ein Nachteil der Verwendung eines CGI betriebenen PHP Webserver ist jedoch die schlechte Performanz, da zusätzlich ein eigener Prozess gestartet werden muss.

Dieser Nachteil kann umgangen werden, wenn der Webserver mittels mod_fastcgid betrieben wird, hierbei wird beim ersten Start des Skriptes ein Prozess im Hintergrund gestartet, welcher eine gewisse Zeit aktiv bleibt, also nicht beim Beenden des Skriptes beendet wird.

Damit jedoch ein solcher CGI Prozess unter dem Benutzer des Webseitenbetreibers betrieben wird, wird zusätzlich suexec benötigt. Suexec ist Bestandteil des Apache Webserver, welcher dazu dient, den ausführenden Prozess unter einem anderen Benutzer auszuführen. Dieses Module birgt jedoch von Hause aus starke Sicherheitsvorkehrungen, allen voran:

- Das Dokumentenverzeichnis ist standardmässig unter /var/www zu finden. Jeder Host, welcher nicht unter diesem Verzeichnis liegt, wird erfolglos mit suexec versuchen zu kooperieren.
- Die auszuführenden Skripte dürfen nicht world-writeable sein.
- Wird eine der Bedingungen nicht erfüllt, weigert sich suexec, das Skript auszuführen.

Gerade die Notwendigkeit, dass das Wurzelverzeichnis unter /var/www zu liegen hat, widerspricht dem grundsätzlichen Layout des Linux Dateisystem – die Benutzer finden Ihre Dateien unter /home. Meist drängt dies zur Notwendigkeit, den Apache mitsamt dem suexec binary selbst zu kompilieren, sodass diese Einstellung angepasst werden kann (die Einstellung wird fix in die Software einkompiliert).

Im Netz existieren zahlreiche Anleitungen, wie sowas zu erreichen ist.

Grundsätzlich gilt jedoch: Der Betrieb mittels suexec und mod_fastcgid ist einzig und allein für Server interessant, auf welchen mehrere hosts von unterschiedlichen Benutzern betrieben werden. Für lokale Maschinen, Entwicklungsmaschinen, Server welche bloss eine einzelne grosse Webseite bedienen etc. ist dies nicht interessant, da mod_php weder den overhead der Ausführung noch die aufwändigere Konfiguration und Inbetriebnahme mit sich bringt.

11 Quellen

1. Apache.org (http://www.apache.org)
2. Wikipedia (http://de.wikipedia.org)
3. Netcraft (http://news.netcraft.com/)