

Perl Database Interface (DBI)

Handout

Revision : 1.1 (14. April 2004)

Thorsten Huber, Hendrik Brückner

Informationstechnik
Berufsakademie Stuttgart

```

    $_=eval
    al("seek\040D
    0");foreach(1..2)
    {<DATA>};my
    @camel1hump;my$camel;
    my$Camel ,while(
    <DATA>){$_=sprintf("%-6
    9s" ,$_);my@dromedary-----1=split(/;/);if(defined($
    _=<DATA>)){@camel1hump p=split(/;/)}while(@dromeda
    ry1){my$camel1hump=0 ;my$CAMEL=3;if(defined($_=shif
    t(@dromedary1 ))&&/\S/){$camel1hump+=1<<$CAMEL;}
    $CAMEL--;if(d efined($_=shift(@dromedary1))&&/\S/){
    $camel1hump+=1 <<$CAMEL;}$CAMEL --;if(defined($_=shift(
    @camel1hump))&&/\S/){$camel1hump+=1<<$CAMEL;}$CAMEL --;if(
    defined($_=shift(@camel1hump))&&/\S/){$camel1hump+=1<<$CAME
    L;};$camel.=split(/;/," \040.m'{/J\047\134}L^7FX'"))[$camel1h
    ump];}$camel.=" \n";}@camel1hump=split(/\n/,$camel);foreach(@
    camel1hump){chomp;$Camel=$_tr/LJF7\173\175\047\061\062\063
    45678/tr/12345678/JL7F\175\173\047/;$_=reverse;print"$_\040
    $Camel\n";}foreach(@camel1hump){chomp;$Camel=$_y/LJF7\173\17
    5\047\12345678/tr/12345678/JL7F\175\173\047/;$_=reverse;p
    rint"$\040$_$Camel\n";}#japh-Erudil";s;s*;;g;eval; eval
    ("seek\040DATA.0.0;");undef$;$_=<DATA>;s$s*$s$g;( );s
    ;^.*_ ;;map{eval'print"$_\n";}/.4}/g; _DATA_ \124
    \1 50\145\040\165\163\145\040\157\1 46\040\1 41\0
    40\143\141 \155\145\1 54\040\1 51\155\ 141
    \147\145\0 40\151\156 \040\141 \163\16 3\
    157\143\ 151\141\16 4\151\1 57\156
    \040\167 \151\164\1 50\040\ 120\1
    45\162\ 154\040\15 1\163\ 040\14
    1\040\1 64\162\1 41\144 \145\
    155\14 1\162\ 153\04 0\157
    \146\ 040\11 7\047\ 122\1
    45\15 1\154\1 54\171 \040
    \046\ 012\101\16 3\16
    3\15 7\143\15 1\14
    1\16 4\145\163 \054
    \040 \111\156\14 3\056
    \040\ 125\163\145\14 4\040\
    167\1 51\164\1 50\0 40\160\
    145\162 \155\151
    \163\163 \151\1
    57\156\056
```

Inhaltsverzeichnis

Abbildungsverzeichnis	iv
1. Die Scriptsprache Perl	1
1.1. Perl Features	1
1.2. Perl Sprachumfang	1
1.2.1. Variablen, Datentypen, Kontext	1
1.2.2. Perl's vordefinierte Variablen	2
2. Perl Database Interface (DBI)	3
2.1. Einführung	3
2.2. Vorteile von Perl DBI	3
2.3. DBI Architektur	4
2.3.1. DBI BackEnds – Database Drivers	4
2.3.2. Vom Perl Script bis zur Datenbank	4
2.3.3. Datenschnittstelle zwischen DBI und DBD	5
3. DBI Programmierung	6
3.1. Transaktionen	6
3.1.1. Automatische Transaktion-Abwicklung	6
3.1.2. Commit Transaktionen	6
3.1.3. Rollback Transaktionen	7
3.1.4. Disconnect – Commit oder Rollback?	7
4. DBI in der Praxis	8
4.1. DBI DataBase Proxy – DBD::Proxy	8
4.1.1. Proxy Server	8
4.1.2. Proxy Client	8
4.2. Perl DBI Integration im Apache Webserver	9
4.2.1. DBI für den Apache – Apache::DBI	9
Literaturverzeichnis	11
Glossar	12
Anhang	13
A. Database Driver List	14
B. Datenbank per Proxy bedienen – iX Artikel	15

B.1. Architektur des Proxy-Servers	16
B.2. Zusammenspiel von Modulen	17
B.3. Zwei Wege zur Installation	18
B.4. Zugang zur Datenbank beschränken	19
B.5. Kompression spart Übertragungszeit	20
C. Perl DBI Examples	22
C.1. Useful DBI Snippets	22
C.1.1. List installed DBI Database Drivers	22
C.2. Google Query	22
C.2.1. Source Code	22
C.3. DBI Proxy Client	24
C.4. DBI Proxy Server	24

Abbildungsverzeichnis

2.1. DBI Architektur – DBI BackEnds	4
2.2. DBI Architektur – Datenfluss	5

1. Die Scriptsprache Perl

Die nächsten Abschnitte erläutern nur sehr kurz ausgewählte Themen von Perl.

1.1. Perl Features

Sprache:

- Interpretersprache
- Kontextsensitive Scriptsprache
- Objekt-Orientiert (seit Perl ≥ 5)
- Für viele Plattformen verfügbar

Anwendung:

- **Textmanipulation**
- System Administration
- Web Development
- Network Programming
- ...

1.2. Perl Sprachumfang

1.2.1. Variablen, Datentypen, Kontext

Folgende Datentypen sind in Perl enthalten:

Scalar

„Eindimensionale“-Werte, zum Beispiel Zahlen (Integer, Float) oder Strings. Skalare Variablen beginnen mit \$.

Beispiel:

1. Die Scriptsprache Perl

```
1     $zahl = 1233;  
2     $zeichenkette = "DasList_eine_Zeichenkette";
```

Array

Liste von skalaren Elementen (Scalars). Array Variablen werden mit @ deklariert.

Beispiel:

```
1     @liste = ( "palim" , 1234 ); # oder  
2     @liste2 = qw( palim 1234 );
```

Hash

Zuordnung von Key-/Value-Paaren. Key und Values können Skalare oder Arrays sein. Hashes werden mit % gekennzeichnet.

Beispiel:

```
1     %myHash = ( 'key1' => "val1", 'key2' => 123 );
```

Das Ergebnis von Operationen ist vom Kontext der Umgebung abhängig. Kontextarten:

Void Context

Erwartet keinen Wert

Scalar Context

Erwartet einen skalaren Typ

List Context

Erwartet ein Array

Weitere Kontexte

Hash Context, Boolean Context, ...

Meistens reicht es aus, sich auf den Scalar und List Context zu beschränken.

Hinweis: Aufgrund der Kontextsensitivität entspricht zum Beispiel der skalare Wert eines Arrays der Anzahl der vorhandenen Elemente im Array!

1.2.2. Perl's vordefinierte Variablen

Perl hilft dem Programmierer bei Standardproblemen. In Perl existieren eine Vielzahl von vordefinierten Variablen, die automatisch von Perl aktualisiert werden.

Die am häufigsten verwendete Variable ist \$_. Diese Variable enthält den jeweilig aktuellen Wert einer Operation: zum Beispiel die aktuelle Zeile von einer zu lesenden Datei oder der aktuelle Key beim Iterieren über einen Hash, ...

2. Perl Database Interface (DBI)

2.1. Einführung

DBI ist ein Perl Modul und bietet eine einheitliche Schnittstelle zur datenbank-unabhängigen Programmierung.

DBI unterstützt eine Vielzahl von Datenbanken, zum Beispiel Oracle, Informix, mSQL, MySQL, Ingres, Sybase, DB2, und PostgreSQL. Zusätzlich werden auch Dateien wie CSV und DBM unterstützt.

Eine Verbindung zu Datenquellen wie ODBC, JDBC und ADO ist ebenfalls möglich.

2.2. Vorteile von Perl DBI

Im Folgenden werden wichtige Vorteile von DBI vorgestellt.

Plattform Unabhängigkeit

Die Plattform Unabhängigkeit wird durch die Verfügbarkeit von Perl auf diversen Betriebssystemen gewährleistet.

Perl's Stärke in der Textverarbeitung

Perl aka *Practical Extraction and Reporting Language* ist *die* Script-Sprache im Bereich Textverarbeitung.

Die Möglichkeiten den professionellen Textanalyse in Verbindung mit Datenbanken bieten enorme Anwendungsmöglichkeiten.

Somit kann DBI im *Data Mining* und *Data Warehousing* für ETL und Analyse Zwecke eingesetzt werden.

Web Anwendungen

Perl ist als de facto Sprache in der für dynamische Web Anwendungen mit CGI¹. Mittels diesen Datenbanken im Hintergrund können leistungsfähige Web Anwendungen realisiert werden.

Zum Beispiel Online Shopping, . . .

¹Der Apache WebServer bietet eine vollständige Integration namens *mod_perl*.

2.3. DBI Architektur

Dieser Abschnitt wird die DBI Architektur erläutern. In den ersten Abschnitten werden die notwendigen Grundlagen vermittelt.

2.3.1. DBI BackEnds – Database Drivers

DBI ist in zwei Software-Komponenten getrennt:

1. Database Interface (DBI), Database Independent
2. Database Drivers (DBD), Database Dependent

Das DBI bietet die Schnittstelle zwischen dem Perl Script und dem jeweiligen Datenbank Treiber. DBI bietet eine einheitliche Programmierschnittstelle (API) unabhängig von der Datenbank.

Die Database Drivers (DBD) sind die spezifischen Treiber für die jeweilige Datenbank. Diese sind auch für die Kommunikation mit der Datenbank zuständig (Ausführen von Operationen etc.).

In Abbildung 2.1 ist der Zusammenhang schematisch dargestellt.

Im [Anhang](#) ist eine Liste mit den verfügbaren Database Drivers enthalten.

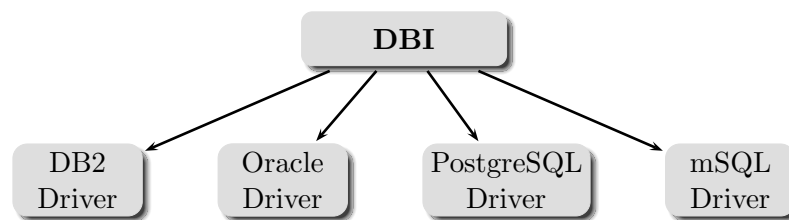


Abbildung 2.1.: DBI Architektur – DBI BackEnds

2.3.2. Vom Perl Script bis zur Datenbank

In der Abbildung 2.2 ist die DBI Architektur in Bezug auf den Datenfluss detailliert dargestellt.

Das Perl Script instanziiert das DBI Modul. Danach wird eine Verbindung zur Datenbank mittels des entsprechenden DBD (Database Driver) Moduls angegeben. DBI sucht und instanziiert das DBD Modul anschließend automatisch.

Das Perl Script kann die vom DBI Modul angebotenen Operation aufrufen. Dabei werden die Methoden von DBI an den ausgewählten DBD weitergereicht². Der DBD führt dann die Operationen (Datenbank spezifisch)auf der Datenbank aus.

Das Resultat wird vom DBD an DBI und anschließend wieder an das Perl Script zurückgegeben.

²DBI fungiert als *Abstrakte Superklasse*

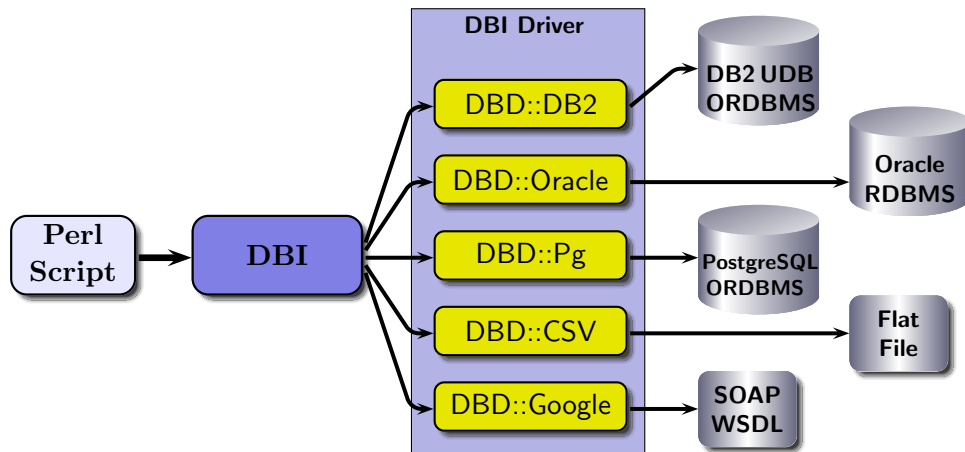


Abbildung 2.2.: DBI Architektur – Datenfluss

2.3.3. Datenschnittstelle zwischen DBI und DBD

Die Daten im Datenfluss zwischen Perl Script, DBI und DBD entsprechen den Perl Datentypen. Allein der DBD ist die Schnittstelle zur Datenbank und implementiert die Umsetzung von Perl Datentypen in datenbank-spezifische Typen und vice versa.

Damit wird auch die DBI DBD Schnittstelle Perl-konform belassen. Daraus resultiert auch die Möglichkeit grundverschiedene Datenbanken einheitlich zu behandeln.

Die Implementierung neuer DBI Driver richtet sich nach der DBI Spezifikation. In DBI sind schon verschiedene Standard-Operationen vordefiniert, nur wenige müssen zusätzlich implementiert werden.

3. DBI Programmierung

3.1. Transaktionen

Dieser Abschnitt beschreibt die Verwendung und Programmierung von Transaktionen mit DBI.

3.1.1. Automatische Transaktion-Abwicklung

Der ISO Standard für SQL beschreibt ein Modell für transaktionsorientierte Datenbanken: Jede Datenbank-Verbindung muss transaktionsbasiert ablaufen. Das bedeutet aber, dass jede Transaktion mit *commit* oder *rollback* abgeschlossen werden muss.

Damit dennoch einzelne Statements ausgeführt werden können, unterstützen transaktionsorientierte Datenbanksysteme einen *AutoCommit* Mechanismus. Ein *AutoCommit* agiert als ein *commit* nach jedem Statement.

In DBI kann das *AutoCommit*-Verhalten aktiviert bzw. deaktiviert werden. DBI stellt sicher, dass eine Deaktivierung bei einem nicht transaktionsorientierten Datenbanksystem zu einem Fehler führt.

Wie auch bei ODBC und JDBC ist bei DBI das *AutoCommit* grundsätzlich aktiviert. Ein explizites Festlegen von *AutoCommit* sollte dennoch vorgenommen werden.

Das (De)Aktivieren von *AutoCommit* erfolgt beim Verbinden zur Datenbank:

```
1 my $dbh = DBI->connect( "dbi:Pg:dbname=phonebook", "username", "password" , {  
2   AutoCommit => 0  
3 } );
```

Dieses Beispiel deaktiviert das *AutoCommit* und ermöglicht damit Transaktionen.

3.1.2. Commit Transaktionen

DBI definiert die Methode `commit()` zum Festlegen einer Transaktion. Der Aufruf erfolgt durch ein existierendes Database-Handle:

```
1 $dbh->commit();
```

Hinweis: Erfolgt der Aufruf von `commit()` bei aktiviertem *AutoCommit* wird eine Warnung ausgegeben.

3.1.3. Rollback Transaktionen

DBI definiert die Methode `rollback()` zum „Zurückrollen“ einer Transaktion. Wiederum erfolgt der Aufruf durch ein vorhandenes Database-Handle:

```
1 $dbh->rollback();
```

Hinweis: Erfolgt der Aufruf von `rollback()` bei aktiviertem *AutoCommit* wird eine Warnung ausgegeben.

3.1.4. Disconnect – Commit oder Rollback?

Wenn die Verbindung zur Datenbank unterbrochen wird, bleibt der Zustand der aktuellen Transaktion undefiniert (*AutoCommit* deaktiviert). Je nach Datenbank wird die ausstehende Transaktion durchgeführt (Oracle, Ingres) oder verworfen (Informix).

Daher sollte vor einem `disconnect()` explizit ein `commit()` oder `rollback()` aufgerufen werden.

Hinweis: Wenn die Applikation aufgrund eines Fehlers (`die()`) beendet wird, liegt die Entscheidung beim Programmierer des Drivers. Im Regelfall wird der Driver ein `rollback()` aufrufen.

4. DBI in der Praxis

Im den folgenden Abschnitten werden einige praktische Anwendungen von Perl DBI vorgestellt.

4.1. DBI DataBase Proxy – DBD::Proxy

In diesem Abschnitt wird die Verwendung des DBI Proxy Servers und Clients beschrieben.

4.1.1. Proxy Server

Das Perl Modul DBI::ProxyServer stellt einen Proxy Server für DBI-Verbindungen zur Verfügung. Die Kommunikation zwischen dem Proxy Server und seinen Clients erfolgt über PerlRPC über einen frei wählbaren TCP-Port.

Das folgende Beispiel stellt die minimale Implementierung eines Proxy Servers dar:

```
1 #!/usr/bin/perl
2 use DBI::ProxyServer;
3 DBI::ProxyServer :: main(@ARGV);
```

Wird dieses Skript wie folgt gestartet, wird ein Proxy Server auf dem TCP Port 4404 gestartet:

```
1 ./ proxy_server . pl --localport 4404
```

4.1.2. Proxy Client

Als Gegenstück zum DBI::ProxyServer dient das Perl Modul DBI::Proxy. Dieses implementiert einen Proxy Client, welcher über einen Proxy Server auf eine via Perl DBI erreichbare Datenbank zureifen kann.

Das folgende Beispiel beschreibt den Zugriff auf den weiter oben gestarteten Proxy Server:

```
1 #!/usr/bin/perl
2 use DBI;
3 $host = "localhost";
4 $port = "4404";
```

```
5 $dsn    = "DBI:Pg:dbname=phonebook";
6 $user   = "user";
7 $passwd = "pass";
8 $proxy  = "DBI:Proxy:hostname=$host;port=$port;dsn=$dsn";
9 $proxy  = $dsn;
10 $dbh    = DBI->connect($proxy, $user, $passwd);
```

Der client authentifiziert sich hierbei mit dem angegebenen Benutzernamen und Passwort durch den DBI Proxy direkt an der Datenbank.

4.2. Perl DBI Integration im Apache Webserver

Die Integration von Perl in den Apache Webserver ist unter dem Begriff *mod_perl* bekannt. Bei *mod_perl* wird der Apache gegen den Perl-Interpreter gelinkt (statisch oder dynamisch). Damit ist es möglich Perl-Skripte im Kontext des Apache auszuführen. Folgende Vorteile ergeben sich:

Enorme Performance Steigerung

Bei CGI musste für jeden Aufruf eines Perl-Skripts der (externe) Perl-Interpreter aufgerufen werden.

Die Interpretation und Ausführung wird vom Apache nun selbst übernommen.

Precompiled Perl-Skripts

mod_perl bietet die Möglichkeit den Bytecode von Perl-Skripten zu speichern und diesen bei späteren Aufrufen erneut zu verwenden (eine weitere Interpretation entfällt).

Dieses Verhalten ist vergleichbar mit der Servlet-Technologie.

4.2.1. DBI für den Apache – Apache::DBI

Für *mod_perl* existiert eine zusätzliche DBI (*Apache::DBI*) Schnittstelle. Diese DBI Schnittstelle bietet in Verbindung mit herkömmlichen DBI Schnittstelle folgende zusätzlichen Features:

Authentifizierung und Autorisierung

Ein zusätzliches Modul¹ erlaubt die Authentifizierung und Autorisierung (*.htaccess*) von Benutzern und Gruppen mittels der DBI Schnittstelle.

Aufbau von Datenbank Verbindungen beim Start von Apache

Beim Start vom Apache Server können mehrere Verbindungen zu Datenbanken hergestellt werden.

Dies kann im Zusammenhang mit der Authentifizierung verwendet werden.

¹Apache::AuthDBI

Persistente Datenbank Verbindungen

Das Apache::DBI Modul verhält sich wie ein Proxy und speichert bestehende Datenbank Verbindungen.

Wenn DBI eine neue Verbindung zu einer Datenbank herstellen will, wird zuerst geprüft, ob in Apache::DBI noch eine offene Verbindung für diese Datenbank existiert. Wenn diese nicht verfügbar ist, wird eine neue aufgebaut und in Apache::DBI abgelegt.

Konfigurierbares Rollback

Aufgrund der persistenten Verbindungen werden Transaktionen nach dem Ablauf eines Scripts nicht automatisch abgeschlossen. Damit die Daten trotzdem einen konsistenten Zustand haben, verwendet Apache::DBI einen Perl-CleanupHandler² nach jedem Request. Dieser Handler ruft ein `rollback ()` auf.

Verifizierung von Datenbank Verbindungen

Apache::DBI nutzt eine `ping` Methode um den Status (Timeout) einer Datenbank Verbindung zu ermitteln.

Diese Methode muss vom Database Driver (DBD) bereitgestellt werden.

²`mod_perl` stellt CallBack-Routinen für verschiedene Situationen bereit

Literaturverzeichnis

- [DB00] Alligator Descartes and Tim Bunce. *Programming the Perl DBI*. O'Reilly, first edition, February 2000. 12
- [DBI] Perl DBI Homepage. WebSite. <http://dbi.perl.org/>.
- [HB04] Hendrik Brückner. P_BL_AT_EX Report Template. P_BL_AT_EX, July 2003/04.
- [Jen] Stephen B. Jenkins. Perl Camel Code. The use of a camel image with the topic of Perl is a trademark of O'Reilly & Associates, Inc.
- [Wie00] Jochen Wiedmann. Datenbank per proxy bedienen. Website, October 2000. 21

Glossar

A

ACID Atomic Consistent Isolated Durable

C

CGI Common Gateway Interface

D

DBD DBI Database Driver

ist der datenbank-abhängige (database dependent) Teil der DBI-Architektur. Der Driver führt Operationen auf der Datenbank aus.

DBI Perl Database Interface

DBM DataBase Management

DBM files are a storage management layer that allows to store information in files as pairs of strings, a key, and a value. DBM files are binary files and the key and value strings can also hold binary data. [DB00]

M

mod_perl Die Integration des Perl Interpreters in den Apache Webserver. Dabei kann mod_perl als separates Modul (Shared Object) oder statisch (static linked) vorliegen.

mod_perl erlaubt das Ausführen von Perl-Skripten im Kontext des Apache.

P

Perl *Practical Extraction and Reporting Language*

Perl is a general-purpose programming language originally developed for text manipulation and now used for a wide range of tasks including system administration, web development, network programming, GUI development, and more. The language is intended to be practical (easy to use, efficient, complete) rather than beautiful (tiny, elegant, minimal). Its major features are that it's easy to use, supports both procedural and object-oriented (OO) programming, has powerful built-in support for text processing, and has one of the world's most impressive collections of third-party modules.

Anhang

A. Database Driver List

DBD::ADO	DBD::ASAny
DBD::Adabas	DBD::AnyData
DBD::CSV	DBD::Chart
DBD::DB2	DBD::DBMaker
DBD::DtfSQLmac	DBD::Empress
DBD::EmpressNet	DBD::Excel
DBD::File	DBD::Fulcrum
DBD::Google	DBD::Illustra
DBD::Informix	DBD::Informix4
DBD::Ingres	DBD::InterBase
DBD::JDBC	DBD::LDAP
DBD::Mimer	DBD::Mock
DBD::NET	DBD::ODBC
DBD::Oracle	DBD::Ovrimos
DBD::Pg	DBD::PgPP
DBD::PgSPI	DBD::PrimeBase
DBD::QBase	DBD::RAM
DBD::Redbase	DBD::SQLite
DBD::SearchServer	DBD::Solid
DBD::Sprite	DBD::Sybase
DBD::Template	DBD::Teradata
DBD::Trini	DBD::Unify
DBD::XBase	DBD::mysql
DBD::mysql-AutoTypes	DBD::mysqlPP
DBD::pNET	

B. Datenbank per Proxy bedienen – iX Artikel

Auf die Oracle-Datenbank ohne spezielle Treiber zuzugreifen, das womöglich noch vom Internet aus und trotzdem sicher, oder auf die Access-Datenbank vom Linux-Webserver: von Perl aus dank DBI-Proxy leicht zu schaffen. DBI, das Database independent Interface for Perl, ist die Standardschnittstelle zum Zugriff auf Datenbanken für Perl-Programmierer. Ähnliches kennt man auch von ODBC oder JDBC, nur ist DBI erheblich einfacher - eingefleischte Perl-Programmierer dürften das nicht weiter verwunderlich finden. Besonders bekannt sind Proxies im Web: Sie haben die Aufgabe, Browsern HTML-Seiten zur Verfügung zu stellen. Die Gründe für ihre Benutzung sind vielfältig: Den einen hindert ein Firewall am direkten Zugang zum WWW, der andere mag den großen Cache von HTML-Seiten, den ein solcher Proxy meist hat, und den dritten beschränkt der Proxy auf eine Liste verfügbarer Webserver. Letzten Endes ist ein Proxy nichts anderes als ein Stellvertreter, der zwischen dem eigentlichen Server und einem Client steht, und dieselben Dienste anbietet. Im Idealfall ist dieser Vorgang für den Client transparent. Analog ist ein Datenbank-Proxy eine Softwareschicht, die sich zwischen den Client (das Anwendungsprogramm) und den Server (die eigentliche Datenbank) schiebt. Was auf den ersten Blick als überflüssiger Aufwand erscheint, ist tatsächlich ein nützliches Prinzip, dessen Vorteile in diesem Fall sind:

- Mit Hilfe des Proxy lassen sich fast beliebige Restriktionen hinsichtlich des Datenbankzugriffs erzwingen;
- er erlaubt den Zugriff auf Datenbanken, die nicht von sich aus netzwerkfähig sind;
- kommerzielle Datenbanken wie Oracle sind häufig nur über proprietäre Schnittstellen ansprechbar, die auf jedem Client vorhanden sein müssen.

Ursprünglich entwickelt wurde der DBI-Proxy aus Sicherheitsgründen. Ein typisches Szenario ist, dass die Datenbank auf einem Intranetserver läuft, der Proxy auf dem Firewall und der Client irgendwo im Internet. Dadurch ist der sichere Zugriff auf interne Datenbanken von außen möglich. Ebenso hilft der Proxy weiter, wenn der Linux-Webserver Daten aus Access unter Windows benutzen soll. Beispiele für solche Anforderungen sind auch dBase- oder Textdateien mit Trennzeichen, die man in Perl als Datenbanken anwenden kann: Der Treiber DBD::CSV macht es möglich. Viele Windows-Benutzer glauben zwar, dass Access netzwerkfähig sei, und verweisen darauf, dass sie von ihrem Arbeitsplatz .mdb-Dateien auf einem Server benutzen. Allerdings läuft in diesem Fall Datenbanksoftware auf der Maschine des Benutzers und eben nicht auf dem Server. Das kann das Arbeiten bremsen, wie das Beispiel der SQL-Abfrage `SELECT email FROM Adressen WHERE name = Larry Wall` zeigt.

Eine wirklich netzwerkfähige Datenbank bekommt in diesem Fall den Text der Abfrage übermittelt. Einschließlich eines gewissen Overheads dürften etwa 500 bis 1000 Bytes über das Netz zum Server fließen. Er wählt den richtigen Datensatz aus und schickt diesen zurück. All dies belastet das Netz kaum. Nicht so im Fall einer Datenbank wie Access: Da die auf dem

Client laufende Software die richtige Zeile aus der Tabelle auswählt, muss im Extremfall die ganze Tabelle oder zumindest der ganze Index auf den Client transferiert werden. Bei großen Tabellen kann das aufwändig sein und lange dauern. Selbst zur Anbindung kommerzieller und netzwerkfähiger Datenbanken benutzen viele den DBI-Proxy. Die eigentlichen Schnittstellen sind meist teuer oder schlecht bis gar nicht portabel, weil sie in C geschrieben sind und kaum ein Hersteller Quelltexte anbietet. In diesem Fall genügt es, dass ein Rechner (der Proxy) die Schnittstelle des Herstellers einsetzt, alle anderen brauchen nur zwei oder drei Perl-Module.

B.1. Architektur des Proxy-Servers

Einer der größten Vorzüge von Perl ist die Menge vorhandener Komponenten und Modulen. Der DBI-Proxy macht davon reichlichen Gebrauch und besteht aus einer Reihe von Komponenten. Das Beispiel geht von einem unter Windows laufenden Perl-Skript aus, das mit Access arbeiten soll. Ferner sei ein ODBC-DSN (Data Source Name) adressen eingerichtet, der den Zugriff auf die Access-Datenbank ermöglicht. DBI kann mehrere Datenbank-Handles samt Statement-Handles verwalten (Abb. 1). Um mit der Datenbank zu arbeiten, benötigt das Skript zunächst ein so genanntes Datenbank-Handle (`$dbh`). Dieser dient als abstrakte Repräsentation der Datenbank und wird durch einen Aufruf der DBI-Methode `connect()` gewonnen:

```
1 my $dsn = "DBI:ODBC:adressen";
2 my $dbh = DBI->connect($dsn, "myusername", "mypassword");
```

Die übergebenen Parameter sind der Datasource Name oder DBI-DSN, im Falle des ODBC-Treibers `DBD::ODBC` nichts anderes als ein Verweis auf die ODBC-DSN, sowie der Benutzername und das Passwort. Das erhaltene Datenbank-Handle erlaubt nun die Ausführung von SQL-Statements:

```
1 $dbh->do("INSERT INTO adressen ('Larry Wall', 'larry@wall.org')");
```

Ergebnistabellen erlauben das Lesen von Daten. Innerhalb von DBI werden sie durch Statement Handle (`$sth`) repräsentiert, aus denen man zeilenweise lesen kann. Das sieht etwa so aus:

```
1 my $sth = $dbh->prepare("SELECT name, email FROM adressen");
2 $sth->execute();
3 while (my $ref = $sth->fetchrow_arrayref()) {
4     print "Name_=$ref->{'name'}, Email_=$ref->{'email'}\n";
5 }
```

DBI erlaubt es, beliebig viele gleichzeitige Datenbank- oder Statement-Handles anzulegen, sodass man einen Baum von Objekten aufbauen kann.

Die grobe Idee des Proxy-Servers ist die folgende: Auf dem Server wird genau derselbe Baum generiert. Die Objekte auf dem Server sind echt, der Client verwendet so genannte Proxy- oder Schattenobjekte.

Um auf dem Client ein Datenbank-Handle zu generieren, ruft man das Proxy-Objekt DBI auf:

```
1 my $dsn = "DBI:Proxy:hostname=ntserver;port_=2000;\
2             dsn=DBI:ODBC:adressen";
3 my $dbh = DBI->connect($dsn, "myusername", "mypassword");
```

Dadurch geschieht Folgendes:

1. Der Hostname beziehungsweise die IP-Nummer des Servers (hostname=ntserver) und die serverseitige DSN (dsn=DBI:ODBC:adressen) werden gelesen.
2. Das Proxy-Objekt DBI transferiert die Parameter zum DBI-Objekt auf dem Server nts-erver.
3. Der Server ruft sein DBI-Objekt mit den empfangenen Parametern auf. Es liefert als Ergebnis ein Datenbank-Handle.
4. Der Server speichert diesen, erzeugt ein Proxy-Objekt und schickt es zum Client, der es als Ergebnis des Methodenaufrufs übergeben bekommt.

Bis auf den geänderten DSN ist dieser Vorgang völlig transparent. Mit den Proxy-Objekten kann man arbeiten, als wären sie die Originale. Man kann die Methode `do` aufrufen, um SQL-Anweisungen auf dem Server auszuführen oder mit `prepare` eine Ergebnistabelle generieren lassen. In allen Fällen werden automatisch die Parameter zum Objekt auf dem Server transferiert, in einen entsprechenden Methodenaufruf gewandelt, und die Ergebnisse fließen zurück zum Client.

B.2. Zusammenspiel von Modulen

Hinter diesem einfachen Schema versteckt sich eine ganze Menge. Die eigentlichen Perl-Module `DBD::Proxy` (Client) und `DBI::ProxyServer` (Server) sind aber überraschenderweise nicht einmal besonders komplex. Das liegt in erster Linie daran, dass beide ausgiebigen Gebrauch von anderen Perl-Modulen oder Komponenten machen.

-
- `DBI`, die eigentliche Datenbankschnittstelle, ist bereits bekannt. Der Proxy-Server, das heißt das Perl-Modul `DBI::ProxyServer`, ist als Subklasse von `DBI` implementiert.
- Das Modul `RPC::PIServer` ist der Serverteil von `PIRPC` (Perl-RPC), ein System zur Implementierung von Client/Server-Anwendungen in Perl, ähnlich Javas `RMI` (Remote Method Invocation) oder Suns `RPC` (Remote Procedure Call), aber erheblich einfacher anwendbar. Die Verwaltung der Proxy-Objekte und die Verbindung zwischen ihnen und den echten Objekten ist seine Aufgabe. Beim Einsatz von `DBI::Proxy` benutzt der Client immer denselben Pseudo-Treiber `DBD::Proxy`, nur der Proxy-Server kommuniziert direkt mit den Datenbanksystemen (Abb. 2).
- `Net::Daemon` ist eine abstrakte Klasse zur Erstellung beliebiger Serveranwendungen: Im Unterschied zu `RPC::PIServer` ist dieses Modul nicht an ein bestimmtes Protokoll gebunden. Es leistet aber bereits eine ganze Menge, zum Beispiel verschiedene, an das jeweilige Betriebssystem angepasste Serverarchitekturen (`Thread`, `fork`, `Single-Client`), Zugriffskontrollen auf der Ebene von IP-Nummern, Verarbeitung von Konfigurationsdateien et cetera.
- `Storable` erledigt die Serialisierung von Daten. Man steckt ein beinahe beliebig komplexes Perl-Objekt in einen Aufruf von `Storable::freeze()` hinein, beispielsweise `Hash-`, `Array-`

sowie Hash-Referenzen von Hash-Referenzen und erhält einen String. Diesen kann man in einer Datei speichern oder über das Netz schicken und später durch einen Aufruf von `Storable::thaw` in dasselbe, wiederhergestellte Perl-Objekt wandeln.

B.3. Zwei Wege zur Installation

Alle diese Komponenten müssen zunächst installiert werden. Nötig sind die Module `Storable`, `Net::Daemon`, `PIRPC` und `DBI` sowie ein passender `DBI`-Treiber. Eine Unix-Maschine soll als Proxy-Client fungieren, ein Windows-Rechner als Server. Mit ActiveStates Perl unter Windows erledigt der folgende Aufruf dort alles Nötige:

```
ppm
install Storable
install Net-Daemon
Install PIRPC
install DBI
install DBD-ODBC
```

`DBI::ProxyServer` setzt unter anderem auf `RPC::PIServer` und `Net::Daemon` auf. `Storable` serialisiert die Daten für den Transport über das Netz (Abb. 3).

Zum Testen der korrekten Funktion bietet sich die `DBI`-Shell an:

```
C:\> dbish DBI:ODBC:adressen myusername mypassword ...
Connecting to 'DBI:ODBC:adressen' as 'myusername' ...
myusername@DBI:ODBC:adressen>
```

Meldet sich am Ende der Systemprompt (`!`), ist alles korrekt eingerichtet, und man kann den Server starten. Das geht am einfachsten mit dem Skript `dbiproxy`, das ebenfalls Teil von `DBI` ist:

```
dbiproxy -logfile C:\temp\dbiproxy.log -localport 2000
```

Nun läuft der Server und wartet auf Port 2000 auf ankommende Verbindungen. `dbiproxy` kennt übrigens eine ganze Reihe von Optionen, die meisten stammen aus der untersten Schicht, nämlich von `Net::Daemon`. Eine Liste aller Optionen liefert

```
dbiproxy -help
```

Läuft der Server, ist der Client unter Unix zu bauen. Auf einem gut eingerichteten System verläuft die Installation dank des Perl-Moduls `CPAN` besonders einfach: Es genügt ein

```
perl -MCPAN -e "install Bundle::DBI"
```

Fehlt das `CPAN`-Modul, muss man sich die nötigen Module `Storable`, `Net::Daemon`, `RPC::PIClient` und `DBI` selbst per `FTP` vom `CPAN` (beispielsweise `ftp.cpan.org`) besorgen. Die Installation verläuft für jedes Modul identisch:

```
gzip -d <Modul>.tar.gz
tar xf Modul.tar
cd Modul
perl Makefile.PL
make
make test make install
```

Zur Prüfung, ob alles geklappt hat, kann man wieder die dbish benutzen:

```
dbish DBI:Proxy:host=ntserver;port=2000;\
dsn=DBI:ODBC:adressen myusername mypassword
...
Connecting to 'DBI:Proxy:host=ntserver;port=2000;\
dsn=DBI:ODBC:adressen' as 'myusername' ...
myusername@DBI:Proxy:host=ntserver;\
port=2000;dsn=DBI:ODBC:adressen>
```

B.4. Zugang zur Datenbank beschränken

Nach der Einrichtung des Proxy-Servers ist dieser praktisch eine im Netz laufende Datenbank. Sofern die zu Grunde liegende Datenbank nicht eigene Zugriffsbeschränkungen implementiert, eine für alle und alles Offene! Das ist möglicherweise in einem Intranet erwünscht, aber auf keinen Fall im Internet. Der Proxy-Server benötigt ein eigenes System der Zugriffskontrolle.

DBI::Proxy bietet dazu umfangreiche Möglichkeiten, die allerdings nicht mehr über die Kommandozeile zugänglich sind. Stattdessen benötigt man eine Konfigurationsdatei in einem recht einfachen Format: Sie enthält Perl-Quelltext, genauer gesagt eine Hash-Referenz. Die Schlüssel dieses Hash entsprechen den Optionen, die auch auf der Kommandozeile verwendbar sind. Beispiel:

```
1 { ' logfile ' => 'C:\temp\dbiproxy.log',
2   ' localport ' => '2000'
3 }
```

Das sind genau die Optionen, die der Proxy-Server weiter oben beim ersten Start mitbekommen hat. Um diese Konfigurationsdatei zu verwenden, ist er zu beenden und erneut zu starten:

```
dbiproxy -configfile c:\etc\dbiproxy.config
```

Ein weiterer Schlüssel in dieser Konfigurationsdatei ist `clients`. Der zugehörige Wert ist eine Array-Referenz mit einer Liste zulässiger Clients. Um zum Beispiel den Zugriff nur von Rechnern mit den IP-Nummern `192.168.1.*` und zusätzlich dem Rechner `larry.wall.org` zu gestatten, könnte man die Konfigurationsdatei so ändern:

```
1 { ' logfile ' => 'C:\temp\dbiproxy.log',
2   ' localport ' => '2000'
3   # Zugriffskontrolle
4   ' clients ' => [
5     # Akzeptiere die IP-Nummern 192.168.1.*
6     { 'mask' => '^192\.168\.1\.\d+$',
7       'accept' => 1
8     },
9     # Akzeptiere larry.wall.org
10    { 'mask' => '^larry\.wall\.org$',
11      'accept' => 1
12    }
13  ]
14 }
```

```
13     # Alles andere ablehnen
14     { 'mask' => '.*',
15       'accept' => 0
16     }
17 ]
18 }
```

Auch eine Einschränkung auf bestimmte Benutzernamen ist möglich. Ein ganz besonderer Leckerbissen sind aber die Zugriffsbeschränkungen auf bestimmte SQL-Queries. Der eingangs angesprochene WWW-Server soll vermutlich nur eines können, nämlich neue Adressen erfassen. Mit anderen Worten, er soll nur eine bestimmte Query ausführen können. Dazu dient ein neuer Eintrag in der Client-Liste der Konfigurationsdatei:

```
1 # www.meinefirma.de mit SQL-Restriktionen
2 { 'mask' => '^larry\.wall\.org$',
3   'accept' => 1,
4   'sql' => {
5     'query1' => 'INSERT INTO _adressen VALUES_(?,_?)'
6   }
7 }
```

Leider geht es in diesem Fall nicht ohne Änderungen am Client-Programm. Ohne die SQL-Restriktionen stünde im CGI-Skript Folgendes:

```
1 $dbh->do("INSERT INTO _adressen VALUES_(?,_?)",
2         undef, $name, $email);
```

```
3
4 Das muss nun ßheien:
```

```
5
6 $dbh->do("query1", undef, $name, $email);
```

Natürlich kann man den Schlüssel query1 durch die ganze Query ersetzen und das Client-Programm unverändert lassen.

B.5. Kompression spart Übertragungszeit

Speziell im Internet ist eine weitere Funktion sinnvoll: Die Kompression der übertragenen Daten. Zu diesem Zweck übergibt man an den Proxy-Server beim Start die Option `compression` mit dem Wert `gzip`, entweder in der Konfigurationsdatei oder auf der Kommandozeile mit `compression=gzip`. Im Client ergänzt man die DSN wie folgt:

```
1 my $dsn =
2 "DBI:Proxy:hostname=ntserver;port=2000"
3 . " ;compression=gzip;dsn=DBI:ODBC:adressen";
```

Ein weiteres Perl-Modul erledigt die Kompression, nämlich `Compress::ZLib`. Es muss wie oben beschrieben installiert sein, um Daten komprimiert übertragen zu können.

Ebenfalls besonders für das Internet geeignet ist die verschlüsselte Kommunikation zwischen Client und Server. Infrage kommen dabei bislang nur die zwei symmetrischen Verschlüsselungsverfahren `DES` und `IDEA`, die die Perl-Module `Crypt::DES` und `Crypt::IDEA` implementieren. Diese sind also gegebenenfalls auf Client und Server zunächst zu installieren.

Auf dem Server geschieht die Konfiguration der Verschlüsselung am besten in der Konfigurationsdatei durch Eintragen beispielsweise dieser Zeilen:

```
1 use Crypt::IDEA();
2 my $key = Crypt::IDEA-> \
3 new('97cd2375efa329aceef2098babdc9721');
```

Um nun für den WWW-Server Verschlüsselung einzuschalten, ändert man den Client-Eintrag wie folgt:

```
1 # www.meinefirma.de mit SQL-Restriktionen
2 # und üVerschlüsselung
3 { 'mask' => '^larry\.wall\.org$',
4   'accept' => 1,
5   'sql' => {
6     'query1' => 'INSERT INTO adressen VALUES(?,?)'
7   },
8   'cipher' => $key
9 }
```

Auch auf dem Client (das heißt dem WWW-Server) muss man die Verschlüsselung einschalten. Das geht beim Aufruf von DBI->connect(), indem man die DSN ändert:

```
1 use Crypt::IDEA();
2 my $dsn =
3 "DBI:Proxy:hostname=ntserver;port=2000" . "; cipher=IDEA;\
4   key=97cd2375efa329aceef2098babdc9721"
5 . "; dsn=DBI:ODBC:adressen";
```

Mit DBI und den weiteren Modulen lässt sich so eine sichere und hinreichend schnelle Client/Server-Lösung aufbauen, mit der man auch Datenbanken ins Netz bekommt, die von Haus aus nur auf Einzelrechnern laufen. (ck)

[Wie00]

C. Perl DBI Examples

C.1. Useful DBI Snippets

C.1.1. List installed DBI Database Drivers

```
#!/usr/bin/perl -w
#
# Enumerates all data sources and all installed drivers
# From: Programming the Perl DBI
#
use DBI;

### Probe DBI for the installed drivers
my @drivers = DBI->available_drivers();

die "No drivers found!\n" unless @drivers; # should never happen

### Iterate through the drivers and list the data sources for each one
foreach my $driver ( @drivers ) {
    print "Driver :_ $driver\n";
    my @dataSources = DBI->data_sources( $driver );
    foreach my $dataSource ( @dataSources ) {
        print "\tData_Source_is_$dataSource\n";
    }
    print "\n";
}
exit;
```

C.2. Google Query

C.2.1. Source Code

```
#!/usr/bin/perl
#
# Google DBI Query Shell
#

# Display warnings, enable strict syntax
use warnings;
use strict ;

# Load DBI modules
use DBI;
use SQL::Parser;
```

```

# Load Readline modules
use Term::ReadLine;
# Setup terminal options
my $term = new Term::ReadLine 'Google_DBI_Query';
my $termAttribs = $term->Attribs;
$termAttribs->{'completion_entry_function'} =
    $termAttribs->{'list_completion_function'};
$termAttribs->{'completion_word'} = [ qw(
    SELECT FROM WHERE ORDER BY LIMIT
    select from where order by limit
    title URL google q
) ];
$term->AddHistory("SELECT title, URL FROM google WHERE q = 'perl'");
select $term->OUT || \*STDOUT; # select output filehandle

# Google Query Setup
my %googleOptions = (
    RaiseError => 1, # Standard DBI options
    PrintError => 0,
    lr          => [ 'en' ], # Language restrictions
    oe          => "utf-8", # Output encoding
    ie          => "utf-8" # Input encoding
);
my $googleAPIKey = glob ".googlekey";

# Create DataBase Handle
my $dbhHandle = DBI->connect( "dbi:Google:"
    , $googleAPIKey
    , undef
    , \%googleOptions );

# Starting the shell
my $text = ""; # text from the user
my $stmt; # db statement
print "Type \\q to exit\n";
while( $text ne "\\q" ){
    $text = "";
    $text = $term->readline( "Google-Query: ", $text );
    next if $text =~ /\q/; # exit if \q
    print "[DEBUG] Prepare: '$text'\n";
    # Prepare and execute statement
    $stmt = $dbhHandle->prepare( $text );
    $stmt->execute();
    # Display result
    while( my $r = $stmt->fetchrow_hashref ){ # iterate rows
        print "-" x 70 . "\n";
        foreach( keys %{$r} ){ # iterate columns
            printf( "%10s | %s\n", $_, $$r{$_} );
        }
        print "-" x 70 . "\n";
    }
}
}

```

C. Perl DBI Examples

```
$stmt->finish();
```

C.3. DBI Proxy Client

```
#!/usr/bin/perl -W

# $Id: proxy_client.pl,v 1.4 2004/04/13 17:31:48 it01130 Exp $

use strict ;
use warnings;

use FindBin;
use FindBin qw($RealBin);
use lib "$RealBin/./share/perl/5.8.3" ;
use lib "$RealBin/./lib/perl/5.8.3" ;

use DBI;

my $host = "localhost" ;
my $port = "4404" ;
my $dsn = "DBI:Pg:dbname=phonebook" ;
my $user = "thorsten" ;
my $passwd = "thorsten" ;

my $proxy = "DBI:Proxy:hostname=$host;port=$port;dsn=$dsn" ;

my $dbh = DBI->connect($proxy, $user, $passwd);

my $res = $dbh->selectall_arrayref("SELECT _nachname, _strasse, _ort, _plz, _vorwahl" .
    " , _durchwahl FROM _aio" .
    " WHERE _nachname LIKE _'LF.net%'");

for my $row (@$res){
    #my ($nachname, $strasse, $ort, $plz) = @$row;
    print join(" , " , @$row) . "\n" ;
}

```

C.4. DBI Proxy Server

```
#!/usr/bin/perl -W

# $Id: proxy_server.pl,v 1.1 2004/04/12 13:54:32 it01130 Exp $

use strict ;
use warnings;

use FindBin;
use FindBin qw($RealBin);
use lib "$RealBin/./share/perl/5.8.3" ;
use lib "$RealBin/./lib/perl/5.8.3" ;

use DBI::ProxyServer;
```

```
print localtime()." : starting DBI::ProxyServer!\n";  
DBI::ProxyServer :: main(@ARGV);  
print localtime()." : shutting down DBI::ProxyServer!\n";
```