CIPP

Cgl Perl Preprocessor

KAPITEL 1

CIPP - CGI Perl Preprocessor

Dieses Kapitel gibt eine Einführung in die Programmiersprache CIPP, die die komfortable Einbettung von Perl und SQL Code in HTML Seiten ermöglicht. In den darauf folgenden Kapiteln werden alle Möglichkeiten von CIPP anhand einer Befehlsrefernz erläutert.

Einführung

CIPP ist ein Akronym für: CGI Perl Preprocessor. Es bietet die Möglichkeit, HTML Seiten „programmierbar“ zu machen, dabei bedient es sich des CGI Konzeptes. Ein CGI-Programm wird vom Webserver aufgerufen und der von diesem Programm produzierte HTML Output wird vom Webserver an den aufrufenden Client-Webbrowser weitergegeben. CIPP ist das Werkzeug, mit dem diese CGI Programme mit minimalem Aufwand erstellt werden können.

CIPP generiert Perl Code

CIPP ist ein Präprozessor, der aus CIPP-Code Perl-Code generiert. Der generierte Perl-Code funktioniert als CGI-Programm, welches ohne weitere Bearbeitung unmittelbar als solches eingesetzt werden kann. Wenn der Entwickler bestimmte Regeln bei der Perl und CIPP Programmierung beachtet, funktioniert der generierte

Perl Code auch in persistenten Umgebungen, z.B. unter Apache/PerlMod und dem Oracle Application Server. Es gibt ein eigenes Kapitel, welches sich ausführlich diesem Thema widmet.

Bestandteile von CIPP Programmen

Ein CIPP-Programm kann zunächst einmal eine einfache HTML-Seite sein. D.h. Grundlage eines CIPP-Programmes ist HTML. Dieser HTML-Code kann mit CIPP-Befehlen erweitert werden, so daß aus der starren HTML-Seite ein Programm mit einer beliebig komplexen Logik wird.

CGI Programmierung hat oft den Nachteil, daß viel Programmoverhead entsteht, wenn nur einfache HTML Seiten generiert werden sollen. Bei CIPP wird genau so viel „Programm“ in eine HTML Seite geschrieben, wie es nötig ist. Lästige Fleißarbeiten, wie z.B. das Einlesen von Formularvariablen, Entgegennehmen von File-Uploads etc.werden transparent von CIPP erledigt. Das ermöglicht dem Entwickler sich vollständig auf das eigentliche Problem zu konzentrieren.

Leistungsfähigkeit von CIPP

Die CIPP-Befehle, die in HTML Seiten eingebaut werden können, decken die wichtigsten Bereiche wie Variablen, Kontrollstrukturen, Ausnahmebehandlung und Datenbankanbindung ab. Da aus einem CIPP-Programm ein Perl-Programm generiert wird, ist ein CIPP Programm vom Prinzip her so leistungsfähig wie ein Perl-Programm, d.h. alles was Perl bietet kann in CIPP Programmen ohne irgendwelche Einschränkungen eingesetzt werden.

CIPP ist aber in einer Hinsicht besonders leistungsfähig: es kann von Hause aus mit den wichtigsten Datenbanken (Oracle, Informix, Sybase u.v.m) umgehen. Es muß nur einmal festgelegt werden, wo sich die Datenbank auf dem System befindet und wie mit ihr kommuniziert werden kann. Von nun an kann mit sehr einfachen CIPP Befehlen auf die Datenbanken zugegriffen werden. Wenn dabei nur auf ANSI SQL Konstrukte und keine herstellerspezifischen Erweiterungen benutzt werden, sind die Programme sogar auf anderen Datenbanksytstemen lauffähig, ohne daß hierfür Änderungen am Quellcode notwendig werden.

Einordnung von CIPP und spirit

spirit ist die Entwicklungsumgebung, die im Zusamenhang mit der Entwicklung von CIPP entstanden ist. Der Einsatz von spirit bei der Verwendung von CIPP Technoligie ist aber optional, da CIPP auch als Apache-Modul verwendet werden

kann. Für die Erstellung großer Projekte bietet sich spirit an, da es gerade im Bereich der Projektverwaltung und Zugriff mehrerer Entwickler auf ein Projekt viele Features bietet.

Demzufolge CIPP ist ein Bestandteil von spirit. spirit ist **die** Entwicklungsumgebung, um umfangreiche Projekte mit den Programmiersprachen CIPP und Perl zu erstellen. spirit bietet die Möglichkeit über einen Webbrowser das Projekt zu entwickeln, wobei alle Komponenten, die zu einem Projekt gehören, in einer hirarchischen Struktur verwaltet werden, so daß immer ein optimaler überblick über die Projekte gewährleistet ist.

Die eigentliche Programmierung geschieht also mit CIPP, die Verwaltung des Projektes und das Erstellen und Editieren von CIPP-Objekten, realisiert spirit.

CIPP/spirit Objekttypen

Eine Applikation, die mit CIPP erstellt wird, stellt sich als eine Sammlung von Objekten dar, die miteinander kommunizieren und aufeinander wirken. Diese Objekte werden mit spirit zu einem Projekt zusammengefaßt. So sind alle Bestandteile, die zu einer webbasierten Applikation gehören, im Zugriff in einer einheitlichen Entwicklungsumgebung.

CIPP (und somit auch spirit) unterscheiden folgende Objekttypen (beim Einsatz von CIPP als Apache-Modul gibt es einige Unterschiede bezüglich der Objekttypen, siehe Kaptiel „CIPP als Apache-Modul“):

1. CGI Objekte

   CGI Programme werden in der Sprache CIPP formuliert und werden über die CGI Schnittstelle (oder auch Apache/mod_perl und Oracle Application Server) angesprochen.

2. HTML Dokumente

   Die klassische statische HTML Seite gibt es in spirit eigentlich nicht mehr. In jeder HTML Seite kann CIPP mit seinem vollen Sprachumfang eingesetzt werden. Vor allem kann auf definierte Konfigurationsvariablen zugegriffen werden, durch die dann das Aussehen einer ganzen Website zentral gesteuert wird. Auf dem Produktivsystem finden sich letzlich statische Seiten wieder, die aber von CIPP anhand dynamischer Parameter generiert wurden. Diese Vorgehensweise kombiniert einen Großteil der Flexibilität einer CGI-Anwendung mit der Geschwindigkeit einer statischen Seite, ohne daß auf kaum standardisierte Webserver-Erweiterungen wie Server Side Includes o.ä. zurückgegriffen werden muß.

3. Include Objekte

Über Include Dokumente werden wiederkehrende Aufgaben in Modulen gekapselt, die in die HTML Seiten bzw. Programme eingebunden werden. Bestimmte immer wiederkehrende Bereiche in einer Website, z.B. Header und Footer, können so zentral verwaltet werden, so daß mit wenigen Eingriffen das Erscheinungsbild einer ganzen Website beeinflußt werden kann.

4. Konfigurationen

   Konfigurationen werden in HTML- und CGI Objekten eingebunden, dienen aber speziell dazu, eigene Konfigurationsparameter zu definieren, auf die in den HTML Dokumenten und CGI Programmen zugegriffen wird. Z.B. generelle Farb- und Fonteinstellungen sollten in Konfigurationsobjekten definiert werden.

5. Bilder

   spirit verwaltet alle Bilder, die innerhalb einer Webapplikation benötigt werden. In den HTML-Seiten und CGI Programmen können die Bilder im Rahmen der Möglichkeiten von HTML eingesetzt werden.

6. Datenbankkonfigurationen

   Ein Datenbankkonfigurationsobjekt nimmt die Einstellungen auf, die zur Ansprache einer Datenbank notwendig sind. Hierzu gehören z.B. Benutzername und Passwort für die Autentifizierung gegenüber der Datenbank.

   Es werden die Systeme aller großen Datenbankhersteller unterstützt. Es gibt für die Programmiersprache Perl ein eigenes Datenbank-Konzept, um datenbankunabhängige Programme erstellen zu können. Die spirit Datenbankanbindung setzt unter anderem auf dieser Schnittstelle auf, so daß alle Datenbanken, für die es einen Perl Treiber gibt, auch von spirit und CIPP unterstützt werden.

7. SQL Programme

   spirit ermöglicht die Verwaltung aller SQL Programmdokumente, die für eine Applikation benötigt werden. SQL Programme, oder auch Teile daraus, können direkt vom Browser des Entwicklers aus auf dem Datenbankserver ausgeführt werden.

Adressierung von CIPP Objekten bei der Verwendung in spirit

CIPP Projekten liegt eine hierarchische Struktur zu Grunde, d.h. alle Objekte, die in einem Projekt verwendet werden, sind in einem Baum angeordnet, der aus Ordnern und Objekten besteht. Order enthalten Objekte sowie weitere Unterordner.

Die Adressierung der Objekte folgt folgendem Schema:

```
Projektname.Ordner1.  ...  .OrdnerN.Objekt
```

Der Projektname wird einmal beim Anlegen des Projektes festgelegt. Ordnerstrukturen können dann mit spirit dynamisch erzeugt werden. Am Ende steht immer der Name eines Objektes.

Zur eindeutigen Adressierung eines Objektes gehören also der Projektname, sowie alle Ordner, die durchlaufen werden müssen, um zu dem Objekt zu gelangen. All diese Komponenten werden durch einen Punkt voneinander getrennt.

Beispiel:

```
MSG.Login.Check
```

bezeichnet das Objekt ‚Check' im Projekt ‚MSG' im Ordner ‚Login'.

Wie man sieht, geht aus dieser Schreibweise nicht hervor, um was es sich eigentlich bei dem Objekt ‚Check' handelt. In spirit werden für die unterschiedlichen Objekttypen verschiedene Icons im Projekt-Browser verwendet. Bei der Verwendung von Objektadressen in CIPP-Programmen wird der Typ erst aus dem Kontext klar, in dem das Objekt benutzt wird. Die Namen von Objekten müssen aber typübergreifend eindeutig sein.

Adressierung von CIPP Objekten bei der Verwendung als Apache Modul

In diesem Fall werden Objekte grundsätzliche über eine URL angesprochen. Diese URL kann absolut oder realtiv sein, sie muß lediglich auf dem lokalen Apache Server exisitieren.

Zum Einsatz von CIPP als Apache Modul gibt es ein eigenes Kapitel, wo die Besonderheiten und Unterschiede diskutiert werden.

Ausführung von CIPP Programmen, die mit spirit erstellt wurden

Wenn spirit ein CGI Programm übersetzt, wird es sofort in das CGI Verzeichnis des Webservers kopiert, so daß es von dort aus gestartet werden kann.

Das obige CGI Programm 'MSG.Login.Check' kann wie folgt aufgerufen werden, wenn der Webserver www.company.com heißt und das CGI-Verzeichnis /cgi-bin:

```
http://www.company.com/cgi-bin/MSG/Login/Check.cgi
```

Sehr wahrscheinlich wird dieses Programm auch einige Eingabeparameter verarbeiten, z.B. den Benutzernamen und ein Password. Diese Parameter werden wie bei CGI Programmen üblich folgendermaßen übergeben:

```
http://www.company.com/cgi-bin/MSG/Login/
Check.cgi?username=foo&amp;password=bar
```

Innerhalb Ihres CGI-Programmes wird auf diese Parameter zugegriffen, als wenn es normale Variablen wären.

Ausführung von CIPP Programmen, unter Einsatz des Apache-Moduls

Da es beim Einsatz des Apache-Moduls keinen Unterschied zwischen Quell- und Produktivdaten gibt, ist die URL zum Ausführen des CIPP Programms identisch mit der URL der entsprechenden Quelldatei.

Zum Einsatz von CIPP als Apache Modul gibt es ein eigenes Kapitel, wo die Besonderheiten und Unterschiede diskutiert werden.

Grundsätzliche Syntaxregeln

Im folgenden werden die syntaktischen Regeln beschrieben, die die grundsätzliche Struktur von CIPP Befehlen beschreiben.

Aufbau von CIPP-Befehlen

Da CIPP Befehle in HTML Dokumente eingebettet werden, orientiert sich die Syntax von CIPP Befehlen an der Syntax von HTML.

Es gibt zwei Arten von CIPP-Befehlen, solche die für sich alleine stehen, und solche, die einen Block umschließen und somit wieder geschlossen werden müssen.

```
<?BEFEHL CIPP-Parameter ... >
```

oder

```
<?BEFEHL CIPP-Parameter ... >
  HTML bzw. CIPP-Code ...
<?/BEFEHL>
```

Jeder CIPP Befehl wird eingeleitet mit der Zeichenfolge **<?**, gefolgt von dem Namen des Befehls und endet mit dem Zeichen **>**. Vor dem Namen des Befehls dürfen auch Leerzeichen stehen. Darüberhinaus kann ein CIPP Befehl einen oder mehrere Parameter entgegennehmen, die dann mit Leerzeichen voneinander getrennt werden. Es gibt zwei Typen von Parametern: solche die einen Wert übergeben und solche die nur als Schalter fungieren, und keinen gesonderten Wert zuweisen.

Ein Parameter mit Wertzuweisung hat folgende Syntax:

```
Parameter_Name = Parameter_Wert
```

Der Wert eines Parameters wird hinter das = Zeichen gesetzt, vor und hinter dem **=** Zeichen dürfen Leerzeichen stehen. Enthält der Parameter-Wert Leerzeichen, so muß er von doppelten Anführungszeichen umgeben sein, z.B.:

```
<?BEFEHL Parameter_1=Wert_Ohne_Leerzeichen
         Parameter_2="Wert mit Leerzeichen">
```

Enthält der Parameter-Wert Leerzeichen und doppelte Anführungszeichen, so müssen die Anführungszeichen des Wertes mit einem **** maskiert werden, z.B.:

```
<?BEFEHL
 Parameter_1="Wert mit \"Anführungszeichen\"">
```

Der Wert eines Parameters darf auch beliebige Variablen enthalten, diese werden, wie in Perl auch, durch ihren Inhalt ersetzt. Eine Ausnahme bildet die Syntax für die Rückgabe von Werten durch einen CIPP-Befehl. Näheres dazu folgt weiter unten.

Ein Schalter, ohne Wertzuweisung hat diese Syntax:

```
Schalter_Name
```

Das heißt, er steht einfach für sich alleine in der Reihe der CIPP-Parameter hinter dem Befehlsnamen.

Grundsätzlich spielt bei der Angabe von Parametern die Groß- und Kleinschreibung der Bezeichner keine Rolle. Bei Parametern mit Wertzuweisung bleibt die Schreibweise der Parameter-Zuweisung natürlich erhalten.

Parameterrückgabe von CIPP-Befehlen

Es gibt eine Reihe von CIPP-Befehlen, die Werte zurückliefern. Dies geschieht immer in der Form, daß über einen Parameter die Variable übergeben wird, in die

das Ergebnis geschrieben werden soll (sozusagen *Pass By Reference*). Handelt es sich um eine skalare Variable, kann nei der Angabe dieser Variable das normalerweise vor skalaren Variablen gestellte **$** Zeichen auch weggelassen werden. Insbesondere werden diese Variablen bei der Übergabe nicht extrapoliert, die sonst übliche Inhaltsersetzung findet also nicht statt. Sie würde an dieser Stelle auch überhaupt keinen Sinn machen.

Kontext von CIPP Befehlen

CIPP Befehle werden in HTML Dokumente eingebettet. Abhänging von dem Befehl entsteht dadurch ein Kontext, der bestimmt, welche Befehle innerhalb diesem verwendet werden können.

1. HTML-Kontext

   Zunächst einmal befindet sich ein CIPP-Programm im HTML-Kontext. D.h. wenn überhaupt kein CIPP Befehl innerhalb eines CIPP-Programmes verwendet wird, handelt es sich dabei im Grunde genommen um eine einfache HTML-Seite, ohne zusätzliche Funktionalität.

   Innerhalb des HTML-Kontextes werden nur Variablenersetzungen vorgenommen. Wie das genau funktioniert, steht im folgenden Abschnitt.

   Im HTML Kontext kann belieber HTML Code verwendet werden und beliebge CIPP Befehle. Es gibt keine CIPP-Befehle, die nicht innerhalb des HTML Kontextes ungültig sind.

   Die meisten CIPP-Befehle verändern den Kontext auch nicht, es gibt nur zwei Ausnahmen:

2. Kontext einer Variablenzuweisung

   Variablen werden in CIPP über den Blockbefehl **<?VAR>** zugewiesen (siehe Befehlsreferenz). Innerhalb des **<?VAR>** Blocks dürfen keine CIPP Befehle benutzt werden. Variablenersetzungen werden hier aber vorgenommen.

3. Perl-Kontext

   Mit dem CIPP-Block-Befehl **<?PERL>** kann beliebger Perl-Code eingebunden werden. D.h. der Block wird als Perl-Code interpretiert, so daß hier keine CIPP-Befehle und auch kein HTML stehen dürfen.

Benutzung von Variablen

CIPP kennt die Formen von Variablen, die auch Perl kennt. D.h. es gibt skalare Variablen (enthalten einen einzelnen Wert), Listen (enthalten eine geordnete Menge von Scalaren) und Hashs (ordnen Skalaren Skalare zu). Darüber hinaus gibt

es in Perl auch Referenzen, die Sie selbstverständlich auch verwendet können, da sich sich als Skalare präsentieren.

Der Zugriff auf Variablen in CIPP unterscheidet sich nicht von dem in Perl. Variablen werden im HTML- und Variablenkontext durch ihren Inhalt ersetzt, dabei muß skalaren Variablen das **$** Zeichen vorangestellt werden, Listen das **@** Zeichen (alle Elemente werden ohne ein Trennzeichen direkt hintereinander ausgegeben) und Hash's das **%** Zeichen (alle Key->Value Elemente werden ohne ein Trennzeichen hintereinander ausgegeben).

Die Standardausgabe von Listen und Hashs ist aber recht unpraktisch, da die Elemente ohne Trennzeichen hintereinander gesetzt werden. Hier kann dann auf den Perl-Befehl **join** zurückgegriffen werden, um die Ausgabe sinnvoll zu formatieren.

Kommentare in CIPP

Ein Kommentar wird in CIPP von einem **#** Zeichen eingeleitet. Dabei wird dieses Symbol nur als Kommentar verstanden, wenn es sich am Anfang einer Zeile befindet, wobei führende Leer- und Tabulatorzeichen erlaubt sind. Kommentarzeilen dürfen also eingerückt sein. So kommentierte Zeilen erscheinen weder in dem generierten Perl-Script noch auf der von dem Perl-Script erzeugten Seite, im Gegensatz zu HTML Kommentaren **<!-- -->**, die in CIPP keinerlei Bedeutung haben und so ungefiltert auf der von dem Perl-Script generierten HTML Seite erscheinen.

Es ist nicht möglich inmitten einer Zeile, z.B. hinter einem CIPP- oder HTML-Befehl, mit dem **#** Zeichen einen Kommentar zu setzen. Das **#** und die dahinterstehende Zeichenkette würden normal mitinterpretiert, das bedeutet in der Regel, daß sie so wie sie sind, als Ausgabe in dem generierten HTML Dokument erscheinen.

Ebenso ist es natürlich unmöglich hinter einem Kommentar noch Befehle zu schreiben, da alle Zeichen, die hinter dem Kommentarzeichen stehen, ignoriert werden.

Folgende Zeilen sind Beispiele für gültige Kommentare:

```
<?PERL>
  # Das ist ein Kommentar, der eingerückt ist
<?/PERL>

# Das ist ein nicht eingerückter Kommentar
```

Das folgende Beispiel dokumentiert eine fehlerhafte Verwendung des Kommentarzeichens:

```
<?PERL>$perl = „/“<?/PERL> # Test-Path
```

Auf der Webseite würde

```
# Test-path
```

ausgegeben, da Kommentare nur am Anfang einer Zeile stehen dürfen.

„use strict“ - Pro und Contra

In Perl gibt es den Befehl **`use strict`**, der den Perl-Compiler anweist, das Deklarieren von Variablen zu erzwingen. D.h. es werden Compilerfehler gemeldet, wenn auf eine nicht deklarierte Variable zugegriffen wird.

Schaltet man dieses Compiler Pragma ein, so nimmt man der Scriptsprache Perl einiges ihrer Leichtfüßigkeit und zwingt sich zu strukturierterer Programmierung, zu der man bei der Verwendung einer Scriptsprache in der Regel nicht gezwungen wird.

Durch diese Anweisung zwingt sich der Programmierer also nun zur Deklaration von Variablen, was einige Vor- und Nachteile mit sich bringt.

Eine Variable wird in Perl mit dem Befehl **`my`** deklariert. Sie wird dadurch ab der Stelle in dem Block, ab der sie deklariert wird, bekannt gemacht und kann danach wie jede andere Variable benutzt werden. Außerhalb des Blocks verliert sie jegliche Gültigkeit. Variablen gleichen Namens außerhalb des Blocks werden durch die deklarierte Variable nicht beeinträchtigt. Insbesondere treten keine Konflikte mit Variablen auf, die aus anderen Quelltextmodulen kommen. Die mit **`my`** deklarierte Variable verliert also auch über Dateigrenzen ihre Gültigkeit, selbst wenn innerhalb des sie umgebenden Blockes Dateien in das Programm eingebunden werden.

Da zeigt sich auch schon der Vorteil dieser lexikalischen Variablen, wie sie laut Perl-Terminologie auch genannt werden. Sie exisitieren nur in dem Kontext (also Datei und Block), in dem sie deklariert wurden. Namensüberschneidungen mit anderen Programmteilen sind so unmöglich.

Gerade hier liegen aber die größten Fehlerquellen in Programmen. Durch die Verwendung von globalen Variablen können in größeren Programm Seiteneffekte auftreten, die nicht mehr kontrollierbar sind.

Praktisch ergibt sich durch die Verwendung von **`use strict`** der Vorteil, daß eine Fehlermeldung ausgegeben wird, wenn nicht deklarierte Variablen verwendet werden. Insbesondere Tippfehler bei Variablennamen werden dadurch sofort erkennbar. Ohne **`use strict`** ist es sehr schwer solche Tippfehler in größeren Programmen zu finden.

Das ist natürlich auch gleichzeitig ein Nachteil, denn alle Variablen müssen nun deklariert werden, was zu unnötig vielen Fehlermeldungen führen kann, wenn dies vergessen wird. Der Vorteil der erleichterten Fehlersuche überwiegt hier aber eindeutig.

Deshalb kann die Empfehlung an jeden Perl Programmierer nur lauten, **`use strict`** in seinen Programmen zu verwenden.

CIPP und use strict

CIPP generiert per Default Programme, die den Befehl **`use strict`** enthalten. Dieses Verhalten kann für jedes CIPP Programm einzeln abgeschaltet werden, indem im Eigenschaften-Dialog die Checkbox „Zwang zur Variablendeklaration“ deaktiviert wird.

Bei aktivem **`use strict`** müssen alle Variablen entweder mit dem **`<?MY>`** Befehl oder bei der Verwendung innerhalb eines CIPP Befehls mit der Option **`MY`** deklariert werden. Die Option **`MY`** gibt es bei jedem CIPP Befehl der eine Variable, z.B. für einen Rückgabewert, anlegt. Ohne die Option geht der Befehl davon aus, daß die Variable bereits existiert, sie muß also vorher bereits deklariert worden sein. Ansonsten beschwert sich der Perl-Compiler über die Verwendung der Variablen und gibt in etwa folgende Meldung aus:

```
Global symbol „variable“ requires explicit
package name at - line 42
```

Das bedeutet, die Variable mit dem Namen **`variable`** muß mit einem Package-Namen verwendet werden, d.h. konkret, sie muß als globale Variable kenntlich gemacht werden.

Globale Variablen gibt es im **`use strict`** Modus also nur als Package-Variablen:

```
$package::variable
```

wird also nie zu einer Fehlermeldung führen, da die Variable voll qualifziert ist und ihren Ursprung im Package **`package`** hat.

Für Details zu dem Umgang mit lokalen, lexikalischen Variablen und Packages sei auf die Perl man-pages oder Sekundärliteratur verwiesen.

Hinweis zu CIPP- und Perl-Compilerfehlern

Bei Fehlermeldungen muß grundsätzlich zwischen CIPP- und Perl-Compilerfehlern unterschieden werden. CIPP Fehler treten bei der Übersetzung des CIPP Quellcodes nach Perl auf und betreffen ausschließlich die CIPP Syntax. CIPP führt generell keine Kontrolle auf eingebundenen Perl-Code durch. Die hier generierten Fehlermeldungen verweisen also zusammen mit den ausgegebenen Zeilennummern auf den Original CIPP Code.

Nach der Übersetzung nach Perl wird vom Perl-Compiler eine Syntaxkontrolle durchgeführt. An dieser Stelle erst werden z.B. die **`use strict`** Verletzungen sichtbar, die Fehlermeldungen kommen nun vom Perl-Compiler und dieser bezieht sich dabei auf den von CIPP generierten Perl-Code. Die hier angegebenen Zeilennummern können also nicht zur Recherche im CIPP Programm verwendet werden.

In zukünftigen Versionen wird eine Übersetzung der Perl-Zeilennummern zu CIPP Zeilennummern erfolgen, um die Fehlersuche zu erleichtern.

CGI-Parameter bei aktivem use strict

CGI-Parameter werden ohne **`use strict`** einfach als Variablen in den Sichtbereich des Programmes eingeblendet. Bei der Verwendung von **`use strict`** ist dies nicht mehr möglich. Die CGI Schnittstelle muß entweder mit **`<?INTERFACE>`** deklariert werden, oder die Parameter müssen einzeln mit **`<?GETPARAM>`** abgeholt werden.

Includes bei aktivem use strict

Um CIPP Includes bei aktivem **`use strict`** verwenden zu können, muß von dem **`<?INCINTERFACE>`** Befehl Gebrauch gemacht werden. Das hat auch zusätzlich den Vorteil, daß das Einhalten der Schnittstelle beim Einbinden von Includes von CIPP kontrolliert wird. Bei einer Verletzung der Schnittstelle wird zur Übersetzungszeit eine Fehlermeldung ausgegeben.

use strict in persistenten Perl Umgebungen

CIPP generiert Perl-Code, der unter dem Oracle Application Server oder Apache mod_perl lauffähig ist. In diesen persistenten Perl Umgebungen sollte unter keinen Umständen auf **`use strict`** verzichtet werden. Gerade hier können unangenehme Seiteneffekte durch globale Variablen auftreten, da der Zustand des Perl-Programms über mehrere HTTP Requests hinweg erhalten bleibt. Von der Verwendung von globalen Variablen sollte spätestens hier vollständig abgesehen werden.

KAPITEL 2

CIPP als Apache Modul

In diesem Kapitel werden die Unterschiede beim Einsatz von CIPP als Apache Modul im Vergleich zur Verwendung in spirit erläutert. Ebenso wird gezeigt, wie der Apache Webserver konfiguriert werden muß, um CIPP Programme ausführen zu können.

Was ist Apache?

Apache ist ein Webserver, der sich großer Beliebtheit und Verbreitung erfreut. Sein modulares Konzept, hohe Sicherheit und Performance machen ihn zum Webserver der Wahl für große und kleine Websites. Dabei ist Apache freie Software, was sicherlich mit zu seiner großen Verbreitung geführt hat.

Für den Apache Server gibt es ein Modul (mod_perl), das den Webserver um den Perl-Interpreter erweitert, so daß viele der Funktionen des Webservers von Perl Routinen übernommen werden können. Insbesondere entfällt durch die Einbindung des Interpreters in den Webserver das bei CGI sonst übliche Aufstarten eines eigenen Prozesses zur Bearbeitung von CGI Requests. Diese Persistenz des Perl-Interpreters bringt also vor allem eine erhöhte Performance mit sich.

CIPP hat zwei Gesichter

CIPP kann in Form von zwei Varianten eingesetzt werden.

CGI Programme generieren - spirit als Entwicklungsumgebung

CIPP wurde ursprünglich dazu konzeptioniert, aus CIPP Quellcode fertige CGI Programme zu erzeugen, die dann auf einem beliebigen Webserver ausgeführt werden können. Die einzige Voraussetzung beim Webserver ist in diesem Fall, daß er Perl-Programme als CGI's ausführen kann. Somit laufen solche CIPP/CGI Programme quasi in jeder Webserver-Umgebung.

Um die Erstellung von CGI Programmen mit CIPP so komfortabel wie möglich zu machen, ist spirit als Entwicklungsumgebung für CIPP-Projekte entstanden.

CIPP als Apache Modul

Später ist dann ein zweites Einsatzgebiet für CIPP hinzugekommen: unter der Verwendung des Apache Webservers und des mod_perl Moduls kann CIPP unter Umgehung des CGI Konzeptes, CIPP Programme direkt innerhalb des Webservers ausführen.

Hierzu wird im Apache das Perl Modul Apache::CIPP_Handler als Request Handler konfiguriert. Dieses Modul übernimmt dann die Steuerung des Übersetzungsvorgangs und die Ausführung der übersetzten CIPP Programme.

Die Unterschiede: CIPP/CGI vs. CIPP/Apache

CIPP/CGI: CIPP als Präprozessor

Der Präprozessor-Charakter von CIPP wird beim Einsatz in reinen CGI Umgebungen besonders deutlich: es gibt eine Quelldatei aus der durch den CIPP Präprozessor ein lauffähiges CGI Programm erstellt wird. Das bedeutet, daß nach jeder Änderung der Quelldatei auch das entsprechende CGI Programm neu generiert werden muß. Ein weiterer Nebeneffekt dieser Vorgehensweise ist, daß sich Abhängigkeiten zwischen CIPP Programmen und anderen Objekten (insbesondere Include- und Datenbankobjekten) zur Übersetzungszeit auswirken müssen. D.h. nach dem Ändern eines Include-Objektes müssen die CGI Objekte, die dieses Include verwenden, neu übersetzt werden, damit diese immer aktuell

sind. Dies ist vergleichbar mit der Funktion, die das Compiler Tool ’make’ erledigt. All dies wird von spirit vollautomatisch und ohne Eingreifen des Entwicklers abgewickelt.

Der Vorteil dieser Arbeitsweise ist eine weitreichende Kontrolle über die Fehlerfreiheit von CIPP Programmen zur Entwicklungszeit, da z.B. das Fehlen von verwendeten Include-Objekten beim ersten Speichern schon auffällt und zu einer Fehlermeldung führt. Generell werden viele Fehler schon zur Übersetzungszeit erkannt. Ein CGI Programm wird erst dann in den Produktivbereich kopiert und „scharf“ gemacht, wenn das Programm auch korrekt ist, so daß der Produktivbereich stets konsistent und weitgehend fehlerfrei ist.

Ein weiterer Vorteil ist, daß auf dem Produktivsystem nur die übersetzten CIPP Programme installiert werden müssen. Die Quelldateien bleiben beim Entwickler. Gerade bei der Entwicklung kommerzieller Programme ein gewichtiger Vorteil.

CIPP/Apache: Transparenz und Performance

Beim Einsatz von CIPP als Apache-Modul ergeben sich einige konzeptionelle Unterschiede. Auffälligster Unterschied ist, daß die Trennung zwischen Quelll- und Produktivdateien entfällt.

Die CIPP Programme liegen, wie alle anderen WWW Dokumente auch, im Sichtbereich des Webservers und können dort über deren URL aufgerufen werden. Das Übersetzen der CIPP Programme und Ausführen im Kontext des Apache Servers geschieht völlig transparent und mit hoher Performance, da verschiedene Caching-Mechanismen diesen Vorgang beschleunigen.

Die Abhängigkeiten von CIPP Programmen zu anderen Objekten bzw. Dokumenten gibt es hier natürlich ebenso wie beim Einsatz in einer reinen CGI Umgebung. Allerdings wirken sich diese erst zur Laufzeit der Programme aus, schon alleine deswegen, weil es den Unterschied zwischen Lauf- und Übersetzungszeit eigentlich nicht mehr gibt. Fehler fallen somit also auch erst auf, wenn die entsprechenden Programme aufgerufen werden.

Da es keine Trennung mehr zwischen Quell- und Produktivdateien gibt, müssen die Quellen also immer mit auf dem Produktivsystem installiert werden. Dies ist u.U. nicht immer erwünscht, läßt sich in diesem Fall aber nicht verhindern.

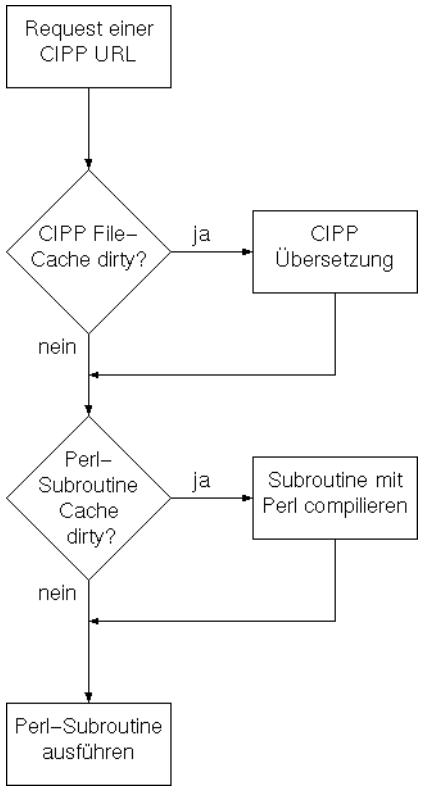
Ein weiterer Unterschied ergibt sich in der Objekt-Adressierung. Bei der Verwendung von CIPP innerhalb von spirit werden die Objekte über eine spezielle Schreibweise angesprochen, die auf der Objekthirarchie innerhalb von spirit basiert.

Beim Einsatz als Apache Modul werden Objekte grundsätzlich über eine URL adressiert, wie man es bei der Entwicklung von HTML Seiten auch gewohnt ist.

Funktionsweise von CIPP als Apache Modul

Im folgenden wird beschrieben, wie genau der Ablauf ist, wenn ein CIPP Programm von dem Apache Modul verarbeitet wird. Die Kommunikation zwischen Apache und CIPP wird vom Apache::CIPP_Handler Modul übernommen, welches auf dem System installiert sein muß.

Zunächst wird der Request vom Apache entgegengenommen und aufgrund der Mapping- und CIPP-Konfiguration als ein CIPP Request klassifiziert. Somit wird zur Beantwortung des Requests das Apache::CIPP_Handler Modul herangezogen.



Obiges Diagramm zeigt vereinfacht den Ablauf innerhalb des Apache::CIPP_Handler Moduls. Vereinfacht insofern, daß hier davon ausgegangen wird, daß das Programm sowohl auf CIPP- als auch auf Perl-Ebene fehlerfrei ist. Wenn Fehler gefunden werden, werden diese ebenfalls gecached und natürlich auch ausgegeben, damit der Entwickler eine Chance hat, sie zu korrigieren ;)

Zum grundsätzlichen Verständnis der Funktionsweise ist die Betrachtung des fehlerfreien Falls aber ausreichend.

Es gibt zwei Caches, die die Ausführung von CIPP Programmen beschleunigen. Zunächst wird der Output des CIPP Präprozessors (der generierte Perl-Code) im Dateisystem gecached, damit nicht für jeden Request erneut die CIPP Übersetzung durchgeführt werden muß. Ist dieser Cache dirty, das Originaldokument also neuer als das entsprechende „Kompilat“ im Cache, wird das CIPP Programm erneut übersetzt und das Ergebnis im Cache abgelegt.

Der zweite Cache hält das Perl Compilat des übersetzten CIPP Programms in Form einer Perl Subroutine im Speicher des Apache Prozesses, so daß der Perl Compilierlauf nur dann durchgeführt werden muß, wenn sich die Originaldatei im Dateisystem-Cache geändert hat. Ansonsten muß zur Beantwortung des Requests überhaupt kein Übersetzungsvorgang mehr initiiert werden: der Request kann sofort aus dem Hauptspeicher heraus beantwortet werden.

Wenn alle Caches up to date sind, bzw. die entsprechenden Übersetzungen durchgeführt worden sind, kann der Request beantwortet werden, indem die Subroutine ausgeführt wird.

Konfiguration des Apache Servers

Um CIPP als Apache Modul verwenden zu können, muß der Apache Webserver entsprechend konfiguriert werden.

Hierzu wird für eine bestimmte Location bzw. die URL eines Verzeichnisses festgelegt, daß alle Dokumente innerhalb des Verzeichnisses von CIPP behandelt werden sollen. Es ist auch möglich, nur Dateien mit einer bestimmten Endung von CIPP behandeln zu lassen. Ebenso ist eine Kombination von beiden Varianten möglich. Einzelheiten hierzu können der Apache Dokumentation entnommen werden.

Am einfachsten ist die Konfiguration, wenn folgende Einstellungen in der Datei access.conf des Apache eingetragen werden:

Einstellungen in access.conf:

```
<Location /CIPP>
SetHandler “perl-script“
PerlHandler Apache::CIPP_Handler

# Cache Verzeichnis für übersetzte CIPP Programme
PerlSetVar cache_dir      /tmp/cipp_cache

# Debug-Informationen ins Error-Log?
PerlSetVar debug          1

# Verwendete Datenbanken
PerlSetVar databases      art, acc

# Default Datenbank
PerlSetVar default_db     art

# Datenbankkonfiguration für Datenbank ’art’
PerlSetVar db_art_data_source  dbi:mysql:article
PerlSetVar db_art_user         webuser
PerlSetVar db_art_password     topsecret
PerlSetVar db_art_auto_commit  1

# Datenbankkonfiguration für Datenbank ’acc’
...
</Location>
```

Oben gezeigter Abschnitt muß sich irgendwo in der Datei access.conf der Apache Konfiguration befinden.

Der Parameter des **`<Location>`** Contatiners gibt die URL eines Verzeichnisses an, für das die genannten Einstellungen gültig sein sollen, in diesem Fall **`/CIPP.`**

Die ersten beiden Zeilen definieren mod_perl bzw. das Perl Modul Apache::CIPP_Handler als Request Handler für dieses Verzeichnis. Diese beiden Zeilen müssen immer so aussehen, unabhängig von anderen folgenden Konfigurationsparametern.

Es folgt nun noch einmal eine tabellarische Erklärung der einzelnen Parameter, die jeweils mit der Direktive **`PerlSetVar`** gesetzt werden müssen:

Parameter	Bedeutung
`cache_dir`	In diesem Verzeichnis werden die übersetzten CIPP Programme gespeichert. Es wird angelegt, wenn es nicht exisitiert. Das Verzeichnis muß von dem Benutzer, als der der Apache Server läuft, angelegt werden können. Entsprechende Rechte im Dateisystem müssen also gesetzt sein.
`debug`	Wenn hier 1 gesetzt ist, wird für jeden Request eine Zeile mit Debugging Informationen in das Apache Error Log geschrieben. U.a. geht daraus der Zustand der Caches hervor.
`databases`	Hier steht eine mit Komma getrennte Liste von CIPP internen Datenbank-Namen. Auf diese Bezeichner kann in CIPP Programm Bezug genommen werden, wenn eine bestimmte Datenbank angesprochen werden soll.
`default_db`	Hier muß eine der in **`databases`** aufgelisteteten Datenbanken genannt sein. Diese wird per Default angesprochen, wenn keine spezielle Datenbank bei SQL Befehlen gesetzt wird.

Die folgenden Parameter müssen jeweils für jede verwendete Datenbank angegeben werden. Dabei steht der * im Parameter-Namen für den internen Namen der entsprechenden Datenbank, wie er beim Parameter **`databases`** aufgeführt ist..

Parameter	Bedeutung
`db_*_data_source`	Hier muß die DBI Datenquelle der Datenbank angegeben werden.
`db_*_user`	Benutzername, der zur Verbindung mit der Datenbank verwendet werden soll
`db_*_password`	Password des Benutzers für die Verbindung zur Datenbank
`db_*_auto_commit`	Wenn dieser Parameter auf 1 gesetzt ist, wird die Datenbankverbindung in den Autocommit-Modus gesetzt. Siehe dazu auch die Beschreibung zum Befehl **`<?AUTOCOMMIT>`** oder die DBI Dokumentation.

KAPITEL 3

CIPP Befehlsreferenz

Dieses Kapitel beschreibt alle CIPP Befehle anhand einer ausführlichen Referenz. Die Befehle werden in alphabetischen Reihenfolge anhand einer Syntaxnotation, einem Beschreibungstext, der Beschreibung aller Parameter sowie ein oder mehreren Beispielen erläutert.

Befehlsgruppen

Zur Übersicht folgen Tabellen der Befehle, geordnet nach der Zugehörigkeit zu bestimmten Befehlsgruppen:

Variablen und Gültigkeitsbereiche

VAR	Definition einer Variablen
MY	Deklaration von blocklokalen Variablen
BLOCK	Bildung von Programmblöcken, zur Steuerung des Gültigkeitsbereichs von Variablen

Konstrollstrukturen

IF	Bedingte Ausführung eines Programmblocks
ELSIF	Fortgeführte bedingte Ausführung von Programmblöcken
ELSE	Alternatvie Ausführung eines Programmblocks nach einem <?IF> oder <?ELSIF> Befehl
WHILE	Kopfgesteuerte Schleife
DO	Fußgesteuerte Schleife
FOREACH	Schleife, die mit einer Laufvariablen über eine Liste iteriert
PERL	Einfügen reinen Perl-Codes
EXECUTE	Ausführung eines CIPP CGI Objektes mit Umleitung der Ausgabe in eine Variable oder in eine Datei
SUB	Definition einer Perl-Subroutine

Import

INCLUDE	Einfügen eines CIPP Include Objektes in den aktuellen CIPP Code
LIB	Einbinden eines Perl-Moduls
CONFIG	Einbinden einer CIPP Konfiguration

Ausnahmebehandlung

TRY	Abgesicherte Ausführung des eingeschlossenen Blocks. Dort auftretende Exceptions werden abgefangen.
CATCH	Ausführung von CIPP Code, falls eine bestimmte Exception im vorangegangenen <?TRY> Block aufgetreten ist
THROW	Explizites Erzeugen einer Exception
LOG	Schreiben eines Eintrages in ein Logfile

SQL

SQL	Ausführung eines SQL Befehls
COMMIT	Abschließen einer Transaktion
ROLLBACK	Zurückfahren einer Transaktion
AUTOCOMMIT	Steuern des Transaktionsverhaltens
DBQUOTE	Quoten einer Variablen zur Verwendung innerhalb es SQL Statements
GETDBHANDLE	Übergibt das Perl-interne Datenbankhandle

URL- und Formular-Handling

GETURL	Ermitteln der URL eines CIPP Objektes
URLENCODE	URL-encodierung einer Variablen
HTMLQUOTE	HTML quoting einer Variablen
HIDDENFIELDS	Generierung einer Reihe von HIDDEN Formularfeldern

Ersetzungen von HTML Tags

IMG	Ersetzt <IMG> Tag
A	Ersetzt <A></A> Tag
FORM	Ersetzt <FORM> Tag
INPUT	Ersetzt <INPUT> Tag
TEXTAREA	Ersetzt <TEXTAREA></TEXTAREA> Tag

Schnittstellen

INTERFACE	Deklaration einer CGI Schnittstelle für ein CIPP CGI Objekt
INCINTERFACE	Deklaration einer Schnistelle für ein CIPP Include Objekt
GETPARAM	Übernehmen eines CGI Eingabeparameters

GETPARAMLIST	Rückgabe einer Liste mit den Namen aller CGI Eingabeparameter
SAVEFILE	Speichern einer vom Client per Datei-Upload gesendeten Datei im Filesystem

Apache

APREDIRECT	Apache-Intern Redirect auf eine andere URL
APGETREQUEST	Liefert das interne Apache-Request Objekt, zur Ausführung spezifischer Apache-Funktionen

Präprozessor

AUTOPRINT	Steuert automatische Ausgabe von HTML Code

Alphabetische Referenz aller CIPP-Befehle

A

Bereich:

Ersetzungen von HTML Tags

Syntax:

```
<?A HREF=CIPP-CGI_oder_HTML-Objekt[#Sprungmarke]
    [ zusätzliche_<A>_Parameter ... ] >
...
<?/A>
```

Beschreibung:

Es wird ein HTML **<A>** Tag generiert, das als HREF auf das entsprechende CGI- oder HTML-Objekt verweist.

Parameter:

HREF
Mit **HREF** wird das Objekt angegeben, auf das mit einem Hyperlink verwiesen werden soll. Dabei kann optional mit **#** eine Sprungmarke angegeben werden, die auf ein in der Zielseite mit **<A NAME>** definiertes Label verweist.

Beim Einsatz von CIPP als Apache Modul wird an dieser Stelle statt der Objektbezeichnung eine URL erwartet.

zusätzliche_<A>_Parameter
Alle weiteren Parameter werden 1:1 in das generierte HTML **<A>** Tag übernommen, d.h. zusätzliche Parameter wie **TARGET** können so angegeben werden.

Beispiel:

Es wird ein textueller Link auf die Seite 'MSG.Main' erzeugt:

```
<?A HREF=MSG.Main>Zurück zum Hauptmenü<?/A>
```

Das Bild ‚MSG.Images.Logo' wird als Link auf die Seite ‚MSG.Main' in die HTML-Seite eingebunden. Zum Einbinden des Bildes wird der CIPP Befehl **<?IMG>** verwendet. Dabei wird die **BORDER** des Bildes auf 0 gesetzt:

```
<?A HREF=MSG.Main>
<?IMG SRC=MSG.Images.Logo BORDER=0>
<?/A>
```

APGETREQUEST

Bereich:

Apache

Syntax:

<?**APGETREQUEST** [**MY**] **VAR**=*Request_Variable* >

Beschreibung:

Dieser Befehl macht nur beim Einsatz im Apache-Webserver Sinn.

<?APGETREQUEST> gibt das Apache-Request Objekt für den aktuell zu bearbeitenden Request zurück. Anhand dieses Objektes können dann spezielle Apache Funktionen ausgeführt werden.

Hinweise zur Verwendung des Request-Objektes können der Dokumentation zum Perl Apache Modul entnommen werden.

Parameter:

VAR
Der Parameter **VAR** benennt die Variable, in der das Apache Request Objekt gespeichert werden soll.

MY
Ist der **MY** Schalter gesetzt, wird die erzeugte Variable implizit mit my deklariert, so daß sie nur bis zum Ende des den **APGETREQUEST** Befehl umgebenden Blocks bekannt ist.

Beispiel:

Hier wird das Request Objekt in die gleichzeitig deklarierte Variable **$ar** übergeben:

```
<?APGETREQUEST MY VAR=$ar>
```

APREDIRECT

Bereich:

Apache

Syntax:

<?**APREDIRECT URL**=*neue_URL* >

Beschreibung:

Dieser Befehl macht nur beim Einsatz im Apache-Webserver Sinn.

APREDIRECT erzeugt einen Apache internen Redirect auf eine andere URL. Der Client nimmt diese interne Umleitung auf eine andere URL nicht wahr, im Gegensatz zu einer URL-Umleitung, die über den Code 302 des HTTP Protokolls realisiert werden kann.

Parameter:

URL
Der Parameter **URL** übergibt die URL, auf die umgeleitet werden soll.

Hinweis:

<?APREDIRECT> sollte nur in CIPP Programmen eingesetzt werden, die den **<?AUTOPRINT OFF>** Befehl abgesetzt haben. Ansonsten können schon Ausgaben aus der Bearbeitung des aktuellen Requests an den Client gesendet worden sein, bevor die neue URL bearbeitet wird. Folglich, darf das Programm, welches den Redirect initiiert, selber keine Ausgabe erzeugen.

Beispiel:

Folgende Befehle alleine in einer Datei leiten intern auf die Startseite der entsprechenden Website um:

```
<?AUTOPRINT OFF>
<?APREDIRECT URL="/">
```

AUTOCOMMIT

Bereich:

SQL

Syntax:

```
<?AUTOCOMMIT ( ON | OFF )
             [ DB=Name_der_Datenbank ]
             [ THROW=Ausnahme ] >
```

Beschreibung:

Mit dem **AUTOCOMMIT**-Befehl kann der Autocommit-Modus einer Datenbankverbindung eingestellt werden.

Ist der Autocommit-Modus aktiv, so wird jede SQL Anweisung implizit in einer eigenen Transaktion ausgeführt. Weder **<?COMMIT>** noch **<?ROLLBACK>** können im Autocommit-Modus verwendet werden.

Jede Anweisung wird bei aktivem Autocommit bei Erfolg ausgeführt (**COMMIT**) und bei einem Fehler zurückgefahren (**ROLLBACK**).

Der Autocommit-Modus einer Datenbankverbindung darf nur verändert werden, wenn diese sich nicht innerhalb einer Transaktion befindet.

Es gibt Datenbanken, die keine Transaktionen beherrschen. Bei diesen kann der Autocommit-Modus nicht deaktiviert werden.

Parameter:

ON | OFF
Autocommit kann mit den Schalterm **ON** aktiviert und mit **OFF** deaktiviert werden. Die beiden Optionen dürfen nicht im selben **<?AUTOCOMMIT>** Befehl verwendet werden.

DB
Der Parameter DB bestimmt die Datenbank, deren Autocommit Modus verändert werden soll.

THROW

Wird der Parameter **THROW** angegeben, wird die entsprechende Ausnahme im Fehlerfall generiert. Defaultmäßig wird die Ausname **autocommit** generiert.

Beispiel:

Der Autocommit-Modus im Zugriff auf die Datenbank ‚MSG.DB' wird aktiviert:

```
<?AUTOCOMMIT ON DB=MSG.DB>
```

Der Autocommit-Modus für die Default-Datenbank wird beendet. Falls die Anweisung fehlschlägt, wird die Ausnahme ‚AUTOCOMMIT_Exception' erzeugt:

```
<?AUTOCOMMIT OFF THROW="AUTOCOMMIT_Exception">
```

AUTOPRINT

Bereich:

Präprozessor

Syntax:

<?**AUTOPRINT OFF** >

Beschreibung:

Mit dem **AUTOCOMMIT**-Befehl kann das Standardverhalten des Präprozessor bezüglich der Generierung von Befehlen zur automatischen Ausgabe von HTML Code Blöcken gesteuert werden.

Parameter:

OFF
Mit der Option **OFF** wird die standardmäßige Ausgabe von HTML Code Blöcken abgeschaltet.

Hinweis:

Dieser Befehl funktioniert nur dann einwandfrei, wenn er am Anfang des Dokumentes steht. Auch Leerzeilen oder -zeichen davor sind nicht erlaubt. **<?AUTOPRINT OFF>** bewirkt ja, daß keinerlei Ausgaben automatisch getätigt werden. Das schließt auch die Ausgabe des HTTP-Headers mit ein. Dieser muß nun also auch von dem CIPP Programm innerhalb eines **<?PERL>** Blocks selbstätig ausgegeben werden.

Zur Zeit ist nur die Option **OFF** implementiert, ein Einschalten des **AUTOPRINT** Modus ist nicht möglich. Dies wird in zukünftigen Versionen u.U. noch realisiert.

Beispiel:

Hier wird eine GIF Datei ausgegeben. Hierzu muß der entsprechende HTTP Header ausgegeben werden, sowie die GIF Datei selber.

Ohne **<?AUTOPRINT OFF>** würden alle Zeilen, die nicht innerhalb von CIPP Tags stehen, autmatisch mit ausgegeben. Somit würde die Ausgabe nicht mehr den reinen Inhalt der GIF Datei enthalten und der Browser könnte das Bild aufgrund dieser korrupten Daten nicht anzeigen.

```
<?AUTOPRINT OFF>
Diese Zeilen hier werden niemals ausgegeben.
Sie werden völlig ignoriert.

<?PERL>
  my $file = "/tmp/image.gif";
  my $size = -s $file;

  print "Content-type: image/gif\n";
  print "Content-length: $size\n\n";

  open (GIF, $file) or die "can't open $file";
  while (<GIF>) {
    print;
  }
  close GIF;
<?/PERL>
```

BLOCK

Bereich:

Variablen- und Gültigkeitsbereiche

Syntax:

```
<?BLOCK>
...
<?/BLOCK>
```

Beschreibung:

Mit dem **BLOCK** Befehl können Programmteile zusammengefaßt werden. Variablen, die innerhalb des Blocks, der durch den **BLOCK**-Befehl gebildet wird, als privat deklariert wurden (z.B. mit dem Befehl **<?MY>**), sind auch nur innerhalb dieses Blocks bekannt. Außerhalb dieses Blocks hat das Programm keinen Zugriff auf diese Variablen.

BLOCK-Befehle dürfen ineinander geschachtelt werden.

Beispiel:

Die Variable **$beispiel** ist nur innerhalb des gebildeten Blocks bekannt.

```
<?BLOCK>
  <?MY $beispiel>
  Hier ist $beispiel bekannt.
<?/BLOCK>
Hier ist $beispiel nicht bekannt.
```

CATCH

Bereich:

Ausnahmebehandlung

Syntax:

```
<?CATCH [ THROW=Ausnahme ]
        [ MY ]
        [ EXCVAR=Variable_für_Ausnahme ]
        [ MSGVAR=Variable_für_Fehlermeldung ] >
...
<?/CATCH>
```

Beschreibung:

Typischerweise folgen <?CATCH> Befehle auf einen <?TRY> Block. Ein <?CATCH> Befehl behandelt eine bestimmte oder alle Ausnahmen, die innerhalb des davor stehenden <?TRY> Blocks abgesetzt wurden. Wichtig ist dabei, daß der <?TRY> Block und der <?CATCH> Befehl im selben Block stehen. Z.B. können mit <?CATCH > keine Ausnahmen behandelt werden, die aus einem <?TRY> Block stammen, der im Block eines <?IF> Befehls steht. Folgendes funktioniert also **nicht**:

```
<?IF COND="$event eq 'new'">
  <?TRY>
    <?THROW THROW=unimplemented>
  <?/TRY>
<?/IF>
<?CATCH>
  # Bis hier kommen keine Ausnahmen durch
  # Sie werden de fakto nie behandelt
<?/CATCH>
```

Parameter:

THROW
Wird **THROW** angegeben, so wird nur die hier übergebene Ausnahme von diesem **<?CATCH>** Befehl behandelt. Fehlt **THROW**, so wird der **<?CATCH>** Block für jede aufgetretene Ausnahme ausgeführt.

EXCVAR
Die hier angegebene skalare Variable nimmt die Bezeichnung der Ausnahme auf.

MSGVAR
Die hier angegebene skalare Variable nimmt die Meldung der Ausnahme auf.

MY
Wird der **MY** Schalter angegeben, so werden die bei **EXCVAR** und **MSGVAR** angegebenen Variablen implizit mit my deklariert und sind somit nur bis zum Ende des den **<?CATCH>** Block umgebenden Blockes bekannt.

Beispiel:

Hier wird versucht, einen Datensatz in eine Datenbank einzufügen. Danach wird das **INSERT** Statement commited, d.h. die Eintragung wird in der Datenbank persistent gemacht. Es können zwei verschiedene Ausnahmen auftreten, zum einem beim **INSERT** und zum anderen beim **COMMIT**. Jede Ausnahme hat einen eigenen **<?CATCH>** Block. Dabei werden die Standardnamen der entsprechenden Ausnahmen verwendet, da bei den beiden Datenbankbefehlen keine **THROW** Parameter mitgegeben wurden. Der zweite **<?CATCH>** Befehl hat keinen **MY** Parameter, da die Variable **$message** schon durch den ersten **<?CATCH>** Befehl deklariert wurde.

```
<?TRY>
  <?SQL SQL="insert into personen
            (vorname,nachname)
             values ('Hannes', 'Borgmann');"><?/SQL>
  <?COMMIT>
<?/TRY>

<?CATCH THROW=sql MY MSGVAR=$message>
  <?LOG MSG="Konnte Daten nicht einüegen: $message"
        TYPE="Datenbank-Fehler">
<?/CATCH>

<?CATCH THROW=commit MSGVAR=$message>
  <?LOG MSG="Kein COMMIT möglich: $message"
        TYPE="Datenbank-Fehler">
<?/CATCH>
```

COMMIT

Bereich:

SQL

Syntax:

```
<?COMMIT [ DB=Name_der_Datenbank ]
         [ THROW=Ausnahme ] >
```

Beschreibung:

Der **<?COMMIT>** Befehl beendet die aktuelle Transaktion und übernimmt dabei alle Änderungen in die Datenbank. Eine Transaktion beginnt beim ersten **<?SQL>** Befehl und endet jeweils immer nach einem **<?COMMIT>** oder **<?ROLLBACK>** Befehl, es sei denn die Datenbank befindet sich im Autocommit-Modus.

Parameter:

DB
Der Parameter **DB** bestimmt die Datenbank, an die der COMMIT Befehl abgesetzt werden soll. Wenn dieser Parameter weggelassen wird, bezieht sich die Abfrage auf die im Projekt voreingestellte Default-Datenbank.

THROW
Die hier angegebene Ausnahme wird im Fehlerfall generiert. Ohne die Angabe von **THROW** wird defaultmäßig die Ausnahme **commit** generiert.

Beispiel:

Ein Datensatz wird in die Datenbank eingetragen und mit dem **<?COMMIT>** Befehl direkt bestätigt. Falls der **<?COMMIT>** Befehl nicht erfolgreich sein sollte, wird eine Ausnahme mit dem Namen **COMMIT_Exception** generiert.

```
<?AUTOCOMMIT DB=MSG.DB OFF>
<?SQL SQL="insert into foo (num, str)
           values (42, 'bar');"
      DB="MSG.DB">
<?/SQL>
<?COMMIT THROW="COMMIT_Exception">
```

CONFIG

Bereich:

Import

Syntax:

```
<?CONFIG NAME=Name_des_Konfigurationsobjektes
         [ RUNTIME ] [ NOCACHE ]
         [ THROW=Ausnahme ] >
```

Beschreibung:

Mit dem **<?CONFIG>** Befehl bewirkt das Laden eines Konfigurationsobjektes zur Laufzeit des Programmes. Es besteht somit die Möglichkeit, allgemeine Parameter in solchen Objekten auszulagern, so daß diese Parameter mehreren Programmen zentral zur Verfügung gestellt werden.

Parameter:

NAME
Gibt den projektinternen Namen des Konfigurationsobjektes an.

Beim Einsatz von CIPP als Apache Modul wird an dieser Stelle statt der Objektbezeichnung eine URL erwartet.

RUNTIME
Wird dieser Schalter angegeben, so wird der Name des Konfigurationsobjektes erst zur Laufzeit ausgewertet. In diesem Fall kann bei der Übersetzung nicht geprüft werden, ob das angegebene Konfigurationsobjekt wirklich exisitiert.

Wenn der **RUNTIME** Schalter gesetzt ist, kann beim **NAME** Parameter auch eine Variable bzw. eine Zeichenkette, die eine Variable enhtält, angegeben werden. Der Inhalt der Variablen wird dann zur Laufzeit eingesetzt.

NOCACHE

Diese Option wirkt sich nur in persistenten Perl-Umgebungen aus (Oracle Application Server, Apache/mod_perl).

Normalerweise werden Konfigurationsdateien nur einmal pro Perl-Instanz eingelesen. Alle folgenden Requests, die von einer Perl-Instanz bearbeitet werden, lesen die Konfigurationsdatei nicht erneut ein. Dieses Verhalten ist aus Performancegründen erwünscht, wenn die Konfigurationsdatei nur statisch Parameter definiert.

Wenn aber Konfigurationsparameter dynamisch erzeugt werden (z.B. in Abhängigkeit von der Browser-Software oder Betriebssystems des Clients), darf dieses Caching nicht erfolgen. Mit **NOCACHE** werden also Konfigurationsdateien bei jedem Request verarbeitet.

THROW

Gibt die Ausnahme an, die generiert werden soll, wenn der Name nicht aufgelöst werden kann. Per Default wird die Ausnahme **config** generiert.

Hinweis:

In nicht persistenten Perl Umgebungen wirkt sich die Angabe von **NOCACHE** natürlich nicht aus, da dort für jeden Request ein eigener Prozeß erzeugt wird und somit die Konfigurationsdatei auch zwangsläufig für jeden Request verarbeitet wird.

Beispiel:

Hier wird die Konfigurationsdatei ‚MSG.Config' eingebunden:

```
<?CONFIG NAME="MSG.Config">
```

Hier erfolgt die Einbindung dynamisch. Der Name steht in der Variablen **$config_name**. Es wird die Ausnahme **Konfiguration** generiert, wenn der Name ungültig ist.

```
<?VAR MY NAME=$config_name>MSG.Config<?/VAR>
<?CONFIG NAME=$config_name RUNTIME
         THROW="Konfiguration„>
```

DBQUOTE

Bereich:

SQL

Syntax:

```
<?DBQUOTE VAR=Variable
          [ MY ]
          [ DBVAR=Variable_für_das_Ergebnis ]
          [ DB=Name_der_Datenbank ] >
```

Beschreibung:

Der Befehl **DBQUOTE** konvertiert eine Zeichenkette in eine Datenbankzeichenkette. Dies ist notwendig, damit bestimmte Zeichen, die innerhalb einer SQL Anweisung eine besondere Bedeutung haben, z.B. das einfache Anführungszeichen, maskiert werden müssen, sonst kann es zu Syntaxfehlern bei der Ausführung kommen.

Das Ergebnis der Operation ist entweder der String „NULL“, wenn die Variable leer war, oder der maskierte Inhalt der Variablen, mit umschließenden einfachen Anführungszeichen, z.B. wird aus >Hallo< dann >‘Hallo‘<. Innerhalb eines SQL Statements dürfen also keine Anführungszeichen um die Variable gesetzt werden (siehe Beispiel).

Hinweis:

Es gibt einen besseren Weg SQL Anweisungen zu parametrisieren, der den Einsatz von **<?DBQUOTE>** überflüssig macht. SQL Anweisungen können Platzhalter enthalten, für die übergebene Variablen automatisch eingesetzt werden. Bei dieser Vorgehensweise ist das Quoten von Zeichenketten unnötig. Eine Diskussion zun Thema Platzhalter findet sich bei der Beschreibung des **<?SQL>** Befehls.

Parameter:

VAR
Der Parameter **VAR** übergibt die skalare Variable, die bearbeitet werden soll.

DBVAR

Der optionalen Parameter **DBVAR** gibt die skalare Zielvariable an.

Wird der Parameter **DBVAR** weggelassen, so steht das Ergebnis der Operation in der Variablen **$db_VAR**, d.h. es wird implizit das Präfix **db_** vor den mit **VAR** angegebenen Variablennamen gesetzt.

MY

Ist der **MY** Schalter gesetzt, wird die erzeugte Zielvariable implizit mit my deklariert, so daß sie nur bis zum Ende des den **DBQUOTE** Befehl umgebenden Blocks bekannt ist.

DB

Der Parameter **DB** bestimmt die Datenbank, für die die Variable bestimmt ist. Wird **DB** weggelassen, so wird die Variable für den Einsatz bei der Default Datenbank bearbeitet.

Beispiel:

Die Zeichenkette, die in der Variable $name steht, wird wie oben beschrieben konvertiert. Das Ergebnis der Konvertierung wird implizit der Variablen **$db_name** zugewiesen.

```
<?DBQUOTE VAR="$name">
```

Die Zeichenkette, die in der Variable **$name** steht, wird wie oben beschrieben konvertiert. Das Ergebnis der Konvertierung wird der Variable **$maskierter_name** zugewiesen. Zusätzlich wird angegeben, auf welche Datenbank sich die Konvertierung beziehen soll.

```
<?DBQUOTE VAR="$name" DBVAR="$maskierter_name"
          DB="MSG.DB">
```

Enthielte die Variable $name zum Beispiel die Zeichenkette „wie geht‘s?“, so wird diese in die Zeichenkette „‘wie geht‘‘s?‘“ umgewandelt. In einem INSERT-Befehl könnte die Variable dann wie folgt verwendet werden:

```
<?SQL SQL="insert into personen (nachname)
           values ( $maskierter_name );">
<?/SQL>
```

DO

Bereich:

Kontrollstrukturen

Syntax:

```
<?DO>
...
<?/DO COND=Bedingung >
```

Beschreibung:

Realisierung einer fußgesteuerten Schleife. Der Schleifenkörper bzw. der **<?DO>** Block wird mindestens einmal ausgeführt. Wenn die Bedingung erfüllt ist, wird **<?DO>** Block solange wiederholt, bis die Bedingung einen falschen Wert liefert. In diesem Fall wird der Block verlassen, und die Anweisungen nach dem Block bearbeitet. **<?DO>** Blöcke dürfen ineinander geschachtelt werden.

Parameter:

COND
Dieser Parameter des **<?/DO>** Befehls gibt die Perl-Bedingung für den **<?DO>** Block an. Solange diese Bedingung wahr ist, wird der **<?DO>** Block wiederholt.

Beispiel:

Die Ausgabe der Zeichenkette ‚Hello World!' und der folgende Trennstrich wird mindestens einmal und höchstens **$i** mal ausgegeben.

```
<?DO>
  Hello World!<BR>
<?/DO COND="$i-- > 0">
```

ELSE

Bereich:

Kontrollstrukturen

Syntax:

<?**ELSE**>

Beschreibung:

Die **<?ELSE>** Anweisung darf nur innerhalb des **<?IF>** Blocks oder nach einem **<?ELSIF>** Befehl stehen. **<?ELSE>** beendet einen **<?IF>** Block, das heißt, wenn die Bedingung des **<?IF>** bzw. **<?ELSIF>** Befehls nicht erfüllt ist, wird der **<?ELSE>** Block ausgeführt.

Generell darf der **<?ELSE>** Befehl nur einmal in einem **<?IF>** Block verwendet werden.

Beispiel:

Nur Larry bekommt eine persönliche Begrüßung.

```
<?IF COND="$name eq 'Larry'">
  Hi Larry!
<?ELSE>
  Guten Tag, Sie kenne ich leider nicht!
<?/IF>
```

ELSIF

Bereich:

Kontrollstrukturen

Syntax:

<?**ELSIF COND**=*Bedingung* >

Beschreibung:

Der **<?ELSIF>** Befehl, darf nur innerhalb eines **<?IF>** Blocks stehen. Der **<?ELSIF**> Block wird nur ausgeführt, wenn die **<?IF>** Bedingung falsch und die **<?ELSIF>** Bedingung wahr ist.

Der **<?ELSIF>** Befehl darf mehrfach hintereinander in einem **<?IF>** Block aber nicht hinter einem **<?ELSE>** Block benutzt werden.

Parameter:

COND
Dieser Parameter gibt die Perl-Bedingung für den **<?ELSIF>** Block an. Wenn diese Bedingung wahr ist, wird der **<?ELSIF>** Block ausgeführt.

Beispiel:

Larry und Linus werden persönlich begrüßt.

```
<?IF COND=“$name eq ’Larry’“>
  Hi Larry!
<?ELSIF COND=“$name eq ’Linus’“>
  Hi Linus!
<?ELSE>
  Guten Tag, Sie kenne ich leider nicht!
<?/IF>
```

EXECUTE

Bereich:

Kontrollstrukturen

Syntax:

```
<?EXECUTE NAME=CIPP-CGI-Objekt
          [ [ MY ] VAR=Ergebnisvariable |
            FILENAME=Dateiname ]
          [ THROW=Ausnahme ]
          [ CGI_Parameter=Wert ] ... >
```

Beschreibung:

Der <?EXECUTE> Befehl ermöglicht die Ausführung eines anderen CGI Objektes, wobei definiert werden kann, ob die Ausgabe des CGI Objektes in eine Datei oder eine Programm-Variable umgeleitet werden soll.

Es ist möglich, Parameter an das CGI Objekt zu übergeben. Diese Parameter werden dem CGI Objekt über die CGI Schnittstelle übergeben, das aufgerufene CGI Objekt muß also keine besonderen Vorkehrungen für den Aufruf durch den <?EXECUTE> Befehl treffen.

Zur Zeit kann <?EXECUTE> nicht im Apache verwendet werden.

Parameter:

NAME
Name des auszuführenden CGI Objektes.

VAR | FILENAME
Nur einer dieser beiden Parameter kann angegeben werden. Wird **VAR** angegeben, so wird die Ausgabe des CGI Objektes in die angegebene skalare Variable geschrieben.

Wird der Parameter **FILENAME** übergeben, so wird die Ausgabe in die entsprechende Datei umgeleitet.

MY

Der **MY** Schalter kann nur gesetzt werden, wenn die Ausgabe in eine Variable umgeleitet wird. Wenn **MY** gesetzt ist, wird die Variable implizit mit **my** deklariert, so daß sie nur bis zum Ende des den **<?EXECUTE>** Befehl umgebenden Blocks bekannt ist.

THROW

Wird der Parameter **THROW** angegeben, wird die entsprechende Ausnahme im Fehlerfall generiert. Defaultmäßig wird die Ausname **execute** generiert.

CGI_Parameter

Es können beliebige zusätzliche Parameter an das CGI Objekt übergeben werden, indem diese einfach als Parameter in den **<?EXECUTE>** Befehl mit aufgenommen werden.

Hinweis:

Der **<?EXECUTE>** Befehl wirkt sich zur Laufzeit aus, d.h. das bezeichnete CGI Objekt wird zur Laufzeit geladen, compiliert und ausgeführt. Der **<?EXECUTE>** Befehl sollte also mit gewisser Vorsicht verwendet werden, da er zur Laufzeit zusätzliche Resourcen verbraucht. Zur Modularisierung eines Projektes ist **<?EXECUTE>** deshalb nicht geeignet, dieses sollte über den **<?INCLUDE>** Befehl realisiert werden.

<?EXECUTE> ist ideal dazu geeignet, Musterseiten mit bestimmten Parametern aufzurufen, so daß daraus statische Seiten entstehen, die im htdocs Tree des Webservers abgelegt werden können. Bei der Programmierung der Templates kann dann auf den vollen CIPP Sprachumfang zurückgegriffen werden.

Beispiel:

Das CIPP-CGI-Objekt ‚MSG.Template.Muster' wird mit zwei Parametern aufgerufen und die von diesem Programm erzeugt Ausgabe in die Datei

/usr/www/htdocs/Muster/Seite1.html

geschrieben:

```
<?EXECUTE NAME="MSG.Template.Muster"
     FILE="/usr/www/htdocs/Muster/Seite1.html"
     TITLE="Musterseite"
     MAILTO="spirit@dimedis.de">
```

FOREACH

Bereich:

Kontrollstrukturen

Syntax:

```
<?FOREACH [ MY ] VAR=Laufvariable
          LIST=Perl-Liste >
...
<?/FOREACH>
```

Beschreibung:

Der <?FORACH> Befehl setzt die Perl foreach Schleife um. Dabei durchläuft eine Laufvariable eine Liste von Werten und der <?FOREACH> Block wird für jeden Wert aus der Liste einmal ausgeführt.

Parameter:

VAR
Der VAR Parameter gibt die skalare Variable an, die als Laufvariable verwendet werden soll und für jeden Durchlauf den jeweils nächsten Wert aus der Liste annimmt.

LIST
Mit dem LIST Parameter wird eine Perl Liste übergeben. Hier sind alle Listen-Notationen, die Perl erlaubt, möglich. Es kann ein Array (mit vorangestelltem @) angegeben werden. Es ist ebenso möglich eine Liste von konstanten Werten oder auch Variablen, getrennt durch Komma, anzugeben (siehe Beispiel).

MY
Wird der MY Schalter gesetzt so wird die Laufvariable implizit mit my deklariert, so daß sie bis zum Ende des den <?FOREACH> Block umgebenden Blocks bekannt ist.

Beispiel:

Hier wird bis drei gezählt.

```
<?FOREACH MY VAR=$i LIST=“,eins‘, ,zwei‘, ,drei‘“>
  <P>Ich zähle: $i
<?/FOREACH>
```

FORM

Bereich:

Ersetzungen von HTML Tags

Syntax:

```
<?FORM ACTION=CGI-Objekt
       [ zusätzliche_<FORM>_Parameter ... ] >
...
<?/FORM>
```

Beschreibung:

Es wird ein HTML **<FORM>** Tag generiert, das als **ACTION** auf das entsprechende CGI-Objekt verweist.

Parameter:

ACTION
Name des CGI Objektes, das durch dieses Formular aufgerufen werden soll.

Beim Einsatz von CIPP als Apache Modul wird an dieser Stelle statt der Objektbezeichnung eine URL erwartet.

zusätzliche_<FORM>_Parameter
Alle weiteren Parameter und Schalter werden wie sie sind in das generierte **<FORM>** Tag übernommen, so daß z.B. ein **TARGET** mit angegeben werden kann.

Wird der Parameter **METHOD** nicht angegeben, so wird defaultmäßig **METHOD=POST** generiert.

Beispiel:

Es wird ein Formular mit Submit-Button generiert, das das CGI-Objekt ‚MSG.Secure.Messager' via POST (Defaulteinstellung) aufruft und dessen HTML-Name ‚Formular' ist:

```
<?FORM ACTION=MSG.Secure.Messager NAME="Formular">
<?INPUT TYPE=SUBMIT VALUE=" Start ">
<?/FORM>
```

GETDBHANDLE

Bereich:

SQL

Syntax:

```
<?GETDBHANDLE [ DB=Datenbank ] [ MY ]
              VAR=Variable_fuer_das_Handle >
```

Beschreibung:

Dieser Befehl gibt ein Handle auf das Perl-interne Datenbank Objekt zurück. Dadurch ist es möglich, Features des darunter liegenden Datenbanktreibers zu nutzen, die CIPP nicht unterstützt. Durch die direkte Ansprache des Datenbanktreibers ist natürlich nicht gewährleistet, daß dieser Programmteil auch auf anderen Datenbankarchitekturen lauffähig ist.

Parameter:

VAR
Der Parameter **VAR** gibt die skalare Variable an, in die das Handle gespeichert werden soll.

DB
Der Parameter **DB** bestimmt die Datenbank, deren Handle zurückgegeben werden soll. Wenn dieser Parameter weggelassen wird, wird das Handle der im Projekt voreingestellten Default-Datenbank zurückgegeben.

MY
Optional kann der Schalter **MY** gesetzt werden. In diesem Fall wird die Variable implizit mit **my** deklariert, so daß sie nur bis zum Ende des den **<?GETDBHANDLE>** Befehl umgebenden Blocks bekannt ist.

Hinweis:

Was für eine Art von Handle durch diesen Befehl zurückgegeben wird, hängt davon ab, mit welchem Datenbanktreiber die entsprechende Datenbank angesteuert wird. Welche Funktionen über das Handle ausgeführt werden können, hängt also stark von der verwendeten Datenbank ab. Zur Zeit unterstützt spirit DBI/DBD und Sybperl als Datenbanktreiber.

Beispiel:

Es wird das Handle der Datenbank 'MSG.Oracle' in die Variable **$dbh** übernommen, die gleichzeitig an dieser Stelle mit **my** deklariert wird. Anschließend wird ein **SELECT** Statement innerhalb eines **<?PERL>** Blocks durchgeführt. Dabei wird davon ausgegangen, daß der verwendete Treiber DBI/DBD konform ist. Zur Erläuterung des **<?PERL>** Code-Abschnitts wird auf die DBI Dokumentation verwiesen.

```
<?GETDBHANDLE DB=MSG.Oracle MY VAR=$dbh>

<?PERL>

  my $sth = $dbh->prepare ( qq{
      select n,s from TEST_table
      where n between 10 and 20
  });
  die „mein_sql\t$DBI::errstr“ if $DBI::errstr;

  $sth->execute;
  die „mein_sql\t$DBI::errstr“ if $DBI::errstr;

  my ($n, $s);
  while ( ($n, $s) = $sth->fetchrow ) {
    print „n=$n s=$s<BR>\n“;
  }
  $sth->finish;
  die „mein_sql\t$DBI::errstr“ if $DBI::errstr;

<?/PERL>
```

Anmerkung:

An diesem Beispiel ist sehr gut zu erkennen, wieviel Arbeit CIPP dem Datenbankprogrammierer abnimmt. Bei Verwendung reinen CIPP Codes liest sich obiges Beispiel wie folgt:

```
<?SQL DB=MSG.Oracle THROW=“mein_sql“
      SQL=“select n,s from TEST_table
           where n between 10 and 20“
      MY VAR=“$n, $s“>
  n=$n s=$s<BR>
<?/SQL>
```

GETPARAM

Bereich:

Schnittstellen

Syntax:

```
<?GETPARAM NAME=Parameter-Name
           [ MY ] [ VAR=Variable ] >
```

Beschreibung:

Mit diesem Befehl kann explizit ein CGI Parameter geholt werden. **<?GETPARAM>** muß z.B. verwendet werden, um Parameter zu empfangen, deren Name sich erst zur Laufzeit ergibt, so daß diese nicht mit dem **<?INTERFACE>** Befehl deklariert werden können.

Parameter:

NAME
Name des CGI Parameters

VAR
Gibt den Variablennamen an, in den der CGI Parameter übernommen werden soll. Hier sind skalare und Listenvariablen erlaubt. Mit Listenvariablen können z.B. Mehrfach-Listen-Selektionen übernommen werden.

Fehlt der VAR Parameter, so wird davon ausgegangen, daß es sich um einen skalaren Parameter handelt. Dieser wird dann in eine Variable geschrieben, die denselben Namen erhält, wie der CGI Parameter.

MY
Wir der optionale Schalter **MY** angegeben, so wird die Zielvariable implizit mit my deklariert, so daß sie bis zum Ende des den **<?GETPARAM>** Befehl umgebenden Blocks bekannt ist.

Beispiel:

Es werden zwei Parameter übernommen. Ein skalarer Parameter mit dem Namen 'event' der dem Defaultverhalten nach der Variablen **$event** zugewiesen wird. Diese wurde zuvor mit **<?MY>** deklariert. Der zweite Parameter 'selectlist' wird der Listen-Variablen **@liste** zugewiesen, die implizit an dieser Stelle mit **my** deklariert wird.

```
<?MY $event>
<?GETPARAM NAME=event>
<?GETPARAM NAME=selectlist MY VAR=@liste>
```

GETPARAMLIST

Bereich:

Schnittstellen

Syntax:

<?**GETPARAMLIST** [**MY**] **VAR**=*Variable* >

Beschreibung:

Die Namen aller über die CGI Schnittstelle übergebenen Parameter werden in einer Listenvariablen zurückgegeben.

Parameter:

VAR
Gibt die Listenvariable an, in die die Namen der CGI Parameter geschrieben werden soll. Diese Variable muß mit einem vorangestelltem **@** angegeben werden.

MY
Optional kann der Schalter **MY** gesetzt werden. In diesem Fall wird die Variable implizit mit **my** deklariert, so daß sie nur bis zum Ende des den **<?GETPARAMLIST>** Befehl umgebenden Blocks bekannt ist.

Beispiel:

Die Liste der CGI-Parameter wird in die Variable **@paramlist** geschrieben, die gleichzeitig an dieser Stelle mit **my** deklariert wird.

```
<?GETPARAMLIST MY VAR=@paramlist>
```

GETURL

Bereich:

URL- und Formular-Handling

Syntax:

```
<?GETURL NAME=CIPP-CGI-Objekt
         [ MY ] URLVAR=URL-Variable
         [ RUNTIME ] [ THROW=Ausnahme ] >
         [ PARAMS=Parametervariablen ]
         [ PAR_1=Wert_1 ... PAR_n=Wert_n ] >
```

Beschreibung:

Der <?GETURL> Befehl löst einen Objektnamen zu einer URL auf und gibt diese zurück. Diese wird URL codiert zurückgegeben, kann also direkt beispielsweise in einem HREF weiterverwendet werden. Folgende Objekttypen haben eine URL, die mit GETURL ermittelt werden kann:

- CGI
- HTML
- Bild

Parameter:

NAME
Dieser Parameter gibt den Namen des Objektes an, dessen URL ermittelt werden soll

.Beim Einsatz von CIPP als Apache Modul wird an dieser Stelle statt der Objektbezeichnung eine URL erwartet.

RUNTIME
Der Schalter **RUNTIME** bewirkt, daß der Objektname erst zur Laufzeit aufgelöst wird. In diesem Fall kann der Objektname auch Variablen enthalten. Wenn **RUNTIME** gesetzt ist, findet zur Übersetzungszeit keine Überprüfung statt, ob das mit **NAME** angegebene Objekt im Projekt exisitiert.

URLVAR

Hier muß eine skalare Variable angegeben werden, die die URL des Objektes aufnehmen soll.

MY

Optional kann der Schalter **MY** gesetzt werden. In diesem Fall wird die **URLVAR** Variable implizit mit **my** deklariert, so daß sie nur bis zum Ende des den **<?GETURL>** Befehl umgebenden Blocks bekannt ist.

THROW

Der Parameter **THROW** gibt die Ausnahme an, die generiert werden soll, wenn der Name zur Laufzteit nicht aufgelöst werden kann. Per Default wird in diesem Fall die Ausnahme geturl generiert.

PARAMS

Wenn das Objekt ein CGI-Objekt ist, dürfen auch Parameter angegeben werden, die an das CGI Objekt übergeben werden sollen. Über **PARAMS** kann eine Menge von Parametern angegeben, die mit Komma voneinander getrennt sein müssen. Dabei können sowohl skalare also auch Listen-Variablen angegeben werden. Skalare Parameter müssen ein **$** und Listen-Parameter ein **@** vorangestellt bekommen. Mit Listenparametern kann z.B. eine Multiple-SELECT-List übergeben werden.

Mit **PARAMS** können nur Parameter übergeben werden, deren Inhalt im aktuellen Dokument in **gleichnamigen** Variablen steht (mit Berücksichtigung der Groß- und Kleinschreibung).

PAR1 ... PARn

Alle zusätzlichen Parameter des **<?GETURL>** Befehls werden als weitere Übergabeparameter interpretiert, deren Wertzuweisung völlig frei ist. Hierbei gibt es aber eine Einschränkung, die sich auf die Groß- und Kleinschreibung der Parameter bezieht. Die auf diese Weise übergebenen Parameter erscheinen im aufgerufenen Programm als **klein** geschriebenen Variablen.

Beispiel:

Das Bildobjekt ‚MSG.Images.Logo' wird eingebunden (Hinweis: dies geht auch einfacher mit **<?IMG>**):

```
<?GETURL NAME=MSG.Images.Logo MY URLVAR=$url>
<IMG SRC="$url">
```

Nun wird die Seite ‚MSG.Secure.Messager' gelinkt, mit einem Benutzernamen und einer Reihe von ID's als Parameter, sowie mit einem EVENT-Parameter, der zusätzlich angegeben wird. Zu beachten ist dabei die Groß- und Kleinschreibung:

```
<?VAR NAME=<B>$Username</B>>hans<?/VAR>
<?VAR NAME=@id>(1,42,5)<?/VAR>
<?GETURL NAME=MSG.Secure.Messager URLVAR=$url
         PARAMS="$Username, @id" EVENT=delete>
<A HREF="$url">Nachrichten löschen</A>
```

In dem CIPP-CGI-Objekt ‚MSG.Secure.Messager' kann auf die Parameter wie folgt zugegriffen werden:

```
<?INTERFACE INPUT="$event, $Username, @id">
<?IF COND="$event eq 'delete'">
  <?MY $id_text>
  <?PERL>$id_text = join (", " @id)<?PERL>
  Sie möchten folgende ID's löschen: $id_text<BR>
<?/IF>
```

HIDDENFIELDS

Bereich:

URL- und Formularhandling

Syntax:

```
<?HIDDENFIELDS [ PARAMS=Parametervariablen ]
               [ PAR_1=Wert_1 ... PAR_n=Wert_n ] >
```

Beschreibung:

Es werden eine Reihe von HTML **<INPUT TYPE=HIDDEN>** Tags generiert, so daß eine Menge von Parametern über ein Formular mit einem CIPP-Befehl an ein anderes CGI-Objekt übergeben werden kann.

Der Inhalt der Parameter wird so codiert, daß keine HTML-Syntaxfehler auftreten können. z.B. werden doppelte Anführungszeichen durch **&"** ersetzt. Da der Browser diese Codierungen beim Abschicken des Formulars wieder zurückverwandelt, erscheinen beim aufgerufenen CGI-Programm die Parameter natürlich wieder in ihrer ursprünglichen Form.

Parameter:

PARAMS

Über **PARAMS** kann eine Menge von Parametern angegeben, die mit Komma voneinander getrennt sein müssen. Dabei können sowohl skalare also auch Listen-Variablen angegeben werden. Skalare Parameter müssen ein **$** und Listen-Parameter ein **@** vorangestellt bekommen. Mit Listenparametern kann z.B. eine Multiple-SELECT-List übergeben werden.

Mit **PARAMS** können nur Parameter übergeben werden, deren Inhalt im aktuellen Dokument in **gleichnamigen** Variablen steht (mit Berücksichtigung der Groß- und Kleinschreibung).

PAR1 ... PARn

Alle zusätzlichen Parameter des **<?GETURL>** Befehls werden als weitere Übergabeparameter interpretiert, deren Wertzuweisung völlig frei ist. Hierbei gibt es aber eine Einschränkung, die sich auf die Groß- und

Kleinschreibung der Parameter bezieht. Die auf diese Weise übergebenen Parameter erscheinen im aufgerufenen Programm als **klein** geschriebenen Variablen.

Beispiel:

Es wird ein Formular generiert, daß die Parameter **$username** und **$password** weitergibt und zusätzlich einen Parameter **$event** enthält und als **ACTION** das CGI-Objekt ‚MSG.Secure.Messager' aufruft:

```
<?FORM ACTION=MSG.Secure.Messager>
<?HIDDENFIELDS PARAMS="$username, $password"
               EVENT=anzeigen>
<INPUT TYPE=SUBMIT VALUE="Nachrichten anzeigen">
<?/FORM>
```

HTMLQUOTE

Bereich:

URL- und Formularhandling

Syntax:

```
<?HTMLQUOTE VAR=zu_codierende_Variable
            [ MY ] HTMLVAR=Ziel_Variable >
```

Beschreibung:

Der Inhalt einer einzelnen Variable wird HTML-gequotet und das Ergebnis wird in eine Ziel-Variable geschrieben. Dabei werden folgende Zeichen in der angegebenen Reihenfolge durch ihre entsprechenden HTML Entities ersetzt:

```
&  =>  &amp;
<  =>  &lt;
„  =>  &quot;
```

Der so codierte String kann nun ohne weiteres in einem HTML-**<TEXTAREA>** verwendet werden und wird automatisch durch den Browser beim Zurücksenden des Formulars rückkonvertiert, so daß beim Empfang der CGI Parameter keine besonderen Maßnahmen mehr getroffen werden müssen.

Parameter:

VAR
Der Parameter **VAR** gibt den Namen der zu kodierenden skalaren Variablen an.

HTMLVAR
Der Name der skalaren Zielvariablen wird mit diesem Parameter angegeben.

MY
Optional kann der Schalter **MY** angegeben werden. In diesem Fall wird die Zielvariable implizit mit **my** deklariert, so daß sie nur bis zum Ende des den **<?HTMLQUOTE>** Befehl umgebenden Blocks bekannt ist.

Beispiel:

Es wird ein **TEXTAREA** aufgebaut, dessen Inhalt aus der Variablen **$text** vorinitialisiert wird:

```
<?HTMLQUOTE VAR=$text MY HTMLVAR=$html_text>
<TEXTAREA NAME=text>$html_text</TEXTAREA>
```

IF

Bereich:

Kontrollstukturen

Syntax:

```
<?IF COND=Bedingung >
...
<?/IF>
```

Beschreibung:

Der **<?IF>** Befehl beschreibt einen Block, der nur ausgeführt wird, wenn die angegeben Bedingung wahr ist.

In einem **<?IF>** Block dürfen auch **<?ELSIF>** und **<?ELSE>** Blöcke mit der entsprechenden Bedeutung vorkommen (siehe Beschreibung von **<?ELSIF>** und **<?ELSE>**).

Parameter:

COND
Hier wird eine Perl-Bedingung erwartet. Es darf ein belieber Perl Ausdruck, der einen Wahrheitswert zurückgibt, eingesetzt werden

Beispiel:

Nur Larry wird hier begrüßt.

```
<?IF COND="$name eq 'Larry'">
  Hi Larry!
<?/IF>
```

IMG

Bereich:

Ersetzungen von HTML Tags

Syntax:

```
<?IMG SRC=Bild-Objekt
      [ zusätzliche_<IMG>_Parameter ... ] >
```

Beschreibung:

Es wird ein HTML **<IMG>** Tag generiert, das als **SRC** auf ein Bild-Objekt verweist.

Parameter:

SRC
Mit SRC wird das Bild-Objekt angegeben, das in die HTML Seite eingebunden werden soll.

Beim Einsatz von CIPP als Apache Modul wird an dieser Stelle statt der Objektbezeichnung eine URL erwartet.

zusätzliche_<IMG>_Parameter
Alle weiteren Parameter und Schalter werden unverändert in das generierte HTML **<IMG>** Tag übernommen, d.h. zusätzliche Parameter wie **BORDER**, **WIDTH** oder **HEIGHT** können so angegeben werden.

Beispiel:

Das Bild ‚MSG.Images.Logo' wird als Link auf die Seite ‚MSG.Main' in die HTML-Seite eingebunden. Dabei wird die BORDER des Bildes auf 0 gesetzt:

```
<?A HREF=MSG.Main>
<?IMG SRC=MSG.Images.Logo BORDER=0>
<?/A>
```

INCINTERFACE

Bereich:

Schnittstellen

Syntax:

```
<?INCINTERFACE [ INPUT=Variablen_Liste ]
               [ OPTIONAL=Variablen_Liste
               [ NOQUOTE=Variablen_Liste ]
               [ OUTPUT=Variablen_Liste ]>
```

Beschreibung:

Der Befehl INCINTERFACE dient zur Deklaration der Schnittstelle eines Include Objektes. Dieser Befehl darf also nur innerhalb eines Include Objektes verwendet werden - in einem CGI Objekt macht er keinen Sinn.

Die Deklaration einer Include Schnittstelle ist notwendig, wenn das Include Objekt in ein Objekt eingebunden wird, welches im 'use strict' Modus läuft. Es wird empfohlen die Schnittstelle immer zu deklarieren, da so zur Übersetzungszeit Fehler bei der Parameterübergabe erkannt werden können.

Es können Pflicht-Eingabeparameter, optionale Eingabeparameter sowie Ausgabeparameter deklariert werden. Bei allen Parametern handelt es sich um benannte Parameter, auch bei den Ausgabeparametern.

Parameter werden also nicht wie sonst bei vielen Programmiersprachen über ihre **Stellung** in einer Liste (Stellungsparameter) identifiziert, sondern immer über einen **Namen**. Bei den Ausgabeparameter führt dies zu einer etwas ungewöhnlichen aber trotzdem sehr effektiven Syntax beim Einbinden von Include-Objekten (siehe Beispiel).

Parameter:

INPUT
Der Parameter **INPUT** gibt die Parameter an, die zwingend übergeben werden müssen. Wenn einer dieser Parameter beim Aufruf fehlt, gibt es während der Übersetzung eine Fehlermeldung, die eine Liste der fehlenden Parameter enthält.

OPTIONAL

Der Parameter **OPTIONAL** gibt an, welche Parameter zusätzlich angegeben werden dürfen, aber nicht müssen. Diese werden innerhalb des Includes in jedem Fall mit my deklariert und sind undef, falls sie nicht übergeben wurden.

OUTPUT

Mit dem Parameter **OUTPUT** können eine Reihe von Ausgabeparametern deklariert werden, d.h. die hier aufgeführten Variablen werden an das aufrufende CIPP Objekt (CGI oder Include) zurück übergeben. Beim Aufruf wird festgelegt, in welche Variablen des CIPP Objektes die hier deklarierten Parameter übernommen werden (siehe Beispiel).

NOQUOTE

Der Parameter **NOQUOTE** listet die Variablen auf, die bei der Übergabe aus dem CIPP Objekt intern nicht gequotet werden sollen. Hier können nur Variablen aufgelistet werden, die entweder in **INPUT** oder **OPTIONAL** deklariert wurden. Per Default erfolgt die Parameterzuweisung intern immer in doppelten Anführungszeichen, so daß als Parameter ohne weiteres konstante Zeichenketten übergeben werden können. Wird eine Variable in **NOQUOTE** aufgeführt, so erfolgt die Zuweisung ohne die Anführungszeichen. So können dann z.B. Listen, Hashs und insbesondere Perl-Referenzen übergeben werden.

Ausgabeparameter werden grundsätzlich intern ohne Anführungszeichen zurückgegeben, so daß diese auch nicht in **NOQUOTE** aufgelistet werden müssen bzw. dürfen.

Alle hier angegebenen Variablenlisten enthalten die Variablen mit Komma voneinander getrennt, wobei Parameter mit vorangestelltem **$** als Scalare, welche mit **@** als Liste und solche mit **%** als Hash interpretiert werden. Es muß darauf geachtet werden, daß Listen- und Hash-Parameter in **NOQUOTE** aufgeführt sind, sonst führt die Zuweisung der Parameter zu unerwünschten Ergebnissen oder Laufzeitfehlern.

Hinweise:

Der **<?INCINTERFACE>** Befehl darf mehrmals in einem CIPP Include auftreten, die Position innerhalb des Quelltextes spielt keine Rolle. Bei mehrfacher Verwendung von **<?INCINTERFACE>** addieren sich die Deklarationen.

Fehlt der **<?INCINTERFACE>** Befehl, werden per Default alle Parameter, die an das Include beim Aufruf übergeben werden, in den Sichtbereich des Includes importiert. Alle Parameter, die mit **<?INCINTERFACE>** deklariert werden, werden blocklokal innerhalb des CIPP Includes mit **my** deklariert, können also nicht mit gleichnamigen Variablen außerhalb des Includes kollidieren.

Der **<?INCINTERFACE>** Befehl muß in persistenten Perl Umgebungen verwendet werden, da hier die vorherige Deklaration stattfinden muß, um die Variablen gegen den Zugriff von außen bzw. anderen Script-Instanzen innerhalb des persistenten Interpreters zu schützen. Die Besonderheiten von CIPP in persistenten Perl Umgebungen werden in einem eigenen Kapitel ausführlich beschrieben.

Wichtiger Hinweis:

Bei der derzeitigen Implementierung von **<?INCLUDE>** wird der eingebundene Quellcode tatsächlich an die entsprechende Stelle des Aufrufs hineinkopiert. Das bedeutet letztlich, der Code eines INCLUDEs steht im lexikalischen Kontext des aufrufenden Programms. Somit sind mit my deklarierte Variablen des aufrufenden Programms innerhalb des INCLUDEs sichtbar, obwohl diese weder über ein **<?INCINTERFACE>** oder anders innerhalb des INCLUDEs deklariert wurden. Dieser Effekt sollte allerdings keinesfalls bewußt (oder auch unbewußt ;) ausgenutzt werden, da eine zukünftige Implementierung von **<?INCLUDE>** diesen Verhalten ändern könnte, indem z.B. der INCLUDE Quelltext ausgelagert wird und somit nicht mehr im lexikalischen Kontext des aufrufenden Progamms steht.

Beispiel:

Im folgenden Beispiel wird eine Schnittstelle deklariert, die die Parameter **$vorname** und **$nachname**, sowie optional eine Liste von **ID**'s entgegennimmt. Diese werden in **NOQUOTE** geführt, da sonst die Liste als String übernommen wird, was nicht zum gewünschten Ergebnis führen würde. Das Include Objekt gibt die zwei Parameter **$scalar** und **@list** zurück, in derm im Beispiel aufgeführten **<?PERL>** Block erfolgt die Zuweisung der Ausgabeparameter.

```
<?INCINTERFACE INPUT="$vorname, $nachname"
               OPTIONAL="@id"
               OUTPUT="$scalar, @list"
               NOQUOTE="@id">
...
<?PERL>
  $scalar="Rückgabe eines Skalars";
  @list="Rückgabe einer Liste"
<?/PERL>
```

Das Einbinden des Include Objektes mit der obigen Schnittstelle kann dann z.B. so aussehen (der Name des Include Objektes ist in diesem Fall 'MSG.Macro.Test'):

```
<?INCLUDE NAME=MSG.Macro.Test
          VORNAME="Larry"
          NACHNAME="Wall"
          ID="(5,4,3)"
          MY
          $s=SCALAR
          @l=LIST>
```

Der Ausgabeparameter **$scalar** des Include Objektes wird der Variablen **$s** und **@list** wird **@l** zugewiesen. Diese Variablen werden wegen der Angabe von **MY** an dieser Stelle implizit deklariert.

Es muß beachtet werden, daß bei der Deklaration der Parmater Kleinschreibung verwendet werdn muß. Bei der Übergabe der Parameter spielt die Schreibweise keine Rolle, es erfolgt intern immer eine Konvertierung in die Kleinschreibung.

Hinweis:

Bei der Verwendung von Ausgabeparametern muß folgendes beachtet werden:

Beim Aufruf eines Includes darf die Variable, der der Wert eines Ausgabeparameters zugewiessen wird, nicht denselben Namen wie der Ausgabeparamter haben. Dies wird zur Übersetzungszeit als Fehler erkannt.

Beispiel für einen fehlerhaften Aufruf:

```
<?INCLUDE NAME=MSG.Macro.Test
             VORNAME="Linus"
             NACHNAME="Torvalds"
             ID="(5,4,3)"
             $s=SCALAR
Fehler! ==>  @list=LIST>
```

Die Variable für den Ausgabeparameter **`LIST`** darf hier nicht **`@list`** heißen.

INCLUDE

Bereich:

Import

Syntax:

```
<?INCLUDE NAME=Name_des_einzubindenden_Objektes
        [ Eingabe_Parameter1=Wert1 ] ...
        [ MY ]
        [ Variable1=Ausgabe_Parameter1 ] ... >
```

Beschreibung:

Der **<?INCLUDE>** Befehl bindet ein Include Objekt in das aktuelle CIPP Objekt ein. Include Objekte können auch von Include Objekten eingebunden werden. Lediglich das rekursive Einbinden von Include Objekten ist nicht erlaubt und wird zur Übersetzungszeit als Fehler erkannt.

Mit Include Objekten können Programmteile auslagert werden, die auch in anderen Objekten immer wieder verwendet werden können.

Parameter:

NAME

Der Parameter **NAME** enthält den projektinternen Namen des Include Objektes.

Beim Einsatz von CIPP als Apache Modul wird an dieser Stelle statt der Objektbezeichnung eine URL erwartet.

Eingabe_Parameter

Dem Include Objekt können Eingabeparameter übergeben werden. Wurde innerhalb des Include Objektes eine Schnittstelle deklariert (siehe **<?INCINTERFACE>**), so wird beim Übersetzungsvorgang überprüft, ob diese auch eingehalten wird.

Der Zugriff auf die Eingabe-Parameter erfolgt innerhalb des Include Objektes über Variablen, die denselben Namen haben, wie der entsprechende Parameter. Dabei müssen diese Variablennamen im Include Objekt immer klein geschrieben werden, ungeachtet der Groß- und Kleinschreibung beim Einbinden des Include Objektes.

Ausgabe_Parameter

Bei der Deklaration der Include Schnittstelle können auch Ausgabeparameter definiert worden sein. Diese werden übernommen, indem beim Einbinden des Include Objektes die Namen dieser Ausgabeparameter Variablen des aufrufenden Objektes zugewiesen werden. Dies führt zu einer ungewöhnlichen aber trotzdem sehr intuitiven Schreibweise: bei der Parameterübergabe wird einer Variablen ein Parameter zugewiesen wird, die Variable steht also **links** vom Gleichheitszeichen, wo hingegen eine Eingabeparameter-Variable **rechts** neben dem Parameternamen steht. (siehe Beispiel)

MY

Wird der Schalter **MY** angegeben, so werden die Ausgabevariablen bei der Übernahme implizit mit **my** deklariert.

Kurzschreibweise:

Der **<?INCLUDE>** Befehl kann auch wie folgt abgekürzt werden:

```
<?Name_des_einzubindenden_Objektes
  [ Eingabe_Parameter1=Wert1 ] ...
  [ MY ]
  [ Variable1=Ausgabe_Parameter1 ] ... >
```

Der Befehlsname **<?INCLUDE>** kann also einfach weggelassen werden. Der Name des einzubindenen Include Objektes wird direkt hinter das **<?** Token geschrieben.

Beispiel:

Das Include Objekt 'MSG.Login.Check' wird in das aktuelle Dokument eingebunden, dabei wird an das Objekt der Parameter **ADMIN_LOGIN**=*1* übergeben. Die Variable **$success** erhält den Wert des Ausgabeparameters **LOGIN_SUCCESS** und wir implizit deklariert:

```
<?INCLUDE NAME="MSG.Login.Check" ADMIN_LOGIN=1
          MY $success=LOGIN_SUCCESS>
```

Weitere Beispiele werden bei der Beschreibung von **<?INCINTERFACE>** aufgeführt.

Anmerkung:

Da die Groß- und Kleinschreibung von Parametern bei CIPP Befehlen grundsätzlich ignoriert wird, geschieht dies auch bei den Parametern, die an ein Include Objekt übergeben werden. Deshalb erscheinen die Parameter innerhalb eines Include Objektes immer als kleingeschriebene Variablennamen.

In unserem Beispiel erfolgt der Zugriff auf den Eingabe-Parameter **`ADMIN_LOGIN`** über die Variable **`$admin_login`**. Der Ausgabe-Parameter **`LOGIN_SUCCESS`** erfolgt über eine Zuweisung zu der Variablen **`$login_success`**.

INPUT

Bereich:

Ersetzungen von HTML Tags

Syntax:

```
<?INPUT [ VALUE=Parameter_Wert ]
        [ zusätzliche_<INPUT>_Parameter ... ] >
```

Beschreibung:

Es wird ein HTML **<INPUT>** Feld generiert, wobei der VALUE des Feldes HTML-kodiert wird, damit es nicht zu HTML-Syntaxfehlern kommen kann, auch wenn z.B. doppelte Anführungszeichen im Inhalt des Feldes vorkommen.

Parameter:

VALUE
Der Parameter **VALUE** gibt den Wert an, der in dem Feld zugewiesen werden soll. Darüber hinaus können beliebige weitere Parameter angegeben werden, diese werden unverändert in das generierte **<INPUT>** Feld übernommen.

Beispiel:

Es werden zwei INPUT Felder generiert, eines zur Eingabe eines Benutzernamens und eines zur Eingabe eines Passwortes. Dabei werden die Felder mit bestimmten Werten vorinitialisiert.

```
<?VAR MY NAME=$username>larry<?/VAR
<?VAR MY NAME=$password>this is my “password“<?/VAR>
<?INPUT TYPE=TEXT SIZE=40 VALUE=$username>
<?INPUT TYPE=PASSWORD SIZE=40 VALUE=$password>
```

Das führt zu folgendem HTML Code:

```
<INPUT TYPE=TEXT SIZE=40 VALUE=“larry“>
<INPUT TYPE=TEXT SIZE=40
       VALUE=“this ist my &quot;password&quot;“>
```

INTERFACE

Bereich:

Schnittstellen

Syntax:

```
<?INTERFACE [ INPUT=Variablen_Liste ]
            [ OPTIONAL=Variablen_Liste ] >
```

Beschreibung:

Dieser Befehl deklariert die CGI Schnittstelle des CGI Objektes. Der **<?INTERFACE>** Befehl darf mehrmals in einem CIPP Programm auftreten, die Position innerhalb des Quelltextes spielt keine Rolle. Bei mehrfacher Verwendung von **<?INTERFACE>** addieren sich die Deklarationen.

Fehlt der **<?INTERFACE>** Befehl, werden per Default alle CGI Parameter, die an das Programm übergeben werden, als Variablen innerhalb des Programmes sichtbar. Dies funktioniert nur wenn das Objekt nicht im 'use strict' Modus läuft. Objekte, die den 'use strict' Modus verwenden müssen immer eine CGI Schnittstelle mit **<?INTERFACE>** deklarieren.

Alle Parameter, die mit **<?INTERFACE>** deklariert werden, werden blocklokal innerhalb des CIPP Programms definiert.

Der **<?INTERFACE>** Befehl muß in persistenden Perl Umgebungen verwendet werden, da hier die vorherige Deklaration stattfinden muß, um die Variablen gegen den Zugriff von außen bzw. anderen Script-Instanzen innerhalb des persistenten Interpreters zu schützen. Die Besonderheiten von CIPP in persistenten Perl Umgebungen werden in einem eigenen Kapitel ausführlich beschrieben.

Parameter:

INPUT

Der Parameter **INPUT** gibt die CGI Parameter an, die zwingend übergeben werden müssen. Wenn einer dieser Parameter beim Aufruf fehlt, wird eine **interface** Exception erzeugt, die als Meldung eine Liste der fehlenden Parameter enthält.

Die hier aufgelisteten Parameter erscheinen automatisch als gleichnamige Variablen innerhalb des CIPP Programms.

OPTIONAL

Der Parameter **OPTIONAL** gibt an, welche Parameter zusätzlich angegeben werden dürfen, aber nicht müssen.

Die **INPUT** und **OPTIONAL** Variablenlisten enthalten die Parameter mit Komma voneinander getrennt, wobei Parameter mit vorangestelltem $ als Scalare und welche mit @ als Liste entgegengenommen werden.

Beispiel:

Im folgenden Beispiel wird eine Schnittstelle deklariert, die die Parameter **$vorname** und **$nachname**, sowie optional eine Liste von ID's entgegennimmt.

```
<?INTERFACE INPUT="$vorname, $nachname"
            OPTIONAL="@id">
```

Ein Formular, welches das CGI Objekt mit der obigen Schnittstelle aufruft, könnte wie folgt aussehen:

```
<?VAR MY NAME=@id NOQUOTE>(1,2,3,4)<?/VAR>

<?FORM ACTION="MSG.User.Save">
<?HIDDENFIELDS PARAMS="@id">
<P>Vorname:
<?INPUT TYPE=TEXT NAME=vorname>
<P>Nachname:
<?INPUT TYPE=TEXT NAME=nachname>
<?/FORM>
```

LIB

Bereich:

Import

Syntax:

<?**LIB NAME**=*Name_des_Perl_Moduls* >

Beschreibung:

Über den Befehl **<?LIB>** ist der Zugriff auf die umfangreiche Standard-Perlbibliothek möglich.

Eigene Module können auch eingebunden werden, dabei können diese Module entweder in das Verzeichnis der Standardbibliothek hineinkopiert werden oder in das Unterverzeichnis **lib/** des **prod/** Verzeichnisses des Projektes.

Parameter:

NAME
Hier wird der Name des Perl Modules übergeben. Dabei wird die Schreibweise erwartet, die in Perl der **use** Befehl erwartet.

Beispiel:

Die Funktionen der Standard-Perlmodule **File::Path.pm** und **Text::Wrap.pm** aus der Perlbibliothek werden verfügbar gemacht:

```
<?LIB NAME=File::Path>
<?LIB NAME=Text::Wrap>
```

Das selbstgeschriebene Modul **Mein_Modul.pm** wird verfügbar gemacht. Das Modul kann allerdings nur gefunden werden, wenn es sich in der Standard-Perlbibliothek befindet oder wenn es in das **prod/lib/** Verzeichnis des Projektes kopiert wurde.

```
<?LIB NAME=Mein_Modul>
```

LOG

Bereich:

Ausnahmebehandlung

Syntax:

```
<?LOG MSG=Fehlermeldung
      [ TYPE=Typisierung_der_Meldung ]
      [ FILENAME=alternatives_Logfile ]
      [ THROW=Ausnahme ] >
```

Beschreibung:

Der <?LOG> Befehl trägt eine Fehler- oder Statusmeldung in die projektspezifische Loddatei ein, um auftretende Ereignisse besser analysieren zu können. Dabei wird ein Ereignis durch eine Typisierung und eine Meldung repräsentiert. Der jeweilige Typ eines Ereignisses wird der Meldung im Logfile vorangestellt und mit einem Doppelpunkt abgeschlossen. Bei der Auswertung der Logdatei kann so nach den Meldungen der einzelnen Typen gesucht werden.

Parameter:

MSG
Übergibt die Meldung, die in das Logfile geschrieben werden soll.

TYPE
Die Angabe eines Types erfolgt über den optionalen Parameter **TYPE**.

FILENAME
Mit dem optionalen Parameter **FILENAME** kann der Name des Logfiles angegeben werden, in das die Meldung abgesetzt werden soll. Wenn dieser nicht angegeben wird, wird das Logfile **cipp.log** im **prod/logs/** Verzeichnis des Projektes verwendet.

THROW
Wird der Parameter **THROW** angegeben, wird diese Ausnahme generiert, wenn das Logfile nicht beschrieben werden konnte. Defaultmäßig wird in diesem Fall die Ausname **log** generiert.

Beispiel:

Falls die Variable **`$fehler`** nicht den Wert **`0`** enthält, wird die Fehlermeldung „Es ist ein interner Fehler aufgetreten“ in die Logdatei des Projekts geschreiben:

```
<?IF COND="$fehler != 0">
  <?LOG MSG="Es ist ein interner Fehler
            aufgetreten">
<?/IF>
```

Die Fehlermeldung „Syntaxfehler im SQL-Befehl“ wird in die Logdatei

`/tmp/my.log`

geschrieben, wobei der Typ des Fehlers „Datenbank-Fehler“ noch vorangestellt wird. Es wird die Ausnahme ’file_io’ generiert, wenn das Logfile nicht geschrieben werden kann:

```
<?LOG MSG="Syntaxfehler im SQL-Befehl"
      TYPE="Datenbank-Fehler"
      FILE="/tmp/my.log"
      THROW=file_io>
```

MY

Bereich:

Variablen- und Gültigkeitsbereiche

Syntax:

```
<?MY [ VAR=Variablen_liste ]
     Variable1 ... VariableN >
```

Beschreibung:

Dieser Befehl deklariert eine oder mehrere private Variablen. Diese Variablen existieren nur bis zum Ende des den **<?MY>** Befehl umschließenden Blocks, z.B. nur innerhalb eines IF- oder BLOCK-Blocks.

Wenn einer Variablen auch direkt ein Wert zugewiesen werden soll, ist der **<?VAR>** Befehl zu verwenden. Eine mit **<?MY>** deklarierte Variable hat den undefinierten Wert.

Paramter:

VAR
Mit dem Parameter **VAR** kann eine Komma-getrennte Liste von Variablen angegeben werden. Diese Variablennamen werden case sensitiv, also unverändert übernommen. Bei der Angabe der Variablennamen darf das vorstehende Zeichen zur Typisierung der Variablen nicht fehlen (**$ @ %**).

Variable1 ... VariableN
Zusätzlich können weitere Variablen innerhalb des **<?MY>** Befehls zusätzlich aufgelistet werden.

Achtung: Diese Variablennamen werden vor der Deklaration in Kleinschreibweise umgewandelt.

Anmerkung:

Es gibt eine Reihe von CIPP Befehlen, die zur Rückgabe von Werten Variablen erzeugen. Diese Befehlen haben alle den optionalen Schalter **MY**, der bewirkt, daß die entsprechende(n) Variable(n) implizit mit **MY** deklariert werden, so daß Sie nur bis zum Ende des umschließenden Blocks gültig sind, so als ob vor dem CIPP Befehl, diese Variable mit **<?MY>** deklariert worden wäre.

Beispiel:

Die folgende Anweisung deklariert drei Variablen: den Skalar **$name**, die Liste **@liste** und das assoziative Array **%Hash**. Die Groß- / Kleinschreibweise ist zu beachten. Diese Variablen sind nur in dem den **<?MY>** Befehl umschließenden Block gültig:

```
<?BLOCK>
  <?MY $name @liSTE VAR="%Hash">
  Das ergibt die Deklaration der folgenden
  Variablen: $name, @liste, %Hash
<?/BLOCK>
Hier sind die Variablen nicht mehr bekannt.
```

Hier wird in einer Schleife die Variable **$nummer** deklariert, die außerhalb dieser Schleife nicht bekannt ist:

```
<?WHILE COND="$i++ < 10">
  <?MY $nummer>
  ...
<?/WHILE>
$nummer gibt es hier nicht mehr.
```

PERL

Bereich:

Kontrollstrukturen

Syntax:

```
<?PERL [ COND=Bedingung ] >
...
<?/PERL>
```

Beschreibung:

Der **<?PERL>** Befehl erzeugt einen Block, in dem reiner Perl Code verwendet werden muß. Innerhalb des **<?PERL>** Blocks dürfen nur Perl-Befehle eingefügt werden, alle anderen Befehle, seien es HTML- oder CIPP-Befehle, dürfen in diesem Block nicht angegeben werden. Der **<?PERL>** Block erlaubt es Anweisungsteile in Perl zu schreiben, um dabei auf den vollen Funktionsumfang diese Programmiersprache zurückgreifen zu können.

Ausgaben auf **STDOUT**, die innerhalb des **<?PERL>** Blocks gemacht werden, erscheinen in der von diesem Objekt generierten HTML-Seite.

Paramter:

COND
Wenn der optionale Parameter **COND** mit angegeben wird, wird der **<?PERL>** Block nur dann ausgeführt, wenn die hier angegebene Perl Bedingung erfüllt ist.

Beispiel:

In der Variable **$text** wird die Zeichenfolge 'nt' durch die Zeichenfolge ‚no thanks' ersetzt und die Variable wird in die HTML Seite hinein ausgegeben:

```
<?PERL>
  $text =~ s/nt/no thanks/g;
  print $text;
<?/PERL>
```

Eine Liste wird zu einem String zusammengefügt, allerdings nur dann, wenn die Liste überhaupt Elemente enthält:

```
<?PERL COND="scalar(@liste) != 0">
  my ($string, $element);
  foreach $element ( @liste) {
    $string .= $element;
  }
  print $string;
<?/PERL>
# das kann man natürlich auch einfacher mit
# der Perl-Funktion ‚join' erledigen, aber dann
# wäre dieses Beispiel so kurz! :)
```

ROLLBACK

Bereich:

SQL

Syntax:

```
<?ROLLBACK [ DB=Name_der_Datenbank ]
           [ THROW=Ausnahme ] >
```

Beschreibung:

Der **<?ROLLBACK>** Befehl beendet die aktuelle Transaktion und macht dabei alle Änderungen der Transaktion in der Datenbank wieder rückgängig. Eine Transaktion beginnt beim ersten **<?SQL>** Befehl und endet jeweils immer nach einem **<?ROLLBACK>** oder **<?COMMIT>** Befehl, es sei denn die Datenbank befindet sich im Autocommit-Modus.

Parameter:

DB

Der Parameter **DB** bestimmt die Datenbank, an die der COMMIT Befehl abgesetzt werden soll. Wenn dieser Parameter weggelassen wird, bezieht sich die Abfrage auf die im Projekt voreingestellte Default-Datenbank.

THROW

Die hier angegebene Ausnahme wird im Fehlerfall generiert. Ohne die Angabe von **THROW** wird defaultmäßig die Ausnahme **commit** generier

Beispiel:

Es werden zwei Datensätze eingefügt. Schlägt eine der Anweisungen fehl, werden in jedem Fall beide Anweisungen rückgängig gemacht, so daß dann de fakto keine Änderungen an der Datenbank vorgenommen wurde:

```
<?TRY>
  <?SQL SQL="insert into foo
             values (42, 'bar')"><?/SQL>
  <?SQL SQL="insert into foo values
             (43, 'bla')"><?/SQL>
<?/TRY>
<?CATCH>
  <?ROLLBACK>
<?/CATCH>
```

SAVEFILE

Bereich:

Schnittstellen

Syntax:

```
<?SAVEFILE FILENAME=Dateiname_auf_Serverseite
           VAR=Variable_des_Upload_Formulars
           [ SYMBOLIC ]
           [ THROW=Ausnahme ] >
```

Beschreibung:

Der **<?SAVEFILE>** Befehl speichert eine Datei, die vom Client gesendet wurde, auf dem Server ab. Damit wird der sogenannte Datei-Upload realisiert.

Parameter:

VAR
Der Parameter **VAR** gibt die Bezeichnung des entsprechenden **<INPUT TYPE=FILE>** Formularfeldes an.

SYMBOLIC
Ist der Schalter **SYMBOLIC** gesetzt, gibt der Parameter **VAR** den Namen der Programmvariablen an, die den Namen des **<INPUT TYPE=FILE>** Formularfeldes enthält.

Die Verwendung von SYMBOLIC wird notwendig, wenn sich der Name des Formularfeldes erst zur Laufzeit ergibt, z.B. wenn mehrere durchnummerierte Felder verwendet werden, die Anzahl der Felder aber erst zur Laufzeit bekannt ist.

FILENAME
Dieser Parameter gibt den Dateinamen an, unter dem die Datei auf dem Server abgelegt werden soll.

THROW
Wird der Parameter **THROW** angegeben, wird die entsprechende Ausnahme im Fehlerfall generiert. Defaultmäßig wird die Ausname **savefile** generiert.

Hinweis:

Der Datei-Upload funktioniert nur, wenn das Formular, welches den Dateinamen abfragt, mit der Option **`ENCTYPE=multipart/form-data`** versehen wird. Ansonsten wird das Programm zur Laufzeit eine Fehlermeldung generieren, daß die vom Client gesendete Datei nicht gelesen werden konnte.

Beispiel:

Der folgende Code zeigt den Aufbau des Formulars, anhand dessen der Client eine Datei zum Server übertragen kann. **`ACTION`** des Formulars ist das CGI Objekt 'MSG.Image.Save', welches die gesendete Datei auf dem Server speichern wird:

```
<?FORM METHOD="POST" ACTION=MSG.Image.Save
       ENCTYPE="multipart/form-data">
Fileupload:
<INPUT TYPE=FILE NAME="filename" SIZE=45><BR>
<INPUT TYPE="reset">
<INPUT TYPE="submit" NAME="submit" VALUE="Upload">
</FORM>
```

Im CGI Objekt 'MSG.Image.Save' wird dann schließlich die Datei des Clients unter dem Dateinamen **`/tmp/upload.tmp`** auf dem Servers gespeichert. Bei nicht erfolgreicher Ausführung wird die Ausnahme **`upload`** abgesetzt.

```
<?SAVEFILE FILENAME="/tmp/upload.tmp"
           VAR="filename"
           THROW=upload>
```

Hier wird derselbe Datei-Upload realisiert, allerdings mit der Verwendung des **`SYMBOLIC`** Schalters:

```
<?VAR MY=$field_name>filename<?/VAR>
<?SAVEFILE FILENAME="/tmp/upload.tmp"
           SYMBOLIC
           VAR="$field_name"
           THROW=upload>
```

SQL

Bereich:

SQL

Syntax:

```
<?SQL SQL=SQL_Anweisung
      [ VAR=Variablen_für_das_Abfrageergebnis ]
      [ PARAMS=Eingabeparameter ]
      [ WINSTART=Startzeile ]
      [ WINSIZE=Anzahl_Zeilen ]
      [ RESULT=Rückgabewert ]
      [ DB=Name_der_Datenbank ]
      [ THROW=Ausnahme ] >
      [ MY ]
...
<?/SQL>
```

Beschreibung:

Mit dem **<?SQL>** Befehl können beliebige SQL-Anweisungen an eine Datenbank abgesetzt werden. Die Anweisung wird von der Datenbank ausgeführt, eventuelle Rückgabewerte bzw. Ergebnismengen können innerhalb des **<?SQL>** Blocks und auch danach verarbeitet werden.

Wenn die SQL Anweisung eine Ergebnismenge liefert (es sich also um eine **SELECT** Anweisung handelt), so wird der durch den **<?SQL>** Befehl gebildete Block für jede Zeile der Ergebnismenge wiederholt, wobei die mit dem **VAR** Parameter übergebenen Ergebnisvariablen die entsprechenden Werte enthalten (siehe Beispiel bei **VAR**).

Parameter:

SQL

Gibt die auszuführende SQL Anweisung an. Die Anweisung darf ein abschließendes Semikolon enthalten, es kann aber auch weggelassen werden.

Die Anweisung kann ?-Platzhalter enthalten, für die während der Ausführung die mit dem Parameter **PARAMS** übergebenen Eingabeparameter eingesetzt werden (siehe Beschreibung von **PARAMS**).

Die Anweisung wird von CIPP in keiner Weise manipuliert sondern direkt an den Datenbanktreiber übergeben, der die Ausführung kontrolliert. Die syntaktische Korrektheit der Anweisung wird von der Datenbank geprüft. Bei Fehlern wird eine Ausnahme generiert (siehe Beschreibung von **THROW**).

Dieses Beispiel zeigt eine einfache konstante **INSERT** Anweisung ohne die Übergabe von Parametern:

```
<?SQL SQL="insert into foo values (42, 'bar')">
<?/SQL>
```

VAR

Der **VAR** Paramater listet die skalaren Variablen auf, die die Spalten einer Ergebnismenge aufnehmen sollen. Dabei werden die Spaltenwerte von links nach rechts auf die gelisteten Variablen verteilt.

Die hier gelisteten Variablen enthalten hinter dem **<?SQL>** Block die Spalten der letzten Zeile der Ergebnismenge. Sie sind auch nach dem **<?SQL>** Block bekannt, wenn sie implizit mit dem **MY** Parameter deklariert wurden (siehe **MY** Parameter).

Im folgenden Beispiel werden die Spalten **num** und **str** aus der Tabelle **foo** gelesen und in die Variablen **$n** und **$s** übernommen und im HTML-Format ausgegeben:

```
<?SQL SQL="select num, str from foo"
      MY VAR="$n, $s">
  <P>n hat den Wert: $n
  <BR>s hat den Wert: '$s'
<?/SQL>
```

PARAMS

Wenn die SQL Anweisung Platzhalter enthält (ein Platzhalter wird durch ein Fragezeichen Symbol dargestellt), so müssen die einzusetzenden Parameter mit **PARAMS** übergeben werden. Es wird eine mit Komma getrennte Liste von Skalar- oder Listen- Ausdrücken bzw. Variablen erwartet. Es können so auch konstanten Zahlen oder Zeichenketten mit **PARAMS** übergeben werden.

Die ? können an jeder Stelle in einer SQL Anweisung verwendet werden, wo ein Literal bzw. Attribut-Wert erwartet wird. Sie können bei **UPDATE** und **INSERT** verwendet werden, um Spalten-Attribute zu setzen. In der **WHERE** Klausel können sie verwendet werden, um eine Bedingung zu parametrisieren.

Durch den Einsatz von Platzhaltern bei der Parametrisierung von SQL Anweisungen entfällt das sonst bei Zeichenketten notwendige escapen von speziellen Zeichen (siehe **<?DBQUOTE>**). Zeichenketten und numerische Variablen werden automatisch korrekt an die Datenbank übergeben. **NULL** Werte werden durch den Perl-Wert **undef** repräsentiert. Darüber hinaus ist diese Art der Parameterübergabe schneller als das direkte Einsetzen der Parameter in die SQL Anweisung.

Es folgen einige Beispiele, die den Einsatz von Platzhaltern demonstrieren:

```
<?VAR MY NAME=$n>42<?/VAR>
<?VAR MY NAME=$s>Hello 'World'<?/VAR>
<?SQL SQL="insert into foo values (?, ?, ?)"
      PARAMS="$n, $s, time()">
<?/SQL>

<?VAR MY NAME=$where_num>42<?/VAR>
<?SQL SQL="select num,str from foo
           where num = ?"
      PARAMS="$where_num">
      MY VAR="$column_n, $column_s">
  n=$column_n s='$column_s'<BR>
<?/SQL>

<?SQL SQL="update foo
           set str=?
           where n=?"
      PARAMS="$s, $where_num">
<?/SQL>
```

WINSTART

Wenn nur ein Ausschnitt einer Ergebnismenge verarbeitet werden soll, so wird mit **WINSTART** die Nummer der Zeile angegeben, ab der die Verarbeitung begonnen werden soll. Die Zählung der Zeilen beginnt bei 1.

Im folgenden Beispiel werden die ersten 5 Zeilen der Ausgabe verworfen und von dieser Position an alle folgenden Zeilen ausgegeben:

```
<?SQL SQL="select num, str from foo"
      MY VAR="$n, $s"
      WINSTART=6
  n=$n s='$s'<BR>
<?/SQL>
```

WINSIZE

Dieser Parameter gibt an, wieviele Zeilen einer Ergebnismenge verarbeitet werden sollen. Er wird in der Regel zusammen mit **WINSTART** verwendet, so daß ein definierter Ausschnitt aus der Ergebnismenge definiert wird.

Im folgenden Beispiel werden die ersten 5 Zeilen übersprungen und dann maximal 5 weitere Zeilen verarbeitet. Die Verarbeitung beginnt also bei der 6. Zeile und endet mit der 10. Zeile:

```
<?SQL SQL="select num, str from foo"
      MY VAR="$n, $s"
      WINSTART=6 WINSIZE=5
  n=$n s='$s'<BR>
<?/SQL>
```

RESULT

Der **RESULT**-Parameter erwartet eine skalare Variable, der dann ein möglicher Resultcode der SQL-Anweisung zugewiesen wird, beispielsweise die Anzahl gelöschter Zeilen einer **DELETE** Anweisung. Der Resultcode ist im inneren des **<?SQL>** Blocks nicht verfügbar, sondern erst dahinter.

In diesem Beispiel wird die Anzahl der gelöschten Zeilen in der implizit mit **my** deklarierten Variablen **$deleted** zurückgegeben:

```
<?SQL SQL="delete from foo where num=42"
      MY RESULT=$deleted>
<?/SQL>
Es wurden $deleted Zeilen gelöscht!
```

DB

Der Parameter **DB** bestimmt die Datenbank, an die die Anweisung abgesetzt werden soll. Die hier angegebene Datenbank muß innerhalb des Projektes definiert worden sein. Es wird der Objektname der Datenbankkonfiguration erwartet.

Wenn dieser Parameter nicht gegeben ist, bezieht sich die Abfrage auf die im Projekt voreingestellte Default-Datenbank.

Die folgende **INSERT** Anweisung wird über die Datenbank mit dem Objektnamen 'MSG.Oracle' abgewickelt:

```
<?SQL SQL="insert into foo values (42, 'bar')"
      DB=MSG.Oracle>
<?/SQL>
```

THROW

Im Falle eines Fehlers bei der Ausführung der SQL Anweisung wird die mit **THROW** übergebene Ausnahme generiert. Defaultmäßig wird die Ausnahme **sql** generiert.

MY

Wird der Parameter **MY** angegeben, werden die Variablen des Abfrageergebnisses bzw. des Rückgabewertes implizit vor der Ausführung der SQL Anweisung mit 'my' deklariert, so daß sie nur bis zum Ende des den **<?SQL>** Block umgebenden Blocks bekannt sind.

In jedem Fall sind die mit **VAR** und **RESULT** angegebenen Variablen hinter dem **<?SQL>** Block bekannt und enthalten die Werte des letzten Durchlaufs des **<?SQL>** Blocks.

Beispiel:

Die Namen von mehreren Personen werden aus der Datenbank ‚MSG.DB' geholt und übersichtlich in einer Tabelle dargestellt.

```
<TABLE>
<TR>
  <TH>Nachname</TH>
  <TH>Vorname</TH>
</TR>

<?SQL SQL="select vorname, nachname from personen"
      DB="MSG.DB"
      MY VAR="$vorname, $nachname">
  <TR>
    <TD>$nachname</TD>
    <TD>$vorname</TD>
  </TR>
<?/SQL>

</TABLE>
```

Einfügen eines Datensatzes in die Datenbank ‚MSG.DB'. Die Spalten des Datensatzes werden mit Platzhaltern gesetzt. Die Variable $inserted enthält danach die Anzahl der eingefügten Datensätze, in diesem Fall also 1:

```
<?SQL SQL="insert into personen (vorname, nachname)
           values (?, ?)"
      PARAMS="$vorname, $nachname"
      DB=MSG.DB RESULT=$inserted>
<?/SQL>
Es wurden $inserted Datensätze eingefügt.
```

SUB

Bereich:

Kontrollstrukturen

Syntax:

```
<?SUB NAME=Name_der_Subroutine >
...
<?/SUB>
```

Beschreibung:

Mit diesem Befehl wird eine Perl-Subroutine definiert. Dabei können im Body des **<?SUB>** Befehls beliebige CIPP und Perl Befehle verwendet werden.

Parameter:

NAME
Name der Subroutine. Dieser Name muß ein gültiger Perl-Subroutinen Name sein.

Hinweis:

In der Regel ist die Verwendung von INCLUDEs, die eine vollständige Schnittstellendefinition für die Ein- und Ausgabe von Paramtern besitzen, eleganter als die Verwendung von Perl-Subroutinen.

Es gibt aber Anforderungen, z.B. wenn objektorientierte Programmiertechniken verwendet werden sollen, die sich nur mit Subroutinen erfüllen lassen.

Ebenso ist der Einsatz von Subroutinen dann günstiger, wenn eine bestimmte komplexe Funktionaliät in einer Seite mehrmals benötigt wird, da ein INCLUDE immer wieder komplett in den Quelltext eingebunden wird und so das Programm entsprechend gößer macht.

In Zukunft wird die Verwendung von Perl-Subroutinen mit CIPP durch weitere Features attraktiver werden. So sind z.B. auch bei Subroutinen kontrollierte Schnittstellen denkbar.

Beispiel:

Es wird eine Subroutine erzeugt, die ein speziell formatiertes Texteingabefeld ausgibt. Labeltext, Name und der Wert des Eingabefeldes werden als Parameter übergeben.

```
<?SUB NAME=print_input_field>
  <?MY $label $name $value>

  # Entgegennahme der Übergabeparameter
  <?PERL>
    ($label, $name, $value) = @_;
  <?/PERL>

  # Ausgabe des Textfeldes
  <P>
  <B>$label:</B><BR>
  <?INPUT TYPE=TEXT SIZE=40 NAME=$name VALUE=$value>
<?/SUB>
```

Diese Subroutine kann nun wie folgt aus einem Perl-Kontext heraus aufgerufen werden:

```
<?PERL>
  print_input_field ('Vorname', 'firstname',
                     'Larry');
  print_input_field ('Nachname', 'surname',
                     'Wall');
<?/PERL>
```

TEXTAREA

Bereich:

Ersetzungen von HTML Tags

Syntax:

```
<?TEXTAREA [ beliebige_<TEXTAREA>_Parameter ... ] >
...
<?/TEXTAREA>
```

Beschreibung:

Es wird ein HTML **<TEXTAREA>** Feld generiert, wobei der Inhalt des Feldes HTML-kodiert wird, damit es nicht zu HTML-Syntaxfehlern kommen kann, auch wenn z.B. „</TEXTAREA>“ im Inhalt des Feldes vorkommt.

Parameter:

-

Beispiel:

Es wird eine TEXTAREA erzeugt, die mit dem Inhalt der Variablen $fulltext vorinitialisiert wird.

```
<?VAR MY NAME=$fulltext><B>Text mit HTML</B><?/VAR>
<?TEXTAREA NAME=fulltext ROWS=10
COLS=80>$fulltext<?/TEXTAREA>
```

Das führt zu folgendem HTML Code:

```
<TEXTAREA NAME=fulltext ROWS=10
COLS=80>&lt;B>Text mit HTML&lt;B></TEXTAREA>
```

THROW

Bereich:

Ausnahmebehandlung

Syntax:

<?**THROW THROW**=*Ausnahme* [**MSG**=*Nachricht*] >

Beschreibung

Mit dem **<?THROW>** Befehl wird eine Ausnahme erzeugt.

Parameter:

THROW
Der **THROW** Parameter bezeichnet die Ausnahme, die abgesetzt werden soll.

MSG
Optional kann man über den Parameter **MSG** der Ausnahme noch eine entsprechende Fehlermeldung mitgeben.

Beispiel:

Hier wird versucht, eine Datei zu öffnen. Falls dies nicht gelingt, wird eine Ausnahme erzeugt, die in diesem Fall zur Beendigung des Programmes führt, da kein **<?TRY>** Block den **<?THROW>** Befehl umgibt.

```
<?MY $fehler>
<?PERL>
  $fehler = 0;
  open (INPUT, '/bar/foo') or $fehler=1;
<?/PERL>

<?IF COND="$fehler == 1">
  <?THROW THROW="open_file" MSG="Datei /bar/foo">
<?/IF>
```

Hinweis:

Wenn innerhalb eines **<?PERL>** Blocks eine Ausnahme abgesetzt werden soll, so kann dies mit der **die** Anweisung geschehen. Der Parameter für die **die** Anweisung muß sich dabei aus der Bezeichnung der Ausnahme, gefolgt von einem Tabulatorzeichen, gefolgt von der Meldung der Ausnahme zusammensetzen:

```
die (“ausnahme\tMeldung der Ausnahme“);
```

Eine so in einem **<?PERL>** Block abgesetzt Ausnahme kann mit **<?CATCH>** wie eine CIPP Ausnahme behandelt werden.

Das obige Beispiel ließe sich also auch wie folgt formulieren:

```
<?PERL>
  open (INPUT, ’/bar/foo’)
    or die “open_file\tDatei/bar/foo“;
<?/PERL>
```

TRY

Bereich:

Ausnahmebehandlung

Syntax:

```
<?TRY >
...
<?/TRY >
```

Beschreibung:

Wenn innerhalb eines CIPP Programms eine Ausnahme abgesetzt wird, bzw. ein fataler Fehler auftritt, führt das zur sofortigen Beendigung des Programms. D.h. es wird der CIPP interne Ausnahmemechanismus aktiv, der eine Fehlermeldung auf dem Browser ausgibt, über STDERR einen Eintrag ins Webserver-Logfile vornimmt und danach das Programm beendet.

Sollen Ausnahmen aber gesondert behandelt werden, so müssen die entsprechenden Anweisungen innerhalb eines **<?TRY>** Blocks stehen. Wird ein **<?CATCH>** Block hinter diesem **<?TRY>** Block angegeben, so können die Ausnahmen dort behandelt werden.

Wird kein **<?CATCH>** Block angegeben, wird die Ausnahme einfach ignoriert und das Programm hinter dem **<?TRY>** Block fortgesetzt.

Näheres zu der Verwendung **<?TRY>** und **<?CATCH>** findet sich in der Beschreibung zu **<?CATCH>**.

Beispiel:

Hier wird versucht, einen Datensatz in eine Datenbank einzufügen. Falls diese Aktion fehlschlägt, wird die Ausnahme **INSERT_Exception** abgesetzt, welche von dem **<?CATCH>** Befehl aufgenommen wird. Der **<?CATCH>** Block veranlaßt dann einen Eintrag in der allgemeinen Logdatei.

```
<?TRY>
  <?SQL SQL="insert into foo values (42, 'bar')"
        THROW="INSERT_Exception">
  <?/SQL>
<?/TRY>

<?CATCH THROW="INSERT_Exception>
  <?LOG MSG="Konnte Datensatz nicht einfuegen"
        TYPE="Datenbank-Fehler">
<?/CATCH>
```

URLENCODE

Bereich:

URL- und Formular-Handling

Syntax:

```
<?URLENCODE VAR=zu_codierende_Variable
            [ MY ] ENCVAR=Ziel_Variable >
```

Beschreibung:

Mit dem **<?URLENCODE>** Befehl wird der Inhalt einer einzelnen Variablen URL codiert und in einer Zielvariablen zurückgegeben.

Wenn Variablen in eine URL zur Übergabe an ein CGI Objekt geschrieben werden sollen, so müssen diese Variablen URL codiert sein.

Parameter:

VAR
Der Parameter **VAR** gibt den Namen der zu codierenden skalaren Variablen an.

ENCVAR
Der Name der skalaren Zielvariablen wird mit dem Parameter **ENCVAR** angegeben.

MY
Optional kann der Schalter **MY** angegeben werden. In diesem Fall wird die Zielvariable implizit mit **my** deklariert, so daß sie nur bis zum Ende des den **<?URLENCODE>** Befehl umgebenden Blocks bekannt ist.

Beispiel:

Hier wird ein CGI Script auf einer externen Website aufgerufen, dabei wird der Parameter $query übergeben, der zur Sicherheit vorher URL-codiert wurde:

```
<?URLENCODE VAR=$query MY ENCVAR=$enc_query>
<A HREF="www.search.com?query=$enc_query">
Suche mir etwas
</A>
```

VAR

Bereich:

Variablen- und Gültigkeitsbereiche

Syntax:

```
<?VAR NAME=Variablenname
      [ MY ]
      [ DEFAULT=Vorgabewert ]
      [ NOQUOTE ]>
...
<?/VAR>
```

Beschreibung:

Dieser Befehl definiert eine Variable. Der Variablen wird das zugewiesen, was vom **<?VAR>** Block umschlossen wird. Dabei kann der Inhalt des **<?VAR>** Blocks als Zeichenkette oder als Perl-Ausdruck interpretiert werden.

Der **<?VAR>** Befehl kann nicht geschachtelt werden und darf auch keine anderen CIPP Befehle enthalten.

Parameter:

NAME

Mit diesem Parameter wird der Name der Variablen angegeben.

In Perl gibt es drei „Typen“ von Variablen: Skalar, Liste und Hash. Der Typ einer Variablen wird durch eines der folgenden Sonderzeichen ausgedrückt, welches vor den Variablennamen geschrieben wird: **$** steht für ein Scalar, **@** für eine Liste und **%** für ein Hash. Einzelheiten zu diesen drei Variablentypen entnehmen Sie bitte der Perl-Dokumentation.

Jeder dieser drei Variablentypen kann mit dem Befehl **<?VAR>** definiert werden, indem das entsprechende Sonderzeichen vor den Variablennamen gesetzt wird. Wird das Sonderzeichen weggelassen, so wird per Default ein **$** Zeichen, als eine skalare Variable, angenommen.

Wenn eine skalare Variable definiert wird, wird der in der Blockanweisung stehende Wert per Default als Zeichenkette interpretiert. Falls man dort aber einen Perl-Ausdruck verwenden möchte, muß dies mit dem Schalter

NOQUOTE gekennzeichnet werden. Bei den anderen Variablentypen Liste und Hash muß in der Blockanweisung ein Perl-Ausdruck stehen, der den entsprechenden Datentyp zurückliefert.

Im folgenden Beispiel wird die Definition der unterschiedlichen Variablentypen demonstriert:

```
<?VAR NAME=$skalar>Eine Zeichenkette<?/VAR>
<?VAR NAME=@liste>(1,2,3,4)<?/VAR>
<?VAR NAME=%hash>( 1 => 'a', 2 => 'b' )<?/VAR>
```

NOQUOTE

Dieser Schalter muß verwendet werden, wenn eine skalare Variable definiert werden soll, der Inhalt des **<?VAR>** Blocks aber nicht als Zeichenkette sondern als Perl Ausdruck interpretiert werden soll. Per Default werden intern bei der Variablenzuweisung immer doppelte Anführungszeichen generiert. **NOQUOTE** weist CIPP an, diese doppelten Anführungszeichen nicht zu generieren, so daß ein Perl Ausdruck auch wirklich ausgewertet wird:

```
<?VAR MY NAME=$element_cnt NOQUOTE>
  scalar(@liste)
<?/VAR>
```

DEFAULT

Wenn ein **DEFAULT** Parameter angegeben wird, so wird der Wert dieses Parameters der Variablen zugewiesen, wenn diese an der aktuellen Stelle den Wert **undef** hat. In diesem Fall wird also nicht der Wert des **<?VAR>** Blocks zugewiesen.

Folgende Konstruktion ermöglicht so die Generierung von Default Variablenwerten. Wenn **$event** an dieser Stelle definiert ist, wird **$event** sich selbst zugewiesen. Andernfalls erhält **$event** den Wert 'show':

```
<?VAR NAME=$event DEFAULT="show">$event<?/VAR>
```

MY

Wird der MY Schalter angegeben, ist die Variable bis zum Ende des den **<?VAR>** Befehl umgebenden Blocks gültig.

Die Verwendung von **MY** zusammen mit **DEFAULT** macht keinen Sinn, da **DEFAULT** nur wirken kann, wenn die Variable bereits vorher deklariert wurde.

Beispiel:

Die Variable **$beispiel** erhält den Wert des Skalars **$wert + 1**. Der Inhalt soll als Ausdruck interpretiert werden, denn sonst würde die Berechnung „$wert + 1“ als String interpretiert, was z.B. bei **$wert=5** die Zeichenkette „5 + 1“ ergeben würde. Das ist hier aber nicht beabsichtigt, denn es soll natürlich die Zahl 6 herauskommen.

```
<?VAR NAME=“$beispiel“ NOQUOTE>
  $wert + 1
<?/VAR>
```

Definition einer blocklokalen bzw. privaten Variablen:

```
<?VAR MY NAME=“$name“>Larry<?/VAR>
```

ist gleichbedeutend mit:

```
<?MY $vorname>
<?VAR NAME=“$name“>Larry<?/VAR>
```

Sinnvolle Anwendung für den **DEFAULT** Parameter (die Variable ist hier per Default eine skalare Variable, also **$event**):

```
<?VAR NAME=event DEFAULT=“show“>$event<?/VAR>
```

Definition einer lokalen Liste:

```
<?VAR MY NAME=@liste>(1, 2, 3, 4)<?/VAR>
```

Die Bildung einer Liste aus einer Zeichenkette mit Trennzeichen:

```
<?VAR MY NAME=$text>a:b:c:d<?/VAR>
<?VAR MY NAME=@liste>split („:“, $text)<?/VAR>
```

Definition eines lokalen Hashs:

```
<?VAR MY NAME=%hash>( ’a’ => 1, ’b’ => 2 )<?/VAR>
```

WHILE

Bereich:

Kontrollstrukturen

Syntax:

```
<?WHILE COND=Bedingung >
...
<?/WHILE>
```

Beschreibung:

Realisierung einer kopfgesteuerten bzw. abweisenden Schleife. Der Schleifenkörper bzw. der **<?WHILE>** Block wird nur solange ausgeführt, wie die Bedingung erfüllt ist. Die Bedingung wird schon vor der ersten Ausführung geprüft, so daß der **<?WHILE>** Block auch abgewiesen werden kann, wenn die Bedinung vor dem Ersten Durchlauf schon falsch liefert.

Parameter:

COND
Der **COND** Parameter gibt die Perl Bedingung an, die erfüllt sein muß, damit der **<?WHILE>** Block durchlaufen wird.

Beispiel:

Das Beispiel erstellt eine Tabelle mit Nameseinträgen aus den zwei gegebenen Arrays **@vorname**, **@nachname**:

```
<TABLE>
<?VAR MY NAME=$i>0<?/VAR>
<?WHILE COND="$i++ < scalar(@nachname)">
  <TR>
    <TD>$nachname[$i]</TD>
    <TD>$vorname[$i]</TD>
  </TR>
<?/WHILE>
</TABLE>
```