ePages 5

Handbuch für
Design und Cartridge-Entwicklung

- Version 5.04 -

Die in diesem Dokument enthaltenen Informationen können jederzeit ohne Benachrichtigung geändert werden.

Sollten Sie Fragen oder Hinweise zu unseren Produkten haben, so wenden Sie sich bitte an folgende Adresse:

ePages Software GmbH
Leutragraben 1
07443 Jena
Deutschland

Tel.: +49 (0) 36 41 / 5 73 – 10 0
Fax: +49 (0) 36 41 / 5 73 – 11 1

E-Mail: support@epages.de, pm@epages.de
WWW: www.epages.de

Jena, März 2007

Inhaltsverzeichnis

1. Einleitung 7
1.1 Inhalt des Handbuches 7
1.2 Voraussetzungen 8
1.3 Typographische Konventionen 8
Grundlagen 9
2. Objektorientierung 11
2.1 Vererbung 11
2.2 Objekt-API 12
3. Attribute 15
3.1 Sprachabhängige Attribute 16
3.2 Attribute mit vordefinierten Werten 17
3.3 Referenz-Attribute 17
3.4 Attribute hinzufügen 17
3.5 Attribut-API 18
4. Rechte und Rollen 21
4.1 Registrieren von Aktionen für Objekte 21
4.2 Zuweisen von Aktionen zu Rollen 22
4.3 Zuweisung von Rechten 23
5. Unterscheidung zwischen *User* und *Customer* 27
6. URL-Aktionen 31
6.1 ViewActions 31
6.2 ChangeActions 31
7. Templates 33
7.1 Technologie 33
7.2 Template-Prozess 33
7.3 Prinzipielle Struktur der Webseite 37
7.4 Überladung von Templates 38
7.5 Template-Debugging 38
8. PageType-Konzept 41
8.1 Logische Struktur 41
8.2 Darstellungsebene 43
8.2.1 Basistemplate 44
8.2.2 Template-Hierarchie 44
8.2.3 Objektmethode *template* 47
8.3 Verarbeiten der PageTypes 48
9. Mehrsprachigkeit – Language-Tags 51
9.1 Syntax für Language-Tags 51
9.2 Verwendung der XML-Sprachdateien 52
9.3 Überladen der XML-Sprachdateien 57
9.4 Lokalisierung von Datenbankinhalten 60
10. TLE 63
10.1 Syntax für TLE 63
10.2 TLE-Variablen 63
10.3 TLE-Anweisungen 65

10.3.1 #IF....66
10.3.2 #INCLUDE....67
10.3.3 #LOCAL....67
10.3.4 #SET....68
10.3.5 #GET....68
10.3.6 #CALCULATE....69
10.3.7 #WITH....69
10.3.8 #LOOP....69
10.3.9 #JOIN....70
10.3.10 #FUNCTION....70
10.3.11 #BLOCK....70
10.3.12 #WITH_LANGUAGE....70
10.3.13 #REM....71
10.4 Fehler-TLE....71
10.4.1 #FormError....71
10.4.2 #FormError_‹InputField›....71
10.4.3 #FORM_ERROR....72
10.4.4 #FormErrors.‹...›....72
10.4.5 #WITH_ERROR....72
10.4.6 #ERROR_VALUE....73
10.5 Formatierung von TLE-Variablen....73
10.6 Operatoren....76
10.7 Erstellen einer TLE-Funktion....77
10.8 Erstellen einer dynamischen TLE-Variable....79
10.9 Erstellen eines TLE-Formatters....81
Cartridge-Entwicklung....83
11. Cartridges....85
11.1 Struktur einer Cartridge....85
11.2 Anlegen einer Cartridge-Struktur....87
11.3 Installer / Cartridge.pm....87
11.4 Installieren - nmake....89
11.5 Deinstallieren....90
11.6 Cartridge-Verzeichnisse kopieren....91
11.7 Backoffice-Erweiterungen....91
12. Erstellung einer Distribution....93
12.1 Encryption....93
Weiterführende Konzepte....95
13. Anlegen von Features....97
14. Formhandling....99
14.1 Fehlerbehandlung für Objektattribute....99
14.2 Fehlerbehandlung für frei definierbare Formulare....99
14.2.1 Definition von FormFields....100
14.2.2 Verwendung von Forms im Perl-Code....101
14.2.3 Validierung bei nicht definierten Datentypen....102
14.2.4 Fehlerbehandlung im Template....103
15. Web Services....107
15.1 ePages–Web Services und Framework....107
15.2 ePages-Web Service generieren....108
15.2.1 Registrieren....108

15.2.2 Autorisierung 109
15.2.3 Implementation 110
15.3 Externe Clients für ePages 5-Web Services 110
15.4 ePages–Web Service-Client implementieren 112
16. Hooks 115
16.1 Bereitstellen eines Hooks 115
16.2 Funktionserweiterung per Hook 116
17. Import / Export von Datenbankinhalten 119
17.1 Import-Dateien 119
17.2 XML-Import 120
17.2.1 Spezialfall : standards.pl 121
17.2.2 Spezialfall: Hooks 121
17.2.3 Spezialfall: Forms 121
17.3 XML-Export 122
18. Scheduler 125
18.1 Konfigurieren von Perl-Script-Tasks 125
18.2 Einfügen von neuen Perl-Script-Tasks 126
18.3 Konfigurieren von UNIX-Shell-Script-Tasks 127
18.4 Starten und Stoppen 127
18.5 Scheduler-Task-Output 128
19. Diagnostics-Cartridge 131
19.1 Installation 131
19.2 Anwendung 131
Design 133
20. Styles 135
20.1 Auswahlstyles 135
20.1.1 Auswahlstyle anlegen 135
20.1.2 Imageset anlegen 137
20.1.3 Symbolset anlegen 138
20.1.4 Sub-Styles 139
Anhänge 141
Anhang A: Performance Tuning 143
21. Allgemeine Maßnahmen 143
21.1 Page Caching 143
21.2 Templateverarbeitung 143
21.3 Prozess-Prioritäten 144
21.4 Verkürzung Antwortzeit des ersten Requests 144
21.5 Debug-Informationen 145
21.6 Shop-Einstellungen 145
21.7 System-Monitoring mit Spy.pl 145
21.7.1 Installation 147
22. Maßnahmen während der Entwicklung 149
22.1 Template-Analyse 149
22.2 Partielles Caching 150
22.3 Nutzung der #LOCAL-Anweisung 152
22.4 Auslagern komplexer TLE-Blöcke 153
Anhang B: Entwicklerhinweise 155

23. E-Mail-Events hinzufügen 155
23.1 MailType anlegen 155
23.2 PageType anlegen und Template zuweisen 156
23.3 Funktion implementieren 156
23.4 Aktion registrieren und Permission definieren 156
24. Erweiterung von Cross-Selling-Typen 159
24.1 Tabelle definieren 159
24.2 Klassen anlegen 159
24.3 Produktattribute erweitern 160
24.4 Pagetypes und Templates anlegen 161
24.5 Aktionen registrieren/Funktionen implementieren 163
25. Anlegen von Shops per Web Service und Script 165
26. Patchen von Cartridges 167
27. Integration der eigenen Online-Hilfe 171
27.1 Bereitstellen der Hilfe-Seite 171
27.2 Zuweisen zur ViewAction 171
27.3 Anzeige-Code im Template 172
28. Dynamische Menüs 173
29. Warenkorb-Template und LineItems 177
29.1 LineItems 178
Anhang C: Anwendungsbeispiele (AWB) 183
30. AWB 1: Einbinden einer eigenen Stylesheet-Datei 183
31. AWB 2: Erweitern des Storefrontstyles 185
32. AWB 3: Änderungen im Template 187
33. AWB 4: Anpassung Backoffice-Design (Branding) 191
34. AWB 5: Deaktivieren des Design-Tools 193
35. AWB 6: Design-Änderungen über PageTypes 195
36. AWB 7: Neue Stapelverarbeitungsaktion im MBO 201
Glossar 207
Index 211

1. Einleitung

ePages 5 bietet als standardisierte Technologieplattform hohe Flexibilität und Erweiterbarkeit, wodurch sich kundenspezifische Anpassungen in kurzer Zeit umsetzen lassen.

Der große Funktionsumfang der Standardsoftware ist die Basis für die schnelle Umsetzung der Geschäftsmodelle bei niedrigen Betriebskosten.

Die Trennung von Kreativdesign und inhaltlich-funktioneller Gestaltung der Webseiten bietet neben dem Vorteil der effektiveren Bearbeitung der beiden Bereiche auch die Möglichkeit der Parallelbearbeitung und damit kürzere Projektlaufzeiten.

Die Flexibilität des Systems wird durch die Bereitstellung von Funktionen durch einzelne Softwarekomponenten, den Cartridges erweitert. Diese in sich gekapselten Funktionalitäten kommunizieren über definierte Programmierschnittstellen und sind wahlfrei kombinierbar.

Für den Entwickler steht somit die Aufgabe, funktionelle Erweiterungen als ebensolche Cartridges zur Verfügung zu stellen.

Das Handbuch beschreibt die Grundlagen zur Anpassung und Erweiterung von ePages 5 bezüglich Design und Cartridge-Funktionalität. Auf Grund der Komplexität des Systems kann es ein Entwickler-Training nicht ersetzen, sondern ist als Ergänzung dazu zu sehen.

Zu aktuellen Trainingsangeboten können Sie sich auf unserer Webseite www.epages.de informieren.

1.1 Inhalt des Handbuches

Dieses Handbuch beschreibt die Strukturen und Konzepte, die ePages 5 zu Grunde liegen. Anhand von grundsätzlichen Erläuterungen in Zusammenhang mit Anwendungsbeispielen sollen Designer und Entwickler in die Lage versetzt werden, die ePages-Standard-Installation ihren Bedürfnissen und Anforderungen anzupassen.

In Teil I: *Grundlagen* ab *Seite 9* werden wesentliche Grundlagen und Konzepte erläutert, die sowohl für Entwickler als auch für Designer von Bedeutung sind.

Teil II: *Cartridge-Entwicklung* ab *Seite 83* beschreibt die grundlegende Vorgehensweise zur Erstellung von Cartridges. Der Entwickler soll in die Lage versetzt werden, das System um eigene Funktionen erweitern zu können, die in eigene Cartridges gekapselt sind.

In Teil III: *Weiterführende Konzepte* ab *Seite 95* werden auf den vorangehenden Kapiteln aufbauende Themen zur effektiven Nutzung des Systems beschrieben.

In Teil IV: *Design* ab *Seite 133* liegt der Schwerpunkt auf Anpassung bezüglich Design und Layout. Dabei wird der Umgang mit dem System hinsichtlich Webseitengestaltung beschrieben. Dabei wird nicht auf das Entwerfen von Webseiten aus der Sicht des Kreativdesigns eingegangen.

Anhang A: Performance Tuning ab *Seite 143* gibt Hinweise zur Erhöhung der Leistungsfähigkeit Ihrer Installation.

Anhang B: Entwicklerhinweise ab *Seite 155* gibt Hilfe zu Problemstellungen, die in der Praxis immer wieder auftauchen.

In *Anhang C: Anwendungsbeispiele (AWB)* ab *Seite 183* finden Sie Beispiele, welche die Erläuterungen in einzelnen Kapiteln unterstützen. In den entsprechenden Kapiteln wird auf das dazugehörige Beispiel verwiesen. Zu jedem Anwendungsbeispiel sind die Quelltexte verfügbar. Mit deren Hilfe können die Beispiele Schritt für Schritt nachvollzogen werden. Um die Funktionsfähigkeit der Cartridges zu gewährleisten, müs-

sen diese in einem Verzeichnis *%EPAGES_CARTRIDGES%/Training* angelegt werden. Alle Cartridge-Beispiele sind so vorbereitet, dass sie an die entsprechende Stelle kopiert und dann installiert werden können, falls man nur die Funktionalität nachvollziehen will.

1.2 Voraussetzungen

Das Handbuch wendet sich an Entwickler und Web-Designer, die bestehende ePages 5-Installationen anpassen, erweitern oder optimieren wollen.

Es wird davon ausgegangen, dass Designer über Kenntnisse in HTML/XHTML, Javascript, CSS und XML verfügen, während für die Cartridge-Entwicklung darüber hinaus Erfahrungen in Perl und SQL notwendig sind.

Weiterhin wird eine lauffähige ePages 5-Standardinstallation vorausgesetzt mit entsprechenden Zugriffsrechten auf Storefront und Backoffice, sowie Datenbanken und Dateisystem.

Weiterhin von Vorteil ist die Kenntnis der Handbücher für Händler, Technischen Administrator und Business-Administrator.

1.3 Typographische Konventionen

Folgende Schriftarten und Formatierungen werden verwendet, um auf spezielle Informationen hinzuweisen:

`nmake install`	Programmiercode und Anweisungen; Mehrzeilige Codebeispiele sind eingerahmt.
`perl import.pl [-help]`	Parameter in eckigen Klammern in Programmaufrufen sind optional.
‹text›	Text in spitzen Klammern, der mit Kleinbuchstaben beginnt, ist als Platzhalter zu sehen und durch aktuelle Parameter zu ersetzen.
All Hooks	Bezug auf Hyperlinks in den Abbildungen der ePages-Anwendung. So gekennzeichnete Links stehen in der Anwendung selbst zur Verfügung.
$hValues	Diese Formatierung weist auf spezielle Namen und Bezeichner, wie Dateinamen, Pfadnamen, Feldnamen und auf Eingabewerte hin.
#(Kontext.)attributname	Anzuwendende allgemeine Syntax für die angegebenen Anweisungen

Hinweis: Nützliche Informationen, die beachtet werden sollten, um effektiv zu arbeiten, werden in Kästen wie diesem angezeigt.

Achtung: Wichtige Informationen, die beachtet werden müssen, werden in Kästen wie diesem angezeigt.

Teil I:

Grundlagen

2. Objektorientierung

ePages 5 nutzt die Vorteile der objektorientierten Programmierung mit Perl.

Dabei sollten folgende Anforderungen und Zielstellung erfüllt werden:

- Konsistente API für die Arbeit mit Objekten und Attributen
- Verwendung von sprachabhängigen Attributen
- Attribute können in unterschiedlichen Tabellen gespeichert werden
- Vererbung von Attributen und Methoden
- Cartridges können neue Attribute zu existierenden Klassen hinzufügen

Dafür wurde eine Klassenstruktur mit den entsprechenden Vererbungsmechanismen entwickelt. Für jede Klasse sind Attribute und Methoden definiert, die vom jeweiligen Objekt verwendet werden können.

Die Methoden einer Klasse sind in einem Perl-Modul implementiert, welches Class-Package genannt wird. Die Attribute werden auch in einem Perl-Modul implementiert, welches aber unabhängig von Class-Package ist. Dadurch können die Attribute für eine Klasse erweitert werden, ohne dass das Class-Package geändert werden muss.

Die Zuordnung von Attributen zu Klassen ist in der Datenbank gespeichert. Die Datenbankinhalte können aus XML-Dateien wie z. B. der *Attributes*.xml* einer jeden Cartridge importiert werden. Siehe dazu auch *Import / Export von Datenbankinhalten*, *Seite 119*.

Eine Übersicht über Klassenstruktur gibt die Diagnostics-Cartridge, siehe *Diagnostics-Cartridge*, *Seite 131*. Die Struktur ist dynamisch erweiterbar und kann in verschiedenen Datenbanken unterschiedlich sein.

Jedes Objekt ist eine Instanz der jeweiligen Klasse. Alle Objekte sind in einer Baumstruktur organisiert. Basis dieser Struktur ist das Objekt *System*. Jedes Objekt kann eine beliebige Anzahl von Child-Objekten haben.

Jedes Objekt wird durch drei jeweils eindeutige Bezeichner beschrieben:

Tabelle 1: Objektbezeichner

Bezeichner	Bedeutung
Objectpath+Alias	Eindeutiger Alias in Zusammenhang mit dem übergeordneten Objekt. Dadurch werden alle Objekte durch die Angabe des Objektpfades, ausgehend von System-Objekt, eindeutig gekennzeichnet. Ein Beispiel ist *System/Shops/DemoShop/Users/admin*. Bei der Angabe des Objektpfades kann System weggelassen werden, so dass das Beispiel wie folgt geschrieben werden kann: */Shops/DemoShop/Users/admin*
Object ID	Eindeutige Nummer in der Datenbank
GUID	Globally Unique IDentifier - Kennung, die allgemein eindeutig ist. Sie kann zur Identifizierung von Objektreferenzen in externen Systemen verwendet werden.

Alle drei Werte können mit der Diagnostics-Cartridge ausgelesen werden.

2.1 Vererbung

Von jeder Klasse können Sub-Klassen abgeleitet werden. Diese Sub-Klassen erben alle Attribute und Methoden von der übergeordneten Klasse und aller wiederum dieser übergeordneten Klassen.

Jede Klasse kann nur eine Basisklasse haben, Mehrfachvererbung ist nicht möglich. Die Basisklasse der gesamten Hierarchie ist die Klasse *BaseObject*. Alle anderen Klassen sind Sub-Klassen, sie haben genau eine übergeordnete Basisklasse.

In Sub-Klassen können die Attribute und Methoden der Basisklasse überschrieben werden. Soll innerhalb einer Klasse auf eine Methode der Basisklasse zugegriffen werden, muss folgende Syntax verwendet werden:

```
$self->SUPER::method(@params);
```

2.2 Objekt-API

Die Objekt-API ist das allgemeine Interface für alle Objekte. Nutzen Sie diese API zum Erzeugen und Löschen von Objekten und dem Setzen und Auslesen von Attributwerten. Die API stellt u. a. folgende Methoden zur Verfügung:

Tabelle 2: Methoden der Objekt-API

Methode	Bedeutung
insert	Erzeugt ein neues Objekt und triggert den Hook *OBJ_Insert‹ClassName›*. Für die meisten Objekte müssen die Parameter Parent und Alias in den Hash *$hValues* übergeben werden.
delete	Löscht ein Objekt und triggert den Hook *OBJ_Delete‹ClassName›*
load	Initialisiert ein vorhandenes Objekt
id	Gibt die Objekt-ID zurück
get	Gibt einen oder mehrere Attributwerte zurück
set	Setzt mehrere Attributwerte und triggert die Hooks *OBJ_BeforeUpdate‹ClassName›* und *OBJ_AfterUpdate‹ClassName›*.

Das Package *DE_EPAGES::Object::API::Object::Object* stellt eine Basis-Implementation für Objekte zur Verfügung, die in einer relationalen Datenbank gespeichert werden.

Das Package *DE_EPAGES::Object::API::Object::Factory* stellt Methoden für den Zugriff auf existierende Objekte und Klassen sowie für das Einfügen und Löschen von Objekten bereit. Die gebräuchlichsten Funktionen sind:

Tabelle 3: Funktionen des Moduls *Factory*

Funktion	Bedeutung
InsertObject	Erzeugt ein neue Instanz eines Objekts
DeleteObject	Löscht bei Angabe der ObjectID das entsprechende Objekt
LoadObject	Gibt bei Angabe der ObjectID das entsprechende Object zurück
LoadObjectByPath	Gibt bei Angabe eines Objektpfades das entsprechende Objekt zurück.
LoadRootObject	Gibt das System-Objekt zurück
LoadClassByAlias	Gibt bei Angabe des Namens die entsprechende Klasse zurück

Codebeispiel 1 zeigt die Anwendung der Object-API:

```
use DE_EPAGES::Object::API::Factory qw( InsertObject LoadObjectByPath );
use DE_EPAGES::Object::API::Language qw( GetPKeyLanguageByCode );

# Load a new object with object path specification
my $Shop = LoadObjectByPath( '/Shops/DemoShop' );

# Load a child object
my $Folder = $Shop->child( 'Products' );

# Create a new objects of a given class
my $Product = InsertObject( 'Product', {Parent => $Folder,Alias => '0815'} );

# Set attribute values for a new object
$Product->set( { IsVisible => 1, Weight => 12.05} );

# Set language dependent attribute values
my $LanguageID_en = GetPKeyLanguageByCode( 'en' );
my $LanguageID_de = GetPKeyLanguageByCode( 'de' );

$Product->set( {
        Name => 'Example Product',
        Description => 'Example Description'
    },
    $LanguageID_en
);

$Product->set( {
        Name => 'Produktname',
        Description => 'Produktbeschreibung'
    },
    $LanguageID_de
);

# Read out attribute values
my $IsVisible = $Product->get( 'IsVisible' );
my $hAttributesDE = $Product->get( [ 'Name', 'Description' ], $LanguageID_de );

{
    local $DE_EPAGES::Object::API::Language::LANGUAGEID = $LanguageID_en;
    my $hAttributes = $Product->get( [ 'Name', 'Description', 'IsVisible' ] );
}

# Delete object
$Product->delete;
```

Codebeispiel 1: Funktionen der Objekt-API

3. Attribute

Attribute beinhalten primär die Eigenschaften und deren Typen von Objekten. Sie sind als Objekte definiert, die Basisklasse ist *Object*. Dadurch erben sie alle Attribute der Basisklasse. Das bedeutet, dass Attribute wiederum Attribute haben. Diese zusätzlichen Attribute werden verwendet, um die Eigenschaften eines Attributes zu beschreiben. Standardmäßig werden verwendet:

Tabelle 4: Attribute zur Beschreibung von Attribut-Eigenschaften

Funktion	Bedeutung
Name	Sprachabhängiger Name, der z. B. für Dokumentationszwecke benutzt wird
Description	Sprachabhängige Beschreibung , die z. B. für Dokumentationszwecke benutzt wird
Type	Bezeichnung für den Datentyp des Attributes; Spezifiziert das Attribut eine Objekt-Referenz oder eine Liste von Referenzen, dann ist der Attributname der Name der Objektklasse der Werte. Referenziert das Attribut auf den Typ *Object*, können Referenzen auf Objekte jeder Klasse enthalten sein.
Length	Maximale Länge; Wird nur für Zeichenketten und sprachabhängigen Zeichenketten verwendet
Package	Name des Perl-Packages, über das auf die Attributwerte zugegriffen wird
IsArray	Gibt an, dass das Attribute mehrere Werte als Array zurück gibt Standardeinstellung: *0*
IsCachable	Das Attribut kann durch das Objekt zwischengespeichert werden. Standardeinstellung: *0*, sprachabhängige Attribute sind generell nicht "cachebar"
IsExportable	Soll mit dem Objekt exportiert werden. Standardeinstellung: *0*
IsMandantory	Definition als Pflichtfeld, d. h. ein Wert muss vorhanden sein; Standardeinstellung: *0*
IsObject	*1* - Der Attributwert ist eine Objektreferenz oder eine Liste von Objekten, über die Objekt-ID erfolgt ein Mapping auf das Objekt. *0* – Attribut ist ein Wert Standardeinstellung: *0*
IsReadOnly	Kann nur gelesen werden.

In ePages 5 werden diese Attribut-Typen verwendet:

- Standardattribute,
- sprachabhängige Attribute,
- Attribute mit vordefinierten Werten,

Folgende Standardattribute werden unterstützt:

Tabelle 5: Standardattribute

Type	Bemerkung
Integer	Integer mit Vorzeichen (32 Bit)
Float	
Boolean	1=true, 0=false
Money	Dezimalzahl fester Länge Der Sybase-Datentyp *money* ist definiert als numeric(19,4), d. h. es können Zahlen bis 15 Zeichen vor dem Dezimaltrenner und 4 Stellen nach dem Trenner gespeichert werden.

Type	Bemerkung
DateTime	Datum und Uhrzeit, basiert auf der Perl-Klasse DateTime Der Sybase-Datentyp datetime erlaubt Werte im Bereich vom 01.01.1753 bis zum 31.12.9999.
Date	Basiert auf der Perl-Klasse DateTime. Dabei wird nur die Datumsangabe genutzt. Der Zeit-Teil ist auf 0:00:00 gesetzt.
Time	Basiert auf der Perl-Klasse DateTime. Dabei wird nur die Zeitangabe genutzt. Der Datums-Teil ist undefiniert.
String	Unicode-Text von beliebiger Länge Kurze Texte werden in der Datenbank in Feldern vom Typ NVARCHAR gespeichert. Sind die Text länger als VARCHAR zulässt, werden die Text in Feldern vom Typ TEXT in einer anderen Tabelle gespeichert.
File	Dateinamen oder URL

3.1 Sprachabhängige Attribute

Sprachabhängige Attribute sind Attribute gleichen Namens, die für verschiedene Sprachen unterschiedliche Werte beinhalten. Typisches Beispiel sind Produktnamen oder –beschreibungen.

Die Basis-Implementation für sprachabhängige Attribute finden Sie im Package *DE_EPAGES::Object::API::Attributes::LocalizedAttribute*.

Folgende Typen stehen zur Verfügung:

Tabelle 6: Typen von sprachabhängigen Attributen

Type	Bemerkung
LocalizedString	Analog Typ *String*
LocalizedFile	Analog Typ *File*

Bei der Übergabe der sprachabhängigen Inhalte müssen Sie jeweils mit angeben, für welche Sprache die Werte übergeben werden. Dafür gibt es zwei Möglichkeiten:

1. Setzen einer globalen Variable, welche die Standardsprache für alle folgenden Attribut-Funktionen festlegt, siehe *Codebeispiel 2*,
2. Angabe der Sprache direkt zu jeder set()- oder get()-Methode für Attribute, siehe *Codebeispiel 3*.

```
use DE_EPAGES::Object::API::Language qw( GetPKeyLanguageByCode );
use utf8;

# set the language globally
{
  local $DE_EPAGES::Object::API::Language::LANGUAGEID = GetPKeyLanguageByCode(
'de' );
  $Product->set( { Name => 'epages 5 für Einzelhändler' } );
}
…
```

Codebeispiel 2: Globales Setzen der Sprache

```
use DE_EPAGES::Object::API::Language qw( GetPKeyLanguageByCode );
use utf8;

# pass the language directly to the attribute function
my $LanguageID = GetPKeyLanguageByCode( 'de' );

$Product->set( { Name => 'epages 5 für Einzelhändler' }, $LanguageID );
…
```

Codebeispiel 3: Definieren der Sprache pro Funktion

3.2 Attribute mit vordefinierten Werten

Für diese Attribute können beliebig viele Werte vergeben werden. Sie werden benutzt, um dem Nutzer eine Auswahlmöglichkeit aus einem vorgegebenen Wertevorrat anzubieten. Beispiele hierfür sind Auswahlboxen für Produkttyp- oder Kundenattribute.

Die Wertevorräte können sprachabhängig oder sprachunabhängig aufgebaut werden. Es gibt folgende Attributtypen, die auf *String* aufsetzen:

- PreDefString
- PreDefLocalizedString
- PreDefMultistring
- PreDefMultiLocalizedString

3.3 Referenz-Attribute

Neben einfachen Werten können Attribute auch Referenzen auf ein Objekt oder auf eine Liste von Objekten enthalten.

Wie diese Referenzen an Attribute übergeben werden, sehen Sie im *Codebeispiel 4*.

```
use DE_EPAGES::Object::API::Factory qw( LoadObjectByPath );

# find an object by path
my $Product = LoadObjectByPath( '/Shops/DemoShop/Products/0815' );

# get reference attributes
my $Shop = $Product->get( 'Shop' );
my $aRelatedProducts = $Product->get( 'RelatedProducts' );

# set reference attributes
$Product->set({ 'Shop' => $Shop });
$Product->set({ 'RelatedProducts' => [ $Product1, $Product2 ] });
```

Codebeispiel 4: Übergabe von Referenzen an Attribute

3.4 Attribute hinzufügen

Bei Bedarf können Sie zu Klassen neue Attribute hinzufügen. Prinzipiell gibt es dafür zwei Wege

1. Zuweisen der neuen Attribute zu einer existierenden Klasse. Damit stehen diese Attribute allen Instanzen dieser Klasse zur Verfügung.
2. Sie erzeugen eine neue Sub-Klasse und weisen dieser Klasse neue Attribute zu. Diese Methode bietet sich an, wenn die neuen Attribute nicht auf die Instanzen der existierenden Klasse anwendbar sind.

Codebeispiel 5 zeigt, wie über die Objekt-API neue Attribute angelegt werden können:

```
use DE_EPAGES::API::Object::Factory qw( LoadClassByAlias );

my $BaseClass = LoadClassByAlias( 'LineItemShipping' );
my $NewClass = $BaseClass->insertSubClass( { Alias => 'LineItemUPS' } );
$NewClass->insertAttribute( {
                                Alias => 'Distance',
                                Type => 'Float',
                                Package =>
'DE_EPAGES::Object::API::Attributes::DefaultAttribute'
                              }
                            );
…
```

Codebeispiel 5: Anlegen eines neuen Attributes

Im Beispiel wird von der Klasse *LineItemShipping* die neue Klasse *LineItemUPS* abgeleitet. Für die neue Klasse wird ein neues Attribut *Distance* vom Typ *Float* angelegt. Alle Funktionen bezüglich dieses Attributes sind im angegebenen Package implementiert.

Ein weiterer Weg, Attribute anzulegen, ist die Definition der Attribute in der Datei *Attributes*.xml* und dem Import dieser Datei. Siehe dazu auch *Import / Export von Datenbankinhalten*, *Seite 119*.

3.5 Attribut-API

Die Standard-Implementation von Attributen auf Basis einfacher Datentypen, sprachabhängiger Zeichenketten oder Objekt-Referenzen sind für viele Anwendungen ausreichend.

In manchen Fällen kann es notwendig sein, eigene Attribut-Zugriffsfunktionen zu entwerfen. Solche Fälle sind z. B.

- wenn nach Attributen gesucht werden soll. Die Implementation der Standard-Attribute erlaubt keine Indexierung der Attributwerte für die Suche. Für die Suche nach Attributwerten sollten diese in einer separaten Tabelle gespeichert werden, für die ein passender Index angelegt werden kann.
- wenn Attributwerte auf Basis anderer Attribute berechnet werden sollen
- wenn die Attributwerte in einer bestehenden Tabelle oder in einer externen Datenbank gespeichert werden sollen

Für solche Fälle müssen Sie eine eigene Attribut-API in einem Perl-Modul implementieren. Die folgenden Funktionen müssen enthalten sein:

Tabelle 7: Funktionen in der eigenen Attribut-API

Funktion	Bemerkung
getAttribute	Gibt einen einzelnen Attributwert zurück
getAttributes	Gibt mehrere Attributwerte in Form eines Hash zurück. In Abhängigkeit von der Datenbank kann diese Funktion schneller sein als getAttribute. Das Feld $aNames enthält nur die Bezeichner (Alias) der Attribute, die in diesem Package implementiert sind.
setAttribute	Setzt ein einzelnes Attribute
setAttributes	Setzt mehrere Attribute mit einem Mal. In Abhängigkeit von der Datenbank kann diese Funktion schneller sein als setAttribute. Der Hash $hValues enthält nur die Attribute, die in diesem Package implementiert sind.
deleteObject	Diese Methode wird aufgerufen, wenn ein Objekt gelöscht wird. Das Feld $aNames enthält die Namen aller Attribute, die auf das zu löschende Objekt anwendbar sind und in diesem Package implementiert sind.
deleteAttribute	Diese Methode wird aufgerufen, wenn eines der Attribute gelöscht wird, z. B. wenn eine Cartridge deinstalliert wird.

Funktion	Bemerkung
getAllAttributes	Gibt einen Hash aller Attributwerte zurück. Sprachabhängige Attribute werden wiederum als Hash zurückgegeben.
defaultAttributes	Gibt einen Hash mit den Standard-Attributen zurück. Diese werden mit benutzt, wenn ein neues Objekt angelegt und die Werte nicht durch InsertObject() übergeben werden.

Das Setzen eines Attributwertes auf *undef* führt zum Löschen des entsprechenden Wertes in der Datenbank.

Attribute, die Objektreferenzen zurückgeben, arbeiten jeweils nur mit der Objekt-ID.

Mit dem Modul *DE_EPAGES::Object::API::BaseAttribute* existiert eine Basis-Implementierung, so dass man nicht alle Funktionen selbst implementieren muss.

4. Rechte und Rollen

Damit ein Nutzer Aktionen ausführen kann, müssen an ihn die entsprechenden Berechtigungen vergeben werden. Diese Berechtigungen werden für ein bestimmtes Objekte vergeben und gelten auch für alle untergeordneten Objekte

Es muss also ein Zusammenhang zwischen Recht, Nutzer, Aktion und Objekt hergestellt werden. Nutzern werden bestimmte Aktionen auf bestimmte Objekte erlaubt oder verboten.

Prinzipiell kann so für jeden Nutzer jede Aktion auf jedem Objekt erlaubt werden. Aufgrund der großen Anzahl von Objekten, Aktionen und Nutzern ist diese Vorgehensweise zu aufwändig. Zur Vereinfachung wurden daher Rollen und Gruppen eingeführt und die Vererbung genutzt.

Eine *Rolle* ist die Zusammenfassung von mehreren zusammengehörigen Aktionen, d. h. diese Aktionen werden häufig zusammen ausgeführt oder bedingen einander. Bei der Zuweisung von Rechten sollten bevorzugt Rollen und nicht einzelne Aktionen verwendet werden.

In einer *Gruppe* werden mehrere Nutzer zusammengefasst. Dieses Verfahren wenden Sie an, wenn Sie verschiedenen Nutzern gleiche Rechte geben wollen. Sie vergeben die Rechte an die Gruppe und ordnen die betreffenden Nutzer der Gruppe zu.

Um die Rechte nicht an viele einzelne Objekte vergeben zu müssen, wird die *Vererbung* genutzt. Rechte können von einem übergeordneten Objekt geerbt werden. Die Rechte werden dabei nicht physikalisch kopiert, sondert es wird lediglich eine Vererbungsbeziehung hergestellt. Infolgedessen werden Änderungen der Rechte auf dem übergeordneten Objekt automatisch für alle untergeordneten Objekte wirksam. Die Vererbungsbeziehung wird sofort beim Anlegen eines Objekts hergestellt.

Durch die verschiedenen Möglichkeiten zum Zusammenfassen von Nutzern, Aktionen und Objekten ist es möglich, dass ein Nutzer gleichzeitig mehrere Rechte für die gleiche Aktion auf einem Objekt hat. Es ist sogar denkbar, dass die Aktion gleichzeitig erlaubt und verboten ist. Um solche Konflikte aufzulösen, werden bei der Kombination von Rechten folgende Regeln angewandt:

- was nicht explizit erlaubt ist, ist verboten
- explizites Verbot geht über Erlaubnis

Um Aktionen Nutzern zur Verfügung zu stellen, sollten Sie prinzipiell wie folgt vorgehen:

1. Implementieren und Registrieren einer Aktion
2. Zuordnen der Aktion zu einer Rolle
3. Die Rolle einem Nutzer oder einer Gruppe auf einem Objekt zuweisen

4.1 Registrieren von Aktionen für Objekte

Aktionen werden in einer Datei *Actions*.xml* definiert und beim Import in der Datenbank registriert. Für jede Cartridge kann es mehrere solcher Dateien geben, sie müssen im Cartridge-Verzeichnis */Database/XML* liegen.

Die Definition einer Aktion sehen Sie in *Codebeispiel 6*.

```
<?xml version="1.0" encoding="UTF-8"?>
<epages>
  <Class reference="1" Path="/Classes/Shop">
    <Object Alias="Actions">

      <Action Alias="MBO-ViewKelkooConfigs"
        Package="DE_EPAGES::Kelkoo::UI::KelkooConfig" FunctionName="View"
        delete="1">
        <AttributeValue Name="HelpFileTopic"
            Value="MBO/index.htm?single=true&amp;context=MBO_Help&amp;
                   topic=MBO_Marketing_Kelkoo" />
        <AttributeValue Name="Name" Language="de" Value="Kelkoo - Allgemein" />
        <AttributeValue Name="Name" Language="en" Value="Kelkoo - General" />
      </Action>

      <Action Alias="NewKelkooConfig"
        Package="DE_EPAGES::Kelkoo::UI::KelkooConfig"
        delete="1"
      />
    </Object>
  </Class>
</epages>
```

Codebeispiel 6: Definition einer Aktion

Zuerst legen Sie fest, für welche Klasse die Aktion definiert wird. Das Tag *Object* mit dem Alias *Actions* stellt einen Folder dar, in den die Aktionen eingetragen werden.

Jede Aktion wird einzeln definiert. Per *Alias* wird der Bezeichner festgelegt. Dieser Name wird auch im Template verwendet. In Package geben Sie an, wo die Perl-Funktion für die Aktion implementiert ist. Der Parameter *FunctionName* gibt den Namen der Funktion im Perl-Modul an, die aufgerufen wird, falls *Alias* und Perl-Funktionsname unterschiedlich sind. Sind beide Namen identisch, kann der Parameter *FunctionName* weggelassen werden, siehe zweite Aktionsdefinition im Beispiel.

Für Anzeigeaktionen (ViewActions) können Attribute vergeben werden. *HelpFileTopic* verweist auf die entsprechende Online-Hilfe zu dieser Aktion. Mit Hilfe von *Name* können Sie sprachabhängige Namen für die Aktion vergeben. Dieser Name wird im MBO im *Verlauf* angezeigt.

4.2 Zuweisen von Aktionen zu Rollen

Rollen und Berechtigungen werden in einer Datei *Permissions*.xml* definiert und beim Import in der Datenbank registriert. Für jede Cartridge kann es mehrere solcher Dateien geben, sie müssen im Cartridge-Verzeichnis */Database/XML* liegen.

Standardmäßig sind bereits vier Rollen angelegt: *Merchant*, *Customer*, *WebService*, *User*. Diese Rollen sind auf verschiedenen Objekten definiert. Da diese Rollen für die Anwendung ausreichend sind, werden Sie in der Praxis die Berechtigungen den vorhandenen Rollen zuordnen, wie in *Codebeispiel 7*.

```
<?xml version="1.0" encoding="UTF-8"?>
<epages>
  <Role reference="1" Path="/Classes/Shop/Roles/Merchant">
    <RoleAction Class="Shop"  Action="MBO-ViewKelkooConfigs"  delete="1" />
    <RoleAction Class="Shop"  Action="NewKelkooConfig"        delete="1" />

  </Role>
</epages>
```

Codebeispiel 7: Zuweisung einer Aktion zu einer Rolle

Sie referenzieren auf die Rolle, der Sie die Aktion zuweisen wollen. Geben Sie dabei den Objektpfad für die Rolle an.

Für die Aktion selbst geben Sie den Namen der Aktion an und die Klassen, zu der die Aktion gehört, vergleiche *Codebeispiel 6*.

Für den Fall, dass Sie eine neue Rolle anlegen müssen, orientieren Sie sich an *Codebeispiel 8*.

```
<?xml version="1.0" encoding="UTF-8"?>
<epages>

  <Class reference="1" Path="/Classes/Shop">

    <Object Alias="Roles">

      <Role Alias="Merchant" delete="1">
        <RoleAction Class="Object" Action="Delete" />
        <RoleAction Class="Object" Action="DeleteFile" />
        <RoleAction Class="Object" Action="SetInvisible" />
…
        <RoleAction Class="Shop"   Action="AddCurrency" />
        <RoleAction Class="Shop"   Action="AddLanguage" />
…
        <RoleAction Class="User"   Action="MBO-ViewUserSettings" />
        <RoleAction Class="User"   Action="SaveUserSettings" />
      </Role>

      <Role Alias="Customer" delete="1">
        <RoleAction Class="Shop"   Action="View" />
      </Role>

      <Role Alias="WebService" delete="1" />

    </Object>

  </Class>

</epages>
```

Codebeispiel 8: Anlegen von Rollen

Sie referenzieren auf die Klasse, für die Sie die Rolle anlegen wollen. Im Objektfolder *Roles* definieren Sie Sie die Rolle mit dem entsprechenden Alias. Im Beispiel oben wird so die Rolle *System/Classes/Shop/Roles/Merchant* definiert.

Innerhalb der Rolle werden alle Aktionen einzeln aufgelistet, mit Angabe der Klasse, für welche diese Aktion registriert ist.

Für das Anlegen von Rollen verwenden Sie auch die Datei *Permissions*.xml*.

4.3 Zuweisung von Rechten

Letztendlich müssen Gruppen oder Nutzer die Berechtigung erhalten, die definierten Aktionen auszuführen, d. h. es müssen die entsprechenden Rechte zugewiesen werden.

Die Rechtezuweisung an eine Gruppe sehen Sie in *Codebeispiel 9*, Sie benutzen wiederum die Datei *Permissions*.xml*.

```
<?xml version="1.0" encoding="UTF-8"?>
<epages>

  <!-- permissions -->
  <Group Alias="Everyone">
    <Object reference="1" Path="/">
      <Permission Class="Shop" Role="Customer" Allow="1" />
    </Object>
  </Group>

</epages>
```

Codebeispiel 9: Rechtevergabe an eine Gruppe

Wird die Gruppe neu angelegt, vergeben Sie einen Alias, anderenfalls referenzieren Sie auf eine vorhandene Gruppe.

In *‹Group›* definieren Sie, welche Nutzer die Berechtigung bekommen sollen. In *Object* legen Sie fest, für welches Objekt die Berechtigungen vergeben werden.

Über *Permission* definieren Sie, für welche Rolle die Mitglieder der angegebenen Gruppe berechtigt sind. Die genaue Rollen-Bezeichnung ergibt sich aus den Angaben in *Class* zusammen mit dem Rollen-Alias.

In *Codebeispiel 9* bedeutet dies: Die Mitglieder der Gruppe *Everyone* sind berechtigt, alle Aktionen der Rolle *Customer* der Klasse *Shop* auf dem Objekt *System* auszuführen. Das Objekt *System* wird durch */* in *Path* repräsentiert.

Über das Attribut *Allow=1* wird das Recht zur Ausführung explizit erteilt. Solange *Allow* nicht gesetzt ist, ist eine Aktion implizit verboten.

Mit *Allow=0* können Sie eine Aktion für einen Nutzer ausdrücklich verbieten, für den Fall, dass diese Aktion über eine andere Rolle erlaubt wurde.

Rechte können auch für einzelne Nutzer vergeben werden. Prinzipiell passiert dies beim Anlegen eines Nutzers. Das bedeutet, wenn sich ein Kunde im Shop registriert, wird für diesen ein Nutzer mit der Rolle *User* angelegt. Wird in der Benutzerverwaltung im Backoffice ein Nutzer angelegt, erhält dieser die Rolle *Merchant* zugewiesen.

Wie Sie einen Nutzer per XML-Datei anlegen und ihm dabei Rechte zuweisen können, sehen Sie in *Codebeispiel 10*.

```
<?xml version="1.0" encoding="UTF-8"?>
<epages>
  <Object Alias="Users" Position="120">

    <User Alias="admin" Password="zruzfz" Name="Shop-Administrator"
          DeleteConfirmation="1" delete="1" >
      <Permission Class="User" Role="User" Allow="1" />
      <Object reference="1" Path="../..">
        <Permission Class="Shop" Role="Merchant" Allow="1" />
        <Permission Class="Shop" Role="Customer" Allow="1" />
        <Permission Class="Shop" Role="WebService" Allow="1" />
      </Object>
   </User>

  </Object>
</epages>
```

Codebeispiel 10: Anlegen eines Nutzers mit Rechtezuweisung

Ein Nutzer wird unter Angabe des Alias, eines Kennwortes und eines Namens als User im Folder *Users* angelegt. Dazu erhält er die Rechte der Rolle *User*, die für die Klasse *User* definiert ist. Diese Rolle erlaubt ihm z. B. das Ändern des Kennwortes und weiterer persönlicher Daten.

Darüber hinaus können Sie ihm weitere Berechtigungen für Rollen auf anderen Objekten zuweisen. In unserem Beispiel bekommt der User *admin* noch die Rechte für alle Aktionen in den Rollen *Merchant*, *Customer* und *WebService*, die für das Shop-Objekt definiert sind.

5. Unterscheidung zwischen *User* und *Customer*

Für ePages 5 wurde ein User-Customer-Konzept entwickelt, auf dessen Basis verschiedene Geschäftsmodelle abgebildet werden können. Diesem Konzept wurden drei Szenarien zu Grunde gelegt: Shop-Szenario, Marktplatz-Szenario und Procurement-Szenario.

Grundidee des Konzeptes ist die Unterscheidung zwischen User und Customer, die jeweils eine andere Sicht repräsentieren.

Als User wird der Anwender gesehen, der real Aktionen innerhalb der Anwendung ausführt, wie z. B. Anmelden oder Bestellungen auslösen.

Ein Customer ist ein Geschäftspartner, für den die Geschäfte getätigt werden, vergleichbar mit einer Firma, für die Bestellungen ausgelöst werden. Er ist sozusagen der Träger der Geschäftsbeziehung.

Die Notwendigkeit der Trennung wird bei Betrachtung der drei Szenarien deutlich:

Shop-Szenario

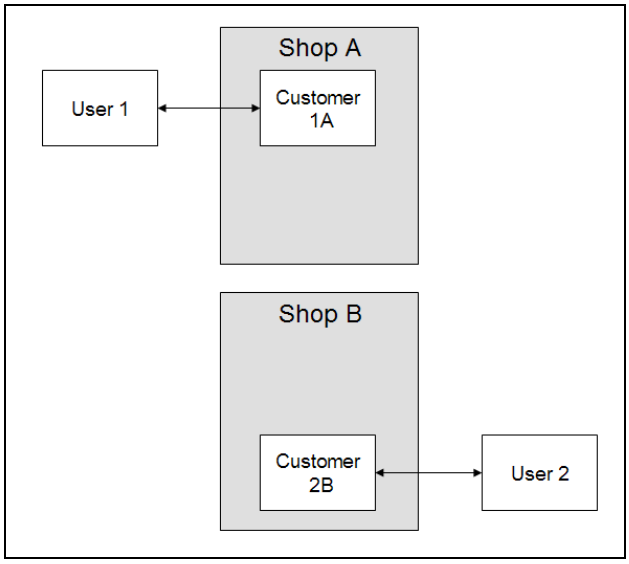

Abbildung 1: Shop-Szenario

Für dieses Szenario ist charakteristisch, dass zwischen User und Customer eine 1:1 –Beziehung besteht. Für den User, der über sein Login bestimmt ist, wird ein Customer angelegt. Für diesen einen Customer nimmt der User Bestellungen vor, bearbeitet Konto- und Lieferdaten usw.

Auf Grund der 1:1-Beziehung müsste für dieses Szenario keine Unterscheidung nach User und Customer erfolgen, d. h. einer Rolle könnten alle Daten allein zugeordnet werden.

Marktplatz-Szenario

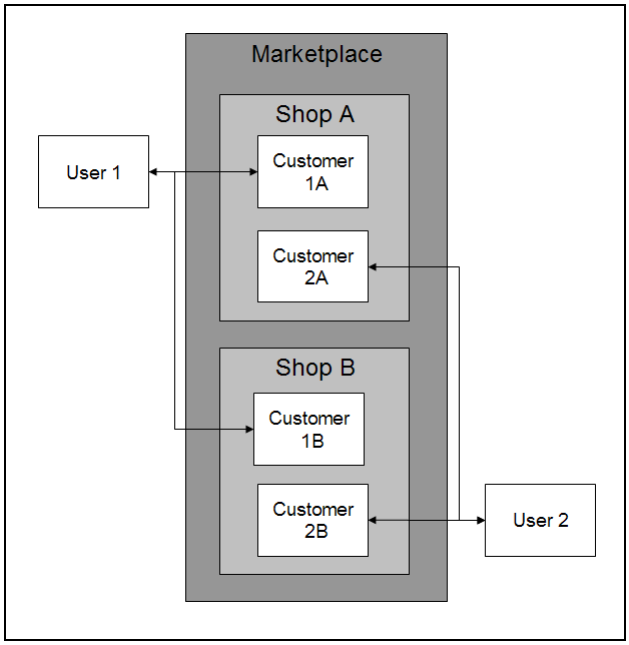

Abbildung 2: Marktplatz-Szenario

Das Marktplatz–Szenario ist dadurch gekennzeichnet, dass ein User mit einem Login Zugang zu mehreren Shops hat, d. h. dass ihm mehrere Customer zugeordnet werden. Es besteht also eine 1:n-Beziehung. Bei diesem Szenario besteht die Anforderung nach Aufteilung der Daten in User-Daten und Customer-Daten. Für jeden Customer werden die spezifischen Daten "seines" Shops gespeichert, wie Kundengruppe oder Staffelpreise. Für den User sind die Daten angelegt, die notwendig sind, um mit seinem Login auf alle Shops des Marktplatzes zuzugreifen.

Procurement-Szenario

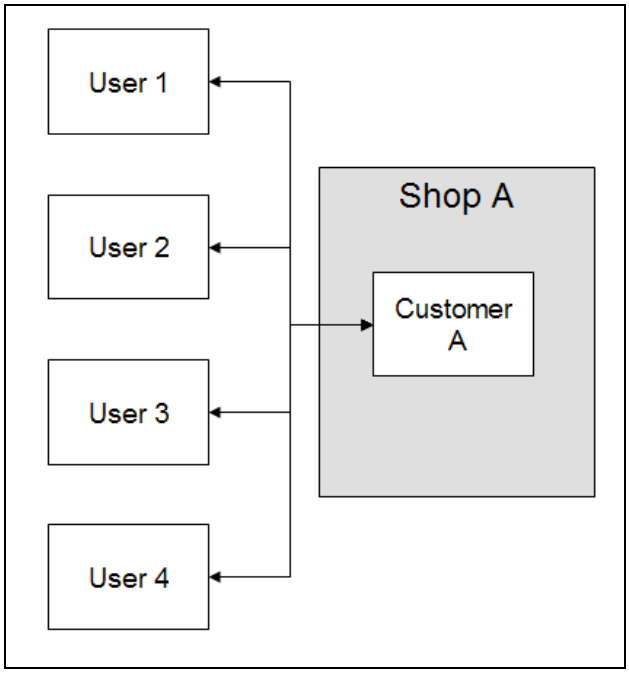

Abbildung 3: Procurement-Szenario

Im Procurement-Szenario tritt die Situation ein,dass ein Kunde (Firma) durch mehrere Nutzer (Mitarbeiter der Firma) vertreten wird. Die Nutzer haben alle ein eigenes Login. Im erweiterten Fall können Sie, gemäß

Berechtigungen, unterschiedliche Aktionen ausführen und Daten bearbeiten. In diesem Fall haben wir eine n:1 Beziehung. Auch hier ist es notwendig, zwischen den User-spezifischen Daten, die für jeden User gespeichert werden und den Shop-spezifischen Daten, die am Customer gespeichert werden, zu unterscheiden.

Daraus ergibt sich, dass für User und Customer jeweils ein eigener, spezieller Datenpool angelegt und bearbeitet werden muss. Wie die Daten zugeordnet sind, können Sie mit Hilfe der Diagnostics-Cartridge auslesen. Die Kenntnis dieser Zusammenhänge ist wichtig, wenn Sie z. B. Datenmodelle erweitern und entscheiden müssen, ob die Daten dem User oder dem Customer zugeordnet werden müssen.

Ein weiteres Beispiel ist die Anbindung externer Systeme, über die Kundendaten im System angelegt und bearbeitet werden. Auch hier ist es wichtig für den Entwickler zu wissen, welche der Daten zum User und welche zum Customer gehören.

6. URL-Aktionen

In ePages 5 sind zwei Typen von Aktionen definiert: *ViewAction* und *ChangeAction*. Mit Hilfe der ViewActions werden Objekte angezeigt, durch die ChangeActions werden die Objektdaten geändert.

Ein Request an die Applikation kann genau eine ViewAction auslösen und mehrere ChangeActions starten. Dabei werden die ChangeActions immer vor der ViewAction ausgeführt.

Die Modifikation und Anzeige eines Objekts wird entweder durch Abfrageparameter in der URL oder durch Form-Parameter eines POST-Requests definiert.

Ein typischer Request ist z. B.

```
http://vm41/epages/Store.admin/de_DE/?ViewAction=MBO-ViewGeneral&ObjectID=4639
```

Die Aktionen sind mit bestimmten Berechtigungen verknüpft. Der Nutzer muss über die entsprechende Berechtigung verfügen, diese Aktionen ausführen zu dürfen. Die Berechtigung muss für alle Aktionen vorliegen, die durch den Request ausgeführt werden, da sonst der Request nicht vollständig beantwortet werden kann. Zu Berechtigungen siehe auch *Rechte und Rollen*, *Seite 21*.

Aktionen werden in xml-Dateien im Unterverzeichnis */Database/XML* einer Cartridge angelegt. Sie werden per XML-Import in der Datenbank registriert.

Die Dateinamen müssen mit *Actions* oder *PageTypes* beginnen, z. B. *ActionsShop.xml* oder *PageTypesMBO.xml*. Die zur Aktion gehörenden Perl-Module werden im Verzeichnis */UI* einer Cartridge implementiert. Beispiele dafür finden Sie in den entsprechenden Verzeichnissen der Standardinstallation.

Lesen Sie dazu auch *Rechte und Rollen*, *Seite 21* und *PageType-Konzept*, *Seite 41*.

6.1 ViewActions

Die ViewAction legt fest, welche Sicht eines Objektes angezeigt wird. Im Parameter *ViewAction* des Requests wird angegeben, welche spezielle Sicht verwendet werden soll. Das entsprechende Objekt wird durch Angabe der ObjectID oder des Objektpfades spezifiziert.

Mit der ViewAction ist ein bestimmter PageType verknüpft, der für die geforderte Darstellung verantwortlich ist.

Optional kann die ViewAction einige Perl-Funktionen ausführen, die zusätzliche TLE-Variable für die Anzeige vorbereiten.

Die Standard-ViewAction *View* wird verwendet, wenn kein Parameter für die ViewAction angegeben ist. Ist kein Objekt spezifiziert, wird das System-Objekt angezeigt.

6.2 ChangeActions

ChangeActions verwenden Sie, um Objektdaten auszulesen, zu bearbeiten, Berechnungen durchzuführen, usw.

Die auszuführenden ChangeActions werden entweder als Form-Parameter oder als Parameter in der URL definiert.

Jede ChangeAction startet verschiedene Perl-Funktionen, welche das angegebene Objekt bzw. dessen Daten modifizieren. Das Objekt muss durch ObjectID oder den Objekt-Pfad spezifiziert sein.

Wollen Sie ein anderes Objekt als das angezeigte bearbeiten, muss dieses über den Parameter *ChangeObjectID* adressiert werden.

7. Templates

Die Anzeige eines Objekts im Browser wird über eine ViewAction ausgelöst. Wie dieses Objekt angezeigt wird, ist in einem Template definiert. Dieses Template ist über einen PageType mit der ViewAction verbunden. Siehe dazu auch *PageType-Konzept*, *Seite 41*.

Templates sind die Basis für die Darstellung im Browser. Es sind Dateien, welche die Information zur Formatierung und Darstellung von Inhalten und Daten enthalten. In den Templates werden XHTML, Javascript und ePages-spezifische Spracherweiterungen verwendet.

Diese Spracherweiterungen sind die *TLE* (Template Language Extension) und die *Language-Tags*.

Die *TLE-Variablen* sind Platzhalter für dynamische Informationen aus der Datenbank. Bei der Anfrage der Seite werden diese Platzhalter durch aktuelle Datenbankinhalte ersetzt.

TLE-Anweisungen sind ePages-spezifische Befehle, um die Webseiten inhaltsgesteuert anzuzeigen. Mit Hilfe dieser Anweisungen erfolgt die Abfrage und Auswertung von dynamischen Daten und das Generieren einer Webseite in Abhängigkeit des Ergebnisses einer TLE-Anweisung.

Language Tags sind Platzhalter für sprachspezifische Ausdrücke und werden zur Laufzeit je nach aktueller Sprache dynamisch durch den richtigen Inhalt ersetzt.

7.1 Technologie

Templates sind modular aufgebaut. Das Template für eine HTML-Seite setzt sich aus mehreren Templates zusammen, die jeweils bestimmte Bereiche der Seite beschreiben. Solche Teiltemplates können für verschiedene Seiten verwendet werden. Diese Wiederverwendbarkeit minimiert den Entwicklungs- und Pflegeaufwand. Siehe dazu auch *Darstellungsebene*, *Seite 43*.

Die Verwendung der TLE und Language-Tags bietet die Möglichkeit, jede Information aus der Datenbank abzufragen und in Echtzeit sprachabhängig anzuzeigen.

Dadurch können, ausgehend von einem Template, aufgrund der variablen Inhalte unterschiedliche HTML-Seiten generiert und angezeigt werden. Auch dies bedeutet eine signifikante Verringerung des Aufwandes für Webdesigner.

Für die Erstellung der Webseiten sind HTML/XHTML- und XML-Kenntnisse Vorraussetzung. Erfahrungen mit Javascript sind nicht notwendig, aber von Vorteil. Die Grundlagen für die Arbeit mit TLE-Sprachelementen und mehrsprachigen Seiten lesen Sie in den Kapiteln *TLE*, *Seite 63* und *Mehrsprachigkeit – Language-Tags* , *Seite 51*.

7.2 Template-Prozess

Bei einem Request müssen je nach Datenbankinhalt und angeforderter Sprache unterschiedliche Webseiten als Antwort ausgeliefert werden. Auf Grund der Komplexität ist es unmöglich, alle voraussichtlich notwendigen Webseiten statisch im Voraus zu produzieren. Die Seiten müssen mit den aktuellen Daten zur Laufzeit dynamisch erzeugt werden.

Ein Nachteil der dynamischen Erzeugung sind die größeren Antwortzeiten im Vergleich zur Anzeige statischer Seiten. Man muss versuchen, eine Balance zwischen Performance und Aktualität finden.

Aus diesem Grund gibt es im Template-Prozess mehrere Stellen, an denen der Entwickler oder Anwender entscheiden kann, ob Seiten aktuell erzeugt werden sollen oder ganz oder teilweise aus bereits vorhandenen HTML-Dateien zusammengesetzt werden.

Daher wird mit einem Request folgender Prozess gestartet, siehe *Abbildung 4*:

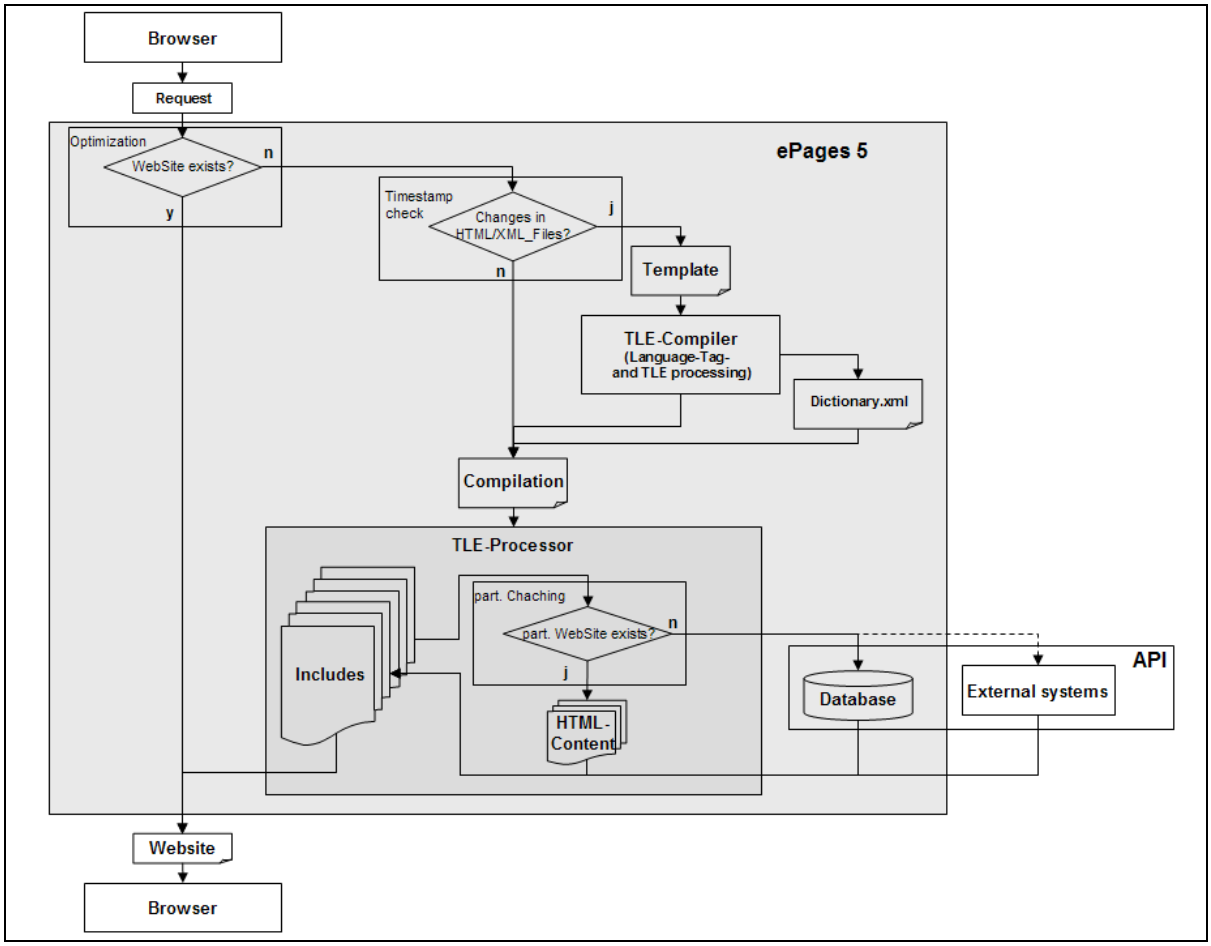

Abbildung 4: Template-Prozess, vereinfachte Darstellung

Nach Eingang des Requests besteht die Möglichkeit zu prüfen, ob die angeforderte Webseite bereits als statische HTML-Seite zur Verfügung steht. Diese Prüfung kann durch den Händler in dessen Administration aktiviert werden, siehe Abschnitt *Optimierung* im *Handbuch für Händler*.

Dies ist die schnellste Möglichkeit, den Request zu beantworten, es werden aber keine Daten aktualisiert.

Anderenfalls beginnt der eigentliche Template-Prozess.

Alle bereits einmal angeforderten Templates werden auf der Festplatte als vorkompilierte Version (Kompilat) mit einem Zeitstempel abgelegt. Dies stellt in gewisser Weise einen Cache dar und verkürzt bei Verwendung die Zugriffszeiten entscheidend.

Beim Eintreffen eines Requests prüft das System zuerst, ob das angeforderte Template kompiliert vorliegt. Ist es vorhanden, wird der Zeitstempel der Cache-Version mit denen des Original-HTML-Templates und den XML-Sprachdateien verglichen. Sind die Zeitstempel der HTML-Datei und der XML-Dateien nicht jünger als die des Kompilats, verwendet das System die vorkompilierte Version und übergibt diese an den TLE-Prozessor.

Ist der Zeitstempel der vorkompilierten Seite älter als der des Templates oder der Sprachdatei, dann ist das Kompilat veraltet und wird verworfen. Daraufhin wird der Übersetzungsprozess eines Templates neu gestartet. Gleiches passiert, wenn für dieses Template noch kein Kompilat vorliegt, es also noch nie übersetzt wurde.

Die Prüfung des Zeitstempels kann aus Gründen der Performance-Steigerung ausgesetzt werden, siehe dazu *Page Caching*, *Seite 143*.

Bei der Erzeugung des Kompilats wird das Template in einem ersten Schritt nach den Language-Tags durchsucht und die sprachspezifischen Inhalte werden eingefügt, siehe dazu Kapitel *Mehrsprachigkeit – Language-Tags* , *Seite 51*. Danach wird das Template weiter verarbeitet und für die Verarbeitung durch den TLE-Compiler als vorkompilierte Datei gespeichert. Diese Datei wird im Dateisystem abgelegt und überschreibt die eventuell vorhandene "veraltete" Version.

Die folgenden Abbildungen zeigen ein Beispiel für die einzelnen Abschnitte dieses Prozesses:

In *Abbildung 5* sehen Sie den Code-Ausschnitt eines Templates. Neben den HTML-Formatierungen fallen die TLE (eingeleitet durch #) und Language-Tags (eingeschlossen in {}) auf.

```
#IF(#Shop.FeatureMaxValue.EnhancedCustomerAccount)
#IF((#Class.Alias NE "CustomerOrder" OR NOT #Session.User.IsAnonymous) AND #INPUT.ViewAction NE "ViewRegistration")
<div class="ContextBox">
  <div class="LoginBox">
  #IF(#Session.User AND NOT #Session.User.IsAnonymous)
    <div class="ContextBoxHead">
      <h1>#Session.User.Name</h1>
    </div>
    <div class="ContextBoxBody">
      <a class="Action" href="?ObjectID=#Session.User.ID&amp;ViewAction=ViewMyAccount">{MyAccount}</a>
      <br />
      <a class="Action" href="?ObjectPath=#Shop.Path[url]&amp;ViewAction=View&amp;ChangeAction=Logout">{Logout}</a>
    </div>
  #ELSE
    <div class="ContextBoxHead">
      <h1>{CustomerLogin}</h1>
    </div>
    <form action="#FUNCTION("BASEURL", #System, 1)#IF(#Pager)#Pager.URLPage#ELSE?ObjectPath=#Path[url]#IF(#INPUT.ViewAction NE "View" AND
#INPUT.ViewAction NE "ViewLostPassWD")&amp;ViewAction=#INPUT.ViewAction#ENDIF#ENDIF#IF(#INPUT.ErrorAction)&amp;ErrorAction=#INPUT.ErrorAction#ENDIF"
method="post">

    #IF(#FormError AND #FormErrors.Form.Login.ErrorCount)
      <div class="ContextBoxBody">
        <div class="DialogError">
      #IF(#FormErrors.Reason.LOGIN_NOT_FOUND)
        {LoginNotFound}#ELSIF(#FormErrors.Reason.LOGIN_INACTIVE)
        {LoginInActive}#ELSIF(#FormErrors.Reason.PASSWORD_MISMATCH)
        {PasswordMismatch}#ELSE
        {EnterLoginAndPassword}#ENDIF
        </div>
      </div>
    #ENDIF
      <div class="ContextBoxBody">
        <input type="hidden" name="ChangeAction" value="SaveLoginForm" />
        <input type="hidden" name="RegistrationObjectID" value="#Shop.ID" />
        <div class="Entry #IF(#FormError_Login AND #FormErrors.Form.Login.ErrorCount)DialogError#ENDIF">
          <div class="InputLabelling">{UserName}</div>
          <div class="InputField">#WITH_ERROR(#FormError)
            <input class="Login" name="Login" value="#IF(#Login)#Login#ENDIF" />#ENDWITH_ERROR
          </div>
        </div>
        <div class="Entry #IF(#FormError_Password AND #FormErrors.Form.Login.ErrorCount)DialogError#ENDIF">
          <div class="InputLabelling">{Password}</div>
          <div class="InputField">
            <input class="Login" name="Password" type="password" value="" />
          </div>
        </div>
      </div>
      <div class="ContextBoxBody">
        <input class="Action" type="submit" value="{Login}" /><br />
      </div>
      <div class="ContextBoxBody">
#BLOCK("MENU","LoginBoxLinks")
  #INCLUDE(#Template)
#ENDBLOCK
      </div>

    </form>
  #ENDIF
  </div>
</div>
#ENDIF
#ENDIF
```

Abbildung 5: Code-Beispiel für Template

Daraus entsteht das Kompilat, von dem ein Ausschnitt in *Abbildung 6* zu sehen ist. Die sprachspezifischen Daten sind eingesetzt und die TLE-Variablen werden in Perl-Funktionsaufrufe umgesetzt.

```
#line 9 "C:\epages\Cartridges/DE_EPAGES/Design/Templates/SF/NavElements/SF.LoginBox.html"
;push @Result, $Processor->replaceTLE('Session.User.ID','', '#Session.User.ID')
#line 9 "C:\epages\Cartridges/DE_EPAGES/Design/Templates/SF/NavElements/SF.LoginBox.html"
;push @Result, "&amp;ViewAction=ViewMyAccount\">Mein Konto</a>\n      <br />\n      <a class=\"Action\" href=\"?ObjectPath="
#line 11 "C:\epages\Cartridges/DE_EPAGES/Design/Templates/SF/NavElements/SF.LoginBox.html"
;push @Result, $Processor->replaceTLE('Shop.Path','url', '#Shop.Path[url]')
#line 11 "C:\epages\Cartridges/DE_EPAGES/Design/Templates/SF/NavElements/SF.LoginBox.html"
;push @Result, "&amp;ViewAction=View&amp;ChangeAction=Logout\">Abmelden</a>\n    </div>\n  "
#line 13 "C:\epages\Cartridges/DE_EPAGES/Design/Templates/SF/NavElements/SF.LoginBox.html"
;}else {push @Result, "\n    <div class=\"ContextBoxHead\">\n      <h1>Kunden-Login</h1>\n    </div>\n    <form action=\""
#line 17 "C:\epages\Cartridges/DE_EPAGES/Design/Templates/SF/NavElements/SF.LoginBox.html"
;push @Result, $Processor->callBlock('BASEURL', sub { my @Result = ();push @Result, $Processor->tle('System','')
#line 17 "C:\epages\Cartridges/DE_EPAGES/Design/Templates/SF/NavElements/SF.LoginBox.html"
;push @Result, 1
#line 17 "C:\epages\Cartridges/DE_EPAGES/Design/Templates/SF/NavElements/SF.LoginBox.html"
; return \@Result;}, undef)
#line 17 "C:\epages\Cartridges/DE_EPAGES/Design/Templates/SF/NavElements/SF.LoginBox.html"
;if($Processor->tle('Pager',''))
#line 17 "C:\epages\Cartridges/DE_EPAGES/Design/Templates/SF/NavElements/SF.LoginBox.html"
) {push @Result, $Processor->replaceTLE('Pager.URLPage','', '#Pager.URLPage')
#line 17 "C:\epages\Cartridges/DE_EPAGES/Design/Templates/SF/NavElements/SF.LoginBox.html"
;}else {push @Result, "?ObjectPath="
#line 17 "C:\epages\Cartridges/DE_EPAGES/Design/Templates/SF/NavElements/SF.LoginBox.html"
;push @Result, $Processor->replaceTLE('Path','url', '#Path[url]')
#line 17 "C:\epages\Cartridges/DE_EPAGES/Design/Templates/SF/NavElements/SF.LoginBox.html"
;if($Processor->tle('INPUT.ViewAction',''))
#line 17 "C:\epages\Cartridges/DE_EPAGES/Design/Templates/SF/NavElements/SF.LoginBox.html"
 ne "View"
#line 17 "C:\epages\Cartridges/DE_EPAGES/Design/Templates/SF/NavElements/SF.LoginBox.html"

#line 17 "C:\epages\Cartridges/DE_EPAGES/Design/Templates/SF/NavElements/SF.LoginBox.html"
 && $Processor->tle('INPUT.ViewAction','')
#line 17 "C:\epages\Cartridges/DE_EPAGES/Design/Templates/SF/NavElements/SF.LoginBox.html"
 ne "ViewLostPassWD"
#line 17 "C:\epages\Cartridges/DE_EPAGES/Design/Templates/SF/NavElements/SF.LoginBox.html"

#line 17 "C:\epages\Cartridges/DE_EPAGES/Design/Templates/SF/NavElements/SF.LoginBox.html"

#line 17 "C:\epages\Cartridges/DE_EPAGES/Design/Templates/SF/NavElements/SF.LoginBox.html"
) {push @Result, "&amp;ViewAction="
#line 17 "C:\epages\Cartridges/DE_EPAGES/Design/Templates/SF/NavElements/SF.LoginBox.html"
;push @Result, $Processor->replaceTLE('INPUT.ViewAction','', '#INPUT.ViewAction')
#line 17 "C:\epages\Cartridges/DE_EPAGES/Design/Templates/SF/NavElements/SF.LoginBox.html"
;}}if($Processor->tle('INPUT.ErrorAction',''))
#line 17 "C:\epages\Cartridges/DE_EPAGES/Design/Templates/SF/NavElements/SF.LoginBox.html"
) {push @Result, "&amp;ErrorAction="
#line 17 "C:\epages\Cartridges/DE_EPAGES/Design/Templates/SF/NavElements/SF.LoginBox.html"
;push @Result, $Processor->replaceTLE('INPUT.ErrorAction','', '#INPUT.ErrorAction')
#line 17 "C:\epages\Cartridges/DE_EPAGES/Design/Templates/SF/NavElements/SF.LoginBox.html"
;}push @Result, "\" method=\"post\">\n  "
#line 18 "C:\epages\Cartridges/DE_EPAGES/Design/Templates/SF/NavElements/SF.LoginBox.html"
;if($Processor->tle('FormError',''))
#line 18 "C:\epages\Cartridges/DE_EPAGES/Design/Templates/SF/NavElements/SF.LoginBox.html"
 && $Processor->tle('FormErrors.Form.Login.ErrorCount',''))
#line 18 "C:\epages\Cartridges/DE_EPAGES/Design/Templates/SF/NavElements/SF.LoginBox.html"

#line 18 "C:\epages\Cartridges/DE_EPAGES/Design/Templates/SF/NavElements/SF.LoginBox.html"
) {push @Result, "\n      <div class=\"ContextBoxBody\">\n        <div class=\"DialogError\">\n      "
#line 21 "C:\epages\Cartridges/DE_EPAGES/Design/Templates/SF/NavElements/SF.LoginBox.html"
;if($Processor->tle('FormErrors.Reason.LOGIN_NOT_FOUND',''))
#line 21 "C:\epages\Cartridges/DE_EPAGES/Design/Templates/SF/NavElements/SF.LoginBox.html"
) {push @Result, "\n        Ein Benutzer mit den eingegebenen Benutzernamen ist im Shop nicht vorhanden."
#line 22 "C:\epages\Cartridges/DE_EPAGES/Design/Templates/SF/NavElements/SF.LoginBox.html"
;}elsif($Processor->tle('FormErrors.Reason.LOGIN_INACTIVE',''))
#line 22 "C:\epages\Cartridges/DE_EPAGES/Design/Templates/SF/NavElements/SF.LoginBox.html"
) {push @Result, "\n        {LoginInActive}"
#line 23 "C:\epages\Cartridges/DE_EPAGES/Design/Templates/SF/NavElements/SF.LoginBox.html"
;}elsif($Processor->tle('FormErrors.Reason.PASSWORD_MISMATCH',''))
#line 23 "C:\epages\Cartridges/DE_EPAGES/Design/Templates/SF/NavElements/SF.LoginBox.html"
) {push @Result, "\n        Das Kennwort für den eingegebenen Benutzernamen ist falsch."
#line 24 "C:\epages\Cartridges/DE_EPAGES/Design/Templates/SF/NavElements/SF.LoginBox.html"
```

Abbildung 6: Kompilat des Templates in *Abbildung 5*

Im weiteren Ablauf wird das Kompilat durch den TLE-Prozessor abgearbeitet, der dabei die Schaltzentrale zwischen dem Kompilat und dem Applikationsserver bildet.

An dieser Stelle gibt es eine weitere Möglichkeit, die Antwortzeiten für den Request zu verkürzen. Für definierte Abschnitte in Templates kann ein partielles Caching aktiviert werden, siehe dazu *Partielles Caching*, *Seite 150*. Damit entscheiden Sie, ob für die einzelnen Abschnitte vorhandener, bei vorigen Durchläufen erzeugter HTML-Code verwendet wird, oder ob eine neue Version mit aktuellen Daten generiert wird.

Ohne partielles Caching wird das Kompilat durch den TLE-Prozessor abgearbeitet, die Datenbankabfragen und Aktionen werden ausgeführt, alle notwendigen Daten ersetzt und die Seite HTML-codiert aufbereitet.

Nachdem alle Includes verarbeitet wurden, wird die resultierende HTML-Seite als Antwort auf den Request an den Browser ausgeliefert.

Die Kompilate sind Text-Dateien mit der Endung *.ctmpl*, welche Perl-Code enthalten. Sie werden in

```
%EPAGES_STATIC%/Store/Templates/DE_EPAGES
```

im einem Unterverzeichnis mit dem Namen der jeweiligen Original-Cartridge gespeichert. Dabei ist *Store* die jeweils aktuelle Datenbank.

> ***Achtung:*** Editieren Sie **nicht** in den Text-Dateien der Kompilate (ctmpl-Dateien). Damit zerstören Sie die Integrität zwischen Template und Kompilat. Arbeiten Sie Ihre Änderungen in das Template ein und lassen Sie das Kompilat automatisch erzeugen!

7.3 Prinzipielle Struktur der Webseite

ePages stellt eine große Anzahl an vordefinierten, voll funktionsfähigen Templates für Ihren Shop zur Verfügung. Sie können diese Templates als Basis für neue Templates benutzen oder direkt Ihren Bedürfnissen anpassen.

Alle aus diesen Templates generierten HTML-Seiten weisen eine ähnliche Grundstruktur auf und enthalten für die Storefront-Seitendarstellung im *Body* die Bereiche in *Abbildung 7*.

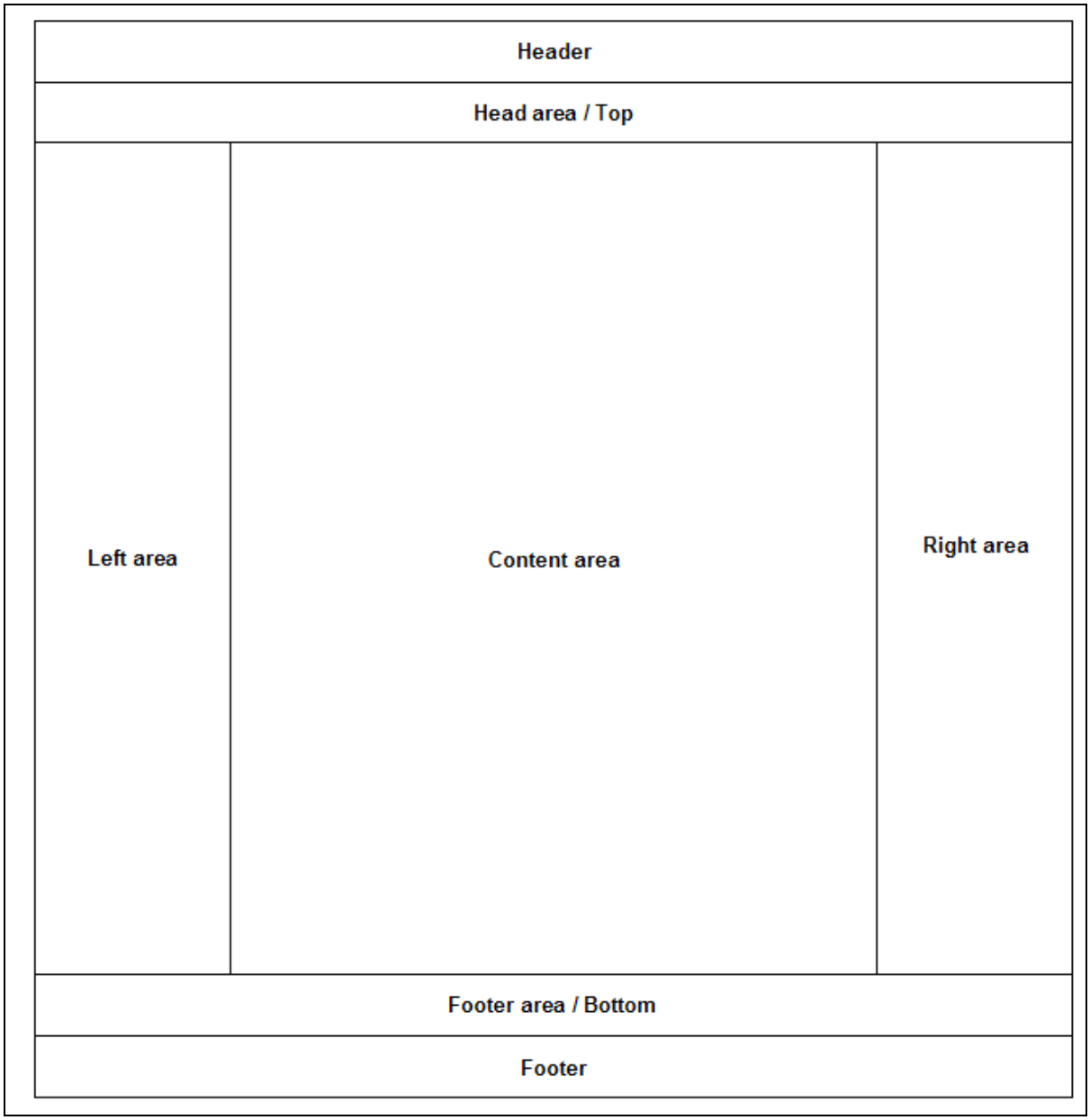

Abbildung 7: Seitenstruktur im Body der HTML-Seiten

Während die "Randbereiche" hauptsächlich Navigations- und Funktionselemente enthalten, zeigt der Arbeitsbereich die Ergebnisse der verschiedenen Funktionen.

Die Struktur wird durch die so genannten PageTypes bereitgestellt, die zur Laufzeit die Zusammensetzung der HTML-Seiten aus den einzelnen Include-Templates steuern.

7.4 Überladung von Templates

Oft besteht der Bedarf oder die Notwendigkeit, Templates zu ändern, um den eigenen Erfordernissen anzupassen. Dabei können Änderungen an Originaltemplates zum einen dazu führen, dass Fehler auftreten, die das gesamte System beeinträchtigen. Zum anderen setzen Sie die Upgradefähigkeit Ihrer ePages-Applikation außer Kraft. Im Falle eines Upgrades werden alle Templates in den Standard-Cartridges überschrieben, so dass damit Ihre individuellen Änderungen verloren gehen.

Achtung: Wollen Sie die Updatefähigkeit Ihres Systems wahren, nehmen Sie kein Änderungen an den Originaldateien in den Originalverzeichnissen vor!

Daher gibt es in ePages einen Mechanismus, der Ihnen hilft, Ihre Änderungen vorzunehmen, ohne Original-Templates zu bearbeiten. Das ist die Überladung von Templates. Der Grundgedanke dabei ist die Bereitstellung unterschiedlicher Templates gleichen Namens und Definition der Abarbeitungsreihenfolge.

Im ePages-System gibt es spezielle Überladungsverzeichnisse. In diese können Sie die zu ändernden Originaldateien kopieren und bearbeiten. Die Originaldateien verbleiben unverändert in ihren Originalverzeichnissen. Die Bearbeitungsreihenfolge ist systemseitig so definiert, dass das System benötigte Dateien zuerst im "Überladungs-Verzeichnis" sucht und verwendet, im Falle nicht vorhandener Dateien auf die Originalverzeichnisse zugreift.

Das Verzeichnis, in dem Sie Ihre geänderten Templates ablegen können finden Sie unter

```
%EPAGES_STORES%/Store/Templates/DE_EPAGES
```

Hier finden Sie in Analogie zum Installationsverzeichnis für jede Standard-Cartridge ein Verzeichnis und darin wiederum entsprechende Template-Verzeichnisse für Storefront und Backoffice.

Wenn Sie ein Standardtemplate ändern wollen, kopieren Sie es aus dem Originalverzeichnis in das Verzeichnis gleichen Namens an o. g. Stelle. Dann können Sie alle Änderungen vornehmen, ohne befürchten zu müssen, das System zu beeinflussen. Ist Ihr geändertes Template noch fehlerhaft oder noch nicht fertig, benennen Sie es einfach um oder löschen es aus dem Verzeichnis, um den Ausgangszustand wieder herzustellen.

Dieses Vorgehen bietet sich an, wenn Sie eine Änderung auf Ihrer lokalen Installation vornehmen und diese auch nicht an andere weitergeben wollen.

Sollen die Veränderungen Bestandteil eines auslieferbaren Paketes sein, sollten Sie diese Änderungen in eine eigene Cartridge fassen. Auch hier können Sie die Überlagerungstechnik nutzen, wenn Sie Ihre geänderten Dateien in den richtigen Verzeichnissen Ihrer Cartridge ablegen. Lesen Sie dazu im Kapitel *Cartridges*, *Seite 85* die Erläuterungen zum Unterverzeichnis *Data/Private*.

Beispiel für die Anwendung der Überladung von Templates finden Sie in *Anhang C: Anwendungsbeispiele (AWB)*, *Seite 183*.

Es können sowohl Storefront-Templates als auch Backoffice-Templates überschrieben werden.

7.5 Template-Debugging

Bevor Sie die Templates gezielt ändern können, müssen Sie wissen, welches Template welchen Teil der Webseite darstellt, d. h. auf welches Template Sie zugreifen müssen. Äußert hilfreich ist hier das Aktivieren der Debugging-Funktion, welche die Namen der verwendeten Templates im HTML-Quelltext der angezeigten Seite mit anzeigt. Vergleichen Sie *Abbildung 8* mit *Abbildung 9*.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="de" xml:lang="de">
<head>
    <title>Milestones - Gut gerüstet für Ihre Ziele</title>

      <script type="text/javascript" src="/WebRoot/Store/epages_scripts.js"></script>

    <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
    <link href="/WebRoot/Store/Shops/DemoShop/Styles/MotorBikes_002F_Red/StorefrontStyle.css" rel="stylesheet" type="text/css" />
    <link href="/WebRoot/Store/SF/Styles/MyStyle.css" rel="stylesheet" type="text/css" />
    <link href="/WebRoot/Store/SF/Styles/MotorBikes/SubStyles/Red/StyleExtension.css" rel="stylesheet" type="text/css" />

</head>

<body>


  <div class="Layout1 GeneralLayout">


    <div class="Header">
            <div class="PropertyContainer">
        <table class="SizeContainer"><tr>
```

Abbildung 8: HTML-Quelltext einer angezeigten Seite ohne Debug-Informationen

Bei aktivierten Debug-Informationen fallen die Kommentarzeilen mit den Template-Informationen sofort auf:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="de" xml:lang="de">

<!-- BEGIN INCLUDE C:\epages5\Cartridges/DE_EPAGES/Presentation/Templates/BasePageType.Head.html 0.078 seconds -->
<head>

<!-- BEGIN INCLUDE C:\epages5\Cartridges/DE_EPAGES/Design/Templates/SF/SF.Title.html 0.016 seconds -->
    <title>Milestones - Gut gerüstet für Ihre Ziele</title>
<!-- END INCLUDE C:\epages5\Cartridges/DE_EPAGES/Design/Templates/SF/SF.Title.html -->


<!-- BEGIN INCLUDE C:\epages5\Cartridges/DE_EPAGES/Presentation/Templates/BasePageType.Script.html 0.016 seconds -->


<!-- BEGIN INCLUDE C:\epages5\Cartridges/DE_EPAGES/Presentation/Templates/BasePageType.Script-Base.html 0.000 seconds -->
    <script type="text/javascript" src="/WebRoot/Store/epages_scripts.js"></script>
<!-- END INCLUDE C:\epages5\Cartridges/DE_EPAGES/Presentation/Templates/BasePageType.Script-Base.html -->


<!-- END INCLUDE C:\epages5\Cartridges/DE_EPAGES/Presentation/Templates/BasePageType.Script.html -->


<!-- BEGIN INCLUDE C:\epages5\Cartridges/DE_EPAGES/Design/Templates/SF/SF.Head-ContentType.html 0.016 seconds -->
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<!-- END INCLUDE C:\epages5\Cartridges/DE_EPAGES/Design/Templates/SF/SF.Head-ContentType.html -->


<!-- BEGIN INCLUDE C:\epages5\Shared\Stores/Store/Templates/DE_EPAGES/Design/Templates/SF/SF.Style.html 0.016 seconds -->
    <link href="/WebRoot/Store/Shops/DemoShop/Styles/MotorBikes_002F_Red/StorefrontStyle.css" rel="stylesheet" type="text/css" />
    <link href="/WebRoot/Store/SF/Styles/MyStyle.css" rel="stylesheet" type="text/css" />
    <link href="/WebRoot/Store/SF/Styles/MotorBikes/SubStyles/Red/StyleExtension.css" rel="stylesheet" type="text/css" />

<!-- END INCLUDE C:\epages5\Shared\Stores/Store/Templates/DE_EPAGES/Design/Templates/SF/SF.Style.html -->

</head>

<!-- END INCLUDE C:\epages5\Cartridges/DE_EPAGES/Presentation/Templates/BasePageType.Head.html -->


<!-- BEGIN INCLUDE C:\epages5\Cartridges/DE_EPAGES/Design/Templates/SF/SF.Body.html 0.719 seconds -->
<body>

<!-- BEGIN INCLUDE C:\epages5\Cartridges/DE_EPAGES/Etracker/Templates/SF/SF.INC-Etracker.html 0.016 seconds -->



<!-- END INCLUDE C:\epages5\Cartridges/DE_EPAGES/Etracker/Templates/SF/SF.INC-Etracker.html -->

<!-- BEGIN INCLUDE C:\epages5\Cartridges/DE_EPAGES/Design/Templates/SF/SF.Layout.html 0.703 seconds -->

<!-- BEGIN INCLUDE C:\epages5\Cartridges/DE_EPAGES/Design/Templates/SF/SF.Layout1.html 0.703 seconds -->
  <div class="Layout1 GeneralLayout">
```

Abbildung 9 HTML-Quelltext einer angezeigten Seite mit Debug-Informationen

Sie können genau erkennen, wo welches Template eingesetzt wird und wo Sie es finden können. Somit ist es für Sie einfacher, ausgehend von dem Bereich der Webseite, den Sie ändern möchten, auf das zu bearbeitende Template zu schließen.

Als weiteren wichtigen Parameter können Sie die Verarbeitungszeit des jeweiligen INCLUDES ablesen. Nutzen Sie diese Daten, um die Performance Ihrer Anwendung zu optimieren.

Um das Debugging zu aktivieren, öffnen Sie die Datei

```
%EPAGES_CONFIG%/log4perl.conf
```

und suchen in der Sektion *tle* folgenden Eintrag:

```
log4perl.category.DE_EPAGES::Presentation::API::Template::INCLUDE=DEBUG
```

siehe *Abbildung 10*.

```
;
; tle
;
;log4perl.category.DE_EPAGES.TLE.API.Execute = DEBUG
;log4perl.category.DE_EPAGES.TLE.API.Lexer   = DEBUG
;log4perl.category.DE_EPAGES.TLE.API.LoopHandler = DEBUG
;log4perl.category.DE_EPAGES.TLE.API.Processor = DEBUG
;log4perl.category.DE_EPAGES.TLE.API.Processor.tle = DEBUG
;log4perl.category.DE_EPAGES.TLE.API.Processor.getTLE=DEBUG
;log4perl.category.DE_EPAGES.TLE.API.XPathHandler = DEBUG
;log4perl.category.DE_EPAGES::Dictionary::API::Template=DEBUG
;log4perl.category.DE_EPAGES::Object::API::Object::Object::getTLE=DEBUG
 log4perl.category.DE_EPAGES::Presentation::API::Template::INCLUDE=DEBUG
;log4perl.category.DE_EPAGES::Presentation::UI::PageTypeServlet::processContent=DEBUG

;
; webinterface and servlets
```

Abbildung 10: Eintrag zum Aktivieren des Debugging

Nach der Installation ist diese Zeile auskommentiert und das Debugging damit deaktiviert. Entfernen Sie das Semikolon am Zeilenanfang und speichern die Datei. Danach sind die Debug-Informationen im HTML-Quelltext zu sehen.

Mehr zur *log4perl.conf* lesen Sie im *Installationshandbuch für Windows*.

8. PageType-Konzept

Ein PageType ist eine xml-Datei, in der die Darstellung eines Objektes strukturell definiert wird. Damit ist ein PageType das Bindeglied zwischen der ViewAction für ein Objekt und der dazugehörigen Darstellung im Browser durch Templates. In PageTypes wird definiert, für welches Objekt bei welcher Anzeigeaktion welches Template verarbeitet wird. Diese Referenz wird über die PageTypes hergestellt und ist in der Datenbank gespeichert.

Neben der Bereichsdefinition wird die entsprechende ViewAction festgelegt. Es kann eine spezielle ViewAction benannt werden. Wird in der xml-Datei keine ViewAction explizit angegeben, wird standardmäßig die ViewAction *View* verwendet.

PageTypes sind die Basis, um Templates modular aufzubauen. Durch sie ist es möglich, Design und Funktionalität so granular zu entwickeln, dass die unterschiedlichen Funktionalitäten der Seiten aus einzelnen Modulen mit hoher Flexibilität und Wiederverwendbarkeit zusammengesetzt werden können.

Auf Grund dieser komplexen Zusammenhänge sollten Sie nach den prinzipiellen Erläuterungen unbedingt die weiter unten angeführten Anwendungsbeispiele durcharbeiten, um ein Verständnis für die Arbeit mit PageTypes zu erlangen.

8.1 Logische Struktur

Im PageType wird die logische Struktur einer Webseite definiert. Dafür werden einzelne Bereiche festgelegt, aus denen die Seite zusammengesetzt wird. Diesen logischen Bereichen werden HTML-Dateien zugeordnet. Diese enthalten den Quelltext zur Beschreibung der einzelnen Teilbereiche, d. h. für jeden Bereich wird eine konkrete HTML-Datei angegeben. Siehe dazu auch *Templates*, *Seite 33* und *Darstellungsebene*, *Seite 43*.

Ein einfaches Beispiel für die XML-Definition eines PageTypes sehen Sie in *Codebeispiel 11*.

```
<?xml version="1.0" encoding="UTF-8"?>
<epages>
  <Cartridge reference="1" Package="DE_EPAGES::Presentation">
    <Class reference="1" Path="/Classes/Object">
      <PageType Alias="Mail" delete="1">
        <Template Name="Page"     FileName="Mail.Page.html" />
        <Template Name="Head"     FileName="Mail.Head.html" />
        <Template Name="Body"     FileName="Mail.Body.html" />
        <Template Name="Content"  FileName="Mail.Content.html" />
      </PageType>
    </Class>
  </Cartridge>
</epages>
```

Codebeispiel 11: XML-Definition eines PageTypes

Mit *Cartridge reference...* geben Sie an, in welcher Cartridge die Template-Dateien liegen.

Die Zuordnung PageType – Objektklasse treffen Sie mit *Class reference ....* Der PageType im Beispiel ist der Objektklasse *Object* zugewiesen. Das bedeutet, dass alle Objekte diesen PageType nutzen können.

Unter *Alias* wird der Name des PageTypes vergeben, *delete="1"* bedeutet, dass der PageType bei der Deinstallation der Cartridge wieder mit aus der Datenbank gelöscht wird.

Template Name ... definiert den Bereich der Webseite (logischer Bereich), in dem die Inhalte angezeigt werden; mit *Filename...* wird festgelegt, welche HTML-Datei für die Darstellung des Bereiches verwendet wird.

PageTypes sind hierarchisch aufgebaut. Die Basis ist der *BasePageType*. Dieser liefert die allgemeine Aufteilung der Seite mit dem Verweis auf die entsprechenden Basistemplates, siehe *Codebeispiel 12*.

```
<?xml version="1.0" encoding="UTF-8"?>
<epages>
  <!-- page types and templates -->
  <Cartridge reference="1" Package="DE_EPAGES::Presentation">
    <Class reference="1" Path="/Classes/Object">
      <PageType Alias="BasePageType" LoginViewAction="ViewLoginForm" delete="1">
        <Menu Template="Head" Position="0">
          <Menu Template="Head-Title"       Position="10" />
          <Menu Template="Head-Script"      Position="20" >
            <Menu Template="Script-Base"      Position="10" />
          </Menu>
        </Menu>
        <Template Name="Page"  FileName="BasePageType.Page.html" />
        <Template Name="Head"  FileName="BasePageType.Head.html" />
        <Template Name="Body"  FileName="BasePageType.Body.html" />
        <Template Name="Pager" FileName="BasePageType.Pager.html" />

        <Template Name="Head-Title"  FileName="BasePageType.Title.html" />
        <Template Name="Head-Script" FileName="BasePageType.Script.html" />
        <Template Name="Script-Base" FileName="BasePageType.Script-Base.html" />
        <Template Name="Script-Event" FileName="BasePageType.Script-Event.html"
        />
        <Template Name="Script-XMLHttpRequest"  FileName="BasePageType.Script-
          XMLHttpRequest.html" />
        <Template Name="Script-Slideshow"  FileName="BasePageType.Script-
          Slideshow.html" />
      </PageType>
    </Class>
  </Cartridge>
</epages>
```

Codebeispiel 12: BasePageType-Definition

In diesem *BasePageType* sind die grundsätzlichen, auf allen Seiten verwendeten Bereiche festgelegt. Diese werden durch die weiteren PageTypes je nach Aufgabe immer spezieller definiert und in weitere Bereiche aufgespaltet. Damit ist der *BasePageType* der Ursprung für die Vererbung innerhalb der PageTypes.

In *Abbildung 11* sehen Sie im Ansatz die PageType-Hierarchie und den Vererbungsverlauf.

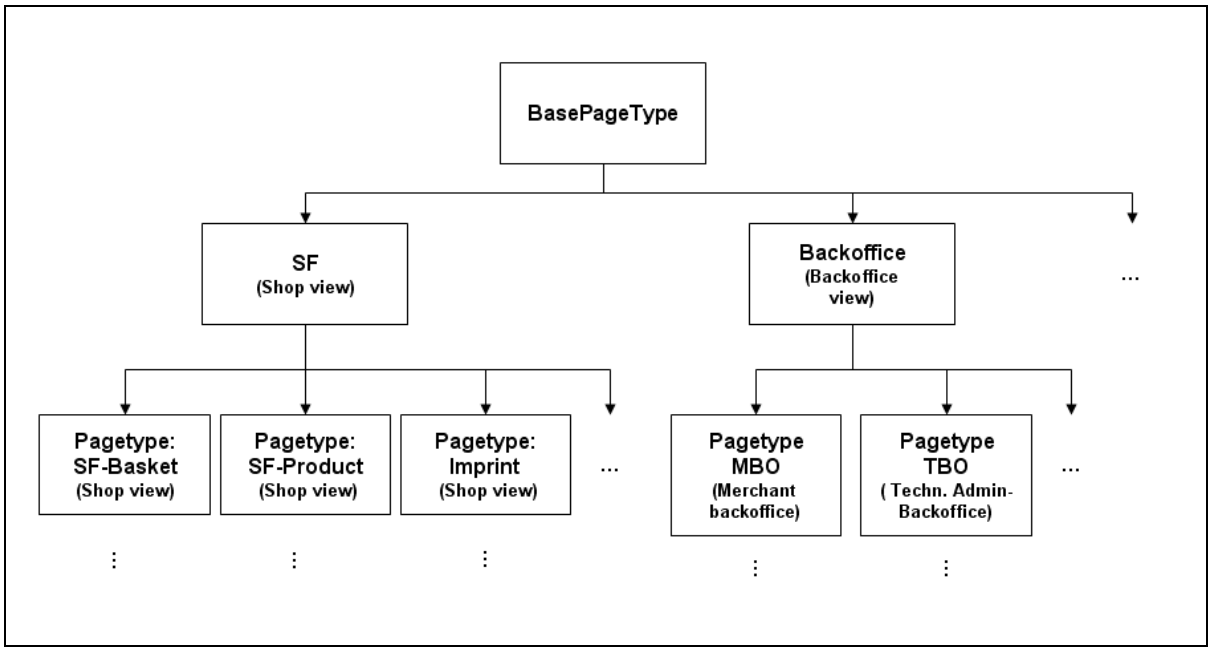

Abbildung 11: PageType-Hierarchie (Ausschnitt)

Durch Vererbung und Erweiterung haben Sie die Möglichkeit, bei der Definition eines PageTypes Bereiche zu übernehmen, einzelne Bereiche durch die Zuordnung eines anderen Templates zu überschreiben oder auch einzelne Bereiche durch Unterbereiche zu erweitern. Ein einfaches Beispiel sehen Sie in *Codebeispiel 13*.

```
<?xml version="1.0" encoding="UTF-8"?>
<epages>
  <Cartridge reference="1" Package="DE_EPAGES::Design">
    <Class reference="1" Path="/Classes/Shop">
      <PageType Alias="SF-Shop" Base="SF" delete="1">
        <Template Name="Content" FileName="SF/SF-Shop.Content.html" />
        <ViewAction URLAction="View" />
      </PageType>
    </Class>
  </Cartridge>
</epages>
```

Codebeispiel 13: PageType mit Vererbung und Überschreiben eines Bereiches

Dieser PageType ist in der Cartridge *Design* definiert und der Objektklasse *Shop* zugeordnet.

Über *Base=...* definieren Sie, von welchem übergeordneten PageType dieser PageType abgeleitet ist. *SF-Shop* ist von *SF* abgeleitet und erbt somit alle Bereichsdefinitionen und Templates.

Dabei wird der logische Bereich *Content* zwar übernommen, aber soll durch ein anderes Template dargestellt werden. Dies wird durch die Zuweisung eines anderen Templates - *SF-Shop.Content.html* - festgelegt.

Mit dem XML-Tag *ViewAction* wird die Aktion *View* der Klasse *Shop* diesem PageType zugewiesen. Das bedeutet, dass bei dieser Aktion das Objekt über diesen PageType dargestellt wird. Sehen Sie dazu auch *AWB 6: Design-Änderungen über PageType*, *Seite 195*.

8.2 Darstellungsebene

Ein PageType findet immer seine Ergänzung in der Darstellungsebene, da mit jedem logischen Bereich, der eingeführt wird, auch ein Template definiert wird, welches die HTML-Darstellung beschreibt.

Aus HTML-Sicht sind die PageTypes Sammlungen von Templates, die festlegen, wie und wo die funktionellen Inhalte und Daten in der HTML-Seite angezeigt werden. Sie bilden sozusagen die Rahmen oder Container für die aktuellen Inhalte, die je nach Funktionalität eingesetzt werden.

Die Templates sind analog zur PageType-Hierarchie aufgebaut. Der *BasePageType* definiert die logischen Bereiche *Page*, *Head* und *Body*, parallel dazu werden die entsprechenden Templates eingeführt, siehe *Codebeispiel 12*. Diese definieren in HTML die allgemeine Grundstruktur, während mit zunehmender Hierarchiestufe die Anzeigefunktionalität immer spezieller wird.

8.2.1 Basistemplate

Das Basistemplate definiert nur die HTML-Seite an sich und erfüllt damit die Forderung nach der allgemeinen Darstellung, siehe *Codebeispiel 14*.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="#INPUT.Language"
xml:lang="#INPUT.Language">
  #INCLUDE("Head")
  #INCLUDE("Body")
</html>
```

Codebeispiel 14: Basistemplate

Alle INCLUDE-Anweisungen innerhalb der Datei tragen Platzhaltercharakter und werden zur Laufzeit durch das aktuelle Template ersetzt. Zu INCLUDE-Anweisungen als TLE lesen Sie Kapitel *TLE*, *Seite 63*.

Die hier definierten Templates sind mit einem allgemeinen Inhalt gefüllt. Sie stellen eine Fallback-Variante dar, falls das Template nicht durch einen Nachfolge-PageType überschrieben wird.

Für das Basistemplate sehen Sie die Inhalte der entsprechenden Templates *BasePageType.Head.html* und *BasePageType.Body.html* in *Codebeispiel 15* und *Codebeispiel 16*.

```
<head>#MENU("Head")
  #INCLUDE(#Template)#ENDMENU
</head>
```

Codebeispiel 15: INCLUDE-Template für Basistemplate -*Head*

```
<body>
</body>
```

Codebeispiel 16: INCLUDE-Template für Basistemplate -*Body*

Sie erkennen, dass durch dieses Templates nur die Rumpfstruktur einer Seite zur Verfügung gestellt wird. Die konkreten Inhalte werden durch Templates dargestellt, die über nachgeordnete PageTypes bereitgestellt werden.

8.2.2 Template-Hierarchie

Die Templates der nächsten Stufe definieren die Inhalte und Struktur von *Head* und *Body*, wobei die allgemeingültigen Anweisungen fest codiert und die zu erwartenden veränderlichen Inhalte mit INCLUDE-Anweisungen belegt werden.

Der Aufbau der HTML-Hierarchie orientiert sich an der PageType-Hierarchie, die per XML in der Datenbank angelegt wurde, siehe *Abbildung 11*.

Die Templates der PageTypes der Stufe 2 bilden das Design der unterschiedlichen Hauptaktionsebenen ab: *Backoffice* für Administrations-Seiten der Technischen Administratoren (TBO), der Business-Administratoren (BBO) und der Händler (MBO), *SF* für die Darstellung der Shop-Seiten.

Diese entsprechenden Templates legen die generelle Darstellung der HTML-Seiten der jeweiligen Arbeitsebene fest und definieren somit die einheitliche Anzeige für HTML-Seiten der Ebenen für alle Funktionen.

Da das Basistemplate für die HTML-Seite den Body nur ganz allgemein beschreibt, müssen die Templates der Ebene 2 diesen Bereich ausführlicher beschreiben. *Codebeispiel 17* zeigt ein Beispiel für den Body der Backoffice Seiten.

```
<body>
  #INCLUDE("Menu")
  <table class="Maintable" cellspacing="0" cellpadding="0">
    <tr>
      <td class="ContextBar">
        #INCLUDE("ContextBar")
      </td>
      <td class="Content">
        #INCLUDE("Content")
      </td>
    </tr>
  </table>
  <div class="Footer">
    <a class="Copyright" href="http://www.epages.de"
       onclick= "openWindow(this.href,'','); return false;">
        {Copyright}
    </a>
  </div>
</body>
```

Codebeispiel 17: Definition des Bodys für Backoffice-Seiten

Im Vergleich dazu zeigt *Codebeispiel 18* ein Beispiel für den Body der Storefront-Seiten:

```
  <div class="Layout1 GeneralLayout">
  #IF(#Shop.ClosedByMerchant OR #Shop.ClosedByProvider)
  <div class="ShopClosed">#Shop.ShopClosedMessage[0]</div>
  #ELSIF(#Shop.LoginRequired AND (NOT #Session.User OR #Sessi-
on.User.IsAnonymous))
    #IF(#Style.HeaderIsVisible)
    <div class="Header">
      #INCLUDE("Header")
    </div>#ENDIF#IF(#Style.TopIsVisible)
    <div class="NavBarTop">
      #INCLUDE("NavBarTop")
    </div>#ENDIF
    <table class="Middle" summary="{LayoutTable}">
      <tr>#IF(#Style.LeftIsVisible)
        <td class="NavBarLeft" abbr="{NavBarLeft}">
          #INCLUDE("NavBarLeft")
        </td>#ENDIF
        <td class="ContentArea" abbr="{ContentArea}">
          {PleaseLogin}
        </td>#IF(#Style.RightIsVisible)
        <td class="NavBarRight" abbr="{NavBarRight}">
          #INCLUDE("NavBarRight")
        </td>#ENDIF
      </tr>
    </table>#IF(#Style.BottomIsVisible)
    <div class="NavBarBottom">
      #INCLUDE("NavBarBottom")
    </div>#ENDIF#IF(#Style.FooterIsVisible)
    <div class="Footer">
      #INCLUDE("Footer")
    </div>#ENDIF
  #ELSE
    #IF(#Style.HeaderIsVisible)
    <div class="Header">
      #INCLUDE("Header")
    </div>#ENDIF#IF(#Style.TopIsVisible)
    <div class="NavBarTop">
      #INCLUDE("NavBarTop")
    </div>#ENDIF
    <table class="Middle" summary="{LayoutTable}">
      <tr>#IF(#Style.LeftIsVisible)
        <td class="NavBarLeft" abbr="{NavBarLeft}">
          #INCLUDE("NavBarLeft")
        </td>#ENDIF
        <td class="ContentArea" abbr="{ContentArea}">

          #INCLUDE("Content")

        </td>#IF(#Style.RightIsVisible)
        <td class="NavBarRight" abbr="{NavBarRight}">
          #INCLUDE("NavBarRight")
        </td>#ENDIF
      </tr>
    </table>#IF(#Style.BottomIsVisible)
    <div class="NavBarBottom">
      #INCLUDE("NavBarBottom")
    </div>#ENDIF#IF(#Style.FooterIsVisible)
    <div class="Footer">
      #INCLUDE("Footer")
    </div>#ENDIF
  #ENDIF
  </div>
```

Codebeispiel 18: Definition des Bodys für Storefront-Seiten

Damit wird die grundlegende Darstellung der HTML-Seiten des Shops definiert. Hier finden Sie auch die Struktur aus *Abbildung 7* wieder.

Die nächste Hierarchieebene sind die PageTypes auf Basis der einzelnen Objektklassen, d. h. zu jeder Objektklasse ist mindestens ein PageType definiert, der die konkrete Darstellung der einzelnen Objekte in der HTML-Seite festlegt. Solche Objektklassen sind z. B. *Basket*, *Product* usw.

Die Anzeige der Objekte erfolgt im Arbeitsbereich in Abhängigkeit der ausgeführten Aktion und der resultierenden Daten.

Demselben Prinzip folgen auch die nachgeordneten Hierarchiestufen, die zum Ziel haben, das Objekt in unterschiedlichen Ansichten darzustellen.

8.2.3 Objektmethode *template*

In den HTML-Dateien werden Templatenamen vorwiegend in Zusammenhang mit der TLE-Anweisung #*INCLUDE* angewendet, z. B. *#INCLUDE("Content")*, siehe *Codebeispiel 18*.

Jedes Objekt kann über die Methode *template* den Templatenamen ändern. Der Methode wird der original Templatename übergeben, z. B.:

```
$Object->template("<templatename>", <$ObjectPageType>)
```

In dieser Methode ist definiert, welches Template das Objekt verwendet. In den meisten Fällen wird das Template verwendet, dessen Name übergeben wurde.

Für Sonderfälle können in dieser Methode andere Templatenamen bzw. Mechanismen zur Auswahl bestimmter Templates definiert werden.

Dies wird z. B. ist für Objekte der Klasse *LineItem* und deren abgeleitete Klassen angewandt. Hier wurde diese Möglichkeit genutzt, um klassenspezifische Templates zu verwenden. Dabei wird der Templatename übernommen und mit dem Alias des aktuellen Objektes erweitert, so dass sich als resultierender Templatename ergibt:

```
<templatenameClassAlias>
```

Ist ein Template mit dieser Bezeichnung nicht definiert, ersetzt diese Methode den aktuellen Alias durch den Alias des nächst übergeordneten Objektes. Auf diese Weise wird die Vererbungslinie des Objektes durchlaufen, bis ein Template gefunden ist. Wird in dem Durchlauf kein gültiger Templatename gefunden, wird das Template verarbeitet, welches nur durch *templatename* bezeichnet ist.

Folgendes Beispiel soll dies verdeutlichen: Bei der Verarbeitung der Anweisung in *Codebeispiel 19* liest der TLE-Compiler den Templatenamen *ContentLine* aus.

```
#WITH(#Shipping)
  #INCLUDE("ContentLine")
#ENDWITH
```

Codebeispiel 19: Beispiel für Template-Aufruf

Durch die #WITH-Anweisung wird der Kontext auf ein Objekt der Klasse *LineItemShipping* gesetzt. Dieses Objekt ersetzt den Templatenamen mit *ContentLineLineItemShipping*, wäre das Template nicht definiert, würde die Methode auf *ContentLineLineItem* zurückgreifen, da *LineItem* die Elternklasse von *LineItemShipping* ist. Findet die Methode entlang der Objektstruktur kein anderes Template, wird als letzte mögliche Variante das allgemeine Template *ContentLine* verarbeitet.

Dieser Mechanismus gestattet es, unter dem einfachen Templatenamen ein Template für den allgemeinen Fall zu definieren und für jede Klasse spezielle Darstellungen festzulegen. Verwendet wird dies z. B. in Zusammenhang mit der Darstellung von LineItems, siehe dazu *LineItems*, *Seite 178*. Siehe auch API-Dokumentation zu *DE_EPAGES/Order/API/Object/LineItem*.

8.3 Verarbeiten der PageTypes

Wie in *Logische Struktur, Seite 41*, erwähnt, wird im PageType die ViewAction definiert, mit der ein Objekt zur Anzeige gebracht wird, sowie die Templates, welche für die Anzeige verwendet werden.

Nach Aufruf der Aktion startet der Prozessor mit dem Template *Page* und arbeitet dann die INCLUDE-Anweisungen ab, bis die Gesamtseite erzeugt ist und dargestellt werden kann.

So kann ein und dasselbe Objekt in einer anderen Umgebung dargestellt werden, je nachdem, in welchem PageType die ViewAction vereinbart wurde.

Abbildung 12 zeigt die Darstellung eines Produktes im Shop, *Abbildung 13* zeigt die Darstellung desselben Produktes in der Administration für den Händler.

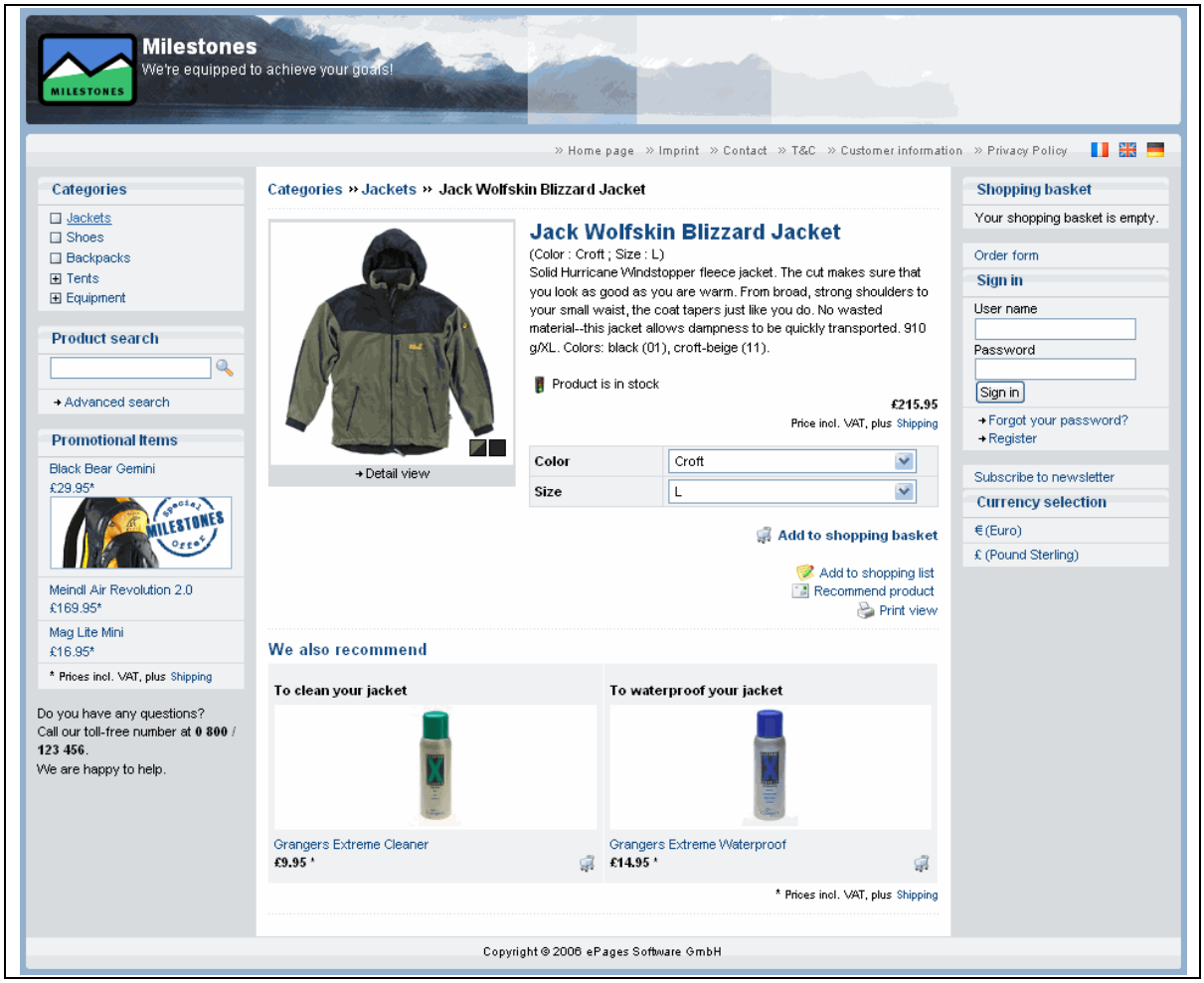

Abbildung 12: Anzeige eines Produktes über SF-PageTypes

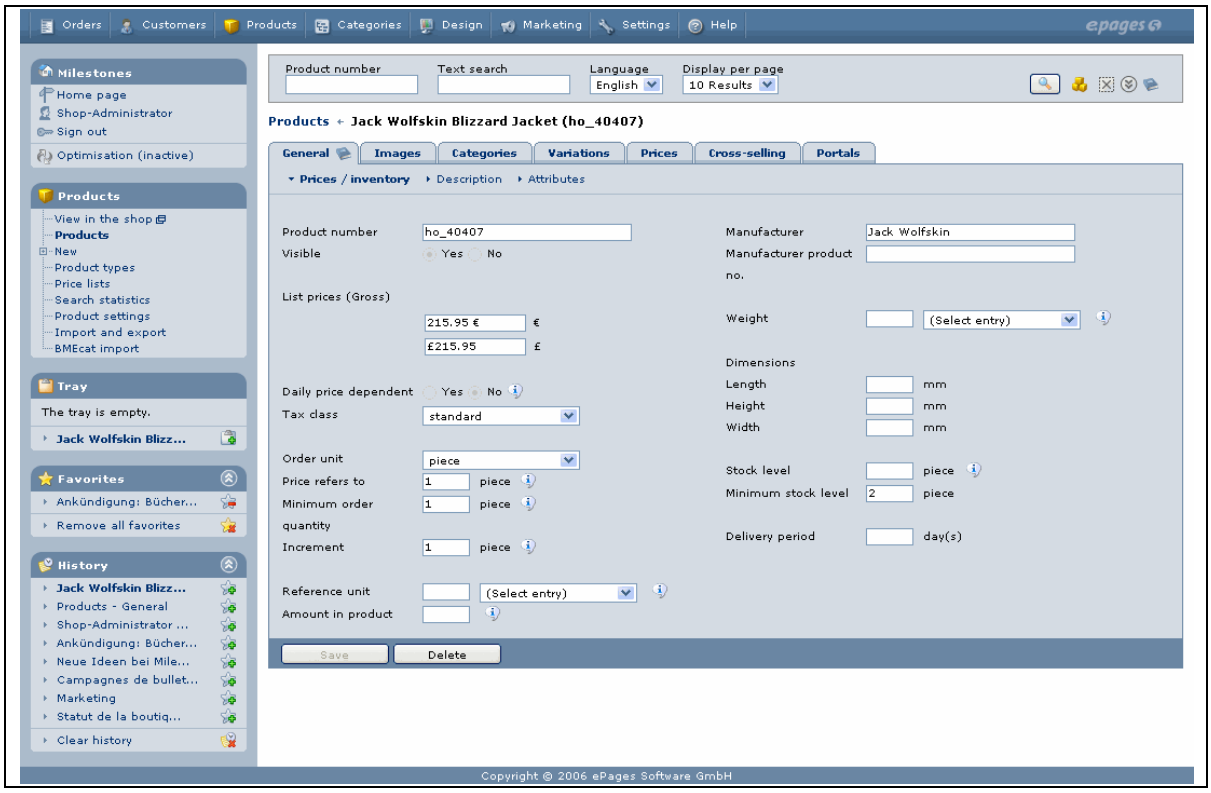

Abbildung 13: Anzeige desselben Produktes über MBO-PageTypes

9. Mehrsprachigkeit – Language-Tags

Mit ePages 5 ist es sehr einfach und komfortabel, Mehrsprachigkeit für die Webapplikation umzusetzen. Dazu wurde die Spracherweiterung *Language-Tags* eingeführt. Language-Tags werden als Platzhalter in Templates an allen Positionen eingesetzt, die je nach Sprachwahl unterschiedliche Inhalte anzeigen sollen.

Hinweis: Die Language-Tags sind keine Platzhalter für sprachabhängige Werte aus der Datenbank, wie z. B. Produktattribute. Die Language-Tags werden nicht durch den Shopbetreiber selbst eingeführt und gepflegt.

Die sprachabhängigen Inhalte für die Language-Tags werden in einfach zu bearbeitenden XML-Dateien abgelegt, wobei hier das Keyword den gleichen Namen wie das Language-Tag hat. Über diese Namensgleichheit ist die Referenz zwischen Platzhalter und Inhalt gewährleistet.

Somit braucht man nur ein Template-Set zu erzeugen und zu pflegen, wenn in allen Templates an den sprachsensitiven Stellen diese Language-Tags konsequent eingesetzt werden.

9.1 Syntax für Language-Tags

Die Syntax für ein Language-Tag im Template ist sehr einfach:

{‹tagname›}

Ein Language-Tag beginnt und endet mit einer geschweiften Klammer. Dazwischen steht der selbst gewählte Variablenname, der auch in der XML-Datei wieder verwendet wird.

Die Referenz dazu in der XML-Sprachdatei hat folgende Struktur:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<epages>
  <Language Language="en">
    <Translation Keyword="tagname">...content...</Translation>
  </Language>
</epages>
```

Codebeispiel 20: Definition eines Language-Tags in XML-Datei

Hinweis: Alle Language-Tags sind *case sensitiv*, d. h. es wird zwischen Groß- und Kleinschreibung unterschieden.

In manchen Fällen enthält der zu einem Language Tag gehörende Text einen Teil, welcher nicht übersetzt werden soll. Das kann z. B. eine Variable sein, die erst zur Laufzeit ausgelesen wird. Solche Textteile werden vom *‹notrans›*Tag eingeschlossen, siehe *Codebeispiel 21*.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<epages>
  <Language Language="en">
    <Translation Keyword="tagname">
      ...content... <notrans>#tleVariable</notrans>...content...
    </Translation>
  </Language>
</epages>
```

Codebeispiel 21: Definition eines Language Tags mit nicht zu übersetzendem Anteil

Für manche Language Tags ist es notwendig, dass für die Übersetzung eine Erklärung mitgegeben wird, damit in der Zielsprache der richtige Begriff verwendet wird. Wenn man zum Beispiel das Language Tag *Export* verwendet, muss im Deutschen definiert werden, ob das Substantiv *Export* oder das Verb *exportieren* eingesetzt werden soll. Auch im Englischen ist nicht klar ob dass Substantiv *export* oder das Verb *[to] export* verwendet werden muss. Um solchen Problemen vorzubeugen, kann man dem Language Tag in der xml-Datei eine Erklärung hinzufügen. Dafür benutzt man das Attribut *Meta*, siehe *Codebeispiel 22*.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<epages>
  <Language Language="en">
    <Translation Keyword="Export" Meta="META: Noun">Export</Translation>
  </Language>
</epages>
```

Codebeispiel 22: Verwendung des Attributs *Meta*

Dadurch wird festgelegt, dass der Übersetzer in der Zielsprache immer das entsprechende Wort für das Substantiv *Export* einsetzt.

9.2 Verwendung der XML-Sprachdateien

Im folgenden Beispiel können Sie verfolgen, wie durch die Verwendung der Language-Tags und der dazugehörigen XML-Dateien die Mehrsprachigkeit für eine HTML-Seite umgesetzt wird. Unabhängig von der Anzahl der anzuzeigenden Sprachen wird dafür nur ein Template benutzt.

Das prinzipielle Vorgehen zeigen wir am Beispiel der Datei *SF.LoginBox.html*, die den Login-Bereich in der Sto*ref*ront anzeigt. Die verwendeten Codeausschnitte sind auf einige sprach-relevante Teile des Quelltextes reduziert, um die Übersichtlichkeit zu wahren.

Zuerst wird eine HTML-Seite entworfen und programmiert:

```
...
    <div class="ContextBoxHead">
      <h1>Customer-Login</h1>
    </div>
...
      <div class="ContextBoxBody">
          <div class="InputLabelling">User name</div>
          <div class="InputField">#WITH_ERROR(#FormError)
...
          <div class="InputLabelling">Password</div>
          <div class="InputField">
            <input class="Login" name="Password" type="password" value="" />
          </div>
        </div>
      </div>
      <div class="ContextBoxBody">
        <input class="Action" type="submit" value="Sign in" /><br />
      </div>
...
```

Codebeispiel 23: Ausgangs-Code für Mehrsprachigkeit

Die Anzeige im Browser sieht wie folgt aus:

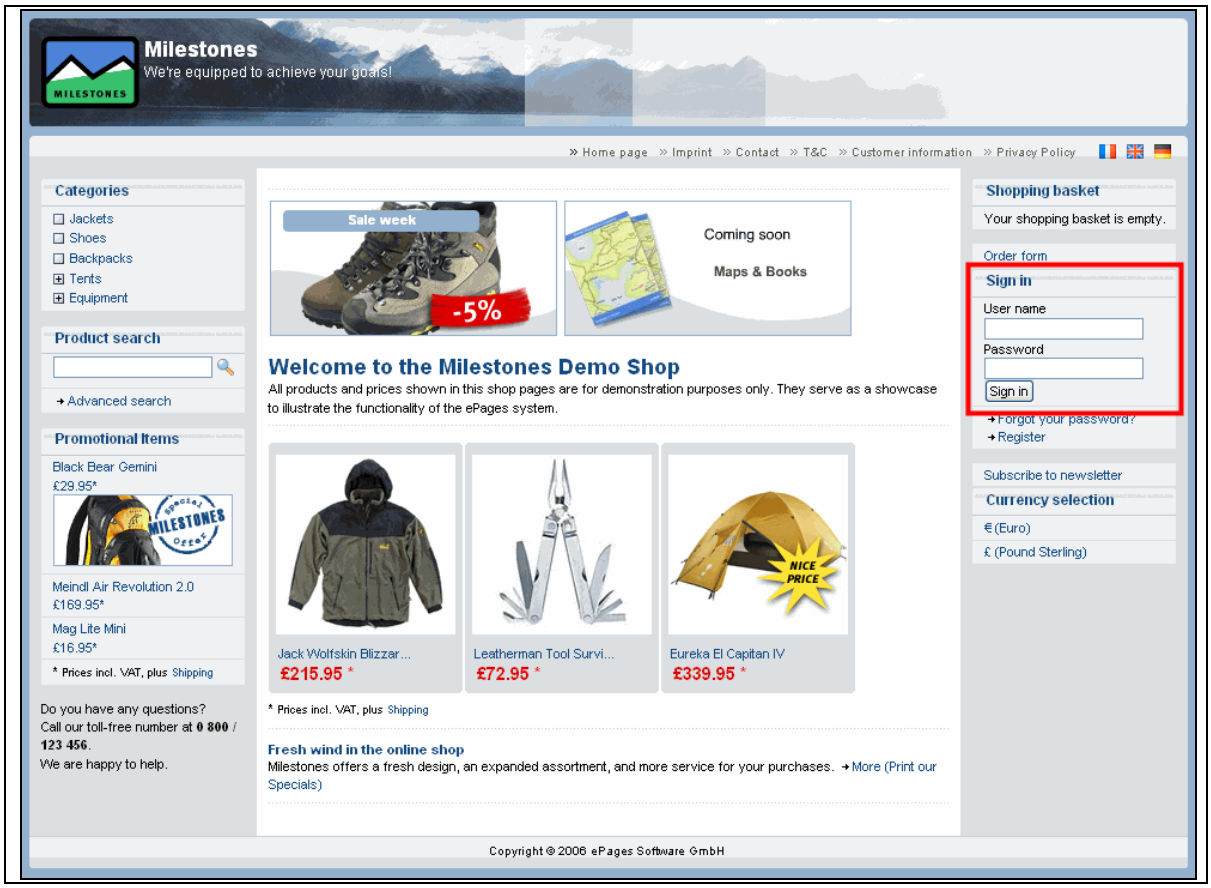

Abbildung 14: Anzeige des sprachunabhängigen Ausgangstemplates

Nun folgt die Überlegung, welche Textteile der HTML-Seite sprachspezifisch dargestellt werden sollen. Für diese werden Language-Tags eingeführt, siehe *Codebeispiel 24*.

```
...
    <div class="ContextBoxHead">
      <h1>{CustomerLogin}</h1>
    </div>
...
      <div class="ContextBoxBody">
          <div class="InputLabelling">{UserName}</div>
          <div class="InputField">#WITH_ERROR(#FormError)
...
          <div class="InputLabelling">{Password}</div>
          <div class="InputField">
            <input class="Login" name="Password" type="password" value="" />
          </div>
        </div>
      </div>
      <div class="ContextBoxBody">
        <input class="Action" type="submit" value="{Login}" /><br />
      </div>
...
```

Codebeispiel 24: Einführung der Language-Tags

Im Browser erscheinen jetzt anstelle der statischen Bezeichner die Language-Tags an den entsprechenden Stellen:

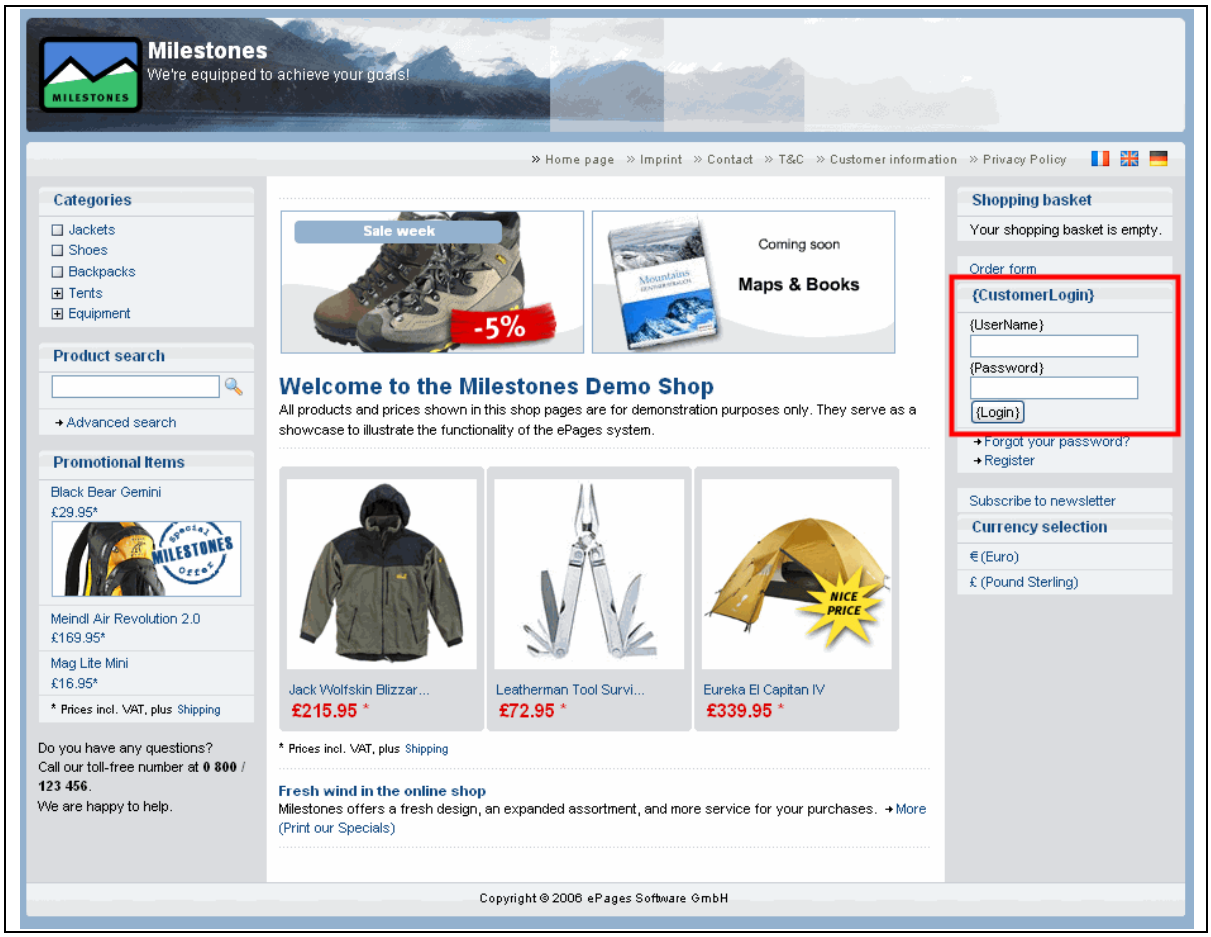

Abbildung 15: Anzeige des Templates mit Language-Tags

Die sprachabhängigen Informationen müssen jetzt in den entsprechenden XML-Sprachdateien bereitgestellt werden und zwar für jede Sprache eine separate XML-Datei.

Die XML-Datei wird in dasselbe Verzeichnis wie die HTML-Datei gespeichert und trägt den gleichen Namen plus eine Erweiterung, die die verwendete Sprache anzeigt. Wenn in unserem Beispiel die Datei für das Template *SF.LoginBox.html*, heißt, so muss die dazugehörige XML-Sprachdatei für deutsch *SF.LoginBox.de.xml*, und für englisch *SF.LoginBox.en.xml*, heißen.

In diese XML-Datei werden die Sprachinhalte für jede Sprache eingetragen, siehe *Codebeispiel 25* und *Codebeispiel 26*.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<epages>
  <Language Language='de'>
    <Translation Keyword="CustomerLogin">Anmeldung</Translation>
    <Translation Keyword="UserName">Benutzername</Translation>
    <Translation Keyword="Password">Kennwort</Translation>
    <Translation Keyword="Login">Anmelden</Translation>
  </Language>
</epages>
```

Codebeispiel 25: XML-Einträge für die Language-Tags in Deutsch

```
<?xml version="1.0" encoding="iso-8859-1"?>
<epages>
  <Language Language='en'>
    <Translation Keyword="CustomerLogin">Sign in</Translation>
    <Translation Keyword="UserName">User Name</Translation>
    <Translation Keyword="Password">Password</Translation>
    <Translation Keyword="Login">Sign in</Translation>
  </Language>
</epages>
```

Codebeispiel 26: XML-Einträge für die Language-Tags in Englisch

Für jede XML-Datei wird die Kodierung angegeben. Wichtig ist, dass Encoding und der in der Datei verwendete Zeichensatz übereinstimmen. Für jede Sprachdatei können Sie bei Bedarf eine eigene Kodierung verwenden.

Bei der Verarbeitung des Templates wird nun je nach angeforderter Sprache die entsprechende XML-Datei ausgelesen, siehe *Abbildung 16* und *Abbildung 17*.

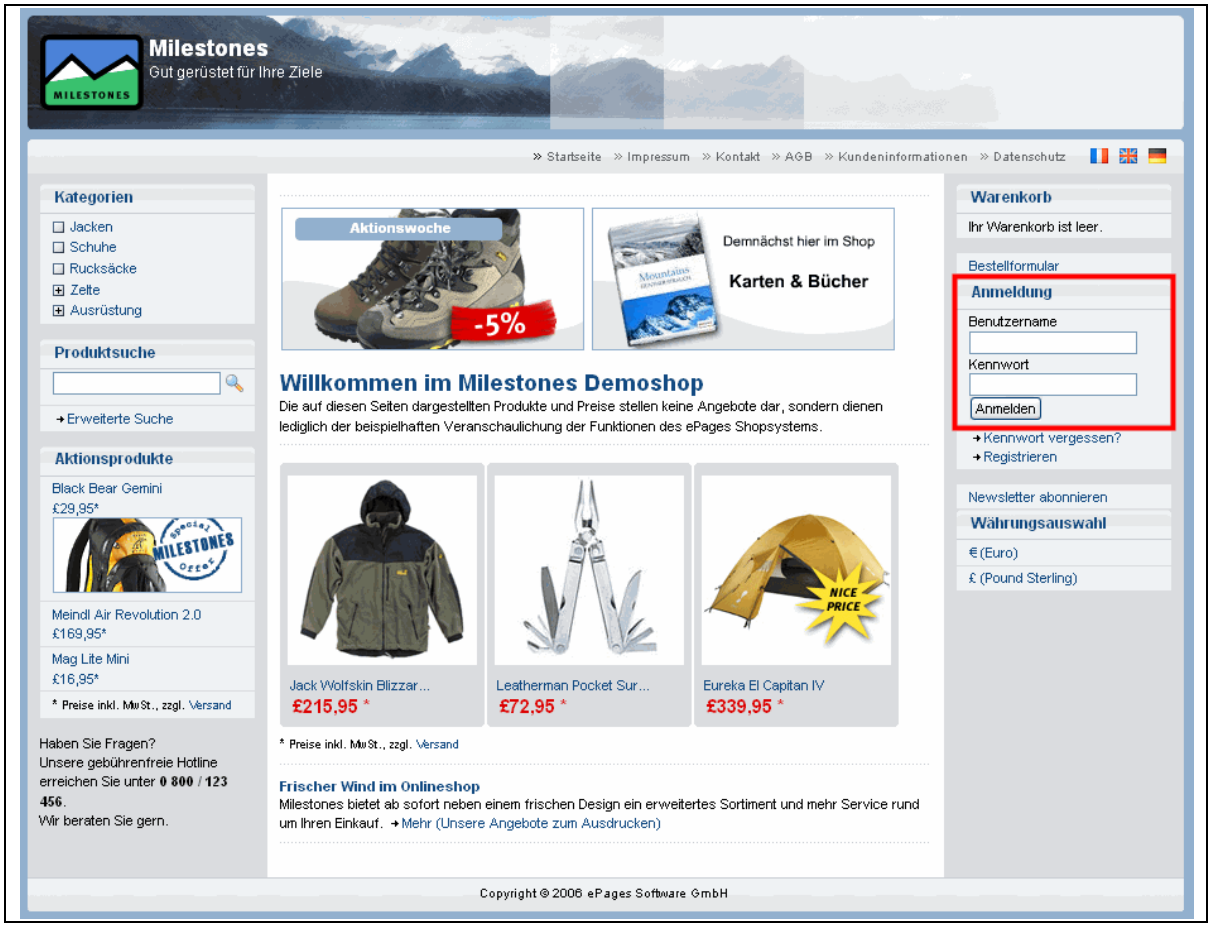

Abbildung 16: Webseite in Deutsch

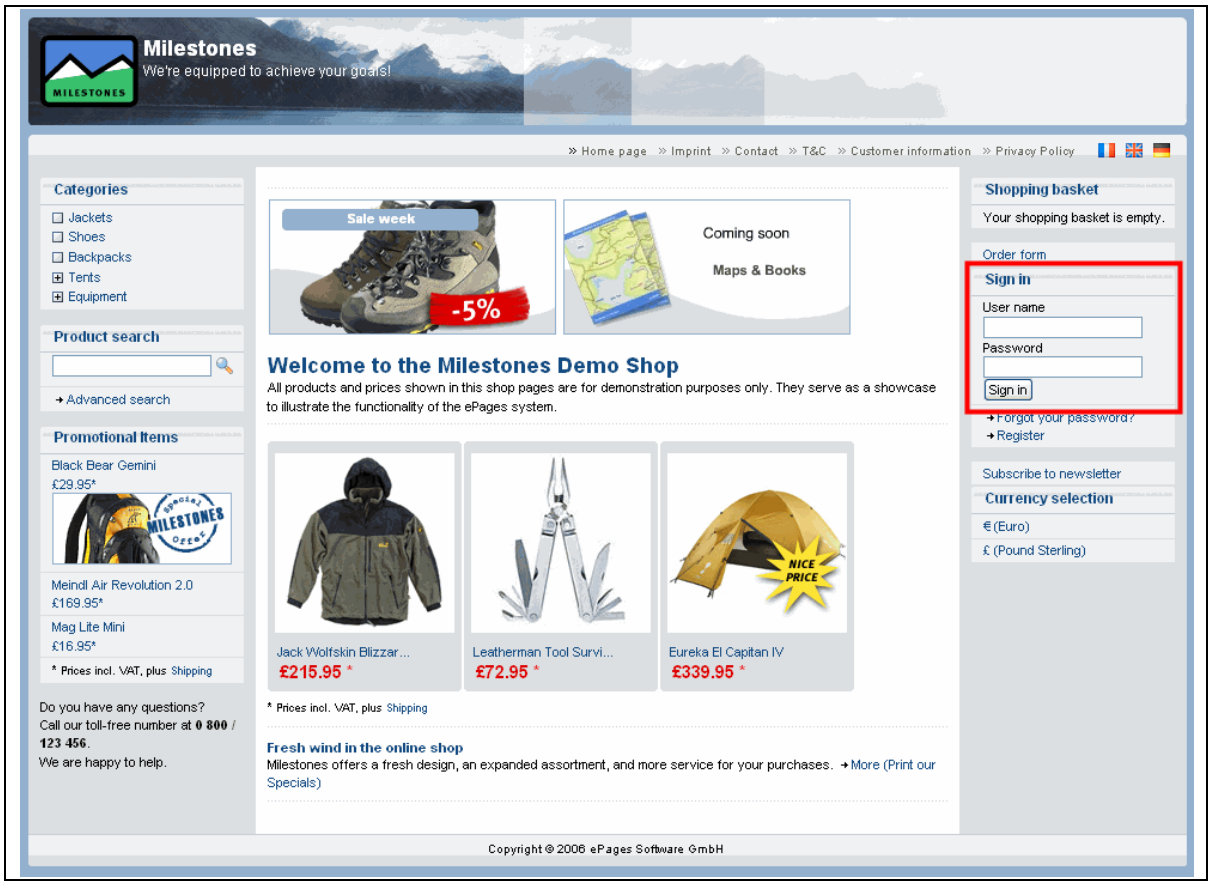

Abbildung 17: Webseite in Englisch

Nach diesem Prinzip führen Sie überall, wo Sie sprachabhängige Inhalte anzeigen wollen, Language-Tags ein und erweitern die XML-Datei entsprechend.

Dadurch ist auch die Erweiterung um eine zusätzliche Sprache sehr einfach. Sie müssen nur die XML-Datei anlegen und die entsprechenden Spracheinträge aufnehmen z. B. für französisch wäre die die Datei *SF.LoginBox.fr.xml* mit dem Eintrag:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<epages>
  <Language Language='fr'>
    <Translation Keyword="CustomerLogin">Connexion</Translation>
    <Translation Keyword="UserName">Nom d'utilisateur</Translation>
    <Translation Keyword="Password">Mot de passe</Translation>
    <Translation Keyword="Login">Connexion</Translation>
  </Language>
</epages>
```

Codebeispiel 27: XML-Eintrag für Language-Tags in Französisch

Als Länderkennzeichen werden die Abkürzungen nach ISO 3166-1 alpha-2 code verwendet.

Hinweise:

1. Voraussetzung für das Anzeigen der zusätzlichen Sprache ist die Aktivierung dieser Sprache für den Shop. Wie eine Sprache aktiviert wird, lesen Sie in den entsprechenden Administrationshandbüchern.

2. Wenn Änderungen, die Sie in den XML-Sprachdateien vorgenommen haben, nicht sofort sichtbar sind, löschen Sie die [ctmpl]-Verzeichnisse. Dies kann auch notwendig werden, wenn Sie Dateien kopieren, die andere Zeitstempel haben.

9.3 Überladen der XML-Sprachdateien

Für die XML-Sprachdateien gibt es die Möglichkeit der Überladung, d. h. man kann durch Verwendung spezieller XML-Dateien die allgemeinen Inhalte überschreiben.

Dazu sind drei verschieden XML-Sprachdateien definiert, deren Abarbeitungsreihenfolge systemseitig festgelegt ist.

Diese sind, in der Reihenfolge ihrer Abarbeitung (Beispiel für Backoffice-Templates in Englisch):

1. *%EPAGES_CARTRIDGES%/DE_EPAGES/Dictionary/Templates/Dictionary.en.xml*
2. *%EPAGES_CARTRIDGES%/DE_EPAGES/‹cartridgename›/Templates/Dictionary.en.xml*
3. *%EPAGES_CARTRIDGES%/DE_EPAGES/‹cartridgename›/Templates/‹templateverzeichnis›/‹templatename›.en.xml*

Dabei werden die XML-Dateien in ihren Sprachinhalten entsprechend der Nummerierung immer spezieller. Das heißt, in 1. stehen die allgemeine Übersetzungen, die für die gesamte Anwendung gelten und auf allen Seiten zu finden sind, während in 3. Übersetzungen erfasst sind, die sich auf ein spezielles Template beziehen und auch nur darin verwendet werden.

Die Datei *%EPAGES_CARTRIDGES%/DE_EPAGES/Dictionary/Templates/Dictionary.en.xml* enthält z. B. die Übersetzungen für solche Texte wie *Ausführen* oder *Speichern*, die auf fast allen HTML-Seiten gleich angezeigt werden, Ausschnitt siehe *Codebeispiel 28*.

```
…
    <Translation Keyword="Back">Back</Translation>
    <Translation Keyword="Next">Next</Translation>
    <Translation Keyword="Finish">Finish</Translation>
    <Translation Keyword="Delete">Delete</Translation>
    <Translation Keyword="New">New</Translation>
    <Translation Keyword="Save" Meta="Verb">Save</Translation>
    <Translation Keyword="Close">Close</Translation>
    <Translation Keyword="Update">Update</Translation>
    <Translation Keyword="Clone">Duplicate</Translation>
    <Translation Keyword="Apply">Apply</Translation>
    <Translation Keyword="RunBatchAction">Execute</Translation>
…
```

Codebeispiel 28: Ausschnitt aus *Dictionary.de.xml*

Der globale Eintrag für *RunBatchAction* wird z. B. an folgender Stelle verwendet:

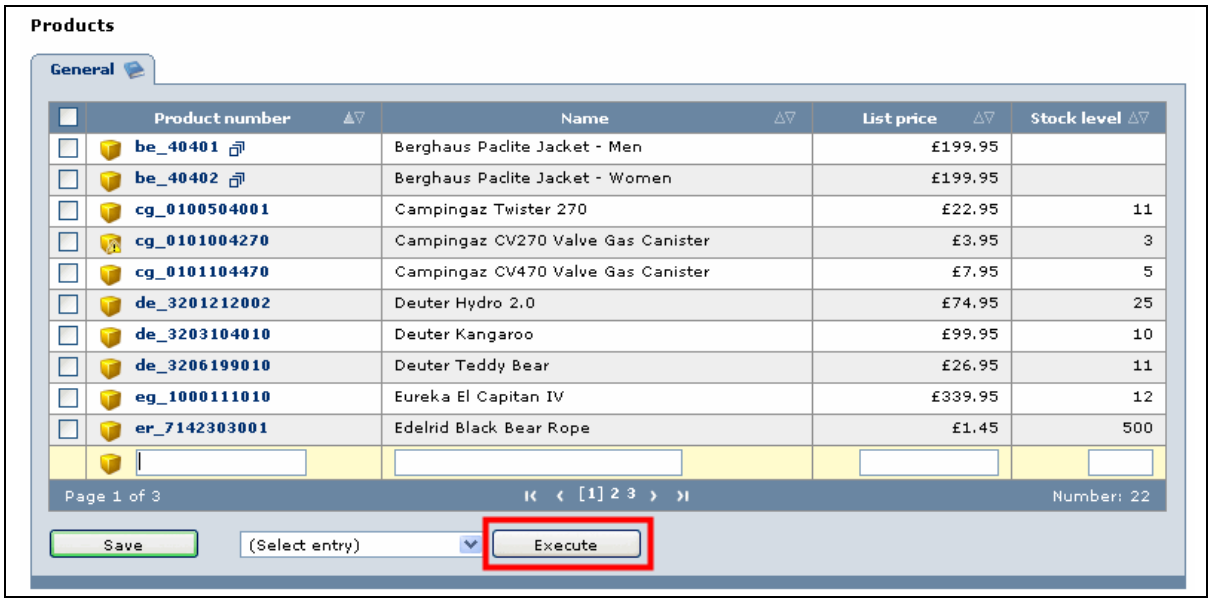

Abbildung 18: Anzeige *Execute* auf Basis *Dictionary.de.xml*

Die Datei *‹templatename›.en.xml* (3.) dagegen enthält nur die Sprachinformationen für das Template gleichen Namens.

Durch Verwendung des gleichen Language Tag-Namens in mehreren dieser XML-Sprachdateien können Sie dieses Language-Tag überladen. Für ein *Keyword* wird der Wert verwendet, der in der speziellsten der drei o. g. Dateien für dieses Keyword gesetzt wird.

Als Beispiel überschreiben wir den Text in *Abbildung 18* mit Hilfe einer Datei vom Typ (3). Das Template, welches die Seite in *Abbildung 18* darstellt, heißt *MBO-Products.TabPage.html*. Im selben Verzeichnis liegt die Datei *MBO-Products.TabPage.en.xml*. Hier fügen wir folgenden Eintrag ein.:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<epages>
  <Language Language="en">
    <Translation Keyword="RunBatchAction">Start action</Translation>
...
  </Language>
</epages>
```

Codebeispiel 29: Überladen eines allgemeinen Language-Tags

Dadurch wird der Inhalt aus der allgemeinen Sprachdatei für *RunBatchAction - Execute - aus Codebeispiel 28* durch den Eintrag *Start action* überschrieben. Nach Verarbeitung des Templates wird folgendes angezeigt:

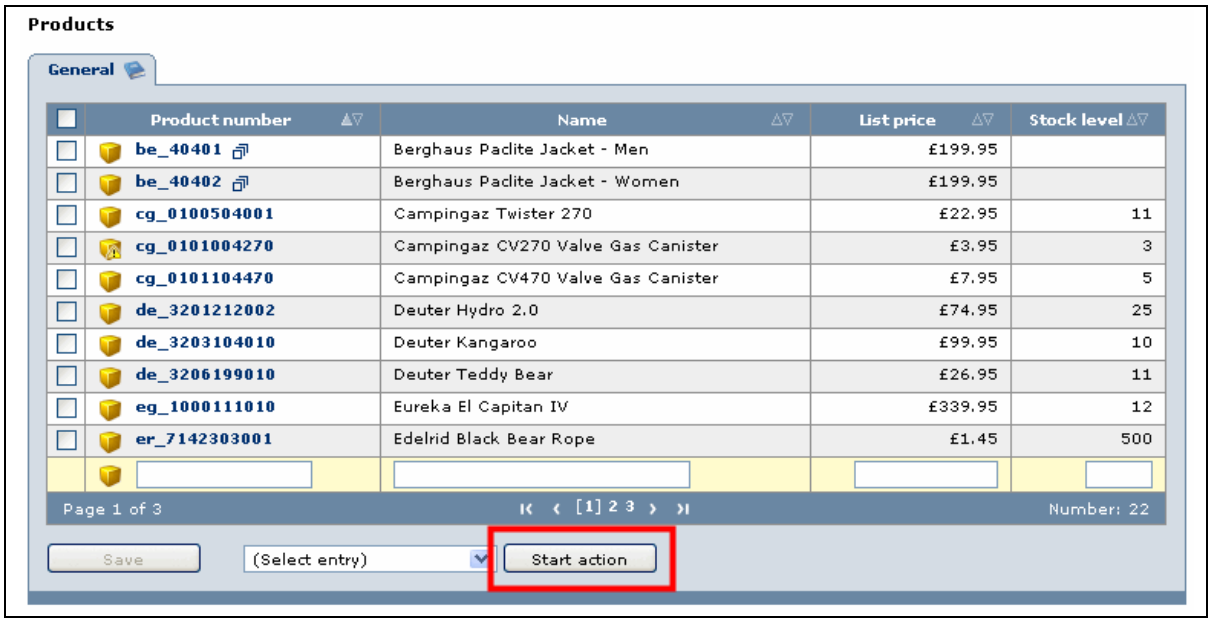

Abbildung 19: Ergebnis des Überladens

Diese Überladung hat so lange Bestand, wie es das Keyword *RunBatchAction* in der Datei *MBO-Products.TabPage.en.xml* gibt. Wird das Keyword aus der Datei oder die Datei selbst gelöscht, kommt wieder der Wert aus der Datei *Dictionary.en.xml* zur Anzeige.

Analog werden die XML-Sprachdateien für die anderen Sprachen angelegt, das *en* wird durch den jeweiligen Language-Code (lc) im Dateinamen ersetzt und für die Language-Tags werden die entsprechenden Übersetzungen eingetragen.

Für die Sprachdateien gilt dann die analoge Reihenfolge:

1. *%EPAGES_CARTRIDGES%/DE_EPAGES/Dictionary/Templates/Dictionary.lc.xml*
2. *%EPAGES_CARTRIDGES%/DE_EPAGES/‹cartridgename›/Templates/Dictionary.lc.xml*
3. *%EPAGES_CARTRIDGES%/DE_EPAGES/‹cartridgename›/Templates/‹templateverzeichnis›/‹templatename›.lc.xml*

Beim Verarbeiten des Templates richtet sich das System nach der aktuell gewählten Anzeigesprache. Ist die aktuelle Sprache englisch, werden die Language-Tags durch die Inhalte aus den XML-Sprachdateien mit der Kennung *en* eingesetzt , ist die Anzeigesprache deutsch, wird *de* ausgewertet.

So kann auf einfache Weise auch auf andere Sprachen, z. B. französisch – *fr*, spanisch – *es* usw. erweitert werden.

Hinweise:

1. Beachten Sie, dass die Sprache, die Sie anzeigen wollen, auch für die Anwendung aktiviert ist! Wie die Sprache aktiviert wird, lesen Sie in den entsprechenden Administrationshandbüchern.

2. Wenn Sie in den Originaldateien Änderungen vornehmen, können diese bei einem späteren Update oder Upgrade überschrieben werden. Um Ihre Installation updatefähig zu halten, nutzen Sie die Möglichkeit zur Überladung von Originaldateien, die auch für die Dictionary-Dateien genutzt werden kann. Siehe dazu *Überladung von Templates*, *Seite 38*.

Sie haben die Möglichkeit, per Script zu überprüfen, ob für eine gewählte Datenbank und Sprache alle Language Tags vollständig ersetzt wurden bzw. ob in den Lokalisierungsdateien redundante Tags vorkommen. Benutzen Sie dafür

```
perl %EPAGES_CARTRIDGES%/DE_EPAGES/Presentation/Scripts/checkLanguageTags.pl
```

Der Aufruf

```
perl %EPAGES_CARTRIDGES%/DE_EPAGES/Presentation/Scripts/checkLanguageTags.pl –help
```

zeigt Ihnen die verfügbaren Optionen und Aufruf-Parameter an. Ein mögliches Beispiel ist

```
perl %EPAGES_CARTRIDGES%/DE_EPAGES/Presentation/Scripts/checkLanguageTags.pl
-storename Store -language en
```

9.4 Lokalisierung von Datenbankinhalten

Auch für zu lokalisierende Datenbankinhalte wie Attributnamen oder Attributbeschreibungen werden Definition und sprachabhängiger Inhalt getrennt. Folgendes Beispiel soll dies verdeutlichen.

In der Cartridge *Product* werden die Features in der Datei *Features.xml* definiert.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<epages>
  <Object reference="1" Alias="Features">
    <Feature Alias="Products" MaxValue="100000" Cartridge="DE_EPAGES::Product"
      delete="1" Position="30" />
    <Feature Alias="Variations" MaxValue="50" Cartridge="DE_EPAGES::Product"
      delete="1" Position="60" />
...
  </Object>
</epages>
```

Codebeispiel 30: Definition von Features

Name und Beschreibung eines Features sollen sprachabhängig im MBO angezeigt werden. Dafür wird pro Sprache eine eigene Sprachdatei angelegt. Dabei gilt folgende Namenskonvention:

Translation.‹filename›.‹language code›.xml

Für unser Beispiel (*Features.xml*) gilt:

- für Deutsch: *Translation.Features.de.xml*
- für Englisch: *Translation.Features.en.xml*

Die entsprechende englische Sprachdatei zu *Codebeispiel 30* sehen Sie in *Codebeispiel 31*.

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<epages>
  <Language Language="en">
    <Object Path="/Features/Products">
      <Attribute Name="Name">Products</Attribute>
      <Attribute Name="Description">Number of products</Attribute>
    </Object>
    <Object Path="/Features/Variations">
      <Attribute Name="Name">Variation attributes for products</Attribute>
      <Attribute Name="Description">Number of attributes which can be used to
        create product variations</Attribute>
    </Object>
...
  </Language>
</epages>
```

Codebeispiel 31: Englische Sprachdatei für *Features.xml*

Die entsprechende deutsche Sprachdatei zu *Codebeispiel 30* sehen Sie in *Codebeispiel 32*

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<epages>
  <Language Language="de">
    <Object Path="/Features/Products">
      <Attribute Name="Name">Produkte</Attribute>
      <Attribute Name="Description">Anzahl von Produkten</Attribute>
    </Object>
    <Object Path="/Features/Variations">
      <Attribute Name="Name">Variationsattribute für Produkte</Attribute>
      <Attribute Name="Description">Anzahl von Attributen, welche zum Anlegen von
        Produktvariationen genutzt werden können</Attribute>
    </Object>
...
  </Language>
</epages>
```

Codebeispiel 32: Deutsche Sprachdatei für *Features.xml*

Die Vorgehensweise hat folgende Vorteile:

- Einheitliche und damit konsistente Übersetzung aller lokalisierbaren Strings
- Cartridgeweise Übersetzung von zu lokalisierbaren Datenbankinhalten möglich
- Importieren der Übersetzung für mehrere Sprachen mit dem Installieren der Cartridge möglich
- XML-Dateien mit Objektdefinitionen brauchen bei sich ändernder Lokalisierung nicht geändert werden

Dieses Konzept gilt für folgende Typen:

- Actions,
- Attributes,
- Features,
- MailTypeTemplates,
- NavBars,
- NavBarElements,
- NavElementGroups,
- PaymentTypes,
- TemplateTypes,
- UnitsOfMeasurement,
- StyleGroups

10. TLE

Mit der Template Language Extension (TLE) ist es möglich, die Inhalte für Webseiten dynamisch zu laden, zu verarbeiten und anzuzeigen. Web-Sites, die Standard-HTML verwenden, können nur "statische" Web-Seiten mit einem ganz bestimmten Inhalt erzeugen. Aus diesem Grund müssen Web-Designer für jede mögliche Anzeigeoption eine eigene Web-Seite erstellen.

Diese Belastung des Web-Designers und der Systemressourcen fällt mit ePages weg, denn ePages erzeugt "dynamische" Web-Seiten zur Laufzeit. Wenn der Kunde einen Link in der Storefront anklickt, wählt ePages das zu verwendende Template und erstellt entsprechend der TLE-Informationen im Template die Seite mit Daten aus der Datenbank. Durch die Möglichkeit, die Daten zur Laufzeit auszulesen und auszuwerten, wird der Entwicklungsaufwand aufwendiger Seiten reduziert.

Die TLE umfasst Variablen und Anweisungen.

10.1 Syntax für TLE

Allen TLE-Variablen und -Anweisungen ist ein # (Doppelkreuz) vorangestellt:

#‹objektattribut›

Bsp: `#Shop.NameOrAlias`

oder

#‹variablenname›

Bsp: `#Alternate`

oder

#‹tle-Anweisung›

Bsp: `#IF(#LongDescription)#LongDescription[0]#ELSE#Description[0]#ENDIF`

Hinweis: Alle TLE-Variablen und –Anweisungen sind casesensitiv, d. h. es wird zwischen Groß- und Kleinschreibung unterschieden.

In den folgenden Kapiteln wird die Syntax für TLE-Variable und –Anweisungen beschrieben und durch Beispiele ergänzt. Die Beispiele sind alle den Original-Dateien entnommen.

Bevor Sie mit den Beispielen in den Dateien arbeiten und diese modifizieren, lesen Sie bitte das Kapitel *Überladung von Templates*, *Seite 38*. Wenden Sie danach die Überladung an, um auf der einen Seite die Anwendung der TLE auszuprobieren und auf der anderen Seite die Funktionsfähigkeit Ihrer ePages 5-Installation zu gewährleisten.

10.2 TLE-Variablen

TLE-Variablen sind Platzhalter für Daten aus der ePages Datenbank. Mit Hilfe dieser Platzhalter werden Daten angezeigt, die sich beim Aufruf der Seite ändern können. Sie fügen eine TLE-Variable wie normalen Text in den HTML-Text ein. Beim Aufruf einer bestimmten Seite wird der Wert der Variablen sofort über einer Datenbankabfrage ermittelt und gemeinsam mit der HTML-Seite angezeigt.

Quellen für die Werte der TLE-Variablen sind:

- Objektattribute,
- URL-Parameter,
- Input-Daten aus HTML-Formularen,
- Cookies,
- ViewActions
- TLE-Funktionen,
- Dynamische TLE-Variable
- Session-Parameter

Die TLE-Variablen aus den genannten Quellen können in allen Templates eingesetzt werden.

Das aktive Objekt ist jenes Objekt, welches durch eine Anzeigeaktion (ViewAction) aufgerufen wurde. Durch dessen PageType werden die notwendigen Templates zur Verfügung gestellt, um die entsprechende Webseite anzuzeigen, siehe auch *Verarbeiten der PageTypes*, *Seite 48*. Diese Objekt bildet den Kontext für die verfügbaren TLE-Variablen.

Wenn Sie z. B. Produktdaten anzeigen möchten und haben die Aktion zum Anzeigen eines Produktes aufgerufen, dann ist das aktive Objekt *Product*. Um im Template den Name des Produktes anzuzeigen, fügen Sie an der entsprechenden Stelle die TLE-Variable *#NameOrAlias* ein. Das System erkennt aus der Aktion, dass es sich um den Kontext des Objektes *Product* handelt und füllt die Variable mit dem Namen des Produktes.

Starten Sie stattdessen die Anzeigeaktion für Nutzer, erkennt das System das Objekt *User* als Kontext. Sie verwenden hier z. B. *#Name*, um den Namen des Nutzers auf der Webseite anzuzeigen.

Um in einem Kontext Daten aus einem andern Kontext anzuzeigen, müssen Sie für die TLE-Variablen den kompletten "Kontextpfad" angeben oder den Kontext ändern.

Wollen Sie z. B. bei der Anzeige eines Produktes *A* den Namen des aktuell angemeldeten Nutzers mit anzeigen, müssen Sie für diese Anzeige den Kontext wechseln, denn der aktuelle Nutzer ist dem Objekt *Product A* nicht zugeordnet.

Der aktuelle Nutzer ist in der aktuellen Session gespeichert. Geben Sie deshalb für die TLE-Variable an: *#Session.User.NameOrAlias*.

Für das Händler-Backoffice und die Storefront gibt es fünf Kontexte, aus denen heraus Sie auf Daten zugreifen können:

- INPUT
- Session
- System
- Shop
- Aktueller Kontext (wie z. B. Produkt, Warenkorb)

Alle Eigenschaften der Objekte können per TLE abgefragt werden.

Beispiele für die Verwendung von TLE-Variablen in Templates sehen Sie im *Codebeispiel 33*.

```
#IF(#IsVisible)
  <div class="ProductDetails">
    <h1>#NameOrAlias</h1>
    <div class="Separator"></div>
    #IF(#ImageMedium)
      <div class="ImageArea">
      #IF(#ImageLarge)
        <a href="#ImageLarge[webpath]" target="_blank">
          <img class="ProductMediumImage" src="#ImageMedium[webpath]"
          alt="#NameOrAlias" title="#NameOrAlias" />
        </a><br/>
         <a class="Action" href="#ImageLarge[webpath]" target="_blank">
           {DetailedView}
         </a>
      #ELSE
        <img class="ProductMediumImage" src="#ImageMedium[webpath]"
        alt="#NameOrAlias" title="#NameOrAlias" />
      #ENDIF
      </div>
    #ELSIF(#ImageSmall) <div class="ImageArea">
      #IF(#ImageLarge)
        <a href="#ImageLarge[webpath]" target="_blank">
          <img class="ProductSmallImage" src="#ImageSmall[webpath]"
          alt="#NameOrAlias" title="#NameOrAlias" />
        </a><br/>
      #ENDIF</div>
    #ENDIF
    <div class="InfoArea">
    #IF(#LongDescription)#LongDescription[0]#ELSE#Description[0]#ENDIF
    <div class="Price">
    #LOOP(#ListPrices)
      #IF(#CurrencyID EQ #INPUT.Currency AND #TaxModel == #Shop.TaxModel)
        #Price[money]
      #ENDIF
    #ENDLOOP
…
```

Codebeispiel 33: Beispiel für Verwendung der TLE-Variablen

Die Daten aus den unterschiedlichen Quellen werden wie folgt ausgelesen:

Objekt: ***#‹(Kontext.)attributname›***

URL-Parameter, Formularfeldinhalte: ***#INPUT.parametername***

Zwischenergebnisse: ***#‹variablenname›***

Hinweis: Bestimmte TLE-Anweisungen ändern den Kontext, siehe *#WITH*, *Seite 69* und *#LOOP*, *Seite 69*.

Eine weitere Möglichkeit, Daten unabhängig vom Kontext anzuzeigen, ist die Verwendung des Attributes *Object.Child*. Mit Hilfe dieses Attributes kann man praktisch jedes Objekt überall anzeigen, wenn der Bezeichner des Objektes bekannt ist. So kann man z. B. die Shopadresse aus dem Impressum auch auf einer Katalogseite anzuzeigen:

```
#Shop.Child.Pages.Child.Imprint.Address
```

10.3 TLE-Anweisungen

TLE-Anweisungen sind ePages-spezifische Befehle, um Templates/Webseiten inhaltgesteuert anzuzeigen.

Das bedeutet, dass Sie z. B. bestimmte Templateabschnitte entsprechend bestimmter Bedingungen abarbeiten lassen oder veränderbare Datenmengen mit Hilfe von Schleifen-Anweisungen komfortabel anzeigen können.

Folgende Anweisungen können Sie einsetzen:

10.3.1 #IF

Mit Hilfe einer #IF-Anweisung können Sie Abschnitte eines Templates bedingt verarbeiten. Einfache Bedingungen erstellen Sie mit #IF und #ENDIF.

Für komplexere Bedingungen stehen das #ELSIF -Tag und das #ELSE -Tag zur Verfügung. Der Wert einer TLE-Variablen wird von der in Echtzeit ausgeführten Auswertung der #IF-Anweisung bestimmt.

Bedingungen und Wertevergleiche können sowohl auf Zeichenketten (TLE-Variablen usw.) als auch auf Zahlenwerte angewendet werden.

Syntax:

#IF (‹ausdruck›) ... #ENDIF

oder

#IF (‹ausdruck›) ... #ELSE ... #ENDIF

oder

#IF (‹ausdruck›) ...

#ELSIF (‹ausdruck›) ...

#ELSIF (‹ausdruck›) ...

...

#ELSE ...

#ENDIF

Ein Beispiel für eine komplexe #IF-Anweisung sehen Sie in *Codebeispiel 34*.

```
#IF(#Class.Alias EQ "Category")
    <div class="ProductListHead">
      <h2><a href="?ObjectPath=#Path[url]">#NameOrAlias</a></h2>
    </div>
    #IF(#Image)
      <div class="ImageArea">
        <a href="?ObjectPath=#Path[url]">
          <img src="#Image[webpath]" alt="#NameOrAlias" title="#NameOrAlias" />
        </a>
       </div>
    #ENDIF
#ELSIF(#Class.Alias EQ "Article")
    <div class="InfoArea">
      <h3><a href="?ObjectPath=#Path[url]">#NameOrAlias</a></h3>
      #Abstract
    </div>
    <div class="Links">
      <a class="Action" href="?ObjectPath=#Path[url]">{More}</a>
      #IF(#Attachment)
        <br />
        <a class="ArticleAttachmentLink" href="#Attachment">
          #IF(#AttachmentTitle)(#AttachmentTitle)#ELSE(#Attachment)#ENDIF
        </a>
      #ENDIF
    </div>
#ENDIF
```

Codebeispiel 34: Beispiel für #IF-Anweisung

10.3.2 #INCLUDE

Die #INCLUDE-Anweisung erlaubt Ihnen, ein Template in ein anderes Template einzubinden. Auf diese Weise können Sie Elemente (wie z. B. Symbolleisten) mit einer einfachen #INCLUDE-Anweisung in mehreren Elementen verwenden. Das eingebundene Template wird zur Laufzeit an Stelle der #INCLUDE-Anweisung eingefügt. Alle Änderungen des HTML-Codes für das eingebundene Template werden umgehend in alle Templates mit der entsprechenden INCLUDE-Anweisung übernommen.

Syntax:

#INCLUDE("‹templatename›"[, "NoDebug"])

Bsp: `#INCLUDE("Content")`

#INCLUDE(#Template)

Bsp: `#INCLUDE(#Template)`

Die zweite Variante wird verwendet, wenn der Templatename noch nicht bekannt ist, sondern dynamisch bereitgestellt wird. Typisch ist die Verwendung in Zusammenhang mit #BLOCK und Menu, siehe Original-Templates.

Der Parameter *NoDebug* ist optional. Wird er im INCLUDE verwendet, wird die Kommentarzeile mit den Templateinformationen im Quelltext nicht mit angezeigt. Siehe dazu *Template-Debugging*, *Seite 38*. Verwenden Sie diesen Parameter in Templates, in denen die HTML-Kommentarzeichen nicht bekannt sind und zu Fehlern führen können. Ein Beispiel dafür sind css-Dateien mit INCLUDE.

10.3.3 #LOCAL

Mit dieser Anweisung legen Sie einen Gültigkeitsbereich im Template fest. Hier vereinbarte Variable bzw. deren aktuelle Werte gelten nur in diesem Bereich.

Syntax:

#LOCAL("‹variablenname›", ‹wert›) ... #ENDLOCAL

```
#LOCAL("TaxClassID", #ID)
  #LOOP(#Shop.TaxMatrix.TaxClasses)
    <option value="#ID"#IF(#TaxClassID AND #TaxClassID NEQ #ID)
    selected="selected"#ENDIF>
      #NameOrAlias
    </option>
  #ENDLOOP
#ENDLOCAL
```

Codebeispiel 35: Beispiel für #LOCAL-Anweisungen

In diesem Beispiel wird die Variable *TaxClassID* in #LOCAL mit einem neuen Wert belegt, der bis #ENDLOCAL gültig ist. Nach #ENDLOCAL nimmt die Variable wieder ihren Ursprungswert an oder wird ungültig, falls sie vor der #LOCAL-Anweisung noch nicht existierte.

10.3.4 #SET

Mit dieser Anweisung setzen Sie eine Variable für den globalen Kontext, d. h. Sie können im gesamten Template darauf zugreifen. Auf eine mit #SET gesetzte Variable kann bis zum Ende der HTML-Seite, die das Template aufruft, zugegriffen werden. Auch wenn die Variable in einem Template vereinbart wird, welches per INCLUDE in die Seite geladen wird, kann bis HTML-Seitenende, also aus dem *Master-Template* darauf zugegriffen werden.

Prinzipiell sollten Sie #SET immer durch #LOCAL/ #ENDLOCAL begrenzen.

Syntax :

#SET("‹variablenname›", ‹wert›)

Bsp: `#SET("FirstNavBarID",#ID)`

oder

#SET("‹variablenname›", ‹ausdruck›)

Bsp: `#SET("Number", #Number + 2)`

10.3.5 #GET

Die Anweisung benutzen Sie, um eine TLE-Variable abzufragen und auf ihren Wert zuzugreifen.

Syntax:

#GET(‹variablenname›)

Bsp: `#GET(#Attribute.Alias)`

Bsp: `#GET("Number")`

oder

#GET(‹ausdruck›)

Bsp: `#GET("Num"."ber")`

Im Fall *#GET("Number")* kann auf #GET verzichtet werden – der Ausdruck *#Number* ist gleichwertig. Wird der Name des anzuzeigenden Attributes erst zur Laufzeit bekannt, die Form wie *#GET(#Attribute.Alias)* verwendet werden.

10.3.6 #CALCULATE

Die Anweisung gibt das Ergebnis eines Berechnungsausdrucks zurück.

Syntax:

#CALCULATE(‹ausdruck›)

Bsp: `#CALCULATE((#ItemNo+1) * 10)`

10.3.7 #WITH

Diese Anweisung löst den aktuellen Kontextbezug für TLE-Variable auf und definiert einen lokal gültigen. Das heißt, Sie legen damit einen neuen Kontext fest, der aber nur innerhalb der #WITH-Anweisung Bestand hat.

Syntax:

#WITH(‹ausdruck›) … #ENDWITH

```
#WITH(#Shop.Categories)
  <option value="#ID">
    #JOIN("/", #PathFromSite) #NameOrAlias #ENDJOIN
  </option>
  #INCLUDE("SubCategories")
#ENDWITH
```

Codebeispiel 36: Beispiel für *#WITH*-Anweisung

Im *Codebeispiel 36* wird per #WITH der Kontext auf *Shop.Categories* gesetzt. Deshalb gibt die Abfrage der *#ID* auch die ID der Shopkategorie zurück. Für das Template, in welches diese #WITH-Anweisung eingebettet ist, galt bis dahin der Kontext Produktkategorie. Eine einfache Abfrage nach *#ID* hätte die ID der aktuellen Produktkategorie ergeben.

10.3.8 #LOOP

Mit #LOOP-Anweisungen können schleifengesteuert verschiedene Listenelemente angezeigt werden. Diese Elemente können Kategorien, Produkte, Artikel im Warenkorb oder Datenstrukturen sein, die ePages in Form von Arrays bereitstellt. Mit #LOOP-Anweisung können Sie in Templates auf einfache Weise Listen erstellen.

Syntax:

#LOOP (‹schleifenvariable›) … #ENDLOOP

Die Schleifenvariable gilt lokal für das Template. Der gesamte HTML-Code und TLE zwischen #LOOP und

#ENDLOOP wird für jedes Element der Schleifenvariable wiederholt.

Innerhalb einer #LOOP-Anweisung wird der Kontext des Templates verlassen und es gilt der Kontext der Schleifenvariablen. Siehe dazu *TLE-Variablen, Seite 63*.

Ein Beispiel für eine #LOOP sehen Sie in *Codebeispiel 37*.

```
#LOOP(#Categories)
  <A HREF="http://#URL_Category">#CategoryName</A>
#ENDLOOP
```

Codebeispiel 37: Beispiel für #LOOP

Hinweis: Die Reihenfolge der Listenelemente wird immer durch die Sortierung in der Administration bestimmt. Siehe dazu im *Handbuch für Händler* Kapitel *Sortieren in Tabellen*.

10.3.9 #JOIN

Diese Anweisung wird verwendet, um eine Schleife zu initiieren, die Zeichenketten ausgibt. Dabei können Sie das Zeichen festlegen, welches als Trennzeichen zwischen die einzelnen Elemente eingefügt wird.

Die Anweisung erweitert also die *#LOOP*-Anweisung um eine Trennzeichendefinition.

Syntax:

#JOIN ("‹zeichen›", ‹schleifenobjekt›) … #ENDJOIN

Bsp: `#JOIN(",",#Users) #Alias #ENDJOIN`

In diesem Beispiel werden alle Nutzer, getrennt durch ein Komma, aufgelistet.

10.3.10 #FUNCTION

Mit Hilfe dieser Anweisung können Sie aus dem Template heraus TLE-Funktionen aufrufen. Die Funktion muss im TLE-Compiler registriert sein.

Syntax:

#FUNCTION("‹funktionsname›", ‹parameter1›, ‹parameter2›, … ‹parameter n›)

Bsp: `#FUNCTION("REFERENCEPRICE", #Product.Object, #Price)`

Die Funktionen geben jeweils einen einzelnen Wert zurück.

10.3.11 #BLOCK

Mit Hilfe der *#BLOCK*-Anweisung wird einer Funktion Template-Code mit übergeben, welcher bei Funktionsaufruf verarbeitet wird. Zurückgegeben wird ein Wert, meist ein String, der den verarbeiteten Template-Code enthält. *#BLOCK* ist eine Erweiterung von *#FUNCTION* um das Template bzw. den Template-Code bis *#ENDBLOCK*.

Syntax:

#BLOCK("‹name›", ‹parameter1›, … ‹parameter n›) … #ENDBLOCK

Bsp: `#BLOCK("MENU","Content")#INCLUDE(#Template)#ENDBLOCK`

Durch den Code im o. g. Beispiel werden alle für das Menu *Content* definierten Einträge auf der Seite dargestellt. Dazu wird die Funktion *Menu* mit dem Parameter *Content* aufgerufen. Das bedeutet, dass aus den entsprechenden PageTypes die Einträge für das Menü mit dem Namen *Content* nacheinander ausgelesen werden. In der TLE *#Template* wird der Name des Templates übergeben, welches die Darstellung des jeweiligen Eintrages übernimmt. Dadurch werden nacheinander die Template-Darstellungen für die einzelnen Einträge in das aufrufende Template eingebunden.

10.3.12 #WITH_LANGUAGE

Sie können diese Anweisung verwenden, um den aktuellen Sprachkontext umzuschalten.

Syntax:

#WITH_LANGUAGE(‹variablenname›) … #ENDWITH_LANGUAGE

```
#LOOP(#Shop.Languages)
  #WITH_LANGUAGE(#LanguageID)
    #LongDescription
  #ENDWITH_LANGUAGE
#ENDLOOP
```

Codebeispiel 38: Beispiel für #WITH_LANGUAGE

Im Beispiel wird eine Schleife über alle aktiven Sprachen des Shops ausgeführt. Für jede Sprache wird über #WITH_LANGUAGE der entsprechende Sprachkontext gesetzt, um die dazugehörige Beschreibung auszulesen. Mit #ENDWITH_LANGUAGE wird diese Sprache dann wieder "abgeschaltet". Außerhalb ist wieder die Sprache aktuell, die vor der *#LOOP*-Anweisung maßgebend war.

10.3.13 #REM

Diese Anweisung gestattet es Ihnen, Bereiche in Ihrem Template zu definieren, die nicht mit verarbeitet werden. In diese Bereiche können Sie Kommentare, Hinweise oder Ähnliches einfügen. Entwickler können diese Möglichkeit nutzen, um während der Entwicklungsphase Code auszublenden.

Syntax:

#REM … #ENDREM

Bsp: `#REM template created by developer X #ENDREM`

Diese Kommentare sind auch nicht in der Quelltextansicht des Browsers sichtbar.

10.4 Fehler-TLE

10.4.1 #FormError

Mit dieser Anweisung fragen Sie ab, ob das zuletzt aufgerufene Formular Fehler enthält. Die Funktion gibt wahr zurück, wenn ein Fehler aufgetreten ist.

Syntax:

#FormError

Bsp: `#IF(#FormError)  Please correct your input!  #ENDIF`

10.4.2 #FormError_‹InputField›

Diese Anweisung benutzen Sie, um Fehler in Eingabefeldern abzufragen.

Syntax:

#FormError_‹inputfeldname›

Eine typische Verwendung zeigt *Codebeispiel 39*.

```
#LOOP(#Products)
  <input name="Price"
    #IF(#FormError_Price)
      Style="color:red"
    #ENDIF
    Value="#Price" />
#ENDLOOP
```

Codebeispiel 39: Beispiel für #FormError_‹InputField›

Hier werden die Produktpreise nacheinander jeweils in Eingabefeldern aufgelistet. Tritt bei einem Produkt im Feld *Preis* ein Fehler auf, wird das entsprechende Eingabefeld rot markiert.

10.4.3 #FORM_ERROR

Diese Anweisung funktioniert exakt wie *#FormError_‹InputField›*, wird aber verwendet, wenn der ERROR-Parameter erst zur Laufzeit bekannt ist.

Syntax:

#FORM_ERROR("‹inputfeldname›")

#FORM_ERROR(‹variablenname›)

Das Beispiel aus *Codebeispiel 39* mit bekanntem Parameter sieht dann wie folgt aus:

```
#LOOP(#Products)
  <input name="Price"
    #IF(#FORM_ERROR("Price"))
      Style="color:red"
    #ENDIF
    Value="#Price" />
#ENDLOOP
```

Codebeispiel 40: Beispiel für #FORM_ERROR

In *Codebeispiel 41* sehen Sie ein Beispiel für die Anwendung für Parameter, die erst zur Laufzeit bekannt sind.

```
#IF(#FORM_ERROR(#Attribute.Alias)) DialogError#ENDIF
```

Codebeispiel 41: Beispiel für #FORM_ERROR und variablem Parameter

10.4.4 #FormErrors.‹...›

Die TLE *#FormErrors* enthält alle Informationen, um eine aussagekräftige Fehlermeldung zu generieren. In der Hauptsache wird hier der Parameter *Reason* genutzt, der eine Fehlerbeschreibung enthält.

Syntax :

#FormErrors.Reason

Bsp: `#IF (#FormErrors.Reason EQ "FORMAT_NOT_INTEGER") {FormatNotInteger} #ENDIF`

Lesen Sie ergänzend dazu *Fehlerbehandlung im Template*, *Seite 103*.

10.4.5 #WITH_ERROR

Mit Hilfe dieser Anweisung können Sie Eingaben aus fehlerhaften Formularen wieder in das Template einarbeiten. Wichtig ist, dass der Feldname identisch ist mit der TLE.

Syntax:

#WITH_ERROR(‹logischer ausdruck›) ... #ENDWITH_ERROR

```
#WITH_ERROR(#FormError)
  <input name="Login" value="#IF(#Login)#Login#ENDIF" />
#ENDWITH_ERROR
```

Codebeispiel 42: Beispiel für #WITH_ERROR

Zu beachten ist, dass die Ersetzung nur im aktuellen Kontext erfolgt. Wird der Kontext geändert, z. B. mit #LOOP oder #WITH, müssen Sie #WITH_ERROR erneut setzen.

10.4.6 #ERROR_VALUE

Benutzen Sie diese Funktion zur Fehlerbehandlung von Auswahlfeldern. Dadurch wird der Code wesentlich übersichtlicher und effektiver.

Syntax:

#FUNCTION("ERROR_VALUE", #ValueIfError, #ValueIfNoError)

```
#WITH_ERROR(#FormError)
 <input name="Alias" value="#Alias">
  #LOCAL("RefUnitID", #FUNCTION("ERROR_VALUE", #RefUnit, #RefUnit.ID))
    <select name="RefUnit" size="1">
      <option value="">{EmptyEntry}</option>#LOOP(#Shop.Units)
      <option value="#ID"#IF(#RefUnitID AND #RefUnitID NEQ #ID)
        selected="1"#ENDIF>#NameOrAlias</option>#ENDLOOP
    </select>
  #ENDLOCAL
#ENDWITH_ERROR
```

Codebeispiel 43: Beispiel für #ERROR_VALUE

Je nachdem, welches Ergebnis die Fehlerfunktion liefert, wird ein gültiger Wert an die Ausgangsvariable übergeben.

10.5 Formatierung von TLE-Variablen

TLE-Variablen können für die Anzeige in der Webseite formatiert werden.

Die Formatierungsanweisung wird in eckigen Klammern an die TLE-Variable angehängt:

#‹TLE-Variable›[‹formatierungsanweisung›]

Sie können folgende Formatierungen verwenden:

Tabelle 8: Formatierungsanweisungen für TLE

Formatter	Bedeutung	Beispiel	Anzeige
[money]	Der Wert wird als Preis angezeigt. Es wird das Währungsformat verwendet, welches durch die aktuelle Währung festgelegt ist.	#Price[money]	25, 95 € $ 25.95
[float]	Der Wert wird als Dezimalzahl angezeigt. Dabei wird das Format verwendet, welches durch die aktuellen Nutzereinstellungen festgelegt ist.	#Quantity[float]	Deutsch: 5.325,26 Englisch (US):5,326.26

Formatter	Bedeutung	Beispiel	Anzeige
[integer]	Der Wert wird als Dezimalzahl angezeigt. Dabei wird das Format verwendet, welches durch die aktuellen Nutzereinstellungen festgelegt ist, jedoch ohne Dezimalstellen	#Quantity[integer]	Deutsch: 5.325 Englisch (US):5,326
[LC]	LowerCase Alle Buchstaben einer Zeichenkette werden als Kleinbuchstaben ausgegeben	#Name[LC]	MUSTERMANN → mustermann
[LCFirst]	Nur der Anfangsbuchstabe einer Zeichenkette wird als Kleinbuchstabe angezeigt.	#Name[LCFirst]	MUSTERMANN → mUSTERMANN
[UC]	UpperCase Alle Buchstaben einer Zeichenkette werden als Großbuchstaben ausgegeben.	#Name[UC]	mustermann → MUSTERMANN
[UCFirst]	Nur der Anfangsbuchstabe einer Zeichenkette wird als Großbuchstabe angezeigt.	#Name[UCFirst]	mustermann → Mustermann
[space:n] [space:-n]	Die anzuzeigende Zeichenkette wird links- oder rechtsbündig angezeigt und nach rechts/von links mit soviel Leerzeichen aufgefüllt, bis die Gesamtzeichenzahl der Angabe in der Klammer entspricht. Das n ist hier ein Platzhalter für eine ganze Zahl. Diese Formatierung ist nur für Plain-Text-Ausgaben sinnvoll, nicht für HTML.	#Name[space:12] #Name[space:-12]	Mustermann →|Mustermann | Elbe →|Elbe | Mustermann →| Mustermann| Elbe →| Elbe|
[slice:n] [slice:-n]	Die anzuzeigende Zeichenkette wird abgeschnitten und mit 3 Punkten beendet (...).Das n ist hier ein Platzhalter für die Anzahl der Stellen, die das Resultat lang ist. Dabei werden die 3 Punkte mitgezählt. Ist die Zeichenkette kürzer als n, werden keine Punkte angefügt. Bei negativem Parameter wird vom Ende der Zeichenkette gezählt, die drei Punkte (...) werden mitgerechnet.	#Name[slice:*9*] #Name[slice:-*9*]	MUSTERMANN → Muster... → ...ermann
[webpath]	Gibt den Pfad zu den aufgerufenen Dateien auf dem Server an. Voraussetzung dafür ist, dass das anzuzeigende Attribut vom Typ Datei oder sprachabhängige Datei ist.	#Image[webpath]	/WebRoot/Store/SF/Shops/Demoshop/.../beispiel.gif
[html]	Ersetzt ‹›&" durch die entsprechenden HTML-Entities (< > & &quot). Diese Formatierung ist die voreingestellte Standard-Formatierung für alle TLE-Variablen.	#Text oder #Text[html]	
[nohtml]	Entfernt alle HTML-Tags (ersetzt diese durch Leerzeichen) Entities werden umgewandelt ü -› ü	#Shop.Slogan[nohtml]	
[0]	Die TLE-Variable soll nicht formatiert werden. Wenn die TLE z. B. Inhalte von Eingabefeldern enthält, die bereits HTML-formatiert sind, soll diese Formatierung nicht geändert werden. Die trifft auf alle Felder zu, die im Backoffice als HTML-Felder gekennzeichnet sind.	#Text[0] #IF (#Shop.BasketOverText) #Shop.BasketOverText[0] #ENDIF	

Formatter	Bedeutung	Beispiel	Anzeige
[uri]	Alle nicht-alphanumerischen Zeichen in den Variablenwerten werden URL-codiert. Nicht-alphanumerische Zeichen sind alle Zeichen außer 0…9, a…z, A…Z, "_".	#Path[uri]	/Shops/Demo.Shop → %2FShops%2F Demo%2EShop
[url]	Alle nicht-alphanumerischen Zeichen in den Variablenwerten werden URL-codiert. Nicht-alphanumerische Zeichen sind alle Zeichen außer 0…9, a…z, A…Z, "_" und „/“.	#Path[url]	/Shops/Demo.Shop → /Shops/Demo%2EShop
[date]	Eine Datum-TLE wird im aktuell eingestellten Datumsformat (nur Datum) angezeigt	#Now[date]	Oct 5, 2005 11:49:33 AM → 05.10.05
[datetime]	Eine Datum-TLE wird im aktuell eingestellten Datumsformat (Datum und Zeit) angezeigt	#Now[datetime]	Oct 5, 2005 11:49:33 AM → 05.10.05 11:49
[time]	Eine Datum-TLE wird im aktuell eingestellten Datumsformat (nur Zeit) angezeigt	#Now[time]	Oct 5, 2005 11:49:33 AM → 11:49
[px]	Für TLE mit Größenangaben (Länge usw.) - wird verwendet für Stylesheet-Angaben	#ContentParagraphSize[px]	10 → 10px
[js]	Für TLE die innerhalb einer Javascript-Zeichenkette verwendet werden. Dort müssen Anführungszeichen und Zeilenumbrüche durch Steuerzeichen markiert werden.	onclick= ”changeSample('#Description[js]');”	I Don’t know → I Don\’t know
[preline]	Zeilenumbrüche werden in br-Tags umgewandelt	#CustomerComment[preline]	Hello, Please deliver punctual this time!!! regards, mr. Customer → Hello,‹br /› Please deliver punctual this time!!!‹br /› regards,‹br /› mr. customer
[neg]	Stellt Werte negativ dar	#Quantity[neg] #Money[neg,money]	5 → -5 25.95 € → -25.95 €

Formatter können durch Komma getrennt nacheinander angegeben werden, z. B.

```
#Shop.NameOrAlias[-20, 30]
```

Das ist wie folgt zu interpretieren:

Durch die erste Formatierung *-20* entsteht aus *#Shop.NameOrAlias[-20*] eine Zeichenkette, die 20 Zeichen lang ist. Fehlende Zeichen werden von vorn aufgefüllt:

```
"          Milestones"
```

Durch die zweite Formatierung *30* wird die Zeichenkette auf 30 Zeichen verlängert. Das Ergebnis sieht wie folgt aus:

```
"          Milestones          "
```

Achtung: Für die sinnvolle Kombination von Formattern sind Sie selbst verantwortlich, es erfolgt kein Plausibilitätsprüfung.

10.6 Operatoren

In den TLE-Anweisungen können Sie folgende Operatoren verwenden:

Tabelle 9: Operatoren für den Vergleich von Zeichenketten

Operator	Bedeutung	Beschreibung	Code-Beispiel
EQ	Gleich (EQual)	Gültig, wenn Parameter gleich sind	#IF(#Class.Alias EQ "Category")
NE	Ungleich (Not Equal)	Gültig, wenn Parameter nicht gleich sind	#IF(#Class.Alias NE "Category")
.	Verkettung	Verknüpfung von Zeichen-ketten	"Mon" . "tag" = "Montag"
IN	Enthalten in		

Tabelle 10: Operatoren für den Vergleich von Zahlenwerten

Operator	Bedeutung	Beschreibung	Code-Beispiel

NEQ
==
	Numerisch gleich (Numeric EQual)	Gültig, wenn numerische Parameter gleich sind	IF(#BillingAddress.ID NEQ #ID)
			IF(#BillingAddress.ID == #ID)

NNE
!=
	Numerisch ungleich (Not Numeric Equal)	Gültig, wenn numerische Parameter ungleich sind	IF(#BillingAddress.ID NNE #ID)
			IF(#BillingAddress.ID ‹› #ID)

NLT
‹
	Numerisch kleiner als (Numeric Less Than)	Gültig, wenn numerischer Parameter 1 kleiner als Parameter 2 ist	IF(#Amount NLT #PC)
			IF(#Amount ‹ #PC)

NLE
‹=
	Numerisch kleiner gleich (Numeric Less or Equal)	Gültig, wenn numerischer Parameter 1 kleiner oder gleich Parameter 2 ist	IF(#Amount NLE #PC)
			IF(#Amount ‹= #PC)

NGT
›
	Numerisch größer als (Numeric Greater Than)	Gültig, wenn numerischer Parameter 1 größer als Parameter 2 ist	IF(#Amount NGT #PC)
			IF(#Amount › #PC)

NGE
=›
	Numerisch größer gleich (Numeric Greater or Equal)	Gültig, wenn numerischer Parameter 1 größer oder gleich Parameter 2 ist	IF(#Amount NGE #PC)
			IF(#Amount =› #PC)

Tabelle 11: Operatoren für die Kombination bedingter Anweisungen

Operator	Bedeutung	Beschreibung	Code-Beispiel
NOT	Nicht	Gültig, wenn Wert nicht gesetzt ist	#IF(NOT #Gender)checked="checked" #ENDIF

Operator	Bedeutung	Beschreibung	Code-Beispiel
OR	Oder	Gültig, wenn ein Ausdruck gültig ist	#IF(#Attribute.IsVisible OR #Value NE "")
AND	Und	Gültig, wenn alle Ausdrücke gültig sind	#IF(#Attribute.IsVisible AND #Value NE "")
DEFINED	Definiert	Gültig, wenn Variable vorhanden	#IF (#DEFINED(#Name))

Hinweis: Der Operator "NOT" hat die höchste Priorität, d. h. er wird vor allen anderen Operatoren angewendet.

Tabelle 12: Rechenoperatoren

Operator	Bedeutung	Beschreibung	Beispiel
+	Plus (Addition)		
-	Minus (Subtraktion)		
*	Multiplikation		
/	Division		
%	Modulo	Rest einer Ganzzahldivision	5 % 3 = 2

10.7 Erstellen einer TLE-Funktion

Wenn Sie eigene TLE-Funktionen bereitstellen wollen, müssen Sie wissen, wie Cartridges angelegt und Hooks angemeldet werden. Siehe dazu *Cartridges*, *Seite 85* und *Hooks*, *Seite 115*.

Um eine TLE-Funktion, siehe *#FUNCTION*, *Seite 70*, oder *#BLOCK*, *Seite 70* zu generieren, müssen Sie neben der Implementation der Funktion diese für den TLE-Prozessor registrieren, indem Sie diese an einem Hook im TLE-Prozessor anmelden.

Zum besseren Verständnis gehen wir von folgendem Beispiel aus: In einem Template sollen die Eigenschaften eines Objektes *CD* angezeigt werden. Dabei können Sie Attribute wie *CDID*, *Title* und *Price* direkt aus der Datenbank abfragen. Eine weitere Eigenschaft *Review* muss von einer externen Quelle ausgelesen werden. Hierfür erstellen Sie die Funktion *CDReview*, die Sie im Template in folgender Form verwenden:

```
#FUNCTION("CDReview", #CDID)
```

Beginnen Sie mit der Implementation der Funktion, siehe *Codebeispiel 44*.

```
package COMPANY::MyCartridge::API::TLE::CDHandler;
…
# FunctionHandler

sub CDReview {
    my $self = shift;
    my ($Processor, $aParams) = @_;

    my $CDID = $aParams->[0];
    my Review = get("http://cdserver/review.cgi?id=$CDID");
    return $Review;
}
sub RegisterHandlerProc {
    my ($Params) = @_;
    __PACKAGE__->new()->register( $Params->{'Processor'} );
    return;
}

…

sub register {
    my $self = shift;
    my ($Processor) = @_;
    GetLog->debug( 'CDHandler.register' );

    $Processor->registerHandler('FunctionHandler', $self, 'CDReview');
    return;
}
…
```

Codebeispiel 44: Implementation einer TLE-Funktion

Dabei ist Folgendes zu beachten:

- Für den Package-Namen sollten Sie sich an die Namenskonvention halten:

  ```
  package <companyname>::<cartridgename>::API::TLE::<handlername>
  ```

- Der Funktionsname im Perl-Code und für die TLE-Funktion müssen übereinstimmen.
- Die Funktion muss am TLE-Prozessor registriert werden. Dafür verwenden Sie die Funktion *register*, den Code entnehmen Sie *Codebeispiel 44*.

Im TLE-Prozessor ist ein Hook für TLE-Funktionen angelegt, an dem die neuen Funktion angemeldet werden muss. Die Anmeldung erfolgt in der Datei *HooksTLE.xml* der Cartridge mit folgender Syntax:

```
<?xml version="1.0" encoding="UTF-8"?>
<epages Cartridge="COMPANY::MyCartridge">

  <Hook reference="1" Name="TLEProcessorRegistration">
    <HookFunction
      FunctionName=
      "COMPANY::MyCartridge::API::TLE::CDHandler::RegisterHandlerProc"
      OrderNo="1"  delete="1" />
  </Hook>

</epages>
```

Codebeispiel 45: Anmeldung der Funktion am Hook im TLE-Prozessor

Wenn der TLE-Prozessor bei Ausführung des Hooks feststellt, dass eine entsprechende Funktion angemeldet ist, prüft er, ob diese Funktion im Template verwendet wird und registriert diese.

Ein Spezialfall der TLE-Funktionen ist die *#BLOCK*-Anweisung, siehe auch *#BLOCK*, *Seite 70*. Hier wird neben den Funktionsparametern der HTML-Quelltext zwischen *#BLOCK* und *#ENDBLOCK* als zusätzlicher Parameter *$cTemplate* übergeben und verarbeitet:

```
my ($Processor, $aParams, $aTemplate) = @_;
```

Mit Hilfe der Diagnostics-Cartridge können Sie alle aktuellen TLE-Funktionen auslesen.

10.8 Erstellen einer dynamischen TLE-Variable

Eine dynamische TLE-Variable ist mit der TLE-Funktion vergleichbar. Im Gegensatz zur TLE-Funktion werden hier keine Parameter übergeben:

```
#<dynVarName>
```

Analog zur TLE-Funktion muss neben der Implementation eine Registrierung und Anmeldung am Hook im TLE-Prozessor erfolgen.

In Erweiterung des Beispiel für eine TLE-Funktion wollen wir zwei Variablen einführen, eine für die Anzeige der CD des Tages, *CdOfTheDay*, und eine zur Anzeige der Top Ten, *TopTen*. Diese können dann im Template wie folgt verwendet werden:

```
#CdOfTheDay
#TopTen
```

Sie beginnen wieder mit der Implementation, siehe *Codebeispiel 46*.

```
package COMPANY::MyCartridge::API::TLE::CDHandler;
…

sub tle {
    my $self = shift;
    my ($Processor, $TLEName) = @_;

    if( $TLEName EQ 'CDOfTheWeek' ) {
        return get("http://cdserver/cdoftheweek.html");
    }
    elsif( $TLEName EQ 'TopTen' ) {
        my @TopTen;
        foreach my Position (1..10) {
            push @TopTen, {
                Artist => 'Artist',
                Title => 'Title',
                Position => $_Position
            };
        }
        return \@TopTen;
    }
    return undef;
}
…
sub existsTLE {
    my $self = shift;
    my ($Processor, $TLEName) = @_;
    GetLog->debug( "CDHandler.existsTLE($TLEName)" );

    return (defined $self->tle($Processor, $TLEName)) ? 1 : 0;
}
…
sub register {
    my $self = shift;
    my ($Processor) = @_;
    GetLog->debug( 'CDHandler.register' );

    $Processor->registerHandler('VariableHandler', $self, 'CDOfTheWeek');
    $Processor->registerHandler('VariableHandler', $self, 'TopTen');
    return;
}
…
```

Codebeispiel 46: Implementation von TLE-Variablen

Dabei ist Folgendes zu beachten:

- Für den Package-Namen sollten Sie sich an die Namenskonvention halten:

  ```
  package <companyname>::<cartridgename>::API::TLE::<handlername>
  ```

- Die Funktion zur Ermittlung der Variableninhalte muss *tle* heißen.
- Die Funktion *existsTLE* muss implementiert sein. Diese prüft, ob eine Variable des im Template angegeben Namens existiert. Ist dies der Fall, wird der Wert ermittelt und angezeigt. Anderenfalls wird der ursprünglich Ausdruck im Template entsprechend verwendet. Wenn im Template z. B. *#FFEEBB* steht, wird geprüft, ob unter diesem Namen eine dynamische Variable vereinbart ist. Ist die Prüfung negativ, wird die Angabe als Farbcode interpretiert.
- Die Funktion muss am TLE-Prozessor registriert werden. Dafür verwenden Sie die Funktion *register*, den Code entnehmen Sie *Codebeispiel 46*.
- Die Funktion muss am Hook des TLE-Prozessors angemeldet werden. Siehe dazu *Codebeispiel 45*, *Seite 78*.

Die *TopTen* können unter Verwendung der *#LOOP*-Funktion angezeigt werden.

10.9 Erstellen eines TLE-Formatters

Das Vorgehen zum Erstellen von TLE-Formattern ist analog zu dem für TLE-Funktionen und dynamischen TLE-Variablen.

Als Beispiel soll ein Formatter entstehen, mit dem Preise in der Form *xx,- €* angezeigt werden, also ohne Dezimalstellen, z. B. *19,- €*. Die Verwendung im Template wäre:

```
#Price[nodec]
```

Sie beginnen wieder mit der Implementierung der Formatierungsfunktion, siehe *Codebeispiel 47*:

```
package COMPANY::MyCartridge::API::TLE::CDHandler;
…
sub Format {
    my $self = shift;
    my ($Processor, $Format, $Value, $TLEName) = @_;

    if( $Format EQ 'nodec' ) {
        my $NoDec = int($Value);
        return "$NoDec,-";
    }
}

sub register {
    my $self = shift;
    my ($Processor) = @_;

    $Processor->registerHandler('FormatHandler', $self, 'nodec');
    return;
}
…
```

Codebeispiel 47: Implementation eines TLE-Formatters

Dabei ist Folgendes zu beachten:

- Für den Package-Namen müssen Sie sich an die Namenskonvention halten:

  ```
  package <companyname>::<cartridgename>::API::TLE::<handlername>
  ```

- Die Funktion zur Formatierung der Variablen muss *Format* heißen.
- Die Funktion muss am TLE-Prozessor registriert werden. Dafür verwenden Sie die Funktion *register*, den Code entnehmen Sie *Codebeispiel 47*.
- Die Funktion muss am Hook des TLE-Prozessors angemeldet werden. Siehe dazu *Codebeispiel 45*, *Seite 78*.

Teil II:

Cartridge-Entwicklung

11. Cartridges

ePages 5 basiert auf Cartridges. Cartridges sind Softwaremodule, die Funktionen und Design bereitstellen und über Abhängigkeiten und Vererbungsmechanismen verbunden sind. Über die API kommunizieren dieses Module miteinander als auch mit anderen Anwendungen.

Für komplexe Änderungen bzw. Funktionserweiterungen sollten Sie als Entwickler stets eigene Cartridges anlegen und in das System einbinden. Damit sichern Sie die Update-Fähigkeit Ihres Systems.

Praktische Beispiele für die Erstellung von Cartridges finden Sie im Anhang in den Anwendungsbeispielen.

11.1 Struktur einer Cartridge

Die Standard-Cartridges sind im ePages 5-Installationsverzeichnis unter

```
%EPAGES_CARTRIDGES%/DE_EPAGES
```

abgelegt. Für jede Cartridge gibt es ein Verzeichnis mit dem Namen der Cartridge, in dem alle notwendigen Dateien gespeichert werden. Folgende Struktur wird verwendet:

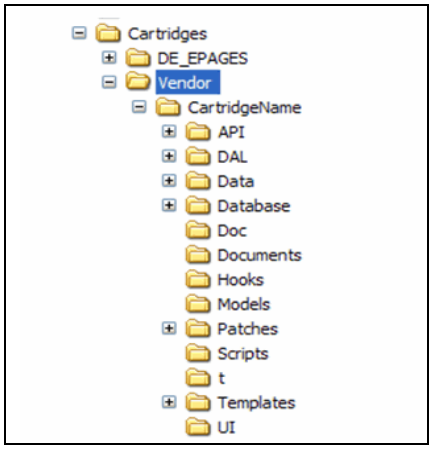

Abbildung 20: Cartridge-Struktur

Für jedes Projekt oder Firma sollte neben */DE_EPAGES* ein eigenes Verzeichnis angelegt werden, das *Projekt- oder Vendor-Verzeichnis*.

```
%EPAGES_CARTRIDGES%/<vendor>
```

Darin werden alle zum Projekt gehörenden Cartridges erstellt. Für jede Cartridge wird ein separates Verzeichnis, das Cartridge-Verzeichnis mit dem Cartridge-Namen erstellt:

```
%EPAGES_CARTRIDGES%/<vendor>/<cartridgeName>
```

Darin können die folgenden Unterverzeichnisse enthalten sein:

Tabelle 13: Wichtige Cartridge-Unterverzeichnisse

Unterverzeichnis	Inhalt
API	Beinhaltet Perl-Module, die öffentliche Funktionen bereitstellen. Öffentlich heißt, die Funktionen können auch von den Modulen anderer Cartridges genutzt werden können. Diese Funktionen sollten daher gut dokumentiert sein.

Unterverzeichnis	Inhalt
DAL	Database Abstraction Layer - Ablegen von Funktionen, die Datenbankzugriffe ausführen Bei Benutzung des PowerDesigners von Sybase werden die Dateien automatisch generiert. Stellt einen Daten-Cache bereit, in dem die Ergebnisse wiederholter Datenbankabfragen zwischengespeichert werden.
Data	Daten zur Darstellung und Steuerung der Anwendung, wie Bilder, Styles, Templates, Steuerdateien für automatisierte Abläufe.
Data/Private	Daten, auf die nur der Applikationsserver zugreift, z. B. Import-Dateien oder Templates. Diese Daten werden bei der Installation in das Verzeichnis *%EPAGES_STORES%/Store* kopiert. Templates, die Original-Templates überschreiben werden in das entsprechende "Überladungs-Verzeichnis" *%EPAGES_STORES%/Store/Templates/DE_EPAGES/‹originalcartridgename›/Templates* kopiert. Siehe dazu auch *Überladung von Templates*, *Seite 38*. Diese Templates müssen in der Cartridge im entsprechenden Verzeichnis *Data/Private/Templates/DE_EPAGES/‹originalcartridgename›/* abgelegt werden. Dabei ist Store die Business Unit, in der die Cartridge installiert wird
Data/Public	Daten, auf die der Webserver zugreift, z. B. Bilder, Styles. Die Dateien werden im Laufe der Cartridge-Installation in das Verzeichnis *%EPAGES_WEBROOT%/Store* kopiert.
Data/Scheduler	Alle Dateien für zeitgesteuerte Abläufe, wie z. B. automatischer Verfügbarkeitsaktualisierung. Dazu gehören die Dateien, welche die Tasks ausführen und entsprechende Steuerdateien. Die Dateien werden nutzerspezifisch nach Betriebssystemen geordnet gespeichert. Siehe dazu *Scheduler*, *Seite 125*.
Data/WebRoot	Hier können z. B. zusätzliche Hilfedateien abgelegt werden, die, analog zur Standard-Online-Hilfe, die Funktion der Cartridge erläutern. Siehe dazu auch *Integration der eigenen Online-Hilfe*, *Seite 171*.
Database	
Database/Sybase	Dateien zum Generieren benötigter Tabellen und zur Definition von Stored Procedures
Database/XML	Import-Dateien im XML-Format, siehe dazu *Import-Dateien*, *Seite 119*.
Doc	API-Dokumentation der Perl-Module Siehe dazu die Anwendung von *Installieren - nmake*, *Seite 89* mit dem Target sourcedoc.
Documents	Ergänzende Dokumente für Entwickler. Für jede Cartridge kann in der Diagnostics-Cartridge eine Beschreibung mit angezeigt werden. Dafür muss im Verzeichnis *Documents* eine Datei *index.html* mit dem entsprechenden Inhalt bereitgestellt werden.
Hooks	Perl-Module, die Funktionalitäten bereitstellen, welche über Hooks aufgerufen werden - siehe dazu *Hooks*, *Seite 115*.
Models	Datenbankmodell der Cartridge, welches mit dem PowerDesigner erstellt wurde. Das Modell wird als .pdm hinterlegt.
Patches	Dateien, die beim Patchen verwendet werden. Siehe dazu *Patchen von Cartridges*, *Seite 167*.
Scripts	Perl-Scripte, die über die Kommandozeile ausgeführt werden. Dies sind Funktionen, die z. B. als Jobs gestartet werden oder als Hilfsfunktionen für Import/Export, Löschen von Datenbankobjekten usw. dienen. Siehe dazu auch *Scheduler*, *Seite 125*.

Unterverzeichnis	Inhalt
t	Test-Cases für API-Funktionen Für jede API-Funktion sollte es mindestens einen Test-Case mit API-Aufruf und Funktionstest geben. Zusätzlich sollten Testdateien, wie beispielsweise Bilder oder .xml-Dateien hier abgelegt werden. Siehe dazu auch *Installieren - nmake*, *Seite 89* – Target test.
Templates	Alle Templates, die für die Funktion dieser Cartridge notwendig sind und keine Original-Templates überschreiben
UI	Module, die Funktionen zur Interaktion mit dem Nutzer (Eingaben, Ausgaben) bereitstellen; Meist private Funktionen, die nicht nach außen sichtbar sind

11.2 Anlegen einer Cartridge-Struktur

Sie können Cartridges manuell anlegen oder das Script zu erstellen aufrufen, welches die notwendige Struktur erstellt und die wichtigsten Dateien anlegt. Wir empfehlen die Nutzung des CreateCartridge-Scriptes, um Fehler beim manuellen Anlegen von Struktur und Dateien zu vermeiden.

Das Perl-Script *CreateCartridge.pl*, rufen Sie wie folgt auf:

```
perl %EPAGES_CARTRIDGES%/DE_EPAGES/Cartridge/Scripts/CreateCartridge.pl <ven-
dor>::<cartridgeName>
```

Dadurch wird im Cartridge-Verzeichnis der ePages-Installation,

```
%EPAGES_CARTRIDGES%
```

das Verzeichnis *‹vendor›* angelegt und in diesem das Cartridge-Verzeichnis *‹cartridgeName›* für die neue Cartridge. In diesem werden alle erforderlichen Verzeichnisse und Dateien erzeugt. Die Inhalte müssen Sie anschließend den Erfordernissen Ihrer Cartridge anpassen.

Im Hauptverzeichnis der neuen Cartridge wird auch die Datei *Makefile.pl* angelegt, mit der Sie dann die Datei *makefile* generieren, siehe *Installieren - nmake*, *Seite 89*.

In Abhängigkeit der Cartridge-Funktionalität und Zielstellung müssen Sie die Struktur anpassen, die einzelnen Dateitypen erstellen und in die entsprechenden Verzeichnisse ablegen. Lesen Sie dazu *Struktur einer Cartridge*, *Seite 85* und orientieren Sie sich an den Anwendungsbeispielen bzw. den Standard-Cartridges.

11.3 Installer / Cartridge.pm

Eine weitere wichtige Datei, die beim Anlegen der Cartridge automatisch im Verzeichnis */API* erzeugt wird, ist *Cartridge.pm*. In dieser Datei wird u. a. definiert, welche Funktionen beim Installieren der Cartridge ausgeführt werden. Es wird ein so genannter Installer angegeben, der steuert ob und wie bestimmte Dateien kopiert oder in die Datenbank importiert werden.

Die Installer sind hierarchisch aufgebaut und vererben die Funktionen. Eine Liste der Installer sehen Sie in *Tabelle 14*. Die Installer sind ihrer linearen Abhängigkeit nach angeordnet, die Basis ist der *BaseInstaller*. Eine Übersicht über die einzelnen Funktionen des jeweiligen Installers finden Sie in der API-Dokumentation des angegeben Packages.

Tabelle 14: Installer

Installer	Package / Bedeutung
BaseInstaller	*DE_EPAGES::Core::API::BaseInstaller* Grundlegende und alle notwendigen Funktionen für Installation/Deinstallation und Patchen einer Cartridge; Kopierfunktionen, Importfunktionen; setzt aktuelle Versionsnummer für betreffende Cartridge
FileInstaller	*DE_EPAGES::Core::API::FileInstaller* Kopieren und Löschen von Files aus dem Unterverzeichnis */Data* an die entsprechenden Stellen im Dateisystem.
XML-Installer	*DE_EPAGES::XML::API::XMLInstaller* Funktionen zum Einlesen und Verarbeiten von XML-Dateien, die Informationen über Cartridge-Abhängigkeiten und Patch-Funktionen beinhalten, *Dependencies.xml*, *Patch.xml*
DatabaseInstaller	*DE_EPAGES::Database::API::DatabaseInstaller* Funktionen zum Anlegen und Bearbeiten des Datenbank-Verzeichnisses und Patchen der entsprechenden *sql*-Dateien
TriggerInstaller	*DE_EPAGES::Trigger::API::TriggerInstaller* Funktionen zum Installieren, Deinstallieren und Patchen der Dateien vom Typ *Hooks*.xml* mit abschließendem Cache-Reset
CartridgeInstaller	*DE_EPAGES::Cartridge::API::CartridgeInstaller* Funktionen u. a. für die Registrierung einer Cartridge in der Store-Datenbank, zur Aktualisierung des Cartridge-Status in der Cartridge-Tabelle, für Ermittlung von abhängigen Cartridges oder für die Aktualisierung der Datenbank-Struktur dieser Cartridge
ObjectInstaller	*DE_EPAGES::Object::API::ObjectInstaller* Funktionen zum Installieren, Deinstallieren und Patchen von Objekten, Klassen und Attributen; Verarbeiten der Dateien vom Typ *Attributes*.xml*
PermissionInstaller	*DE_EPAGES::Permission::API::PermissionInstaller* Funktionen zum Installieren und Deinstallieren von Aktionen und Berechtigungen; Verarbeiten der Dateien vom Typ *Actions*.xml* und *Permissions*.xml*
PresentationInstaller	*DE_EPAGES::Permission::API::PresentationInstaller* Funktionen zum Installieren und Deinstallieren von PageTypes und Formularen; Verarbeiten der Dateien vom Typ *PageTypes*.xml* und *Forms*.xml*
ShopInstaller	*DE_EPAGES::Shop::API::ShopInstaller* Funktionen zum Installieren und Deinstallieren von Features; Verarbeiten der Dateien vom Typ *Features*.xml*
DesignInstaller	*DE_EPAGES::Design::API::DesignInstaller* Funktionen zum Installieren und Deinstallieren von Design- und Navigationselementen sowie E-Mail-Formulare und vordefinierte Such-Anweisungen; Verarbeiten der Dateien vom Typ *NavBars*.xml*, *NavElements*.xml*, *Search*.xml*, *MailTypeTemplates*.xml*
ShippingInstaller	*DE_EPAGES::Shipping::API::ShippingInstaller* Funktionen zum Installieren und Deinstallieren von Versandmethoden; Verarbeiten der Dateien vom Typ *Shipping*.xml*
PaymentInstaller	*DE_EPAGES::Payment::API::PaymentInstaller* Funktionen zum Installieren und Deinstallieren von Zahlungsmethoden und zugehörigen Logos und Logo-Verknüpfungen; Verarbeiten der Dateien vom Typ *PaymentLogos*.xml* und *PaymentTypes*.xml*

Beim automatischen Anlegen einer Cartridge wird als Standard-Installer der DesignInstaller angegeben, siehe *Codebeispiel 48*.

```
#============================================================================
# §package      Training::AddBatchAction::API::Cartridge
# §state        public
#----------------------------------------------------------------------------
# §description  main cartridge class for install/patch/uninstall
#============================================================================
package Training::AddBatchAction::API::Cartridge;
use base qw (DE_EPAGES::Design::API::DesignInstaller);

use strict;
...
```

Codebeispiel 48: Definition des Cartridge-Installers

Der DesignInstaller sorgt dafür, dass alle allgemein gebräuchlichen Dateien an die entsprechenden Stellen kopiert bzw. in die Datenbank importiert werden, sofern sie in der Cartridge bereitgestellt sind.

Hinweis: Für Cartridges, die auf der Site installiert werden sollen, muss der *PresentationInstaller* verwendet werden.

11.4 Installieren - nmake

Für die Installation und Anmeldung der Cartridge im System wird das Kommando *nmake* für Windows bzw. *make* für Linux benutzt. Dabei werden notwendige Schritte wie Kompilieren, Linken, Dateien kopieren usw. automatisch bzw. scriptgesteuert ausgeführt. Voraussetzung dafür ist die Datei *makefile*, die für jede Cartridge erzeugt werden muss und die entsprechende betriebssystem-spezifische Anweisungen enthält.

Generieren Sie zuerst *makefile* für Ihre Cartridge. Öffnen Sie die Konsole in Ihrem Cartridge-Verzeichnis und geben folgenden Befehl ein:

```
perl Makefile.pl
```

Nach Ausführung des Befehls ist die Datei *makefile* im gleichen Verzeichnis angelegt.

Hinweis: *Makefile.pl* erzeugt ein betriebssystem-spezifisches *makefile*.

Um eine Cartridge zu installieren, muss *makefile* mit dem Target-Parameter *install* für eine bestimmte Business-Unit ausgeführt werden:

```
nmake install STORE=Store
```

Dabei ist *Store* der logische Name der Datenbank der Business-Unit. Nach Abschluss sind die Funktionen der Cartridge im System bekannt und können in der Business-Unit genutzt werden, in welcher die Cartridge installiert wurde.

Mit Hilfe von *nmake* können noch verschiedene andere Aktionen ausgeführt werden, abhängig vom logischen Ziel, welches mit dem Aufruf angegeben wird. Der allgemeine Aufruf für *nmake* über die Windows-Kommandozeile sieht daher wie folgt aus:

```
nmake <target> STORE=<Store>
```

Dabei können folgende logische Ziele verwendet werden:

Tabelle 15: Targets für *nmake*

Target	Bedeutung
install	Die Cartridge bzw. die Funktionen werden in der Datenbank installiert und damit bekannt gemacht, benötigte Tabellen und Stored Procedures werden angelegt. Weiterhin erfolgt bei Bedarf ein Import notwendiger Daten. Dateien werden in die entsprechenden Verzeichnisse kopiert. Ist die neue Cartridge von anderen Cartridges abhängig und diese sind noch nicht installiert, wird die Installation für die benötigten Cartridges mit ausgeführt.
uninstall	Alle cartridge-relevanten Einträge werden aus der Datenbank entfernt. Cartridges, die von dieser abhängen, werden mit deinstalliert.
clean	Ergibt eine "auslieferungsfähige" Cartridge-Struktur. Die Struktur wird bereinigt, alle notwendigen Dateien bleiben erhalten, überflüssige werden gelöscht. Gelöscht werden Dateien mit der Endung *.pdb, die Datei makefile und das Verzeichnis */Generated* werden entfernt.
register	Dadurch werden Cartridge-Funktionen auf der zentralen Administrations-Datenbank (Site) bekannt gemacht und können dort angezeigt werden. Nach der Registrierung können die Features der Cartridges durch den Business-Administrator verwaltet und den einzelnen Shoptypen zugeordnet werden
unregister	Der Cartridge-Eintrag wird wieder aus der Site-Datenbank entfernt.
test	Alle Test-Cases aus dem Verzeichnis */t* werden ausgeführt. Im Fehlerfall wird eine Meldung angezeigt.
sourcedoc	Erzeugen der Online-Dokumentation aus den entsprechend kommentierten Perl-Modulen Ihrer API-Funktionen Dabei wird ein Verzeichnis */Doc* angelegt, in das die Hilfedateien im HTML-Format generiert werden. Die Hilfe kann nur dann erzeugt werden, wenn Module nicht verschlüsselt sind. Siehe dazu auch *Encryption*, *Seite 93*.
generate	Generieren von Perl-Code, SQL-Dateien für Tabellen und Stored Procedures und Anlegen von XML-Import-Dateien aus dem Daten-Modell; Voraussetzung ist ein vorliegendes Datenbankmodell in Form einer **.pdm* Datei, die aus dem PowerDesigner heraus erzeugt wurde. Im Verzeichnis */Generated* werden die Unterverzeichnisse */API*, */DAL*, */Database* und */t* mit den entsprechenden Dateien angelegt. Diese Dateien werden in die entsprechenden Cartridge-Verzeichnisse kopiert und weiter bearbeitet.

11.5 Deinstallieren

Um eine Cartridge zu deinstallieren, öffnen Sie die Konsole in Ihrem Cartridge-Verzeichnis und geben folgenden Befehl ein:

```
nmake uninstall STORE=Store
```

Dabei ist *Store* der logische Name der Datenbank der Business-Unit, auf welcher die Cartridge installiert ist. Alle Datenbankeinträge, für die in den entsprechenden xml-Definitionen das Attribut *delete="1"* gesetzt ist, werden gelöscht.

Alle Dateien, die aus der Cartridge in die entsprechenden Überladungsverzeichnisse kopiert wurden, werden gelöscht. Siehe dazu *Überladung von Templates*, *Seite 38*.

Sollten nach Deinstallation noch Inhalte oder Elemente der gelöschten Cartridge angezeigt werden, kann es sein, dass diese Daten noch im Cache gespeichert sind. Überprüfen und leeren Sie bei Notwendigkeit folgende Caches:

- Browser-Cache
- Optimierung im MBO
- Verzeichnis *%EPAGES_STATIC%*

11.6 Cartridge-Verzeichnisse kopieren

Änderungen in einer Cartridge erfordern meist ein zeitaufwendiges Deinstallieren und erneutes Installieren der Cartridge. Wenn sich die Änderungen ausschließlich im Verzeichnis */Data* befinden, ist ein Kopieren der Daten an die entsprechenden Stellen ausreichend. Dafür gibt es ein Script. Mit diesem kann man für eine Cartridge die Dateien aus dem Verzeichnis */Data* in die jeweilige Business Unit kopieren. Das Script wird wie folgt aufgerufen:

```
perl C:\epages5\Cartridges\DE_EPAGES\Installer\Scripts\copyCartridgeData.pl
     [options] [flags] cartridges
```

Optionen sind:

- `-passwd` : Datenbanknutzerkennwort
- `-storename` : Name der Datenbank, auf welcher die Cartridge installiert ist
- `-type` : Type der Cartridge-Daten (Private, Public, WebRoot, Scheduler)

Flags:

- `-help` : Anzeige der verfügbaren Optionen

Beispiel:

```
perl C:\epages5\Cartridges\DE_EPAGES\Installer\Scripts\copyCartridgeData.pl
     DE_EPAGES::Design -storename Store -type Public
```

11.7 Backoffice-Erweiterungen

Neben Änderungen in der Storefront gehören Funktionserweiterungen im Backoffice zu häufig gestellten Anforderungen. Hier werden die Funktionen in verschieden Ebenen zur Verfügung gestellt, die ineinander verschachtelt sind. Es können 5 solcher Ebenen unterschieden werden, deren Anordnung und Beziehung zueinander auch im Layout der Backoffice-Seiten sichtbar sind, siehe *Abbildung 21*.

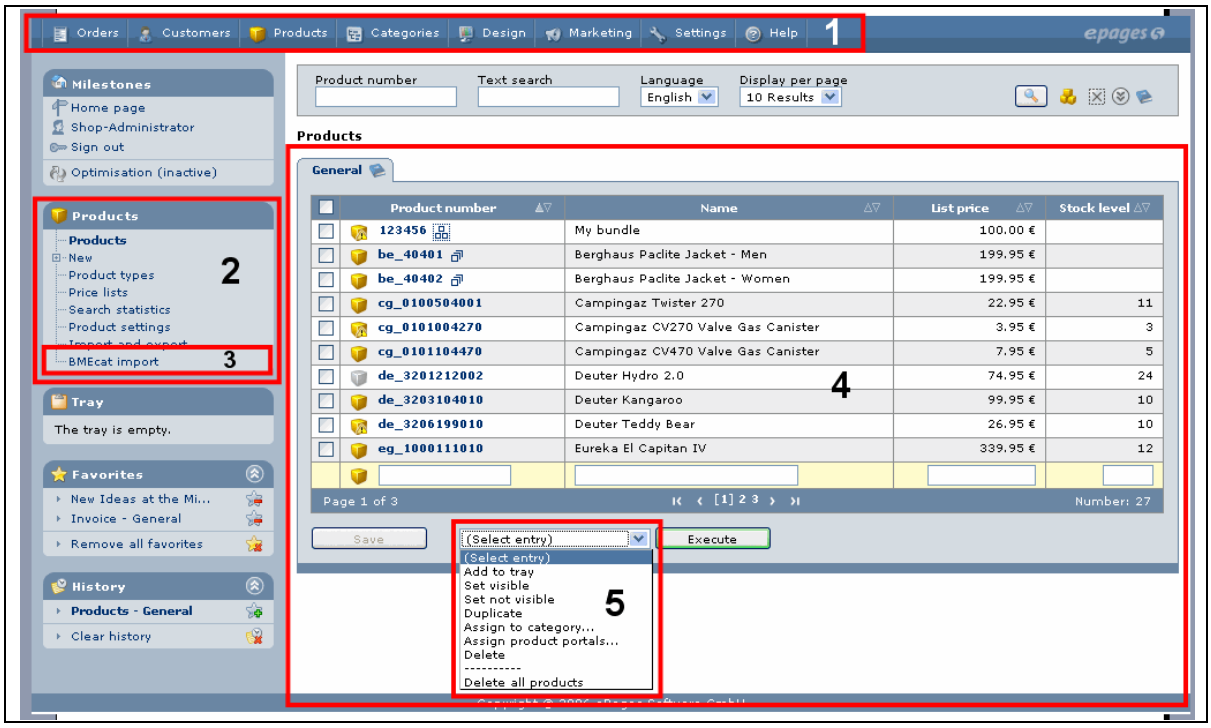

Abbildung 21: Funktionsebenen im Backoffice

Die folgende Bereiche bieten sich zur Integration neuer Funktionen an:

Tabelle 16: Funktionsbereiche im Backoffice

Bereich	Bedeutung
1 Hauptnavigationspunkte (Manager)	Hauptnavigationspunkte sind separate Funktionsbereiche ohne Bezug zu den anderen Managern mit eigener, gekapselter Funktionalität. Wollen Sie die Anwendung um einen neuen logischen Administrationsbereich erweitern, erstellen Sie einen neuen Hauptnavigationspunkt. Beispiel: Lieferantenverwaltung
2 Navigationsbox im linken Navigationsbereich	Navigationsboxen werden in die linken Navigationsbereiche eines jeden Moduls integriert und stellen verschiedene Funktionsblöcke zur Verfügung. Dabei kann man unterscheiden zwischen modulabhängiger Funktionalität, wie das Kontextmenü oder übergreifenden Funktionen wie Ablage oder Favoriten. Beispiel: Box zur Sprachumschaltung oder Anzeige von Datum/Uhrzeit
3 Menüeintrag in einer Navigationsbox	Weitere Funktionen in einzelnen Navigationsboxen Beispiel: Link zur Homepage oder zum Intranet im Administratormenü
4 Karteikarte	Funktionalität im Inhaltsbereich, prinzipiell in Karteikarten angeordnet. Beispiel: Karteikarte Statistik im Modul Bestellungen
5 Stapelverarbeitungsaktion	Erweiterung der Stapelverarbeitungsaktionen für eine Tabelle Beispiel: siehe unten

Ein praktisches Beispiel für die Erweiterung der Backoffice-Funktionalität durch eine zusätzliche Stapelverarbeitungsaktion finden Sie in *AWB 7: Neue Stapelverarbeitungsaktion im MBO, Seite 201*.

12. Erstellung einer Distribution

Ein wichtiger Grund, funktionelle Erweiterungen in eine Cartridge zu kapseln, ist die Wahrung der Upgradefähigkeit des Systems. Ein weiterer Grund ist die Bereitstellung der Funktionen für andere Systeme. Dazu muss die Cartridge einfach übertragbar und installierbar sein.

Dazu erstellen Sie eine Distribution. Eine Distribution ist eine Cartridge mit allen für die Funktionen notwendigen Dateien, die auf dem Zielsystem nur noch installiert werden muss.

Eine solche Distribution erstellen Sie mit Hilfe von *nmake* mit dem Target *clean*. Wenn Sie Ihre Cartridge auf Ihrem System erfolgreich gestestet haben und eine Distribution erzeugen wollen, geben Sie in der Konsole im Hauptverzeichnis Ihrer Cartridge den Befehl

```
nmake clean
```

ein. Dadurch wird die Cartridge-Struktur bereinigt, alle nicht mehr notwendigen Dateien werden gelöscht.

Die so vorbereitete Cartridge kann nun auf das Zielsystem kopiert und installiert werden.

Wollen Sie die Funktionalität bereitstellen, aber den Quellcode vor unberechtigter Wiederverwendung bzw. Änderung schützen, bietet sich an, die Quelltext-Dateien zu verschlüsseln:

12.1 Encryption

Um den Quelltext zu verschlüsseln, steht Ihnen ein Verschlüsselungs-Tool zur Verfügung. Das Programm heißt *encrypt.exe* befindet sich im Verzeichnis

```
%EPAGES%/bin
```

Um Ihre Dateien zu verschlüsseln, geben Sie folgenden Befehl ein:

```
encrypt [-s] <perlmodul>.pm
```

Den Parameter *-s* können Sie verwenden, um bei der Verschlüsselung Kommentarzeilen aus dem Code zu entfernen. *Encrypt.exe* muss für jede Datei einzeln ausgeführt werden. Als Ergebnis entsteht eine verschlüsselte Datei *‹perlmodul›.pm*. Beim Aufruf des Programms ohne Parameter werden die möglichen Parameter und deren Verwendung angezeigt.

Hinweis: Durch *encrypt.exe* wird keine Sicherheitskopie der zu verschlüsselnden Datei angelegt. Wir empfehlen daher dringend, vor Beginn der Verschlüsselung alle Dateien zu sichern.

Teil III:

Weiterführende Konzepte

13. Anlegen von Features

Features sind Funktionen, die für die einzelnen Shoptypen wahlfrei zur Verfügung gestellt oder in Featurepacks aufgenommen werden können. Das Anlegen und Aktivieren der einzelnen Features ist Aufgabe des Business-Administrators.

Dazu müssen diese Funktionen im Quelltext und in der Datenbank entsprechend definiert und verarbeitet werden. Dabei gehen Sie wie folgt vor:

1. Definieren Sie das Feature. Legen Sie dazu in der entsprechenden Cartridge im Verzeichnis */Database/XML* die Datei *Features*.xml* an bzw. erweitern Sie eine vorhandene Datei.
2. Integrieren Sie entsprechend Ihren Anforderungen die Prüfung des Features in den Perl-Code oder in den Template-Quelltext.
3. Führen Sie *nmake install* für Ihre Cartridge aus. Dadurch wird das Feature in die Datenbank importiert.
4. Führen Sie *nmake register* für Ihre Cartridge aus. Dadurch wird das Feature in der Administrationsdatenbank des Business-Administrators bekannt gemacht und er kann dieses Feature für Shops aktivieren oder deaktivieren

Folgendes Beispiel soll das Vorgehen näher erläutern:

Jeder Händler kann in seinem Shop eine bestimmte Anzahl von Produkten bearbeiten. Diese Anzahl ist als Feature angelegt und kann somit für einzelne Shoptypen unterschiedlich festgelegt werden.

Das Feature ist in der Datei

```
%EPAGES_CARTRIDGES%/DE_EPAGES/Product/Database/XML/Features.xml
```

definiert, siehe *Abbildung 22*.

```
<Feature Alias="Products" MaxValue="100000" Cartridge="DE_EPAGES::Product" delete="1" Position="30">
  <AttributeValue Name="Name" Language="en" Value="Products" />
  <AttributeValue Name="Name" Language="de" Value="Produkte" />
  <AttributeValue Name="Description" Language="en" Value="Number of Products" />
  <AttributeValue Name="Description" Language="de" Value="Anzahl von Produkten" />
</Feature>
```

Abbildung 22: Definition des Features in der XML-Datei

Für jedes Feature wird ein *MaxValue* angegeben. Wenn Sie *MaxValue="1"* definieren, bedeutet dies, dass dieses Feature aktiviert (*1*) oder deaktiviert (*0*) werden kann. Wird ein Wert größer als 1 angeben, steht dieses Feature so oft wie angegeben zur Verfügung. So wird die Anzahl der Produkte auch als Feature definiert, mit einer Begrenzung, wie viele Produkte im Shop angelegt werden dürfen. So bedeutet *MaxValue="100000"*, dass der Nutzer 100000 Produkte anlegen kann.

Im Template für die MBO-Anzeige wird abgefragt, ob das Feature gesetzt ist bzw. es wird auf den maximalen Wert getestet:

```
<tr>
  <td class="Image">
    #IF(NOT #Shop.FeatureMaxValue.Products)
    <span class="disabled"><img src="#StoreRoot/BO/icons/mbo_manager_ico_l_products_disabled.gif" alt="{Products}" title="{Products}" /></sp
    #ELSE
    <a href="?ViewAction=MBO-ViewProducts&amp;ObjectID=#Shop.ProductFolder.ID"><img src="#StoreRoot/BO/icons/mbo_manager_ico_l_products.gif"
    #ENDIF
  </td>
  <td>
    <h3>
    #IF(NOT #Shop.FeatureMaxValue.Products)
    <span class="disabled">{Products}</span>
    #ELSE
    <a href="?ViewAction=MBO-ViewProducts&amp;ObjectID=#Shop.ProductFolder.ID">{Products}</a>
    #ENDIF
```

Abbildung 23: Feature-Prüfung im Template

Je nachdem, wie der Business-Administrator den Wert für das Feature gesetzt hat, können Produkte angelegt werden. Wie der Business-Administrator die Werte für Features bearbeiten kann, sehen Sie im folgenden Bild:

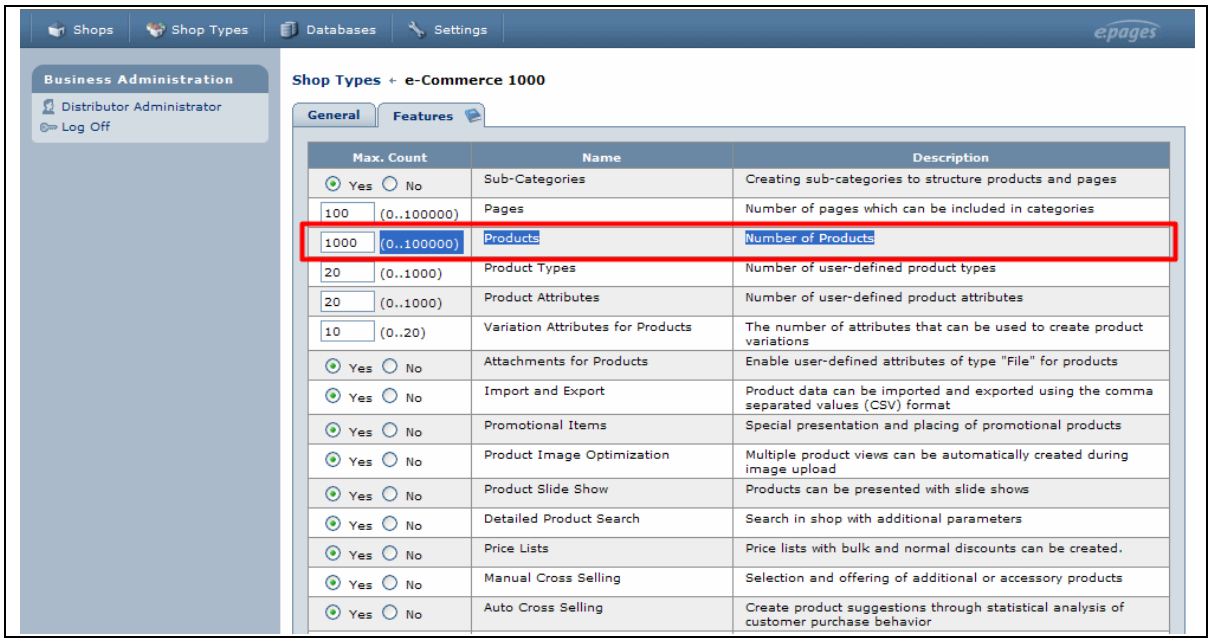

Abbildung 24: Aktivierung der Features durch den Business-Administrator

Hier wird auch noch einmal der Unterschied deutlich zwischen Features mit *MaxValue*="*1*" oder größer *1*. Für gleich *1* gibt es nur die Option *ja* oder *nein*, für größer *1* kann der Business-Administrator einen beliebigen Wert einsetzen. Das Maximum ist in der XML-Datei festgesetzt.

In einem Perl-Modul würde die Abfrage wie folgt aussehen:

```
...
   my $Feature = LoadObjectByPath('/Features/Product');
    if ($Shop->featureMaxValue( $Feature)) { ... } else { ... } ;
...
```

Codebeispiel 49: Abfrage von Features im Perl-Quelltext

Die Prüfung in Templates wird zur Aktivierung bzw. Deaktivierung von Feature-Funktionen wie Links und/oder Schaltflächen verwendet. Die Prüfung im Perl erlaubt die entsprechende Fehlerbehandlung, wenn die gesetzten Grenzwerte überschritten werden.

14. Formhandling

Die in Templates einzugebenden Daten müssen bestimmten Spezifikationen bezüglich verwendbarer Zeichen und regionaler Formate entsprechen und können durch Grenzwerte eingeschränkt sein. Die Prüfung der Eingabewerte erfolgt z. B. bei der Aktion *Speichern*, fehlerhafte Daten dürfen nicht gespeichert werden.

Das Formhandling erleichtert und vereinheitlicht die Darstellung und Verarbeitung von Eingabefeldern in Templates. Die Dateneingabe wird strukturiert und mit einer einheitlichen Fehlerbehandlung versehen. Dafür stehen dem Entwickler so genannte *Forms* zur Verfügung. In diesen Forms können einzelne Eingabefelder oder Feldgruppen festgelegt werden, für die durch Festlegung von u. a. Typ und Grenzwerten die weitere Verarbeitung der Inhalte bestimmt wird. Solche Felder werden als *FormFields* bezeichnet.

Das Formhandling bietet die Möglichkeit, die Verarbeitung und Fehlerbehandlung für die Formfelder zentral zu definieren und einheitlich in jedem Formular auszuführen.

Folgende Gestaltungsrichtlinien sind damit umgesetzt und sollten eingehalten werden:

- Pflichtfelder sind mit einem * gekennzeichnet
- Bei Eingabefehlern wird das komplette Formular wieder mit den eingegebenen Werten angezeigt, die Werte werden erst gespeichert, wenn alle Eingabewerte korrekt sind.
- Enthält das Formular mehrere Eingabefehler, werden alle fehlerhaften Felder gleichzeitig hervorgehoben angezeigt, damit alle mit einem mal korrigiert werden können.

Weiter Möglichkeiten sind:

- Der Focus kann auf das erste Fehlerfeld gesetzt werden.
- Es kann in den Fehlerfeldern ein Beispiel für eine korrekte Eingabe angezeigt werden.

Anzeige, Prüfung und daraus resultierende weitere Verarbeitung der Eingabewerte basieren auf zugeordnetem Typ, Pflichtfeld-Definition und Wertebereich. Sie haben zwei Möglichkeiten, diese Parameter für Attribute zu definieren, in den *Attributes*.xml* und *Forms*.xml*.

14.1 Fehlerbehandlung für Objektattribute

Eindimensionale Objektattribute sind einfache Attribute wie z. B. Gewicht oder Steuerklasse. Sie werden in der Datei *Attributes*.xml* definiert. Diese Attribute werden bei der Templateverarbeitung auf Basis ihrer Typdefinition automatisch geprüft.

Die Prüfung erfolgt durch die Ausführung der Validierungsfunktion *attributeValues* im "Standard-Save" *SUPER::Save($Servlet)* aus *DE_EPAGES::Presentation::UI::Object* .

14.2 Fehlerbehandlung für frei definierbare Formulare

In den Formularen können Sie auch mehrdimensionale Attribute verwenden. Die Behandlung von mehrdimensionalen Attributen, wie Preise oder Subprodukte, muss extra definiert werden. In einem solchen Fall steht zwar der Typ des Attributes an sich fest, aber da eine variable Anzahl von Werten angezeigt wird, muss diese variable Anzeige und Behandlung festgelegt werden. Gleiches gilt für Parameter, die nicht direkt zu diesem Objekt gehören.

Für diese Fälle steht Ihnen die Datei *Forms*.xml* zur Verfügung, die Sie für jede Cartridge bei Bedarf anlegen können.

Folgende Typen können Sie in der FormField-Definition verwenden:

Tabelle 17: Datentyp-Definitionen in Forms

Typ	Beschreibung	Festlegung für MinValue bzw. MaxValue
string	Textfelder	Maximallänge des Texts. Ist das Feld als Pflichtfeld markiert, dann ist die minimale Länge 1.
integer	ganze Zahl, unformatiert	minimaler und/oder maximaler Wert
float	Gleitkommazahl, unformatiert	minimaler und/oder maximaler Wert
reg_date	Datum im regionalen Format	nicht anwendbar
reg_time	Zeit im regionalen Format	nicht anwendbar
reg_datetime	Datum und Zeit im regionalen Format	nicht anwendbar
reg_money	Währung im regionales Format mit Währungsangabe	minimaler und/oder maximaler Wert
reg_integer	ganze Zahl (regionales Format)	minimaler und/oder maximaler Wert
reg_float	Gleitkommazahl im regionalen Format	minimaler und/oder maximaler Wert
checkbox	logischer Wert (wahr oder falsch)	nicht anwendbar
file	geladene Datei	minimale oder maximale Größe in Bytes
email_address	Text im E-Mail-Adressenformat	nicht verwendet
ip_address	nur im Format IPv4 möglich	nicht verwendet

Für jeden dieser Typen ist eine entsprechende Eingabe-Fehlerbehandlung implementiert.

Für das Template, in dem Sie die FormFields verwenden, müssen Sie entsprechende Aktionen für das Speichern des Formulars definieren und implementieren. In der Funktion müssen dem Action-Handler die entsprechenden Forms übergeben werden.

Bei Ausführung werden die FormFields dann automatisch den typ-spezifischen Fehlertests unterzogen und im Fall einer oder mehrerer Eingabefehler werden die entsprechenden Fehler-TLE gefüllt. Siehe auch *Fehler-TLE*, *Seite 71*. Das Template wird erneut angezeigt. Durch Auswertung der Fehler-TLE-Inhalte im Template werden die entsprechenden Fehlermeldungen eingeblendet.

Wenn Sie das Formhandling in eigenen Templates benutzen wollen, müssen Sie prinzipiell wie folgt vorgehen:

1. Identifikation von FormFields im Template
2. Definition der FormFields in der Forms*.xml der Cartridge
3. Implementierung der Behandlung der Feldwerte beim Speichern des Formulars. Eventuell Definition eigener Fehlerbehandlungsroutinen für Werte, die nicht den Standardtypen entsprechen
4. Auswertung der Fehler-TLE im Template

Ein Beispiel zum Formhandling wird im Training in der *Polling-Cartridge* implementiert.

14.2.1 Definition von FormFields

Beim Entwurf des Templates müssen Sie für Eingabefelder zuerst feststellen, welcher Fehlerbehandlungs-Mechanismus für die einzelnen Felder in Frage kommt. Wie im Vorausgegangenen erwähnt, werden die Eingabewerte für eindimensionale Objektattribute standardmäßig geprüft.

Für Sie bleibt die Aufgabe, sich um die mehrdimensionalen und nicht zum Objekt gehörenden Attributwerte zu kümmern. Für diese legen Sie eine Datei *Forms*.xml* an. Für den Dateinamen gilt die Namenskonvention für XML-Import-Dateien, siehe *Import-Dateien*, *Seite 119*. Eine einfache FormField-Definition sehen Sie in *Codebeispiel 50*:

```
<?xml version="1.0" encoding="UTF-8"?>
<epages Cartridge="DE_EPAGES::Shop">
  <Class reference="1" Path="/Classes/Shop">
    <Form Name="SetPolling" delete="1">
      <FormField Name="PollingA" Type="integer" Mandatory="1" MinValue="1"
       MaxValue="10"/>
      <FormField Name="PollingB" Type="integer" Mandatory="1" />
      <FormField Name="PollingC" Type="integer" Mandatory="1" />
    </Form>
  </Class>
</epages>
```

Codebeispiel 50: Beispiel für FormField-Definition

Im Beispiel erkennen Sie, dass Forms klassengebunden sind. Weiterhin bekommt jedes Form einen eindeutigen Namen. So ist es auch möglich, für eine Klasse unterschiedliche Forms für die Dateneingabe zu definieren.

Jedes Feld innerhalb eines Forms muss einen eindeutigen Namen haben. Für jedes Feld wird ein Typ festgelegt. Für die Pflichtfeld-Definition *Mandatory* sind die Werte *0* oder *1* möglich. Je nach Typ, siehe *Tabelle 17*, kann eine Bereichsbegrenzung angegeben werden.

Die Definition von Forms für mehrdimensionale Attribute ist etwas komplexer. Der Grund dafür ist, dass beim Anlegen des Templates nicht feststeht, wie viele Werte eines solchen Attributes eingegeben werden müssen. Ein Beispiel dafür ist der Produktpreis. In Abhängigkeit von den eingestellten Währungen werden entsprechend viele Eingabefelder für den Preis angezeigt. In Forms wird in einem solchen Fall eine Loop-Definition verwendet, siehe *Codebeispiel 51* oder auch *Codebeispiel 55*:

```
<?xml version="1.0" encoding="UTF-8"?>
<epages Cartridge="DE_EPAGES::Product">
  <Class reference="1" Path="/Classes/Product">
...
   <Form Name="Save" delete="1">
      <FormLoop Name="ProductPrices">
        <FormField Name="CurrencyID" Type="string" Mandatory="1" MinValue="3" />
        <FormField Name="Price" Type="reg_money" />
      </FormLoop>
    </Form>
...
  </Class>
</epages>
```

Codebeispiel 51: Form für mehrdimensionale Felder

Alle betreffenden Felder, im Beispiel *CurrencyID* und *Price*, werden in einer *FormLoop* definiert. Dies ist die Voraussetzung, dass die Fehlerbehandlung für diese Felder so oft ausgeführt wird, wie Eingabefelder im Template angezeigt werden. Der Name der *FormLoop* innerhalb eines Forms muss eindeutig sein. Die Anzahl der Formfields in der FormLoop muss gleich sein der Anzahl der Eingabefelder im Template, für die mehrere Werte erfasst werden können.

Die Forms werden im Verlauf der Cartridge-Installation in der Datenbank registriert. Manuell werden Forms mit folgendem Befehl in die Datenbank importiert:

```
perl %EPAGES_CARTRIDGES%/DE_EPAGES/Presentation/Scripts/import.pl Forms.xml
```

Die Grundlagen für Datei-Import lesen Sie in *Import / Export von Datenbankinhalten*, *Seite 119*.

14.2.2 Verwendung von Forms im Perl-Code

Sie müssen dafür sorgen, dass die Forms im Perl-Code entsprechend verarbeitet werden. Dafür steht Ihnen die Methode *$Servlet->form->form* zur Verfügung. Durch diese Methode werden die FormFields auf Basis

der regionalen Einstellungen nach Typ und Bereichsbeschränkung validiert. Auftretende Fehler werden in eine Fehlerliste gespeichert, auf die über die Fehler-TLE zugegriffen werden kann.

Die Syntax für die Methode im Perl-Code ist:

```
my $<hashreferenz> = $Servlet->form->form($<object>, '<formname>');
```

Dabei muss ‹object› ein Objekt der Klasse sein, für die das Form registriert wurde, siehe *Codebeispiel 50* und *Codebeispiel 51*.

Folgendes Beispiel demonstriert die Anwendung:

```
...
sub SaveMailSettings {

    my $self = shift;
    my ($Servlet) = @_;
    my $Form = $Servlet->form;

    # get the current object from the servlet
    my $System = $Servlet->object();

    # validate input data and get unformatted values
    my $hValues = $Servlet->form->form( $System, 'MailConnection' );

    # save the new values
    $System->set( $hValues );

    return;
}
...
```

Codebeispiel 52: Validierung der Forms in Perl

14.2.3 Validierung bei nicht definierten Datentypen

In Forms können Sie auch Felder definieren, die nicht den Standard-Typen entsprechen. Für solche Form-Fields müssen Sie die Validierung im Perl-Code selbst umsetzen. Dasselbe gilt für Felder, für welche z. B. umfangreichere Prüfungen durchgeführt werden müssen, als die Validierung entsprechend der Typdefinition vorsieht.

Solch eine erweiterte Prüfung sehen Sie in *Codebeispiel 53*.

```
...
sub SaveSettings {

    # validate input data and get unformatted values
    # third parameter = 1 means skipExecuteFormError

    $FormValues = $Form->form($PaymentMethod, 'Settings', 1);

    #--- additional check: check the merchant secret
    my $Secret = $FormValues->{'Secret'};
    if (defined($Secret) && $Secret !~ /^[A-Fa-f0-9]+$/) {
        $Form->addFormError({'Name' => 'Secret', 'Reason' => 'HEXDIGITSONLY',
                             'ViewAction' => 'MBO-ViewSettings'});
    }

    $Form->executeFormError();

    #--- standard save
    $self->SUPER::Save($Servlet);
}
...
```

Codebeispiel 53: Erweiterte Prüfung für Forms

Zuerst wird das Form eingelesen. Vor Ausführung der speziellen Prüfung wird die allgemeine Fehlerprüfung für das Form ausgeführt. Durch den letzten Übergabeparameter *1* wird festgelegt, dass trotz eventuell auftretender Fehler die Abarbeitung nicht abgebrochen wird.

Im weiteren Ablauf wird für das Feld *Secret* vom Typ *string* eine zusätzliche Prüfung durchgeführt. Es muss geprüft werden, ob die verwendeten Zeichen in einem bestimmten Bereich liegen. Im Fehlerfall wird ein entsprechender Eintrag mit Begründung in die Fehlerliste eingetragen.

Danach erfolgt die allgemeine Fehlerbehandlung.

14.2.4 Fehlerbehandlung im Template

Alle während der Validierung festgestellten Fehler werden in eine Fehlerliste, die *FormErrorLoop*, eingetragen. Diese *FormErrorLoop* ist Voraussetzung dafür, dass mit Hilfe der Fehler-TLE die aufgetretenen Fehler im Template ausgewertet und entsprechend angezeigt werden können. Welche Fehler-TLE dafür zur Verfügung stehen, lesen Sie in *Fehler-TLE*, *Seite 71*.

Prinzipiell kann man die Fehlerbehandlung in verschiedene Fälle einteilen:

- Fehleranzeige für einzelne Felder
- Fehleranzeige in Listen
- Anzeige aller Fehler des Templates in einer Liste

14.2.4.1 Fehleranzeige für einzelne Felder

Hier wird für einzelne Felder unter Angabe des Feldnamens gezielt abgefragt, ob ein Fehler aufgetreten ist. Dadurch kann auch für jedes Feld bei Bedarf eine spezielle Fehlerbehandlung im Template erfolgen.

```
  #IF(#FormError)
    <div class="DialogMessage" id="MessageWarning">
      <h3>{InputError}</h3>
      {PleaseCorrectErrors}
    </div>
  #ENDIF
  #WITH_ERROR(#FormError)
    <table>
      <tr #IF(#FormError_SMTPServer)class="DialogError" #ENDIF>
        <td class="InputLabelling">{MailServer}</td>
        <td>
          <input type="text" name="SMTPServer" size="20"
          value="#SMTPServer"/>
        </td>
      </tr>
      <tr #IF(#FormError_SMTPPort)class="DialogError" #ENDIF>
        <td class="InputLabelling">{ServerPort}</td>
        <td><input type="text" name="SMTPPort" size="20"
            value="#SMTPPort[integer]"/></td>
      </tr>
    </table>
  #ENDWITH_ERROR
```

Codebeispiel 54: Fehlerbehandlung im Template für einzelne Felder

Im Beispiel wird zuerst generell nach dem Auftreten von Fehlern gefragt. Wenn ja, dann wird ein Warnhinweis gezeigt.

Danach wird per *#FormError_‹feldname›* für die Felder *SMTPServer* und *SMTPPort* geprüft, ob Fehler eingetragen sind. Ist dies der Fall, wird die Anzeige für diese Felder geändert.

14.2.4.2 Fehleranzeige in Listen

Die Besonderheit von Listen ist, dass mehrere Werte für einen Eingabefeldnamen erfasst werden. Um diese Felder prüfen zu können, werden in Forms *Loops* definiert, siehe *Codebeispiel 51*. Für die Anzeige im Template sind analoge Strukturen erforderlich. Auch hier müssen die Fehler mit Hilfe einer Loop ausgewertet und angezeigt werden, damit jeder Fehler in der Liste angezeigt werden kann.

```
#LOOP(#ListPrices)
  #WITH_ERROR(#FormError)
    <tr>
      <td #IF(#FormError_Price OR #FormError_CurrencyID)
          class="DialogError"#ENDIF>
        <input type="text" name="Price" value="#Price[money]" class="Price"/>#
      </td>
      <td>
        #Currency.Symbol
        <input type="hidden" name="CurrencyID" value="#CurrencyID" />
      </td>
    </tr>
  #ENDWITH_ERROR
#ENDLOOP
```

Codebeispiel 55: Fehlerbehandlung im Template für Listen

Im Beispiel werden in Abhängigkeit der eingestellten Währungen mehrere Preiseingabefelder für ein Produkt angezeigt. Werden bei einem oder mehreren Preisen Eingabefehler festgestellt, sind die entsprechenden Felder bei der Wiederanzeige des Templates markiert.

14.2.4.3 Anzeige aller Fehler des Templates in einer Liste

Eine weitere Möglichkeit der Fehleranzeige ist die Auflistung aller Fehler in einer separaten Liste. In diesem Fall werden keine Eingabefelder markiert, sondern ein Gesamtüberblick über die aufgetretenen Fehler mit Beschreibung. Ein Anwendungsbeispiel ist die Anzeige von Fehlern beim Daten-Export.

Für diese Fehleranzeige wird die Fehler-TLE *#FormErrors.‹fehlerTLEName›* verwendet:

```
#IF(#FormError)
  <ul>
  #LOOP(#FormErrors.Errors)
   <br />Error: #Reason, #Value, #Name (#Index).
  #ENDLOOP
  #LOOP(#FormErrors.Reasons)
    <li>Reason=#Reason
  #ENDLOOP
  </ul>
#ENDIF
```

Codebeispiel 56: Anzeige einer Fehlerliste

Welche Fehlerinformationen Sie auslesen und anzeigen können, lesen Sie in der API-Dokumentation unter *Presentation/API/Form.pm*.

15. Web Services

Web Services bieten die Möglichkeit, plattformübergreifend Daten auszutauschen. Zum Übertragen werden Protokolle wie HTTP, SMTP oder FTP verwendet. Die Daten sind nach SOAP-Standard strukturiert und werden in einem XML-Dokument übertragen.

ePages 5 stellt neben eigenen Web Services auch ein Framework zur Verfügung, welches die Erstellung und Einbindung von Web Services unterstützt.

Dies wird im Nachfolgenden beschreiben. Voraussetzungen sind grundlegende Kenntnisse zu Web Services, XML, SOAP-Lite, WSDL und Perl-Programmierung.

Zu diesem Thema bietet ePages auch ein spezielles Training an. Das Training untersetzt dieses Kapitel mit Beispielen, weiterführenden Informationen und Erläuterungen. Weiterhin werden externe Tools zur Erzeugung und Validierung von Web Services und Clients vorgestellt.

15.1 ePages–Web Services und Framework

ePages benutzt intern Web Services, um Daten zwischen der zentralen Administrations-Datenbank und den Shop-Datenbanken zu überragen.

Für den Datenaustausch mit externen Systemen stehen folgende Web Services zur Verfügung:

Tabelle 18: Web Services in ePages

Datenbereich	Web Service
Produktdaten und Katalogstrukturen	ProductService CatalogService AssignmentService
Preislisten	PriceListService PriceListAssignmentService
Kundendaten	CustomerService
Nutzerdaten	UserService
Bestelldaten	OrderService
Rechnung- und Lieferscheindaten	OrderDocumentService
Anlegen von Shops/Shopmanagement	ShopConfigService

Mit Hilfe der Diagnostics-Cartridge können Sie sich alle in ePages 5 registrierten Web Services anzeigen lassen.

Basis für die Implementation der Web Services ist die Cartridge *WebService*. Sie stellt einen Basis-Web Service zur Verfügung und bindet das Modul *SOAP::Lite* mit ein. Weiterhin ist darin ein Basis-Permission-Check implementiert.

Die Beschreibung zu jedem der in *Tabelle 18* genannten Web Services ist in der zugehörigen WSDL-Datei enthalten. Diese WSDL-Dateien finden Sie in den Verzeichnissen:

```
%EPAGES_WEBROOT%/<store>/WSDL
```

bzw.

```
%EPAGES_WEBROOT%/Site/WSDL
```

Komplexe Datentypen, die in einem Web Service verwendet werden, sind in der zugehörigen XSD-Datei definiert.

Die Web Service-Requests werden im System wie folgt abgearbeitet:

- Client-Request wird entgegengenommen
- Prüfung, ob der Web Service in der Datenbank registriert ist
- Prüfung, ob die erforderliche Berechtigung zur Ausführung der Funktion vorliegt
- Übergabe des Requests an das SOAP::Lite-Modul, welches das übergebene XML-Dokument in entsprechende Perl-Objekte umwandelt
- Ausführung der entsprechenden Funktionen in Perl-Modulen und Rückgabe der Ergebnisse
- Umwandlung des Ergebnisses in ein XML-Dokument durch das SOAP::Lite-Modul
- Übertragung der Response mit dem XML-Dokument an den Client

Hinweis: Die Voraussetzung für das Ausführen von Web Services ist die Aktivierung des Features *Program Interface for Web Services* im BBO. Die Beschreibung dafür finden Sie im *Handbuch für Business-Administratoren*.

In der Web Service-Cartridge wurde ein Logging eingeführt, um eventuelle Soap-Fehler anzuzeigen. Dieses Logging kann in der Datei

```
%EPAGES_CONFIG%/log4perl.conf
```

aktiviert werden. Entfernen Sie in der Sektion *SOAP server* das Kommentarzeichen für die entsprechende Zeile:

```
;
; SOAP server
;
;log4perl.category.DE_EPAGES::WebService::API::SoapServer::make_fault = DEBUG
;log4perl.category.DE_EPAGES::WebService::API::SoapServer::find_target = DEBUG
;log4perl.category.DE_EPAGES::WebService::API::WebService::BaseService::CheckPermi
 ssion = DEBUG
```

Mehr zur *log4perl.conf* lesen Sie im *Installationshandbuch für Windows*.

15.2 ePages-Web Service generieren

Legen Sie neue Web Services in einer eigenen Cartridge an. Dafür gibt es in der Cartridge-Struktur folgende spezielle Elemente:

- Das Unterverzeichnis */API/WebService* – hier werden die Perl-Module für den Web Service gespeichert
- Das Unterverzeichnis /Data/Public/WSDL – hier werden die WSDL-Dateien für Web Services angelegt. Diese Deskriptor-Dateien sind optional. Sie werden bei der Installation in das Verzeichnis *%EPAGES-WEBROOT%/<store>/WSDL* übertragen.
- Die Dateien *Actions*.xml* und *Permissions*.xml* müssen Sie um die Web Service-spezifischen Einträge erweitern. Dadurch werden die Web Services in der Datenbank registriert und die entsprechenden Berechtigungen vergeben.

Nach dem Anlegen der Cartridge müssen Sie folgende Schritte ausführen, um einen Web Service zu erstellen:

1. Registrieren
2. Autorisieren
3. Implementieren

15.2.1 Registrieren

Jeder Webservice muss in der Datenbank registriert werden. Dies geschieht über die Datei *Actions*.xml* der Cartridge. Legen Sie diese Datei an oder erweitern Sie eine eventuell bestehende. Im Lauf der Installation wird die Datei in die Datenbank importiert und die Webservices und ihre Methoden werden in den entsprechenden Tabellen registriert.

In *Codebeispiel 57* sehen Sie den prinzipiellen Aufbau eines Eintrages zur Registrierung eines Web Services.

```
<?xml version="1.0" encoding="UTF-8"?>
<epages>
  <Object Alias="WebServices">
    <WebService Alias="MathService"
                URI="urn://epages.de/WebService/MathService/2005/05" delete="1">
      <Object Alias="Methods">
        <WebServiceMethod Alias="Add"
          Package="Training::MyWebService::API::WebService::MathService" />
      </Object>
    </WebService>
  </Object>
</epages>
```

Codebeispiel 57: Registrieren eines Web Services

Der Web Service wird im Objektordner mit dem Alias *WebServices* angemeldet. Für den Web Service selbst müssen im Element *WebService* ein Alias und ein URI angeben werden. Der URI (Unique Resource Identifier) ist ein URL-ähnlicher String zur Identifikation des Web Service. Dabei hat sich durchgesetzt, den Erstellungszeitraum mit anzugeben (2005/05).

Innerhalb *‹WebServices›* müssen Sie die Methoden definieren. Diese müssen grundsätzlich dem Objekt *Methods* zugeordnet werden.

Im Element *WebServiceMethod* wird eine Methode definiert. Unter *Alias* wird der Name der Methode angegeben, *Package* gibt an, wo die Methode in Perl implementiert ist. Jede Methode muss einzeln eingetragen werden.

15.2.2 Autorisierung

Zu jedem Web Service müssen Sie angeben, wer diesen ausführen darf. Das heißt, zu jedem Web Service wird eine Rolle definiert, welche die Berechtigung hat, die Methoden aufzurufen.

Diese Berechtigungen werden in der Datei *Permissions*.xml* festgelegt. In *Codebeispiel 58* sehen Sie einen entsprechenden Eintrag.

```
<?xml version="1.0" encoding="UTF-8"?>
<epages>
  <Class reference="1" Path="/Classes/WebService"/>
    <Role reference="1"   Path="/Classes/Shop/Roles/WebService">
      <WSRoleMethod WebService="MathService" Method="Add" delete="1" />
    </Role>
</epages>
```

Codebeispiel 58: Festlegen der Berechtigung

Zuerst legen Sie fest, für welche Klasse eine Berechtigung vergeben werden soll. Im Element *Role* geben Sie an, welche Rolle die Berechtigung zur Ausführung der Methode bekommt. In *WSRoleMethod* wird konkret die Methode angegeben und zu welchem Web Service diese Methode gehört.

Die Rolle WebService ist automatisch jedem Administrator eines Shops zugeordnet. Das heißt, die Rollenzuweisung in *Codebeispiel 58* erlaubt jedem Shop-Administrator, die Methode *Add* des Web Service *MathService* aufzurufen.

Wollen Sie eine Funktion öffentlich zur Verfügung stellen, d. h. sollte jeder die Funktion per Web Service aufrufen können, sollten Sie:

- als Basis für den Web Service *BaseService* benutzen,
- als Parameter den Shop übergeben,
- eine neue Rolle anlegen, z. B. *PublicWebService*,
- und diese Rolle der Gruppe *Everyone* zuordnen.

15.2.3 Implementation

Die Funktionen selbst werden in Perl-Modulen im Cartridge-Verzeichnis */API/WebService* implementiert.

Grundlage für die Web Service-Implementation ist die Basisklasse *BaseService* aus dem Package *DE_EPAGES::WebService::API::WebService::BaseService*. Diese Klasse stellt u. a. zur Verfügung:

- einen Permission-Check
- das User-Objekt des aktuellen Web Service-Users
- ein Datenmapping zwischen angegebener XSD-Datei und angegebenem URI

Je nachdem, ob der Web Service für den Shop oder für Provider-Aktionen zur Verfügung stehen soll gibt es zwei weitere Sub-Klassen von *BaseService*, die Sie als Basis für Ihre Web Services verwenden können:

- *DE_EPAGES::Shop::API::WebService::BaseShopService*: Diese Klasse benutzt das Shop-Objekt und autorisiert das jeweilige Händler-Login. Diese Klasse ist vorzugsweise für die Umsetzung von Storefront- und Backoffice-Funktionalität zu verwenden.
- *DE_EPAGES::ShopConfiguration::API::WebService::BaseProviderService*: Diese Klasse benutzt das Distributor-Objekt und autorisiert das Distributor-Login.

Codebeispiel 59 zeigt die Einbindung einer der Basis-Klassen:

```
package  Training::MyWebService::API::WebService::MathService;
use base DE_EPAGES::Shop::API::WebService::BaseShopService;
use SOAP::Lite;

sub Add {
    my $self = shift;    #First argument is service object
    my ($s1, $s2) = @_;  #2 doubles expected from client
    my $sum = $s1 + $s2; #WS-implemented code
    return $sum;
}
1;
```

Codebeispiel 59: Web Service-Perl-Modul

15.3 Externe Clients für ePages 5-Web Services

Um die ePages 5-Web Services aufzurufen, müssen für die externen Systeme die entsprechenden Clients bereitgestellt werden.

Für bestimmte Programmiersprachen, z. B. Java und C, kann der Code auf Basis der WSDL-Dateien automatisch generiert werden. Für Perl müssen Sie die Clients manuell erstellen.

Prinzipiell müssen Sie im Client-Modul ein Soap-Objekt erzeugen, dem Sie die URI, also die Bezeichnung des Web Services, und den Zielpunkt (Proxy), die Adresse des entsprechenden Soap-Services, übergeben.

Zusätzlich erwarten ePages-Standard-Web Service-Requests eine Autorisierung mit Angabe von Login und Passwort. Die allgemeine Form für den Proxy-Aufruf ist:

```
http://login:password@epagesserver/epages/Store.soap
```

Die Besonderheit im Bezug auf ePages besteht darin, dass das Login in Form eines Objektpfades übergeben werden muss, z. B.

```
/Shops/DemosShop/Users/admin
```

In dieser Form kann das Login aber nicht als Proxy-Aufruf übergeben werden, da in dem Aufruf Slashes (/) nicht erlaubt sind. Um dieses Problem zu umgehen, werden Anmeldeinformationen über die Credentials ausgelesen. *Codebeispiel 60* zeigt eine mögliche Variante:

```
use SOAP::Lite;

my $NAMESPACE   = 'urn://epages.de/WebService/MathService/2005/05';
my $CREDENTIALS = '/Shops/DemoShop/Users/admin:admin';
my $SERVER      = 'localhost:8080';
my $STORE       = 'Store';

my $PROXY_URL = URI->new( "http://$SERVER/epages5/$STORE.soap" );

$PROXY_URL->userinfo( $CREDENTIALS );

my $soap = SOAP::Lite->uri( $NAMESPACE )->proxy( $PROXY_URL->as_string );

my $sum = $soap->Add( 175.6, 3.2 )->result;

print "Add = $sum\n";
```

Codebeispiel 60: Externer Client

Mit Hilfe der Funktion *userinfo* und dem Parameter *CREDENTIALS* werden die notwendigen Anmeldeinformationen ausgelesen und an *PROXY_URL* übergeben. Die Funktion as_string generiert den korrekten Proxy-Aufruf für das Beispiel in der Form:

```
http://admin:admin@localhost:8080/epages5/Store.soap
```

Nachdem das Soap-Objekt erzeugt wurde, kann darüber eine Methode der unter *NAMESPACE* genannten Web Services ausgeführt werden.

Aus dem Beispiel ist zu erkennen, dass für die Autorisierung Login und Kennwort eines Shop-Administrators verwendet werden. Daraus geht hervor, dass, sobald dieser Shop-Administrator seine Anmeldedaten ändert, auch die Web Service-Aufrufe angepasst werden müssen.

Hier bietet sich an, in der Benutzerverwaltung des Shops für die Web Services einen eigenen Administrator anzulegen, dessen Daten auch nur in Absprache mit den Web Service-Verantwortlichen geändert werden sollten.

Sollten Sie als Web Server den MS IIS verwenden, beachten Sie, dass für die erfolgreiche Übergabe von Login und Passwort in einer URL die Abschaltung der integrierten Windows-Authentifikation für das virtuelle epages-Directory notwendig ist. Anderenfalls können Login und Passwort nicht zum Service übertragen werden.

Diese Einstellungen ändern Sie in den Eigenschaften des MS IIS, siehe *Abbildung 25*.

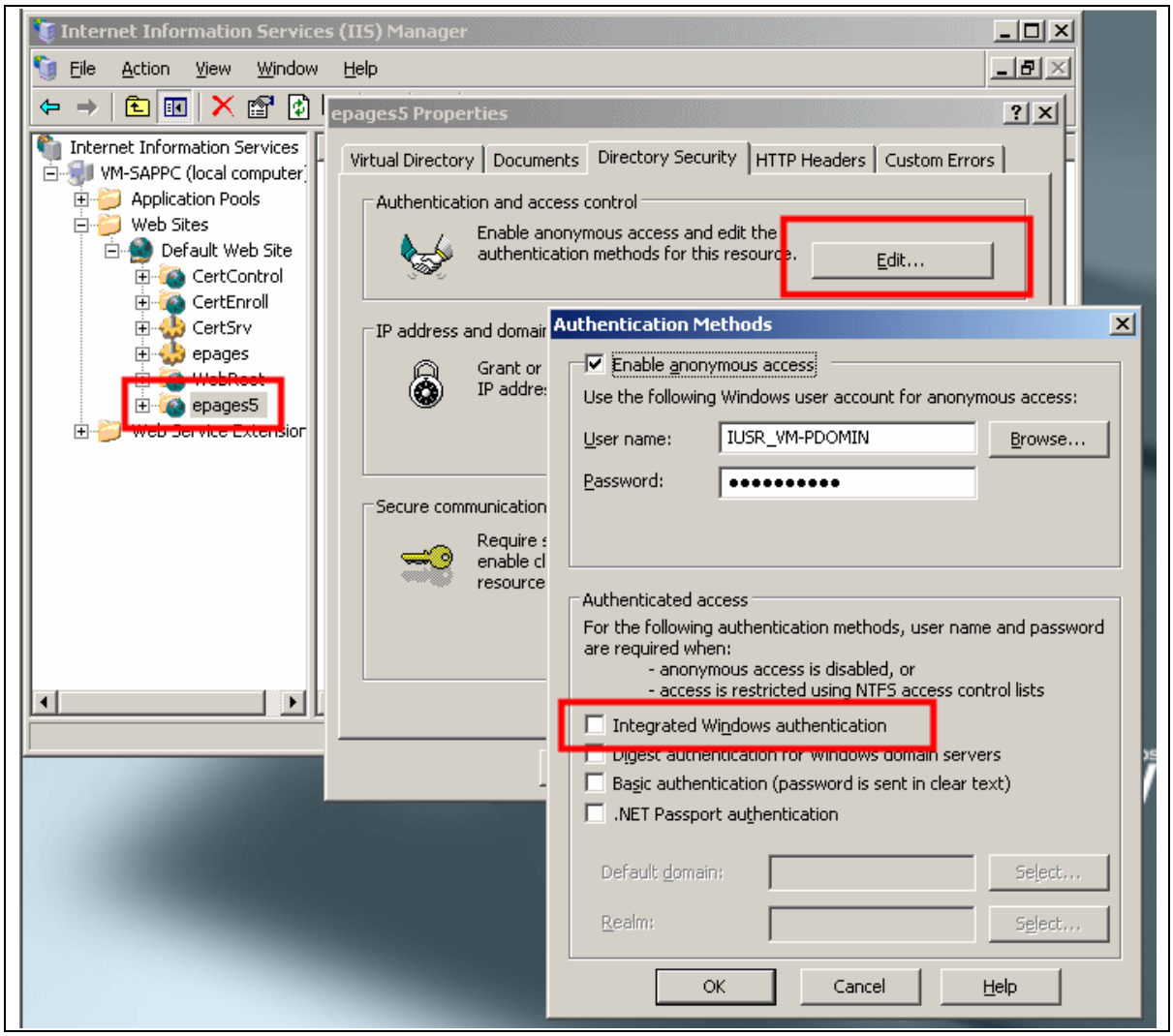

Abbildung 25: Abschalten der Windows-Authentifikation

Falls der Web Service durch eine WSDL-Datei beschrieben wird, können Sie die Funktionen aufrufen, indem Sie ein Service-Objekt erzeugen und diesem die URL des Web Services übergeben, siehe *Codebeispiel 61*.

```
use SOAP::Lite;

my $service = SOAP::Lite
 ->service('http://epagesserver/Store/WSDL/MathService.wsdl');

my $response = $service-> Add( 175.6, 3.2 );

print $response;
```

Codebeispiel 61: Externer Client mit WSDL-URL

Dann rufen Sie die gewünschte Methode über das Service-Objekt auf.

15.4 ePages–Web Service-Client implementieren

Externe Web Services können Sie aus Storefront- und Backoffice-Templates heraus aufrufen. Dazu müssen Sie eine eigene TLE-Funktion erstellen. Über diese TLE-Funktion wird der Web Service aufgerufen. Die Ergebnisse werden im Rückgabewert der Funktion übergeben und können im Template angezeigt werden. Lesen Sie hierzu *Erstellen einer TLE-Funktion*, *Seite 77*.

Eine Möglichkeit der Web Service-Integration in ein Template sehen Sie in *Codebeispiel 62*.

```
…
#IF(#INPUT.StockSymbol)
  #SET("StockQuote",
     #FUNCTION("WS_STOCKQUOTE", #INPUT.StockSymbol))
#ENDIF
      <div class="ClearBoth"></div>
    </div>
#IF(#StockQuote)
    <!-- Explanatory text -->
      <b>{Results}</b> #StockQuote<br/>
    <!-- /Explanatory text -->
#ENDIF
…
```

Codebeispiel 62: Aufruf eines Web Service aus dem Template heraus

Im Beispiel wird über die Funktion *WS_STOCKQUOTE* der entsprechende externe Web Service aufgerufen. Dabei wird ein vom Web Service benötigter Parameter übergeben. Die Web Service-Antwort wird an die Variable *StockQuote* übergeben, die dann im Template angezeigt wird.

Die Funktion zum Aufruf des Web Services sehen Sie in *Codebeispiel 63*:

```
…
sub WS_STOCKQUOTE {
    use SOAP::Lite;
    my $self = shift;
    my ($Processor, $aParams) = @_;

    my ($StockSymbol) = @$aParams;
    return unless $StockSymbol =~ /[a-zA-Z]+/;

    my $soap = SOAP::Lite
    ->uri('urn:xmethods-delayed-quotes')
    ->proxy('http://services.xmethods.net/soap');

    my $quote = $soap->getQuote( $StockSymbol )->result;

    my $output = "$StockSymbol = $quote \n";


return $output;
}
…
```

Codebeispiel 63: Aufruf des externen Web Service

16. Hooks

Hooks sind definierte Stellen im Perl-Code einer Cartridge, von denen aus andere, externe Funktionen aufgerufen werden. Damit können Sie die Funktionalität erweitern, ohne die Cartridge zu ändern. Diese Zusatzfunktionen müssen für den entsprechenden Hook registriert werden.

Während des Prozesses wird an jedem Hook geprüft, ob eine zusätzliche Funktion für diesen Hook angemeldet ist. Nach Ausführung der Funktion erfolgt ein Rücksprung zum "normalen" Prozess.

Die Basisfunktion für die Bereitstellung von Hooks ist die Funktion *TriggerHook()* der Cartridge *Trigger*.

Durch den Codegenerator werden automatisch standardisierte Triggerpunkte erzeugt. Jede ePages-Klasse stellt Hooks bereit, wenn Objekte oder Einträge in Tabellen erzeugt, aktualisiert und gelöscht werden:

Tabelle 19: Standard-Hooks

Objekt Hook	Tabellen-Hook	Aufruf
OBJ_Insert$Class	API_Insert$Table	Nachdem ein neues Objekt vom Typ $Class bzw. ein neuer Tabelleneintrag in Tabelle $Table erzeugt wurde
OBJ_Delete$Class	API_Delete$Table	Bevor ein Objekt vom Typ $Class bzw. ein Tabelleneintrag in Tabelle $Table gelöscht wird
OBJ_BeforeUpdate$Class	API_BeforeUpdate$Table	Bevor ein Attribut für ein Objekt vom Typ $Class bzw. bevor ein Tabelleneintrag in Tabelle $Table geändert wird
OBJ_AfterUpdate$Class	API_AfterUpdate$Table	Nachdem Attribute für ein Objekt vom Typ $Class bzw. nachdem ein Tabelleneintrag in Tabelle $Table geändert wurde

Aus den Hooknamen ist folgende Bildungsvorschrift erkennbar:

1. Präfix *OBJ_* oder *API_* zeigt an, ob die Funktion auf ein Objekt oder eine Tabelle angewendet wird
2. Name der Aktion, z. B. *AfterUpdate*
3. Name der Klasse bzw. Tabelle

Eine Übersicht aller verfügbaren Hooks können Sie sich mit Hilfe der Diagnostics-Cartridge, Punkt **All Hooks**, anzeigen lassen.

16.1 Bereitstellen eines Hooks

Um einen Hook in einer Cartridge bereitzustellen, muss im Code der entsprechende Funktionsaufruf implementiert und in der Datenbank registriert sein. Dazu sind folgende Schritte erforderlich:

1. Einfügen der *TriggerHook*-Funktion in den Quelltext:

```
package COMPANY::Cartridge::API::Example;

use strict;
use DE_EPAGES::Trigger::API::Trigger qw ( TriggerHook );

sub Example {
   TriggerHook('OBJ_MyHookAction_MyHookObject', { 'Param1' => 'Value1' });
}

1;
```

Codebeispiel 64: Beispiel zum Einfügen der Trigger-Funktion für einen Hook

Bei der Vergabe des Hooknamens empfehlen wir, sich an die o. g. Bildungsvorschrift zu halten.

Als Parameter wird ein Hash übergeben. Dieser enthält die Parameter, die in der Funktion verwendet werden, Hookdetails, wie *HookCount* oder *HookName* und eine Referenz auf das Objekt oder den PrimaryKey der Tabelle.

2. Eintragen dieser Hookfunktion in die entsprechende Datei *Hooks*.xml* der Cartridge, siehe dazu auch *Hooks*, *Seite 115*.

```
<?xml version="1.0" encoding="UTF-8"?>
<epages>
  <Hook Name="OBJ_MyHookAction_MyHookObject" delete="1" />
</epages>
```

Codebeispiel 65: Beispiel für Registrieren des Hooks in der XML-Datei

3. Importieren der Datei *Hooks*.xml* unter Verwendung des *Trigger-Import-Scripts*, um die Funktion in der Datenbank zu registrieren:

```
%EPAGES_PERL%\bin\perl %EPAGES_CARTRIDGES%\DE_EPAGES\Trigger\Scripts\import.pl
-storename Store Hooks.xml
```

Kontrollieren Sie mit Hilfe der Diagnostics-Cartridge, ob der Hook korrekt in die Datenbank eingetragen wurde.

16.2 Funktionserweiterung per Hook

Um existierende Hooks für Funktionserweiterungen zu nutzen, müssen Sie in drei Schritten vorgehen:

1. Implementation der Funktionserweiterung in einem externen Perl-Modul. Für die Definition der Funktion gibt es folgende Namenskonvention:

   ```
   package <company>::<cartridge>::Hooks::<class_or_table>;

   sub On<action><class_or_table>{}
   ...
   ```

   Codebeispiel 66: Namenskonvention für Hook-Prozeduren

   Das folgende Beispiel zeigt die Umsetzung dieser Namenskonvention:

```
package COMPANY::Cartridge::Hooks::MyObject;

use strict;

sub OnMyHookProcMyObject {
    my ($hParams) = @_;
    GetLog->debug( "Value1=" . $hParams->{'Value1'} );
}

1;
```

Codebeispiel 67: Beispiel für externe Funktionserweiterung

2. Eintragen der Referenz Funktion - Hook in die Datei *Hooks*.xml*, siehe dazu auch *Hooks*, *Seite 115*:

```
<?xml version="1.0" encoding="UTF-8"?>
<epages>
  <Hook Name="MyHook" reference="1">
   <HookFunction FunctionName="COMPANY::Cartridge::Hooks::
     MyObject::MyHookProc::OnMyHookProcMyObject " OrderNo="1" delete="1"/>
  </Hook>
</epages>
```

Codebeispiel 68: Beispiel für Funktion zu Hook-Referenz

Dabei verwenden Sie die Datei *Hooks*.xml* der Cartridge, in der Sie die Funktionserweiterung implementieren.

Die *OrderNo* ist vom Typ *Integer* und kann wahlfrei vergeben werden.

Hinweis: Sobald einem Hook mehrere Funktionen zugeordnet sind, werden diese in der Reihenfolge entsprechend der OrderNo ausgeführt.

3. Importieren der Datei *Hooks*.xml* unter Verwendung des *Trigger-Import-Scripts*, um die Referenz in der Datenbank zu registrieren:

```
%EPAGES_PERL%\bin\perl %EPAGES_CARTRIGES%\DE_EPAGES\Trigger\Scripts\import.pl
-storename Store Hooks.xml
```

Kontrollieren Sie mit Hilfe der Diagnostics-Cartridge, ob die Funktion dem Hook korrekt zugeordnet wurde. Klicken Sie dazu in der Liste der Hooks auf den Hooknamen, um die zugeordneten Funktionen anzuzeigen.

17. Import / Export von Datenbankinhalten

Ein Großteil der Daten wird per XML-Datei bereitgestellt und muss in die Datenbank eingelesen werden. Dies sind z. B. Beschreibungen der Klassenstruktur, Funktions- und Attribut-Definitionen oder Inhalte. Beispiele für die einzelnen Import-Dateien haben Sie bereits in den Anwendungsbeispielen kennen gelernt.

Die Importe werden entweder bei der Installation der Cartridges durchgeführt oder können von der Kommandozeile aus gestartet werden. Je nach Art der zu importierenden Daten müssen unterschiedliche Import-Handler benutzt werden. Wie manuelle Importe ausgeführt werden, lesen Sie in *XML-Import*, *Seite 120*.

17.1 Import-Dateien

Im Nachfolgenden werden die XML-Import-Dateien erläutert, die bei der Cartridge-Entwicklung verwendet und während der Installation automatisch in die Datenbank eingelesen werden.

Diese XML-Dateien unterliegen einer Namenskonvention. Die Dateinamen müssen mit dem Typ-Namen beginnen, können aber wahlfrei erweitert werden. Die Endung muss *.xml* sein. Es können pro Dateityp mehrere Dateien angelegt werden, die sich nur in der wahlfreien Erweiterung unterscheiden dürfen.

So können z. B. für die XML-Datei für PageTypes folgende Namen verwendet werden: *PageTypesMBO.xml* oder *PageTypesSF.xml*. Für *Attributes*.xml* ist beispielsweise erlaubt *AttributesShop.xml*.

Beispiele für die Anwendung dieser Dateien finden Sie in den Beispielen und in den Standard-Cartridges Ihrer ePages-Installation.

Die folgende Tabelle gibt einen Überblick über die XML-Dateien, die Sie in Ihren Cartridges für Importe verwenden können. Die fakultative Namenserweiterung wird durch * (Wildcard) repräsentiert.

Tabelle 20: Import-Dateien in Cartridges

Import-Datei	Bedeutung
Actions*.xml	Definition von ViewActions und ChangeActions, siehe *URL-Aktionen*, *Seite 31*. Die Funktionen, welche über diese Aktionen aufgerufen werden, sind in den Modulen im Cartridge-Verzeichnis */UI* abgelegt. Jede ViewAction kann mit einem Online-Hilfethema verknüpft werden, siehe *Integration der eigenen Online-Hilfe*, *Seite 171*.
Attributes*.xml	Import von Objektattributen Auch diese Datei wird bei Verwendung des PowerDesigners automatisch generiert und enthält alle Attribute, die über den PowerDesigner festgelegt wurden. Weitere Attribute müssen manuell definiert werden.
DefaultShop*.xml	Definition von Werten, welche für bestimmte Objekte und Attribute gesetzt werden, wenn ein neuer Shop angelegt wird; wird bei einer Cartridge-Installation nicht automatisch mit importiert, Import muss entsprechend mit implementiert werden, z. B. in *Cartridge.pm*
Dependencies.xml	Festlegung der Abhängigkeit zu anderen Cartridges In der Datei definieren Sie, welche Cartridges vorhanden bzw. installiert sein müssen, um die Funktionalität Ihrer Cartridge zu gewährleisten. Falls eine in der Datei eingetragene Cartridge zum Zeitpunkt der Installation Ihrer Cartridge noch nicht installiert wurde, wird diese fehlende Cartridge durch das System automatisch installiert. Um die Abhängigkeiten der Cartridges untereinander zu ermitteln, benutzen Sie die Diagnostics-Cartridge. Siehe dazu *Diagnostics-Cartridge*, *Seite 131*.

Import-Datei	Bedeutung
Features*.xml	Definition von Cartridge-Funktionen als Features. Als Feature ist eine Funktion im Shop durch den Business-Administrator wahlfrei aktivierbar und damit Grundlage für Definition unterschiedlicher Shop-Typen.
Forms*.xml	Definition von Eingabefeldern, die Sie in Formularen verwenden mit Angabe von Bereichsbeschränkung und –prüfung; Siehe dazu *Formhandling*, *Seite 99*.
Hooks*.xml	Definition und Registrierung von Hooks, siehe *Hooks*, *Seite 115*. Wenn Sie mit dem PowerDesigner ein Datenbankmodell erstellen, werden für die Tabellen automatisch entsprechende Hooks generiert und auch als XML-Import-Datei ausgelagert. Weitere Hooks müssen manuell definiert werden.
MailType*.xml	Definition von MailTypes für E-Mail-Ereignisse; diese werden in der Händler-Administration in den E-Mail-Einstellungen aufgelistet. Siehe dazu auch *E-Mail-Events hinzufügen*, *Seite 155*
NavElements*.xml	Definition von Navigationselementen, welche dem Händler zur Verfügung gestellt werden
PageTypes*.xml	Definition logischer Bereiche für die Webseiten und Zuordnung von Templates Siehe dazu *PageType-Konzept*, *Seite 41*.
Permissions*.xml	Definition, welche Aktionen von welchem Nutzer ausgeführt werden dürfen Jeder Aktion können bestimmte Rollen zugeordnet werden, siehe *Rechte und Rollen*, *Seite 21*
PortalSites*.xml	Definition, für welche Länder das jeweilige Portal zur Verfügung steht und Voreinstellung bestimmter Daten wie LocaleID, Registrierungs-URL oder Steuermodell
Search*.xml	Definition von Datenbank-Suchanfragen einschließlich deren mögliche Parameter und Sortierschlüssel
System.xml	Setzen von Attributen im System bei der Installation der Cartridge, z. B. Voreinstellungen für die Web Server-Einstellungen oder die Angabe der verwendeten API-Version in der eBay-Cartridge; Wird bei einer Cartridge-Installation nicht automatisch mit importiert, Import muss entsprechend mit implementiert werden, z. B. in *Cartridge.pm*
TemplateTypes*.xml	Definition von alternativen Darstellungsoptionen für Produkte und Kategorien Als Beispiel dafür siehe in der Händler-Administration die Darstellungs-Optionen für die einzelnen Produkttypen.

17.2 XML-Import

Um XML-Importe manuell ausführen zu können, stehen Ihnen als Entwicklungswerkzeuge verschiedene Import-Scripte zur Verfügung.

Diese kommandozeilenbasierten Tools validieren die XML-Dateien beim Einlesen, Fehlermeldungen werden am Bildschirm und in der Logdatei *%EPAGES_LOG%/error.log* angezeigt. Sie brauchen nach Abschluss des Imports den ePages-Service nicht neu zu starten. Die Zwischenspeicher der Applikationsserver werden automatisch zurückgesetzt. Wenn Sie Ihre Daten auf anderem Weg in die Datenbank einlesen z. B. über direkte SQL-Anweisungen, erfolgt keine automatische Cache-Aktualisierung.

Das Standard-Import-Script heißt *import.pl* und liegt im Verzeichnis

```
%EPAGES_CARTRIDGES%/DE_EPAGES/Object/Scripts
```

und wird in der Konsole wie folgt aufgerufen:

```
perl import.pl [options] [flags] <dateiname>.xml
```

Welche einzelnen Optionen und Parameter wie verwendet werden, wird Ihnen angezeigt, wenn Sie

```
perl import.pl [-help]
```

eingeben.

Wenn Sie z. B. eine XML-Datei zur PageType-Definition importieren wollen, geben Sie ein:

```
perl import.pl –storename Store <cartridgepfad>/PageTypes.xml
```

Mit *Store* bezeichnen Sie die Datenbank, in die importiert werden soll. Da kein Objektpfad explizit angegeben ist, wird standardmäßig */* (*System*) verwendet.

Wollen Sie im Gegensatz dazu z. B. Zahlungsmethoden speziell für den Demoshop importieren, müssen Sie den Objektpfad zum Demoshop beim Import mit angeben. Daher sieht der Importbefehl wie folgt aus:

```
perl import.pl -storename Store -path "/Shops/DemoShop" <cartridge-
pfad>/PaymentMethods.xml
```

Auch hier wird der Objektpfad implizit durch den Vorsatz *System* ergänzt, so dass sich als kompletter Objektpfad *System/Shops/DemoShop* ergibt.

Den Objektpfad zum jeweiligen Ziel können Sie mit Hilfe der Diagnostics-Cartridge ermitteln. Zur Arbeit mit der Diagnostics-Cartridge lesen Sie *Diagnostics-Cartridge*, *Seite 131*. Ein Beispiel zur Demonstration sehen Sie im Kapitel *XML-Export*, *Seite 122*.

Mit der Standardinstallation wird auch der Demoshop mit angelegt. Die gesamten Daten dieses Shops werden in der Import-Datei *DemoShop.xml* im Verzeichnis

```
%EPAGES_CARTRIDGES%/DE_EPAGES/DemoShop/Database/XML
```

bereitgestellt und eingelesen. In dieser Datei finden Sie Beispiele für die Importformate für alle relevanten Objekte eines Shops.

17.2.1 Spezialfall : standards.pl

Es gibt Daten, die nicht als Objekte in die Klassenstruktur von ePages importiert werden können. Dies sind Währungen sowie Sprach- und Länder-Codes in den XML-Dateien

- *%EPAGES_CARTRIDGES%/DE_EPAGES/Object/Database/XML/Currencies_4217.xml*
- *%EPAGES_CARTRIDGES%/DE_EPAGES/Object/Database/XML/Countries_3166.xml*
- *%EPAGES_CARTRIDGES%/DE_EPAGES/Object/Database/XML/Languages.xml*
- *%EPAGES_CARTRIDGES%/DE_EPAGES/Object/Database/XML/Locales.xml*

Diese werden über einen speziellen Import-Befehl eingelesen. Für diesen Fall müssen Sie in der Konsole eingeben:

```
perl standards.pl -storename <Store> <dateiname>.xml
```

17.2.2 Spezialfall: Hooks

Für den Import von Hooks muss ein spezielles Import-Script verwendet werden. Dieses befindet sich im Verzeichnis

```
%EPAGES-CARTRIDGES%/DE_EPAGES/Trigger/Scripts
```

Um Dateien vom Typ *Hooks*.xml* zu importieren ist folgender Aufruf notwendig:

```
perl %EPAGES-CARTRIDGES%/DE_EPAGES/Trigger/Scripts/import.pl
<dateipfad>/Hooks*.xml
```

17.2.3 Spezialfall: Forms

Für den Import von Formularen muss ebenfalls ein spezielles Import-Script verwendet werden. Dieses befindet sich im Verzeichnis

```
%EPAGES-CARTRIDGES%/DE_EPAGES/Presentation/Scripts
```

Um Dateien vom Typ *Forms*.xml* zu importieren ist folgender Aufruf notwendig:

```
perl %EPAGES_CARTRIDGES%/DE_EPAGES/Presentation/Scripts/import.pl
-storename <store> <dateipfad>/Forms*.xml
```

17.3 XML-Export

In Ergänzung zum Import können Sie auch Daten im XML-Format exportieren. Die Syntax ist analog zum Standard-Import, das Script liegt im gleichen Verzeichnis. Über den Befehl

```
perl %EPAGES_CARTRIDGES%/DE_EPAGES/Object/Scripts/export.pl [-help]
```

in der Konsole erhalten Sie auch wieder die möglichen Optionen. Die allgemeine Befehlszeile lautet:

```
perl export.pl [options] [flags] <objects>
```

Hier ist es notwendig, die Objekte anzugeben, die exportiert werden sollen. Nutzen Sie die Export-Funktion, um korrekt strukturierte XML-Dateien als Vorlage für den Import zu erhalten.

Folgendes Beispiel soll die Verwendung von Export und Import demonstrieren. Nehmen wir an, Sie wollen für den Demoshop eine zusätzliche Versandmethode bereitstellen. Diese Methode wollen Sie nicht in der Administration anlegen, sie soll importiert werden.

Um eine Vorlage für die Import-Datei zu bekommen, exportieren Sie die Versandmethoden des Demoshops. Da Sie Objekte eines speziellen Shops exportieren wollen, müssen Sie den Objektpfad zu diesem Shop angeben. Diesen Objektpfad lesen Sie mit Hilfe der Diagnostics-Cartridge aus.

Öffnen Sie die Diagnostics-Cartridge und rufen Sie von der Hauptseite aus **All Shops** auf:

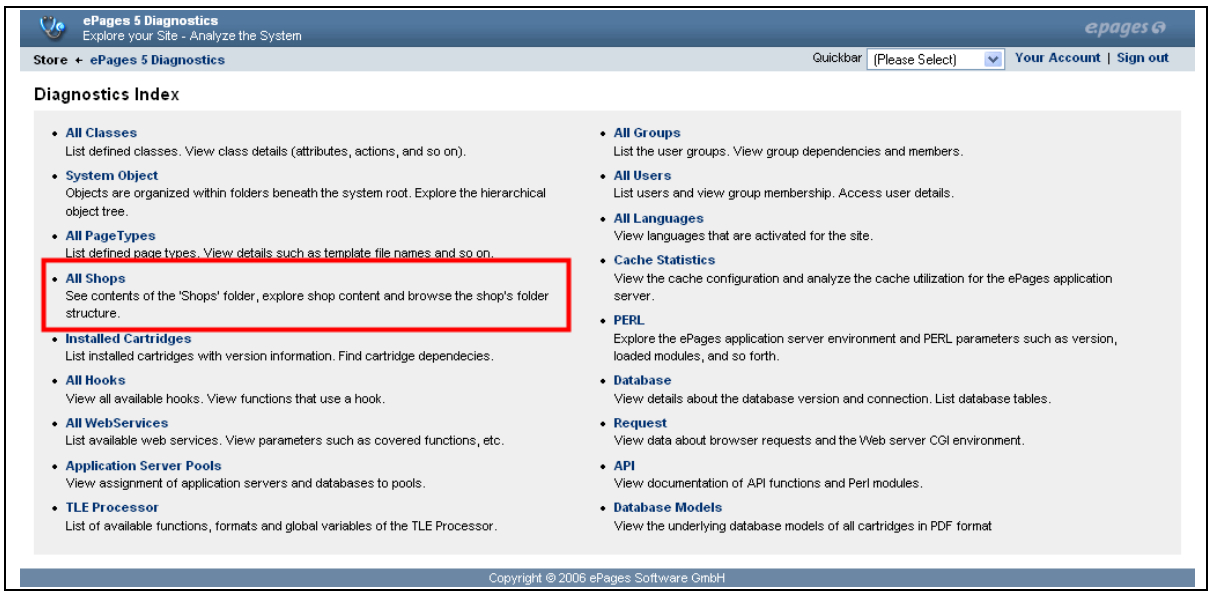

Abbildung 26: Aufruf des ObjektFolders *Shops*

Auf der Seite für den Folder *Shops* sind unter *Child Objects* alle bestehenden Shops aufgelistet, u. a. auch der *Demoshop*, siehe *Abbildung 27*.

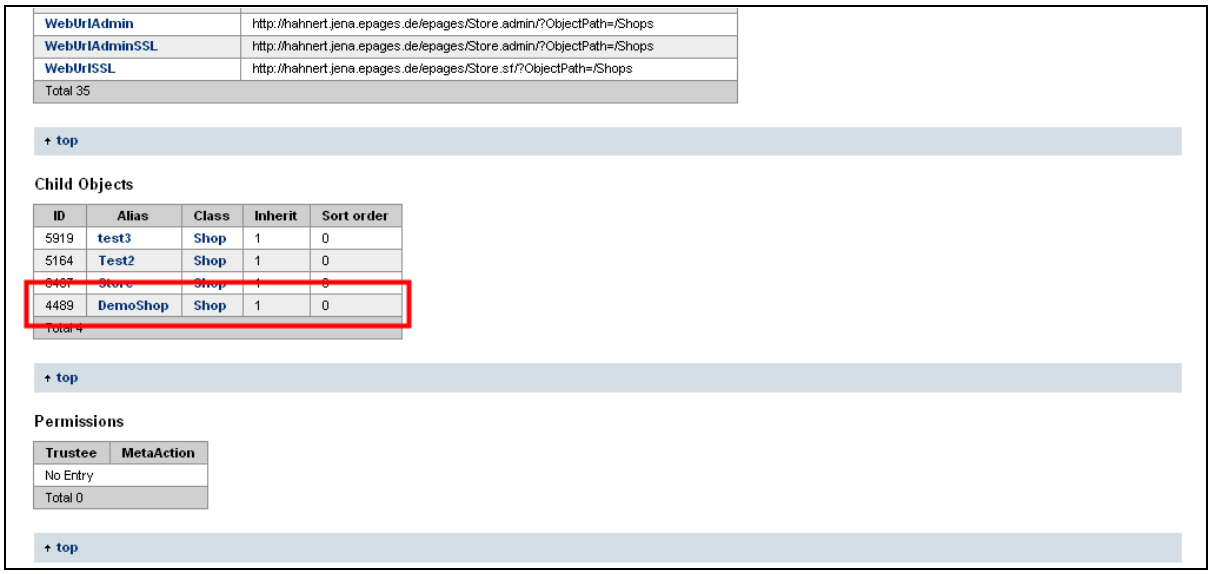

Abbildung 27: Auflistung aller Shops

Klicken Sie nun auf **DemoShop**, um die Parameter für den Demoshop anzuzeigen:

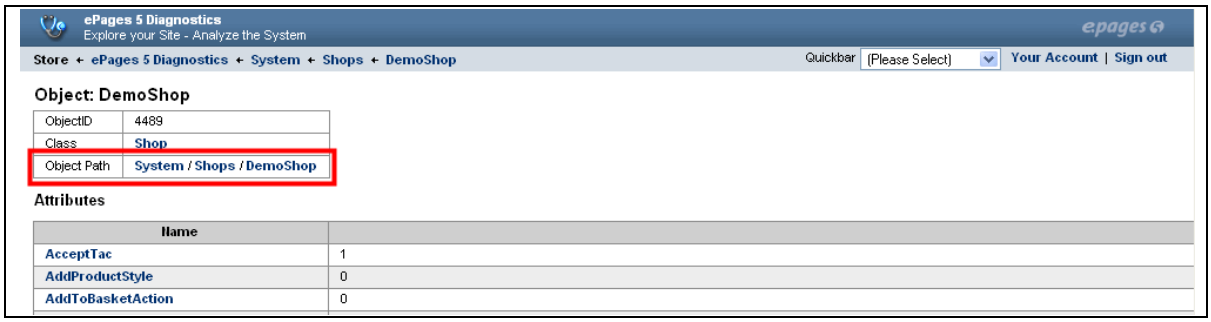

Abbildung 28: Parameter für den DemoShop

Im oberen Teil der Seite können Sie den Objektpfad ablesen: **System / Shops / DemoShop**.

Geben Sie nun folgenden Befehl für den Export der Versandmethoden ein:

```
export.pl -storename Store -path "/Shops/DemoShop" -file ShippingMethods.xml
ShippingMethods
```

Bei der Pfadangabe müssen Sie *System* weglassen, der Pfad wird automatisch um das Root-Objekt *System* ergänzt. Damit sagt die Anweisung folgendes: Exportiere aus der Datenbank *Store* aus dem DemoShop alle Objekte des Folders *ShippingMethods* in die Datei *ShippingMethods.xml*.

Diese Datei ist die Grundlage für den Import einer neuen Versandmethode. Öffnen Sie die Datei und machen sich mit der XML-Struktur vertraut.

Dabei ist folgendes zu beachten:

- In der Exportdatei sind für die Objekt die GUID mit angegeben. Für die Anlage neuer Objekte müssen Sie diese GUID löschen, anderenfalls werden die bestehen Objekte mit den neuen Daten aktualisiert. Sie können aber auch die Option *–withoutguids* verwenden. Dadurch werden die GUID gar nicht erst mit exportiert.
- Achten Sie analog auf den Alias. Ändern Sie den Alias nicht, wird ein bestehendes Objekt mit diesem Alias aktualisiert.

- Verwenden Sie einen neuen Alias, um ein neues Objekt anzulegen.

Die neue Versandmethode soll eine Methode mit Freigrenze sein und z. B. von UPS ausgeführt werden. Suchen Sie in der Datei eine eine passende Methode und überschreiben die Parameter. Der Inhalt der Datei sollte danach wie folgt aussehen:

```
<?xml version='1.0' encoding='utf-8'?>
<epages>
  <!--export level 1-->
  <Object Alias='ShippingMethods' Position='100' Inherit='1'>
    <ShippingMethodFreeLimit Alias='UPS' TaxClass='normal' Position='50'
      IsDefault='0' Inherit='1' ContainerSubTotalAttrName='LineItemsSubTotal'
      IsActivated='1' ShippingType='/ShippingTypes/ExemptionLimit'>
      <AttributeLanguage Language='ger' Name='UPS' />
      <AttributeLanguage Language='eng' Name='UPS' />
      <ShippingLevel UpperBound='100' CurrencyID='EUR' BaseValue='10'
         TaxModel='gross' />
      <ShippingLevel UpperBound='80' CurrencyID='GBP' BaseValue='8'
         TaxModel='gross' />
    </ShippingMethodFreeLimit>
  </Object>
</epages>
```

Codebeispiel 69: XML-Struktur für Import einer Versandmethode

Speichern Sie die Änderungen ab. Abschließend wird die bearbeitete XML-Datei wieder importiert. Dazu geben Sie folgenden Befehl in die Konsole ein:

```
import.pl -storename Store -path "/Shops/DemoShop" ShippingMethods.xml
```

Auch hier geben Sie *System* nicht mit an. Die Daten aus der Datei *ShippingMethods.xml* werden in den *DemoShop* in der Datenbank *Store* importiert.

Im MBO können Sie überprüfen, ob die neue Versandmethode korrekt importiert wurde:

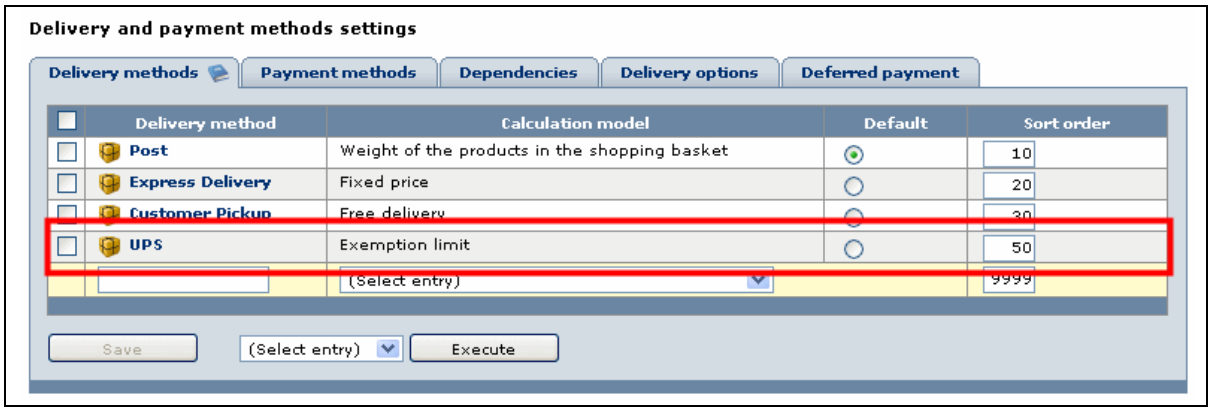

Abbildung 29: Neue Versandmethode importiert

18. Scheduler

Innerhalb der Applikation gibt es verschiede Aktionen, die in regelmäßigen Abständen ausgeführt werden können oder müssen. Dazu gehören solche Aktionen wie Aktualisieren von eBay-Angeboten oder Produktverfügbarkeiten, aber auch regelmäßige "Aufräumarbeiten" in der Datenbank.

Die zeitgesteuerte Ausführung solcher Aktionen erfolgt nach einem betriebssystem-übergreifenden Scheduler-Konzept.

Dabei ist eine Aktion die Ausführung eines Perl- oder (nur für Unix) Shell-Scripts. Der zeitliche Ausführungsplan für eine Aktion wird in einer Steuerdatei festgelegt.

Ein Task besteht aus der auszuführenden Aktion und dem dazugehörigen zeitlichen Ausführungsplan.

Der Scheduler entspricht der Summe der Tasks. Mit Hilfe von Kommandozeilen-Tools wird der Scheduler gestartet und angehalten.

18.1 Konfigurieren von Perl-Script-Tasks

Perl-Script-Tasks werden in der Steuerdatei *%EPAGES_CONFIG%/Scheduler.conf* konfiguriert. Die Steuerdatei ist im Ini-Datei-Format geschrieben, sie ist case-sensitiv. Jeder Task hat seine eigene Sektion, deren Titel der Name des Tasks ist. Kommentarzeilen beginnen mit '#'. In seiner Sektion wird jeder Task über folgende Parameter konfiguriert:

Tabelle 21: Parameter für die Task-Konfiguration

Parameter	Bedeutung
IsActive	Wenn nicht 1, steht der Task nicht im Scheduler
Command	Auszuführendes Perl-Script
Machine	Der Task wird auf den Maschinen ausgeführt, die angegeben sind. Ist keine Maschine angegeben, wird der Task auf allen Maschinen des Systems ausgeführt. Bei Angabe einer Maschine darf für diese nur der erste Teil des Hostnamens, der Compternamе, angeben werden.
Cron	Ausführungszeit im UNIX crontab-Format. Wenn nicht gesetzt, wird die Task nicht unter UNIX ausgeführt
Schtasks	Ausführungszeit im Windows *schtasks*-Format
At	Ausführungszeit im Windows at-Format. Ist weder *Schtasks* noch *At* gesetzt, wird die Task nicht unter Windows ausgeführt. Kennt der Windows-Rechner das Kommando *schtasks*, wird *At* ignoriert.
Options	Optionen für das Perl-Script wie in Command definiert.
Loop	Loops werden benutzt, um ein Perl-Script mehrfach mit den verschiedenen Loop-Optionen (zusätzlich zu den Standardoptionen) aufzurufen. Der hier verwendete Wert muss einem Eintrag der Sektion *[LOOP]* entsprechen.

Beispielsweise sieht der Abschnitt für die Task AutomateAutoCrossSelling in der Datei *%EPAGES_CONFIG%/Scheduler.conf* folgendermaßen aus:

```
…
[AutomateAutoCrossSelling]
At=01:35 /every M,T,W,Th,F,S,Su
Command=
$ENV{EPAGES_CARTRIDGES}/DE_EPAGES/CrossSelling/
     Scripts/automateAutoCrossSelling.pl
Cron=35 1 * * *
IsActive=1
Loop=Store
Machine=
Options=-nooutput
Schtasks=/st 01:35 /sc DAILY
…
```

Codebeispiel 70: Beispiel für Task-Konfiguration in *Scheduler.conf*

Loops werden benutzt, um ein Perl-Script mehrfach mit den verschiedenen Loop-Optionen, zusätzlich zu den Standardoptionen, auszuführen. Ein Beispiel für eine Loop-Sektion sehen Sie hier:

```
[LOOPS]
# command loops (separated by ',')
Store=-storename Store
StoreEnvironment=-storename Store -environment DE
Backup=-storename Backup
BackupDB=-storename Backup -dbname storedb, -storename Backup -dbname sitedb
```

Codebeispiel 71: Sektion *[LOOPS]* in *Scheduler.conf*

Die letzte Zeile besagt z.B., dass das auszuführende Perl-Script zweimal:

a. mit den Parametern -storename Backup -dbname storedb und
b. mit den Parametern -storename Backup -dbname sitedb

ausgeführt werden soll.

Änderungen an der Steuerdatei %EPAGES_CONFIG%/Scheduler.conf werden erst dann wirksam, wenn der Scheduler neu gestartet wurde, siehe *Starten und Stoppen*, *Seite 127*.

18.2 Einfügen von neuen Perl-Script-Tasks

Perl-Script-Tasks werden immer innerhalb einer Cartridge definiert. Im Verzeichnis */Data/Scheduler/* der Cartridge wird die Steuerdatei *Scheduler.conf* angelegt. Hier werden die Tasks konfiguriert. Diese Steuerdatei hat dasselbe Format wie *%EPAGES_CONFIG%/Scheduler.conf*.

Das dazugehörige Perl-Script wird typischerweise im Verzeichnis */Scripts* der Cartridge angelegt.

Bei der Cartridge-Installation werden die Sektionen aus */Data/Scheduler/Scheduler.conf* nach *%EPAGES_CONFIG%/Scheduler.conf* übernommen, wenn der Abschnitt nicht schon in *%EPAGES_CONFIG%/Scheduler.conf* enthalten ist.

Weiterhin sind folgende Hinweise zu beachten:

- Ist eine Sektion schon in der *%EPAGES_CONFIG%/Scheduler.conf* enthalten, wird diese durch die Installation **nicht** überschrieben.
- Enthält die *Scheduler.conf* der Cartridge für den Parameter *Loop* einen Wert , der noch nicht in der globalen Steuerungsdatei *%EPAGES_CONFIG%/Scheduler.conf* in der Sektion *[LOOPS]* enthalten ist, muss dieser Eintrag in der Sektion *[LOOPS]* per Hand ergänzt werden.
- Sektionen müssen bei Bedarf per Hand aus der *%EPAGES_CONFIG%/Scheduler.conf* entfernt werden.

18.3 Konfigurieren von UNIX-Shell-Script-Tasks

Scheduler für UNIX Shell-Scripte werden global im Verzeichnis *$EPAGES_CONFIG/Scheduler.d/* konfiguriert. Während es bei den Perl-Script-Tasks für jeden Task eine Sektion in der globalen Steuerdatei gibt, wird für jeden UNIX-Shell-Script-Task eine eigene Steuerdatei in *$EPAGES_CONFIG/Scheduler.d/* angelegt. Inhaltlich gleichen sich Steuerdatei für UNIX-Shell-Script-Tasks und Sektion für Perl-Script-Tasks.

$EPAGES_CONFIG/Scheduler.d/ enthält folgende Dateien:

- *appserver-*.env*: Tasks, die (in der Crontab) von ep_appl abgearbeitet werden
- *dbserver-*.env*: Tasks, die (in der Crontab) von ep_db abgearbeitet werden
- *webserver-*.env*: Tasks, die (in der Crontab) von ep_web abgearbeitet werden

Andere Dateinamen sind nicht erlaubt. Die .env-Dateien sind (wie immer bei UNIX) case-sensitiv. Nachfolgend ein Beispiel für die Datei *dbserver-RotateLogs.env*, ausgeführt von *ep_db*:

```
#!/bin/sh
# run? (1 - yes, else - no)
ISACTIVE=1
# what?
COMMAND="$EPAGES/bin/logfilemgmt.sh"
# where? (separated by ','; unset -> any)
MACHINE=
# when? (minute/hour/day of month/month/day of week) [see 'man crontab']
CRON="7 * * * *"
# notify who? (unset -> $LOGNAME@localhost)
RECIPIENT=
# search directory/ies for log files (separated by ' ')
SEARCHDIRS="$SYBASE_ASE_LOG"
# -s SIZE[ckm]: required size (in bytes, KB, MB) to compress
SIZE="-s 10m"
# -d DAYS: remove compressed files older than DAYS (unset -> don't remove)
DAYS=
# -a DIR: move compressed files into ARCHIVE directory
ARCHIVE="-a $SYBASE_ASE_LOG/Archive"
# -z ZIPPER: use compression instead of 'gz' (allowed: gz,bz2,lzo,zip,Z)
ZIPPER=
# what command options?
OPTIONS="$SIZE $DAYS $ARCHIVE $ZIPPER $SEARCHDIRS"
# what command loop? (separated by ',')
LOOP=
```

Codebeispiel 72: Beispiel für *dbserver-RotateLogs.env*

Die Variablen *ISACTIVE*, *COMMAND*, *MACHINE*, *CRON*, *OPTIONS* und *LOOP* werden vom Scheduler ausgewertet, alle anderen Variablen sind Hilfsvariablen. Die Bedeutung der Variablen entspricht den Scheduler-Variablen der Perl-Scripte, siehe *Tabelle 21*.

Der Parameter *RECIPIENT* enthält die Adresse, an die alle Fehler-E-Mails gesendet werden.

Die Werte der Variablen müssen gequotet werden, wenn sie Sonderzeichen (wie Leerzeichen z.B. bei *CRON*) enthalten.

Der Parameter *LOOP* enthält alle tatsächlichen Loops, es gibt im Gegensatz zu dem Perl-Script-Scheduler keine gemeinsame Loop-Verwaltung.

18.4 Starten und Stoppen

Zur Ausführung der Scheduler-Tasks gibt es Hilfs-Scripte, die sich in %EPAGES%/bin befinden:

- epagesScheduler.cmd für Windows und
- epagesScheduler.sh für Unix

Für die Scripte gibt es folgende Argumente:

Tabelle 22: Argumente für die Scheduler-Steuerscripte

Argument	Bedeutung
start	Startet alle Scheduler-Tasks aus *%EPAGES_CONFIG%/Scheduler.conf* bzw. aus *%EPAGES_CONFIG%/Scheduler.d/*.env*, d.h. die Tasks werden in die Scheduler-Tabellen (crontab, at/schtasks-Tabelle) eingetragen
stop	Stoppt alle Scheduler-Tasks aus *%EPAGES_CONFIG%/Scheduler.conf* bzw. aus *%EPAGES_CONFIG%/Scheduler.d/*.env*, d.h. die Tasks werden aus den Scheduler-Tabellen (crontab, at/schtasks-Tabelle) ausgetragen. Laufende Prozesse werden nicht angehalten.
show	Zeigt alle laufenden Scheduler-Tasks aus *%EPAGES_CONFIG%/Scheduler.conf* bzw. aus *%EPAGES_CONFIG%/Scheduler.d/*.env*

Bei UNIX werden die Tasks in folgende Crontabs eingetragen:

- Perl-Script-Tasks in die ep_appl-crontab,
- Shell-Script-Tasks, die mit *appserver-* beginnen, in die *ep_appl*-crontab,
- Shell-Script-Tasks, die mit *dbserver-* beginnen, in die *ep_db*-crontab,
- Shell-Script-Tasks, die mit *webserver-* beginnen, in die *ep_web*-crontab

Bei UNIX wird der Scheduler durch

```
/etc/init.d/epages5 start
```

neu gestartet:

- `/etc/init.d/epages5 start_httpd` - startet die Tasks für ep_web
- `/etc/init.d/epages5 start_service` - startet die Tasks für ep_appl
- `/etc/init.d/epages5 start_db` - startet die Tasks für ep_db

Die auszuführenden Tasks laufen innerhalb eines Task-Wrappers. Für UNIX sieht ein Beispiel wie folgt aus:

```
35 1 * * * /opt/eproot/bin/wrapScheduler.sh AutomateAutoCrossSelling
```

Bei Windows wird entsprechend *wrapScheduler.cmd* aufgerufen. Der Task-Wrapper führt folgende Befehle aus:

1. Gibt es ein File *%EPAGES_CONFIG%/Scheduler.d/TASK.env* (hier z. B. *AutomateAutoCrossSelling.env*), dann wird der Task als Shell-Skript behandelt. Wenn der Task nicht mit *appserver-*, *dbserver-* oder *webserver-* beginnt, wird er nicht ausgeführt. Die Environment, die in *%EPAGES_CONFIG%/Scheduler.d/*.env* definiert ist, wird geladen und COMMAND mit OPTIONS ausgeführt.
2. Gibt es eine Sektion TASK in *%EPAGES_CONFIG%/Scheduler.conf*, wird der Task als Perl-Script behandelt, welches mit den in der Sektion definierten Optionen ausgeführt wird.

Der Task wird nur dann ausgeführt, wenn der vorhergehende Task beendet ist. Läuft der Task noch (und es gibt die Datei *TASK.pid* (z. B. *$EPAGES_LOG/Scheduler/cyansun.AutomateAutoCrossSelling.pid*)), wird eine Fehler-E-Mail versandt.

18.5 Scheduler-Task-Output

Bei Perl-Script-Tasks werden alle Fehler-E-Mails an die im TBO angegebene *Error-Mail-Address* gesendet. Bei Shell-Script-Tasks werden die Fehler-E-Mails an die in der .env-Datei gesetzte Variable RECIPIENT gesendet.

Die Tasks haben verschiedene Output-Files MACHINE.TASK.EXT, die abhängig vom User in folgenden Verzeichnissen gespeichert werden:

- ep_appl: *$EPAGES_LOG/Scheduler*
- ep_db: *$SYBASE/ASE-12_5/init/logs*
- ep_web: *$HTTPD_ROOT/logs*

Es gibt folgende Task-Output-Dateien, die durch den Wrapper *wrapScheduler.cmd* bzw. *wrapScheduler.sh* geschrieben werden:

Tabelle 23: Task-Output-Dateien

Task-Output-Datei	Bedeutung
MACHINE.TASK.pid (temporär)	Zeigt an, dass der vorhergehende TASK noch nicht beendet ist, enthält bei UNIX die Process ID MACHINE.TASK.log
MACHINE.TASK.pid (permanent)	Enthält (Fehler-)Output aller bisherigen TASK-Aufrufe
MACHINE.TASK.run (temporär)	Enthält den (Fehler-)Output des Tasks; ist dieses File nicht leer, wird eine Fehler-E-Mail geschrieben und das MACHINE.TASK.log erweitert
MACHINE.TASK.head (temporär)	Enthält einen Header, der, wenn .run nicht leer ist, zusammen mit .run versendet / an .log angehängt wird
MACHINE.TASK.mail (temporär)	E-Mail, die versendet wird, enthält .head und MACHINE.TASK.run

MACHINE.TASK.head sieht typischerweise so aus:

```
*** Output from program $RUN_SCRIPT ***
COMMAND: $COMMAND $OPTIONS > $LOG_FILE.run 2>&1
LOOP: $LOOP
DATE: $DATE
HOSTNAME: $HOST
USERNAME: $LOGNAME
```

19. Diagnostics-Cartridge

Die Diagnostics-Cartridge ist ein Werkzeug zur Anzeige von Objekt- und Klassenstrukturen sowie Datenbankinhalten. Dieses Tool kann von Entwicklern und Designern gleichermaßen genutzt werden, z. B. zum Nachvollziehen von Vererbungslinien, Zuordnung von PageTypes zu Objekten oder zur Abfrage von konkreten Attributinhalten.

Die Diagnostics-Cartridge wird durch die Standardinstallation zur Verfügung gestellt, muss aber separat installiert werden.

19.1 Installation

Die Diagnostics-Cartridge wird im Verzeichnis *%EPAGES_CARTRIDGES%/DE_EPAGES/Diagnostics* mit ausgeliefert, muss aber bei Bedarf installiert werden:

1. `perl Makefile.pl`
2. `make install STORE=Store`

Dabei bezeichnet *Store* die Business-Unit, in die Sie die Cartridge installieren. Welche Datenbanken Sie für Store angeben können, ist in der Datei *%EPAGES-CONFIG%/Database.conf* definiert. Sie können die Cartridge für alle Datenbanken installieren. Dadurch erhalten Sie einen Überblick über alle Objekte, deren Inhalte und Beziehungen, die in der jeweiligen Datenbank gespeichert sind.

> ***Achtung:*** Mit Hilfe der Diagnostics-Cartridge greifen Sie direkt auf die Datenbank zu. Falls Sie die Cartridge auf Live-Systemen benutzen, ändern Sie sofort das Standard-Kennwort, um unbefugten Zugriffen vorzubeugen.

19.2 Anwendung

Die Cartridge rufen Sie über folgende URL auf:

```
http://<servername>/epages/<db>. diagnostics/?ViewAction=ViewDiagIndex
```

Dabei bezeichnet *‹servername›* den Namen des Servers, auf dem die Installation läuft und *‹db›* die Datenbank, die Sie mit der Cartridge untersuchen wollen, z. B. *Store*. Natürlich muss die Cartridge für die Datenbank installiert sein.

Zuerst wird Ihnen die Login-Seite angezeigt und Sie müssen sich anmelden. Der Standard-Benutzername ist *developer*, das Standard-Kennwort ebenfalls *developer*.

> ***Achtung:*** Mit Hilfe der Diagnostics-Cartridge greifen Sie direkt auf die Datenbank zu. Falls Sie die Cartridge auf Live-Systemen benutzen, ändern Sie sofort das Standard-Kennwort, um unbefugten Zugriffen vorzubeugen.

Die Änderung der Anmeldedaten rufen Sie über den Link *Your Account*, siehe *Abbildung 30*, auf.

Nach dem Aufruf sehen Sie die folgende Einstiegsseite des Tools:

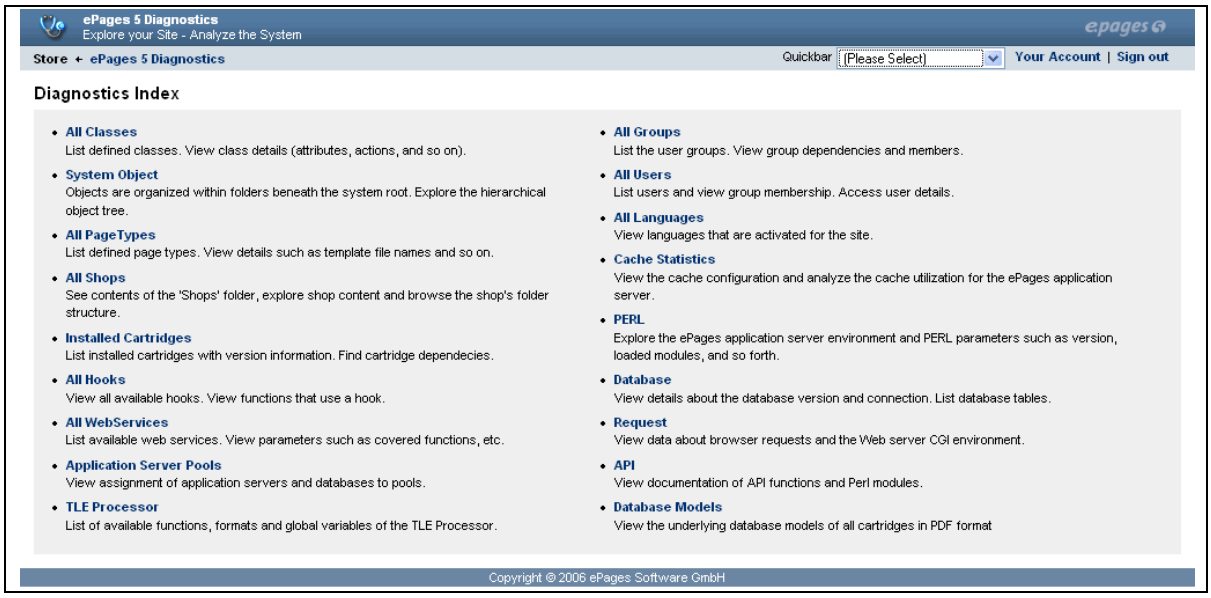

Abbildung 30: Diagnostics-Tool

Hinweis: In der Klassenstruktur wird über den Package-Namen auf die API-Dokumentation verlinkt. Diese wird in einem separaten Fenster geöffnet.

Teil IV:

Design

20. Styles

Ein Style beinhaltet alle notwendigen Informationen zur Darstellung der Shop-Seiten im Browser bezüglich Struktur, Layout und Design. Über die Styles haben Sie eine Möglichkeit, die Shop-Seiten Ihren Bedürfnissen anzupassen. Styles bieten eine einfache Möglichkeit, Storefront-Anpassungen ohne Template- oder PageType-Änderungen vorzunehmen.

Die direkte Gestaltung der Webseiten bezüglich Designs (Farben, Schriftarten usw.) und Layout erfolgt über Cascading Stylesheets, die aus den Styles generiert werden. Für die Storefront–Darstellung wird aus den Style-Information die jeweilige *StorefrontStyle.css* generiert.

Die Arbeit mit Styles ist nachfolgend beschrieben, weitere Beispiele finden Sie in *Anhang C: Anwendungsbeispiele (AWB), Seite 183*.

20.1 Auswahlstyles

Die Darstellung des Shops erfolgt über Auswahlstyles. Diese sind im MBO aufgelistet und der Händler wählt unter **Styles** den passenden aus.

Die Anweisungen für die Darstellung werden in xml-Dateien bereitgestellt. Für jeden Auswahlstyle ist ein Verzeichnis mit dem Bezeichner des Styles angelegt, in dem sich neben anderen auch die entsprechende xml-Datei befindet. Bei der Auswahl des Styles wird die xml-Datei imporiert, daraus die *StorefrontStyle.css* generiert und im angelegten Verzeichnis gespeichert. Diese *StorefrontStyle.css* wird vom Browser für die Darstellung des Shops genutzt.

Neue Auswahlstyles werden über Cartridges bereitgestellt. Pro Style wird in der Cartridge ein eigenes Verzeichnis eingerichtet:

```
<Cartridgename>/Data/Public/SF/Styles/<styleName>
```

Die folgenden Dateien müssen für einen Auswahlstyle im genannten Verzeichnis bereitgestellt werden:

Tabelle 24: Dateien für Auswahlstyle

Datei	Beschreibung
export.xml	enthält Attribute und Werte für die Darstellung von Auswahlstyles
img_small_stylepreview.gif	174 x 107 px – Vorschau Grafik klein
img_medium_stylepreview.gif	450 x 280 px – Vorschau Grafik mittel
img_colorpreview.gif	16 x 16px Grafik – Vorschau Farbvariante
StyleExtension.css	Stylesheet-Datei zur Erweiterung

Weiterhin gehören zu einem Style Hintergrundbilder und Symbole. Diese werden in eigenen Image- und Symbol-Sets bereitgestellt. Für jeden Set-Typ gibt es ein eigenes Verzeichnis in der Cartridge:

```
<Cartridgename>/Data/Public/SF/ImageSet/<imageSetName>
```

```
<Cartridgename>/Data/Public/SF/Icon/<iconSetName>
```

20.1.1 Auswahlstyle anlegen

Ein Auswahlstyle kann wie folgt erstellt werden:

1. Erweiterte Designmöglichkeit im MBO freischalten

Das Design-Tool im MBO bietet die Möglichkeit, bestehende Styles zu exportieren. Damit können Sie komfortabel das erforderliche xml-Format erzeugen. Entfernen Sie dazu die Einträge

```
#REM<!-- und -->#ENDREM
```

aus folgenden Templates:

```
%DEPAGES_CARTRIDGES%/DE_EPAGES/Design/Templates/MBO/Styles/MBO-
Styles.TabPage.html

%DEPAGES_CARTRIDGES%/DE_EPAGES/Design/Templates/MBO/Layout/MBO-
Layout.TabPage.html
```

Dadurch wird für jeden Style eine Export-Funktion freigeschaltet und angezeigt.

2. Design mit Hilfe des Design-Tools gestalten

   Gestalten Sie im MBO auf Basis eines vorhanden Styles Ihren neuen Style *MyStyle*. Wie Sie mit dem Design-Tool arbeiten, lesen Sie im *Handbuch für Händler*, im Kapitel *Gestaltung*.

   Hinweis: Bilder sollten nicht hochgeladen, sondern über ein Imageset bereitgestellt werden. Siehe dazu *Imageset anlegen*, *Seite 137*.

3. Style exportieren

   Klicken Sie in der Style-Liste bei *MyStyle* auf den Link **Export**. Die Datei *export.xml* wird in folgendes Verzeichnis gespeichert:

   ```
   %EPAGES_WEBROOT%/Store/Shops/<ShopName>/Styles/MyStyle
   ```

   Legen Sie eine Cartridge an und kopieren Sie die Datei in das Cartridge-Verzeichnis:

   ```
   /Data/Public/SF/Styles/MyStyle
   ```

   Entfernen Sie folgenden Teil aus der Datei:

   ```
     <Style reference='1' Alias='MyStyle'>
       <!--export level 2-->
       <Style StyleTemplate='/StyleTemplates/MileStones' reference='1' Path='.'
   />
     </Style>
   ```

   Codebeispiel 73: Ausschnitt aus *export.xml*

4. Vorschaubilder und Datei *StyleExtension.css* erstellen

   Legen Sie im selben Verzeichnis alle weiteren Dateien aus *Tabelle 24* an. Die Datei *StyleExtension.css* kann leer bleiben.

5. Auswahlstyle in der Datenbank registrieren

   Der neue Style muss in der Datenbank angemeldet werden. Dazu erzeugen Sie die Datei *StyleTemplates.xml* im Cartridge-Verzeichnis:

   ```
   /Database/XML/
   ```

   Der Quelltext ist folgender:

```
<?xml version='1.0' encoding='utf-8'?>
<epages>
  <StyleTemplates reference="1" Class="Object" Alias="StyleTemplates">
    <StyleTemplate Alias="MyStyle" StyleDir="SF/Styles/MyStyle"
      CustomizeLevel="1" delete="1">
      <AttributeValue Name="Name" Language="de" Value="MyStyle" />
      <AttributeValue Name="Name" Language="en" Value="MyStyle" />
    </StyleTemplate>
  </StyleTemplates>
</epages>
```

Codebeispiel 74: *StyleTemplates.xml*

Dadurch erscheint der Style nach der Installation in **MBO** ›› **Styles** unter **Vorlagen** ›› **Alle Vorlagen anzeigen**.

6. Auswahlstyle einer Kategorie zuweisen

   Der neue Style soll auch einer bestimmten Style-Kategorie zugewiesen werden. Dazu erzeugen Sie die Datei *StyleGroups.xml* im Cartridge-Verzeichnis:

   ```
   /Database/XML/
   ```

   Der Quelltext ist folgender:

```
<?xml version='1.0' encoding='iso-8859-1'?>
<epages>
  <StyleGroups reference="1" Class="Object" Alias="StyleGroups">
    <StyleGroup reference="1" Alias="BusinessSection">
      <Object reference="1" Alias="SubStyleGroups">
        <StyleGroup reference="1" Alias="Touristic">
          <StyleGroupMap StyleTemplate="/StyleTemplates/MyStyle"
            Position="5" />
        </StyleGroup>
      </Object>
    </StyleGroup>
  </StyleGroups>
</epages>
```

Codebeispiel 75: *StyleGroups.xml*

Dadurch erscheint der Style nach der Installation in **MBO** ›› **Styles** unter **Vorlagen** ›› **Auswahl nach Branchen** ›› **Reise & Tourismus**.

7. Cartridge installieren

   Installieren Sie die Cartridge, wie im Kapitel *Cartridges*, *Seite 85* beschrieben. Während der Installation werden die Vorschaubilder und Dateien aus dem Cartridge-Verzeichnis

   ```
   /Data/Public/SF/Styles/MyStyle
   ```

   in das Verzeichnis

   ```
   %EPAGES_WEBROOT%/Store/SF/Styles/MyStyle
   ```

   kopiert. Die Dateien *StyleTemplates.xml* und *StyleGroups.xml* werden in die Datenbank importiert. Danach ist der neue Style verfügbar. Löschen Sie eventuell alle Caches, um eine aktuelle Anzeige zu erzeugen.

20.1.2 Imageset anlegen

In dem Imageset, welches zu jedem Style gehört, werden die Hintergrundbilder, Logo und Content-Separatoren abgelegt. Das Imageset wird in der Style-Cartridge Style bereitgestellt und zwar im Cartridge-Verzeichnis

```
/Data/Public/Store/SF/ImageSet
```

Gehen wir davon aus, dass Sie eine neuen Style nach *Auswahlstyle anlegen*, *Seite 135* angelegt haben. Um das dazugehörige Imageset anzulegen, gehen Sie wie folgt vor.

1. Neuen Style exportieren/sichern

   Haben Sie die Cartridge schon installiert und den neuen Style mit dem Design-Tool weiter bearbeitet, exportieren Sie diesen. Dadurch erhalten Sie eine aktuelle *export.xml*. Diese kopieren Sie wieder in das entsprechende Verzeichnis Ihrer Cartridge.

2. Imageset kopieren

   Als Vorlage für Ihr neues Imageset verwenden Sie ein bereits vorhandenes. Die Standard-Imagesets liegen in Verzeichnis

   ```
   %EPAGES_WEBROOT%/Store/SF/ImageSet
   ```

   Pro Imageset gibt es ein Verzeichnis mit dem Namen des Imagesets. Kopieren Sie den Inhalt eines Imageset-Verzeichnisses in Ihre Cartridge in das Verzeichnis

   ```
   /Data/Public/ImageSet/MyImageSet
   ```

   Dabei ist *MyImagesSet* der Name Ihres neuen Imagesets. Die kopierten Grafikdateien bearbeiten Sie entsprechend.

   Hinweis: Es sollten immer alle Grafikdateien eines Sets verwendet werden. Nicht benötigte Grafiken speichern Sie als transparentes GIF.

3. Imageset eintragen

   Der Name des Imagesets muss in die Datei *export.xml* eingetragen werden. Mit dem Parameter *LayoutImageSet* wird festgelegt, welches Imageset zu dem Style gehört. Suchen Sie in der *export.xml* diesen Parameter und definieren ihn wie folgt:

   ```
   LayoutImageSet='SF/ImageSet/MyImageSet'
   ```

4. Cartridge installieren

   Ist die Cartridge bereits installiert, müssen Sie diese zuerst wieder deinstallieren. Lesen Sie dazu *Deinstallieren*, *Seite 90*.
   Installieren Sie die Cartridge, wie unter *Auswahlstyle anlegen*, *Seite 135* im entsprechenden Abschnitt beschrieben. Dadurch wird das Imageset in das Verzeichnis

   ```
   %EPAGES_WEBROOT%/Store/SF/ImageSet
   ```

   kopiert und steht für den neuen Style zu Verfügung.

20.1.3 Symbolset anlegen

Symbolsets werden prinzipiell wie Imagesets bereitgestellt. Dabei sind folgende Änderungen zu beachten:

- Symbolsets werden im Cartridge-Verzeichnis

```
/Data/Public/SF/Icon
```

abgelegt. Jedes Symbolset liegt in einem eigenen Verzeichnis mit dem Namen des Symbolsets.

- Die Standard-Symbolsets liegen im Verzeichnis

```
%EPAGES_WEBROOT%/Store/SF/Icon
```

In diesem Verzeichnis wird auch das neue Symbolset während der installation gespeichert.

- Der Parameter für die Zuweisung des Symbolsets in der *Datei export.xml* heisst *LayoutIconSet*.

20.1.4 Sub-Styles

Für Styles können Varianten angelegt werden. Diese werden als Sub-Styles bezeichnet. Der Hauptanwendungsfall sind Farbvariationen. Die Sub-Styles werden in eigenen Verzeichnissen im Unterverzeichnis */SubStyles* ihres jeweiligen Basis-Styles angelegt

```
<Cartridgename>/Data/Public/SF/Styles/MyStyle/SubStyles
```

und während der Installation auch in das entsprechende Unterverzeichnis kopiert:

```
%EPAGES_WEBROOT%/Store/SF/Styles/MyStyle/SubStyles/<substylename>
```

Sub-Styles gernerieren Sie wie unter *Auswahlstyle anlegen*, *Seite 135* beschrieben. Folgende Besonderheit ist zu beachten:

Die Datei *StyleTemplates.xml* für Substyles sieht wie folgt aus:

```
<StyleTemplate Alias="MyStyle" StyleDir="SF/Styles/MyStyle" CustomizeLevel="1"
               Position="5" delete="1" >
  <Object Alias="SubStyleTemplates">
    <StyleTemplate Alias="Red"  StyleDir="SF/Styles/MyStyle/SubStyles/Red"
                   CustomizeLevel="1" Position="10"/>
    <StyleTemplate Alias="Blue" StyleDir="SF/Styles/MyStyle/SubStyles/Blue"
                   CustomizeLevel="1" Position="20"/>
  </Object>
</StyleTemplate>
```

Codebeispiel 76: *StyleTemplates.xml* für SubStyles

Die Datei *StyleGroups.xml* für Substyles sieht wie folgt aus:

```
<StyleGroupMap reference="1" StyleTemplate="/StyleTemplates/MyStyle" >
  <StyleTemplateVariation
     StyleTemplate="/StyleTemplates/MyStyle/SubStyleTemplates/Red"
     Position="10" />
  <StyleTemplateVariation
     StyleTemplate="/StyleTemplates/MyStyle/SubStyleTemplates/Blue"
     Position="20" />
</StyleGroupMap>
```

Codebeispiel 77: *StyleGroups.xml* für SubStyles

Die Imagesets für Sub-Styles liegen parallel zum Imageset des jeweiligen Basis-Styles, hier werden keine Unterverzeichnisse verwendet. Die Verzeichnisse von Sub-Style-Imagesets sollten die Namen des Basis-Styles mit enthalten. So sollte das Verzeichnis für das Imageset des Sub-Styles *Sport* unter dem Basis-Style *Basic* dann *BasicSport* heißen.

Anhänge

Anhang A: Performance Tuning

21. Allgemeine Maßnahmen

Eine Hauptanforderung an Ihr ePages-System ist eine gute Performance. Es gibt verschiede Bereiche, in denen Sie Einfluß auf die Performance Ihrer Installation nehmen können, angefangen bei der Installation, über optimierte Einstellungen in Konfigurations-Dateien bis hin zu optimiertem Quellcode in HTML und Perl.

Im nachfolgenden sind Möglichkeiten beschrieben, deren Beachtung und Umsetzung zu Performancesteigerungen führen.

21.1 Page Caching

Bezüglich des Caching ist es wichtig die Balance zwischen Performance und Aktualität der Seiten zu finden. Dabei gibt es die Möglichkeit, komplette HTML-Seiten zwischenzuspeichern.

Bei der Zwischenspeicherung von HTML-Seiten werden diese Seiten als Dateien gespeichert und bei nachfolgenden Anfragen unverändert angezeigt. Die Einstellungen für diese Zwischenspeicherung steuern Sie über die Händler-Administration, Funktion *Optimierung* in Menüpunkt *Einstellungen*. Lesen Sie dazu das entsprechende Kapitel im *Handbuch für Händler*. Grundsätzlich sollten Sie hier eine möglichst lange Gültigkeit der Seite einstellen und nach Bedarf die Aktualisierung manuell anstoßen.

Eine weitere Möglichkeit, die Performance durch Caching zu verbessern ist das partielle Caching von Templates. Siehe dazu *Partielles Caching*, *Seite 150*.

Hinweis: Die in der Optimierung eingestellte "Verfallszeit" für Caches wirkt sich auf Page Caching und partielles Caching aus.

21.2 Templateverarbeitung

Neben der Speicherung kompletter HTML-Seiten werden auch vorkompilierte Dateien zwischengespeichert, für die dann noch die aktuellen dynamischen Daten geladen werden müssen – die *ctmpl*-Dateien. Zu *ctmpl*-Dateien lesen Sie *Template-Prozess*, *Seite 33*.

Bezüglich der *ctmpl*-Dateien haben Sie die Möglichkeit, die Prüfung des Zeitstempels auszuschalten. Dadurch wird nicht mehr auf mögliche Veränderungen geprüft und die Seite wird beim Request aus dem vorhandenen Kompilat mit aktuellen Daten aus der Datenbank neu erstellt und angezeigt. Die Steuerung dieser Prüfung geschieht über den Parameter *DisableCtmplStat* in der Datei

```
%EPAGES_CONF%/DataCache.conf
```

Tabelle 25: Parameter für *DisableCtmplStat*

Parameter	Bedeutung
DisableCtmplStat =0	Diese Einstellung ist Standard und prüft die HTML- und xml-Dateien auf die mögliche Änderungszeit.
DisableCtmplStat =1	Bei dieser Einstellung wird die Prüfung übersprungen

> ***Achtung:*** Bei DisableCtmplStat =1 werden Änderungen an Templates (**.html*) und Dictionaries (z. B. **.de.xml*) nicht mehr sichtbar. Diese Einstellung sollten Sie nur am Live-System vornehmen.

Beachten Sie, dass bei ausgeschalteter *ctmpl*-Prüfung die Aktualisierung der HTML-Seiten eingeschränkt ist. Wenn Sie in dem Status die HTML-Seiten über *Optimierung* neu erstellen lassen, werden die vorhandenen Kompilate benutzt und mit aktuellen Daten neu erstellt. Neue Kompilate werden nicht erzeugt, auch wenn Änderungen in Templates oder Sprachdateien vorliegen.

21.3 Prozess-Prioritäten

In der Sektion *[GLOBAL]* der Datei

```
%EPAGES_CONF%/WebInterface.conf
```

lassen sich Prioritäten für verschiedene Prozesse festlegen. Mit diesen Konfigurationsmöglichkeiten kann man lang laufende Prozesse, wie Importe, Duplizieren usw. umpriorisieren.

Dadurch verringert sich die Wartezeit für die anderen Requests, weil der Server nicht durch lang laufende Prozesse blockiert wird.

Eine feine Anpassung der Prioritäten ist aber für die entsprechende Rechnerkonfiguration nötig.

Solche Einstellungen sind z. B.

Tabelle 26: Prozess-Prioritäten

Prozess-Priorität	Bedeutung
PRIORITY=HIGH	Priorität, mit der der Applikationsserver läuft
MONITOR_PRIORITY=ABOVE_NORMAL	Priorität, mit der der Applikationsserver im Monitormodus Requests ausführt
MANUAL_MONITOR_PRIORITY=NORMAL	Priorität, mit der der Applikationsserver im manuellen Monitormodus Requests ausführt (Importe usw.)

Standardeinstellung ist:

- PRIORITY=ABOVE_NORMAL
- MONITOR_PRIORITY=ABOVE_NORMAL
- MANUAL_MONITOR_PRIORITY=NORMAL

> **Hinweis:** Für Linux/Unix lassen sich die Prozesse nicht höher als NORMAL einstellen, da man dafür *root*-Berechtigung benötigt. Über diese Berechtigung verfügt der Nutzer *ep_appl*, unter dem der ePages-Service läuft, nicht.

21.4 Verkürzung Antwortzeit des ersten Requests

Sobald der erste Request an einen der Applikationsserver gesendet wird, wird dessen Cache mit Informationen über die Objektstruktur gefüllt und Perl-Module werden kompiliert. Dies führt zu einer relativ langen Requestzeit. Diese erste Antwortzeit können Sie verkürzen, indem Sie die Objektstruktur gleich beim Start eines Applikationsserver laden und die Perl-Module vorkompilieren lassen. Kommentieren Sie dazu in der Datei

```
%EPAGES_CONF%/Hooks.conf
```

den Eintrag

```
[AppServerStartup]
; DE_EPAGES::Presentation::Hooks::AppServerStartup::OnAppServerStartup=10
```

aus, indem Sie das Semikolon entfernen.

21.5 Debug-Informationen

Die Grundlagen zu Debug-Informationen lesen Sie in *Template-Debugging*, *Seite 38*. Die Debug-Informationen über Quelle und Laufzeit von Includes blähen die zu ladenden HTML-Seiten auf. Dadurch kommt es zu längeren Ladezeiten. Schalten Sie daher im Live-Betrieb die Debug-Informationen aus, indem Sie in der Datei

```
%EPAGES_CONF%/log4perl.conf
```

den Eintrag

```
log4perl.category.DE_EPAGES::Presentation::API::Template::INCLUDE=DEBUG
```

auskommentieren. Setzen Sie ein Semikolon vor diese Zeile.

21.6 Shop-Einstellungen

Auch durch sinnvoll gewählte Einstellungen in der Händler-Administration kann die Performance erhöht werden. Hier einige Beispiele:

- Die Anzahl der Produkte auf der Startseite erhöht die Ladezeit (je nach Optimierungseinstellungen)
- Anzahl der Aktionsprodukte überprüfen, da diese Produkte bei Anzeige des entsprechenden Navigationselementes *Promotional products* geladen werden müssen (je nach Optimierungseinstellungen)
- Unnötige Navigationselemente ausblenden

21.7 System-Monitoring mit Spy.pl

Mit dem Spy-Monitor überprüfen Sie den Request Router-Aktivitäten und –Daten. Sie erhalten einen Überblick über alle Applikationsserver der Installation. Mit Hilfe des Monitors erhalten Sie Informationen, die Sie zur Optimierung des Systems und zur Steigerung der Performance benötigen. Das sind Informationen wie

- Woher kommen die Cache-Overflows?
- Wie schnell sind die Maschinen?
- Wie ist die Auslastung der einzelnen Maschinen?
- Wie ist das System ausbalanciert?

Starten Sie den Monitor nach der Installation über folgende URL:

```
http://<yourserver>/Monitor/spy.pl
```

Nach Absenden des Requests wird eine Webseite mit verschiedenen Sektionen angezeigt, die alle 5 Sekunden neu aufgebaut wird. Folgende Sektionen werden angezeigt:

Sektion *CacheStatistics*

Inhalt:

- Typ und Anzahl der der Cache-Updates jedes Applikationsservers
- Gespeicherte Daten, welche zwischen den betreffenden Applikationsservern ausgetauscht werden

Spalten:

- Pool: application server pool name
- ServerIP:Port server IP and port address
- PID: - process identifier
- Status: idle/busy/reserved
- LastContact: - last contact in seconds
- Overflow: - RR has cache overflow for this AS
- CacheItemCount: count of cache updates
- CacheItems: - cache update entries

Sektion *Server*

Inhalt:

- Auflistung aller Applikationsserver
- Die Pools, denen die Applikationsserver zugeordnet sind
- Andere Parameter, wie aktueller Request, letzte URL, letzte Seite

Spalten:

- Pool: application server pool name
- ServerIP:Port: server IP and port address
- PID: process identifier
- Status: idle/busy/reserved
- LastContact: last contact in seconds
- Hits: count of requests
- LongReq: count of long requests (longer than 5 seconds)
- MeanReq: count of normal requests (less than 5 seconds)
- MeanTime: mean execution time (for requests less than 5 seconds)
- CacheItems: count of cache updates
- Site: current/last request was assigned to this site
- URL: current/last request URL

Sektion *Pools*

Inhalt:

- Auflistung aller Applikationsserver-Pools
- Wie viel und welche Applikationsserver zu den einzelnen Pools zugeordnet sind
- Übersicht über Cache-Daten

Spalten:

- Name: pool name
- IdleServer: count of idle servers
- CountServer: count of servers assigned to pool
- CountMC: count of other MessageCenters/RequestsRouters (which have AS of this pool)
- CountServerCache: count of update cache entries for AS
- CountMCCache: count of update cache entries for other MC

Sektion *RequestRouters*

Inhalt:

- Auflistung aller Request Router mit IP, Port, Status und Ping

Spalten:

- Server IP: IP address of MC/RR
- Port: port address of MC
- Status: currently started -1 (not pinged), not alive 0, successfully pinged 1
- Last Ping: last contact

21.7.1 Installation

Sie finden die Datei *spy.pl* nach der Installation im Verzeichnis *%EPAGES_SHARED%\Monitor*.

Prüfen Sie, ob in der Datei *WebInterface.conf* in der Sektion für den Request Router folgender Eintrag vorhanden ist:

```
MONITOR=10041
```

Falls der Eintrag nicht vorhanden ist, fügen Sie ihn ein.

Installation auf Windows mit IIS 6

- Sie können einen Test durchführen, indem Sie das Script *spy.pl* auf der Kommandozeile ausführen. Wird eine Ausgabe in der Konsole angezeigt, läuft der Request Router und die entsprechenden Berechtigungen sind vorhanden.
- Erzeugen Sie in den IIS 6 für die Standardwebseite ein virtuelles Verzeichnis für das Verzeichnis, in dem die *spy.pl* liegt:

  1. Standardwebseite ›› Neu ›› Virtuelles Verzeichnis ››Alias: Monitor, Path: "%EPAGES_SHARED\Monitor
  2. Setzten Sie folgende Berechtigungen für das virtuelle Verzeichnis: READ, EXECUTE
  3. Tragen Sie folgendes Mapping für das virtuelle Verzeichnis ein (Monitor" ›› Eigenschaften ›› virtuelles Verzeichnis ››Konfiguration ›› Zuordnungen):
     Ausführbare Datei: "‹epages5directory›\Perl\bin\perl.exe "%s" %s"
     Erweiterung: ".pl"

- Setzen Sie in der Servererweiterungen "Alle unbekannten CGI Erweiterungen" auf "Erlaubt"
- Starten Sie die IIS 6 neu

Installation auf UNIX

- Setzen Sie die Berechtigung:

  - . /etc/default/epages5
  - cd $EPAGES_SHARED/Monitor
  - chown ep_appl:ep_web *.pl
  - chmod u=rwx,g=rwx *.pl

- Bearbeiten Sie die *httpd.conf*:

  - ScriptAlias /Monitor/spy.pl "/opt/eproot/Shared/Monitor/spy.pl"
  - ‹Location /Monitor/*›
    Order Deny,Allow
    Deny from all
    Allow from ‹IP-Adresse›
    ‹/Location›

22. Maßnahmen während der Entwicklung

Beim Entwickeln können Sie durch bestimmte Methoden oder Verwendung bestimmter Funktionen die Performance Ihres Systems erhöhen, bzw. performante Cartridges erstellen.

22.1 Template-Analyse

Es ist wichtig festzustellen, welche Ausführungszeiten die einzelnen Includes aufweisen. Mit diesen Informationen können Sie gezielt die Includes identifizieren, die hohe Ladezeiten des Templates verursachen.

Eine Möglichkeit ist das Aktivieren der Debug-Informationen, siehe dazu *Debug-Informationen*, *Seite 145*.

Damit ermitteln Sie alle Includes und deren Ausführungszeiten.

Mit Hilfe der TLE-Funktion *TimeThis* können Sie gezielt die Ausführungszeit einzelner Includes bestimmen. Folgender TLE-Block misst die Ausführungszeit des INCLUDE-Templates *Body* und zeigt diese rot auf der Webseite an.

```
#BLOCK("TimeThis", "Body", 1)
  #INCLUDE("Body")
#ENDBLOCK
```

Codebeispiel 78: Messung der Ausführungszeit eines Includes

Der *TimeThis*-Block kann auch für Teilbereiche innerhalb eines Templates verwendet werden. Ist der zweite Parameter *0* oder nicht angegeben, erscheint die Zeit im HTML-Quelltext als HTML-Kommentar.

Eine weitere Möglichkeit, eine Übersicht über alle Template-Includes anzuzeigen, ist die Ausführung des Scriptes *analyzeIncludes.pl* im Verzeichnis

```
%EPAGES_CARTRIDGES%\DE_EPAGES\TLE\Scripts
```

Folgendes Beispiel zeigt die Anwendung und die entsprechende Ausgabe:

```
perl analyzeIncludes.pl http://dmo/epages/Store.sf/?ObjectPath=/Shops/DemoShop
get http://dmo/epages/Store.sf/?ObjectPath=/Shops/DemoShop 1.438
 0.094 BasePageType.Head
  0.031 SF.Title
  0.016 BasePageType.Script
   0.016 BasePageType.Script-Base
  0.000 SF.Head-ContentType
  0.016 SF.Style
  0.016 SF-Shop.MetaTags
 1.156 SF.Body
  0.000 SF.INC-Etracker
  1.125 SF.Layout
   1.125 SF.Layout1
    0.125 SF.Header
     0.016 SF.ShopName
     0.063 SF.LoginBox
      0.016 SF.LoginBoxLinks-UserLostPassword
      0.000 SF.LoginBoxLinks-Register
      0.016 SF.LoginBoxLinks-Newsletter
     0.031 SF.ProductSearchBox
    0.094 SF.NavBarTop
     0.000 SF.HomePageLink
     0.016 SF.ImprintLink
     0.016 SF.ContactLink
     0.016 SF.TermsAndConditionsLink
     0.016 SF.CustomerInformationLink
     0.016 SF.BasketLink
    0.078 SF.NavBarTop
     0.016 SF.HomePageLink
     0.016 SF.ImprintLink
     0.000 SF.ContactLink
     0.000 SF.TermsAndConditionsLink
     0.016 SF.CustomerInformationLink
     0.016 SF.BasketLink
    0.266 SF.NavBarLeft
     0.016 SF.TrustedShopSeal
     0.016 SF.ProductSearchBox
     0.063 SF.CategoriesListBox
     0.156 SF.SpecialOfferBox
     0.000 SF.InfoText
    0.281 SF-Shop.Content
    0.125 SF.NavBarRight
     0.016 SF.MiniBasketBox
     0.031 SF.CurrencyBox
     0.016 SF.SpecialOfferLink
     0.063 SF.LoginBox
      0.000 SF.LoginBoxLinks-UserLostPassword
      0.016 SF.LoginBoxLinks-Register
      0.016 SF.LoginBoxLinks-Newsletter
    0.063 SF.NavBarBottom
     0.000 SF.Copyright
     0.031 SF.Logo
     0.016 SF.TrustedShopSeal
    0.094 SF.Footer
     0.063 SF.LocaleFlags
     0.000 SF.Copyright
     0.031 SF.ExternalHomePageLink
 Total: 1.25
```

Codebeispiel *79*: Auflistung aller Includes einer URL mit Ausführungszeit

22.2 Partielles Caching

Mit Hilfe des partiellen Caching können Sie für einzelne Templateabschnitte steuern, ob sie aktuell erzeugt oder als vorbereiteter HTML-Code geladen werden. Dadurch haben Sie die Möglichkeit, die Zeit für die Erstellung der angeforderten Seite zu verkürzen.

Grundlage dafür ist die TLE-Funktion *CachedInclude*, die innerhalb einer *#BLOCK*-Anweisung ausgeführt wird. Die Syntax für die Funktion lautet:

```
#BLOCK("CachedInclude", <object> , <filename>) ... #ENDBLOCK
```

In *object* geben Sie das Objekt an, für welches die Daten zwischengespeichert werden. Unter *filename* wird die Datei mit dem statischen HTML-Code gespeichert.

Das Caching bezieht sich auf den gesamten Inhalt innerhalb der *#BLOCK*-Anweisung. Nach dem Verarbeiten der #BLOCK-Anweisung wird der entstandene HTML-Code als statische HTML-Datei unter dem angegebenen Dateinamen abgespeichert. Beim nächsten Seitenaufruf prüft die Funktion *CachedInclude*, ob die Datei vorhanden ist. Existiert sie, wird der HTML-Code aus der Datei verwendet.

Partielles Caching bietet sich für Templates an, bei denen die Anzahl wahrscheinlicher Varianten nicht so hoch ist. Das gilt auch für Templates bzw. Includes, deren Inhalt über längere Zeiträume konstant bleibt.

Templates, die z. B. benutzerspezifische Daten anzeigen, wie z. B. Warenkorb oder benutzerspezifische Produktpreise sollten nicht zwischengespeichert werden, da die Daten sonst nicht aktuell sind.

Prinzipiell sollten Sie nur die wirklich langsamen Includes auf partielles Caching hin untersuchen, da auch der Overhead für das Caching Ausführungszeit benötigt.

Beispiele für partielles Caching sind Auflistungen wie Aktionsprodukte oder Navigationselemente, weil diese im Normalfall für alle Benutzer immer gleich angezeigt werden.

Die Dateien werden im Verzeichnis

```
%EPAGES_STATIC%/Store/Shops/<shopname>/
```

abgelegt.

Eine solche Datei soll so allgemein wie möglich sein, um sie an vielen Stellen einsetzen zu können. Trotzdem muss für jede Variante eine eigene Datei angelegt werden. Daher kommt dem Dateinamen eine besondere Bedeutung zu. Spiegelt der Name die Varianz nicht ausreichend wieder, werden Dateien überschrieben. Dann stehen die verschiedenen Inhalte nicht zu Verfügung. Aber es dürfen nicht zu viele Dateien erzeugt werden, um das System nicht zu belasten.

Für die Dateinamen gilt daher Folgendes:

- Der Dateiname setzt sich aus mehreren Bestandteilen zusammen.
- Ein Teil beschreibt den Inhalt des Seitenbereiches, der zwischengespeichert werden soll.
- Ein Teil besteht aus einer TLE, welche durch ihren Inhalt die entsprechende Variante definiert.
- Die Teile sind durch _ getrennt.
- Durch das System wird die GUID des Objektes vorangestellt.
- Die Dateiendung ist *content*.

Folgendes Beispiel soll dies verdeutlichen:

```
#BLOCK("CachedInclude", #Shop.Object, "SF.SpecialOfferBox" . #CacheIncludesNames)
   ...
#ENDBLOCK
```

Codebeispiel 80: Steuerung des partiellen Caching

Das Objekt, für dessen Anzeige ein bestimmter Bereich zwischengespeichert wird, ist das *Shop*-Objekt. Der Bereich, welcher zwischengespeichert werden soll, ist in der Storefront der Bereich, in dem die Aktionsprodukte angezeigt werden. Dies wird definiert durch *SF.SpecialOfferBox*. Die TLE *#CacheIncludesNames* beinhaltet die Angabe der aktuellen Anzeigesprache, siehe *Tabelle 27*. Diese TLE wird verwendet, wenn sich der Inhalt nur in Abhängigkeit von der Sprache ändert.

Für Deutsch als Anzeigesprache ergibt sich als Dateiname:

```
INC-436B4797-2D4C-607A-5558-AC14080F2485-SF.SpecialOfferBox_de.content
```

Beim nächsten Aufruf dieses Abschnitts erzeugt das System den Dateinamen unter der Verwendung wieder dieser TLE, erkennt, dass für *de* der HTML-Inhalt gespeichert ist und kann diesen sofort verwenden.

Für Englisch als Anzeigesprache ergibt sich als Dateiname:

```
INC-436B4797-2D4C-607A-5558-AC14080F2485-SF.SpecialOfferBox_en.content
```

So kann je nach Spracheinstellung der jeweils richtige HTML-Code verwendet werden.

Folgende TLE sind vordefiniert:

Tabelle 27: Dateinamen-TLE für partielles Caching

TLE	Inhalt
CacheIncludesNames	Angabe der Sprache
CacheIncludesPrices	Angabe von Sprache, Region und Währung
CacheIncludesPager	Angabe von aktueller Seite der Tabelle, Sortierung, Sortierrichtung, Tabellenlänge (Anzahl der Zeilen) und Anzahl der angezeigten Seitenzahlen in der Fußzeile der Tabelle
CacheIncludesPagerPrices	Angabe von Sprache, Region, Währung, aktueller Seite der Tabelle, Sortierung, Sortierrichtung, Tabellenlänge (Anzahl der Zeilen) und Anzahl der angezeigten Seitenzahlen in der Fußzeile der Tabelle

Als Beispiel ändern wir für *Codebeispiel 80*, *Seite 151* die TLE:

```
#BLOCK("CachedInclude",
        #Shop.Object, "SF.SpecialOfferBox" . #CacheIncludesPrices)
   ...
#ENDBLOCK
```

Codebeispiel 81: Geänderte TLE

Daraus ergibt sich folgender neuer Dateiname:

```
INC-436B4797-2D4C-607A-5558-AC14080F2485-SF.SpecialOfferBox_de_de_DE_EUR.content
```

22.3 Nutzung der #LOCAL-Anweisung

Bei der Verarbeitung von Listen sollten Attribute oder Elemente nicht mehrfach ausgelesen, sondern als lokale Variable bereitgestellt werden.

Benutzen Sie die #LOCAL-Anweisung, um den wiederholten Zugriff auf hierarchische Objektstrukturen zu beschleunigen.

Folgendes Beispiel soll dies verdeutlichen:

```
#LOOP(#Category.Products)#NameOrAlias #ENDLOOP
#IF(NOT #COUNT(#Category.Products)) list empty #ENDIF
```

Codebeispiel 82: Wiederholtes Auslesen von Objekten

Im *Codebeispiel 82* wird für jeden Durchlauf die Objekthierarchie abgefragt. Dies ist relativ zeitaufwändig. Die deutlich schnellere Variante ist das Auslesen der Produkte der betreffenden Kategorie und abspeichern in einer TLE-Variablen. Beim Durchlauf wird dann immer auf diese Variable zugegriffen:

```
#LOCAL("CategoryProducts", #Category.Products)
  #LOOP(#CategoryProducts) NameOrAlias #ENDLOOP
  #IF(NOT #COUNT(#CategoryProducts)) list empty #ENDIF
#ENDLOCAL
```

Codebeispiel 83: Verwendung der #LOCAL-Anweisung zur Performance-Steigerung

22.4 Auslagern komplexer TLE-Blöcke

Lagern Sie komplexe TLE-Blöcke als TLE-Funktionen oder Klassen-Attribute aus. Verzichten Sie jedoch darauf, HTML in die Perl-Funktionen zu verlagern.

Folgendes Beispiel soll diese Methode verdeutlichen: Es soll eine Liste aller sichtbaren Produkte angezeigt werden. Vor der Liste wird die Anzahl der gefundenen Produkte angezeigt. Eine reine HTML-Lösung sehen Sie hier:

```
#LOCAL("TempProducts", #Products)
 #LOCAL("VisibleProductsCounter",0)
  #LOOP(#TempProducts)
   #IF(#IsVisible)
    #SET("VisibleProductsCounter", #VisibleProductsCounter + 1)
   #ENDIF
  #ENDLOOP
count of visible products = #VisibleProductsCounter
 #ENDLOCAL
 #LOOP(#TempProducts)
  #IF(#IsVisible)
   #NameOrAlias #Description
  #ENDIF
 #ENDLOOP
#ENDLOCAL
```

Codebeispiel 84: Funktion – HTML-codiert

Sie können diese Funktion performanter gestalten, indem Sie die Daten mit Hilfe einer Perl-Funktion sammeln, auswerten und in TLE-Variablen speichern und nur die Anzeige der Resultate in HTML codieren:

```
…
sub getAttributes {
    shift;
    my ($Object, $aNames) = @_;
    my $hInfo = {};
    foreach my $Name (@$aNames) {
        if ($Name EQ 'VisibleProducts') {
            $hInfo->{$Name} = [grep { $_->get('IsVisible') } @{$Object-
>get('Product')}];
        }
    }
    return $hInfo;
}
…
```

Codebeispiel 85: Perl-Funktion zur Datenaufbereitung

Im HTML-Quelltext wird dann diese Funktion angewendet.

```
#LOCAL("TempVisibleProducts", #VisibleProducts)
count of visible products = #COUNT(#TempVisibleProducts)
 #LOOP(#TempVisibleProducts)
   #NameOrAlias #Description
 #ENDLOOP
#ENDLOCAL
```

Codebeispiel 86: Anzeige der Daten in HTML auf Basis der vorbereiteten TLE

Anhang B: Entwicklerhinweise

23. E-Mail-Events hinzufügen

Um eigene E-Mail-Ereignisse zum System hinzuzufügen, müssen Sie prinzipiell wie folgt vorgehen:

- MailType anlegen
- PageType anlegen und Template zuweisen
- Funktion implementieren
- Aktion registrieren
- Permission definieren

Kapseln Sie diese Funktion wie üblich in einer Cartridge. Für die Erläuterung der einzelnen Schritte orientieren wir uns an der Cartridge *ContactForm*.

23.1 MailType anlegen

MailTypes werden in der Datei *MailTypeTemplates*.xml* definiert. Für * können Sie eine wahlfreie Dateinamenerweiterung angeben. Die Datei muss im Verzeichnis */Database/XML* Ihrer Cartridge gespeichert werden.

Den erforderlichen Quelltext sehen Sie in *Codebeispiel 87*.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<epages>
  <System reference="1" Path="/">
    <MailTypeTemplates Class="Object" Alias="ShopMailTypeTemplates">
      <MailTypeTemplate Alias="ShopContactForm"
        PageType="/PageTypes/Mail-ShopContactForm"
        HasToField="1" delete="1">
        <AttributeValue Name="Position" Value="20" />
      </MailTypeTemplate>
    </MailTypeTemplates>
  </System>
</epages>
```

Codebeispiel 87: MailType-Definition

Mit der MailType-Definition können Sie optionale Parameter angeben, die entsprechende Eingabefelder in der Detailansicht des entsprechenden E-Mail-Events zur Verfügung stellen. Diese Parameter sind:

Tabelle 28: Parameter bei der MailType-Definition

Parameter	Bedeutung
HasSubject="1"	Das Eingabefeld für die Betreffzeile wird angezeigt und der Händler kann einen entsprechenden Betreff eintragen.
HasAdditionalText="1"	Das Eingabefeld für zusätzlichen E-Mail-Text wird angezeigt und der Händler kann einen entsprechenden Inhalt eintragen.
HasHeader="1"	Das Eingabefeld für eine Kopfzeile über dem Standardtext wird angezeigt und der Händler kann einen entsprechenden Inhalt eintragen.
HasToField="1"	Das Eingabefeld für einen Empfänger wird angezeigt und der Händler kann einen entsprechenden Empfänger eintragen.

23.2 PageType anlegen und Template zuweisen

Die Darstellung der einzelnen E-Mails werden in HTML-Templates festgelegt, die über PageTypes mit dem MailType verbunden sind.

Für alle Mail-PageTypes existiert ein Basis-PageType *ShopMail*. In diesem sind die grundsätzlichen Bereiche *Page*, *Head*, *Body* und *Content* mit den Basistemplates definiert. Für die einzelnen MailTypes werden dann je nach Bedarf für die einzelnen Bereiche angepasste HTML-Templates zugewiesen.

Die PageTypes für MailTypes werden in der Datei *PageTypesMail.xml* im Verzeichnis */Database/XML* Ihrer Cartridge definiert.

Bei der PageType-Definition muss als Alias der Name verwendet wird, der auch im MailType als PageType-Bezeichner verwendet wird, vergleiche *Codebeispiel 87* und *Codebeispiel 88*.

```
<?xml version="1.0" encoding="UTF-8"?>
<epages>
  <Cartridge reference="1" Package="DE_EPAGES::ContactForm">
    <Class reference="1" Path="/Classes/Shop">
      <PageType Alias="Mail-ShopContactForm" Base="ShopMail" delete="1">
        <Template Name="Body"  FileName="Mail/Mail-ShopContactForm.Body.html" />
        <Template Name="Content"
                  FileName="Mail/Mail-ShopContactForm.Content.html" />
      </PageType>
    </Class>
  </Cartridge>
</epages>
```

Codebeispiel 88: Pagetype-Definition für einen MailType

Im Beispiel werden für die Bereiche *Body* und *Content* spezielle HTML-Templates zugewiesen. Die HTML-Dateien werden im Unterverzeichnis */Mail* des Template-Verzeichnisses Ihrer Cartridge abgelegt.

23.3 Funktion implementieren

Die Funktion, die das Versenden der Mail ausführt, wird als Perl-Modul im Verzeichnis */UI* Ihrer Cartridge implementiert.

Wichtig ist hier, dass im Quelltext der korrekte Name für den MailType übergeben wird. Zusätzlich müssen Sie darauf achten, dass die optionalen Parameter, die Sie in der MailType-Definition mit angegeben haben, im Code auch entsprechend verarbeitet werden.

23.4 Aktion registrieren und Permission definieren

Da das Versenden der E-Mail an eine Aktion gebunden ist, muss die Aktion auch in der Datenbank registriert werden, siehe dazu auch *Registrieren von Aktionen für Objekte*, *Seite 21*. Dazu nutzen Sie die Datei *Actions*.xml* Ihrer Cartridge oder legen sie dafür an. Die Aktion wird in der XML-Datei wie folgt registriert:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<epages>

  <Class reference="1" Path="/Classes/Shop">
    <Object Alias="Actions">
      <Action Alias="ViewContactForm" Package="DE_EPAGES::ContactForm::UI::Shop"
        FunctionName="ViewCached" delete="1" />
      <Action Alias="SendContactMail" Package="DE_EPAGES::ContactForm::UI::Shop"
        FunctionName="SendContactMail" delete="1" />
    </Object>
  </Class>

</epages>
```

Codebeispiel 89: Registrieren der Aktion

Dabei ist wiederum zu beachten, dass der Funktionsname im Perl-Modul mit *FunctionName* übereinstimmt.

In der Datei *Permissions*.xml* legen Sie fest, wer die Funktion ausführen darf, in unserem Fall jeder Kunde:

```
<?xml version="1.0" encoding="UTF-8"?>
<epages>

  <Role reference="1" Path="/Classes/Shop/Roles/Customer">
    <RoleAction Class="Shop"  Action="ViewContactForm"  delete="1" />
    <RoleAction Class="Shop"  Action="SendContactMail"  delete="1" />
  </Role>

</epages>
```

Codebeispiel 90: Vergabe der Berechtigung für die Aktion

Hier verwenden Sie für *Action* den Alias, unter dem Sie die Aktion registriert haben.

24. Erweiterung von Cross-Selling-Typen

Über Cross-Selling-Typen werden bestimmte Beziehungen bzw. Verknüpfungen zwischen Produkten hergestellt. In ePages sind die Typen *CrossSelling* für manuelles Cross-Selling, *Accessory* für Zubehör und *ProductComparison* für Produktvergleich implementiert.

Nachfolgend wird das prinzipielle Vorgehen bei der Anlage weiterer Cross-Selling-Typen am Beispiel des Typs *ReplacementParts* für Ersatzteile beschrieben.

Dabei werden nur die Besonderheiten bzw. wichtige Eigenschaften und Dateien erläutert, die grundlegenden Kenntnisse zum Anlegen und Inhalt von Cartridges sind in der vorangegangenen Kapiteln beschrieben und werden vorausgesetzt.

Für jeden Cross-Selling-Typ muss eine Tabelle angelegt werden und die Klasse *Product* ist um drei Attribute zu erweitern.

Der grundlegende Ablauf ist folgender:

- Tabelle definieren
- Klassen anlegen
- Produktattribute erweitern
- Pagetypes und Templates anlegen
- Aktionen registrieren/Funktionen implementieren

24.1 Tabelle definieren

Sie legen eine Tabelle an mit Bezug zum Namen des Cross-Selling Typs, den Sie einführen wollen, in unserem Beispiel *ReplacementPart*. Für diese Tabelle werden die drei Spalten *replacementpartid*, *productid* und *targetproductlid* definiert. Dabei ist die *productid* die ID des Produktes, dem das Ersatzteil zugeordnet wird und *targetproductid* die ID des Produktes, welches als Ersatzteil verwendet wird.

Die Tabelle wird über eine sql-Datei im Cartridge-Verzeichnis */Database/Sybase/Tables* angelegt. Ein mögliches Beispiel sehen Sie in *Codebeispiel 91*.

```
…
CREATE TABLE replacementpart (
    replacementpartid    int           not null
,   productid            int           not null
,   targetproductid      int           not null

,   constraint pk_replacementpart               primary key (replacementpartid)
,   constraint fk_replacementpart_product       foreign key (targetproductid)
        references product                          (productid)
,   constraint fk_replacementpart_product_1     foreign key (productid)
        references product                          (productid)
,   constraint fk_replacementpart_object        foreign key (replacementpartid)
        references object                           (objectid)
)
GO
…
```

Codebeispiel 91: Anlegen der Tabelle für neuen Cross-Selling Typ

Parallel dazu legen Sie im Cartridge-Verzeichnis */API/Object* ein Perl-Modul an, in welchem die Funktionen zur Bearbeitung der Datensätze in der Tabelle implementiert sind.

24.2 Klassen anlegen

Weiterhin müssen Sie eine neue Klasse mit den zugehörigen Attributen anlegen. Der neue Cross-Selling Typ basiert auf *Object* und wird in der Datei *AttributesReplacementPart.xml* im Cartridge-Verzeichnis */Database/XML* definiert, siehe *Codebeispiel 92*.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<epages>
  <Object reference="1" Path="/Classes">

    <Class Alias="ReplacementPart" Base="Object"
           Package="Training::ReplacementPart::API::Object::ReplacementPart"
           ExportLevel="1" delete="1">
      <AttributeValue Name="Name" Language="en" Value="ReplacementPart" />
      <AttributeValue Name="Description" Language="en"
                      Value="product replacement parts" />
      <Object Alias="Attributes">
        <Attribute Alias="Product" Type="Product" IsMandatory="1" IsCacheable="1"
                   IsExportable="1" ExportLevel="2" IsObject="1"
                   Packa-
ge="Training::ReplacementPart::API::Attributes::ReplacementPart"
                   Position="10" >
          <AttributeValue Name="Name" Language="en" Value="Product" />
          <AttributeValue Name="Description" Language="en"
                          Value="product " />
        </Attribute>
        <Attribute Alias="TargetProduct" Type="Product" IsMandatory="1"
                   IsCacheable="1" IsExportable="1" ExportLevel="2" IsObject="1"
                   Packa-
ge="Training::ReplacementPart::API::Attributes::ReplacementPart"
                   Position="20" >
          <AttributeValue Name="Name" Language="en" Value="TargetProduct" />
          <AttributeValue Name="Description" Language="en"
                          Value="Replacement part product " />
        </Attribute>
      </Object>
    </Class>

  </Object>
</epages>
```

Codebeispiel 92: Klassen-Definition für neuen Cross-Selling Typ

Die Funktionen zum Bearbeiten der Attribute implementieren Sie in einem entsprechenden Perl-Modul im Cartridge-Verzeichnis */API/Attributes*.

24.3 Produktattribute erweitern

Pro Cross-Selling Typ muss die Klasse *Product* um drei Attribute erweitert werden:

Tabelle 29: Zusatzattribute

Name	Beschreibung
‹crosssellingtypename›	Beinhaltet alle Produkte, die als Cross-Selling-Produkte einem Masterprodukt oder einem Produkt ohne Variationen direkt zugeordnet wurden
SubProduct‹crosssellingtypename›	Beinhaltet alle Produkte, die als Cross-Selling-Produkte einem Variationsprodukt direkt zugeordnet wurden
Visible‹crosssellingtypename›	Beinhaltet alle Produkte, die als Cross-Selling-Produkte einem Produkt zugeordnet wurden und im Shop sichtbar sind. Im Falle eines Variationsproduktes werden zuerst die sichtbaren Cross-Selling-Produkte des Masterproduktes angezeigt, gefolgt von den direkt zugeordneten Cross-Selling-Produkten.

Für unser Beispiel heißen die drei Attribute demnach:

- ReplacementParts
- SubProductReplacementParts
- VisibleReplacementParts

Definiert werden diese Attribute in der Datei *AttributesProduct.xml* im Cartridge-Verzeichnis */Database/XML*:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<epages>
  <Object reference="1" Path="/Classes">

    <Class reference="1" Alias="Product">
      <Object Alias="Attributes">
        <Attribute Alias="ReplacementParts" Type="ReplacementPart" IsArray="1"
                   IsCacheable="0" IsExportable="0" IsObject="1"
                   Package="Training::ReplacementPart::API::Attributes::Product"
>
          <AttributeValue Name="Name" Language="en" Value="ReplacementParts" />
        </Attribute>
        <Attribute Alias="SubProductReplacementParts" Type="ReplacementPart"
                   IsArray="1" IsCacheable="0" IsExportable="0" IsObject="1"
                   Package="Training::ReplacementPart::API::Attributes::Product"
>
          <AttributeValue Name="Name" Language="en" Va-
lue="VariationReplacementParts"
          />
        </Attribute>
        <Attribute Alias="VisibleReplacementParts" Type="ReplacementPart" IsAr-
ray="1"
                   IsCacheable="0" IsExportable="0" IsObject="1"
                   Package="Training::ReplacementPart::API::Attributes::Product"
>
          <AttributeValue Name="Name" Language="en" Value="Visible Replacement-
Parts"
          />
      </Object>
    </Class>

  </Object>
</epages>
```

Codebeispiel 93: Zusätzliche Produkt-Attribute für neuen Cross-Selling Typ

Die Zugriffsfunktionen auf diese Attribute implementieren Sie in einem entsprechendem Perl-Modul im Cartridge-Verzeichnis */API/Attributes*.

24.4 Pagetypes und Templates anlegen

Für die Darstellung des neuen Cross-Selling Typs sollten Sie neue PageTypes mit den entsprechenden Templates generieren, sowohl für das Backoffice als auch für die Storefront. Bezüglich Backoffice-Darstellung müssen Sie sich entscheiden, ob die Ersatzteile in einer eigenen Karteikarte oder zusammen mit den anderen Cross-Selling-Typen angezeigt und bearbeiten werden.

Der Vorteil der eigenen Karteikarte besteht in der Eigenständigkeit der Funktionalität auch bei Weitergabe der Cartridge an andere Nutzer.

Zeigen Sie die Ersatzteile zusammen mit den anderen Cross-Selling-Typen an, müssen Sie dafür ein Original-Template überschreiben. Problematisch wird dies, wenn Sie die Cartridge weitergeben und ein anderer

Nutzer hat ebenfalls dieses Template überschrieben. Hier kann es zu Konflikten kommen. Berücksichtigen Sie dies bei Ihrer Entscheidung.

In unserem Beispiel legen wir eine eigene Karteikarte an. Da die einzelnen Karteikarten für die Produktdetails unter Verwendung eines Menüs angezeigt werden, müssen Sie dieses Menü erweitern. Zur Arbeit mit Menüs lesen Sie *Dynamische Menüs*, *Seite 173*. Eine mögliche PageType-Definition sehen Sie in *Codebeispiel 94*.

```
<?xml version="1.0" encoding="UTF-8"?>
<epages>
  <!-- page types and templates -->
  <Cartridge reference="1" Package="Training::ReplacementPart">
    <Class reference="1" Path="/Classes/Product">

      <PageType reference="1" Alias="MBO-Product">
        <Menu reference="1" Template="Tabs">
          <Menu Template="Tab-ReplacementParts" URLAction="MBO-
ViewReplacementParts"
                Position="700" delete="1" />
        </Menu>
        <Template Name="Tab-ReplacementParts"
                  FileName="MBO/MBO-Product.Tab-ReplacementParts.html" />
      </PageType>

      <PageType Alias="MBO-ReplacementParts" Base="MBO-Product" delete="1">
        <Template Name="TabPage" FileName="MBO/MBO-ReplacementParts.TabPage.html"
/>
        <ViewAction URLAction="MBO-ViewReplacementParts" />
      </PageType>

    </Class>

  </Cartridge>
</epages>
```

Codebeispiel 94: PageType für Backoffice-Darstellung des neuen Cross-Selling Typ

Im Beispiel wird zum einen das Menü erweitert, welches die einzelnen Karteikarten anzeigt. Das Menü selbst ist im Pagetype *MBO-Product* definiert. Hier wird darauf referenziert. Dem Reiter der Karteikarte wird ein eigenes Template zugeordnet.

Der PageType *MBO-ReplacementParts* basiert auf *MBO-Product* und stellt ein eigenes Template für die Darstellung des Inhaltes der Karteikarte zur Verfügung, die in der zugeordneten HTML-Datei implementiert ist.

Dazu sind auch die entsprechenden ViewActions definiert.

Analog ist das Vorgehen für die Anzeige der Ersatzteile im Shop. Auch hier werden die Detaildaten eines Produktes mit Hilfe eines Menüs dargestellt. Daher müssen Sie auch dieses Menü nur erweitern, siehe *Codebeispiel 95*.

```
<?xml version="1.0" encoding="UTF-8"?>
<epages>
  <Cartridge reference="1" Package="Training::ReplacementPart">

    <Class reference="1" Path="/Classes/Product">
      <PageType reference="1" Alias="SF-Product">
        <Menu reference="1" Template="Content">
          <Menu Template="Content-ReplacementParts" Position="90" />
        </Menu>
        <Template Name="Content-ReplacementParts"
                  FileName="SF/SF-Product.Content-ReplacementParts.html" dele-
te="1"/>
      </PageType>

    </Class>

  </Cartridge>
</epages>
```

Codebeispiel 95: Erweitern des Menüs für die Shop-Darstellung des neuen Cross-Selling Typs

Wie die Ersatzteile im Shop angezeigt werden, bestimmen Sie im zugeordneten HTML-Template.

24.5 Aktionen registrieren/Funktionen implementieren

Die Aktionen, die der Händler im Backoffice bezüglich Ersatzteile ausführen kann, müssen Sie in der Datenbank registrieren und die dazugehörigen Funktion zur Verfügung stellen.

Für die Aktionen legen Sie im Verzeichnis */Database/XML* die Datei *Actions*.xml* an und registrieren die Aktionen analog *Codebeispiel 96*.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<epages>
  <Class reference="1" Path="/Classes/Product">
    <Object Alias="Actions">
…
      <Action Alias="RemoveReplacementPartProducts"
              Package="Training::ReplacementPart::UI::Product" delete="1" />
…
    </Object>
  </Class>
</epages>
```

Codebeispiel 96: Registrieren der Aktionen

Die dazugehörigen Funktionen werden in Perl-Modulen im Cartridge-Verzeichnis */UI* implementiert:

```
package Training::ReplacementPart::UI::Product;
use base qw( DE_EPAGES::Presentation::UI::Object );
…
sub RemoveReplacementPartProducts {
    my $self    = shift;
    my $Servlet = shift;

    my $Product = $Servlet->object;
    $self->DeleteListedObjects( $Servlet );
    $Product->folder('ReplacementParts')->renumberChildren;
    return;
}
…
```

Prüfen Sie weiterhin, ob folgende Daten bereitgestellt bzw. bearbeitet wurden oder noch zu bearbeiten sind:

- Festlegen der Abhängigkeiten von anderen Cartridges: *Dependencies.xml*
- Vergeben von entsprechenden Berechtigungen: *Permissions*.xml*
- Bei Bedarf Bereitstellen von Hooks
- Anwendung des Formhandling
- Im Falle einer Lokalisierung: Verwenden der Language Tags und Bereitstellen der Sprachdateien
- Beachten Sie die Referenzen zu anderen Objekten, z. B. dass diese bei Löschaktionen eventuell mit gelöscht werden müssen.

25. Anlegen von Shops per Web Service und Script

Neben dem Anlegen von Shops in der Business Administration ist es möglich, Shops per Web Service oder Script zu generieren. Zu Web Services für ePages lesen Sie auch *Web Services*, *Seite 107*.

Dies hat u. a. den Vorteil, Shops über externe Systeme oder zeitgesteuert anlegen zu lassen.

Dazu steht der Web Service *ShopConfigService* in der Cartridge *ShopConfiguration* zur Verfügung. Für den Aufruf der ePages-Web Services gibt es in der Cartridge *WebServices* einen Client, *Client.pm*, auf Perl-Basis. Diese Klasse verfügt über angepasste Fehlerbehandlungs-Funktionen. Verwenden Sie diesen Client, wenn Sie den Web Service aus einer ePages-Umgebung heraus aufrufen. Für den Aufruf aus anderen Systemen können Sie den *Service WebServiceClient.pm* benutzen. Dieser steht im ePages-Partnerweb zum Download bereit.

Die Methoden des *ShopConfigService*-Web Service sind in der dazu gehörigen WSDL-Datei mit allen zugehörigen Parametern definiert.

Den Web Service *ShopConfigService.pm* finden Sie in

```
%EPAGES_CARTRIDGES%/DE_EPAGES/ShopConfiguration/API/WebService/ ,
```

die dazugehörige WSDL-Datei *ShopConfigService.wsdl* in

```
%EPAGES_CARTRIDGES%/DE_EPAGES/ShopConfiguration/Data/Public/WSDL/
```

bzw. nach der Installation in

```
%EPAGES_SHARED%/WebRoot/Site/WSDL/ .
```

Die wichtigsten Funktionen dieses Web Service sind *create*, *update* und *delete*. Um einen Shop anzulegen, rufen Sie die Funktion *create* mit mindestens folgenden Parametern auf:

Tabelle 30: Parameter für *create*

Type	Bemerkung
Alias	Eindeutiger Shopname innerhalb des Provider
ShopType	Dieser Shoptyp muss vorhanden sein, damit der Shop alle entsprechenden Eigenschaften und Features zugewiesen bekommt.
Database	Datenbank, auf der der Shop angelegt werden soll, muss existieren
ShopAlias	Eindeutiger Shopname im Bezug auf die Datenbank
ImportFiles	Angabe aller Dateien, über die die entsprechenden Eigenschaften und Inhalte für den Shop bereitgestellt werden. Hier ist auch die xml-Datei anzugeben, welche für die Definition des verwendeten ShopTypes verwendet wurde. Vorraussetzung für den Import ist, dass der Applikationsserver auf die Importdateien zugreifen kann.

Diese Funktion entspricht dem Vorgang des Shop-Anlegens in der Business-Administration. Beim Anlegen über die Oberfläche wird allerdings *Alias = Shop*Alias gesetzt und es können keine zusätzlichen Import-Dateien angeben werden. Es wird nur die über den ShopType festgelegte xml-Datei verwendet.

Die Funktion *update* verlangt die gleichen Parameter wie *create*. Der Name der Datenbank kann nicht geändert werden, d. h. ein "Verschieben" des Shops von einer Datenbank auf eine andere ist so nicht möglich. Über Verwendung anderer Import-Dateien können z. B. anderssprachliche Inhalte in den Shop importiert werden.

Bei Ausführung der Funktion *delete* werden alle Shopdaten aus der Store-Datenbank gelöscht. Basisinformationen auf der Site-Datenbank bleiben erhalten und werden für die Shop-Historie verwendet. Sollen alle Daten eines Shops komplett aus dem System gelöscht werden, müssen Sie die Funktion *deleteShopRef* ausführen.

Zu beachten ist, dass der Service auf der Site läuft, d. h. der Proxy muss prinzipiell wie folgt aufgerufen werden:

```
http://<servername>/epages/Site.soap
```

Der Aufruf von ePages-Web Service ist in *Externe Clients für ePages 5-Web Services*, *Seite 110* beschrieben.

Die o. g. Funktionen lassen sich auch per Script ausführen. Die entsprechenden Scripte finden Sie in

```
%EPAGES_CARTRIDGES%/DE_EPAGES/ShopConfiguration/Scripts/
```

Durch die Ausführung der Scripte wird implizit die entsprechende Funktion des ShopConfigService ausgeführt. Im Folgenden ein Beispiel zum Anlegen eines Shops per Script:

```
perl createShop.pl -proxy http://localhost:80/epages/Site.soap
-wsuser /Providers/Distributor/Users/admin -passwd admin
-alias Store.TestShop -shoptype MerchantPro -storename Store -shopalias TestShop
%EPAGES_STORES%/Site/ShopImport/BusinessCard.xml
```

Dabei werden die Parameter für den Web Service explizit mit übergeben. Scripte können nur lokal gestartet werden, eignen sich aber für Testzwecke und zeitgesteuerte Ausführung.

Eine Besonderheit gibt es für das Script *deleteShop.pl.* Hier werden die Funktionen *delete* und *deleteShopRef* nacheinander ausgeführt, so dass alle Shop-Daten komplett gelöscht werden.

26. Patchen von Cartridges

Patchen ist das Anpassen von zugehörigen Daten an eine höhere Version einer Cartridge. Als Voraussetzung müssen vor Beginn des Patchens in der Cartridge alle zu ändernden Dateien gegen neue ausgetauscht und neue Dateien hinzugefügt sein.

Hinweis: Legen Sie vor Beginn eines Patches unbedingt ein Backup der aktuellen Version an.

Grundlage für das Patchen ist ein Framework, welches das Durchführen eines Patches in zwei Schritten unterstützt:

1. Aktualisieren bzw. Ändern der Datenbankstruktur
2. Aktualisieren der Daten auf Basis der neuen Datenbankstruktur

Um diese Vorgänge ausführen zu können, muss die Datei *Cartridge.pm* gemäß *Codebeispiel 97* erweitert werden:

```
…
sub new {
    my ($class, %options) = @_;

    my $self = __PACKAGE__->SUPER::new(
        %options,
        'CartridgeDirectory' => 'Training/PatchMe',
        'Version'            => '2.0',          # current version number
        'Patches'            => ['1.0','1.1'],  # list of version numbers that
can be
                                                   updated to current version
    );
    return bless $self, $class;
}
…
sub patchDBStructure_1_1 {
    my $self = shift;

    # get the curent database handle
    my $dbi=GetCurrentDBHandle();

    # change the database structure
#    $dbi->do( "ALTER TABLE xxx ADD yyy INTEGER DEFAULT 0 NOT NULL" );
    # update the data
#    $dbi->do( "UPDATE xxx SET yyy = zzz + 21" );

    return '2.0';
}
…
sub patch_1_0 {
    my $self = shift;

    # use API functions to migrate the data

    return '1.1';
}
…
sub patch_1_1 {
    my $self = shift;

    # use API functions to migrate the data

    return '2.0';
}
…
```

Codebeispiel 97: Anpassung in *Cartridge.pm* für das Patchen

In die Funktion *new* werden zwei neue Parameter aufgenommen, *Version* und *Patches*. Parameter *Version* gibt die Zielversion an, d. h. auf welche Version aktualisiert werden soll. Im Parameter *Patches* werden alle Versionen aufgelistet, von denen auf die Zielversion aktualisiert werden kann. Im obigen Beispiel heißt das, dass diese Cartridge von den Versionen 1.0 und 1.1 auf die Version 2.0 aktualisiert werden kann.

Die Funktion *patchDBStructure_** beinhaltet alle Methoden, um notwendige Änderungen an der Datenbank auszuführen.

Dabei gilt folgende Namenskonvention: An den Funktionsnamen *patchDBStructure_* wird die Nummer der Version angehängt, von welcher auf die Zielversion aktualisiert wird. Dabei sind Punkte durch Unterstriche zu ersetzen. Die Nummer der Zielversion wird in der *return*-Anweisung definiert. Gemäß Beispiel sind in der Funktion *patchDBStructure_1_1* alle Datenbankänderungen implementiert, die beim Wechsel von Version 1.1 auf 2.0 notwendig sind.

Sollten Datenbankänderungen beim Wechsel von 1.0 auf 1.1 erforderlich sein, müssen diese in einer Funktion *patchDBStructure_1_0* mit der *return*-Anweisung *return '1.1';* enthalten sein.

Solange nicht alle Datenbankänderungen ausgeführt sind, sollten Sie keine API-Funktionen aufrufen.

Die Funktionen *patch_** beinhalten alle Methoden, um die Daten entsprechend der geänderten Datenbankstruktur anzupassen, zu verschieben usw.

Auch hier gilt die Namenskonvention, dass die Nummer der Ausgangsversion an den Funktionsnamen angehängt wird, wobei Punkte durch Unterstriche ersetzt werden. Die *return*-Anweisung beinhaltet die Nummer der Zielversion.

Der Patch-Prozess wird gestartet nach Aufruf des Scriptes:

```
perl %EPAGES_CARTRIDGES%\DE_EPAGES\Installer\Scripts\patch.pl -storename Store
     <vendor>::<cartridgename>
```

Nach dem Start werden zuerst alle Funktionen *patchDBStructure_** ausgeführt. Im Anschluss werden die Funktionen *patch_** verarbeitet.

Sollten während des Prozesses Fehler auftreten, kann nach Beseitigung der Fehler das Script erneut gestartet werden. Bereits erfolgreich ausgeführte Funktionen werden nicht noch einmal verarbeitet. Der Vorgang beginnt dann mit der Ausführung der vorher fehlerhaften Funktion.

In der Store-Datenbank gibt es eine Tabelle *Cartridge*, in der alle installierten Cartridges aufgelistet sind. Für jede Cartridge gibt es u. a. die Attribute *dbstructureversion* und *version*. In *dbstructureversion* wird die neue Versionsnummer eingetragen, sobald die Funktionen *patchDBStructure_** fehlerfrei ausgeführt wurden. In *version* wird die neue Versionsnummer eingetragen, sobald die Funktionen *patch_** fehlerfrei ausgeführt wurden. Durch einen Vergleich der beiden Einträge können Sie kontrollieren, ob der Patch erfolgreich verlaufen ist. Dieser Vergleich gibt keine Auskunft über die Richtigkeit der Daten nach dem Patch.

Man kann von jeder Version auf die Zielversion patchen, empfohlen wird aber die schrittweise Ausführung von einer Version auf die jeweils nächste.

Zusammenfassend die Vorgehensweise beim Patchen von Cartridges zum Test des Patches:

1. Anfertigen einer Sicherheitskopie der aktuellen Version und der aktuellen Datenbank und aller anderen Daten und Dateien, die durch den Patch beeinflusst werden können.
2. Erhöhen der Versionsnummer in der Funktion *new()* in *Cartridge.pm*
3. Aufnahme der aktuellen Versionsnummer in die Liste der zu patchenden Versionen
4. Implementieren der Funktionen *patchDBStructure_** und *patch_** je nach Notwendigkeit
5. Starten *patch.pl*
6. Aktualisieren der Cartridgeliste der betreffenden Datenbank in der Technischen Administration
7. Bereinigung der Cartridge-Verzeichnisse, u. a. nicht mehr benötigte Dateien löschen.

Vorgehensweise beim Patchen von Cartridges auf einem Live-System:

1. Anfertigen einer Sicherheitskopie der aktuellen Version und der aktuellen Datenbank und aller anderen Daten und Dateien, die durch den Patch beeinflusst werden können.
2. Kopieren aller neuen und geänderten Cartridge-Dateien in das Verzeichnis mit der aktuellen Version
3. Starten *patch.pl*
4. Aktualisieren der Cartridgeliste der betreffenden Datenbank in der Technischen Administration
5. Bereinigung der Cartridge-Verzeichnisse, u. a. nicht mehr benötigte Dateien löschen.

27. Integration der eigenen Online-Hilfe

Die Funktion zur Anzeige einer Hilfe wird in einem oder mehreren Includes eines Templates integriert. So finden Sie auf der Seite zur Anzeige der Produkte im MBO einen Hilfe-Aufruf im Suchbereich und eine Hilfe-Funktion in der jeweils aktiven Karteikarte.

Welche Hilfe angezeigt wird, definieren Sie zusammen mit der ViewAction zur Anzeige des Bereiches, für den die Hilfe abgezeigt werden soll.

Um eine Hilfe-Seite für eine Seite einzurichten, müssen Sie wie folgt vorgehen:

1. HTML-Seite mit Hilfe-Inhalt generieren und an der entsprechenden Stelle im Datei-Verzeichnis bereitstellen
2. Verknüpfung der Hilfe-Seite mit der entsprechenden ViewAction
3. Implementieren des entsprechenden Anzeige-Codes im Template

27.1 Bereitstellen der Hilfe-Seite

Die Dateien der Standard-Online-Hilfe sind im Verzeichnis

```
%EPAGES_WEBROOT%/Doc/Help/<language>/<administration>
```

abgelegt. Die Hilfeseiten für die deutsche Händler-Administration finden Sie z. B. in

```
%EPAGES_WEBROOT%/Doc/Help/de/MBO
```

Parallel zu den Verzeichnissen für die jeweilige Administration können Sie Ihre Hilfe-Seiten in einem separaten Verzeichnis ablegen und zur Verfügung stellen. Sprachabhängige Hilfe-Seiten verteilen Sie auf die entsprechenden Verzeichnisse für die einzelnen Sprachen.

Stellen Sie Ihre Funktion und die dazugehörige Hilfe in einer Cartridge zur Verfügung, werden die Hilfe-Seiten im Laufe des Installationsprozesses, siehe *Installieren - nmake*, *Seite 89*, an die richtigen Stellen kopiert. Voraussetzung dafür ist die Bereitstellung der korrekten Struktur in Ihrer Cartridge.

Legen Sie dazu in Ihrer Cartridge folgendes Unterverzeichnis an:

```
/Data/WebRoot/Doc/Help/<language>/<cartridgename>
```

In dieses Verzeichnis legen Sie die HTML-Dateien für Ihre Hilfe-Seiten. Aus Gründen der Übersichtlichkeit empfehlen wir die Ablage der Bilder in einem eigenen Unterverzeichnis.

Für eine deutsche Hilfe-Seite für Ihre Cartridge müsste die HTML-Datei in Ihrem Cartridge-Verzeichnis wie folgt abgelegt werden:

```
/Data/WebRoot/Doc/Help/de/MyCartridge/MyOwnHelp.html
```

Hinweis: Sie können von Ihrer Hilfe-Seite auf die ePages-Standard-Hilfe verlinken. Integrieren Sie dazu den folgenden Link in Ihre Seite, z. B.: *Hilfe*.

27.2 Zuweisen zur ViewAction

Die Hilfe für eine Seite müssen Sie mit der Aktion zur Anzeige dieser Seite verknüpfen. Dies geschieht in der Datei *Actions*.xml* Ihrer Cartridge.

Die Syntax sehen Sie im folgenden Beispiel:

```
…
<Action Alias="MBO-ViewMyCartridgeGeneral" Packa-
ge="DE_EPAGES::MyCartridge::UI::Shop"
        FunctionName="View" delete="1" >
   <AttributeValue Name="HelpFileTopic" Value="MyCartridge/MyOwnHelp.html" />
</Action>
…
```

Codebeispiel 98: Zuweisen der Hilfe zur ViewAction

27.3 Anzeige-Code im Template

Bei der Erstellung neuer Karteikarten im MBO auf Basis der vorhandenen ist die Hilfe automatisch integriert.

Um die Hilfe-Seite in eigenen Templates aufrufen zu können, muss ein entsprechendes Element in das Template integriert werden. Als Standardelement im ePages-System wird ein Buch-Symbol verwendet.

Diesem Icon ist folgende Funktion zugeordnet:

```
…
<a href="#WebRoot/Doc/Help/{LanguageID}/#HelpFileTopic"
   onclick="openWindow(this.href,'','HelpWindow'); return false;">
   <img id="HelpButton" class="HelpButton ClickIcon"
      src="#StoreRoot/BO/icons/dialog_ico_s_helpbook.gif"
      alt="{ContextHelp}" title="{ContextHelp}"
      onmouseover="this.src='#StoreRoot/BO/icons/dialog_ico_s_helpbook_open.gif'"
      onmouseout="this.src='#StoreRoot/BO/icons/dialog_ico_s_helpbook.gif'"
   />
</a>
…
```

Codebeispiel 99: Funktion zum Anzeigen der passenden Hilfe-Seite

Beispiele, wie dieser Code im Template angewandt wird, finden Sie in den Dateien *Backoffice.Tabs.html*, (Anwendung auf Karteikarten) und *MBO-ProductManager.Toolbar.html* (Anwendung im Suchbereich). Diese Dateien finden Sie in Ihrer ePages-Installation.

28. Dynamische Menüs

Dynamische Menüs werden an Stellen eingesetzt, an denen verschiedene Funktionalitäten flexibel angezeigt werden sollen. Zu den Vorteilen dieses Vorgehens gehören die Erweiterbarkeit der Menüs durch Cartridges, die Lokalisierung der einzelnen Menüeinträge oder die Vererbung der Menüs auf abgeleitete PageTypes.

Typische Menüs sind die Hauptnavigationsleiste, die Kontextbar, Kontextmenü oder die Karteikarten zur Anzeige von Objektdetails.

Ein Menü wird in einem PageType definiert. Innerhalb dieser Definition oder über andere Cartridges werden die Menüeinträge zugeordnet. Der Inhalt eines Menüeintrages ist in einem Template beschrieben. Die Zuordnungen werden über die Menü- und Template-Namen hergestellt.

Anhand der Hauptnavigationsleiste als Anwendungsbeispiel können Sie die allgemeine Vorgehensweise nachvollziehen:

Im PageType *MBO* wird ein Menü mit Einträgen definiert und ein Template für die Menü-Darstellung zugewiesen, siehe *Codebeispiel 100*.

```
…
 <PageType Alias="MBO" Base="Backoffice" delete="1">
   <Menu Template="Menu" Position="0" delete="1">
     <Menu Template="Menu-Marketing" Class="Shop" URLAction="MBO-
ViewMarketingGeneral"
           Position="60" />
     <Menu Template="Menu-Settings" Class="Shop" URLAction="MBO-
ViewStatusGeneral"
           Position="70" />
   </Menu>
…
   <Template Name="Menu" FileName="MBO/MBO.Menu.html" />
   <Template Name="Menu-Marketing" FileName="MBO/MBO.Menu-Marketing.html" />
   <Template Name="Menu-Settings" FileName="MBO/MBO.Menu-Settings.html" />
 </PageType>
…
```

Codebeispiel 100: Menüdefinition aus der Cartridge *Presentation*

Im Tag *Menu* wird das Menü mit einem eindeutigen Bezeichner angelegt. Innerhalb des Menu-Tags platzieren Sie die einzelnen Menüeinträge.

Zu jedem Eintrag muss ein Bezeichner und optional die entsprechende Aktion angegeben werden. Die Aktion beschreiben Sie durch den Namen der Aktion selbst, und das Objekt, auf welches diese Aktion registriert ist.

Über Position bestimmen Sie die Reihenfolge, in der die Einträge angezeigt werden.

Für das Menü in seiner Gesamtheit sowie für jeden einzelnen Menüeintrag muss ein Template festgelegt werden, in dem Sie Inhalt und Darstellung beschreiben. Diese Zuweisung ist wie jede andere Bereichszuweisung im PageType über *‹Template Name=... />* zu treffen. Dabei muss der Name des Bereiches dem Bezeichner des Menüs bzw. des Menüeintrages entsprechen.

Für unser Beispiel bedeutet dies:

- Es ist ein Menü definiert mit dem Bezeichner *Menu*
- Das Menü beinhaltet die Einträge *Menu-Marketing* und *Menu-Settings* mit Angabe der Aktion und Position
- Der Menüeintrag *Menu-Marketing* wird durch das Template *MBO.Menu-Marketing.html* dargestellt

- Der Menüeintrag *Menu-Settings* wird durch das Template *MBO.Menu-Settings.html* dargestellt
- Das Menu selbst wird durch das Template *MBO.Menu.html* dargestellt

Der Name eines Menü-Templates muss innerhalb eines PageTypes eindeutig sein. Deshalb wird empfohlen, eine Namenskonvention, bestehend aus *‹menüname›-‹eintragname›*, einzuhalten.

Das Menü selbst wird im Template häufig mit Hilfe einer Loop angezeigt:

```
<table class="Menu" width="100%" cellspacing="0" cellpadding="0" summary="">
  <tr>
    <td>
      <p>
        <span>
          #BLOCK("MENU","Menu")
            <img src="#StoreRoot/BO/icons/manager_img_separator.gif" alt="" />
            #INCLUDE(#Template)
          #ENDBLOCK
          <img src="#StoreRoot/BO/icons/manager_img_separator.gif" alt="" />
        </span>
      </p>
    </td>
…
  </tr>
</table>
```

Codebeispiel 101: Menüanzeige

Ein Template für einen Menüeintrag sehen Sie in *Codebeispiel 102*:

```
<a href="?ViewAction=#URLAction&amp;ObjectID=#Shop.ID"
   Onmouseover = "changeImage(
   'mana-
ger_marketing','#StoreRoot/BO/icons/mbo_manager_img_marketing_mouseover.gif'
   )"
   onmouseout= "changeImage(
   'mana-
ger_marketing','#StoreRoot/BO/icons/mbo_manager_img_marketing_unselected.gif'
   )">
  <img src="#StoreRoot/BO/icons/mbo_manager_img_marketing_unselected.gif" alt=""
  id="manager_marketing" /> {Marketing}
</a>
```

Codebeispiel 102: HTML-Code für Menüeintrag

Die Hauptnavigationsleiste ist ein typisches Beispiel für ein Menü, welches durch andere Cartridges erweitert wird. So liefert z. B. die Cartridge *Customer* den Eintrag *Kunden*, über den der Hauptnavigationspunkt *Kunden* aufgerufen wird.

Die entsprechende Menü-Erweiterung sehen Sie in *Codebeispiel 103*

```
…
 <PageType reference="1" Alias="MBO">
   <Menu reference="1" Template="Menu">
     <Menu Template="Menu-Customers" Class="Shop" URLAction="MBO-SearchCustomers"
           Position="20" delete="1" />
     </Menu>
   <Template Name="Menu-Customers"
             FileName="MBO/MBO.Menu-Customers.html" delete="1" />
 </PageType>
…
```

Codebeispiel 103: Menü-Erweiterung

Sie müssen auf den PageType referenzieren, in dem das Menü angelegt wurde. Dann geben Sie, ebenfalls per Referenz, den Bezeichner des Menüs an, welches Sie erweitern wollen.

Den neuen Eintrag definieren Sie genauso wie die Einträge beim Anlegen eines Menüs.

Diesem Eintrag weisen Sie unter Verwendung des Bezeichners ein Template mit den Darstellungsanweisungen zu.

Nach Installation der Cartridge ist der neue Eintrag im Menü vorhanden.

Ein weiteres Beispiel zur Erweiterung vom Menüs finden Sie im *AWB 7: Neue Stapelverarbeitungsaktion im MBO*, *Seite 201*.

29. Warenkorb-Template und LineItems

Das Warenkorb-Template bietet die Grundstruktur für die Webseiten der einzelnen Schritte von der Warenkorbansicht bis hin zur Anzeige der Bestellbestätigung. Dieses komplexe Template ist mit Hilfe von Menüs aufgebaut, um optimale Voraussetzungen für Erweiterung und Anpassung zu schaffen.

Abbildung 31 zeigt die prinzipiellen Aufbau des Warenkorb-Templates, der im PageType *SF-Basket* definiert ist.

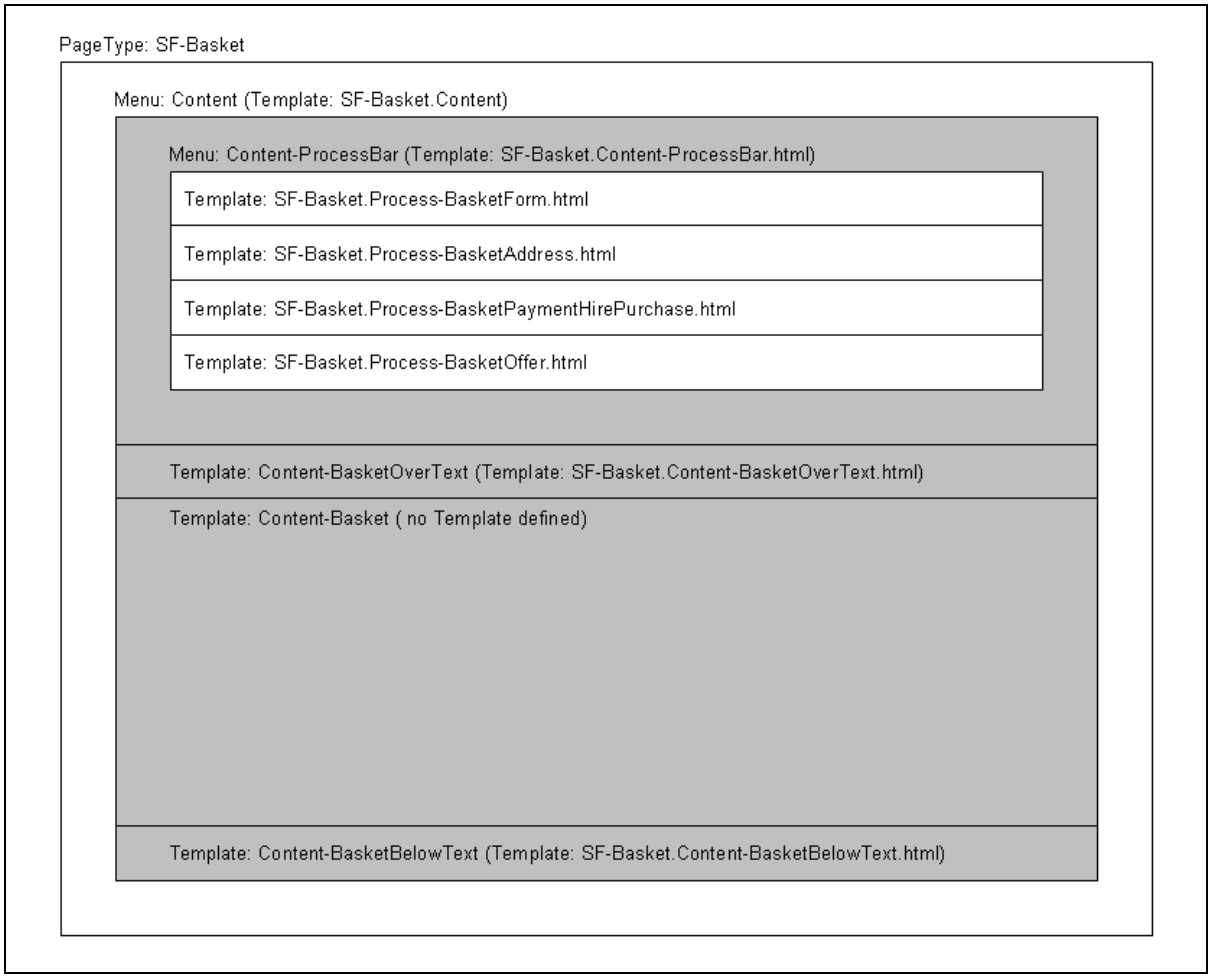

Abbildung 31: Prinzipieller Aufbau des Warenkorb-Templates

Dabei ist zu beachten, dass *SF-Basket* den Arbeitsbereich der Seite definiert, die umliegenden Seitenbereiche, wie Kopfzeile oder linker bzw. rechter Bereich, werden durch die übergeordneten PageTypes dargestellt.

Für das Menü *Content-Basket* ist kein Standard-Template bereitgestellt, hier wird das dazugehörige Template für jeden Schritt des Bestellprozesses speziell bereitgestellt.

Dafür wird für jeden der Schritte ein eigener PageType definiert, der auf *SF-Basket* basiert. In diesem werden die entsprechenden Strukturänderungen festgelegt oder spezielle Templates zugewiesen. In *Codebeispiel 104* sehen Sie den PageType *SF-BasketOffer* für den Schritt, in dem die Bestellübersicht angezeigt wird

```
…
<PageType Alias="SF-BasketOffer" Base="SF-Basket" delete="1">
   <Template Name="Content-Basket" FileName="SF/SF-BasketOffer.Content-
Basket.html" />
   <ViewAction URLAction="ViewOffer" />
</PageType>
…
```

Codebeispiel 104: Template-Definition für Menü Content-Basket

In diesem Beispiel wird dem Menü *Content-Basket* das Template zugewiesen, in welchem die Anzeige der Bestellübersicht implementiert ist.

Daraus ergeben sich prinzipiell folgende Möglichkeiten, den Bestellprozess anzupassen:

- Überschreiben bestehender Templates
- Hinzufügen neuer Schritte auf Basis neuer PageTypes.
- Erweitern der Menüstruktur des Warenkorb-Templates

Beim Hinzufügen zusätzlicher Schritte müssen Sie beachten, dass Sie diese Schritte auch in die Fortschrittsanzeige aufnehmen, d. h. das Menü *Content-ProcessBar* erweitern.

Ein Beispiel für die Erweiterung der Menüstruktur ist die Cartridge *Coupon*. Der Bereich zur Eingabe von Gutschein-Codes soll im Warenkorb angezeigt werden. Um diesen Bereich in das Template einzufügen, wird das Menü *Content* um einen Eintrag erweitert. Im *Codebeispiel 105* sehen Sie die entsprechende PageType-Definition:

```
…
<PageType Alias="SF-BasketForm" reference="1">
  <Menu reference="1" Template="Content" >
    <Menu Template="Content-RedeemCoupon"   Position="50" delete="1"/>
  </Menu>
  <Template Name="Content-RedeemCoupon"
            FileName="SF/SF-BasketForm.Content-RedeemCoupon.html" delete="1" />
  <Template Name="ContentLineLineItemCoupon"
            FileName="SF/SF-BasketForm.ContentLineLineItemCoupon.html" delete="1"
/>
</PageType>
…
```

Codebeispiel 105: PageType mit Menü-Erweiterung

Es wird auf den PageType *SF-BasketForm* und auf das dort definierte Menü *Content* referenziert. Für das Menü wird ein zusätzlicher Eintrag mit der entsprechenden Template-Zuweisung erstellt. Dieses Template beschreibt die Darstellung des zusätzlichen Seitenbereiches und dessen Funktionalität.

Der Warenkorb selbst, mit der Liste der bestellten Produkte, den ausgewählten Versand- und Zahlungsmethoden sowie verschiedenen Preisnachlässen, wird unter der Verwendung von *LineItems* dargestellt:

29.1 LineItems

LineItems sind die einzelnen Positionen in der Tabelle eines Warenkorbs oder einer Bestellung mit den zugehörigen Dokumenten. Sie beinhalten die Informationen zu den einzelnen Produkten, Zahlungsmethoden, Rabatten usw., aus denen sich der Wert des Warenkorbs berechnet.

Die LineItems werden sowohl im Shop als auch im Backoffice angezeigt und zwar in der Regel an folgenden Stellen:

Tabelle 31: Anzeige von LineItems

Bereich	Anzeige
Shop	Warenkorbformular
	Bestellzusammenfassung
	Auftragsbestätigung (Shopansicht und Druckansicht)
	Mein Konto: Bestellung (Shopansicht und Druckansicht)
	Minibasket (Navigationselement)
Backoffice	Bestellung - Anzeige und Bearbeiten
	Dokumente (Rechnung, Lieferschein, UPS-Lieferschein, Gutschrift) - Anzeige und Bearbeiten
Mail	Auftragsbestätigung
	Statusänderungen

In Abhängigkeit von Funktion und Inhalt müssen die LineItems teilweise unterschiedlich dargestellt werden. Aus diesem Grund kann jedem LineItem-Typ ein eigenes Template zur Verfügung gestellt werden.

Voraussetzung ist die Definitionen eines entsprechenden LineItem-Typs. Ausgehend von einem "Basis"-LineItem der Klasse *LineItem* kann für jeden Positionstyp ein eigener LineItem-Typ abgeleitet werden. Für die gebräuchlichsten Positionen sind die LineItems definiert, die hierarchisch aufgebaut sind und voneinander erben. Die Übersicht über diese Standard-LineItems können Sie sich mit Hilfe der Diagnostics-Cartridge anzeigen lassen:

- Unter *All Classes* wird die gesamte Liste angezeigt
- Unter *All Classes* ›› *LineItem* sehen Sie die erste Hierarchieebene und die einzelnen Zweige der Struktur verfolgen

Ist für einen LineItem-Typ kein eigenes Template definiert, wird das Template des übergeordneten LineItem-Typs verwendet. Zur Verwendung von Templates übergeordneter Klassen siehe *Objektmethode template*, *Seite 47*.

Entsprechend der funktionellen Darstellungen ergeben sich folgende Ansichten für LineItems:

Tabelle 32: Ansichten für LineItems

Ansicht	Template-Typ	Funktion
LineItems	ContentLine	Standardansicht in Warenkörben und Bestellungen
EditLineItems	EditContentLine	Standardansicht in Bestellungen mit Bearbeitungsfunktion
MiniLineItems	MiniContentLine	Anzeige im Navigationselement Warenkorb, klein
SupplyLineItems	SupplyContentLine	Standardansicht in Lieferscheinen
EditSupplyLineItems	EditSupplyContentLine	Standardansicht in Lieferscheinen mit Bearbeitungsfunktion
NegLineItems	NegContentLine	Standardansicht in Gutschriften (negative Werte)
NegEditLineItems	NegEditContentLine	Standardansicht in Gutschriften (negative Werte) mit Bearbeitungsfunktion

Nach diesem Prinzip können Sie jederzeit bei Bedarf eigene LineItem-Typen bzw. eigene Ansichten definieren.

Am Beispiel der Anzeige einer Bestellung im MBO in normaler Ansicht und in Bearbeitungsfunktion soll das Prinzip nochmals verdeutlicht werden. Für die einfache Anzeige der Warenkorbpositionen in einer Bestellung im MBO wird der Template-Typ *ContentLine* verwendet:

```
…
#WITH(#LineItemContainer)
…
  #LOOP(#LineItems)
    #INCLUDE("ContentLine")
  #ENDLOOP
…
  #LOOP(#SalesDiscounts)
    #INCLUDE("ContentLine")
  #ENDLOOP
  #LOOP(#Discounts)
    #INCLUDE("ContentLine")
  #ENDLOOP
  #WITH(#Shipping)
    #INCLUDE("ContentLine")
  #ENDWITH
  #WITH(#Payment)
    #INCLUDE("ContentLine")
  #ENDWITH
  #IF(#DEFINED(#PaymentDiscount))#WITH(#PaymentDiscount)
    #INCLUDE("ContentLine")
  #ENDWITH#ENDIF
…
  #LOOP(#Taxes)
…
  #ENDLOOP
…
#ENDWITH
```

Codebeispiel 106: Darstellung der LineItems in der Bestellansicht

Welches Template für die einzelnen LineItem-Typen konkret verwendet wird, ist im PageType definiert, der für die Darstellung der speziellen Seite festgelegt ist.

Dabei wird nach folgender Template-Definition gesucht: *ContentLine[‹lineitemtyp›]*. Für die Anzeige der Versandmethode wird z. B. nach dem Template *ContentLineLineItemShipping* gesucht und die dazugehörige HTML-Datei für die Darstellung verwendet.

Ist kein Template *ContentLineLineItemShipping* definiert, sucht dad System entsprechende der Klassenstruktur nach einem Template Namens *ContentLineLineItem*. Ist auch dieses nicht definiert, greift das System auf das allgemeine Template *ContentLine* zurück.

Hier wird deutlich, wie die Bildungsvorschrift für Template-Namen dazu verwendet werden kann, pro Klasse spezielle Templates zu definieren und anzuwenden. Fehlt die spezielle Definition, wird das allgemeinere Template der übergeordneten Klasse verarbeitet.

Wird für unser Beispiel die Bestellung im Bearbeitungsmodus angezeigt, wird anstelle des Template-Typs *ContentLine* der Template-Typ *EditContentLine* verwendet:

```
…
#WITH(#LineItemContainer)
…
  #LOOP(#LineItems)
    #INCLUDE("EditContentLine")
  #ENDLOOP
…
  #LOOP(#SalesDiscounts)
    #INCLUDE("EditContentLine")
  #ENDLOOP
  #LOOP(#Discounts)
    #INCLUDE("EditContentLine")
  #ENDLOOP
  #WITH(#Shipping)
    #INCLUDE("EditContentLine")
  #ENDWITH
  #WITH(#Payment)
    #INCLUDE("EditContentLine")
  #ENDWITH
  #IF(#DEFINED(#PaymentDiscount))#WITH(#PaymentDiscount)
    #INCLUDE("EditContentLine")
  #ENDWITH#ENDIF
…
  #LOOP(#Taxes)
…
  #ENDLOOP
…
#ENDWITH
…
```

Codebeispiel 107: Darstellung der LineItems in der Bearbeitungsansicht für Bestellungen

Dieser stellt die entsprechenden Funktionen zum Editieren der Daten bereit. Für die Darstellung der Versandmethode im Bearbeitungsfall wird diesmal gemäß Bildungsvorschrift das Template *EditContentLineLIneItemShipping* mit der zugeordneten HTML-Datei verwendet.

Anhang C: Anwendungsbeispiele (AWB)

30. AWB 1: Einbinden einer eigenen Stylesheet-Datei

In diesem Beispiel soll eine eigene statische Stylesheet-Datei (css) in das System eingebunden werden. Die entsprechenden Dateien finden Sie den mitgelieferten Cartridge-Beispielen, im Verzeichnis */E1_MyStaticStyle*.

Die Stylesheet-Datei wird über einen entsprechenden Eintrag im Header des entsprechenden Templates eingebunden. Gehen Sie wie folgt vor:

1. Cartridge anlegen

   Legen Sie eine Cartridge mit dem Name MyStaticStyle an. Grundlagen zu Anlegen eine Cartridge lesen Sie in *Cartridges*, *Seite 85*.

2. Stylesheet-Datei anlegen

   Legen Sie die Stylesheet-Datei in folgendem Cartridge-Verzeichnis an:

   ```
   /Data/Public/Shops/DemoShop/Styles/MyStyle.css
   ```

   Und fügen Sie folgenden Code ein:

   ```
   .NewsList a {
     color: red !important;
   }
   ```

   Codebeispiel 108: Code in *Mystyle.css*

3. Link zur Stylesheet-Datei registrieren

   Der Verweis auf die zusätzliche Stylesheet-Datei muss in den Header der Webseiten eingebunden werden. Der Header wird über einen PageType definiert. Diese Definition müssen Sie erweitern. Legen Sie dazu im Cartridge-Verzeichnis

   ```
   /Database/XML/
   ```

   die Datei *PageTypesSF.xml* mit folgendem Inhalt an:

   ```
   <?xml version="1.0" encoding="UTF-8"?>
   <epages>
     <!-- page types and templates -->
     <Cartridge reference="1" Package="Training::MyStaticStyle">
       <Class reference="1" Path="/Classes/Shop">
         <PageType reference="1"  Alias="SF">
           <Menu Template="Head" reference="1">
             <Menu Template="Head-MyStyle" Position="5" delete="1" />
           </Menu>
           <Template Name="Head-MyStyle"  FileName="SF/SF.Head-MyStyle.html"
             delete="1" />
         </PageType>
       </Class>
     </Cartridge>
   </epages>
   ```

   Codebeispiel 109: Code in der *PageTypesSF.xml*

Dadurch wird der Header um den Bereich *Head-MyStyle* erweitert und das Template *SF.Head-MyStyle.html* eingebunden. In diesem Template steht der Code für den Verweis auf die konkrete Stylesheet-Datei:

```
<link href="#Shop.WebPath/Styles/MyStyle.css"
  rel="stylesheet" type="text/css" />
```

Codebeispiel 110: Verweis auf die Stylesheet-Datei

Die Datei *SF.Head-MyStyle.html* wird angelegt im Cartridge-Verzeichnis

```
/Templates/SF/
```

Die Grundlagen hierfür lesen Sie in *PageType-Konzept*, *Seite 41* und *Dynamische Menüs*, *Seite 173*.

4. Cartridge installieren

   Installieren Sie die Cartridge wie in *Installieren - nmake*, *Seite 89* beschrieben. Das Ergebnis sehen Sie auf der Startseite des DemoShops. Die Überschriften der News-Artikel in der News-Liste werden rot geschrieben.

31. AWB 2: Erweitern des Storefrontstyles

Neben den statischen Stylessheet-Erweiterungen können Sie eigene Stylesheet-Angaben auch so gestalten, dass sie auf Änderungen aus dem Design-Tool reagieren. Dazu müssen Sie TLE in Ihren Stylesheet-Angaben verwenden. Grundlagen hierfür lesen Sie in *Templates*, *Seite 33*, *TLE*, *Seite 63*, und *Cartridges*, *Seite 85*.

Im folgenden Beispiel wird gezeigt, wie Sie eigene Stylesheet-Angaben so einbinden, dass diese über das Design-Tool mit geändert werden können. Grundlage dafür ist die Datei *SF-Style.StyleExtension-PartnerStyles.css*, die für solche Zwecke zur Verfügung steht. Die Originaldatei befindet sich im Verzeichnis:

```
%EPAGES_CARTRDIGES%/DE_EPAGES/Design/Templates/SF
```

Diese Datei müssen Sie in einer eigenen Cartridge überladen. Gehen Sie wie folgt for:

1. Cartridge anlegen

   Legen Sie eine Cartridge mit dem Namen *MyDynamicStyle* an. Lesen Sie dazu *Anlegen einer Cartridge-Struktur*, *Seite 87*.

2. Überladungs-Template anlegen

   Legen Sie in der Cartridge folgendes Verzeichnis an:

   ```
   /Data/Private/Templates/DE_EPAGES/Design/Templates/SF
   ```

   Legen Sie in diesem Verzeichnis die Datei *SF-Style.StyleExtension-PartnerStyles.css* an und tragen folgenden Code ein:

   ```
   .NewsList a {
      color: #ContentHotDealPriceColor[color] !important;
   }
   ```

   Codebeispiel 111: Inhalt *SF-Style.StyleExtension-PartnerStyles.css*

   Analog zum vorigen Beispiel wird die Farbe für die Überschriften der News-Artikel in der News-Liste auf der Startseite des DemoShops geändert. In diesem Fall aber hängt die Farbe vom Wert der TLE *ContentHotDealPriceColor* ab. Diese Farbe können Sie im Design-Tool einstellen.

3. Cartridge installieren

   Installieren Sie die Cartridge wie in *Installieren - nmake*, *Seite 89* beschrieben.

4. Style neu generieren

   Rufen Sie im MBO das Design-Tool für den aktuellen Style auf. Ändern Sie unter **Anpassen** ›› **Inhaltsbereich** ›› *Preisangaben* den Wert für das Feld *Startseite* und speichern Sie. Dadurch wird die *StorefrontStyle.css* neu generiert. Die Datei liegt im Verzeichnis:

   ```
   %EPAGES_WEBROOT/Stores/Shops/DemoShop/Styles/<currentStyle>
   ```

   Öffnen Sie diese Datei. Im unteren Teil gibt es die Sektion *StyleExtension-PartnerStyles*. In dieser Sektion finden Sie die Stylesheet-Angaben aus *Codebeispiel 111*, mit dem aktuellen Wert, den Sie mit dem Design-Tool gesetzt haben. Die Wirkung sehen Sie auf der Startseite. Sollten keine Änderungen zu sehen sein, löschen Sie alle Caches.

32. AWB 3: Änderungen im Template

Inhalt dieses Beispiels ist die Überladung der Startseite des DemoShops und die Verwendung von TLE in Templates. Grundlagen dafür lesen Sie in *Templates*, *Seite 33*, *TLE*, *Seite 63*, und *Cartridges*, *Seite 85*.

Sie müssen das Template feststellen, dessen Inhalt Sie ändern müssen und dieses überladen. Gehen Sie wie folgt vor:

1. Template identifizieren

   Stellen Sie fest, welches Template den Inhaltsbereich der Startseite anzeigt. Aktivieren Sie dazu das Debugging, siehe Template-Debugging, Seite 38 und lassen Sie den Quelltext der Startseite anzeigen. Sie erkennen Cartridge-Pfad und Namen des gesuchten Templates, in unserem Fall *SF-Shop.Content.html*.

2. Cartridge anlegen

   Legen Sie eine Cartridge mit dem Namen *MyHomePage* an. Lesen Sie dazu *Anlegen einer Cartridge-Struktur*, *Seite 87*.

3. Überladungs-Template anlegen

   Legen Sie in der Cartridge folgendes Verzeichnis an:

   ```
   /Data/Private/Templates/DE_EPAGES/Catalog/Templates/SF
   ```

   Legen Sie in diesem Verzeichnis die Datei *SF-Shop.Content.html* mit folgendem Inhalt an:

```
<!-- Section 1 Simple TLE -->
<h1>Hello World</h1>
#Shop.NameOrAlias

<div class="Separator"></div>

<!-- Section 2: LOOP main categories -->
<ul>
#LOOP(#Shop.Categories.VisibleSubCategories)
  <li><a href="?ObjectPath=#Path" >#NameOrAlias</a></li>
#ENDLOOP
</ul>

<div class="Separator"></div>

<!-- Section 3. LOOP main categories / highlight "Tents" -->
<ul>
#LOOP(#Shop.Categories.VisibleSubCategories)
  <li><a href="?ObjectPath=#Path"
         #IF(#Alias EQ "Tents") style="font-weight:bold" #ENDIF
      >#NameOrAlias</a></li>
#ENDLOOP
</ul>

<div class="Separator"></div>

<!-- Section 4. Change Object Context / LOOP main categories -->
#WITH(#Shop.Categories)
  <h1>#NameOrAlias</h1>
  <ul>
  #LOOP(#VisibleSubCategories)
    <li><a href="?ObjectPath=#Path" >#NameOrAlias</a></li>
  #ENDLOOP
  </ul>
#ENDWITH

<div class="Separator"></div>

<!-- Section 5. Variables and Calculation -->
#LOCAL("ValueA",10)
#LOCAL("ValueB",20)
  #ValueA + #ValueB =  #CALCULATE(#ValueA + #ValueB)<br />
  #SET("ValueB",25)
  #ValueA + #ValueB =  #CALCULATE(#ValueA + #ValueB)<br />
#ENDLOCAL
#ENDLOCAL
```

Codebeispiel 112: Beispielinhalt für die *SF-Style.StyleExtension-PartnerStyles.css*

Section 1 zeigt die einfache Anzeige von TLE-Variablen.

Section 2 zeigt die Verwendung der TLE-Anweisung *LOOP* am Beispiel der Auflistung von Kategorien.

Section 3 zeigt die Anwendung einer *IF*-Anweisung in einer *LOOP*.

Section 4 zeigt die Verwendung einer WITH-Anweisung. Durch diese wird ein bestimmter Objekt-Kontext gesetzt. Die nachfolgend Verwendeten TLE-Variablen beziehen sich dann auf dieses Kontext.

Section 5 zeigt die Verwendung von TLE-Variablen bei Berechnung und die Wirkungsweise der *LOCAL*-Anweisung.

Erläuterungen zu den einzelnen TLE-Variablen und Anweisungen lesen Sie in *TLE*, *Seite 63*.

4. Cartridge installieren

Installieren Sie die Cartridge wie in *Installieren - nmake*, *Seite 89* beschrieben und rufen Sie die Startseite des DemoShops auf.

33. AWB 4: Anpassung Backoffice-Design (Branding)

Die Anpassung des Backoffice-Designs ist eine oft gestellte Anforderung der Nutzer. Wir zeigen den prinzipiellen Weg zur Änderung des Designs durch Anwendung von Stylesheets und Überlagerung. Im Beispiel soll für die Änderung des MBO-Backoffice-Designs lokale Überlagerung genutzt werden. Falls Sie Ihre Designlösung installierbar in einer eigenen Cartridge kapseln wollen, lesen Sie Kapitel *Cartridges*, *Seite 85*.

Wie schon weiter oben erwähnt, liegt die entsprechende Stylesheet-Datei im Verzeichnis:

```
%EPAGES_WEBROOT%/Store/BO/BackofficeStyle.css
```

Im Unterverzeichnis */icons* liegen die dazugehörigen Bilder. Die Verwendung dieses Stylessheets wird in einer HTML-Datei definiert:

```
%EPAGES_CARTRIDGES%/DE_EPAGES/Presentation/Templates/Backoffice.Style.html
```

In dieser Datei legen Sie fest, welche Stylesheets für die Darstellung des Händler-Backoffice verwendet werden, siehe *Codebeispiel 113* oder Originaldatei im angegebenen Verzeichnis.

```
<link href="#StoreRoot/BO/BackofficeStyle.css" rel="stylesheet" type="text/css"
/>
```

Codebeispiel 113: Festlegung der Styles für das Backoffice

Für ein neues Design können Sie nun die *BackofficeStyle.css* ändern oder Ihre Änderungen in einer eigenen .css-Datei bereitstellen, die nur diese Änderungen enthält. Wir empfehlen das Anlegen einer neuen Datei. Legen Sie also eine neue *css*-Datei, z. B. *NewBackofficeStyle.css* im gleichen Verzeichnis an. Falls Sie eigene neue Icons verwenden wollen, stellen Sie diese im Unterverzeichnis */icons* zur Datei *BackofficeStyle.css* bereit:

```
%EPAGES_WEBROOT%/Store/BO/icons
```

Hinweis: Falls Sie versehentlich die .css-Datei oder die Icons überschreiben, finden Sie die Originaldateien immer wieder in den Original-Cartridges. Überschreiben Sie nicht die Dateien in den originalen Cartridge-Verzeichnissen.

Im nächsten Schritt müssen Sie Ihre neue Datei mit den Style-Änderungen dem System bekannt machen. Dazu müssen Sie die Datei *Backoffice.Style.html* anpassen. Da die Originaldatei aber nicht geändert werden soll, legen Sie im "Überladungs-Verzeichnis" eine Datei gleichen Namens an, in der Sie die Veränderung erfassen:

Kopieren Sie die Datei *Backoffice.Style.html* in das Verzeichnis:

```
%EPAGES_STORES%/Store/Templates/DE_EPAGES/Presentation/Templates
```

Zur Überladung lesen Sie *Überladung von Templates*, *Seite 38*. Öffnen Sie diese Datei und erweitern Sie den Quelltext wie in *Codebeispiel 114*.

```
<link href="#StoreRoot/BO/BackofficeStyle.css" rel="stylesheet" type="text/css"
/>
<link href="#StoreRoot/BO/NewBackofficeStyle.css" rel="stylesheet" ty-
pe="text/css" />
```

Codebeispiel 114: Einbindung der Style-Änderungen

Wenn Sie nun das Händler-Backoffice aufrufen, sollten das geänderte Design angezeigt werden. In *Abbildung 32* und *Abbildung 33* sehen Sie den Vergleich für ein Beispiel.

Abbildung 32: Standard-Backoffice-Design im Original

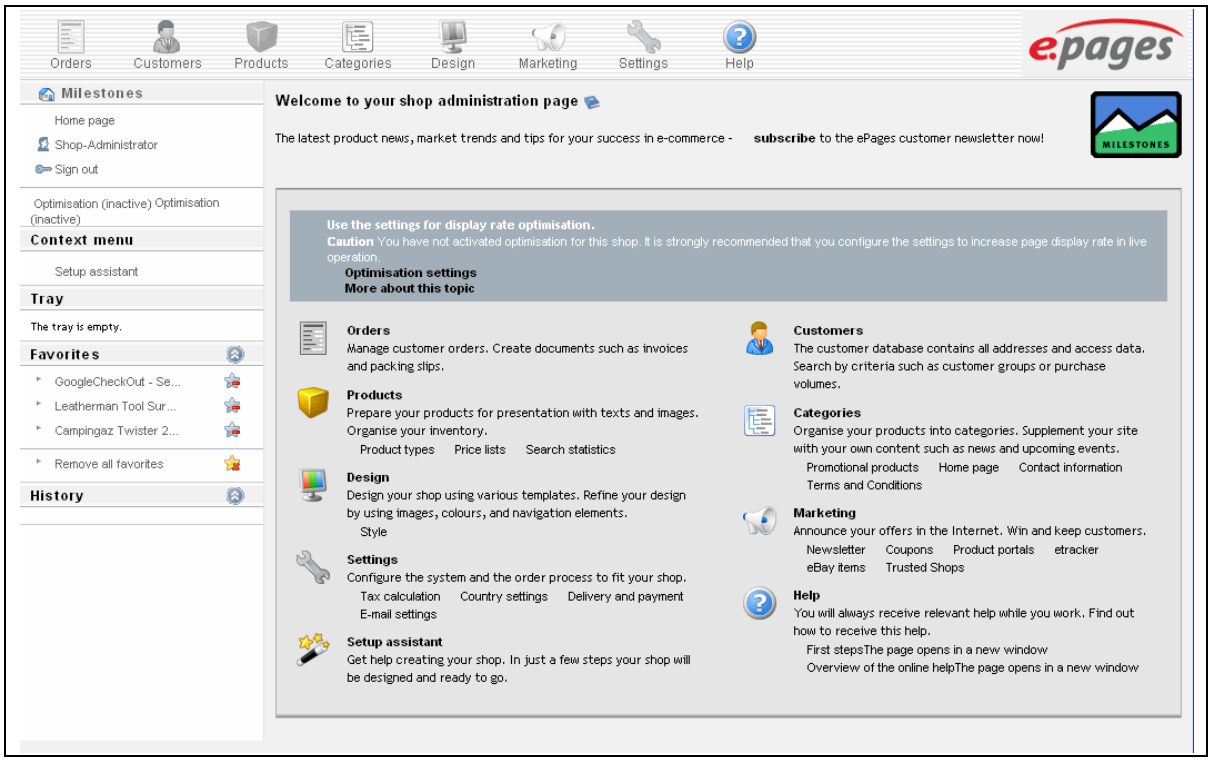

Abbildung 33: Backoffice nach beschriebener Design-Änderung

Der WYSIWIG-Editor *TinyMCE* als plattformunanabhängige web-basierte HTML-Applikation ist in das ePages-System integriert. Hier sind die Designdefinitionen auf verschiedene html-, js- und css-Dateien verteilt. Um hier Designanpassungen vorzunehmen, müssen Sie die entsprechenden Dateien im Verzeichnis *%EPAGES_WEBROOT%/Store/tinymce* suchen und ändern. Diese Änderungen können dann aber bei Upgrades wieder überschrieben werden.

34. AWB 5: Deaktivieren des Design-Tools

Ziel dieses Beispiels ist es, die Händleradministration soweit zu verändern, dass die Händler keinen Zugriff mehr auf Design-Tool und Einrichtungsassistenten haben, so dass sie das Design ihres Online-Shops nicht ändern können. Ein Weg dazu ist die Überlagerung bestimmter Templates. Grundlagen dafür lesen Sie in *Templates*, *Seite 33*, *TLE*, *Seite 63*, und *Cartridges*, *Seite 85*.

Der Aufruf von Design-Tool und Einrichtungsassistent erfolgt über Links in Templates. Diese Links müssen identifiziert und aus dem Template entfernt werden.

Über folgende Links kann der Händler auf die Gestaltungswerkzeuge zugreifen:

- Links zum Einrichtungsassistenten im Kontextmenu
- Links zum Einrichtungsassistenten und zur *Gestaltung* in der Funktionsübersicht der Startseite
- Link zur *Gestaltung* in der Hauptnavigationsleiste
- Links zur *Gestaltung* in der Sektion *Verwandte Themen* auf verschiedene Seiten

Um das Design-Tool zu deaktivieren, gehen Sie wie folgt vor:

1. Template identifizieren

   Sie müssen feststellen, durch welches Template, welche Daten auf der Webseite angezeigt werden. Dazu aktivieren Sie die Debug-Informationen im Quelltext, siehe *Template-Debugging*, *Seite 38*. Für unser Beispiel gilt:

   - Der Link zum Einrichtungsassistenten ist der einzige Eintrag in der Kontextbox. Deshalb soll die gesamte Kontextbox nicht mit angezeigt werden. Suchen Sie nach dem Template, welches die Boxen in der linken Navigationsleiste anzeigt.
   - Aus der Hauptnavigationsleiste ist der Punkt *Gestaltung* zu entfernen. Suchen Sie nach dem Template, welches die Hauptnavigationsleiste darstellt.
   - Die Funktionsübersicht auf der Startseite beinhaltet je einen Link zum Einrichtungsassistenten und zur *Gestaltung*. Diese Links müssen entfernt werden.
   - Wenn bei den *Verwandten Themen* der Link in die Navigation der einzige Eintrag ist, darf der gesamte Eintrag nicht angezeigt werden. Anderenfalls ist der Punkt mit dem Link zu löschen. Suchen Sie nach dem Template, welches die *Verwandten Themen* darstellt.

   Rufen Sie die Händleradministration auf und lassen sich die Funktionen im Browser anzeigen. Im Quelltext der Webseite lesen Sie aus der Include-Anzeige den Template-Namen mit Angabe der Cartridge ab. Zusätzlich wird die entsprechende Aktion angezeigt. Damit können Sie alle Templates suchen, in denen diese Aktion noch ausgeführt wird. So ermitteln Sie alle notwendigen Templates, die Sie bearbeiten müssen.

2. Cartridge anlegen

   Legen Sie eine Cartridge mit dem Namen *HideDesign* an. Lesen Sie dazu *Anlegen einer Cartridge-Struktur*, *Seite 87*.

3. Überladungs-Template anlegen

   Kopieren Sie die identifizierten Templates aus den Original-Verzeichnissen der Installation in die entsprechenden Verzeichnisse der Cartridge. So kopieren Sie z. B. die Datei *Backoffice.ContextBar.html* von

   ```
   %EPAFES_CARTRIDGES%/Cartridges/DE_EPAGES/Presentation/Templates/
   ```

   in das Cartridge-Verzeichnis

```
/Data/Private/Templates/DE_EPAGES/Presentation/Templates/
```

Die kopierten Templates bearbeiten Sie so, dass die o. g. Funktionen nicht mehr sichtbar sind.
Im Original der Datei *Backoffice.ContextBar.html* werden alle notwendigen Boxen für die linke Navigationsleiste der Händler-Administration angezeigt, siehe *Codebeispiel 115*.

```
#BLOCK("MENU","ContextBar")
  #INCLUDE(#Template)
#ENDBLOCK
```

Codebeispiel 115: Template zur Anzeige der Boxen in der linken Navigationsleite MBO

Den Code müssen Sie nun so ändern, dass die Kontextbox mit dem Link zum Einrichtungsassistenten nicht mit angezeigt wird, siehe *Codebeispiel 116*.

```
#BLOCK("MENU","ContextBar")
 #IF( NOT (
            (
               (#VIEWACTION.Alias EQ "MBO-ViewUserSettings")
               OR
               (#VIEWACTION.Alias EQ "MBO-ViewMBO")
            )
            AND
               (#Template.Name EQ "ContextMenuBox")
         )
    )
  #INCLUDE(#Template)
 #ENDIF
#ENDBLOCK
```

Codebeispiel 116: Ausblendung der Kontextbox

4. Cartridge installieren

   Installieren Sie die Cartridge wie in *Installieren - nmake*, *Seite 89* beschrieben. Löschen Sie die Caches, entfernen Sie die betreffenden ctmpl-Dateien und starten Sie den ePages-Service neu.

Die Dateien hierfür finden Sie im Beispiel *E5_HideDesign* der mitgelieferten Beispiel-Cartridges. In der Cartridge sind die Templates zum Ausblenden der Kontextbox und des Hauptmenüpunktes *Design* enthalten. Die Templates zum Ausblenden der entsprechenden Related Topic müssen Sie bei Bedarf selbst identifizieren, in die Cartridge kopieren und ändern. Hier sind die entsprechenden Links einfach auszukommentieren.

35. AWB 6: Design-Änderungen über PageTypes

Schwerpunkt dieses Beispiels ist die Arbeit mit PageTypes. Dabei beschränken wir uns auf Design-Änderungen, um dieses Cartridge-Beispiel nicht unnötig komplex zu gestalten. Grundlagen dafür lesen Sie in *Templates*, *Seite 33*, *PageType-Konzept*, *Seite 41* und *Cartridges*, *Seite 85*.

Das Ändern von bestehenden bzw. das Anlegen neuer PageTypes muss immer in eigenen Cartridges vorgenommen werden.

In diesem Beispiel soll die Anzeige von redaktionellen Seiten (Artikeln) geändert werden. Ausgangpunkt ist die Artikelanzeige im DemoShop, siehe *Abbildung 34*.

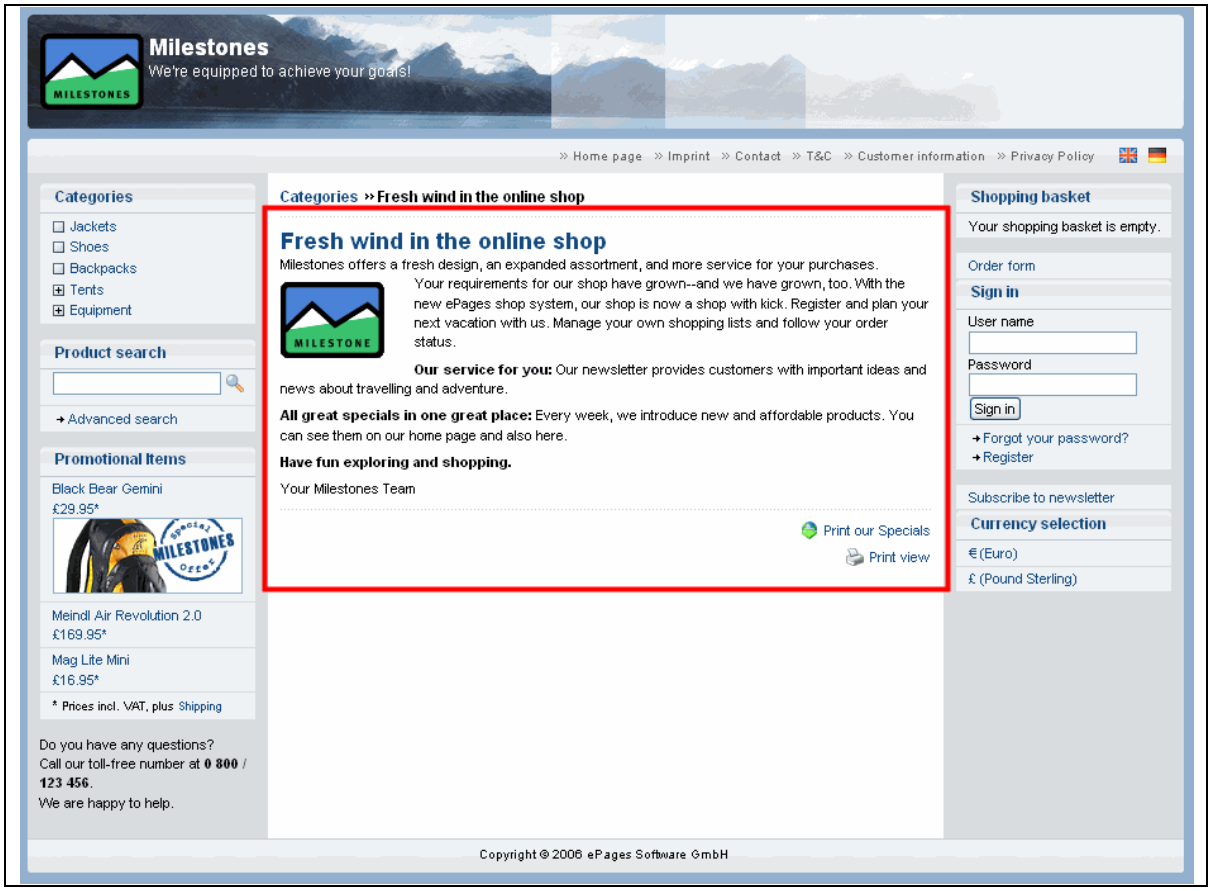

Abbildung 34: Artikelansicht als Ausgangspunkt für das Beispiel

Dabei soll das Template zur Artikelanzeige so geändert werden, dass der Inhalt über zwei INCLUDES eingebunden wird. Das eine INCLUDE stellt alle textrelevanten Daten bereit. Das andere INCLUDE bindet die Funktionen zum Drucken und zur Anzeige der Anlage ein. Um das Ergebnis anschaulicher zu gestalten, sollen die Funktionen über dem Artikelinhalt angeordnet werden und das Bild unter dem Text.

1. Template und PageType identifizieren

   Stellen Sie fest, mit welchem Template Artikel angezeigt werden und in welchem PageType die Zuweisung erfolgt. Mit Hilfe der Debug-Informationen erkennen Sie, dass der Inhalt eines Artikels durch das Template

   ```
   %EPAGES_CARTRIDGES%/DE_EPAGES/Content/Templates/SF/SF-Article.Content.html
   ```

   dargestellt wird.

Nun müssen Sie herausfinden, in welchem Pagetype dieses Template zugewiesen wird. Template und PageType, in dem das Template zugewiesen wird, liegen prinzipiell in einer Cartridge. Die PageTypes für die Cartridge *Content* sind im Verzeichnis */Database/XML/* in der Datei *PageTypesSF.xml* definiert. Die Templatezuweisung sehen Sie im Codebeispiel 117.

```
<?xml version="1.0" encoding="UTF-8"?>
<epages>
  <!-- page types and templates -->
  <Cartridge reference="1" Package="DE_EPAGES::Content">
…
    <Class reference="1" Path="/Classes/Article">
      <PageType Alias="SF-Article" Base="SF" delete="1">
        <Template Name="Content" FileName="SF/SF-Article.Content.html" />
        <ViewAction URLAction="View" />
      </PageType>
      <PageType Alias="SF-ArticlePrint" Base="SF-Article" delete="1">
        <Menu Template="Head" Base="Head" Position="0">
          <Menu Template="Head-PrintContentStyle" Position="40" />
        </Menu>
        <ViewAction URLAction="ViewPrint" />
      </PageType>
    </Class>
…
```

Codebeispiel 117: *Content*-Definition im PageType *SF*

Der PageType *SF-Article* basiert auf den PageType *SF* und überschreibt den dort definierten Bereich *Content* mit dem genannten Template.
Nachdem Template und PageType bekannt sind, sollen folgende Aufgaben umgesetzt werden:

- Ändern des Templates *SF-Article.Content.html*, so dass der Inhalt über INCLUDES eingebunden wird und Bereitstellen als Überlagerungstemplate
- Bereitstellen der INCLUDE-Dateien
- Erweiterung des Pagetypes

2. Cartridge anlegen

   Legen Sie eine Cartridge mit dem Namen *MyArticleDesign* an. Lesen Sie dazu *Anlegen einer Cartridge-Struktur, Seite 87*.

3. Überladungs-Template anlegen

   Legen Sie in der Cartridge folgendes Verzeichnis an:

   ```
   /Data/Private/Templates/DE_EPAGES/Content/Templates/SF
   ```

   Legen Sie in diesem Verzeichnis die Datei *SF-Article.Content.html* mit folgendem Inhalt an:

```
#INCLUDE("Content-PrintButton")
<h3>
  #LOCAL("LastObjectID",#ID)
    #LOOP(#PathFromSite)
      #IF(#ID NNE #LastObjectID)
        <a class="BreadcrumbItem"
href="?ObjectPath=#Path[url]">#NameOrAlias</a>
      #ENDIF
    #ENDLOOP
  #ENDLOCAL
  <span class="BreadcrumbLastItem">#NameOrAlias</span>
</h3>
<div class="Article">

  <div class="Separator"></div>

  #INCLUDE("Article_AttachmentSection")

  <div class="Separator"></div>

  #INCLUDE("Article_Content")

</div>
```

Codebeispiel 118: Änderung im Template zur *Content*-Beschreibung

Den Quelltext für die beiden INCLUDES stellen Sie in zwei neuen Dateien bereit:

- *SF-Article.Article_Content.html* und
- *SF-Article.Article_AttachmentSection.html*.

Da Sie mit diesen Dateien keine Original-Templates überladen, sondern neue Templates einführen, werden diese in der Cartridge im Verzeichnis für cartridge-spezifische Storefront-Templates angelegt:

```
/Templates/SF
```

Den Quelltext für die INCLUDE-Templates übernehmen Sie der Einfachheit halber aus dem Original-Template. Ändern Sie den Quelltext in *SF-Article.Article_Content.html* so, dass das Bild unterhalb des Textes angezeigt wird.

4. PageType erweitern

   Durch die Einführung der INCLUDES muss auch der betreffende PageType *SF-Article* geändert werden. Die beiden INCLUDES müssen im PageType als neue Bereiche definiert und die entsprechenden Templates müssen zugeordnet werden.
   Diese Änderung nehmen Sie **nicht** in der PageType-Definition in der Cartridge *Content* vor. Legen Sie dazu im Verzeichnis */Database/XML* der Cartridge *MyArticleDesign* die Datei *PageTypes.xml* an, siehe *Codebeispiel 119*.

```
<?xml version="1.0" encoding="UTF-8"?>
<epages>
  <!-- page types and templates -->
  <Cartridge reference="1" Package="Training::MyArticleDesign">
    <Class reference="1" Path="/Classes/Article">
      <PageType reference="1" Alias="SF-Article" >
        <Template Name="Article_Content"
          FileName="SF/SF-Article.Article_Content.html" delete="1"/>
        <Template Name="Article_AttachmentSection"
          FileName="SF/SF-Article.Article_AttachmentSection.html" delete="1"/>
      </PageType>
    </Class>
  </Cartridge>
</epages>
```

Codebeispiel 119: PageType-Definition für Cartridge *MyArticleDesign*

Über *Cartridge reference...* geben Sie an, zu welcher Cartridge die PageType-Definition gehört.

Im Tag *Class* stellen Sie über *reference...* den Bezug zu dem Objekt her, dem der PageType zugeordnet wird. Dies können Sie aus der Originaldefinition des PageTypes ablesen.

Der Ausdruck *reference="1"* legt fest, dass mit den folgenden Anweisungen ein bestehender PageType erweitert wird. Der zu erweiternde PageType wird im *Alias...* angegeben.

Im PageType-Element werden nun die beiden Bereiche *Article_Content* und *Article_AttachmentSection* definiert und die entsprechenden Templates zugeordnet.

Hier wird auch der Zusammenhang deutlich zwischen der Bereichsdefinition im PageType und der Verwendung im Template. Die Namen der Bereiche werden als Parameter für die INCLUDES verwendet.

5. Abhängigkeiten definieren

   Die grundlegenden Dateien sind damit erstellt. Damit die Cartridge funktioniert, muss noch eine Datei erstellt werden, deren Funktion an anderer Stelle, siehe *Import-Dateien*, *Seite 119,* detaillierter erklärt wird, um hier nicht von der Zielstellung des Beispiels abzulenken. Legen Sie die Datei an der entsprechenden Stelle an oder kopieren diese aus den Beispielquelltexten. Diese Datei wird im selben Verzeichnis angelegt wie *PageTypes.xml*:

   ```
   Dependencies.xml
   ```

   In der Datei wird festgelegt, von welchen anderen Cartridges die Cartridge Funktionen nutzt.

   ```
   <?xml version="1.0" encoding="UTF-8"?>
   <epages>
     <Dependency Package="DE_EPAGES::Design" />
   </epages>
   ```

   Codebeispiel 120: Festlegen der Abhängigkeit zu anderen Cartridges

6. Cartridge installieren

   Installieren Sie die Cartridge wie in *Installieren - nmake*, *Seite 89* beschrieben.

Nachdem der Prozess beendet ist, lassen Sie sich im Shop einen Artikel anzeigen. Die Änderungen im Template sollten in der Anzeige sichtbar sein.

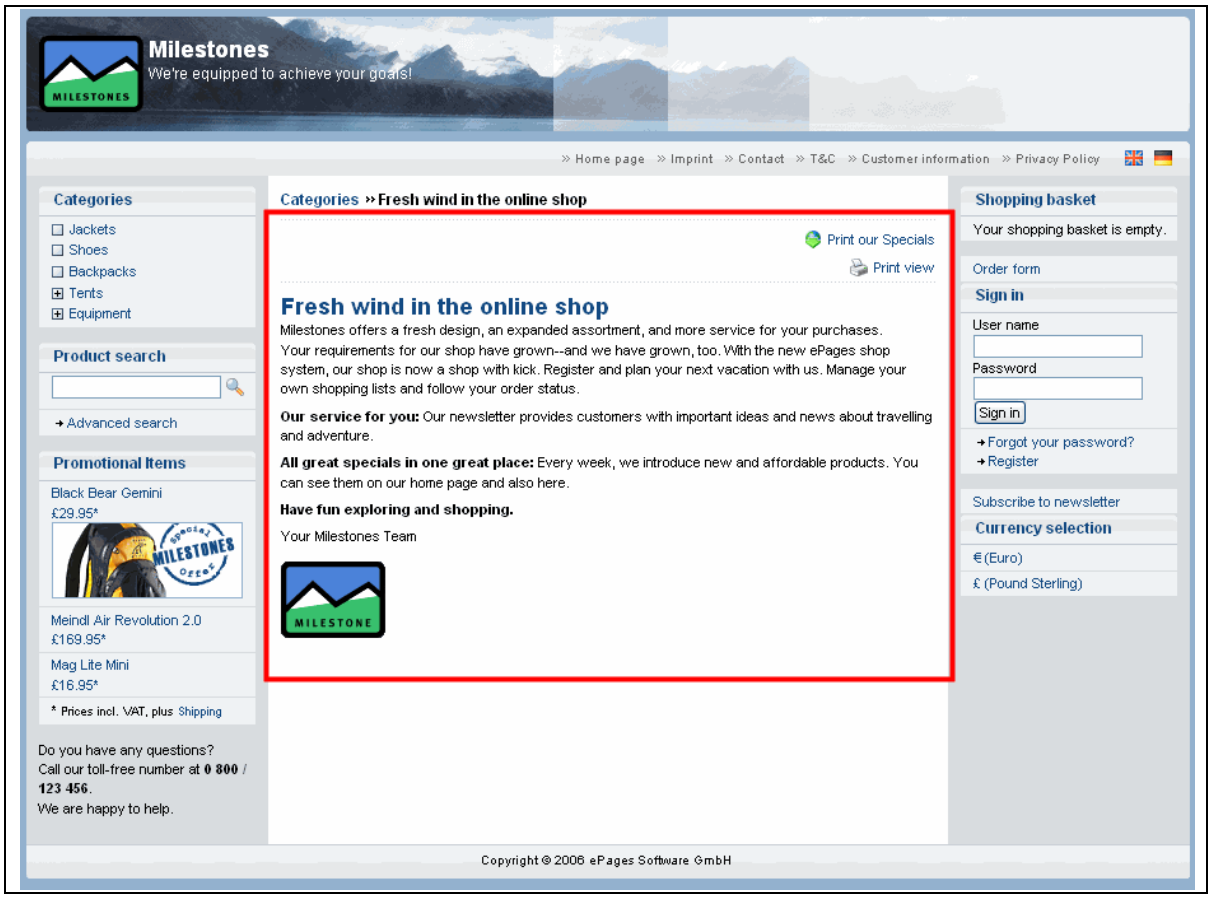

Abbildung 35: Geänderte Artikelansicht

Hinweis: Sollte die Änderung nicht gleich sichtbar sein, löschen Sie die entsprechenden ctmpl-Dateien im Verzeichnis */STATIC*.

36. AWB 7: Neue Stapelverarbeitungsaktion im MBO

Ausgehend von den Kenntnissen, die Sie sich bis hierhin angeeignet haben, werden Sie jetzt mit Hilfe einer Cartridge die bestehende Funktionalität der Anwendung im Händler-Backoffice erweitern. Grundlagen dafür lesen Sie in *Templates*, *Seite 33*, *PageType-Konzept*, *Seite 41* und *Cartridges*, *Seite 85*.

Im MBO gibt es für Tabellen oft Stapelverarbeitungsaktionen, mit denen man mehrere Einträge auf einmal bearbeiten kann, siehe *Abbildung 36*.

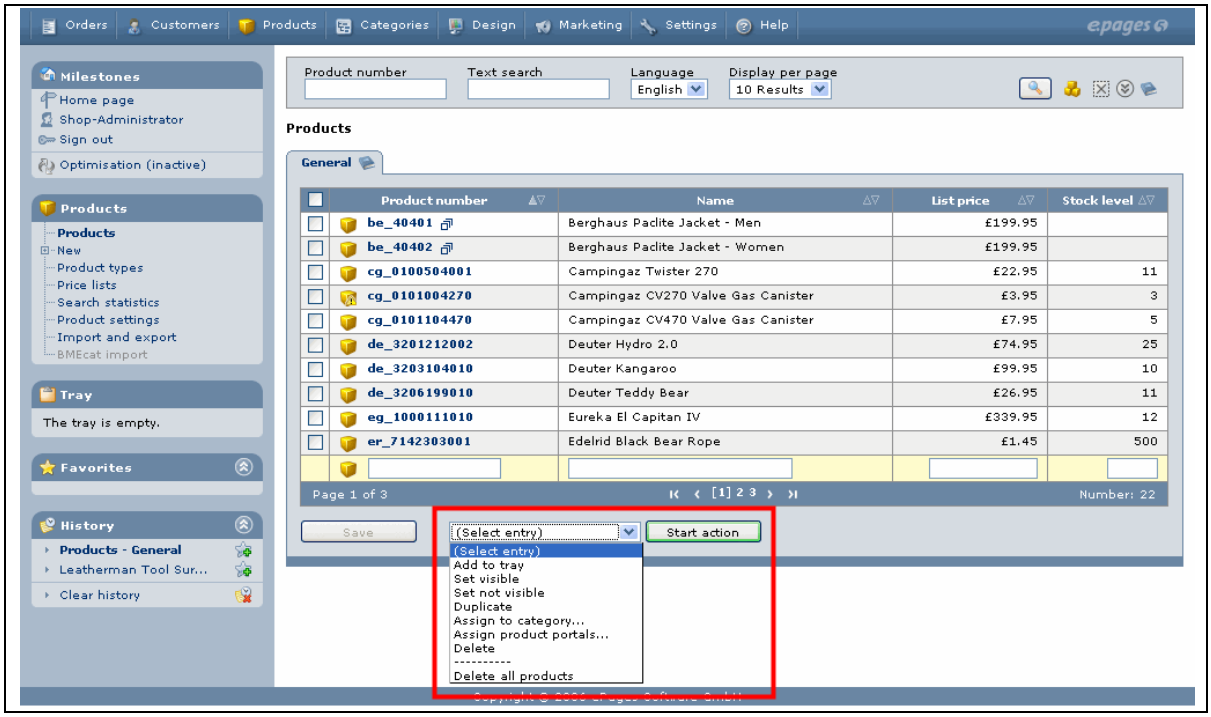

Abbildung 36: Stapelverarbeitungsaktionen für Produkte

In diesem Beispiel wollen wir die Stapelaktionen für Produkte um eine Aktion erweitern. Die Aktion soll die markierten Produkte in der Produkttabelle als *Neue Produkte* markieren. Diese werden im Shop speziell dargestellt. Die neue Aktion heißt *SetToNew*.

1. PageType identifizieren

   Stellen Sie fest, mit welchem Template die Stapelverarbeitungsaktionen angezeigt werden und in welchem PageType die Zuweisung erfolgt. Das ist der PageType *MBO-Products*. Diesem PageType müssen Sie neue Aktion hinzufügen.

2. Cartridge anlegen

   Legen Sie eine Cartridge mit dem Namen *AddBatchAction* an. Lesen Sie dazu *Anlegen einer Cartridge-Struktur*, *Seite 87*.

3. PageType erweitern

   Um die neuen Aktion einzuführen, müssen sie den PageType *MBO-Products* erweitern. Legen Sie in der Cartridge im Verzeichnis

   ```
   /Database/XML
   ```

   die Datei *PageTypesMBO.xml* mit folgendem Quelltext an:

```
<?xml version="1.0" encoding="UTF-8"?>
<epages>
  <!-- page types and templates -->
  <Cartridge reference="1" Package="Training::AddBatchAction">
    <Class reference="1" Path="/Classes/ProductFolder">
      <PageType Alias="MBO-Products" reference="1">
        <Menu Template="BatchActions" >
          <Menu Template="BatchAction-SetToNew" URLAction="SetToNew"
            Position="90" delete="1"/>
        </Menu>
        <Template Name="BatchAction-SetToNew"
          FileName="MBO/MBO-Products.BatchAction-SetToNew.html" delete="1"/>
      </PageType>
    </Class>
  </Cartridge>
</epages>
```

Codebeispiel 121: Erweitern eines bestehenden PageTypes

Über *reference="1"* im Tag *PageType* geben Sie an, dass Sie den PageType *MBO-Products* erweitern wollen. In diesem gibt es ein Element *Menu* mit dem Namen *BatchActions*, in dem bereits verschiedene Stapelaktionen definiert sind.

Sie legen nun ebenfalls ein Element *Menu* an mit dem gleichen Namen *BatchActions*. In diesem vereinbaren Sie die neue Stapelaktion *BatchAction-SetToNew* mit der URLAction *SetToNew*. Die Angabe *Position* bestimmt, an welcher Stelle in der Liste aller Stapelaktionen die neue Aktion stehen soll. Zur Laufzeit wird dann dieser Menü-Eintrag zusammen mit den Standardeinträgen verarbeitet und angezeigt, wie das Ergebnis in *Abbildung 37* zeigt.

4. Template anlegen

   In der PageType-Definition haben Sie der neuen Stapelaktion auch ein Template zugeordnet. Dieses Template heißt *MBO-Products.BatchAction-SetToNew.html* und Sie erstellen es im Verzeichnis */Templates/MBO* Ihrer Cartridge. Dieses Template beschreibt, wie die Stapelaktion auf der Seite dargestellt werden soll. In unserem Fall wird nur ein weiterer Eintrag für das Auswahlfeld für die Stapelaktionen angelegt. Der Quelltext für das Template sieht wie folgt aus:

```
<option value="#URLAction">{SetNew}</option>
```

Codebeispiel 122: Template für neue Stapelverarbeitungsaktion

5. Dictionary-Dateien anlegen

   Im Template-Quelltext haben Sie für den Namen der Stapelverarbeitungsaktion ein Language-Tag *{SetNew}* eingesetzt. Damit wollen Sie sprachflexibel bleiben, siehe *Mehrsprachigkeit – Language-Tags*, *Seite 51*.
   Nun müssen Sie für jede Sprache, die Sie anzeigen wollen, auch die entsprechende Sprachdatei anlegen, in welcher der Platzhalter durch eine konkrete Bezeichnung in der jeweiligen Landessprache ersetzt wird.
   Für unser Beispiel wollen wir zwei Sprachen, Deutsch und Englisch, anzeigen. Legen Sie dazu die Dateien *Dictionary.de.xml* und *Dictionary.en.xml* an. Grundlagen zu den Sprachdateien lesen Sie in *Mehrsprachigkeit – Language-Tags*, *Seite 51*.
   Die Dateien *Dictionary.de.xml* und *Dictionary.en.xml* erstellen Sie im Verzeichnis */Templates* Ihrer Cartridge und fügen folgenden Code ein:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<epages>
  <Language Language="de">
    <Translation Keyword="SetNew">Als neues Produkt anzeigen</Translation>
  </Language>
</epages>
```

Codebeispiel 123: Deutscher Inhalt für Language-Tag *SetNew*

bzw.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<epages>
  <Language Language="en">
    <Translation Keyword="SetNew">Display as new product</Translation>
  </Language>
</epages>
```

Codebeispiel 124: Englischer Inhalt für Language-Tag *SetNew*

Je nach gewählter Anzeigesprache wird die Stapelverarbeitungsaktion im Händler-Backoffice entsprechend angezeigt.

6. Funktionalität bereitstellen

   Um die Produkte als *Neues Produkt* zu kennzeichnen, muss ein Produktattributwert geändert werden. Um die Funktionalität zu implementieren, muss ein Perl-Modul geschrieben und in der Cartridge bereitgestellt werden.
   Die Datei, die den Perl-Code enthält, legen Sie im Verzeichnis */UI* an. In unserem Beispiel ist das die Datei *ProductFolder.pm* mit folgendem Quelltext:

```
package Training::AddBatchAction::UI::ProductFolder;
use base qw( DE_EPAGES::Presentation::UI::Object );

use strict;

use DE_EPAGES::Object::API::Factory qw ( LoadObject );

sub SetToNew {
    my $self = shift;
    my $Servlet = shift;

    my $MasterObject = $Servlet->object;
    my $Form = $Servlet->form;
    my $hValues = $Form->form($MasterObject, 'ListedObjects');
    my @ListObjects = map { LoadObject( $_->{'ListObjectID'} ) }
                      @{$hValues->{'ListObjectIDs'}};
    $_->set( { 'IsNew' => 1 } ) foreach @ListObjects;

    return;
}

1;
```

Codebeispiel 125: Code-Beispiel zum Setzen des Produktattributes *IsNew*

In der Funktion wird das Produktattribut *IsNew* für alle ausgewählten Produkte auf *1* gesetzt.

7. Abhängigkeit festlegen

   Sie müssen sich wieder überlegen, von welcher Cartridge Sie Funktionen übernehmen. Im vorigen Beispiel haben Sie Funktionen von *Design* geerbt. Jetzt sind speziellere Voraussetzungen notwendig. Sie wollen mit Produkten arbeiten, auf produkt-relevante Funktionen zugreifen bzw. diese erweitern. Also

leiten Sie Ihre Cartridge von *Product* ab. Diese Abhängigkeit legen Sie in der Datei *Dependencies.xml* im Verzeichnis */Database/XML* fest. Den erforderlichen Quelltext sehen Sie in *Codebeispiel 126*.

```
<?xml version="1.0" encoding="UTF-8"?>
<epages>
  <Dependency Package="DE_EPAGES::Product" />
</epages>
```

Codebeispiel 126: Festlegung der Abhängigkeit von einer anderen Cartridge

8. Aktion registrieren

   Weiterhin haben Sie über den PageType eine neue Aktion definiert. Diese Aktion muss vorher auch in der Datenbank angelegt und bekannt gemacht werden. Erstellen Sie dazu die Datei *ActionsProduct.xml* im Verzeichnis */Database/XML* mit folgendem Inhalt:

```
<?xml version="1.0" encoding="UTF-8"?>
<epages>
  <Class reference="1" Path="/Classes/ProductFolder">
    <Object Alias="Actions">
      <Action Alias="SetToNew"
        Package="Training::AddBatchAction::UI::ProductFolder"  delete="1" />
    </Object>
  </Class>
</epages>
```

Codebeispiel 127: Anmeldung der neuen Aktion in der Datenbank

9. Berechtigungen festelegen

   Für diese Aktion müssen Sie noch festlegen, wer sie ausführen darf. Da die Aktion nur im Händler-Backoffice zur Verfügung stehen soll, müssen Sie in der Datei *Permissions.xml* die Berechtigung wie folgt vergeben:

```
<?xml version="1.0" encoding="UTF-8"?>
<epages>
  <Role reference="1" Path="/Classes/Shop/Roles/Merchant">
    <RoleAction Class="ProductFolder"  Action="SetToNew"  delete="1" />
  </Role>
</epages>
```

Codebeispiel 128: Anmeldung der Berechtigungen in der Datenbank

10. Cartridge installieren

    Installieren Sie die Cartridge wie in *Installieren - nmake*, *Seite 89* beschrieben. Nach Beendigung des Prozesses rufen Sie das Händler-Backoffice und dort die Produktseite auf. Auf der Seite mit der Produktliste klappen Sie das Auswahlfeld für die Stapelverarbeitungsaktionen auf. Die neue Aktion ist dort aufgeführt, siehe *Abbildung 37*.

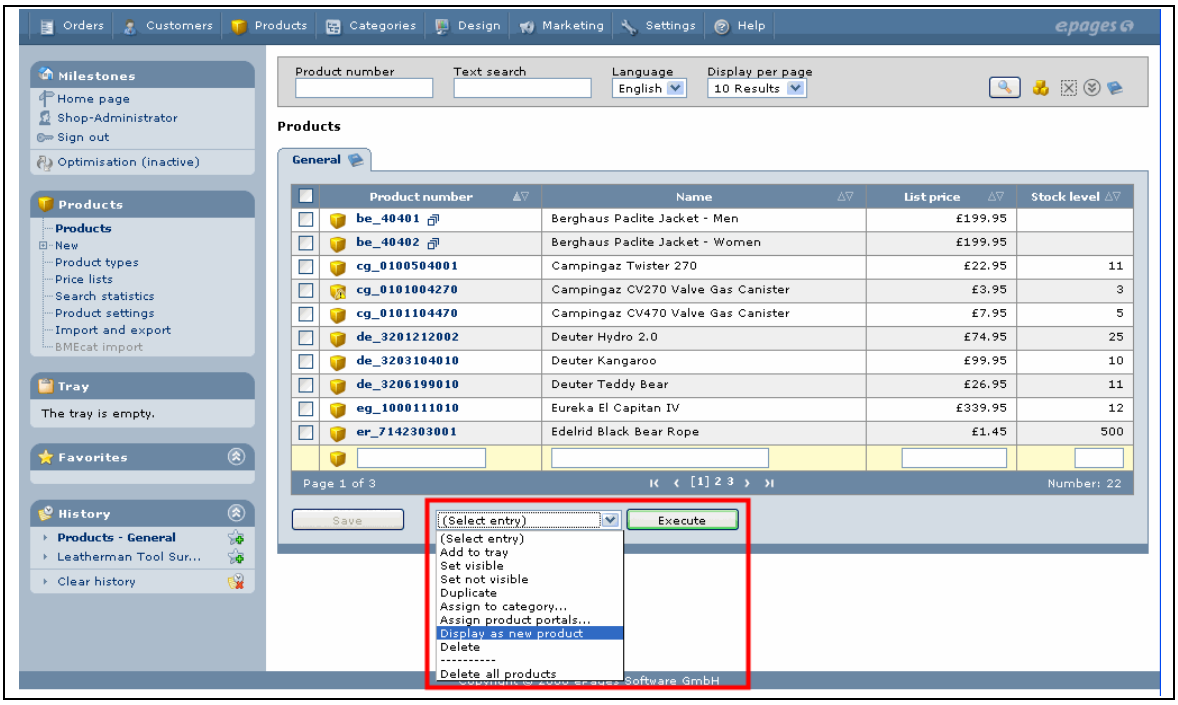

Abbildung 37: Neue Stapelverarbeitung hinzugefügt.

Da Sie auch für Deutsch eine Sprachdatei angelegt haben, sehen Sie auch in der deutschen Anzeige einen korrekten Eintrag für die neue Stapelverarbeitungsaktion:

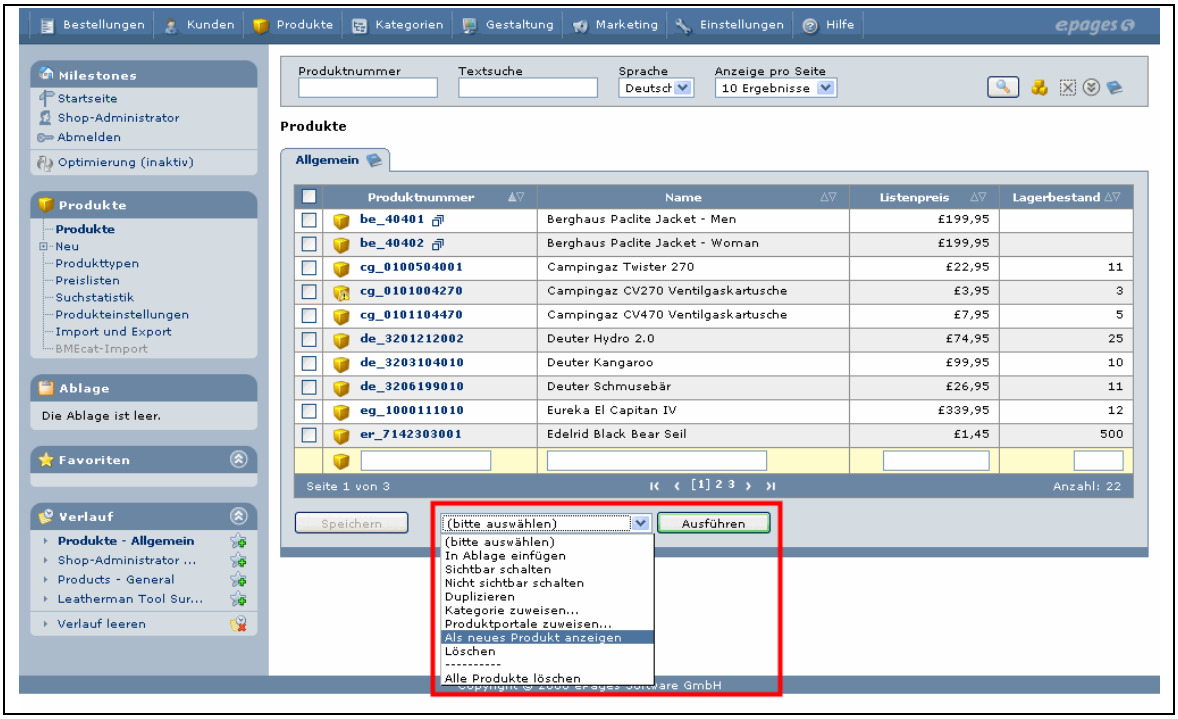

Abbildung 38: Neue Stapelverarbeitung in der deutschen Anzeige

Glossar

Applikationsserver
Hier ein Prozess des ePages-Programms. Auf einem Rechner können mehrere Applikationsserver gestartet werden. Dies geschieht durch den ePages-Service (Dienst).
Auch Server, der in einem Netzwerk eine Anzahl von Diensten bzw. Anwendungen (Applikationen) bereitstellt. In unserem Fall der Server, auf dem die ePages-Applikation läuft.

Backoffice
Gesamtheit aller Webseiten zur Administration, mit deren Hilfe Händler ihre Shops und die Administratoren die Anwendung verwalten.

Business-Unit
Einheit aus Datenbank und zugeordneten Cartridges. Die in den Cartridges mitgelieferten Funktionen bestimmen die Funktionalität der Business-Unit.

Cartridge
Softwarekomponenten auf der Basis von Perl. Jede Cartridge bringt definierte Funktionen mit, so dass durch deren Kombination Business-Units mit unterschiedlicher Funktionalität angelegt werden können.

Data-Cache
Zwischenspeicher im Applikationsserver. Hier werden Daten abgelegt, die oft nachgefragt werden. Dadurch wird nicht jedes Mal die Abfrage an die Datenbank weitergeleitet und die Antwortzeiten sind erheblich kürzer.

Fallback
Mechanismus zum Überladen von Originaldateien. In einem speziellen Verzeichnis werden Dateien angelegt, welche die Funktionalität der Originaldateien überschreiben und an deren Stelle verarbeitet werden sollen. Systemseitig ist eine Verarbeitungsreihenfolge der Dateien definiert, bei der zuerst in dem speziellen Verzeichnis nach den entsprechenden Dateien gesucht wird. Sind diese dort nicht vorhanden, werden die Originaldateien verwendet.

Folder
Ein Folder ist eine logische Ebene in der Objekt-Struktur. Jedes Objekt hat die Möglichkeit, Referenzen auf andere Objekte zu speichern und wird damit zum Folder für diese Objekte. Ein Beispiel sind Bestellungen, die einem Kunden zugeordnet sind. Dabei ist das Kunden-Objekt der Folder für die Objekte der einzelnen Bestellungen.
Es muss keine Vererbungsbeziehungen gemäß Klassenstruktur geben.

Language-Tag
HTML-Erweiterung, welche die Mehrsprachigkeit der Anwendung unterstützt. Mit Hilfe der Language-Tags werden im Template Platzhalter eingesetzt, die bei der Template-Verarbeitung nach Anforderung sprachsensitiv durch den entsprechenden Inhalt ersetzt werden. Die Sprachinhalte werden in XML-Dateien abgelegt und daraus ausgelesen.

Lokalisierung
Vorbereiten der Daten und Inhalte einer Webseite für die Anzeige in einer anderen Sprache. Dazu müssen systemseitig die entsprechenden Sprachen, Formate und Währungen bereitgestellt werden. Die Templates werden unter Verwendung der Language-Tag auf die Anzeige in mehreren Sprachen vorbereitet. Manche Attribute sind lokalisierbar, d. h. Sie können für diese Attribute für verschiedene Sprachen verschiedene Werte erfassen. Diese werden dann für die jeweilige Sprache angezeigt, z. B. Beschreibungen oder Namen.

Shoptypen Produkt des Business-Administrators, das er an Händler/Shopbetreiber verkauft oder vermietet. Jeder Shoptyp wird mit fest definiertem Funktionsumfang und normalerweise auch zu unterschiedlichen Preisen angeboten. Die Shoptypen sind die Basis für die Shops, die sich die Händler auswählen.

Site a) Objektklasse in jeder Datenbank, Basisklassen von *Shop*
b) Name der Administrationsdatenbank in der Standardinstallation, auf der TBO und BBO installiert sind.

Site-Datenbank Datenbank der ePages-Standard-Installation, auf der die Administrationsdaten der Anwendung gespeichert sind. Zugriff haben die Technischen und Business-Administratoren. Hier werden zum einen die zur Anwendung gehörenden Datenbanken und bestimmte Grundeinstellungen z. B. für Web Services und Webserver verwaltet. Zum anderen administrieren hier die Business-Administratoren Shops und Shoptypen.

Stapelverarbeitung Aktion, die sich auf mehrere ausgewählte Elemente auf einmal bezieht. Diese Möglichkeit wird in Tabellen angeboten, wo mehrere gleiche Aktionen zu einer Mehrfach- oder Stapelaktion zusammengefasst werden können, z. B. das Löschen von mehreren Tabellenzeilen auf einmal.

Store-Datenbank Datenbank der ePages-Standard-Installation auf der die Daten einzelner Shops gespeichert sind. Auf dieser Datenbank administrieren die Händler ihre Shops und bearbeiten die Produkt-, Kunden- und Bestelldaten. Von hier werden die dynamischen Inhalte der Storefront bereitgestellt.

Storefront Die "Kundenseite" eines Webshops. Die Gesamtheit aller Webseiten , die zu einem Shop gehören.

Style Alle Daten und Anweisungen zur Darstellung der Webseiten. Neben einer .css-Datei gehören Grafiken, Schaltflächenbilder und Farbzusammenstellungen dazu.

Template HTML-Datei zur Beschreibung einer Webseite oder bestimmter, definierter Bereiche. Neben "normalen" Anweisungen zur Gestaltung von Webseiten werden die ePages-spezifischen Erweiterungen – *TLE* und *Language-Tags* – verwendet, welche die Einbindung dynamischer Inhalte und die Anzeige in mehreren Sprachen erlauben.

TLE Template Language Extension. ePages-spezifische Spracherweiterungen, die zusätzlich in HTML-Quelltext eingebunden werden können. Sie ermöglichen die Einbindungen und Auswertung dynamischer Inhalte und so eine flexible Programmierung von Webseiten

Web Services Web Services dienen der Kommunikation zwischen verschiedenen Anwendungen. Sie bieten die Möglichkeit, Applikationen zu verknüpfen, die auf verschiedenen Plattformen und mit verschiedenen Programmiersprachen implementiert sein können, und zwischen ihnen Daten auszutauschen. Web Services verwenden zur Datenübertragung Standard-Internet-Protokolle, wie HTTP, SMTP und FTP, wobei HTTP am häufigsten eingesetzt wird, da hier eine direkte Reaktion auf eine Anfrage möglich ist, während SMTP und FTP nur asynchrone Datenübertragungen zulassen.

Webshop Internetapplikation, die alle Funktionen in sicht trägt, um Produkte oder Dienstleistungen zu verkaufen. In diesem Fall wird der Shop auf Basis eines Shoptyps generiert, den der Business-Administrator definiert hat. Der Händler generiert den Shop online, passt Struktur und Design an und erfasst seine Produkte und Dienstleistungen und eröffnet so seinen Internet-Vertriebskanal.

Index

A

Attribute 15
Auswahlstyles 135
Anlegen *135*

B

Basistemplate 44

C

Cartridges 85
Anlegen der Struktur *87*
Installation *89*
Installer *87*
Struktur *85*
Targets *89*
ChangeActions 31

D

Debugging 38
Diagnostics-Cartridge 131
Distribution 93
Dynamische TLE-Variable erstellen 79

E

Encryption 93
Export 119
Export.pl *122*

F

Fallback 38
Features 97
Fehlerbehandlung im Template 103
FormFields 100
Formhandling 99, 100
Forms im Perl-Code 101

H

Hooks 115

I

Import 119
Forms *122*
Hooks *121*
Import.pl *120*
Standards.pl *121*

K

Kompilat 34

L

Language-Tags 33, 35, 51
Syntax *51*
XML-Datei *52*

M

make 89
makefile 89
Mehrsprachigkeit 51

N

nmake 89

O

Objekt-API 12
Objektorientierung 11

P

PageTypes
Basistemplate *44*
Darstellungsebene *43*
Konzept *41*
Logische Struktur *41*
Vererbung *43*

R

Rechte 21
Rollen 21

S

Scheduler 125
Neue Perl-Script-Tasks *126*
Output *128*
Perl-Script-Tasks *125*
Starten *127, 128*
Stoppen *127*
UNIX-Shell-Script-Tasks *127*
Styles 135
Sub-Styles 139

T

Tasks 125
Template 33
Prozess *33*
Template-Prozess 33
Templates
Hierarchie *44*
TLE 63
Anweisungen *65*
Fehlerbehandlung *71*
Fehler-TLE *71*

Syntax *63*
Variable *63*
TLE-Anweisungen 65
Operatoren *76*
TLE-Formatter erstellen 81
TLE-Funktion erstellen 77
TLE-Variablen 63
Datenquellen *64*
Formatierung *73*

U

Überladung 38
URL-Aktionen 31
ChangeActions *31*
ViewActions *31*
User/Customer-Trennung 27

V

Vererbung 11
ViewAction 43
ViewActions 31

W

Web Services 107
Framework *107*
Webseitenstruktur 37

X

XML-Sprachdateien 52
Überladen *57*