ePages 5

Design and Cartridge Development Guide

- Version 5.04 -

Should you have questions or suggestions about our products, please contact us at the following address:

ePages Software GmbH
Leutragraben 1
07743 Jena
Germany

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, April 2007

Table of Contents

1. Introduction

ePages 5 is a standardized technology platform which offers a high level of flexibility and extensibility, thereby allowing you to quickly implement customer-specific customisations.

The many functions of the standard software provide the foundation for quick implementation of varied business models with low operating costs.

The separation of creative design and functionality allows you to more easily modify each area and also to reduce project length by working on each of these areas in parallel.

Cartridges are software components that are used to add new functions to the system and increase system flexibility. These encapsulated functions enable communication via programming interfaces and can be freely combined.

Developers should create functional extensions for the product as cartridges.

This guide describes the basics of customizing and extending ePages 5 in regards to design and cartridge functionality. Because of the complexity of the system, it is not meant to replace developer training, but to supplement it.

Visit our Web page at www.ePages.eu for further information about the seminars currently being offered.

1.1 Contents of this Guide

This guide describes the structures and concepts that are used in ePages 5. This guide uses examples and explanations to show designers and developers how to customize the default ePages installation to fit their needs.

Part I: *Basic Principles*, *on page 9* basic principles and concepts are explained that are important for developers and designers.

Part II: *Cartridge Development on page 79* describes the basic method used to create cartridges. Developers should be able with this information to develop functions to extend the system through cartridges.

In part III: *Additional Concepts* from *on page 91* describes further topics that are important to effectively use the system.

In part IV: *Design*, *on page 127* emphasis on customisation of design and layout. The method of designing Web pages with the system is described. The creative part of designing Web pages is not discussed here.

Appendix A: Performance Tuning on page 137 provides information about improving the performance of your installation.

Appendix B: Developer Notes on page 149 includes information about specific problems and solutions that occur fairly often.

In *Appendix C: Usage Examples (UE)*, *on page 177* you can find examples which support the explanations of the individual chapters. In the respective chapters, the example that fits is referenced. The source code for every usage example is included in this guide. Using the source code, you will be able to see how the examples work step-by-step. In order to guarantee the functionality of the cartridges, they must be installed in the *%EPAGES_CARTRIDGES%/Training* directory. All cartridge examples are prepared so that they can be copied into the correct location and installed in case one would like to test the functionality.

1.2 Requirements

This handbook is intended for developers and Web designers that would like to customize, extend, or optimise existing ePages 5 installations.

It is assumed that designers have knowledge about HTML/XHTML, JavaScript, CSS, and XML. For cartridge development, experience in PERL and SQL is also necessary.

A running ePages 5 default installation with the correct access rights on the storefront and back office, as well as in the databases and the file system is assumed.

The knowledge contained in the Merchant User, Technical Administration, and Business Administration guides is also valuable.

1.3 Typographic Conventions

The following fonts and formats are used to show special information:

`nmake install`	Program code and instructions Multiple lines of code are framed.
`perl import.pl [-help]`	Parameters in square brackets in program calls are optional.
‹text›	Text in angled brackets that begins with lowercase letters is a placeholder and will be substituted with real parameters.
All Hooks	References to hyperlinks in the images of the ePages application. Links that are shown like this are available in the application itself.
$hValues	This formatting refers to special names and IDs, such as file names, path names, field names, and entry fields.
#(context.)attributename	to be used as a general syntax for typed instructions

Note: Useful information that should be considered to work effectively are shown in a box like this.

Caution: Important information that must be followed are shown in a box like this.

Part I:

Basic Principles

2. Object Orientation

ePages 5 uses the advantages of object-oriented programming with PERL.

And for this, the following requirements and goals should be fulfilled:

- Consistent API in order to work with objects and attributes
- Use of language-dependent attributes
- Attributes can be saved in separate tables
- Inheritance of attributes and methods
- Cartridges can add new attributes to existing classes

A class structure with the necessary inheritance mechanisms was developed for this. For each class, attributes and methods are defined that can be used by the respective object.

The methods of a class are implemented in a PERL module, which is called a class package. The attributes are also implemented in a PERL module, which is independent of a class package. In this way, attributes for a class can be extended without changing the class package.

The assignment of attributes to classes is saved in the database. The database contents can be imported from XML files of each cartridge such as *Attributes*.xml*. For more on this, see *Import / Export of database contents, on page 113*.

The Diagnostics cartridge provides an overview of the class structure. See *Diagnostics Cartridge, on page 125*. The structure is dynamically extendable and can differ in various databases.

Each object is an instance of its respective class. All objects are organized in a tree structure. The basis of this structure is the *System* object. Each object can have an unlimited number of child objects.

Each object is described through three unique IDs:

Table 1: Object IDs

ID	Description
Objectpath+Alias	A unique alias related to the superior object. In this way, all objects are uniquely identified through the object path, starting with the "System" object. An example is *System/Shops/DemoShop/Users/admin*. When entering the object path, "System" can be left out. You could also write the example this way: */Shops/DemoShop/Users/admin*
Object ID	Unique number in the database.
GUID	Globally Unique IDentifier - an identifier that is unique through the application. It can be used to identify object references in external systems.

All three values can be read with the Diagnostics cartridge.

2.1 Inheritance

Subclasses can be derived from every class. These subclasses inherit all attributes and methods from the parent class and all classes from its parent classes as well.

Each class can only have one base class. Multiple inheritances are not possible. The base class of the complete hierarchy is the *BaseObject* class. All other classes are subclasses. They only have exactly one superior base class.

Attributes and methods of a base class can be overwritten in subclasses. If a class must reference a method of the base class, the following syntax must be used:

```
$self->SUPER::Save($Servlet);
```

2.2 Object API

The object API is the general interface for all objects. Use this API to create and delete objects and to set and read attribute values. The API provides the following methods (among others):

Table 2: Methods of the object API

Method	Meaning
insert	Creates a new object and triggers the hook *OBJ_Insert ‹ClassName›*. For most objects, the parameters parent and alias in the hash *$hValues* must be passed.
delete	Deletes an object and triggers the hook *OBJ-Delete‹ClassName›*
load	Initializes an existing object
id	Returns the Object ID.
get	Returns one or multiple attribute values
set	Sets multiple attribute values and triggers the hooks *OBJ_BeforeUpdate›ClassName›* and *OBJ_AfterUpdate‹ClassName›*.

The package *DE_EPAGES::Object::API::Object::Object* provides a basic implementation of objects that are stored in a relational database.

The package *DE_EPAGES::Object::API::Object::Factory* provides methods to access existing objects and classes as well as for inserting and deleting objects. The most common functions are:

Table 3: Functions of the *Factory* module

Function	Description
InsertObject	Creates a new instance of an object
DeleteObject	Deletes an object by reference of its Object ID
LoadObject	Returns an object referenced by its ObjectID
LoadObjectByPath	Returns an object referenced by the object path
LoadRootObject	Returns the System object
LoadClassByAlias	Returns the corresponding class based upon the name

Code example 1 Shows the usage of the 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( { Price => 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;
```

Code example 1: Functions of the object API

3. Attributes

Attributes contain primarily the object types and properties. They are defined as objects. The base class is *Object*. Because of this, they inherit all attributes of the base class. This means that attributes also have attributes. These additional attributes are used to describe the properties of an attribute. The following are used by default:

Table 4: Attributes to describe attribute properties

Function	Description
Name	Language-dependent name that is used for documentation purposes, for example
Description	Language-dependent description that is used for documentation purposes, for example
Type	ID for the data type of the attribute; Specifies the attribute of an object reference or a list of references. If a list of references, the attribute name is the name of the object class of the values. References the attribute of the type *Object*. They can contain references to objects of any class.
Length	Maximum length; Only used for strings and language-dependent strings
Package	Name of the PERL package that is used to access attribute values
IsArray	Defines that attributes returns multiple values as an array Default setting: *0*
IsCachable	The object can cache the attribute. Default setting: *0*, language-dependent attributes are not generally "cacheable"
IsExportable	Should be exported with the object. Default setting: *0*
IsMandantory	Definition as a mandatory field. This means that a value must exist; Default setting: *0*
IsObject	*1* - The attribute value is an object reference or a list of objects. The mapping for the object occurs via the object ID. *0* – Attribute is a value Default setting: *0*
IsReadOnly	Can only be read.

In ePages 5, these attribute types are used:

- Default attributes,
- Language-dependent attributes,
- Attributes with pre-defined values

The following default attributes are supported:

Table 5: default attributes

Type	Comment
Integer	Integers with prefixes (32 Bit)
Float	
Boolean	1=true, 0=false
Money	Fixed-length floating-point numbers The Sybase data type *money* is defined as numeric(19,4). That means that numbers up to 15 characters before the decimal point and 4 after the decimal separator are saved.

Type	Comment
DateTime	Date and time, based upon the PERL class DateTime The Sybase data type datetime allows values from Jan 1, 1753 to December 31, 9999.
Date	Based upon the PERL class DateTime. Only the date portion is used. The time portion is set to 0:00:00.
Time	Based upon the PERL class DateTime. Only the time portion is used. The date portion is undefined.
String	Unicode text of unlimited length Short texts are saved in the database in fields of type NVARCHAR. If the texts are longer than VARCHAR allows, the text is saved in fields of type TEXT in another table.
File	File name or URL

3.1 Language-dependent Attributes

Language-dependent attributes are attributes of the same name that can have various values in different languages. Typical examples are product names or descriptions.

The basic implementation of language-dependent attributes can be found in the package *DE_EPAGES::Object::API::Attributes::LocalizedAttribute*.

The following types are available:

Table 6: Types of language-dependent attributes

Type	Comment
LocalizedString	Analog type *String*
LocalizedFile	Analog type *File*

During transfer of language-dependent content, you must always enter for which language the values are transferred. There are two ways to do this:

1. By setting a global variable which determines the default language for further attribute functions, see *Code example 2*,
2. Entering the language directly for each set()- or get() method for attributes. See *Code example 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' } );
}
…
```

Code example 2: Setting the language globally

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

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

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

Code example 3: Defining the language for each function

3.2 Attributes with Pre-defined Values

For these attributes, an unlimited quantity of values can be defined. They are used to give the user the possibility to select from a predefined list of values. Examples of this are selection boxes for product type attributes or customer attributes.

These lists of values can be language-dependent or language-independent. There are two attribute types that are based upon *String*:

- PreDefString
- PreDefLocalizedString
- PreDefMultistring
- PreDefMultiLocalizedString

3.3 Reference Attributes

In addition to simple values, attributes can also contain references to an object or a list of objects.

You can see how to pass these references to attributes in *Code example 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 ] });
```

Code example 4: Passing references to attributes

3.4 Adding Attributes

If necessary, you can add new attributes to classes. There are two main ways to do this:

1. Assign new attributes to an existing class. In this case, all of these attributes are available to all instances of this class.
2. They create a new subclass and assign new attributes to this class. This method is useful if new attributes are not applicable to instances of an existing class.

Code example 5 shows how new attributes are created using the object API:

```
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'
                              }
                            );
…
```

Code example 5: Creating a new attribute

In the example, the new class *LineItemUPS* is derived from the class *LineItemShipping*. A new attribute *Distance* of type *Float* is created for the new class. All functions related to this attribute are implemented in the package indicated.

Another way to create attributes is to define and then import the attributes in the *Attributes*.xml* file. For more on this, see *Import / Export of database contents*, *on page 113*.

3.5 Attribute API

The default implementation of attributes based upon simple data types, language-dependent strings or object references are sufficient for many applications.

In some cases, it can be necessary to create your own attribute access functions. Such cases are, for example

- if attributes must be searched. The implementation of default attributes does not allow any indexing or attribute values for search. In order to be able to search by attribute values, these should be saved in a separate table, which is able to have an appropriate index.
- if attribute values are going to be calculated based upon other attributes
- if the attribute values are stored in an existing table or in an external database

For situations like this, you must implement your own attribute API in a Perl module. The following functions must be contained:

Table 7: Functions of your own attribute API

Function	Comment
getAttribute	Returns an individual attribute value
getAttributes	Returns multiple attribute values in the form of a hash. Depending upon the database, this function can be faster than getAttribute. The field $aNames only contains the ID (alias) of the attributes that are implemented in this package.
setAttribute	Sets a single attribute
setAttributes	Sets multiple attributes at once. Depending upon the database, this function can be faster than setAttribute. The hash $hValues only contains the attributes that are implemented in this package.
DeleteObject	This method is called to delete an object. The field $aNames contains the names of all attributes that can be used with the object that is to be deleted and that are implemented in this package.
deleteAttribute	This method is called to delete one of the attributes, for example if a cartridge is being removed.

Function	Comment
getAllAttributes	Returns a hash of all attribute values. Language-dependent attributes are returned as a hash.
defaultAttributes	Returns a hash with default attributes. These are used when creating a new object if the values are not passed through InsertObject().

Setting an attribute value to *undef* leads to deleting the respective value from the database.

Attributes that return object references only function with the object ID.

The module *DE_EPAGES::Object::API::BaseAttribute* provides a basic implementation so that you do not have to implement all the functions yourself.

4. Rights and Roles

In order for a user to perform actions within the system, he must first be assigned permissions. These permissions are assigned for specific objects and apply to all child objects.

A relation must therefore be created between rights, users, actions, and objects. Users are can be assigned permissions to perform specific actions to specific objects.

Theoretically, a permission for every action could be assigned to every user for every object. However, because of the large quantity of objects, actions, and users, this would be too time-consuming. Therefore, to simplify things, roles and groups were created and inheritance is used.

A *Role* is a collection of many related actions. These actions are often performed together or somehow related. During assignment of rights, roles should be used instead of individual actions.

Multiple users can be placed into a *Group*. Use this method to grant various users the same rights. Assign rights to the group and then assign the users to the group.

To avoid having to assign rights to many individual objects, use *inheritance*. Rights can be inherited by a parent object. Rights are not physically copied. Instead, an inheritance relationship is created. Because of this, changes to rights of the parent object are applied to all child objects as well. The inheritance relationship is created immediately during creation of an object.

Through various possibilities to group users, actions, and objects, it is possible that user can have multiple rights for the same action on a single object. It is even possible that an action can be simultaneously allowed and forbidden. To resolve conflicts like this, the following rules apply to rights management:

- anything that is not explicitly allowed is forbidden
- any action which is both allowed and forbidden for a user will be forbidden

To provide actions for users, you should perform the following steps:

1. Implement and register an action
2. Assign the action to a role
3. Assign the role to a user or a group

4.1 Registering Actions for Objects

Actions are defined in the *Actions*.xml* file and registered in the database during import. For each cartridge, multiple files of this type can exist. They must be in the cartridge directory at */Database/XML*.

How an action is defined is shown in *Code example 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>
```

Code example 6: Defining an action

First select to which class the action will apply. The *object* tag with the alias *Actions* represents a folder where actions are entered.

Each action is individually defined. *Alias* is used to assign the ID. This name is also used in the template. In the package you can enter where the PERL function for the action is implemented. The parameter *FunctionName* contains the name of the function in the PERL module that is called if *Alias* and PERL function names are different. If both names are the same, the parameter *FunctionName* can be omitted. See the second action defined in the example.

Attributes can be assigned for display actions (ViewActions). *HelpFileTopic* refers to the corresponding online Help for this action. Using *Name* you can assign language-dependent names to actions. This name is displayed in the MBO in the *history*.

4.2 Assigning Actions to Roles

Roles and permissions are defined in the file *Permissions*.xml* and are registered in the database during import. For each cartridge, multiple files of this type can exist. They must be in the cartridge directory at */Database/XML*.

By default, four roles are defined: *Merchant*, *Customer*, *WebService*, *User*. These roles are defined for various objects. Because these roles are sufficient for the application, you will usually assign permissions to the existing roles, as shown in *Code example 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>
```

Code example 7: Assigning an Action to a Role

You reference the role that you would like to assign to the action. To do so, enter the object path for the role.

For the action itself, enter the name of the action and the classes that are assigned to the action. Compare *Code example 6*.

In case you want to create a new role, use *Code example 8* as an example.

```
<?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>
```

Code example 8: Creating roles

Refer to the class that you would like to create for the role. You can define the role with an appropriate alias in the *Roles* object folder. In the example above, the role *System/Classes/Shop/Roles/Merchant* is defined.

Within a role, all actions are listed individually with the class that is registered for this action.

To create roles, also use the file *Permissions*.xml*.

4.3 Assigning Rights

Groups and users must have the permissions required in order to perform the defined actions. This means that they must be assigned the correct rights.

Assignment of rights to a group is shown in *Code example 9*. They use the file *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>
```

Code example 9: Assigning rights to a group

If a new group is created, assign an alias. Otherwise, refer to an existing group.

In *Group* you can define which users receive which rights. In *Object*, you can determine to which object the permissions apply.

In *Permission* you define which roles the members of the group are assigned to. The exact role name consist of the entries in *Class* and the role-alias.

In *Code example 9* this means: Members of the group *Everyone* are allowed to perform all actions of the role *Customer* of the class *Shop* on the *System* object. The object *System* is represented by */* in *Path*.

The right is explicitly allowed via the attribute *Allow=1*. As long as *Allow* is not set, the action is implicitly forbidden.

If *Allow=0*, you can forbid an action for a specific user in case this action was allowed in another role.

Rights can also be assigned for individual users. This generally happens when creating a new user. This means that if a customer registers in the shop, a user with the *User* role is created for him. If a user is created in the user management the back office management, this user receives the role *Merchant*.

Instructions for creating a user using an XML file and assigning rights is shown in *Code example 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>
```

Code example 10: Creating a user with rights assignment

A user is created with an alias, a password and a user name in the *Users* folder. He receives the rights for the *User* role that are defined for the *User* class. This role allows him, for example, to change his password and other personal data.

In addition, you can assign additional permissions for roles to other objects. In our example, the user *Admin* receives the rights for all actions in the roles *Merchant*, *Customer* and *WebService* that are defined for the shop object.

5. Difference Between *User* and *Customer*

ePages 5 contains a user-customer concept that allows various business modules to be created with a single solution. Three primary scenarios were created: The shop-scenario, the marketplace scenario, and the procurement scenario.

The foundation of the concept is the differentiation between users and customers that both represent a different view.

A "user" is a person who performs real actions within the application, for example registering or creating orders.

A customer is a business partner for whom the actions are performed, comparable to a company for which the orders are made. He "carries" the business relationship, so to speak.

The necessity of separating these both is easy to see when one views three scenarios:

Shop scenario

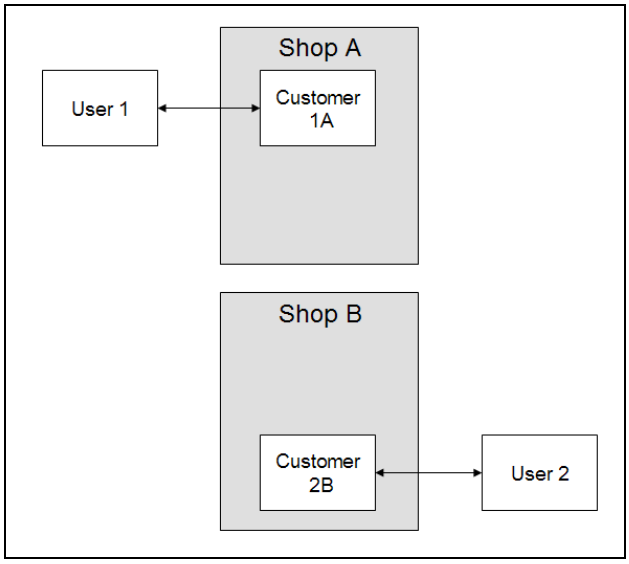

Figure 1: Shop scenario

It is characteristic for this scenario that a 1:1 relationship exists between the user and the customer. A customer is created by the user who is determined via a sign-in. The user creates orders for this single customer, edits account and shipping information, and so on.

Because of the 1:1 relationship, no division between the user and the customer exists in this scenario. This means that only one role can be assigned all information.

Marketplace scenario

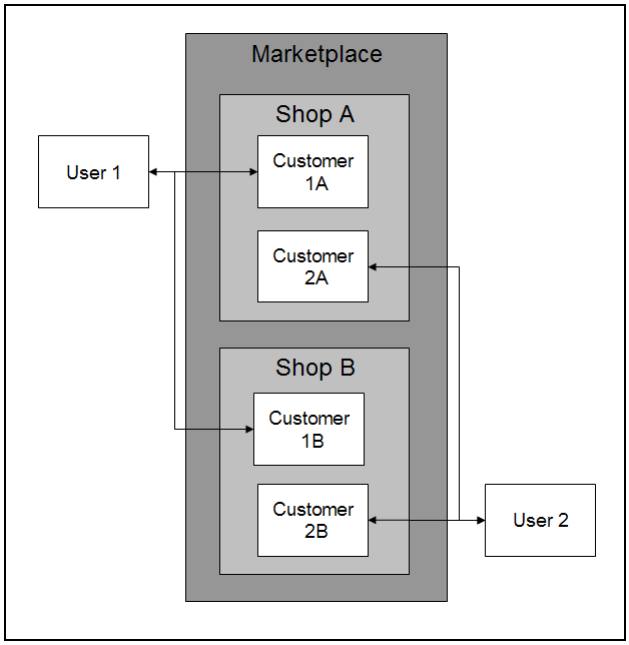

Figure 2: Marketplace scenario

The marketplace scenario is notable for the fact that a user with one set of sign-in data has access to multiple shops. This means that multiple customers are assigned to this user. Therefore, the relationship is 1:n. With this scenario, a requirement is that the data are divided into user data and customer data. The specific data for "his" shop are saved for each customer, such as customer group or bulk discounts. The data are created for the user that are necessary to be able to access all shops in the marketplace with his sign-in data.

Procurement Scenario

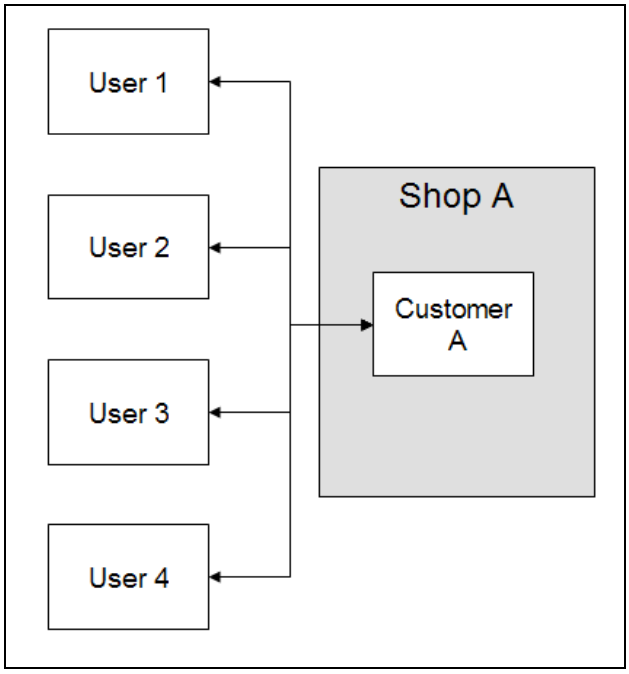

Figure 3: procurement scenario

In the procurement scenario, the customer (company) is represented by multiple users (employees). These users all have their own user names. In this case you can, according to your permissions, perform various

actions and edit various data. In this case, we have an n:1 relationship. Here it is also necessary to differentiate between user-specific data that are saved for every user and shop-specific data that are saved by customer.

For this reason it is clear that users and customers each have their own specific data pool that must be created and edited. The structure of the data can be seen using the Diagnostics Cartridge. Knowledge of these relations is important if you, for example, want to extend the data model and must decide whether data are assigned to the user or the customer.

Another example is for the connection of external systems that create and edit customer data in the system. It is also important for the developer to know for this which data belong to the user and which belong the customer.

6. URL Actions

In ePages 5, two types of actions are defined: *ViewAction* and *ChangeAction*. ViewActions are used to create various views of objects. Through ChangeActions, object data are changed.

A request to the application can activate one ViewAction and start multiple ChangeActions. The ChangeActions are always performed before the ViewActions.

Modification and display of an object is defined by query parameters in the URL or through form parameters of a POST request.

A typical request could be

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

The actions are assigned to specific permissions. The user must have the corresponding permissions in order to perform these actions. The permissions must exist for all actions that are performed by the request. Otherwise, the request cannot be completely answered. For more about “Permissions”, see also *Rights and Roles*, *on page 21*.

Actions are saved in XML files in the */Database/XML* subdirectory of a cartridge. Actions are registered in the database via XML import.

The file names must begin with *Actions* or *PageTypes*, for example *ActionsShop.xml* or *PageTypesMBO.xml*. The Perl modules that belong to the action are implemented in the */UI* directory of a cartridge. Examples for this can be found in the corresponding directories of a default installation.

For more information, see *Rights and Roles*, *on page 21* and *PageType Concept*, *on page 39*.

6.1 ViewActions

The ViewAction determines which view of an object is displayed. The *ViewAction* parameter of the request, specifies which specific view should be used. The corresponding object is specified through entering the ObjectID or the object path.

The ViewAction is linked to a specific PageType that is necessary for the requested view.

A ViewAction can optionally perform a number of PERL functions that prepare additional TLE variables for display.

The default ViewAction *View* is used if no parameters are specified for the ViewAction. If no object is specified, the System object is shown

6.2 ChangeActions

Use ChangeActions to read or edit object data, perform calculations, and so on.

The ChangeActions that will be performed are either saved as form parameters or defined as parameters in the URL.

Each ChangeAction starts various PERL functions that modify the specified object or its data. The object must be specified by the ObjectID or the object path.

If you would like to edit another object than the one shown, this must be addressed using the parameter *ChangeObjectID*.

7. Templates

Displaying an object in the browser is performed using a ViewAction. How this object is displayed is defined in a template. This template is connected to a ViewAction by a PageType. For more on this, see *PageType Concept*, *on page 39*.

Templates are the foundation for rendering in a browser. They are documents that contain information about the format and structure of the contents and data. XHTML, JavaScript, and ePages-specific language extensions are used in the templates.

These language extensions are *TLEs* (Template Language Extensions) and the *Language Tags*.

TLE variables are placeholders for dynamic information from the database. When a page is requested, these placeholders are replaced by current database content.

TLE statements are ePages-specific commands used to display Web pages depending on the content. These statements are used to query and evaluate dynamic data and to generate a Web page depending on the results of these statements.

When a template is being created, TLE statements serve as placeholders for language-specific expressions. These statements are dynamically replaced with the correct content at runtime depending on the current language.

7.1 Technology

Templates are built modularly. The template for an HTML page consists of multiple templates that each describe a specific portion of the page. These templates can then be used for various pages. This reusability minimises development and maintenance. For more on this, see *Display Levels*, *on page 41*.

Using TLEs and language tags gives you the option of requesting any piece of data in the database and even displaying language-dependent information in real time.

This means that you can use a template to generate and display different HTML pages with variable content. This results in a significant reduction in work for the Web designer.

In order to create Web pages, you must know how to work with HTML/XHTML and XML. Experience with JavaScript is not necessary, but is helpful. Working effectively with TLE language elements and multilingual pages is described in chapters *TLE*, *on page 61* , and *Multiple Languages–Language Tags* , *on page 49*

7.2 Template Process

When a request is made, different Web pages are generated depending on the contents of the database and the language requested. Due to the complexity of this process, it is impossible to create all the Web pages that could be required beforehand. Pages must be dynamically generated with current data at runtime.

One disadvantage of generating pages dynamically is the long response time in comparison to the time necessary to display static pages. A balance must be found between performance and the accuracy of current information.

For this reason, the template process includes several areas where the developer or user can decide whether brand new pages should be generated with current information or whether new pages can be put together in whole or in part from pre-existing HTML files.

A request triggers the following process. See *Figure 4*:

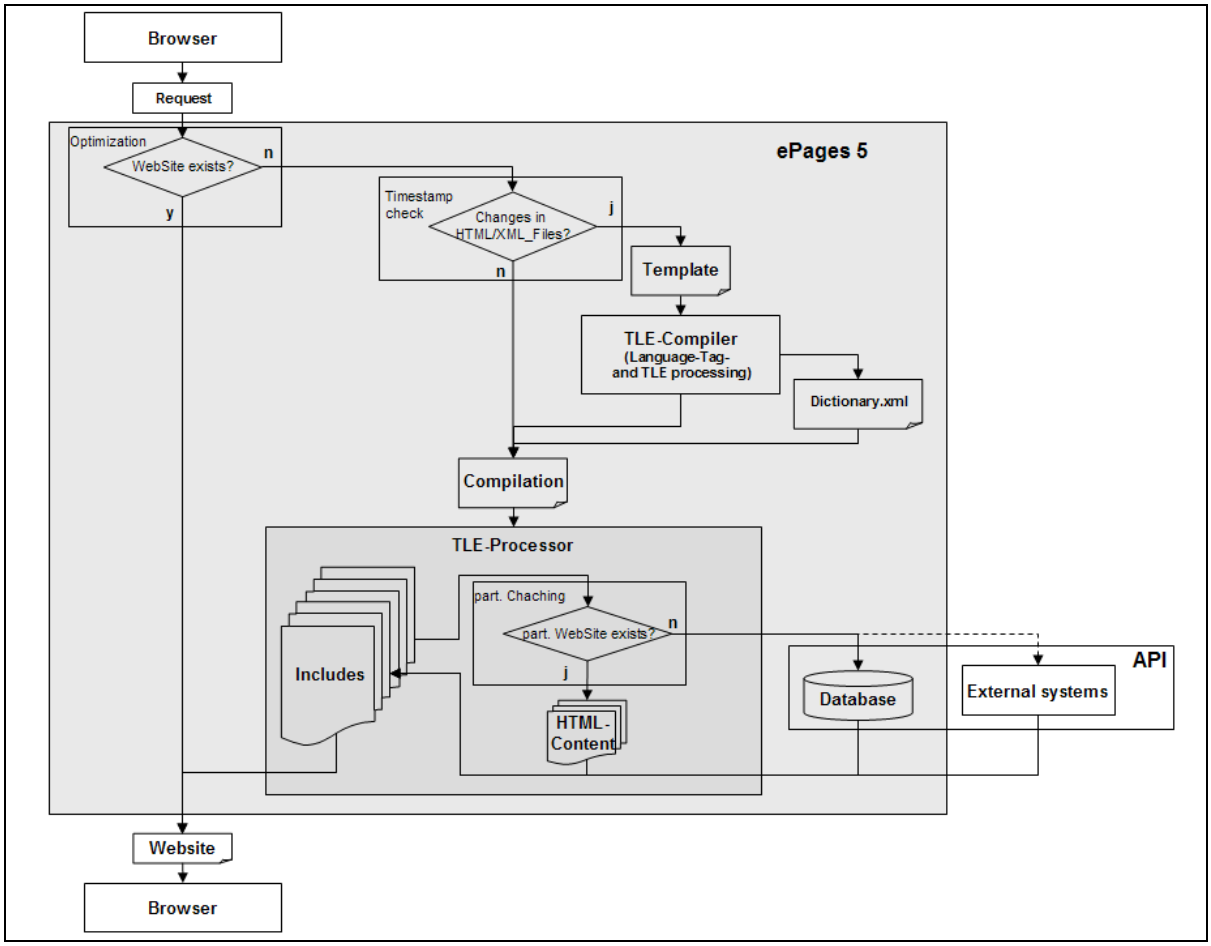

Figure 4: the template process, simplified representation

After a request has been made, it is possible to check whether the Web page requested is already available as a static HTML page. This check can be activated on the merchant's administration page, see *Optimisation* in the *Merchant User Guide*.

This is the fastest option for replying to the request. However, none of the page information will have been updated.

Otherwise, the actual template process begins.

All the templates that have already been requested once are stored on the hard drive along with a time stamp as precompiled versions (compiled files). These compiled files can be considered a type of cache that reduces access times significantly.

When a request is received, the system first checks whether the template requested already exists in a precompiled state. If it is present, the time stamp of the cached version is compared with the original HTML template and the XML language files. If the time stamp on the HTML file and on the XML files are not newer than the one on the compiled file, the system uses the precompiled version and sends it to the TLE processor.

If the time stamp on the precompiled page is older than the one on the template or on the language files, the compiled filed is therefore outdated and is deleted. After this, the template translation process is restarted. The same thing happens when no compiled file for this template exists, that is, if the template has never been translated.

The function for checking the time stamp can be deactivated in order to increase performance, see *Page Caching, on page 137*.

When a compiled file is generated, the first step is to check the template for language tags and to insert language-specific contents, see *Multiple Languages–Language Tags , on page 49*. After this, the template is processed and saved as a compiled file to be further processed by the TLE compiler. This file is saved in the file system and overwrites any existing "outdated" version.

The following figure shows an example of the individual parts of this process:

In *Figure 5*, you see the code section of a template. In addition to the HTML formatting, the TLEs (beginning with a #) and language tags (enclosed in {}) stand out.

```
#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
```

Figure 5: code example for the template

This is how a compiled file is structured. You can view a section of it in *Figure 6*. The language-specific data are inserted and the TLE variables are converted to PERL function calls.

```
#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"
```

Figure 6: compiled file for the template in Figure 5

Later the compiled file is processed by the TLE processor. The TLE processor basically serves as the switch board between the compiled file and the application server.

At this point, there is another opportunity for reducing the response times for the request. Partial caching can be activated for defined template sections, see *Partial Caching*, *on page 144*. This lets you decide whether the HTML code generated in previous operations will be reused for individual sections or whether to generate an entirely new version with current information.

Without partial caching, the compiled file is processed by the TLE processor, the database queries and actions are executed, all the necessary information is replaced, and the page is coded in HTML.

After all these parts, which are known as includes, have been processed, the resulting HTML page is sent to the browser.

The compiled files are text files with a *.ctmpl* extension and contains PERL code. They are stored in the subdirectory under

```
%EPAGES_SHARED%/Static/Store/Templates/DE_EPAGES
```

with the name of the original cartridge used. In this example, *Store* is the current database being used.

> ***Caution:*** Do **not** edit the compiled text files (ctmpl files). This will destroy the integrity between the template and the compiled file. Make your changes in the template and automatically regenerate the compiled file!

7.3 Basic Web Page Structure

ePages provides a large number of predefined, fully functional templates for your shop. You can use these templates as the basis of new templates or customise the templates yourself.

All HTML pages generated from these templates share a similar basic structure. The *Body* section for storefront page layout contains the sections seen in *Figure 7*.

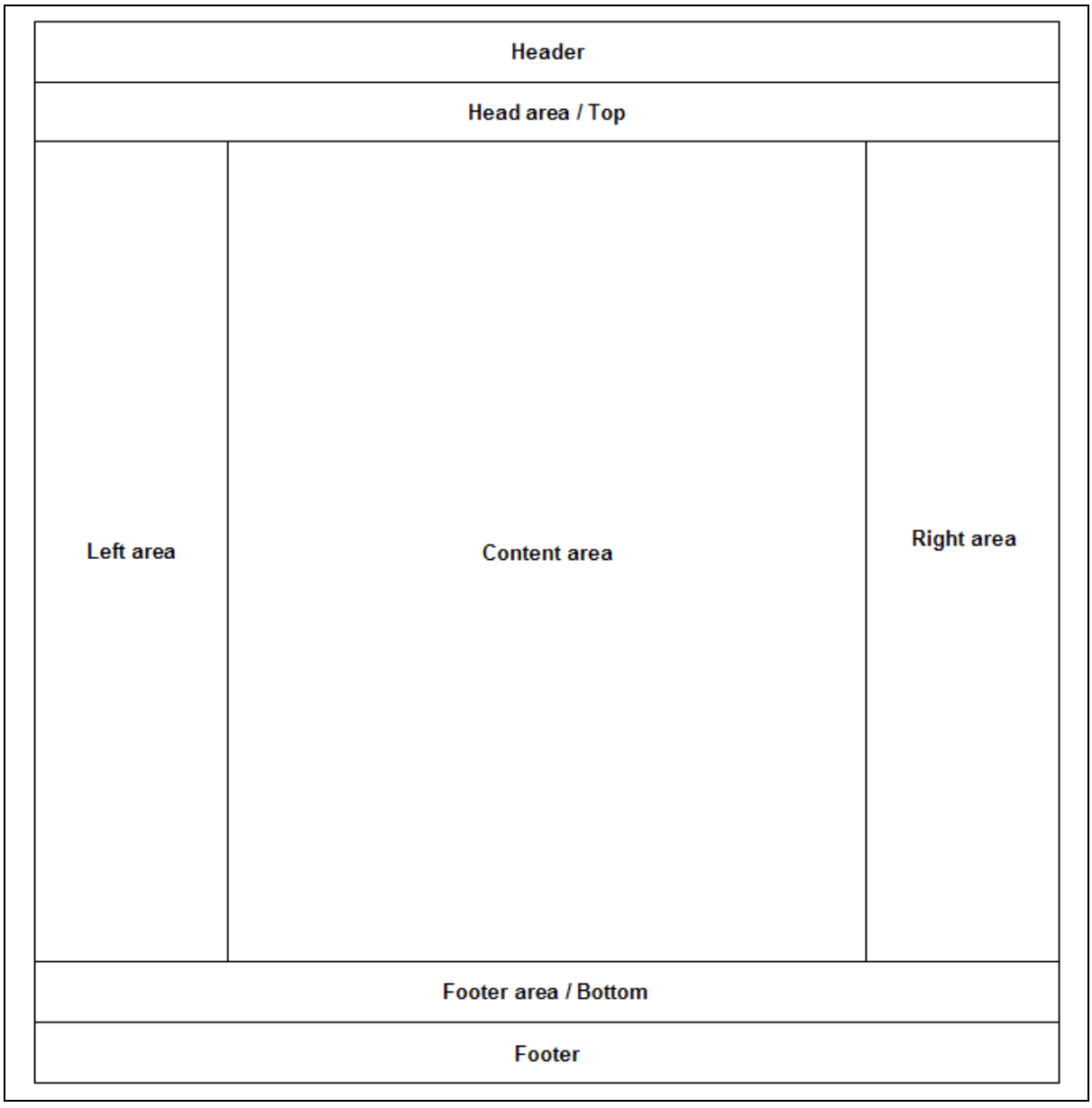

Figure 7: page structure in the body section of HTML pages

While the "edge areas" contain mainly navigation and function elements, the working area displays the results of the various functions.

The structure is defined by PageTypes that control how the HTML pages are put together using the individual include templates at runtime.

7.4 Overlaying Templates

Often there is the need to customise templates to fit your personal requirements. However, errors that result from this in the original templates can cause problems for the whole system. If you change the original files, you will also not be able to upgrade your ePages application properly. If you have made customized changes in any of the original templates in the default cartridges, an upgrade will overwrite these changes and they will be lost.

> ***Caution***: If you want your system to remain updateable, do not make any changes to the original files in the original directories!

There is a mechanism in ePages for this that helps you to make changes without editing the original templates. That is template overlaying. The basic proposition is to provide various templates of the same name and define their usage order.

In the ePages system, there are specialised overlaying directories. You can copy the original files to these and edit them. The original files remain unchanged in their original directories. The editing sequence is defined in the system in such a way that first the system makes an attempt to find the files needed in the "overlay directory". If the files are there, they are used. If they are not there, the system accesses the original directories.

The directory in which you can save your modified templates is located at

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

Here, you will find, as with the installation directory, a directory for every default cartridge and in them the corresponding template directories for the store front and the back office.

If you want to change a default template, copy it from the original directory into the directory with the same name at the location mentioned previously. Then you can make all your changes without having to worry that the system will be influenced. If your modified template still has errors or is not yet finished, simply rename it or delete it from the directory to restore the original conditions.

This procedure is convenient if you are for making changes to your local installation that you do not want to pass on.

If these changes are part of a packet to be delivered, you should collect these changes in their own cartridge. You can also use the overlaying technique when you put your modified files in the correct directories in your cartridge. Read about this in chapter *Cartridges*, *on page 81* in the explanations about the subdirectory *DATA/Private*.

An example for using template overlaying can be found in *Appendix C: Usage Examples (UE)*, *on page 177*.

Storefront templates and back office templates can both be overwritten.

7.5 Template Debugging

Before you can change the templates, you need to know which template displays which part of the Web page. In other words, which template you must change. It is very helpful to activate the debugging function. This displays the names of the templates used in the HTML source code of the page displayed. Compare *Figure 8* with *Figure 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>
```

Figure 8: the HTML source code of a displayed page without debugging information

If debugging information is activated, the comment lines with template information are very easy to see:

```
<!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">
```

Figure 9 HTML source code of a displayed page with debugging information

Using the debugging information, you can determine exactly where which template is inserted and where you can find it. This makes it simpler for you, coming from the area of the Web page that you would like to change, to find to the template you must edit.

As an additional important parameter, you can see the processing type of each INCLUDE. Use this data to optimise the performance of your application.

In order to activate debugging, open the following file:

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

and search in the section *tle* for the following entry:

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

see *Figure 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
```

Figure 10: entry for activating debugging

After installation, these lines are commented out which deactivates debugging. Remove the semicolon at the beginning of the line and save the file. After this, you can see the debugging information in the HTML source code.

You can find more about the *log4perl.conf* file in the *Installation Guide for Windows*.

8. PageType Concept

A PageType is an XML file in which the display of an object is defined structurally. This makes a PageType a connector between the ViewAction for an object and the display in the browser through templates. They also determine which template is to be processed depending on the object and the display action processed. This reference is made via the PageTypes and is saved in the database.

In addition to the area defined, the corresponding ViewAction is determined. A special ViewAction can be named. If no ViewAction is explicitly named in the XML file, the *View* ViewAction is used by default.

PageTypes are the basis for structuring templates as modules. Using PageTypes, it is possible to develop the design and function to such a detail that the different page functions can be composed from the individual modules with a great level of flexibility and re-usability.

Due to these complex interrelations, you should work through the practical examples using the basic principles. This will help you better understand how to work with PageTypes.

8.1 Logical Structure

A PageType defines the logical structure of a Web page. Individual areas are determined from which the page is combined. These logical sections are assigned to HTML files. They contain the source code for describing the individual sections, that is, an actual HTML file is indicated for every section. For more details, see *Templates*, *on page 31* and *Display Levels*, *on page 41*.

You can view a simple example for the XML definition of a PageType in *Code example 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>
```

Code example 11: XML definition of a PageType

Cartridge reference... indicates the cartridge in which the template files are located.

Class reference ... indicates the PageType–object class assignment. The PageType in the example is assigned to the object class called *Object*. This means that any object can use this PageType.

The name of the PageType is assigned under *Alias*. The attribute and value *delete="1"* indicate that when the cartridge is uninstalled, the PageType will also be deleted from the database.

Template Name ... defines the logical section of the Web page where the content will be displayed. *Filename...* indicates which HTML file is used to display this section.

PageTypes are hierarchical. The basis is the *BasePageType*. The BasePageType divides the page generally with a reference to the corresponding original template. See *Code example 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>
```

Code example 12: BasePageType definition

The basic sections used on all the pages are assigned in this *BasePageType*. Depending on the desired effect, additional PageTypes define these sections even more finely thereby resulting in additional more specific sections. This means that the *BasePageType* is the source for inheritance within the PageType.

The starting point for the PageType hierarchy and the direction of the inheritance are illustrated in *Figure 11*.

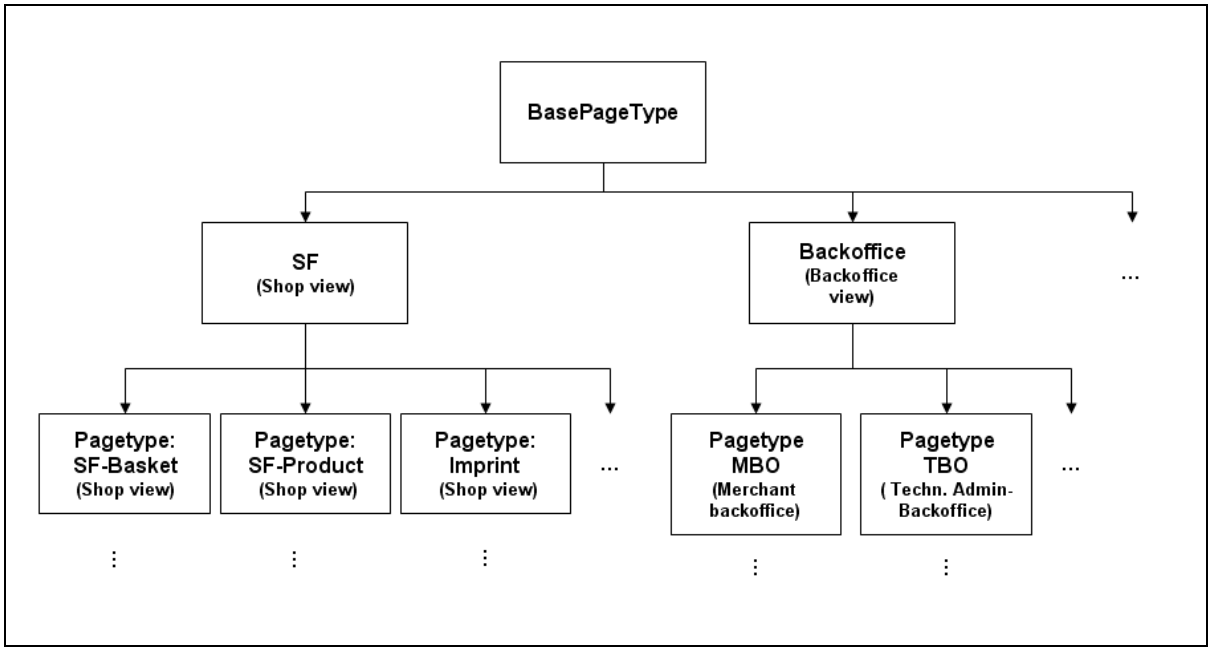

Figure 11: PageType hierarchy (excerpt)

When you define a PageType, you can use inheritance and extensions to apply particular sections, to assign another template to overwrite specific sections or to also extend specific sections with subsections. A simple example is shown in *Code example 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>
```

Code example 13: PageType with inheritance and overwriting an section

This PageType is defined in the *Design* cartridge and assigned to the object class *Shop*.

Use *Base=...* to define the parent PageType from which this particular PageType is derived. *SF-Shop* is derived from *SF* and therefore inherits all the section definitions and templates.

Although in this case, the logical section called *Content* has been applied, it will be displayed using a different template. This is done by assigning another template, *SF-Shop.Content.HTML*.

The *View* action is assigned to the *Shop* class of this PageType using the XML tag *ViewAction*. This means that this action will display the object using this PageType. For more information, see *UE 6:* , *on page 189*.

8.2 Display Levels

A PageType always finds its complement at the display level, since for every logical section introduced, a template which describes the HTML layout has been defined.

From the HTML point of view, PageTypes are collections of templates that determine how and where functional content and data are displayed on the HTML pages. You can say they build the framework or container for the current content that is inserted, depending on the function.

Templates are structures with the same hierarchy as page types. The *BasePageType* defines the *Page*, *Head* and *Body* logical areas. Parallel to this, the respective templates are defined. See *Code example 12*. While these templates define the general structure in HTML, as the number of levels in the hierarchy increase, the layout function becomes ever more specialized.

8.2.1 Original Template

The original template defines only the HTML page itself and thereby fulfils the general layout requirement. See *Code example 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>
```

Code example 14: Original template

Each INCLUDE statement within the data functions as a type of placeholder and is replaced at runtime with the current template. For more information about INCLUDE statements used as TLEs, read chapter *TLE*, *on page 61*.

The templates defined here are filled with general content. They represent a fallback variant in case the template is not overwritten by a successor PageType.

For the original template, you see the content of the corresponding templates *BasePageType.Body.HTML* and *BasePageType.Head.HTML* in *Code example 15* and *Code example 16*.

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

Code example 15: INCLUDE template for the original template -*Head section*

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

Code example 16: INCLUDE template for original template -*Body* section

You can see that this template makes only the body structure of a page available. The actual content is displayed using templates provided by subordinate PageTypes.

8.2.2 Template Hierarchy

The templates at the next level define the content and structure of the *Head* and *Body* sections. However, general statements are hard coded and the variable content is filled by INCLUDE statements.

The structure of the HTML hierarchy is based on the PageType hierarchy created in the database using XML. See *Figure 11*.

Level 2 PageTypes display the design of the different main action levels: *Back office* for the administration pages for the technical administrators (TBO), the business administrators (BBO) and the merchants (MBO) and also *SF* for shop page layout.

These templates determine the general layout of the HTML pages for each working level and therefore define the uniform layout of HTML pages at each level for all functions.

Since the original template for the HTML page only describes the body very generally, the templates of the second level describe this area more specifically. *Code example 17* shows an example for the body of back office pages.

```
<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>
```

Code example 17: defining the body for back office pages

In contrast, *Code example 18* shows an example for the body of storefront pages:

```
  <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
#Session.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>
```

Code example 18: defining the body for store front pages

This defines the basic layout of the HTML pages of the shop. Here, you can also see the structure in *Figure 7*.

PageTypes based on individual object classes make up the next level in the hierarchy. This means that for every object class, at least one PageType is defined that assigns the actual layout of the individual objects on the HTML page. Examples of these object classes are *Basket*, *Product* and so on.

The objects are displayed in the working area depending on the action executed and the resulting data.

The same principle also applies to the subordinate levels in the hierarchy that display the object in different views.

8.2.3 Object Method *template*

In the HTML files, template names are used mainly in connection with the TLE statement #*#INCLUDE*, for example, *#INCLUDE("Content")*, see *Code example 18*.

Each object can change the template name using the *template* method. The original template name is passed to the method, for example.

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

The template the object uses is defined in this method. In most cases, the template that is used is the one where the name has been passed.

In special cases, other template names or mechanisms for selecting specific templates can be defined in this method.

This is applied to objects in the *LineItem* class and its derived classes, for example. This option has been used for applying class-specific templates. The template name is thereby applied and extended with the class alias of the current object in such a way that the resulting template name is as follows:

```
<templatenameClassAlias>
```

If a template with this name is not defined, this method replaces the current class alias with the class alias of the next higher object. In this way, the line of inheritance of the object is searched until a template is found. If no valid template name is found during the search, the template simply labelled as *template name* is processed.

The following example demonstrates this: While the statement in *Code example 19* is being processed, the TLE compiler reads out the template name *ContentLine*.

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

Code example 19: Example of a template call

The #WITH statement assigns the context to an object belonging to the *LineItemShipping* class. This object replaces the template name with *ContentLineLineItemShipping*. If the template is not defined, the method falls back on *ContentLineLineItem* since *LineItem* is the parent class of *LineItemShipping*. If the method finds no other template along the object structure, the general *ContentLine* template is processed as the last possible variant.

This mechanism allows the method to define a general use template and to assign specific layouts to each class using the simple template name. This is used, for example, in connection with the layout of LineItems. For more information, see *LineItems*, *on page 172*. Also see the API documentation at *DE_EPAGES/Order/API/Object/LineItem*.

8.3 Processing PageTypes

As mentioned in *Logical Structure*, *on page 39*, the ViewAction that is used to display an object as well as the templates that are used for the layout are defined in the PageType.

After the action has been called, the processor starts with the *Page* template and then processes the INCLUDE statements until the entire page has been regenerated and can be displayed.

In this way, the same object can be displayed in different environments depending on which PageType the ViewAction has been assigned to.

Figure 12 shows the layout of a product in the shop. *Figure 13* shows the layout of the same product on the merchant administration page.

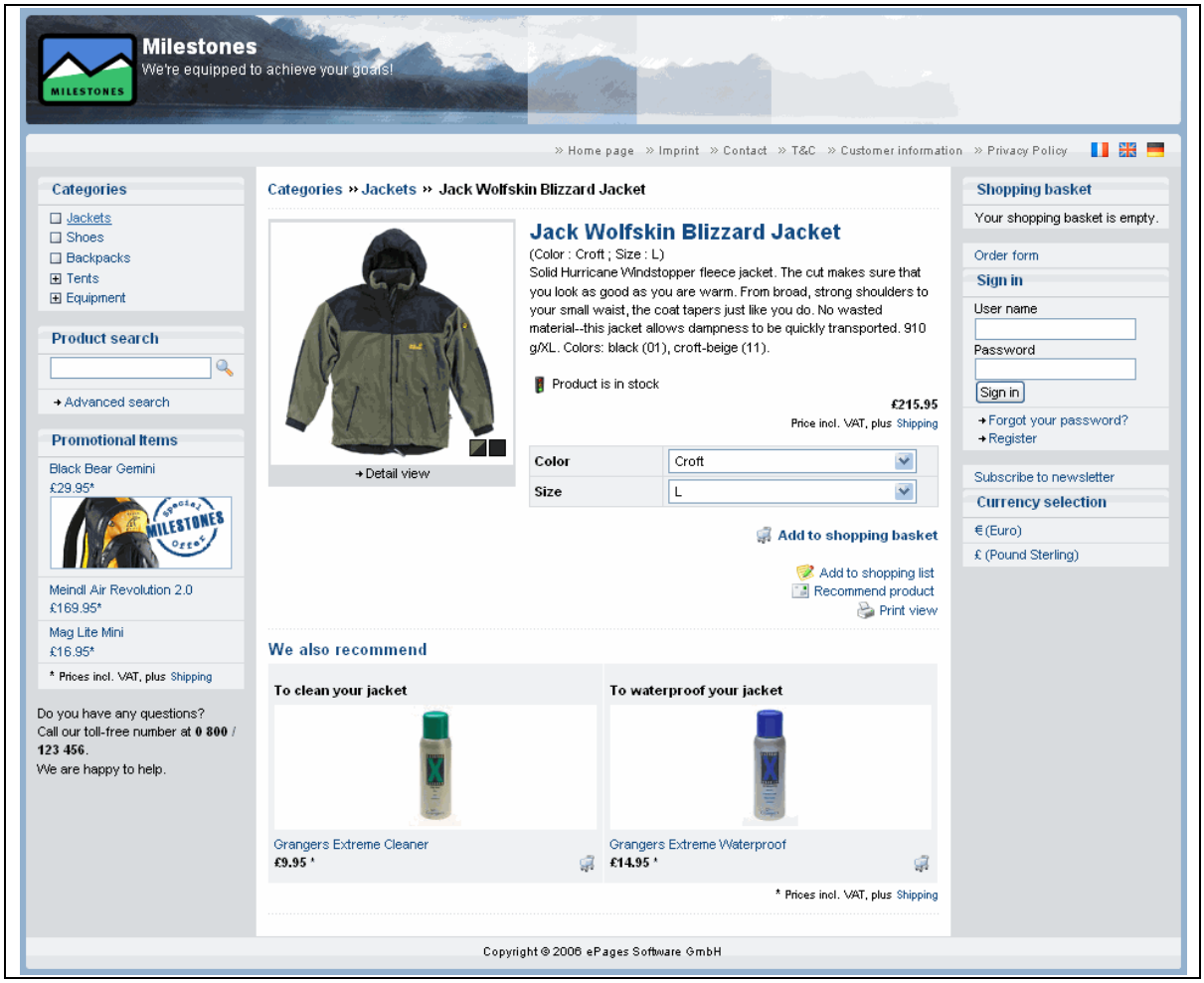

Figure 12: Layout of a product using SF PageTypes

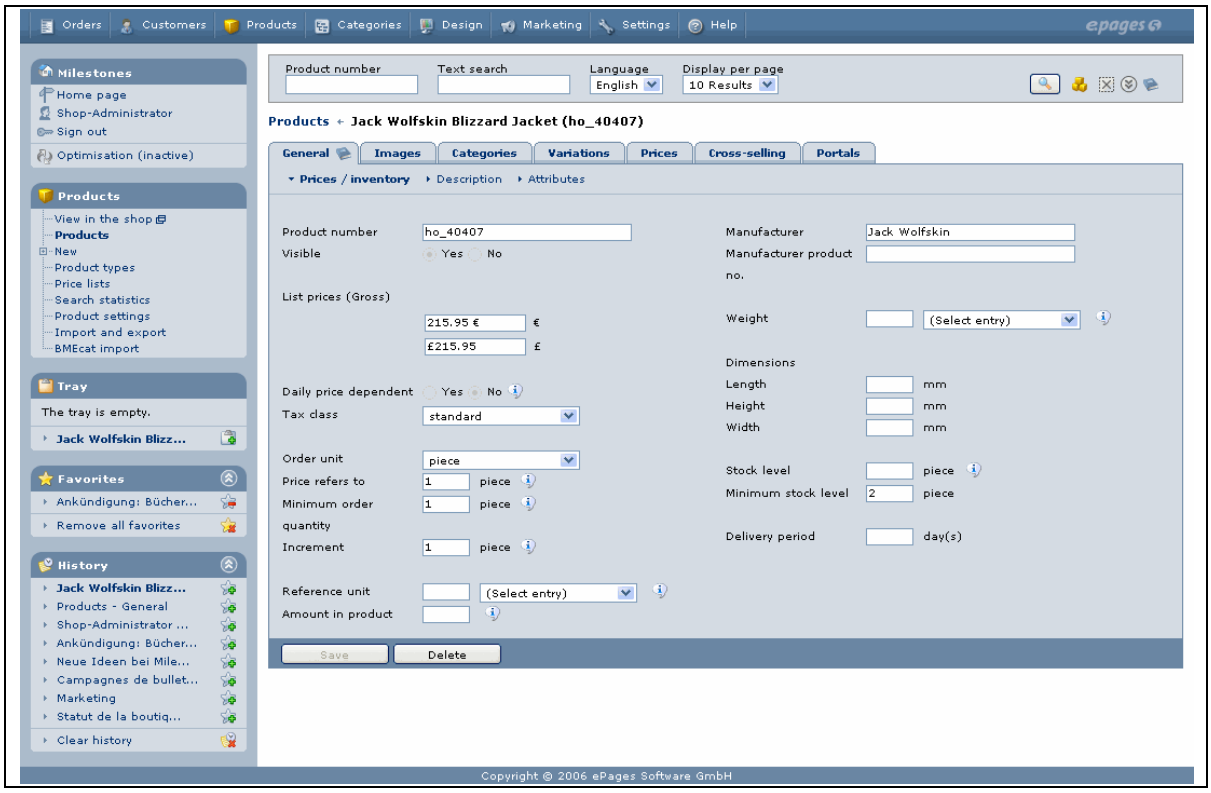

Figure 13: Layout of the same product using MBO PageTypes

9. Multiple Languages–Language Tags

With ePages 5, it is simple and easy to implement multiple languages into your Web application. For this purpose, we have introduced the *language tag* extension. Language tags are inserted as placeholders in templates at any position where variable content should be displayed and depending on the language chosen.

Note: Language tags are not placeholders for language-dependent values in the database such as product attributes. Language tags are also not created or managed by the shop operator himself.

The language-dependent content for language tags is stored in easy-to-edit XML files. In this case, the keyword has the same name as the language tag. Using the same name guarantees the reference between the placeholder and the content.

This means that you only need to generate and maintain one template set in order to consistently insert these language tags at language-sensitive positions in all the templates.

9.1 Syntax for Language Tags

The syntax of a language tag is very simple:

{‹tagname›}

A language tag begins and ends with curly brackets. The variable name you have chosen is located between the brackets. This variable name is also reused in the XML file.

The reference for this in the XML language file is structured as follows:

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

Code example 20: Definition of a language tag in an XML file

Note: All language tags are *case sensitive*. This means that a difference is made between upper case letters and lower case letters.

In some cases, a language tag contains text that should not be translated. This could be a variable, for instance, that is generated during runtime, for instance. These texts should be enclosed in a *‹notrans›* tag. See *Code example 21*.

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

Code example 21: definition of language tag with non-translatable content

For some language tags, it is necessary to provide some additional information about the tag in order for the translator to be able to translate the tag correctly. For example, if the word *Export* is to be translated by

itself, it is important to explain whether the verb *Export* for a button or the noun *Export* for a header, for example, is meant. To avoid such issues, a language tag can be supplemented with additional information in the XML file. To do so, use the *META* attribute. See *Code example 22*.

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

Code example 22: using the *Meta* attribute

This tells the translator in the target language to use the correct word for the noun *Export*.

9.2 Using XML Language Files

In the following example, you can see how multiple languages are implemented on an HTML page using language tags and the associated XML files. Only one template is used regardless of the number of languages to be displayed.

The principle way of doing this is demonstrated in the example of the *SF.LoginBox.HTML* file, which displays the sign-in area in the storefront. To give you a better overview, the sections of code used have been reduced to a few language-relevant parts in the source code.

First, an HTML page is designed and coded:

```
...
    <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>
...
```

Code example 23: Source code for multiple languages

This is displayed in the browser as follows:

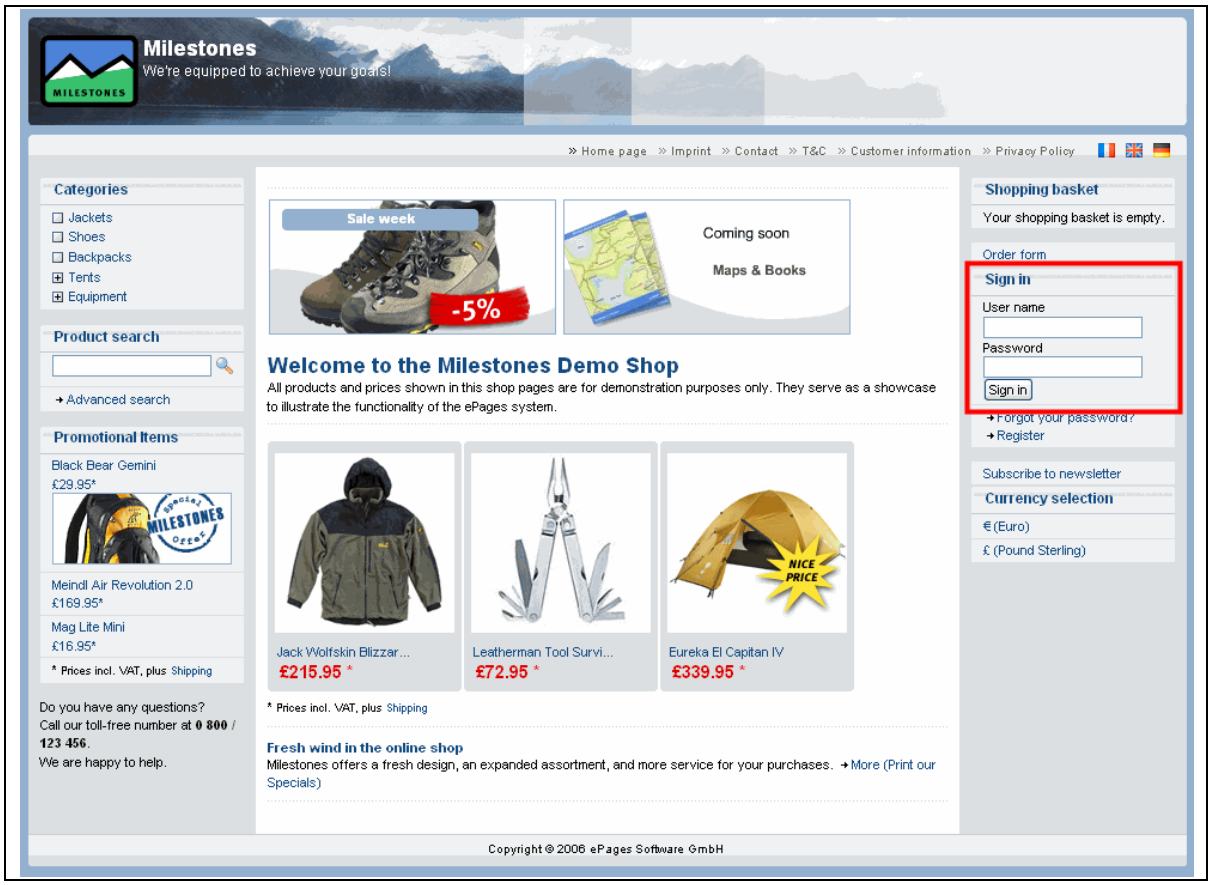

Figure 14: display of the language-independent source template

Now the question is, which parts of the text on the HTML page should be displayed as language-specific. This is where language tags come into play. See *Code example 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>
...
```

Code example 24: inserting language tags

Now, instead of static IDs, the browser displays the language tags in the corresponding positions:

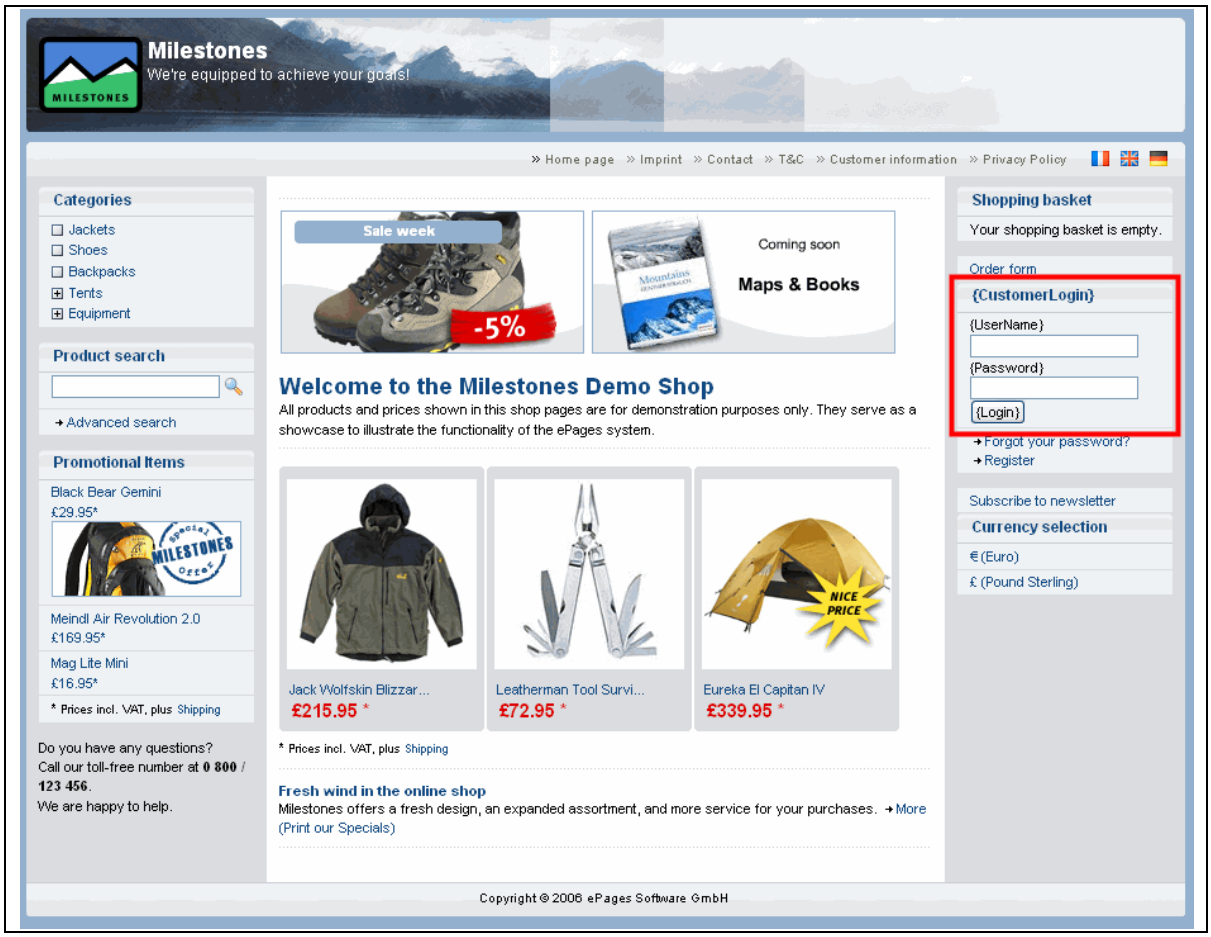

Figure 15: display of the template with language tags

The language-dependent information must now be included in the corresponding XML language files. Every language has a separate XML file.

The XML file is saved in the same directory as the HTML file and has the same name plus an extension that indicates the language used. In our example, if the file for the template is named *SF.LoginBox.HTML*, this means that the associated XML language file for German is named *SF.LoginBox.de.xml*, and for English, it is named *SF.LoginBox.en.xml*.

The language content for every language is inserted into this XML file, see *Code example 25* and *Code example 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>
```

Code example 25: XML entries for the language tags in German

```
<?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>
```

Code example 26: XML entries for the language tags in English

The encoding is indicated for every XML file. The important thing is to make sure the encoding and the character set used in the file agree. You can use a different encoding for every language file, if necessary.

Now when the template is processed, the XML file for the requested language is read out. See *Figure 16* and *Figure 17*.

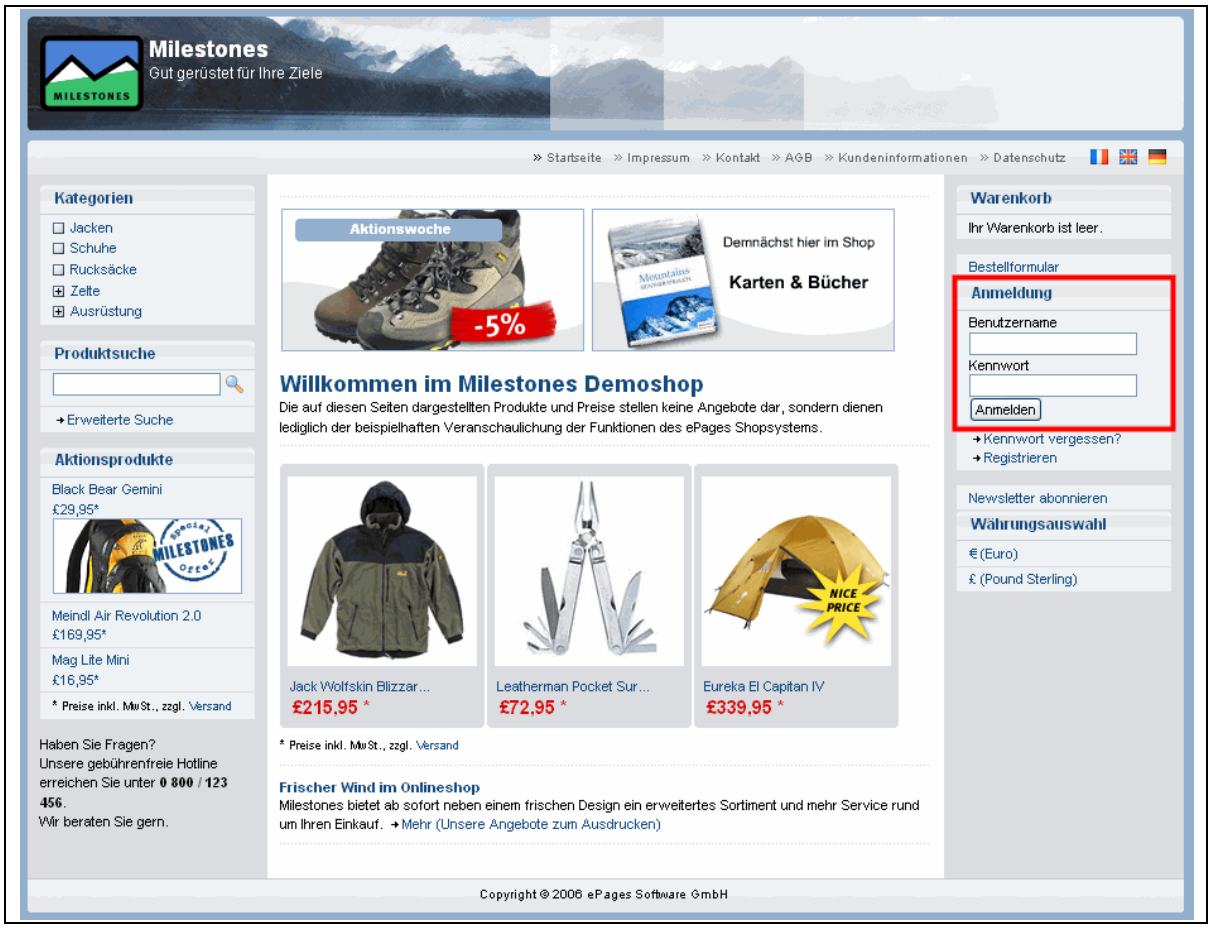

Figure 16: Web page in German

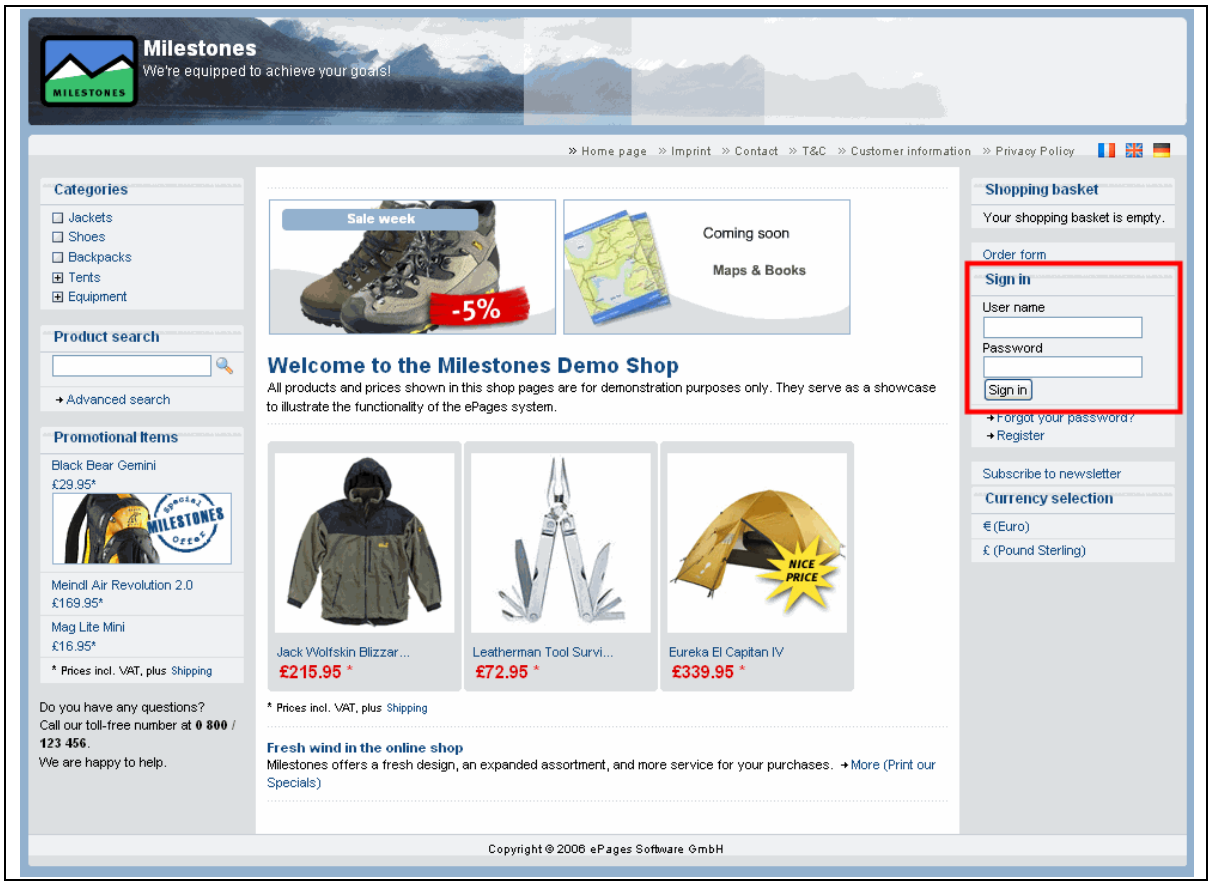

Figure 17: Web page in English

Using this principle, you can insert language tags wherever you want to display language-dependent content and extend the corresponding XML file.

This makes adding still another language very simple. You only have to create the XML file and collect the corresponding language entries, for example, for French, the file would be *SF.LoginBox.fr.xml* with the entry:

```
<?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>
```

Code example 27: XML entry for the language tags in French

The ISO 3166-1 alpha-2 code abbreviations are used for the country IDs.

Note:

1. The prerequisite for displaying an additional language is activating that language for the shop. To find out how to activate a language, refer to the corresponding administration manuals.

2. If the changes you made to the XML language files are not immediately visible, delete the [ctmpl] directories. This may also be necessary when you copy files that have different time stamps.

9.3 Overlaying XML Language Files

You have the option of overlaying XML language files, that is, you can use specialized XML files that overwrite general content.

For this purpose, three different XML language files have been defined. The sequence in which these files are processed is determined by the system.

In the sequence they are processed, they are as follows (example for back office templates in English):

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/‹templatedirectory›/‹templatename›.en.xml*

During this process, the XML files become increasingly specialized in their language content as reflected in the numbering. This means that the first line contains general translations that apply to the entire application and can be found on all the pages, while the third line contains translations that refer to a specific template and also are only used in that template.

For example, the file *%EPAGES_CARTRIDGES%/EN_EPAGES/Dictionary/Templates/Dictionary.en.xml* contains the translations for texts such as *Delete* and *Save*, that are displayed the same way on almost every HTML page. For an example, see *Code example 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>
…
```

Code example 28: Selection from *Dictionary.en.xml*

The global entry for *RunBatchAction* is used in the following location

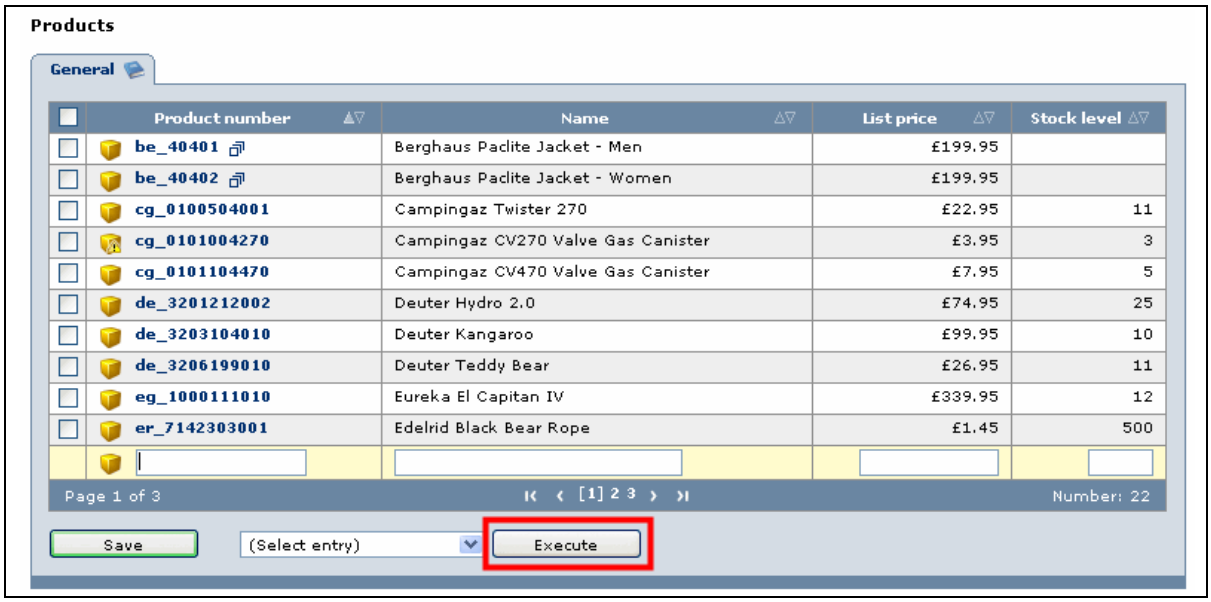

Figure 18: Displaying *Execute* based on *Dictionary.en.xml*

The file *‹templatename›.en.xml* (3.) on the other hand, contains only the language information for the template of the same name.

You can overwrite this language tag by using the same language tag name in more than one of these XML language files. For a *Keyword*, the value used is the one set in the most specific of the three files mentioned above for this keyword.

As an example, we will overwrite the text in *Figure 18* using a type (3) file. The template that displays the page in *Figure 18* is called *MBO-Products.Content.HTML*. In the same directory, you can find a file named *MBO-Products.TabPage.en.xml*. We add the following entry to it:

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

Code example 29: overlaying a general language tag

This means that the content from the general language file for the language tag *RunBatchAction – Execute from Code example 28* is overwritten with the entry *Start action*. After processing the template, the following is displayed:

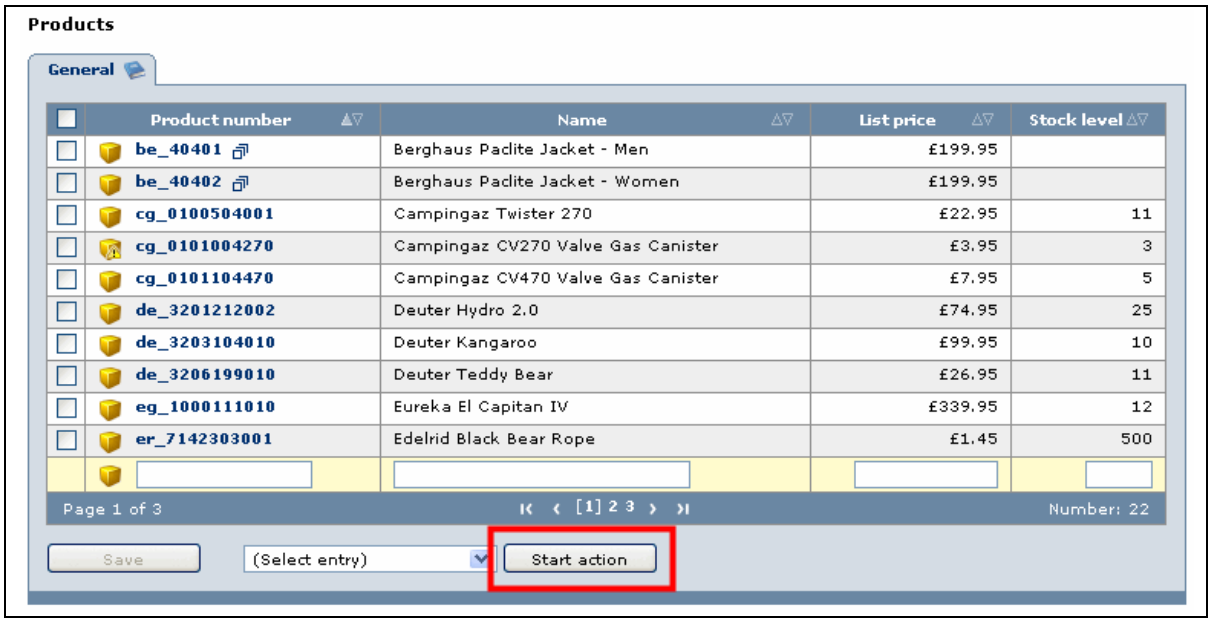

Figure 19: result of the overlaying

This overlaying is valid as long as the keyword *RunBatchAction* is present in the *MBO-Products.Content.en.xml* file. If the keyword is deleted from the file or the file itself is deleted, the original value in the *Dictionary.en.xml* file is displayed.

The XML language files for other languages are created the same way. The *en* is replaced with Language Code (lc) in the file name and the translations are entered for the language tags.

For the language files, the same order applies:

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/‹template directory›/‹template name›.lc.xml*

When the template is processed, the system is set to the currently displayed language. If the current language is English, the language tags are replaced by the content of the XML language files with the identifier *en*. If the display language is German, *de* is the identifier evaluated.

This makes it easy to extend to other languages, for example, French – *fr*, Spanish – *es*, and so on.

> Note:
>
> 1. Make sure that the language that you would like to display is also activated for the application! To activate a language, refer to the corresponding administration manuals.
>
> 2. If you make any changes in the original files, they can be overwritten by a later update or upgrade. In order to keep your installation updateable, use the option of overlaying the original files. This also applies for dictionary files. For more on this, see Overlaying Templates, on page 35.

You have the option of using a script to check whether all the language tags for a selected database and language have been completely replaced or whether redundant tags appear in the localization files. For this, use the following script:

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

The call

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

displays the available options and call parameters. One possible example of this can be seen in the following:

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

9.4 Localising Database Content

For localising database content such as attribute names or attribute descriptions, definitions and language-dependent content are also separated. The following example demonstrates this.

The features of the *Features.xml* file are defined in the *Product* cartridge.

```
<?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>
```

Code example 30: defining features

The name and description of a feature should be shown in the MBO language-dependent One language file per language is created for this. The following naming convention applies:

Translation.<filename>.<language code>.xml

For our example, (*Features.xml*), the following applies:

- for German: *Translation.Features.de.xml*
- for English: *Translation.Features.en.xml*

The corresponding English language file, which corresponds to *Code example 30*, can be seen in *Code example 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>
```

Code example 31: English language file for *Features.xml*

The German language file which corresponds to *Code example 30* can be seen in *Code example 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>
```

Code example 32: German language file for *Features.xml*

This has the following advantages:

- Consistent translation of all localisable strings
- Translation of localisable database content by cartridge is possible
- Importing the translation for multiple languages during installation of the cartridge is possible
- XML files with object definitions do not need to be changed if the localization changes

This concept applies to the following types:

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

10. TLE

Using template language extensions (TLE), it is possible to dynamically load, process and display Web page content. Web sites that use standard HTML can only generate "static" Web pages with very specific content. This means that Web designers have to create a separate Web page for every possible display option.

ePages removes this burden from the Web designer and the system resources by generating "dynamic" Web pages at runtime. When the customer clicks a link in the storefront, ePages chooses the templates to be used and creates the page with data from the database according to the TLE information in the template. By reading out and evaluating the data at runtime, the resources required for complex pages are reduced.

TLEs are made up of variables and statements.

10.1 Syntax for TLE

All TLE variables and statements are preceded by a hash symbol (#):

#‹ObjectAttribute›

Example: `#Shop.NameOrAlias`

or

#‹variablename›

Example: `#Alternate`

or

#‹TLEstatement›

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

Note: All TLE variables and statements are case sensitive. This means that a difference is made between upper case letters and lower case letters.

In the following chapters, the syntax for TLE variables and statements will be described and demonstrated using examples. All the examples are taken from the original files.

Before you work with and modify the examples in the files, please read *Overlaying Templates*, *on page 35*. After this, use overlaying to not only practice using TLEs but to also make sure that your ePages 5 installation functions properly.

10.2 TLE Variables

TLE variables are placeholders for dynamic information from the database. With the help of these placeholders, data are displayed that might change whenever a page is opened. You add TLE variables to the HTML file just as you add normal text. When a specific page is opened, the values of the variables are immediately determined via a database query and displayed together with the HTML page.

Sources for values for TLE variables are:

- Object attributes,
- URL parameters,
- Input data from HTML forms,
- Cookies,
- ViewActions
- TLE functions,
- Dynamic TLE variables
- Session parameters

The TLE variables from the sources mentioned can be inserted in any template.

The active object is the object which is opened by a ViewAction. Through its page type, the necessary templates are provided to show the Web site in question. See also *Processing PageTypes*, *on page 45*. This object creates the context for the available TLE variables.

For example, if you would like to display product data and have called the action for displaying a product, the active object is *Product*. In order to display the name of the product in the template, add the TLE variable *#NameOrAlias* at the appropriate position. Through this action, the system recognizes that it is dealing with the context of the object *Products* and fills the variable with the name of the product.

If you instead execute the display action for users, the system recognizes the object *User* as the context. Here, for example, *#Name* is used to display the name of the user on the Web page.

To display data from one context in a different context, you need to indicate the entire "context path" for the TLE variables or change the context.

If you would like to display, for example, the name of the user currently signed in when displaying product *A*, you need to switch the context for this display since the current user is not assigned to the object *Product A*.

The current user is saved in the current session. Therefore, for the TLE variable indicate the following: *#Session.User.NameOrAlias*.

There are five contexts for accessing data for both the Merchant Back Office and the store front:

- INPUT
- Session
- System
- Shop
- Current context (for example, products, shopping basket)

Any object property can be queried using a TLE.

Examples for using TLE variables in templates can be found in *Code example 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
…
```

Code example 33: using TLE variables

The data from the various sources are read as follows:

Object: ***#‹(context.)attributename›***

URL parameters, form field contents: ***#INPUT.parametername***

Interim results: ***#‹variablenname›***

Note: Certain TLE statements change the context. See *#WITH*, *on page 67*, and *#LOOP*, *on page 67*.

Anther option for displaying data independent of the context is to use the *Object.Child* attribute. You can use this attribute to display practically any object anywhere if the object ID is known. For instance, you can take the shop address on the Contact Information page and also display it on a catalogue page:

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

10.3 TLE Statements

TLE statements are ePages-specific commands used to display templates/Web pages depending on the content.

This means that you can process specific template sections according to certain conditions or easily display variable amounts of data using loop statements.

You can insert the following statements:

10.3.1 #IF

With the help of an #IF statement, you can process sections of a template according to certain conditions. You can create simple conditions with #IF and #ENDIF.

For more complex conditions, you can use the #ELSIF tag and the #ELSE tag. The value of a TLE variable is determined by evaluating the #IF statement in real time.

Conditions and value comparisons can be used not only for character strings (TLE variables, and so on) but also for numerical values.

Syntax:

#IF (‹expression›) ... #ENDIF

or

#IF (‹expression›) ... #ELSE ... #ENDIF

or

#IF (‹expression›) ...

#ELSIF (‹expression›) ...

#ELSIF (‹expression›) ...

...

#ELSE ...

#ENDIF

You can see an example of a complex #IF statement in *Code example 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
```

Code example 34: example of an #IF statement

10.3.2 #INCLUDE

The #INCLUDE statement lets you imbed a template within another template. In this way, you can use elements (for example, icon bars) in more than one element using a simple #INCLUDE statement. The imbedded template is inserted at runtime instead of the #INCLUDE statement. Any changes in the HTML code for the imbedded template are immediately applied to all the templates using the corresponding INCLUDE statement.

Syntax:

#INCLUDE("<templatename>"[, "NoDebug"])

Example:`#INCLUDE("Content")`

#INCLUDE(#Template)

Example: `#INCLUDE(#Template)`

The second variation is used when the template name is not yet known but is provided dynamically. This variation is typically used together with #BLOCK and Menu. See original templates.

The *NoDebug* parameter is optional. If it is used in an INCLUDE, the comment line with the template information in the source is not shown. For more on this, see *Template Debugging*, *on page 36*. Use these parameters in templates in which the HTML comment characters are not known and that could lead to errors. An example of this are CSS files with INCLUDE.

10.3.3 #LOCAL

Use this statement to define an area of validity in the template. Here, specified variables or their current values are valid only in this area.

Syntax:

#LOCAL("<variablenname>", <wert>) ... #ENDLOCAL

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

Code example 35: example of #LOCAL statements

In this example, the variable *TaxClassID* in #LOCAL is given a new value that is valid until #ENDLOCAL. After #ENDLOCAL, the variable is given its original value or becomes invalid if it did not exist before the #LOCAL statement.

10.3.4 #SET

Use this statement to set a variable for the global context. This means that you can access this context from anywhere within the whole template. A variable set with #SET can be accessed up to the end of the HTML page that called the template. Even when the variable is specified in a template that is loaded into the page per INCLUDE, it can be accessed up to end of the HTML page, that is, from the *master template*.

In general, you should always restrict #SET using #LOCAL/ #ENDLOCAL.

Syntax :

#SET("<variablename>", <value>)

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

or

#SET("variablename>", <expression>)

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

10.3.5 #GET

You use the statement to request a TLE variable and to access its value.

Syntax:

#GET(<variablename>)

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

Example: `#GET("Number")`

or

#GET(<expression>)

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

In the case of *#GET("Number")*, you can dispense with GET. The expression *#Number* is equivalent. If the name of the attribute to be displayed is first detected at runtime, use a form such as *#GET (#Attribute.Alias)*.

10.3.6 #CALCULATE

The statement returns the result of a calculation expression.

Syntax:

#CALCULATE(‹expression›)

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

10.3.7 #WITH

This statement triggers the current context reference for TLE variables and defines local validity. This means that you set a new context that is only exists within the #WITH statement.

Syntax:

#WITH(‹expression›) … #ENDWITH

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

Code example 36: *#WITH* statement

In *Code example 36*, #WITH is used to set the context to *Shop.Categories*. This is why the #*ID* query also returns the shop category ID. The product category context was in effect up to that point for the template, in which this #WITH statement is imbedded. A simple query for #*ID* would have resulted in the ID of the current product category.

10.3.8 #LOOP

You can use #LOOP statements to display different list elements. These elements can be categories, products, items in the shopping basket or data structures that ePages provides in arrays. It is easy to create simple lists using #LOOP statements in templates.

Syntax:

#LOOP (‹loopvariable›) … #ENDLOOP

Loop variables apply to the template locally. The entire HTML code and TLE between #LOOP and

#ENDLOOP is repeated for each element in the loop variable.

Within a LOOP statement, the template context is exited and the context of the loop variables applies. For more on this, see *TLE Variables*, *on page 61*.

You can see a #LOOP example in *Code example 37*.

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

Code example 37: #LOOP

Note: The sequence of the list elements is always determined by sorting in the back office. For more information, refer to the *Sorting in Tables* chapter in the *Merchant User Guide*.

10.3.9 #JOIN

This statement is used to initiate a loop that returns a character string. You can use this to specify the character to be inserted between the individual elements as a separator.

The statement thereby extends the *#LOOP* statement with a separator definition.

Syntax:

#JOIN ("‹character›", ‹loopobject›) … #ENDJOIN

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

In this example, all the users are listed, separated by a comma.

10.3.10 #FUNCTION

Use this statement to call TLE functions from the template. The function must be registered in the TLE compiler.

Syntax:

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

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

The functions each return a single value.

10.3.11 #BLOCK

Use the *#BLOCK* statement to transfer template code to a function. This code is processed when a function is called. A value is returned, usually a string which contains the processed template code. *#BLOCK* is an extension of the *#FUNCTION* around the template or the template code up to *#ENDBLOCK*.

Syntax:

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

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

Using the code in the example above, all the entries defined for the *Content* menu are displayed on the page. In addition, the *Menu* function is called using the *Content* parameter. This means that the entries for the menu with the name *Content* are read in sequence from the corresponding PageTypes. In the TLE *#Template*, the name of the template that displays the corresponding entry is passed. In this way, the template displays for the individual entries in the calling template are integrated sequentially.

10.3.12 #WITH_LANGUAGE

You can use this statement to switch the current language context.

Syntax:

#WITH_LANGUAGE(‹variablename›) … #ENDWITH_LANGUAGE

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

Code example 38: example of #WITH_LANGUAGE

In the example, a loop is executed for all active shop languages. #WITH_LANGUAGE is used for each language to set the corresponding language context so that the associated description can be read out. #ENDWITH_LANGUAGE "turns" this language "off" again. The language in effect before the *#LOOP* statement is again current.

10.3.13 #REM

This statement lets you define areas in your template that should not be processed. You can insert comments, notes, and so on in these areas. Developers can use this option to hide code during the development phase.

Syntax:

#REM … #ENDREM

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

These comments are also not visible in the View Source window in the browser.

10.4 Error TLE

10.4.1 #FormError

Use this statement to query whether the form last called contains errors. The function returns a value of true if an error has occurred.

Syntax:

#FormError

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

10.4.2 #FormError_‹InputField›

You use this statement to query errors in entry fields.

Syntax:

#FormError_‹inputfieldname›

Code example 39 illustrates a typical application.

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

Code example 39: example of #FormError_‹InputField›

Here, each product price is listed sequentially in entry fields. If an error occurs for a product in the *Price* field, the corresponding entry field is highlighted in red.

10.4.3 #FORM_ERROR

This statement functions exactly like #FormError_‹InputField› but is used when the ERROR_Parameter is first known at runtime.

Syntax:

#FORM_ERROR("‹inputfieldname›")

#FORM_ERROR(‹variablename›)

The example in *Code example 39* with a known parameter is as follows:

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

Code example 40: example of #FORM_ERROR

In *Code example 41*, you see an example for using parameters that are first known at runtime.

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

Code example 41: example of #FORM_ERROR and a variable parameter

10.4.4 #FormErrors.‹…›

The *TLE* #FormErrors contains all the information necessary to generate a meaningful error message. Usually the *Reason* parameter is used to show an error description.

Syntax :

#FormErrors.Reason

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

For more details, see *Error Handling Templates*, *on page 99*.

10.4.5 #WITH_ERROR

Use this statement to re-incorporate entries from erroneous forms into the template. The important thing is that the field name is identical to the TLE.

Syntax:

#WITH_ERROR(‹logical expression›) … #ENDWITH_ERROR

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

Code example 42: example of #WITH_ERROR

Make sure that the replacement occurs only in the current context. If the context is changed, for example, with #LOOP or #WITH, you have to reset #WITH_ERROR.

10.4.6 #ERROR_VALUE

Use this function to handle errors of selection fields. This allows the code to be much easier and more effective to read.

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
```

Code example 43: example of #ERROR_VALUE

Depending upon which result the error function provides, a valid value is transferred to the calling variable.

10.5 Formatting TLE Variables

TLE variables can be formatted for display on a Web page.

Formatting statements are attached to TLE variables using square brackets:

#‹TLE-Variable›[‹formatting instructions›]

You can use the following formats:

Table 8: Formatting statements for TLEs

Formatter	Description	Example	Display
[money]	The value is displayed as a price. The currency format used is the one specified by the current user preferences.	#Price[money]	25,95 € $ 25.95
[float]	The value is displayed as a decimal number. The format used is the one specified by the current user preferences.	#Quantity[float]	German: 5.325,26 English (US): 5,326.26
[integer]	The value is displayed as a decimal number. The format used is the one specified by the current user preferences, however without decimals.	#Quantity[integer]	German: 5.325 English (US):5,326
[LC]	LowerCase All the letters in a character string are returned as lower case letters.	#Name[LC]	DOE → doe
[LCFIRST]	Only the initial letter of a character string is displayed in lower case.	#Name[LCFIRST]	DOE → dOE

Formatter	Description	Example	Display
[UC]	UpperCase All the letters of a character string are returned as upper case letters.	#Name[UC]	doe → DOE
[UCFIRST]	Only the initial letter of a character string is displayed in upper case.	#Name[UCFIRST]	doe → doe
[space:n] [space:-n]	The character string in question is displayed either left-aligned or right-aligned and is filled with as many empty spaces necessary, either to the right or to the left, until the total number of characters correspond to the number indicated in the brackets. The n functions here as a placeholder for an integer. This format makes sense only for plain text output, not for HTML.	#Name[space:12] #Name[space:-12]	Doe →|Doe | Elbe →|Elbe | doe →| doe| Elbe →| Elbe|
[slice:n] [slice:n]	The character string to be displayed is cut off and ended with three periods (...). The "n" here is a placeholder for the number of positions in the length of the result. The three periods are included in the character count. If the character string is shorter than n, no periods are added. If you have a negative parameter, it will be counted from the end of the string, the three periods (...) will be counted.	#Name[slice:9] #Name[slice:-9]	DOE → RUBENS... → ...ermann
[webpath]	This indicates the path to the requested files on the server. The prerequisite for this is that the attribute to be displayed is of type file or language-dependent file.	#Image[webpath]	/WebRoot/Store/SF/Shops/Demoshop/.../example.gif
[html]	Replaces ‹›&" with the corresponding HTML entities (< > & &quot). This format is the default format for all TLE variables.	#Text or #Text[html]	
[nohtml]	Removes all HTML tags (replaces them with spaces) Entities are changed ü -› ü	#Shop.Slogan[nohtml]	
[0]	The TLE variable should not be formatted. If the TLE contains content from entry fields that is already formatted in HTML, for example, this formatting should not be changed. This affects all the fields that have been marked as HTML fields in the back office.	#text[0] #IF (#Shop.BasketOverText) #Shop.BasketOverText[0] #ENDIF	
[uri]	All non-alpha-numeric characters in the variables values are URL-coded. Non-alpha-numeric characters are any character except 0...9, a...z, A...Z, "_".	#Path[uri]	/Shops/Demo.Shop → %2FShops%2FDemo%2EShop
[url]	All non-alpha-numeric characters in the variables values are URL-coded. Non-alpha-numeric characters are any character except 0...9, a...z, A...Z, "_" and "/".	#Path[url]	/Shops/Demo.Shop → /Shops/Demo%2EShop
[date]	A date TLE is displayed in the currently-set date format (only date).	#Now[date]	Oct 5, 2005 11:49:33 AM → 05.10.05

Formatter	Description	Example	Display
[datetime]	A date TLE is displayed in the currently-set date format (date and time).	#Now[datetime]	Oct 5, 2005 11:49:33 AM → 05.10.05 11:49
[time]	A date TLE is displayed in the currently-set date format (only time).	#Now[time]	Oct 5, 2005 11:49:33 AM → 11:49
[px]	For TLEs with size information (length and so on.) This is used for style sheet information.	#ContentParagraph Size[px]	10 → 10px
[js]	For TLEs that are used within a JavaScript character string. Here, quotation marks and line breaks must be tagged with control characters.	onclick= ”changeSample('#Description[js]');”	I Don’t know → I Don\’t know
[preline]	Line breaks are converted to ‹br /› tags.	#CustomerComment [preline]	Hello, Please deliver punctually this time!!! Regards, Mr. Customer → Hello,‹br /› Please deliver punctually this time!!!‹br /› Regards,‹br /› Mr. Customer
[neg]	Displays values in the negative	#Quantity[neg] #Money[neg,money]	5 → -5 25.95 € → -25.95 €

Formatters can be separated by commas, for example

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

This will be interpreted as follows:

Through the first formatting *-20* of *#Shop.NameOrAlias[-20*] a string is returned that is 20 characters too long. The following characters are filled up from the front:

```
"          Milestones"
```

Through the second formatting *30* the string is extended to 30 characters. This is displayed in the browser as follows:

```
"          Milestones          "
```

Caution: You are responsible for a logical combination of formatters. No plausibility test is done.

10.6 Operators

You can use the following operators in TLE statements:

Table 9: operators for comparing numerical values

Operator	Description	Description	Code example
EQ	Equal (EQual)	Valid if parameters are equal	#IF(#Class.Alias EQ "Category")
NE	Not equal (Not Equal)	Valid if parameters are not equal	#IF(#Class.Alias NE "Category")
.	Concatenation	The linking of character strings	"Mon" . "day" = "Monday"
IN	Contained in		

Table 10: operators for comparing numerical values

Operator	Description	Description	Code example

NEQ
==
	Numerically equal
	(Numerically EQual)
		Valid if numerical parameters are equal	IF(#BillingAddress.ID NEQ #ID)
			IF(#BillingAddress.ID == #ID)

NNE
!=
	Not numerically equal
	(Not Numerically Equal)
		Valid if numerical parameters are not equal	IF(#BillingAddress.ID NNE #ID)
			IF(#BillingAddress.ID ‹› #ID)

NLT
‹
	Numerically less than
	(Numerically Less Than)
		Valid if numerical parameter 1 is smaller than parameter	IF(#Amount NLT #PC)
			IF(#Amount ‹ #PC)

NLE
‹=
	Numerically less than or equal
	(Numerically Less or Equal)
		Valid if numerical parameter 1 is smaller than or equal to parameter	IF(#Amount NLE #PC)
			IF(#Amount ‹= #PC)

NGT
›
	Numerically greater than
	(Numeric Greater Than)
		Valid if numerical parameter 1 is greater than parameter	IF(#Amount NGT #PC)
			IF(#Amount › #PC)

NGE
=›
	Numerically greater than or equal
	(Numerically Greater or Equal)
		Valid if numerical parameter 1 is greater than or equal to parameter	IF(#Amount NGE #PC)
			IF(#Amount =› #PC)

Table 11: Operators for combining conditional statements

Operator	Description	Description	Code example
NOT	Not	Valid if a value is not set	#IF(NOT #Gender)checked="checked" #ENDIF
OR	Or	Valid if an expression is valid	#IF(#Attribute.IsVisible OR #Value NE "")
AND	And	Valid if all expressions are valid	#IF(#Attribute.IsVisible AND #Value NE "")
DEFINED	Defined	Valid if a variable is present.	#IF (#DEFINED(#Name))

Note: The operator "NOT" has the highest priority, that is, it is used before all other operators.

Table 12: mathematical operators

Operator	Description	Description	Example
+	Plus (addition)		
-	Minus (subtraction)		
*	Multiplication		
/	Division		
%	Modulo	Remainder of integer division	5 % 3 = 2

10.7 Creating a TLE Function

If you would like to provide your own TLE functions, you need to know how cartridges are created and how hooks are registered. For more details, see *Cartridges*, *on page 81* and *Hooks*, *on page 109*.

In order to generate a TLE function, see *#FUNCTION*, *on page 68*, or *#BLOCK*, *on page 68* you need to not only implement the function but also register it in the TLE processor by registering it in a hook in the TLE processor.

In order for you to better understand, we will use the following example. The properties of an object called CD are to be displayed in a template. You can query attributes for this such as *CDID*, *Title* and *Price* directly from the database. A further property *Review* must be read out from an external source. For this, create the function *CDReview* for using in the following form in the template:

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

Start implementing the function, see *Code example 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;
}
…
```

Code example 44: Implementation of a TLE function

When doing this, please note the following:

- For the package name, you should observe the naming convention:

```
package <company name>::<cartridge name>::API::TLE::<merchant name>
```

- The function name in the PERL code and for the TLE function must correspond.
- The function must be registered in the TLE processor. To do this, use the function *register*. For an example of code, see *Code example 44*.

A hook for TLE functions is created in the TLE processor. New functions have to be registered in this hook. Registration is carried out in the *HooksTLE.XML* file in the cartridge and with the following 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>
```

Code example 45: registering the function in the hook in the TLE processor

When executing a hook, if the TLE processor determines that a corresponding function is registered, it checks whether this function is used in the template and registers it.

A special TLE function case is the *#BLOCK* statement, see also *#BLOCK*, *on page 68*. At this point, in addition to the function parameters for the HTML source code between *#BLOCK* and *#ENDBLOCK*, *$cTemplate* is applied and processed as an additional parameter:

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

Use the Diagnostics cartridge to read out all the current TLE functions.

10.8 Creating Dynamic TLE Variables

A dynamic TLE variable is comparable to the TLE function. In contrast to the TLE function, no parameters are passed here:

```
#<dynVarName>
```

Like the TLE function, a TLE variable needs to be registered in the hook in the TLE processor, in addition to being implemented.

We will extend the TLE function example by introducing two variables, one for displaying the CD of the day, CdOfTheDay*, and one for displaying the Top Ten, TopTen*. They can then be used in the template as follows:

```
#CdOfTheDay
#TopTen
```

Start again with the implementation, see *Code example 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;
}
…
```

Code example 46: implementing TLE variables

When doing this, please note the following:

- For the package name, you should observe the naming convention:

  ```
  package <company name>::<cartridge name>::API::TLE::<merchant name>
  ```

- The function for determining variable contents needs to be called *tle*.
- The function *existsTLE* must be implemented. This checks whether a variable for the name indicated in the template exists. If this is the case, the value is determined and displayed. Otherwise, the original expression in the template is used accordingly. For example, if *#FFEEBB* is in the template, a check is made of whether a dynamic variable has been specified for this name. If the result is negative, the entry is interpreted as a colour code.
- The function must be registered in the TLE processor. To do this, use the function *register*. For an example of code, see *Code example 46*.
- The function must be registered in the hook in the TLE processor. For more on this, see *Code example 45*, *on page 76*.

The *#TopTen* can be displayed using the *#LOOP* function.

10.9 Creating a TLE Formatter

The process for creating TLE formatters is the same as for TLE functions and dynamic TLE variables.

As an example, we would like to create a formatter for displaying prices in the *xx,- €* format, that is, without decimal places, for example. *19,- €*. This would be used in the template as follows:

```
#Price[nodec]
```

Start again with implementing the formatting function, see *Code example 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;
}
…
```

Code example 47: implementing a TLE formatter

When doing this, please note the following:

- For the package name, you must observe the naming convention:

  ```
  package <company name>::<cartridgename>::API::TLE::<merchant name>
  ```

- The function for formatting the variables needs to be called *Format*.
- The function must be registered in the TLE processor. To do this, use the function *register*. For an example of code, see *Code example 47*.
- The function must be registered in the hook in the TLE processor. For more on this, see *Code example 45*, *on page 76*.

Part II:

Cartridge Development

11. Cartridges

ePages 5 is based on cartridges. Cartridges are software modules which provide functions and design and are connected via dependencies and inheritance mechanisms. These modules communicate with each other and with other applications via the API.

For complex changes or function extensions, you should, as a developer, always create cartridges and integrate them into the system. This will make sure your system remains updateable.

Practical examples for the creation of cartridges can be found in the appendixes in.

11.1 Cartridge Structure

The default cartridges are found in the following ePages 5 installation directory

```
%EPAGES_CARTRIDGES%/DE_EPAGES
```

Every cartridge has a directory with the name of the cartridge in which all necessary files are saved. The following structure is used:

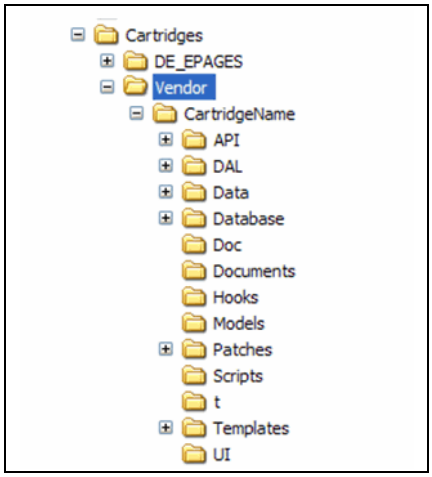

Figure 20: cartridge structure

You should create a separate directory for every project or company next to */DE_EPAGES* called the *Project or Vendor Directory*.

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

All the cartridge that belong to the particular project are created in that directory. A separate directory is created for each cartridge with the name of the cartridge:

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

It can contain the following subdirectories:

Table 13: important cartridge subdirectories

Subdirectory	Content
API	Contains PERL modules that offer public functions. Public means that the functions can also be used by modules of other cartridges. These functions should therefore be well-documented.

Subdirectory	Content
DAL	Database Abstraction Layer - functions that execute database functions When using the PowerDesigner from Sybase, the files are automatically generated. Provides a data cache, in which the results of repeated database queries are cached.
Data	Data for layout and control of the application, such as images, styles, templates, and configuration files for automatic processes.
Data/Private	Contains data that is only accessed by the application server, for example, import files or templates. During installation, these data are copied to the *%EPAGES_STORES%/Store* directory. Templates that overlay original templates are copied to the "overlay directory" *%EPAGES_STORES%/Store/Templates/DE_EPAGES/‹originalcartridgename›/Templates*. For more on this, see *Overlaying Templates*, *on page 35*. These templates must be placed in the directory: *Data/Private/Templates/DE_EPAGES//‹originalcartridgename›/*. Store is the business unit where the cartridge is installed
Data/Public	Data that the Web server accesses, for example, images and styles. These files are copied to the cartridge directory *%EPAGES_WEBROOT%/Store* during installation.
Data/Scheduler	All files for scheduled processes, such as automatic availability updates. This includes the files for executing tasks and the corresponding configuration files. The files are saved according to operating system and user-specific criteria. For more on this, see *Scheduler*, *on page 119*.
Data/WebRoot	Here you can enter additional Help files that explain the functions of the cartridge, in addition to the standard online Help. For more on this, see *Integrate your own online Help*, *on page 165*.
Database	
Database/Sybase	Files to generate the necessary tables and to define stored procedures.
Database/XML	Import files in XML format. See *Import Files*, *on page 113*.
Doc	API documentation of the PERL modules For further information on this, refer to the use of *Installing - nmake*, *on page 85* with the target sourcedoc.
Documents	Additional documents for developers. A description can be shown in each cartridge of the Diagnostics Cartridge. In the *Documents* directory, the file *index.html* must be created with the correct content.
Hooks	PERL modules that offer functionality that are called by hooks - see *Hooks*, *on page 109*.
Models	Database model of the cartridge which was created with PowerDesigner. The model is saved here as a .pdm.
Patches	Files that are used for patching. For more on this, see *Patching Cartridges*, *on page 161*.
Scripts	PERL scripts that are executed via the command prompt. These are functions that, for example, are started as jobs or serve as Help functions for Import/Export, deleting of database objects, and so on. For more on this, see *Scheduler*, *on page 119*.
t	Test cases for API functions There should be at least one test case with an API call and function test for every API function. In addition, test files, for example, images or .xml files should be saved here. For more on this, see. *Installing - nmake*, *on page 85* - Target test.
Templates	All templates that are necessary for the function of this cartridge and that do not overwrite any original templates.
UI	Modules that provide functions for interacting with the user (input, output); Usually private functions that are not publicly visible

11.2 Creating a Cartridge Structure

You can create cartridges manually or call the cartridge installer. This installer creates the necessary structure and the most important files. We recommend using the CreateCartridge script to avoid the errors which can occur when manually creating the structure and files.

You call the PERL script *CreateCartridge.pl* as follows:

```
perl %EPAGES_CARTRIDGES%/DE_EPAGES/Cartridge/Scripts/CreateCartridge.pl
<vendor>::<cartridgeName>
```

This installs the ‹vendor› directory in the following cartridge directory of the ePages installation:

```
%EPAGES_CARTRIDGES%
```

In the *‹vendor›* directory, the new cartridge directory *‹cartridgeName›* is created for the new cartridge. All necessary directories and files are generated in this subdirectory. You need to customize the content according to the requirements of your cartridge.

The *Makefile.pl* file is also created in the root directory of the new cartridge. You use this to then generate the *makefile* file. See*Installing - nmake, on page 85*.

You need to customize the structure that creates individual file types and their corresponding directories. When you do this, you also need to take the functional dependencies and the purpose of the cartridge into account. For more information, refer to *Cartridge Structure, on page 81* and pay special attention to the usage examples and the default cartridges.

11.3 Installer / Cartridge.pm

Another important file that is automatically generated in the */API* directory is *Cartridge.pm*. This file defines among other things which functions are to be executed when the cartridge is installed. The installer that controls if and how certain files can be copied or imported into the database is also indicated.

The installers are structured as hierarchies and inherit the functions. A list of installers is shown in *Table 14*. The installers are shown in their linear dependency. The basis is the *BaseInstaller*. An overview of the individual functions of each installer is available in the API documentation of the packages specified.

Table 14: installers

installers	Package / Description
BaseInstaller	*DE_EPAGES::Core::API::BaseInstaller* Fundamental and all necessary functions for installing/uninstalling patches and cartridges; Copy function, import function; sets the current version number of the impacted cartridge
FileInstaller	*DE_EPAGES::Core::API::FileInstaller* Copying and deletion of files from the subdirectory */Data* in the correct portions of the file system.
XML Installer	*DE_EPAGES::XML::API::XMLInstaller* Functions to read and edit XML files that contain information about cartridge dependencies and patch functions. *Dependencies.xml,* , *Patch.xml*
DatabaseInstaller	*DE_EPAGES::Database::API::DatabaseInstaller* Functions for creating and editing database directories and patches of the corresponding *sql* files

installers	Package / Description
TriggerInstaller	*DE_EPAGES::Trigger::API::TriggerInstaller* Functions for installing, deinstalling and patching files of the type *Hooks*.XML* with final cache reset
CartridgeInstaller	*DE_EPAGES::Cartridge::API::CartridgeInstaller* Functions, among other things for the registration of a cartridge in a store database, to update the cartridge status in the cartridge table, to determine dependent cartridges of for updating the database structure of this cartridge.
ObjectInstaller	*DE_EPAGES::Object::API::ObjectInstaller* Functions for installing, uninstalling, and patching objects, classes, and attributes; Managing files of type *Attributes*.xml*
PermissionInstaller	*DE_EPAGES::Permission::API::PermissionInstaller* Functions for installing, uninstalling, and actions and permissions Managing files of type *Actions*.xml* and *Permissions*.xml*
PresentationInstaller	*DE_EPAGES::Permission::API::PresentationInstaller* Functions for installing, uninstalling, and PageTypes and forms; Managing files of type *PageTypes*.xml* and *Forms*.xml*
ShopInstaller	*DE_EPAGES::Shop::API::ShopInstaller* Functions for installing, uninstalling features; Managing files of type *Features*.xml*
DesignInstaller	*DE_EPAGES::Design::API::DesignInstaller* Functions for installing and uninstalling design and navigation elements as well as e-mail forms and pre-defined search instructions; Managing files of type *NavBars*.xml*, *NavElements*.xml*, *Search*.xml*, *MailTypeTemplates*.xml*
ShippingInstaller	*DE_EPAGES::Shipping::API::ShippingInstaller* Functions for installing, uninstalling delivery methods; Managing files of type *Shipping*.xml*
PaymentInstaller	*DE_EPAGES::Payment::API::PaymentInstaller* Functions for installing, uninstalling, and payment methods and their respective logos and logo links; Managing files of type *PaymentLogos*.xml* abd *Paymenttypes*.xml*

When a cartridge is automatically created, the default installer is the design installer. See *Code example 48*.

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

use strict;
...
```

Code example 48: definition of the cartridge installer

The DesignInstaller makes sure that all the common files are copied to the corresponding locations or are imported into the database, in as far as they are set up in the cartridge.

Note: The *PresentationInstaller* must be used to install cartridges that must be installed on the site.

11.4 Installing - nmake

For installation and registering the cartridge in the system, *nmake* s used for Windows and *make* for Linux. Necessary steps like compiling, linking, copying files, and so on are performed automatically or via scripts. The prerequisite for this is the *makefile* file that must be generated for every cartridge and that contains the corresponding operating system-specific directions.

First generate the *makefile* for your cartridge. Open the console in your cartridge directory and enter the following command:

```
perl Makefile.pl
```

After the command is executed, the *makefile* file is created in the same directory.

Note: *Makefile.pl* generates an operating system-specific *makefile*.

In order to install a cartridge, the *makefile* with the target parameter *install* must be executed for a specific business unit, for example:

```
nmake install STORE=Store
```

‹Store› is the logical name for the database of the business unit. After finishing, the cartridge functions are known to the system and can be used in the business unit for which the cartridge was installed.

Additional actions can be executed via the *nmake* depending on the target indicated by the call. The general call for the *nmake* via the Windows command line therefore appears as follows:

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

When this is done, the following logical targets can be used:

Table 15: Targets for *nmake*

Target	Description
install	The cartridge or rather its functions are installed in the database and are thereby made known to the system. The necessary tables and stored procedures are also created. Furthermore, additional data are imported when necessary. The files are copied into the corresponding directories. If the new cartridge is dependent on other cartridges and these cartridges are not yet installed, they are then installed.
uninstall	All cartridge-relevant entries are removed from the database. Cartridges that are dependent on these entries are also uninstalled.
clean	Creates a "delivery ready" cartridge structure. The structure is cleaned, all the necessary files remain intact, and superfluous files are deleted. Files with the *.pdb extension are deleted. In addition, the makefile file and the directory */Generated* are removed.
register	This makes cartridge functions known to the central administration database (site) where they can also be displayed. After registering, the cartridge features are managed by the business administrator and assigned to individual shop types
unregister	The cartridge entry is removed from the site database.
test	All test cases from the */t* directory are executed. If an error occurs, an error message is displayed.

Target	Description
sourcedoc	Generate online documentation from the corresponding commented PERL modules of your API functions This creates a directory */Doc* in which the Help files are generated in HTML format. Help can only then be generated if the modules have not been encrypted. For more on this, see *Encryption*, *on page 89*.
generate	Generation of Perl code, SQL files for tables and stored procedures and creation of XML import files from the data model. The prerequisite is an existing database model in a **.pdm* file generated by the PowerDesigner. A */Generated* directory is created containing the subdirectories */API*, */DAL*, */Database*, and */t* with the corresponding files. These files can be copied from the /Generated directory into the corresponding cartridge directory and further edited.

11.5 Uninstalling

To uninstall a cartridge, open the console in your cartridge directory and enter the following command:

```
nmake uninstall STORE=Store
```

Store is the logical name for the database of the business unit in which the cartridge is installed. All database entries that contain the XML definition for the attribute *delete="1"* are deleted.

All files that were copied to the corresponding overlay directory are deleted. For more on this, see *Overlaying Templates*, *on page 35*.

If, after uninstalling, content or elements of the deleted cartridges are still shown, they may still be in the cache. Check and empty the following caches if necessary:

- Browser cache
- Optimisation in MBO
- *%EPAGES_STATIC%* directory

11.6 Copying Cartridge Directories

Changes in a cartridge require a long uninstall and a new install of the cartridge. If the changes have only been performed in the */Data* directory, copying the files the respective places is enough. There are two ways to do this. With this, you can copy the files from the */Data* directory for a specific cartridge into the corresponding business unit. The script is called as follows:

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

Options are:

- `-passwd` : Database user password
- `-storename` : Name of the database where the cartridge is installed
- `-type` : Type of cartridge data (Private, Public, WebRoot, Scheduler)

Flags:

- `-help` : Shows which options are available

Example:

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

11.7 Back Office Extensions

In addition to changes in the store front, function extensions in the back office are among those requirements requested most often. Here, the functions in various levels that are nested in each other are made available. Up to five levels can be defined, whose order and relationship to each other is also visible in the back office pages layout. See *Figure 21*.

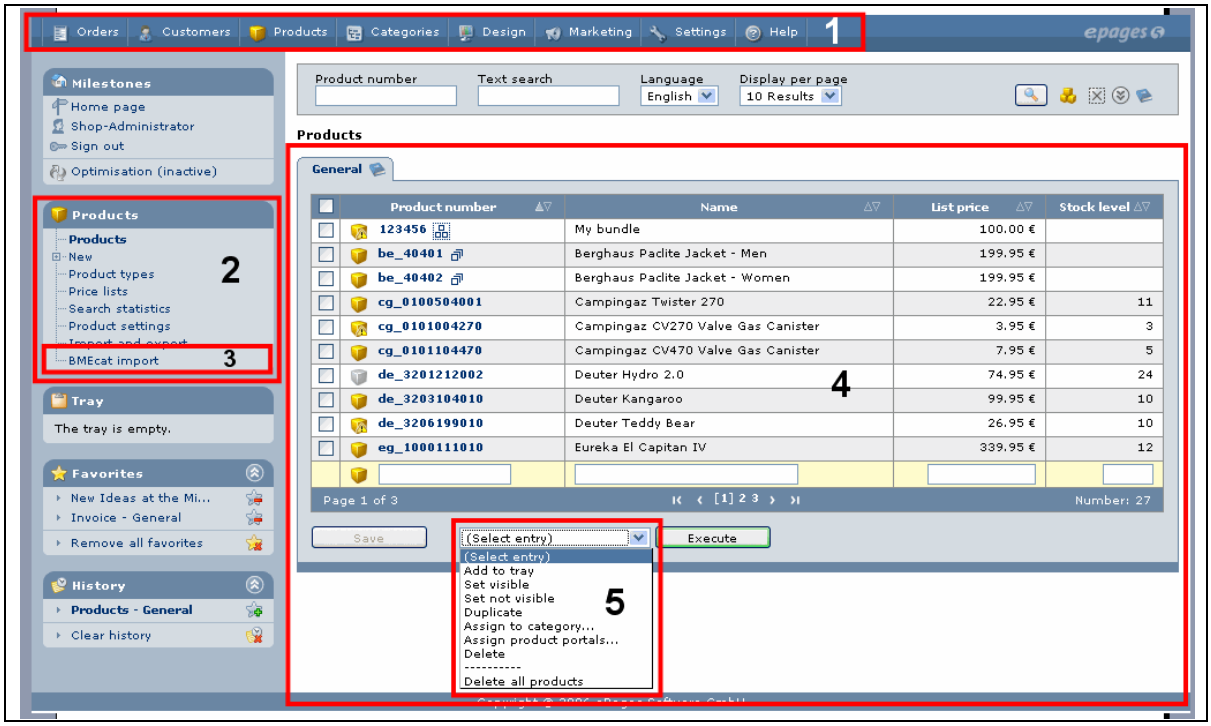

Figure 21: Function levels in the back office

The following areas are suitable for integrating new functions:

Table 16: Function areas in the back office

Area	Description
1 Main navigation points (Manager)	Main navigation points are separate functional areas with no reference to the other managers with their own encapsulated functionality. If you would like to add a new logical administration area to the application, create a new main navigation point. Example: Vendor management
2 Navigation Box in the Left Navigation Area	Navigation boxes are integrated into the left navigation areas of each module and make various blocks of functions available. There are two different kinds of navigation boxes: the first represents module-dependent functionality, such as the context menu, and general functions such as the Tray or Favourites. Example: Box for changing the language or to show the date and time
3 Menu Entry in a Navigation Box	Further functions in individual navigation boxes Example: Link to the home page or to the intranet in the Administrator menu
4 Tab	Functionality in content area, usually organized in tabs. Example: Tab for statistics in the Orders module

Area	Description
5 Batch processing commands	Extension of batch processing commands for a table Example: See below

A practical example for extending the back office functionality through an additional batch processing command is available in *UE 7: New Batch Processing Commands in the MBO*, *on page 195*.

12. Creating a Distribution

One important reason to encapsulate functional extensions in a cartridge is to protect the ability to upgrade the system. An additional reason is to provide functions that can be used by other systems. This means the cartridge need to be easily transferable and installable.

For this, you create a distribution. A distribution is a cartridge with all of the files necessary for its functions. These files only need to be installed on the target system.

You can create this type of distribution using *nmake* with the target *clean*. Once you have successfully tested your cartridge on your system and want to generate a distribution, enter the following command in the console in the main directory of your cartridge:

```
nmake clean
```

This cleans the cartridge structure by deleting all the unnecessary files.

The prepared cartridge can now be copied and installed on the target system.

If you would like to provide a range of functionality, but at the same time would like to protect the source code from unauthorized reuse or alteration, you can encrypt the source code files:

12.1 Encryption

You can use the encryption tool to encrypt the source code. The program is called *encrypt.exe* and is located in the following directory:

```
%EPAGES%/bin
```

Enter the following command to encrypt your files:

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

You can use the *-s* parameter to remove comments from the code during encryption. *Encrypt.exe* needs to be execute for each individual file. As a result, the encrypted file *‹perlmodule›.pm* is created. When calling the program without parameters, the possible parameters and their usage is shown.

Caution: *encrypt.exe* does not create a backup file of the file which is encrypted. We therefore strongly recommend that you back up all files before you begin encrypting.

Part III:

Additional Concepts

13. Creating Features

Features are functions that can be made available for the shop types available or put together into feature packs. Creating and activating of individual features is the task of the business administrator.

For this, these functions need to be defined and modified in the source code and in the database. Perform these steps to create such an offer:

1. Define the feature. In the corresponding cartridge in the */Database/XML* directory, create the *Features.xml* file or extend an existing one.
2. Integrate the feature check in the PERL code or in the template source code according to your requirements.
3. Execute *nmake install* for your cartridge. This imports the feature into the database.
4. Execute *nmake register* for your cartridge. This makes the feature known in the administration database of the business administrator. He can then select this feature for his shops

The following example demonstrates the procedure in more detail:

Each merchant can process a certain number of products in his shop. This number is a feature and can therefore be set differently for various shop types.

This feature is defined in the following file:

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

see *Figure 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>
```

Figure 22: Definition of the feature in the XML file

A *MaxValue* is indicated for every feature. If you set *MaxValue="1"* , this means that this feature can be activated (*1*) or deactivated (*0*). If a value greater than 1 is set, this feature is available as often as indicated. In this way, the number of products is also defined as a feature. This restricts how many products are allowed to be created in the shop. For example, *MaxValue="100000"* means that the user can create 100000 products.

In the display template for the merchant back office, a request is made to determine whether the feature is set or what its maximum value is:

```
<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
```

Figure 23: Feature checking in the template

Depending upon how the business administrator set the value for the feature, products may be able to be added. In the following, you can see how the business administrator can edit the feature values:

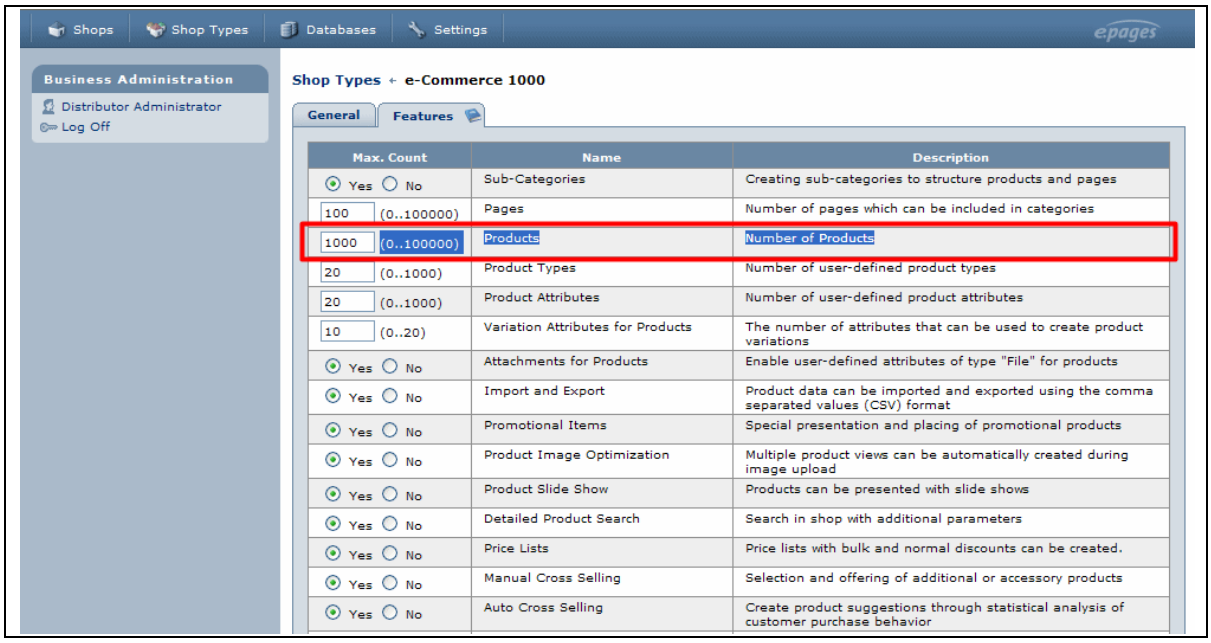

Figure 24: Feature activation by the business administrator

Here again, the difference between features with *MaxValue="1"* or greater *1* is shown. For equal to *1*, there is only the option *Yes* or *No*. For greater than *1*, the business administrator can set any value. The maximum is set in the XML file.

The query in a PERL module is as follows:

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

Code example 49: Query of features in the PERL source code

Testing in templates is used to activate or deactivate feature-functions such as links or buttons. The test in PERL allows corresponding error handling if the limits are exceeded.

14. Form Handling

The data that must be entered in templates must follow certain specifications about allowed characters and regional formats and can be limited by minimum and maximum values. Testing the entry values occurs during the *Save* command. Data with errors are not allowed to be saved.

Form handling simplifies and unifies display and editing of entry fields in templates. The data entry is structured and provided with a unified error management. Forms *are available to do this for the developer. In these forms, individual entry fields or field groups are assigned. This assignment controls the type and limits of entries as well as the further processing of the content. These fields are called FormFields*.

The form handling offers the ability to centrally define the processing and error handling for form fields and to perform these the same for every form.

The following design guidelines are applied here and should be kept:

- Required fields are marked with an asterisk *
- If entry mistakes are made, the complete form is shown again with the entered values shown. These values are only saved if all entry values are correct.
- If the form contains multiple entry errors, all rows with entry errors are shown highlighted, so that they can all be corrected at once.

Additional possibilities are:

- The focus can be placed on the first error field.
- In the error fields, an example of a correct entry can be shown.

Display, testing, and processing of the entry values are related to the assigned type, mandatory field definition and value range. You have two ways to define these parameters for attributes: in the *Attributes*.xml* and in *Forms*.xml*.

14.1 Error Handling for Object Attributes

Single-dimensional object attributes are simple attributes, for example weight or tax class. They are defined in the *Attributes*.xml* file. These attributes are verified automatically during template processing based upon their type definition.

The verification occurs through the execution of the validation function *attributeValues* in "Standard-Save" *SUPER::Save($Servlet)* from *DE_EPAGES::Presentation::UI::Object* .

14.2 Error Handling for Freely-Definable Forms

You can also use multi-dimensional attributes in forms. The treatment of multi-dimensional attributes, such as prices or sub-products, must be defined separately. In this case, the type of attribute is already available, but since a variable quantity of values is shown, this variable display and handling must be defined. The same applies for parameters that do not belong directly to this object.

For these cases, the file *Forms*.xml* is available that you can create for each cartridge if necessary.

You can use the following types in the FormField definition:

Table 17: Data type definitions in forms

Type	Description	MinValue/MaxValue Determination
string	Text field	Maximum length of the text. If this field is marked as a mandatory field, then the minimum length is 1.
integer	whole number, unformatted	Minimum and/or maximum value
float	floating-point number, unformatted	Minimum and/or maximum value
reg_date	Date in the regional format	not usable
reg_time	Time in regional format	not usable
reg_datetime	Date and time in regional format	not usable
reg_money	Currency in regional format with currency displayed	Minimum and/or maximum value
reg_integer	Whole number (regional format)	Minimum and/or maximum value
reg_float	floating-point number in regional format	Minimum and/or maximum value
checkbox	logical value (true or false)	not usable
file	Loaded file	Minimum or maximum size in bytes
email_address	Text in e-mail address format	not used
ip_address	Only possible in IPv4 format	not used

For each of these types, a corresponding entry error handling is implemented.

You must define and implement the corresponding actions for saving the form for the template in which you use FormFields. The Action-Handler must be passed the correct forms in the function.

During execution, the FormFields are automatically subjected to type-specific error tests and in case one or multiple entry errors, the corresponding error-TLEs are filled. See also *Error TLE*, *on page 69*. The template is shown again. Through evaluation of error-TLE contents in the template, the corresponding error messages are shown.

If you want to use form handling in your own templates, you must do the following:

1. Identify the FormFields in the template.
2. Define the FormFields in the Forms*.xml of the cartridge.
3. Implement the handling of field values during saving of the form. Possible definition of your own error handling routines for values that do not match standard types.
4. Evaluation of the error TLEs in the template

An example of form handling is implemented in training in the *Polling Cartridge*.

14.2.1 Definition of FormFields

During creation of the templates, you must first determine which error handling mechanism would be appropriate for the individual fields. Was previously explained, the entry values for single-dimension object attributes are tested by default.

The assignment remains for you to take care of the multi-dimensional attribute values that do not belong to the object. Create a *Forms*.xml* file for these. The naming conventions for XML import files applies to the file names, see *Import Files*, *on page 113*. A simple FormField definition can be seen in *Code example 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>
```

Code example 50: Example of a FormField statement

In the example, you can see that forms are class-based. In addition, each form receives a unique name. It is also possible to define different forms for the data entry of a class.

Each field within a form must have a unique name. A type is assigned for each field. The values *0* and *1* are possible for the mandatory field definition *Mandatory*. Depending upon the type, see *Table 17*, a area limit can be entered.

The definition of forms for multi-dimensional attributes is somewhat more complex. The reason for this is that during installation of the template, it is not clear how many values of a specific attribute must be entered. An example of this is the product price. Depending upon the entered currencies, that number of entry fields for the price are shown. A loop definition is used in such cases in form. See *Code example 51* or also *Code example 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>
```

Code example 51: Form for multi-dimensional fields

All impacted fields, in the example *CurrencyID* and *Price* are defined in a *FormLoop*. This is the prerequisite that the error handling for these fields are performed for as many entry fields are shown in the template. The name of the *FormLoop* within a form must be unique. The number of FormFields in the FormLoop must be equal to the quantity of entry fields in a template for multiple values can be entered.

The forms are registered in the database during cartridge installation. The forms are imported manually into the database with the following command:

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

For the basic principles about file importing, see *Import / Export of database contents, on page 113*.

14.2.2 Using Forms in Perl Code

You have to make sure that forms in PERL code are processed correctly. The method *$Srvlet->form->form* is available for this. With this method, the FormFields will be validated based upon the regional settings according to type and area limits. Errors that occur are saved in an error list that can be accessed by the error TLE.

The syntax for the method in PERL code is:

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

The ‹object› must be an object of the class for which the form was registered. See *Code example 50* and *Code example 51*.

The following example shows the usage:

```
...
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;
}
...
```

Code example 52: Validating forms in PERL

14.2.3 Validation of Undefined Data Types

You can also define fields in forms that do not match the standard types. For such FormFields, you must do the validation in the PERL code itself. The same applies to fields which for example may be validated extensively according to the validation the type definition.

You can see an extended validation in *Code example 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);
}
...
```

Code example 53: Extended validation for forms

First, the form is read. Before execution of the specific validation, the general error test for the form is performed. The last parameter transfer, *1* determines that in spite of possibly occurring errors, the processing will not be canceled.

In the further processing, an additional test is performed for the field *Secret* of type *string*. A verification is performed to determine whether the characters used are in a certain area. In case of an error, an entry with an explanation of the error is entered into the error list.

Afterwards, the general error handling occurs.

14.2.4 Error Handling Templates

All errors that are determined during validation are entered into an error list, the *FormErrorLoop*. This *FormErrorLoop* is a prerequisite for analyzing and displaying the errors in the template using the error TLE. The error TLE's that are available for this are listed in *Error TLE, on page 69*.

Error handling can be divided into three cases:

- Error display for individual fields
- Error Display in Lists
- Display of all errors of a template in a list

14.2.4.1 Error display for individual fields

Here a query is made for individual fields whether an error has occurred. Because of this, each field can have, if necessary, a specific error handling in the template.

```
#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
```

Code example 54: Error handling in the template for individual fields

In the example, a general query about errors occurs first. If so, a warning message is shown.

Afterwards, *#FormError_‹fieldName›* tests the fields *SMTPServer* and *SMTPPort* to determine whether errors have been entered. If so, the view for these fields is changes.

14.2.4.2 Error Display in Lists

The unique thing about lists is that multiple values for an entry field can be entered. To be able to validate these fields, *Loops* are defined in forms. See *Code example 51*. For the display in the template, analogue structures are necessary. Errors must be validated and displayed with the Loop so that every error can be shown in the list.

```
#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
```

Code example 55: Error handling in the template for lists

In the example, multiple price entry fields are activated for a product and shown depending upon the currencies that are activated. If entry errors are determined for one or multiple prices, the corresponding fields are highlighted in the re-displayed template.

14.2.4.3 Display of all errors of a template in a list

Another possibility of error display is listing of all errors in a separate list. In this case, no entry fields are highlighted, but an overview of the errors is shown with a descriptions. A usage example is the display of errors in the data export.

For this error display, the error TLE *#FormErrors.‹ErrorTLEName›* is used:

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

Code example 56: Display of error list

The error information that can be read and displayed is available in the API documentation for the module *Form.pm at Presentation/API*.

15. Web Services

Web services offer the ability to exchange data between platforms. Protocols such as HTTP, SMTP, or FTP are used to do so. The data are structured according to the SOAP standard and are transferred to an XML document.

ePages 5 offers a framework, in addition to its own Web services, that supports the creation and integration of Web services.

This is described in the following section. The prerequisites are knowledge in Web services, XML, SOAP lite, WSDL, and PERL programming.

ePages offers a specific training for this topic. The training supports this chapter with examples, additional information, and explanations. In addition, external tools to validate Web services and clients are also introduced.

15.1 ePages Web Services and Framework

ePages uses Web services internally to transfer data between the central administration database and the shop database.

Data exchange with external systems is available in the following Web services:

Table 18: Web services in ePages

Data Area	Web service
Product data and catalog structures	ProductService CatalogService AssignmentService
Price Lists	PriceListService PriceListAssignmentService
Customer Data	CustomerService
User Data	UserService
Order Data	OrderService
Billing and packing slip data	OrderDocumentService
Creating shops/shop management	ShopConfigService

Use the Diagnostics Cartridge to view all Web services registered in ePages 5.

The basis for implementing Web services is the *WebService* cartridge. It provides a basis Web service and binds with the module *SOAP::Lite* as well. A basic permission check is implemented there as well.

The description for each of the Web services named in *Table 18* is contained in the respective WDSL file. These WSDL files are found in these directories:

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

or

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

Complex data types that are used in a Web service are defined in the corresponding XSD file.

The Web service requests are handled in the system as follows:

- Client request is received
- Verification whether the Web service is registered in the database
- Verify whether the required permissions to execute the function exist.
- Transfer of the request to the SOAP::Lite module which converts the provided XML document to the corresponding PERL objects
- Execution of the corresponding functions in PERL modules and return the results
- Conversion of the result in an XML document through the SOAP::Lite module
- Transfer the response with the XML document to the client

Note: The prerequisite for executing Web services is activating the feature *Program Interface for Web Services* in the business back office. The description about how to do this is available in the *Business Administration Guide*.

A logging feature has been integrated into the Web service cartridge in order to shop possible SOAP errors. This logging can be activated in this file:

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

Remove the comment character for the *SOAP server* line:

```
;
; 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
```

You can find more about the *log4perl.conf* file in the *Installation Guide for Windows*.

15.2 Generate ePages Web Service

Create a new Web service in your own cartridge. The following elements exist in the cartridge structure:

- The subdirectory */API/WebService* - PERL modules for the Web service are contained here.
- The subdirectory /Data/Public/WSDL – WSDL files for Web services are created here. These descriptor files are optional. They are transferred to the *%EPAGES-WEBROOT%/‹store›/WSDL* directory during installation.
- The files *Actions*.xml* and *Permissions*.xml* must be extended with the Web service-specific entries. In this way, the Web services are registered in the database and the corresponding permissions are granted.

After creating the cartridge, you must perform the following steps, to create a Web service:

1. Register
2. Authorize
3. Implement

15.2.1 Register

Every Web service must be registered in the database. This is done using the *Actions*.xml* file of the cartridge. Create this file or extend the one that is already there. During installation, the file will be imported into the database and the Web services and their methods are registered in the corresponding tables.

In *Code example 57* you can see the structure of an entry for registering a Web service.

```
<?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"
          ="Training::MyWebService::API::WebService::MathService" />
    </Object>
    </WebService>
    </Object>
</epages>
```

Code example 57: Registering a Web service

The Web service is registered in the Object folder *WebServices*. An Alias and a URI must be provided for the Web service element itself in the element *WebService* . The URI (Unique Resource Identifier) string is a similar to a URL and is used to identify the Web service. It has also become standard to enter the creation period (2005/05).

Within *‹Webservices›*, you must define the methods They must be assigned to the *Methods* object.

In the *WebServiceMethod* tag, a method is defined. The name of the method is provided under *Alias*. *Package* shows where the method is implemented in PERL. Each method must be entered individually.

15.2.2 Authorization

You must enter who is allowed to execute each Web service. That means, for each Web service a role must be defined which has the right to call the methods.

These permissions are set in the *Permissions*.xml* file. In *Code example 58*, you can see one such entry.

```
<?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>
```

Code example 58: Setting the permission

First set to which class the permission should be assigned. Enter the role which has permission to execute the method in the *Role* element. In *WSRoleMethod*, the method is provided along with which Web service this method belongs to.

The WebService role is automatically assigned to every administrator of a shop. That means, that the role assignment in *Code example 58* allows every shop administrator to call the method *Add* of the Web service *MathService* .

If you would like to offer a function publicly, meaning everyone can call the function via Web service, you should:

- use basis for the Web service *BaseService*,
- pass the shop as the parameter
- create a new role, for example *PublicWebService*,
- and assign this role to the group *Everyone*.

15.2.3 Implementation

The functions themselves are implemented in the PERL modules of the Cartridge directory */API/WebService*.

The foundation for the Web service implementation is the base class *BaseService* from the package *DE_EPAGES::WebService::API::WebService::BaseService*. This class provides, among others, the following:

- a permission check
- the user object of the current Web service user
- data mapping between the entered XSD file and the entered URI

Depending upon whether the Web service is available for the shop or for provider actions, there are two additional subclasses of BaseService that you can use as the basis for your Web services:

- *DE_EPAGES::Shop::API::WebService::BaseShopService*: This class uses the shop object and authorized the respective merchant login. This class is primarily used for implementation of storefront and back office functionality.
- *DE_EPAGES::ShopConfiguration::API::WebService::BaseProviderService*: These classes use the distributor object and authorize the distributor login.

Code example 59 shows the integration of one of the base classes:

```
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;
```

Code example 59: Web service PERL module

15.3 External Clients for ePages 15.3 Web Services

To call the ePages 5 Web service, the necessary external systems must be offered to the corresponding clients.

For certain programming languages, such as Java and C, the code can be generated automatically based upon the WSDL files. For PERL, you must create the clients manually.

You must create a SOAP object in the client module that you pass the URI (the name of the Web service) and the goal point (proxy) the address of the SOAP service to.

Additionally, ePages standard Web service requests expect authorization via user name and password. The general form for calling the proxy is:

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

The unique thing relating to ePages is that the login data must be transferred in the form of an object path, for example:

```
/Shops/DemoShop/Users/admin
```

In this form, the login cannot be passed as a proxy call, because slashes (/) are not allowed. To get around this problem, login information are read from the credentials. *Code example 60* shows a possible variation:

```
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";
```

Code example 60: External client

The necessary login data are transferred to the *PROXY_URL* using the function *userinfo* and the parameter *CREDENTIALS*. The function as_string generates the correct proxy call for the example in the form of:

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

After the SOAP object was created, a Web services method listed in *NAMESPACE* can be executed.

From the example, you can see that for the authorization a login and password of a shop administrator is used. This means that, as soon as the shop administrator changes his sign-in information, the Web service calls must be changed as well.

It is a good idea to create an administrator for Web services in the user management of a shop whose data is only changed in consultation with the person responsible for Web services.

If you use the MS IIS Web server, note that for effective passing of user name and password in a URL, it is necessary to turn off integrated Windows authentication for the virtual epages directory. Otherwise, the user name and password cannot be transferred to the service.

These settings can be changed in the MS IIS properties. See *Figure 25*.

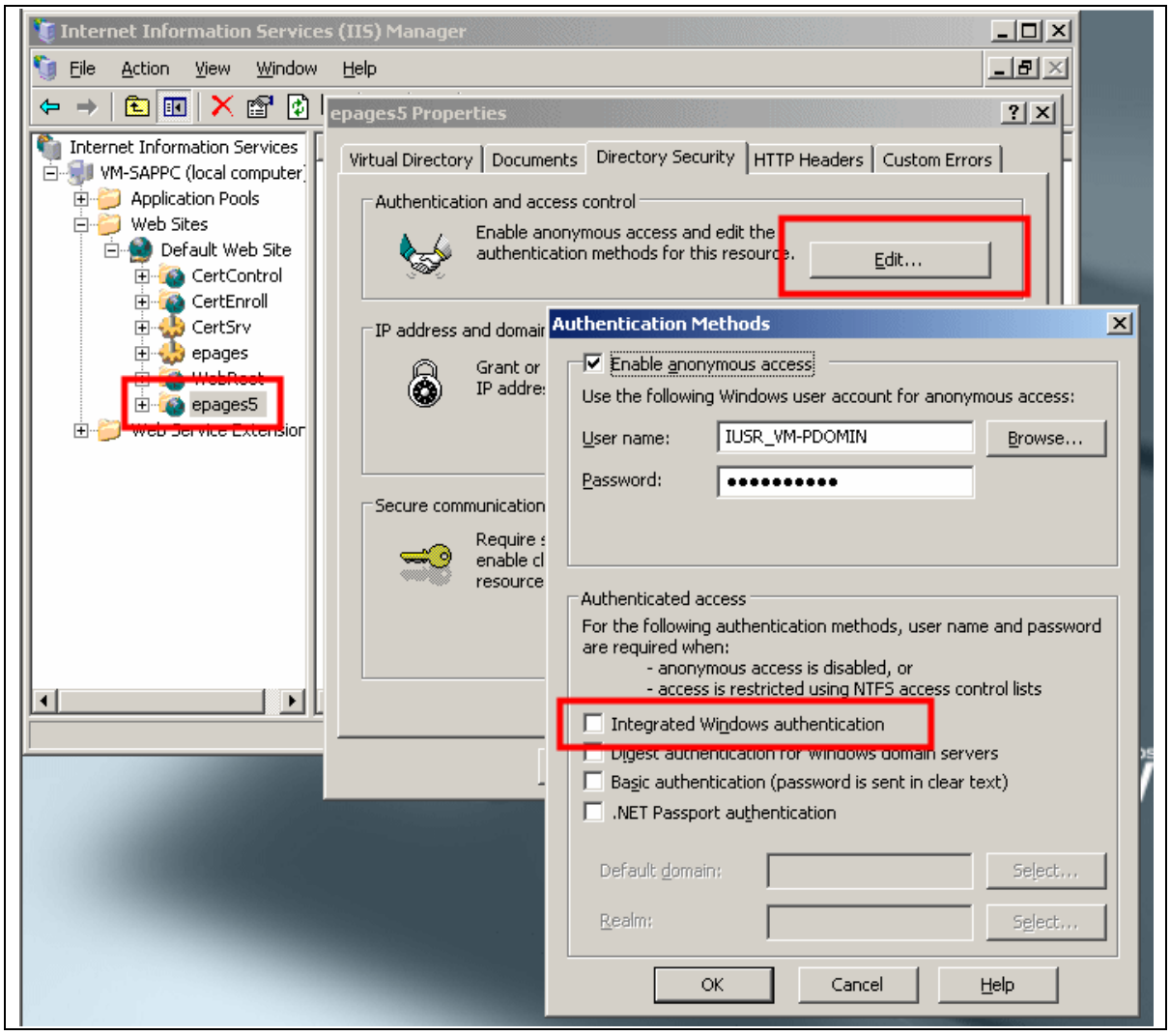

Figure 25: Deactivating Windows authentication

In case the Web services is written by an WSDL file, you can open the functions by creating a service object and passing the URL of the Web service to it. See *Code example 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;
```

Code example 61: External client with WSDL URL

Then you can call the desired method using the service object.

15.4 Implementing an ePages Web Service Client

External Web services can be called from the storefront and back office templates. To do so, you must create your own TLE function. Use this TLE function to call the Web service. The results are shown in the return value of the function and can be shown in the template. Read *Creating a TLE Function, on page 75*.

One possibility of Web service integration in a template is shown in *Code example 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
…
```

Code example 62: Calling a Web service from a template

In the example, the external Web service is called through the function *WS_STOCKQUOTE*. A parameter required by the Web service is passed. The Web service answer is passed to the variable *StockQuote* that is then shown in the template.

The function to call the Web service is shown in *Code example 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;
}
…
```

Code example 63: Calling an external Web service

16. Hooks

Hooks are defined areas in functions, from which other external functions are called. They let you extend the function of a module without modifying the module. These module extensions must be registered for the corresponding hook.

During the process, every hook is checked for whether an additional function has been registered for this hook. After the function has been executed, the "normal" process is resumed.

The basic function for setting up hooks is the *TriggerHook()* function in the *Trigger* cartridge.

The code generator automatically generates standardized trigger points. Every ePages class provides hooks whenever objects or table entries are generated, updated and deleted.

Table 19: Default hooks

Object hook	Table hook	Call
OBJ_Insert$Class	API_Insert$Table	After a new object of type $Class or a new table entry has been inserted in $Table.
OBJ_Delete$Class	API_Delete$Table	Before an object of type $Class or a table entry is deleted from $Table.
OBJ_BeforeUpdate$Class	API_BeforeUpdate$Table	Before attributes for an object of type $Class are modified or before a table entry in $Table is modified.
OBJ_AfterUpdate$Class	API_AfterUpdate$Table	After attributes for an object of type $Class have been modified or before a table entry in $Table has been modified

Hook names are structured as follows:

1. Prefix *OBJ_* or *API_* indicates whether the function is used on an object or a table.
2. Name of the action, for example, *AfterUpdate*
3. Name of the class or table

You can view an overview of all the available hooks using the Diagnostics cartridge under **All Hooks**.

16.1 Providing a Hook

In order to provide a hook in a cartridge, the corresponding function call must be implemented in the code and registered in the database. For this, the following steps are necessary:

1. Insert the *TriggerHook* function in the source code:

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

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

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

1;
```

Code example 64: Inserting the trigger function for a hook

We recommend naming the hook according to the structure explained above.

A hash is passed as a parameter. This hash contains the parameters used in the function including hook details such as *HookCount* or *HookName* and a reference to the object or the PrimaryKey in the table.

2. Enter the hook function in the corresponding *Hooks*.xml* file in the cartridge. For more on this, see *Hooks*, *on page 109*.

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

Code example 65: Registering the hook in the XML file

3. To register the function in the database, import the *Hooks*.XML* file using the *Trigger-Import script*:

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

Use the Diagnostics cartridge to make sure that the hook has been correctly registered in the database.

16.2 Function Extensions Using Hooks

There are three steps required for using existing hooks for function extensions:

1. Implement the function extension in an external PERL module. The following naming convention is used to define the function:

```

```

Code example 66: Naming convention for hook procedures

The following example shows how to implement this naming convention:

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

use strict;

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

1;
```

Code example 67: External function extension

2. Enter the function-hook reference in the *Hooks*.xml* file. For more on this, see *Hooks*, *on page 109*:

```
<?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>
```

Code example 68: A function-hook reference

Then use the *Hooks*.xml* file in the cartridge in which you want to implement the function extension.

The *OrderNo* is of type *Integer* and can be assigned optionally.

Note: As soon as more than one function has been assigned to a hook, these functions are executed according to the order of the OrderNo.

3. Import the *Hooks*.xml* file using the *Trigger-Import script* in order to register the reference in the database:

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

Use the Diagnostics cartridge to make sure the function has been correctly assigned to the hook. To do this, click the hook name in the list of hooks to display the assigned functions.

17. Import / Export of database contents

A large part of the data is provided in an XML file and must be read into the database. These are, for example, descriptions of class structures, function and attribute definitions, or content. You have already become familiar with examples of the individual import files in the practical examples.

The imports are either executed when the cartridge is installed or can be started from the command prompt. Depending on the type of data to be imported, different import handlers need to be used. For information on how to execute manual imports, refer to *XML Import*, *on page 114*.

17.1 Import Files

XML import files that are used in cartridge development and automatically read into the database during installation are explained in the following.

These XML files are subject to a naming convention. The file names must begin with the Type name used but can be extended as you like after that. The file must end with *.xml*. You can create multiple files per file type that are differentiated only by the optional extension.

An example of the names you can use for the XML file for PageTypes is as follows: *PageTypesMBO.xml* or *PageTypesSF.xml*. For *Attributes.xml*, you could use *AttributesShop.xml*.

Examples for using these files can be found in examples and in the default cartridges of your ePages installation.

The following table gives an overview of the XML files that you can use in your cartridges for importing. The optional name extension is represented by * (wildcard).

Table 20: Import files in cartridges

Import file	Description
Actions*.xml	Definition of ViewActions and ChangeActions, see *URL Actions*, *on page 29*. The functions called by these actions are located in the modules in the */UI* cartridge directory. Every ViewAction can be linked with an online Help topic, see *Integrate your own online Help*, *on page 165*.
Attributes*.xml	Import of object attributes This file is also automatically generated when using the PowerDesigner and contains all the attributes that were assigned using the PowerDesigner. Additional attributes need to be defined manually.
DefaultShop*.xml	Definition of values set for specific objects and attributes when a new shop is created; this is not automatically imported when a cartridge is installed. The import must be implemented accordingly, for example, using *Cartridge.pm*.
Dependencies.xml	Assignment of dependencies to other cartridges This file defines which cartridges need to be present and installed to guarantee cartridge function. If one of the cartridges listed in the file was not installed during the installation process of your cartridge, the system automatically installs the missing cartridge. Use the Diagnostics cartridge to determine the dependencies between cartridges. For more on this, see *Diagnostics Cartridge*, *on page 125*.
Features*.xml	Definition of cartridge functions as features. A feature is a function in the shop that the business administrator can activate optionally and can, therefore, be used as a basis for defining various shop types.

Import file	Description
Forms*.xml	Definition of the entry fields used in forms including information on the field restrictions and validation; For more on this, see *Form Handling*, *on page 95*.
Hooks*.xml	Hook definition and registration, see *Hooks*, *on page 109*. If you create a database model with the PowerDesigner, corresponding hooks are automatically generated for the tables and also stored as an XML import file. Additional hooks need to be defined manually.
MailType*.xml	Definition of the MailTypes for e-mail events; they are listed in the e-mail settings on the merchants administration page. For more on this, see *Adding E-Mail Events*, *on page 149*
NavElements*.xml	Definition of the navigation elements available to the merchants.
PageTypes*.xml	Definition of logical areas for the Web pages, and template assignment. For more on this, see *PageType Concept*, *on page 39*.
Permissions*.xml	Definition of which actions can be executed by which user. Every action can be assigned a certain role, see *Rights and Roles*, *on page 21*
PortalSites*.xml	Definition for which countries the portal is available and pre-set of specific data such as LocaleID, Registration URL, or tax model
Search*.xml	Definition of database search queries including possible parameters and sorting keys
System.xml	Attribute assignment in the system when a cartridge is installed, for example, default settings for the Web server settings or information about the API version used in the eBay cartridge; this is not automatically imported when a cartridge is installed. The import must be implemented accordingly, for example, using *Cartridge.pm*.
TemplateTypes*.xml	Definition of alternative display options for products and categories An example of this is the display options for the individual product types on the merchant's administration page.

17.2 XML Import

In order to manually execute XML imports, you can use various development tools in the form of import scripts.

These command line-based tools validate the XML files when they are read in. Error messages are displayed on screen and in the *%EPAGES_LOG%error.log* log file. You do not have to restart the ePages service after finishing the import. The application server cache is automatically reset. If you read your data into the database using a different method, for example, via direct SQL statements, no automatic cache refresh occurs.

The import script is called *import.pl* and is located in the following directory:

```
%EPAGES_CARTRIDGES%/DE_EPAGES/Object/Scripts
```

It is called in the console as follows:

```
perl import.pl [options] [flags] <dateiname>.xml
```

The individual options and parameters and how they are used are displayed by simply entering the following

```
perl import.pl [-help]
```

For example, if you want to import an XML file for PageType definition, enter the following:

```
perl import.pl –storename Store <cartridgepath>/PageTypes.xml
```

Store indicates the database into which the file will be imported. Since no object path is explicitly indicated, */*(*System*) is used by default.

On the other hand, if you would like to import payment methods, for example, especially for the demoshop, you need to indicate the object path to the demoshop for the import. That is why the import command appears as follows:

```
perl import.pl -storename Store -path "/Shops/DemoShop"
<cartridgepfad>/PaymentMethods.xml
```

Again, the object path is implicitly extended with the prefix *System* resulting in the complete object path *System/Shops/DemoShop*.

You can determine the object path to the respective target using the Diagnostics cartridge. To learn more about working with the Diagnostics cartridge, refer to *Diagnostics Cartridge*, *on page 125*. You will find an example of this in chapter *XML Export*, *on page 116*.

The default installation also includes the demoshop. All the data for this shop are prepared and read into the import files *DemoShop.xml* in the following directory

```
%EPAGES_CARTRIDGES%/DE_EPAGES/DemoShop/Database/XML
```

This file contains examples of the import formats for all the applicable objects in a shop.

17.2.1 Special Case : standards.pl

Some data cannot be imported as objects into the ePages class structure. This includes currencies as well as language and country codes in the XML files:

- *%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*

These data are read in with a special import command. To do this, enter the following into the console:

```
perl standards.pl -storename <Store> <dateiname>.xml
```

17.2.2 Special Case: Hooks

To import hooks, a special import script needs to be used. This is located in the following directory:

```
%EPAGES-CARTRIDGES%/DE_EPAGES/Trigger/Scripts
```

To import files of type *Hooks*.xml*, the following call is necessary:

```
perl %EPAGES-CARTRIDGES%/DE_EPAGES/Trigger/Scripts/import.pl
<dateipfad>/Hooks*.xml
```

17.2.3 Special Case: Forms

To import forms, still another special import script needs to be used. This is located in the following directory:

```
%EPAGES-CARTRIDGES%/DE_EPAGES/Presentation/Scripts
```

To import files of type *Forms*.xml*, the following call is necessary:

```
perl %EPAGES_CARTRIDGES%/DE_EPAGES/Presentation/Scripts/import.pl
-storename <store> <dateipfad>/Forms*.xml
```

17.3 XML Export

In addition to importing, you can also export data in XML format. The syntax is similar to the import function and the script is located in the same directory. Enter the following command:

```
perl %EPAGES_CARTRIDGES%/DE_EPAGES/Object/Scripts/export.pl [-help]
```

Various options will be displayed. The general command line is as follows:

```
perl export.pl [options] [flags] <objects>
```

At this point, it is necessary to indicate the objects to be exported. Use the export function to generate correctly structured XML files as templates for importing.

The following example demonstrates export and import. Let us assume you would like to set up an additional delivery method for the demoshop. You do not want to create this method manually. It needs be importable.

To receive a template for the import file, export the demoshop delivery methods. Since you want to export the objects for a particular shop, you need to indicate the object path to this shop. Use the Diagnostics cartridge to read the object path.

Open the Diagnostics cartridge and call All Shops **from the main page:**

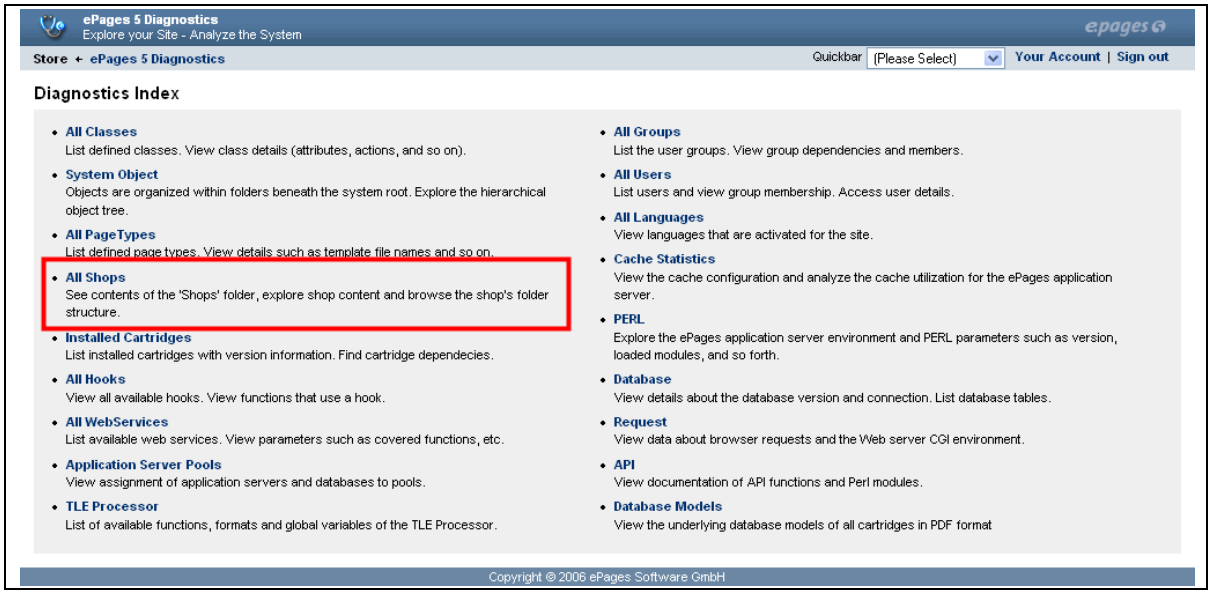

Figure 26: Call for the *Shops* object folder

On the page for the *Shops* folder, all the existing shops are listed are under *Child Objects* including the *Demoshop*. See *Figure 27*.

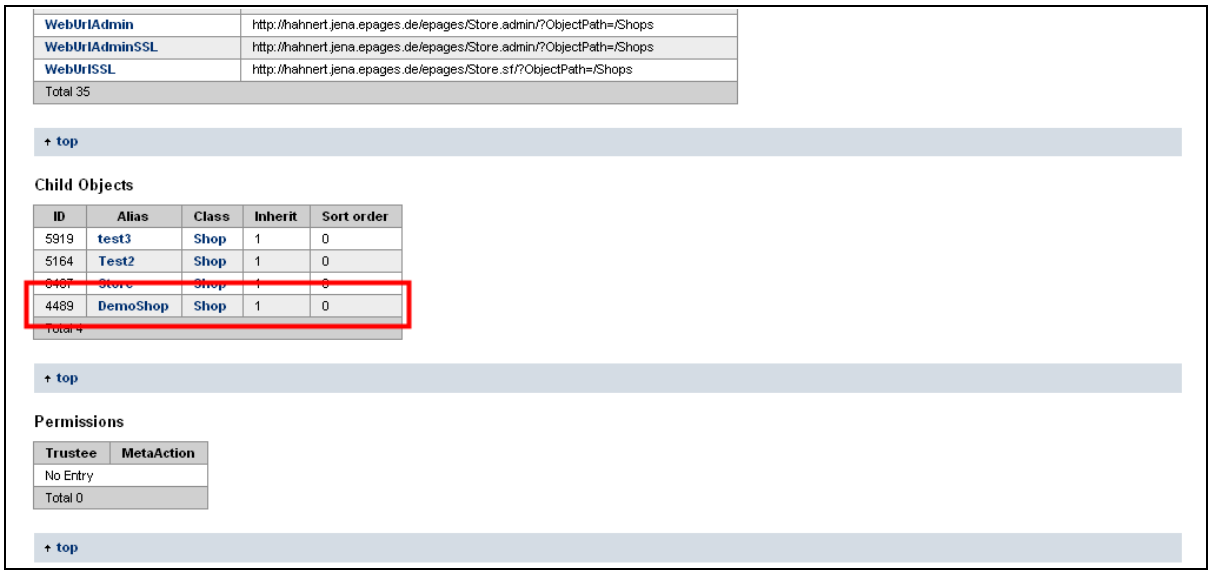

Figure 27: Listing of all shops

Now click **DemoShop** to display the parameters for the demoshop:

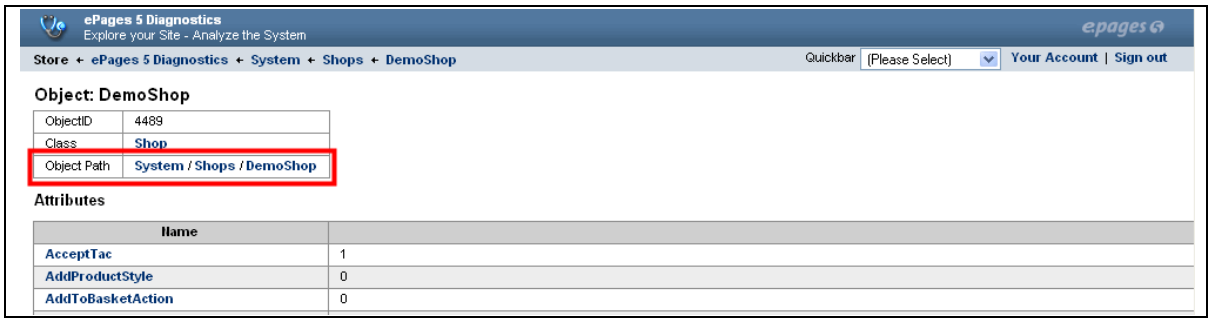

Figure 28: Parameters for the demoshop

In the top part of the page, you can see the following object path: **System / Shops / DemoShop**

Now enter the following command to export the delivery methods:

```
export.pl -storename Store -path "/Shops/DemoShop" -file ShippingMethods.xml
ShippingMethods
```

When entering the path information, you need to leave out *System*. The path is automatically extended with the root object *System*. The statement says the following: export all the objects in the *ShippingMethods* folder out of the demoshop database *Store* into the *ShippingMethods.xml* file.

This file is the basis for importing a new delivery method. Open the file and become familiar with the XML structure.

When doing this, please note the following:

- The GUID (global unique identifier) for the object is indicated in the export file. In order to create new objects, you must delete this GUID. Otherwise, the existing objects will be updated with the new data. You can also use the option *–withoutguids*. Then, the GUID will not be exported.
- At the same time, pay attention to the alias. Do not change the alias. An existing object will be updated with this alias.
- Use a new alias to create a new object.

The new delivery method should be a method indicates how much the total should be to receive free shipping and, for example, UPS delivery. Look for a fitting method in the file and overwrite the parameters. After that, the file content should appear as follows:

```
<?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>
```

Code example 69: XML structure for importing a delivery method

Save the changes. Finally, the edited XML file is imported again. For this, enter the following command in the console:

```
import.pl -storename Store -path "/Shops/DemoShop" ShippingMethods.xml
```

Again at this point, do not indicate the *System*. The data from the *ShippingMethods.XML* file are imported into the *Demoshop* in the *Store* database.

In the MBO, you can check whether the new delivery method has been correctly imported:

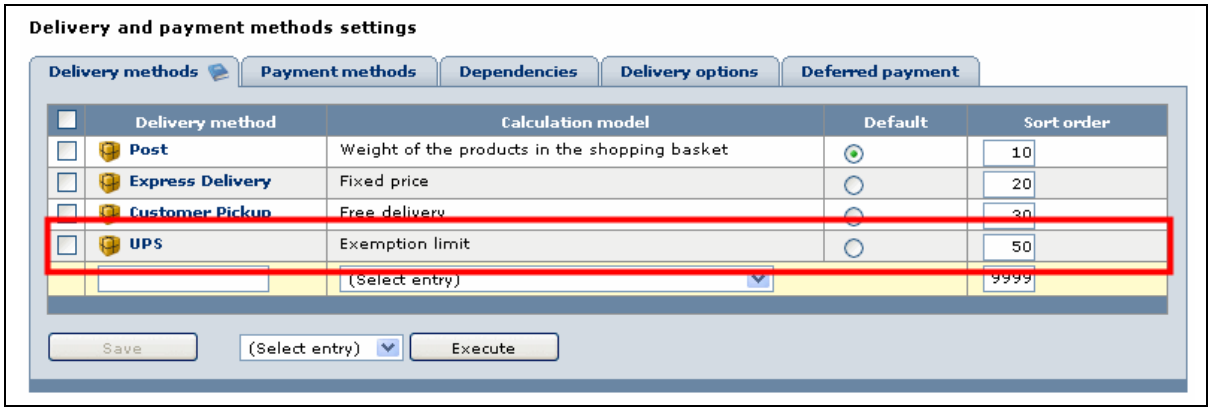

Figure 29: new delivery method imported

18. Scheduler

Within the application, there are various actions that can be or must be carried out at regular intervals. This includes actions such as updating eBay offers or product availability, as well as, regular database "clean up work".

The time-based execution of such actions works on a operating system-based scheduler concept.

The execution is performed by a Perl or (for UNIX) shell script. The timed execution plan for an action is managed in a configuration file.

A task consists of the action to be performed and the time-based plan.

The scheduler maintains all of these tasks. Command line tools are used to start and stop the scheduler.

18.1 Configuring Perl Script Tasks

Perl script tasks are managed in the *%EPAGES_CONFIG%/Scheduler.conf* configuration file. The configuration file is in the INI file format. It is case sensitive. Each task has its own section whose title is the name of the task. Comment lines begin with '#'. Each task is configured using the following parameters in its section.

Table 21: parameters for task configuration

Parameter	Description
IsActive	If this is not 1, the task is not run by the scheduler.
Command	Perl script to be executed
Machine	The task will be performed on the machines that are entered. If no machine is entered, the task will be performed on all machines on the system. When entering a machine, the first part of the host name, the computer name, should only be entered.
Cron	Execution time in UNIX cron tab format. If this is not set, the task will not be executed under UNIX.
Schtasks	Execution time in Windows *schtasks* format.
At	Execution time in Windows at format. If neither *schtasks* or *at* is set, the task will not be executed under Windows. If the Windows computer recognises the *schtasks* command, *at* is ignored.
Options	Options for the Perl script as defined in the command.
Loop	Loops are used to run a Perl script multiple times with various loop options (in addition to the standard options). The value entered here must match an entry of the *[LOOP]* section.

For example, the section for the task AutomateAutoCrossSelling in the *%EPAGES_CONFIG%/Scheduler.conf* file looks like this:

```
…
[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
…
```

Code example 70: example for task configuration in *Scheduler.conf*

Loops are used to run a Perl script multiple times with various loop options (in addition to the standard options). You can see a #LOOP example here:

```
[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
```

Code example 71: *[LOOPS]* section in *Scheduler.conf*

The last line explains, for example, that the Perl script is executed twice:

a. with the parameters -storename Backup -dbname storedb and
b. with the parameters -storename Backup -dbname sitedb

Changes to the %EPAGES_CONFIG%/Scheduler.conf configuration file will only be applied once the scheduler is restarted. See *Starting and Stopping*, *on page 121*.

18.2 Creating New Perl Script Tasks

Perl script tasks are always defined within a cartridge. In the */Data/Scheduler/* directory of the cartridge, the *Scheduler.conf* configuration file has to be created. The tasks are configured here. This configuration file has the same format as *%EPAGES_CONFIG%/Scheduler.conf*.

The Perl script which belongs to this is usually contained in the */Scripts* directory of the cartridge.

During cartridge installation, the sections from */Data/Scheduler/Scheduler.conf* are copied to *%EPAGES_CONFIG%/Scheduler.conf* if the section is not yet contained in *%EPAGES_CONFIG%/Scheduler.conf*.

The following are also important to remember:

- If a section is already contained in *%EPAGES_CONFIG%/Scheduler.conf*, it is **not** copied over during installation.
- If *Scheduler.conf* of the cartridge contains the *Loop* parameter that is not yet contained in the global configuration file *%EPAGES_CONFIG%/Scheduler.conf* in the *[LOOPS]* section, this section must be added to the *[LOOPS]* section manually.
- Sections must be removed manually from the *%EPAGES_CONFIG%/Scheduler.conf* if necessary.

18.3 Configuring UNIX Shell Script Tasks

Scheduler for UNIX shell scripts are configured globally in the *$EPAGES_CONFIG/Scheduler.d/* directory. In contrast to Perl script tasks, for which each task contains a section in the global configuration file, another configuration file is created for each UNIX shell script task in *$EPAGES_CONFIG/Scheduler.d/*. The contents of the configuration file for UNIX shell script tasks and the section for Perl script tasks are identical.

$EPAGES_CONFIG/Scheduler.d/ contains the following files:

- *appserver-*.env*: Tasks that are processed by ep_appl (in the crontab)
- *dbserver-*.env*: Tasks that are processed by ep_db (in the crontab)
- *webserver-*.env*: Tasks that are processed by ep_web (in the crontab)

Other file names are not allowed. The .env files are case-sensitive (as are all files under UNIX). The following is an example for the file *dbserver-RotateLogs.env*, executed by *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=
```

Code example 72: example for the file *dbserver-RotateLogs.env*

The variables *ISACTIVE*, *COMMAND*, *MACHINE*, *CRON*, *OPTIONS* and *LOOP* are processed by the scheduler. All other variables are optional variables. The meaning of the variables matches the scheduler variable of the Perl scripts. See *Table 21*.

The *RECIPIENT* parameter contains the address where all error e-mails are sent.

The values of the variables must be quoted if they contain special characters (such as spaces, for example in *CRON*).

The *LOOP* parameter contains all actual loops. In contrast to the Perl script scheduler, there is no shared loop management.

18.4 Starting and Stopping

To execute scheduler tasks, help scripts have been placed in %EPAGES%/bi:

- epagesScheduler.cmd for Windows and
- epagesScheduler.sh for Unix

The following arguments can be used for the scripts:

Table 22: Arguments for scheduler scripts

Argument	Description
start	Starts all scheduler tasks from *%EPAGES_CONFIG%/Scheduler.conf* or from *%EPAGES_CONFIG%/Scheduler.d/*.env*, meaning that the tasks are entered in the scheduler tables (crontab, at/schtasks table)
stop	Stops all scheduler tasks from *%EPAGES_CONFIG%/Scheduler.conf* or from *%EPAGES_CONFIG%/Scheduler.d/*.env*, meaning that the tasks are removed from the scheduler tables (crontab, at/schtasks table) Running processes are not stopped.
show	Shows all running scheduler tasks from *%EPAGES_CONFIG%/Scheduler.conf* or from *%EPAGES_CONFIG%/Scheduler.d/*.env*

Tasks are put in the following CRON tabs in UNIX:

- Perl script tasks in the ep_appl-crontab,
- Shell script tasks that begin with *appserver-* are entered into the *ep_appl* crontab,
- Shell script tasks that begin with *dbserver-* are entered into the *ep_db* crontab,
- Shell script tasks that begin with *webserver-* are entered into the *ep_web* crontab

In UNIX, the scheduler is restarted with the following command:

```
/etc/init.d/epages5 start
```

as follows:

- - starts the tasks for ep_web
- - starts the tasks for ep_appl
- - starts the tasks for ep_db

The tasks which are to be executed run inside a task wrapper. For UNIX, this looks as follows:

```
35 1 * * * /opt/eproot/bin/wrapScheduler.sh AutomateAutoCrossSelling
```

In Windows, the *wrapScheduler.cmd* is opened. The task wrapper executes the following commands:

1. If there is a file *%EPAGES_CONFIG%/Scheduler.d/TASK.env* (here *AutomateAutoCrossSelling.env*), the task is performed as a shell script. If the task does not begin with *appserver-*, *dbserver-* or *webserver-*, it will not be executed. The environment, that is defined in *%EPAGES_CONFIG%/Scheduler.d/*.env* is loaded and COMMAND with OPTIONS is executed.
2. If there is a TASK section in *%EPAGES_CONFIG%/Scheduler.conf*, the task is executed as a Perl script that is executed with the options defined in that section.

The task is only executed if the previous task has been completed. If the task is still running (and the *TASK.pid* file exists, for example *$EPAGES_LOG/Scheduler/cyansun.AutomateAutoCrossSelling.pid*), an error e-mail is sent.

18.5 Scheduler Task Output

For Perl script tasks, all error e-mails are sent to the *Error-Mail-Address* entered in the TBO. For shell script tasks, the error e-mails are sent to the value for the RECIPIENT variable in the .env file.

The tasks have various output files MACHINE.TASK.EXT that are saved in the following directories depending on the user:

- ep_appl: *$EPAGES_LOG/Scheduler*
- ep_db: *$SYBASE/ASE-12_5/init/logs*
- ep_web: *$HTTPD_ROOT/logs*

The following task output files are written by the *wrapScheduler.cmd* or *wrapScheduler.sh*:

Table 23: Task output files

Task output file	Description
MACHINE.TASK.pid (temporary)	Shows that the previous TASK is not yet finished, contains the Process ID MACHINE.TASK.log in UNIX.
MACHINE.TASK.pid (permanent)	Contains (error) output from all previous TASK calls
MACHINE.TASK.run (temporary)	Contains (error) output of the task: if this file is not empty, an error e-mail will be written and the MACHINE.TASK.log is appended
MACHINE.TASK.head (temporary)	If a header is contained that, if .run is not empty, is attached together with .run and appended to .log
MACHINE.TASK.mail (temporary)	E-mail that is sent, contains .head and MACHINE.TASK.run

MACHINE.TASK.head usually looks like this:

```
*** Output from program $RUN_SCRIPT ***
COMMAND: $COMMAND $OPTIONS > $LOG_FILE.run 2>&1
LOOP: $LOOP
DATE: $DATE
HOSTNAME: $HOST
USERNAME: $LOGNAME
```

19. Diagnostics Cartridge

The Diagnostics cartridge is a tool for displaying object and class structures, as well as database content. This tool can be used by both developers and designers, for example, for tracing lines of inheritance, for assigning PageTypes to objects or for querying actual attribute content.

The Diagnostics cartridge is included in the default installation, but needs to be installed separately.

19.1 Installation

The diagnostics cartridge is delivered in the *%EPAGES_CARTRIDGES%/DE_EPAGES/Diagnostics* directory, but may need to be installed:

1. `perl Makefile.pl`
2. `make install STORE=Store`

Store indicates the business unit in which you install the cartridge. The *%EPAGES-CONFIG%/Database.conf* file indicates the databases for Store. You can install the cartridge for all the databases. This give you an overview of all the objects including their content and relationships that are saved in the respective database.

> ***Caution:*** You can use the Diagnostics cartridge to access the database directly. If you use the cartridge on live systems, you need to change the default password immediately in order to prevent unauthorized access.

19.2 Usage

Call the cartridge using the following URL:

```
http://<servername>/epages/<db>. diagnostics/?ViewAction=ViewDiagIndex
```

Here *‹servername›* is the name of the server where the installation is running and *‹db›* is the database that you want to examine with the cartridge, for example *Store*. Of course, the cartridge for the database needs to be installed.

First, the sign-in page is displayed where you need to sign in. The default user name is *developer* and the default password is also *developer*.

> ***Caution:*** You can use the Diagnostics cartridge to access the database directly. If you use the cartridge on live systems, you need to change the default password immediately in order to prevent unauthorized access.

Change the sign in data in *Your Account* , see *Figure 30*.

After the call, you see the tool introduction page. See:

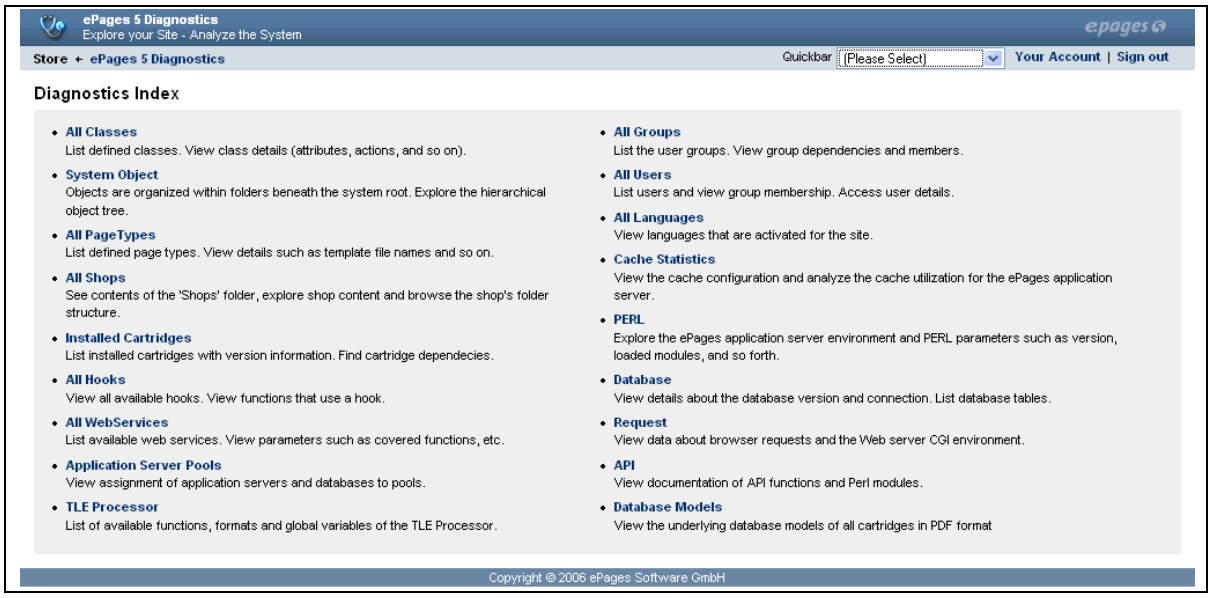

Figure 30: Diagnostics tool

Note: The package name is liked to the API documentation in the class structure. This is then opened in a separate window.

Part IV:

Design

20. Styles

A style contains all necessary information to display shop pages in the browser with structure, layout, and design. You can use styles to customize the Web pages of your ePages installation to fit your needs. Styles present an easy way to make storefront changes without changing templates or PageTypes.

The design of Web pages (colours, fonts, and so on) and the layout occurs using Cascading Style Sheets. The storefront display is generated from the style information of the respective *StorefrontStyle.css* file.

Working with styles is described in the following sections. Additional examples are shown in *Appendix C: Usage Examples (UE), on page 177*.

20.1 Selection Styles

Display in the shop is done using selection styles. These are listed in the MBO and the merchant selects the appropriate one under **Styles**.

The instructions for display are provided in XML files. Each selection style has a directory with the ID of the style which contains the necessary XML files. During selection of the style, the XML file is imported, the *StorefrontStyle.css* is generated from that and saved in the directory selected. This *StorefrontStyle.css* is used by the browser for display of the shop.

New selection styles are made available using cartridges. Each cartridge is created in its own directory:

```
<Cartridgename>/Data/Public/SF/Styles/<styleName>
```

The following files must be placed in the respective directory for the selection style:

Table 24: Selection styles files

File	Description
export.xml	contains attributes and values for the display of selection styles
img_small_stylepreview.gif	174 x 107 px – small preview graphic
img_medium_stylepreview.gif	450 x 280 px – medium preview graphic
img_colorpreview.gif	16 x 16px graphic – preview colour variation
StyleExtension.css	Stylesheet file for extension

In addition, background images and icons also belong to a style. These are available in their own image and icon sets. For each set type, there is another directory in the cartridge:

```
<Cartridgename>/Data/Public/SF/ImageSet/<imageSetName>


<Cartridgename>/Data/Public/SF/Icon/<iconSetName>
```

20.1.1 Creating Selection Styles

A selection style can be created as follows:

1. Extended design possibility activated in MBO

   The MBO design tool offers the ability to export existing styles. This allows you to easily create the required XML format. To do so, remove the entries
   `#REM<!--` and `-->#ENDREM`
   from the following 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
```

This will activate and display the export function for each style.

2. Manage the design using the design tool

   Create a new style called *MyStyle* in the MBO based upon an existing style. You will find a more detailed description of the design tool in the in the *Merchant User Guide* in the *Design* chapter.

> **Note:** You should not upload images, but provide them using an image set. For more on this, see *Creating an Image Set, on page 131*.

3. Export style

   Click in the style list at *MyStyle* on the **Export** link. The *export.xml* file will be saved in the following directory.

```
%EPAGES_WEBROOT%/Store/Shops/<ShopName>/Styles/MyStyle
```

   Create a cartridge and copy the file to the cartridge directory:

```
/Data/Public/SF/Styles/MyStyle
```

   Remove the following from the file:

```
  <Style reference='1' Alias='MyStyle'>
    <!--export level 2-->
    <Style StyleTemplate='/StyleTemplates/MileStones' reference='1' Path='.'
/>
  </Style>
```

   Code example 73: Selection from *Dictionary.en.xml*

4. Preview images and file *StyleExtension.css* creation

   Place all additional files from *Table 24* in the same directory. The *StyleExtension.css* fill can remain empty.

5. Register selection style in the database

   The new style must be registered in the database. To do so, create the *StyleTemplates.xml* file in the cartridge directory:

```
/Database/XML/
```

   The source code is as follows:

```
<?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>
```

Code example 74: *StyleTemplates.xml*

After this, the style will appear after installation in **MBO** ›› **Styles** at **Templates** ›› **Show all templates**.

6. Assign selection style to a category

   The new style should be assigned to a specific style category. To do so, create the *StyleGroups.xml* file in the cartridge directory:

   ```
   /Database/XML/
   ```

   The source code is as follows:

```
<?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>
```

Code example 75: *StyleGroups.xml*

After this, the style will appear after installation in **MBO** ›› **Styles** at **Templates** ›› **Display by sector** ›› **Travel & Tourism**.

7. Install the cartridge.

   Install the cartridge as described in chapter *Cartridges*, *on page 81*. During installation, the preview images and files from the cartridge directory

   ```
   /Data/Public/SF/Styles/MyStyle
   ```

   are copied to the following directory:

   ```
   %EPAGES_WEBROOT%/Store/SF/Styles/MyStyle
   ```

   The *StyleTemplates.xml* and *StyleGroups.xml* files are imported into the database. After that, the new style is available. Delete all caches to show the current display.

20.1.2 Creating an Image Set

The image set contains background images, logo, and content-separators. The image set is in the Style cartridge in the cartridge directory

```
/Data/Public/Store/SF/ImageSet
```

We will assume that you have placed a new style there according to *Creating Selection Styles*, *on page 129*. To create the image set for this style, proceed as follows.

1. Export/save new style

   If you have already installed the cartridge and edited the new style with the design tool, export it. Then you will have a current *export.xml* file. Copy this into the respective directory of your cartridge.

2. Copy image set

   Use a previous template for your image set. The standard image sets are found in at

   ```
   %EPAGES_WEBROOT%/Store/SF/ImageSet
   ```

   There is a directory with the name of the image set for each image set. Copy the content of an image set directory in your cartridge in the following directory

   ```
   /Data/Public/ImageSet/MyImageSet
   ```

   MyImageSet is the name of your new image set. The graphics file can be edited as necessary.

   Note: You should always use all graphic files of the set. Save unnecessary graphics as transparent GIFs.

3. Register the image set

   The name of the image set must be entered into *export.xml*. Use the *LayoutImageSet* parameter to set which image set belongs to the style. Look for this parameter in *export.xml* and define it as follows:

   ```
   LayoutImageSet='SF/ImageSet/MyImageSet'
   ```

4. Install the cartridge

   If the cartridge is already installed, you must uninstall it first. For more details see *Uninstalling*, *on page 86*.
   Install the cartridge as described in *Creating Selection Styles*, *on page 129* in the corresponding section. This imports the feature into the database

   ```
   %EPAGES_WEBROOT%/Store/SF/ImageSet
   ```

   and is available for the new style.

20.1.3 Creating an Icon Set

Icon sets are created similar to image sets. The following differences exist:

- Icon sets are placed in the cartridge directory at

  ```
  /Data/Public/SF/Icon
  ```

  Each icon set has its own directory with the name of the icon set.

- The default icon sets are in the directory

  ```
  %EPAGES_WEBROOT%/Store/SF/Icon
  ```

The new icon set will be saved in this directory during installation. The parameter for assigning icon sets in the *export.xml* file is called *LayoutIconSet*.

20.1.4 Sub-Styles

Variations can be created for styles. These are called sub-styles. The main usage case is for colour variations. The sub-styles are created in their own directories in the subdirectory */Sub-Styles* of their respective parent styles

```
<Cartridgename>/Data/Public/SF/Styles/MyStyle/SubStyles
```

and during installation are copied in the respective subdirectory:

```
%EPAGES_WEBROOT%/Store/SF/Styles/MyStyle/SubStyles/<substylename>
```

Sub-styles can be generated as described in *Creating Selection Styles*, *on page 129*. However, the following exception applies:

The *StyleTemplates.xml* file for sub-styles looks like this:

```
<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>
```

Code example 76: *StyleTemplates.xml* for sub-styles

The *StyleGroups.xml* file for sub-styles looks like this:

```
<StyleGroupMap reference="1" StyleTemplate="/StyleTemplates/MyStyle" >
  <StyleTemplateVariation
     StyleTemplate="/StyleTemplates/MyStyle/SubStyleTemplates/Red"
     Position="10" />
  <StyleTemplateVariation
     StyleTemplate="/StyleTemplates/MyStyle/SubStyleTemplates/Blue"
     Position="20" />
</StyleGroupMap>
```

Code example 77: *StyleGroups.xml* for sub-styles

The image sets for sub-styles are also located in the same place as the image set of the corresponding parent style. No subdirectories are used. The directories for sub-styles and image sets should contain the names of the parent styles. The directory for the image set of the sub-style *Sport* under the parent style *Basic* should be called *BasicSport*.

Appendixes

Appendix A: Performance Tuning

21. General Procedures

A key requirement of your ePages system is good performance. You can do this in various areas, during installation, using configuration files up to optimising HTML and PERL source code.

The following sections contain descriptions of ways to improve performance.

21.1 Page Caching

In caching, it is important to determine the balance between performance and updated content of your pages. You have the ability to cache entire HTML pages.

During caching HTML pages, these pages are saved as files and shown unchanged by later requests. The setting for this caching can be managed in the merchant administration with the *Optimisation* function in the menu item *Settings*. For more about this, read the corresponding chapter in the *Merchant User Guide*. Generally, you should have the longest possible length of validity for the pages and, when necessary, manually update for changes to pages.

Another possibility to improve performance through caching is to partially cache templates. For more on this, see *Partial Caching, on page 144*.

Note: The expiration time set for caches in optimisation applies to page caching and partial caching.

21.2 Template Processing

In addition to saving complete HTML pages, pre-compiled files are cached that only require the current dynamic content to be reloaded --these are the *ctmpl* files. For more information about *ctmpl* files, refer to *Template Process, on page 31*.

You can deactivate verification of the timestamp for *ctmpl* files. Possible changes are not tested any more and the page will be compiled from existing compiled data with new data from the database. Managing this verification is done with the parameter *DisableCtmlpStat* in the file

```
%EPAGES_CONF%/DataCache.conf
```

Table 25: Parameters for *DisableCtmlpStat*

Parameter	Description
DisableCtmplStat =0	This setting is the default and tests the HTML and XML files for a possible update time.
DisableCtmplStat =1	If this is set, the verification is skipped.

Caution: If *DisableCtmplStat =1*, changes to the templates (**.html*) and Dictionaries (for example **.de.xml*) will no longer be visible. This setting should only be made with a live system.

Note that if you deactivate the *ctmpl* test, the HTML pages are not updated as often. If you create the HTML pages again in the status using *Optimization*, the existing compiled data is used and substituted with current data. New compiled data are not created, even if changes exist to templates or language files.

21.3 Process Priorities

In the section *[GLOBAL]* of the file

```
%EPAGES_CONF%/WebInterface.conf
```

priorities for various processes are determined. With these configuration possibilities, long-running processes such as imports, duplication, and so on can be prioritised.

The reduces the wait for other requests because the server is not blocked through long-running processes.

A detailed customisation of priorities depends however upon the computer configuration.

Such settings are, for example.

Table 26: Process Priorities

Process Priority	Description
PRIORITY=HIGH	The application server runs with this priority
MONITOR_PRIORITY=ABOVE_NORMAL	Priority that the application server performs requests in monitor mode.
MANUAL_MONITOR_PRIORITY=NORMAL	Priority that the application server executes requests in manual monitor mode (imports and so on.)

The default setting is:

- PRIORITY=ABOVE_NORMAL
- MONITOR_PRIORITY=ABOVE_NORMAL
- MANUAL_MONITOR_PRIORITY=NORMAL

Note: For Linux/Unix, these processes cannot be set higher than NORMAL without having *root* permissions. This permission is not available for the user *ep_appl* under which the ePages service runs.

21.4 Reducing Response Times of the Initial Request

As soon as the initial request is sent to one of the application servers, this cache is filled with information about the object structure and PERL modules are compiled. This creates a fairly long request time. This initial request time can be reduced by loading the object structure during start of an application server and pre-compiling the PERL modules. To do so, in the file,

```
%EPAGES_CONF%/Hooks.conf
```

comment out the entry

```
[AppServerStartup]
; DE_EPAGES::Presentation::Hooks::AppServerStartup::OnAppServerStartup=10
```

by removing the semicolon.

21.5 Debugging Information

For the basics about debugging information, refer to *Template Debugging*, *on page 36*. Debugging information about source and runtimes of includes increase the load time of the HTML pages that must be loaded. This increases loading times. Turn off debugging mode in live operation in the file

```
%EPAGES_CONF%/log4perl.conf
```

by commenting out the entry

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

Put a semicolon in front of it.

21.6 Shop Settings

Performance can be improved by using specific settings in the merchant administration. Here are some examples:

- The number of products on the home page increases loading time (depending upon optimisation settings)
- Check the number of promotional products, since these products all need to be loaded when displaying the navigation element *Promotional products* (depending upon optimisation settings).
- hide unnecessary navigation elements

21.7 System Monitoring with Spy.pl

Use the spy monitor to verify the request router activity and data. This provides an overview of all application servers of an installation. Use the monitor to see information used to optimise the system and increase performance. These information include:

- Where do cache overflows come from?
- How fast are the machines?
- What is the utilisation of each individual machine?
- How is the system balanced?

Start the monitor after installation with the following URL:

```
http://<yourserver>/Monitor/spy.pl
```

After sending the requests, a Web site with various sections is shown that is reloaded every 5 seconds. The following sections are shown:

CacheStatistics section

Contents:

- Type and number of the cache updates of each application server
- Saved data which are communicated between application servers

Columns:

- 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

Server section

Contents:

- Listing of all application servers
- Pools to which the application servers are assigned
- Other parameters such as current request, last URL, last page

Columns:

- 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

Pools section

Contents:

- Listing of all application server pools
- How many and which application servers are assigned to the individual pools
- Overview of cache data

Columns:

- 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

RequestRouters *section*

Contents:

- List of all request routers with IP, port, status, and ping

Columns:

- 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

You can find the *spy.pl* file in the *%EPAGES_SHARED%\Monitor* directory after installation.

Check whether the following entry is located in the *WebInterface.conf* in the section for the request router:

```
MONITOR=10041
```

If the entry is not there, add it.

Installation on Windows with IIS 6

- You can perform a test by executing the *spy.pl* script from the command line. If information is shown in the console, the request router is running and the required permissions exist.
- Create a virtual directory in IIS 6 for the standard Web site for the directory which contains *spy.pl*:

  1. Default Web site ›› New ›› Virtual directory ›› Alias: Monitor, Path: "%EPAGES_SHARED\Monitor
  2. Set the following permissions for the virtual directory: READ, EXECUTE
  3. Enter the following mapping for the virtual directory (Monitor ›› Properties ›› Virtual directory ›› Configuration ›› Assignments):
     Executable file: "‹epages5directory›\Perl\bin\perl.exe "%s" %s"
     Extension: ".pl"

- In the server extensions, set "All unknown CGI extensions" to "Allowed"
- Restart IIS 6

Installation on UNIX

- Set the permission:

  - . /etc/default/epages5
  - cd $EPAGES_SHARED/Monitor
  - chown ep_appl:ep_web *.pl
  - chmod u=rwx,g=rwx *.pl

- Edit *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. Procedures During Development

During development, you can use specific methods or use specific functions in order to improve the performance of your system, for example, create high-performing cartridges.

22.1 Template Analysis

It is important to determine which execute times the individual includes show. With this information, you can identify the includes that increase the loading times of a template.

One possibility is to activate debugging information. See *Debugging Information*, *on page 138*.

This allows you to determine all includes and their execution times.

Using the TLE function *TimeThis*, you can determine the execution time of individual includes. The following TLE block measures the execution time of the INCLUDE template *Body* and shows this in red on the Web site.

```
#BLOCK("TimeThis", "Body", 1)
  #INCLUDE("Body")
#ENDBLOCK
```

Code example 78: Measuring the execution time of an include

The *TimeThis* block can also be used for section within a template. If the second parameter is *0* or not provided, the time is shown in the HTML as HTML commentary.

Another possibility to see an overview of all template includes is to execute the script *analyzeIncludes.pl* in the directory

```
%EPAGES_CARTRIDGES%\DE_EPAGES\TLE\Scripts
```

The following example shows the usage and the output:

```
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
```

Code example 79: Listing all includes of a URL with their execution times

22.2 Partial Caching

Use partial caching to determine whether individual template sections are created on-demand or loaded as prepared HTML code. This allows you to possibly reduce the time for creating the required page.

The TLE function *CachedInclude* executed within a *#BLOCK* instruction is the basis for this. The syntax for the function is:

```
#BLOCK("CachedInclude", <object> , <filename>) ... #ENDBLOCK
```

In *object*, enter an object for which the data will be cached. The static HTML code is saved under the file *filename*.

Caching refers to the complete content within the *#BLOCK* command. After processing the #BLOCK command, the created HTML code is shown as a static HTML file under the entered name. During the next page request, the function *CachedInclude* will test whether the file exists. If it exists, the HTML code will be used from the file.

Partial caching is a great idea for templates that do not have a very high number of possible variations. This also applies to templates or includes which have the same content for long periods of time.

Templates that contain user-specific data, for instance, such as shopping basket or user-specific product prices should not be cached, as then the data shown are not current.

In general, you should only investigate very slow includes for partial caching, as otherwise the overhead for caching also requires execution time.

Examples of partial caching are lists of promotional items or navigational elements, since these are shown the same for everyone normally.

The files are created in the directory

```
%EPAGES_STATIC%/Store/Shops/<shopname>/
```

A file like this should be as general as possible to be able to use in as many places as possible. However, every variation must have its own file. Therefore, the file name has a special meaning. If the name does not describe the variation clearly, files will be overwritten. Then the various contents will no longer be available. Not too many files should be created, however, in order to avoid too much system stress.

The following applies to file names:

- The file name consists of multiple sections.
- One part describes the content of the page area that is cached.
- One part consists of a TLE whose content defines the respective variation.
- These parts are separated by an underscore _.
- The GUID is prepended by the system.
- The file ending is *content*.

The following example demonstrates this:

```
#BLOCK("CachedInclude", #Shop.Object, "SF.SpecialOfferBox" . #CacheIncludesNames)
   ...
#ENDBLOCK
```

Code example 80: Configuring partial caching

The object, for whose display a specific area is cached is the *shop* object. The area which is supposed to be cached is in the storefront area, in which the promotional products are shown. It is defined in *SF.SpecialOfferBox*. The TLE *#CacheIncludesNames* contains information about the current display language, see *Table 27*. This TLE is used if the content changes only dependent upon the language.

For German as the display language, the file name is:

```
INC-436B4797-2D4C-607A-5558-AC14080F2485-SF.SpecialOfferBox_de.content
```

During the next call of this section, the system generates the file name using this TLE, notices that HTML content is saved for *de* and uses it immediately.

For English as the display language, the file name is:

```
INC-436B4797-2D4C-607A-5558-AC14080F2485-SF.SpecialOfferBox_en.content
```

Depending upon language settings, the correct HTML code can then be used.

The following TLE's are predefined:

Table 27: Filename TLEs for Partial Caching

TLE	Contents
CacheIncludesNames	Contains the language
CacheIncludesPrices	Contains language, region, and currency
CacheIncludesPager	Contains the table, sorting, sort order, table length (number of rows), and number of show page numbers in the footer of the table of the current page
CacheIncludesPagerPrices	Contains language, region, currency, current page of the table, sorting, sort order, table length (number of rows) and number of pages shown in the footer of the table

As an example, we will change for *Code example 80*, *on page 145* the TLE:

```
#BLOCK("CachedInclude",
        #Shop.Object, "SF.SpecialOfferBox" . #CacheIncludesPrices)
   ...
#ENDBLOCK
```

Code example 81: Changed TLE

This creates the following new file name:

```
INC-436B4797-2D4C-607A-5558-AC14080F2485-SF.SpecialOfferBox_de_de_DE_EUR.content
```

22.3 Using #LOCAL Instructions

During processing of lists, attributes or elements should not be read out multiple times, but instead should be provided as local variables.

Use the #LOCAL instruction to accelerate recurring access to hierarchal object structures.

The following example demonstrates this:

```
#LOOP(#Category.Products)#NameOrAlias #ENDLOOP
#IF(NOT #COUNT(#Category.Products)) list empty #ENDIF
```

Code example 82: Recurring object reads

In *Code example 82*, for every run the object hierarchy is queried. This is relatively time-consuming. The much faster variation is to read the products of the corresponding category and to save them in a TLE variable. During the run, this variable is always accessed:

```
#LOCAL("CategoryProducts", #Category.Products)
  #LOOP(#CategoryProducts) NameOrAlias #ENDLOOP
  #IF(NOT #COUNT(#CategoryProducts)) list empty #ENDIF
#ENDLOCAL
```

Code example 83: Using the #LOCAL instruction to improve performance

22.4 Separating Complex TLE Blocks

Separate complex TLE blocks or TLE functions or class attributes into their own code blocks. However, do not move the HTML into PERL functions.

The following example demonstrates this method: It should show a list of all visible products. The number of found products is shown from the list. A purely HTML solution is shown here:

```
#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
```

Code example 84: Function, HTML coded

You can improve the performance of this function in that you collect the data with a PERL function, process them, and save in TLE variables and only show the results in HTML:

```
…
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;
}
…
```

Code example 85: PERL function for data preparation

This function is then used in the HTML source code.

```
#LOCAL("TempVisibleProducts", #VisibleProducts)
count of visible products = #COUNT(#TempVisibleProducts)
 #LOOP(#TempVisibleProducts)
   #NameOrAlias #Description
 #ENDLOOP
#ENDLOCAL
```

Code example 86: Show data in HTML based upon the prepared TLE

Appendix B: Developer Notes

23. Adding E-Mail Events

To be able to add your own e-mail events to the system, do the following:

- Creating a MailType
- Creating a PageType and Assigning it to a Template
- Implementing the function
- Register the action
- Define the permission

Encapsulate the function into a cartridge as usual. For information about individual steps, we can refer to the *ContactForm* cartridge.

23.1 Creating a MailType

MailTypes are defined in the file *MailTypeTemplate*.xml*. You can substitute any file name extension for the asterisk (*). The file must be saved in the */Database/XML* directory.

You will find the necessary source code in *Code example 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>
```

Code example 87: MailType definition

You can enter optional parameters for the MailType definition that provide the necessary entry fields to the detail view of the corresponding e-mail event. These parameters are:

Table 28: Parameters used for MailType Definition

Parameter	Description
HasSubject="1"	The entry field for the subject line is shown and the merchant can enter a subject.
HasAdditionalText="1"	The entry field for additional e-mail text is shown and the merchant can enter the corresponding content.
HasHeader="1"	The entry field for a header is shown above the standard text and the merchant can enter content.
HasToField="1"	The entry field for a recipient is shown and the merchant can enter a recipient.

23.2 Creating a PageType and Assigning it to a Template

The presentation of individual e-mail messages is determined by individual HTML templates that are connected to the MailType using PageTypes.

A base PageType exists for all mail PageTypes called *ShopMail*. The areas *Page*, *Head*, *Body*, and *Content* are defined with the base templates here. Customized HTML templates Individual are assigned to each area of the for individual MailTypes if necessary.

The PageTypes for MailTypes are defined in the file *PageTypesMail.xml* in the directory */Database/XML* of your cartridge.

During PageType definition, the name must be used as the alias that is also used in the MailType as the PageType ID. Compare *Code example 87* and *Code example 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"
                  Template Name="Content" FileName="Mail/Mail-
ShopContactForm.Content.html" />
      </PageType>
    </Class>
  </Cartridge>
</epages>
```

Code example 88: PageType definition for a MailType

In the example, special HTML templates are assigned for the *Body* and *Content* areas. The HTML files are saved in the */Mail* subdirectory of the template directory of your cartridge.

23.3 Implementing the Function

The function that performs the sending of the mail is implemented in the */UI* directory of your cartridge as a PERL module.

It is important to pass the correct name for the MailType in the source code. Compare and. In addition, you must make sure that the optional parameters that you have entered in the MailType definition are also processed correctly in the code.

23.4 Registering an Action and Defining a Permission

Because sending the e-mail message is connected to an action, the action must be registered in the databank. See also *Registering Actions for Objects*, *on page 21*. Use or create the *Actions*.xml* file of your cartridge for this. The action is registered in the XML file as follows:

```
<?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>
```

Code example 89: Registering an action

Note that the function name must be the same as in the PERL module *FunctionName*.

Determine who can perform the function in the *Permissions*.xml* file. In our case, every customer:

```
<?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>
```

Code example 90: Assigning permission for the action

Here you use the alias for *Action* under which you registered the action.

24. Extension of Cross-Selling Types

Use cross-selling types to create relations or links between products. In ePages, the following types implemented: *CrossSelling* for manual cross selling, *Accessory* for accessories, and *ProductComparison* for product comparisons.

In the following, the normal process for creating additional cross-selling types is described in the example of the type *ReplacementPart* for replacement parts.

The unique features or important properties and files are explained. The basic concepts about creating and managing content of cartridges that is described in the previous chapters is assumed.

A table must be created for each cross-selling type and the *Product* class must be extended with three additional attributes.

The process is as follows:

- Define table
- Create classes
- Extend product attributes
- Creating Templates and PageTypes
- Register and implement functions

24.1 Define Table

You create a table with a reference to the name of the cross-selling type that you want to introduce. In our example, it is called *ReplacementPart*. The three columns *replacementpartid*, *productid* and *targetproductid* are defined for this table. The *productid* is the ID of the product that is assigned to the replacement part. The *targetproductid* is the ID of the product which is used as a replacement part.

The table is created using an SQL file in the cartridge directory */Database/Sybase/Tables*. One possible example of this is shown in *Code example 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
…
```

Code example 91: Creating a table for a new cross-selling type

At the same time, create a PERL module in the cartridge directory */API/Object* in which the functions for editing data sets in the table is implemented.

24.2 Create Classes

In addition, you must create a new class with the corresponding attributes. The new cross-selling type is based upon *Object* and is defined in the *AttributesReplacementPart.xml* file in the cartridge directory */Database/XML*. See *Code example 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"

Package="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"

Package="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>
```

Code example 92: Class definitions for a new cross-selling type

Implement the functions for editing the attributes in a corresponding PERL module in the cartridge directory */API/Attributes*.

24.3 Extending Product Attributes

The *Product* class must be extended with three attributes for each cross-selling type:

Table 29: Additional Attributes

Name	Description
‹crosssellingtypename›	Contains all products that as cross-selling products were directly assigned to a master product or a product without variations
SubProduct‹crosssellingtypename›	Contains all products that as cross-selling products were directly assigned to a product variation
Visible‹crosssellingtypename›	Contains all products that as cross-selling products were assigned to a product and are visible in the shop. In the case of a product variation, the visible cross-selling products of the master product are shown followed by the directly-assigned cross-selling products.

For our example, the three attributes are called:

- ReplacementParts
- SubProductReplacementParts
- VisibleReplacementParts

These attributes are defined in the *AttributesProduct.xml* file in the cartridge directory */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"
Value="VariationReplacementParts"
          />
        </Attribute>
        <Attribute Alias="VisibleReplacementParts" Type="ReplacementPart"
IsArray="1"
                   IsCacheable="0" IsExportable="0" IsObject="1"
                   Package="Training::ReplacementPart::API::Attributes::Product"
>
          <AttributeValue Name="Name" Language="en" Value="Visible
ReplacementParts"
          />
      </Object>
    </Class>

  </Object>
</epages>
```

Code example 93: Additional product attributes for a new cross-selling type

Implement the access functions for these attributes in a corresponding PERL module in the cartridge directory */API/Attributes*.

24.4 Creating Templates and PageTypes

You should generate new PageTypes with the corresponding templates for the display of new cross-selling type both in the back office and in the storefront. Regarding the back office presentation, you must decide whether the replacement parts are shown in their own tab or are shown and edited together with other cross-selling types.

The advantage of using a dedicated tab exists in the independence of the functionality when transferring the cartridge to other users.

If you show the replacement parts together with other cross-selling types, you must overwrite the original template. This can create problems if you give someone the cartridge and the user has already overwritten this template. This can create conflicts. Note this for your decision.

In our example we create our own tab. Since the individual tabs for product details are shown using the menu, you must extend this menu. For more information on working with menus, see *Dynamic Menus*, *on page 167*. A possible PageType definition is available in *Code example 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>
```

Code example 94: PageType for back office display of a new cross-selling type

In the example, the menu is extended that shows the individual tabs. The menu itself is defined in the *MBO-Product* PageType. It is referenced here. The order of the tabs is set by its own template.

The PageType *MBO-ReplacementParts* is based upon *MBO-Product* and represents its own template for the display of the content of the tab that is implemented in the assigned HTML file.

The corresponding ViewActions are defined here.

This is the same process for displaying replacement parts in the shop. Here, the detail data about a product are also displayed using a menu. Therefore, you must only extend this menu. See *Code example 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"
delete="1"/>
      </PageType>

    </Class>

  </Cartridge>
</epages>
```

Code example 95: Extending the menu for showing the shop in a new cross-selling type

You determine how substitute parts are shown in the shop with the assigned HTML template.

24.5 Register and Implement Functions

The actions that the merchant can perform regarding replacement parts must be registered in the database and the functions that belong to this must be activated.

In the */Database/XML* directory, create the *Actions*.xml* file and register the actions as shown in *Code example 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>
```

Code example 96: Registering the actions

The functions themselves are implemented in the PERL modules of the cartridge directory */UI*:

```
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;
}
…
```

Verify whether the following data were provided or already processed or must be processed:

- Setting the dependencies of other cartridges: *Dependencies.xml*
- Assign the corresponding permissions: *Permissions*.xml*
- If necessary, provide hooks
- Usage of form handling
- In case of localization: Usage of language tags and making language files available
- Note that the references to other objects, for example that these during deletion must possibly be deleted as well.

25. Creating Shops via Web Services and Scripts

In addition to creating shops in the business administration, it is possible to create shops via Web service or scripts. For more information about using Web services for ePages, see *Web Services*, *on page 101*.

This has the advantage that shops can be created via external systems or time-based.

To do so, use the Web service *ShopConfigService* in the *ShopConfiguration* cartridge. When calling the ePages Web service, there is a *WebServices* cartridge a client called *Client.pm* based on PERL. This class has a customized error handling function. Use this client to call the Web service from an ePages environment. You can use *Service WebServiceClient.pm* for calling from other systems. This is available in the ePages PartnerWeb.

The methods of the *ShopConfigService* Web service are defined in the WSDL file with all the parameters that belong to it.

The Web service *ShopConfigService.pm* can be found in

```
%EPAGES_CARTRIDGES%/DE_EPAGES/ShopConfiguration/API/WebService/ ,
```

the WSDL file which belongs to it (*ShopConfigService.wsdl*) can be found in

```
%EPAGES_CARTRIDGES%/DE_EPAGES/ShopConfiguration/Data/Public/WSDL/
```

or, after installation in

```
%EPAGES_SHARED%/WebRoot/Site/WSDL/ .
```

The most important functions of this Web service are *create*, *update*, and *delete*. To create a shop, open the *create* function with at least the following parameters:

Table 30: Parameters for *create*

Type	Comment
Alias	Unique shop name relating to the provider
ShopType	This shop type must exist so that the shop can receive all properties and features.
Database	Database that the shop is created in must exist
ShopAlias	Unique shop name relating to the database
ImportFiles	Contains all files that are made available for the shop with information about their properties and content. The XML file is listed here which is used for defining the ShopType used. The prerequisite for the import is that the application server can access the import files.

This function matches the process of creating a shop in the business administration. During shop creation in the user interface, however, the *Alias = Shop* alias is set and no additional import files can be entered. Only the XML file which is determined via the ShopType is used.

The *update* function requires the same parameters as *create*. The name of the database cannot be changed. This means that "moving" a shop from one database to another cannot be done this way. Use other import files to import multi-language content into the shop, for example.

When executing the function *delete*, all shop data are deleted from the store database. The basic information about the site database remain and are used for the shop history. Use the function *deleteShopRef* to delete all shop data completely from the system.

Note that the service is running on the site. This means that the proxy must be called as follows:

```
http://<servername>/epages/Site.soap
```

Calling the ePages Web service is described in *External Clients for ePages 15.3 Web Services*, *on page 104*.

The functions described previously can be executed via scripts. You can find the corresponding scripts in

```
%EPAGES_CARTRIDGES%/DE_EPAGES/ShopConfiguration/Scripts/
```

Through executing the scripts, the corresponding function of ShopConfigService is executed. The following is an example of creating a shop using a 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
```

The parameters for the Web service must be passed explicitly for this. Scripts can only be started locally but are useful for tests and for scheduled executions.

A special feature applies for the script *deleteShop.pl.* In this script, the functions *delete* and *deleteShopRef* are executed sequentially in order to delete all shop data.

26. Patching Cartridges

Patching is the fitting of data to a later version of a cartridge. As a prerequisite, before beginning the patch, all files that must be replaced with new files in the cartridge and new files must be added.

Note: Before patching, create a backup of the current version.

The foundation for patching is a framework with supports patches in two steps:

1. Updating or changing the database structure
2. Updating data based upon the new database structure

To be able to execute these processes, the *Cartridge.pm* file must be extended according to *Code example 97*:

```
…
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 current 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';
}
…
```

Code example 97: Modification of *Cartridge.pm* for patching

Two new parameters are entered into the *new* function, *Version* and *Patches*. The *Version* parameter presents the target version. This indicates which version the patch updates to. In the *Patches* parameter, all versions are listed to which the target version can be updates. In the previous example this means that this cartridge can be updated from version 1.0 and 1.1 to version 2.0.

The *patchDBStructure_** function contains all methods to make all necessary changes in the database.

The following naming convention applies: The number of the source version to be updated is appended to the function name patchDBStructure_ . Periods should be substituted with underscores. The number of the target version is defined in the *return* command. According to the example, all database changes are implemented in the *patchDBStructure_1_1* function that are necessary to upgrade from version 1.1 to 2.0.

If database changes are required during an update from 1.0 to 1.1, these must be contained in the function patchDBStructure_1_0 *with the* return *command return '1.1';*.

Do not call any API functions until all database changes are complete.

The functions *patch_** contain all methods for modifying or moving data according to the changed database structure.

The naming convention applies here that the number of the source version is appended to the function name, whereby periods are substituted by underscores. The *return* command contains the number of the target version.

The patch process is started after calling the script:

```
perl %EPAGES_CARTRIDGES%\DE_EPAGES\Installer\Scripts\patch.pl -storename Store
     <vendor>::<cartridgename>
```

After starting, all the functions *patchDBStructure_** are executed. After this, the functions *patch_** are processed.

If errors occur during the process, the script can be restarted after fixing the errors. Functions that were already executed correctly are not executed again. The process begins with the execution of functions that previously had errors.

There is a table in the store database called *Cartridge* where all cartridges are listed which are installed. Each cartridge has, among others, the *dbstructureversion* and *version attributes*. A new version number is entered into *dbstructureversion* as soon as the *patchDBStructure_** functions are executed successfully. The new version number is entered into *Version* as soon as the *patch_** functions are successfully executed. Through comparing both entries, you can check whether the patch was successful or not. This comparison provides no information about the accuracy of the data after the patch.

You can patch from any version to the target version, however it is recommended to execute step-by-step updates from one version to the next.

To summarize the process of patching cartridges up through testing the patch:

1. Create a backup on the current version and the current database and all other data and files that might be influenced by the patch.
2. Increase the version number in the *new()* function in *Cartridge.pm*
3. Include the current version number in the list of versions that can be patched
4. Implement the *patchDBStructure_** and *patch_** functions as necessary
5. Start *patch.pl*
6. Update the cartridge list of the affected database in the technical administration
7. Clean the cartridge directories of files that are not used any longer among other things.

The process of patching cartridges on a live system:

1. Create a backup on the current version and the current database and all other data and files that might be influenced by the patch.
2. Copy all the new and changed cartridge files into the directory with the current version
3. Start *patch.pl*
4. Update the cartridge list of the affected database in the technical administration
5. Clean the cartridge directories of files that are not used any longer among other things.

27. Integrate your own online Help

The function used to show help is integrated in a template through one or more includes. On the display page for products in the merchant back office you can see a Help call in the search area and a Help function in the active tab.

You define which Help is shown together with the ViewAction to show the area for which the help is to be shown.

To create a Help page for a page, do the following:

1. Create a HTML page with Help content and copy it into the necessary place in the file directory.
2. Link the Help page with the corresponding ViewAction
3. Implement the display code in the template

27.1 Make the Help Page Available

The files for the default online Help are found in the directory

```
%EPAGES_WEBROOT%/Doc/Help/<language>/<administration>
```

The Help pages for the German merchant administration pages, for example, can be found in

```
%EPAGES_WEBROOT%/Doc/Help/de/MBO
```

In addition to the directories for the administration page, you can create a separate directory for your help pages and make these available. Language-dependent Help pages can be distributed among the various directories for the individual languages.

If you create a cartridge for your function and its Help, the Help pages are copied to the correct places during the installation process. See *Installing - nmake, on page 85*. The prerequisite for this is that you provide the correct structure in your cartridge.

To do so, create a new subdirectory in your cartridge:

```
/Data/WebRoot/Doc/Help/<language>/<cartridgename>
```

Create the HTML files for your Help pages in this directory. We recommend placing the images in their own subdirectory for organizational reasons.

For a German Help file for your cartridge, the HTML file in your cartridge directory must be placed as shown here:

```
/Data/WebRoot/Doc/Help/de/MyCartridge/MyOwnHelp.html
```

Note: You can link from your Help page to an ePages standard Help page. Integrate the following link into your page to do so, for example: *Help*.

27.2 Assigning a ViewAction

You must link the Help for a page to the action for displaying this page. You do this in the *Actions*.xml* file of your cartridge.

The syntax is shown in the following example:

```
…
<Action Alias="MBO-ViewMyCartridgeGeneral"
Package="DE_EPAGES::MyCartridge::UI::Shop"
        FunctionName="View" delete="1" >
  <AttributeValue Name="HelpFileTopic" Value="MyCartridge/MyOwnHelp.html" />
      </Action>
…
```

Code example 98: Assigning Help to a ViewAction

27.3 Display Code in Templates

If you create new tabs in the MBO based upon the existing, the Help is automatically integrated.

To be able to see the Help pages in their own templates, an element for this must be integrated into the template. A book icon is used as the standard element for this in the ePages system.

This icon is assigned to the following function:

```
…
<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>
…
```

Code example 99: Function for showing the correct Help page

Examples, such as those used in this code in the template can be found in the files *Backoffice.Tabs.html*, (for usage on tabs) and *MBO-ProductManager.Toolbar.html* (usage in the search area). These files are located in your ePages installation.

28. Dynamic Menus

Dynamic menus are used in places where various functionality should be shown flexibly. The advantages of this method are the extendibility of the menu via cartridges, localization of individual menu entries or inheritance of menus for of child PageTypes.

Typical menus are the main navigation bar, the context bar, the context menu, or the tabs to show object details.

A menu is defined in a PageType. Menu entries are sorted within its definition of through other cartridges. The content of a menu entry is described in a template. The assignments are created using menu and template names.

Using the main navigation bar as an example, you can understand the usual process:

A menu with entries is defined in the PageType *MBO* and a template for the menu display is assigned. See *Code example 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>
…
```

Code example 100: Menu definition from the *Presentation* cartridge

In the *Menu* tag, the menu is created with a unique ID. Set individual menu entries within the menu tag.

For each entry, an ID and an action must be entered. You describe the action with a name and the object for which the action is registered.

Use Position to determine the order in which the entries are shown.

For the complete menu as well as for each menu entry, you must use the template to describe content and presentation. This assignment, as every area assignment in the PageType is performed in *‹Template Name=… />*. The name of the area must match the ID of the menu or the menu item.

For our example, this means:

- A menu is defined with the ID *Menu*
- The menu contains the entries *Menu Marketing* and *Menu Settings* with the action and the position listed
- The menu entry *Menu-Marketing* is displayed via the *MBO.Menu-Marketing.html* template.
- The menu entry *Menu-Settings* is displayed via the *MBO.Menu-Settings.html* template.
- The menu itself if displayed through the *MBO.Menu.html* template.

The name of a menu template must be unique within a PageType. Therefore, it is recommended to retain naming conventions which consist of *‹menu_name›-‹entry_name›*.

The menu itself is often displayed in the template using a loop:

```
<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>
```

Code example 101: Menu display

A template for the menu entry can be seen in *Code example 102*:

```
<a href="?ViewAction=#URLAction&amp;ObjectID=#Shop.ID"
   Onmouseover = "changeImage(

'manager_marketing','#StoreRoot/BO/icons/mbo_manager_img_marketing_mouseover.gif'
   )"
   onmouseout= "changeImage(

'manager_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>
```

Code example 102: HTML code for menu entry

The main navigation bar is a typical example of a menu which was extended through other cartridges. For example, the *Customer* cartridge provides the *Customer* entry that is used to access the customer manager.

The corresponding menu extension can be seen in *Code example 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>
…
```

Code example 103: Menu extension

You must reference the PageType in which the menu was created. Then enter, using a reference, the ID of the menu which you would like to extend.

You define the new entry like the entries during creation of a menu.

You use the ID to assign this entry to a template with presentation information.

After installing the cartridge, the new entry is available in the menu.

Another example for extending menus is available in *UE 7: New Batch Processing Commands in the MBO*, *on page 195*.

29. Shopping basket template and LineItems

The shopping basket template provides the basic Web page structure for each individual step from the shopping basket view up to the display of the order confirmation. This complex template is structured using menus in order to create optimal conditions for extending and customizing your shopping basket.

Figure 31 shows the basic structure of the shopping basket template defined in PageType *SF-Basket*.

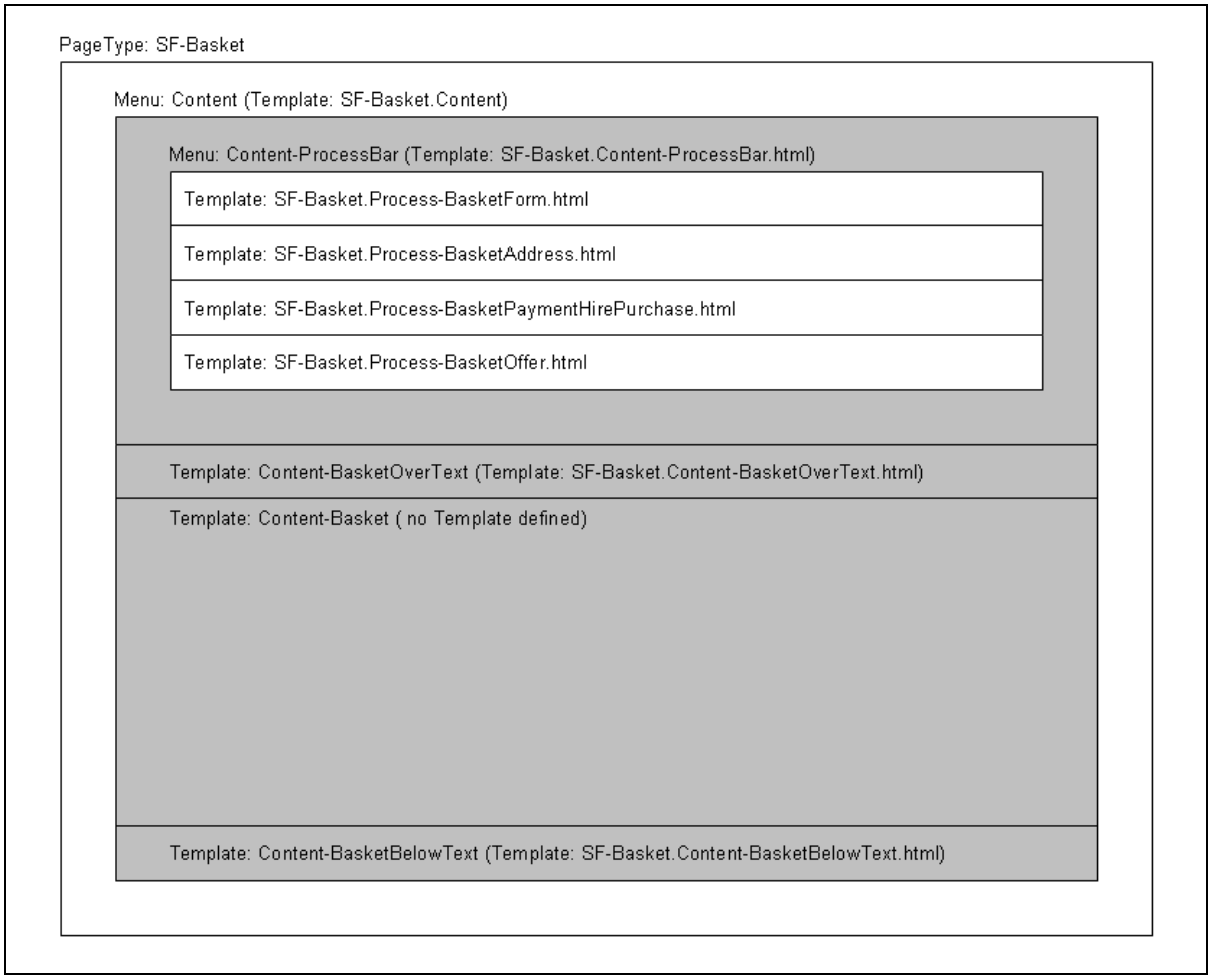

Figure 31: Basic shopping basket template structure

Note that *SF-Basket* defines the working area of the page. The surrounding page areas, such as the header or left or right area, are displayed using the parent PageTypes.

There is no default template for the *Content-Basket* menu. The associated template for each step of the order process is provided individually.

Each step is defined in its own PageType based on *SF-Basket*. The corresponding structural changes are defined or specific templates are assigned in these PageTypes. In *Code example 104*, you see the PageType *SF-BasketOffer* for the step where the order overview is displayed:

```
…
<PageType Alias="SF-BasketOffer" Base="SF-Basket" delete="1">
   <Template Name="Content-Basket" FileName="SF/SF-BasketOffer.Content-
Basket.html" />
   <ViewAction URLAction="ViewOffer" />
</PageType>
…
```

Code example 104: Template definition for the Content-Basket menu

In this example, the template, in which the order overview display is implemented, is assigned to the *Content-Basket* menu.

This results in the following basic options for customizing the order process:

- Overwriting existing templates
- Adding new steps based on new PageTypes
- Extending the menu structure of the shopping basket template

When adding additional steps, note that you also need to apply these steps in the status display, that is, you need to extend the *Content-ProcessBar* menu.

One example of extending the menu structure is the *Coupon* cartridge. The area for entering coupon codes should be displayed in the shopping basket. To add this area in the template, an entry is added to the *Content* menu. In the *Code example 105*, you see the corresponding 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>
…
```

Code example 105: PageType with menu extension

Reference is made to the *SF-BasketForm* PageType and the *Content* menu defined there. An additional entry with the corresponding template assignment is created for the menu. This template describes the layout of the additional page area and its functionality.

The shopping basket itself, with the list of products ordered, the selected shipping and payment methods, as well as various price discounts, is displayed using LineItems *:*

29.1 LineItems

LineItems are the individual items in a shopping basket table or in an order including the associated documents. They contain information about the individual products, payment methods, percentage discounts, and so on used to calculate the value of the shopping basket.

LineItems are displayed not only in the shop but also in the back office, and generally in the following locations:

Table 31: Displaying LineItems

Area	Display
shop	Shopping basket form
	Order summary
	Order confirmation (shop view and print view)
	My account: Order (shop view and print view)
	Minibasket (navigation element)
Back office	Order - display and edit
	Documents (invoice, packing slip, UPS packing slip, credit note) - display and edit
E-mail	Order confirmation
	Status changes

Depending on the function and content, LineItems must be displayed different. For this reason, each LineItem type has its own template.

The prerequisite for this is the definition of a corresponding LineItem type. Using a "base" LineItem in the LineItem class, a separate LineItem type can be derived for each item type. For the most widely-used line items, LineItems have been defined that are structured hierarchically and inherit from each other. You can display the overview of these default LineItems using the Diagnostics cartridge:

- The entire is list is displayed under *All Classes*.
- You can see the first level in the hierarchy and follow the individual branches of the structure under *All Classes* ›› *LineItem*.

If no separate template has been defined for a LineItem type, the template for the parent LineItem type is used. For more on using the templates of parent classes, see *Object Method template*, *on page 45*.

The following views for LineItems are a result of the functional layouts:

Table 32: LineItem views

View	Template type	Function
LineItems	ContentLine	Standard view for shopping baskets and orders
EditLineItems	EditContentLine	Standard view for orders with the edit function
MiniLineItems	MiniContentLine	Display in the navigation element for the minibasket
SupplyLineItems	SupplyContentLine	Standard view for packing slips
EditSupplyLineItems	EditSupplyContentLine	Standard view for packing slips with the edit function
NegLineItems	NegContentLine	Standard view for credit notes (negative values)
NegEditLineItems	NegEditContentLine	Standard view for credit notes (negative values) with the edit function

According to this principle, you can define your own LineItem types or separate views at any time and whenever necessary.

An example of an order display in the MBO in normal view including the edit function illustrates this principle. The template type *ContentLine* is used for the simple display of shopping basket items in an order in the MBO:

```
…
#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
```

Code example 106: Displaying LineItems in the order view

The template specifically used for the individual LineItem type is defined in the PageType assigned to display the specific page.

For this, a search is made for the template definition, for example, *ContentLine[<lineitemtyp>]*. To display the delivery method, a search is made for the template *ContentLineLineItemShipping* and the associated HTML file used for this display.

If no *ContentLineLineItemShipping* template has been defined, the system looks for a template called *ContentLineLineItem according to the class structure*. If this has also not been defined, the system resorts to using the general template *ContentLine*.

This illustrates how to use the structural guidelines for creating template names for defining and applying specific templates per class. If the specific definition is missing, the more general template in the parent class is processed.

For our example, if the order is displayed in edit mode, the template type *EditContentLine* is used instead of the template type *ContentLine* :

```
…
#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
…
```

Code example 107: Displaying LineItems in the edit view for orders

This provides the corresponding functions for editing the information. This time, to display the delivery method when editing needs to be done, the *EditContentLineLineItemShipping* template along with the assigned HTML file is used according to structural guidelines.

Appendix C: Usage Examples (UE)

30. UE 1: Integrating your own .css file

This example shows you how to include your own static stylesheet file (css) into the system. The files to do so are in the attached cartridge examples, in the */E1_MyStaticStyle* directory.

The stylesheet file is included using a header in the respective templates. Proceed as follows:

1. Create a cartridge.

   Create a cartridge with the name MyStaticStyle. For the basics about creating a cartridge, see *Cartridges, on page 81*.

2. Create stylesheet file

   Place the stylesheet file in the following cartridge directory:

   ```
   /Data/Public/Shops/DemoShop/Styles/MyStyle.css
   ```

   Then enter the following code:

   ```
   .NewsList a {
     color: red !important;
   }
   ```

   Code example 108: Code in *Mystyle.css*

3. Register link to stylesheet file

   The reference to the additional stylesheet file must be included in the header of the Web site. The header is defined using a PageType. You must extend this definition. To do so, create the *PageTypesSF.xml* file in the

   ```
   /Database/XML/
   ```

   cartridge directory with the following content:

   ```
   <?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>
   ```

   Code example 109: Code in the *PageTypesSF.xml*

   The header will be extended with the section *Head-MyStyle* and the template *SF.Head-MyStyle.html* is included. In this template, the code for the reference to the stylesheet file is found:

```
<link href="#Shop.WebPath/Styles/MyStyle.css"
  rel="stylesheet" type="text/css" />
```

Code example 110: Reference to stylesheet file

The file *SF.Head-MyStyle.html* is created in the cartridge directory

```
/Templates/SF/
```

The basis for this can be found in *PageType Concept*, *on page 39* and *Dynamic Menus*, *on page 167*.

4. Install the cartridge

   Install the cartridge as described in chapter *Installing - nmake*, *on page 85*. You see the results on the home page of your demo shop. The headings of the news articles in the news list are written in red.

31. UE 2: Extending the Storefront Style

In addition to static stylesheet extensions, you can create your own stylesheet entries so that they respond to changes from the design tool. To do so, you must use TLE's in your stylesheet entries. The basis for this can be found in *Templates*, *on page 31* and *TLE*, *on page 61* and *Cartridges*, *on page 81*.

The following example shows how to use your own style sheet entries so that these can be changed using the Design Tool. The basis for this is the *SF-Style.StyleExtension-PartnerStyles.css* file that can be used for this. The original file is located in the following directory:

```
%EPAGES_CARTRDIGES%/DE_EPAGES/Design/Templates/SF
```

You must overlay this file with your own cartridge. Proceed as follows:

1. Create a cartridge

   Create a cartridge with the name *MyDynamicStyle*. For more details see *Creating a Cartridge Structure*, *on page 83*.

2. Create the overlay template

   Create the following directory in the cartridge:

   ```
   /Data/Private/Templates/DE_EPAGES/Design/Templates/SF
   ```

   Place the file *SF-Style.StyleExtension-PartnerStyles.css* in this directory and enter the following code:

   ```
   .NewsList a {
      color: #ContentHotDealPriceColor[color] !important;
   }
   ```

   Code example 111: Content of *SF-Style.StyleExtension-PartnerStyles.css*

   As with the previous example, the colour for the headings of the news articles in the news list of the start page of the demoshop changed. In this case, the color is dependent upon the value of the *ContentHotDealPriceColor* TLE. You can set this colour in the design tool.

3. Install the cartridge

   Install the cartridge as described in *Installing - nmake*, *on page 85*.

4. Generate new style

   Open the design tool for the current style in the MBO. Change the value for the file *Home page* under **Customise** ›› **Content area** ›› *Prices* and save. A new *Storefrontstyle.css* file will then be generated. The file is in the directory:

   ```
   %EPAGES_WEBROOT/Stores/Shops/DemoShop/Styles/<currentStyle>
   ```

   Open this file. In the bottom part is the section *StyleExtension-PartnerStyles*. In this section, you can find stylesheet entries from *Code example 111*, with the current value that you set with the design tool. You can see the effect on the home page. If you cannot see any changes, delete the cache.

32. UE 3: Changes in the template

Content of this example is overlaying the home page of the demo shop and the use of TLE's in templates. The basis for this can be found in *Templates*, *on page 31* and *TLE*, *on page 61* and *Cartridges*, *on page 81*.

You must first determine the template whose content you want to change and overlay it. Proceed as follows:

1. Identify template

   Decide which template shows the content area of the home page. Activate debugging. See Template Debugging, on page 36. View the source code of the home page. You can see the cartridge path and name of the template you need. In our case, *SF-Shop.Content.html*.

2. Create a cartridge

   Create a cartridge called *MyHomePage*. For more details see *Creating a Cartridge Structure*, *on page 83*.

3. Create the overlay template

   Create the following directory in the cartridge:

   ```
   /Data/Private/Templates/DE_EPAGES/Catalog/Templates/SF
   ```

   Place the file *SF-Shop.Content.html* in this directory with the following code:

```
<!-- 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
```

Code example 112: Example content for *SF-Style.StyleExtension-PartnerStyles.css*

Section 1 shows the simple display of TLE variables.

Section 2 shows the usage of the *LOOP* TLE statement with the example of listing categories.

Section 3 shows the usage of an *IF* statement in a *LOOP*.

Section 4 shows the usage of a WITH statement. A specific object context is set by this. The following TLE variables refer to this context.

Section 5 shows the usage of TLE variables with calculation and the usage of the LOCAL statement.

Explanations about the TLE variables and instructions can be found in *TLE*, *on page 61*.

4. Install the cartridge

Install the cartridge as shown in *Installing - nmake*, *on page 85* describe and open the home page of the demo shop.

33. UE 4: Customizing the Back Office Design (Branding)

Users often require the back office design to be customized. We will demonstrate how to use style sheets and overlaying to change the design. In this example, the changes to the MBO back office design should be done using local overlaying. If you would like to encapsulate your design in its own cartridge to make installable, refer to *Cartridges*, *on page 81*.

As mentioned previously, the corresponding style sheet file is located in the following directory:

```
%EPAGES_WEBROOT%/Store/BO/BackofficeStyle.css
```

The subdirectory */icons* is used for the associated images. The use of this style is defined in the following HTML file:

```
%EPAGES_CARTRIDGES%/DE_EPAGES/Presentation/Templates/Backoffice.Style.html
```

In this file, you specify which styles will be used to display the merchant back office. See *Code example 113* or the original file in the directory indicated.

```
<link href="#StoreRoot/BO/BackofficeStyle.css" rel="stylesheet" type="text/css"
/>
```

Code example 113: Specifying the styles for the back office

If you want a new design, you can modify the *BackofficeStyle.css* file or integrate your changes in your own .css file. We recommend creating a new file. Define a new *css* file, for example, called *NewBackofficeStyle.css* in the same directory. If you want to use your own icons, put them in the */icons* subdirectory for the *BackofficeStyle.css file*:

```
%EPAGES_WEBROOT%/Store/BO/icons
```

Note: If you accidentally overwrite the .css file or the icons, you can always retrieve the original files from the original cartridges. Do not overwrite the files in the original cartridge directories.

For the next step, you need to let the system know of your new file with the style changes. For this, you need to customize the *Backoffice.Style.html* file. But since the original files should not be modified, create a file of the same name in the "overlay directory" and make your changes in this file:

Copy the *Backoffice.Style.html file into the following directory*:

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

For information about overlaying, refer to *Overlaying Templates*, *on page 35*. Open this file and extend the source code as in *Code example 114*.

```
<link href="#StoreRoot/BO/BackofficeStyle.css" rel="stylesheet" type="text/css"
/>
<link href="#StoreRoot/BO/NewBackofficeStyle.css" rel="stylesheet"
type="text/css" />
```

Code example 114: Integrating the style changes

If you now call up the merchant back office, the modified design should be displayed. Compare *Figure 32* and *Figure 33* as an example.

Figure 32: The original default back office design

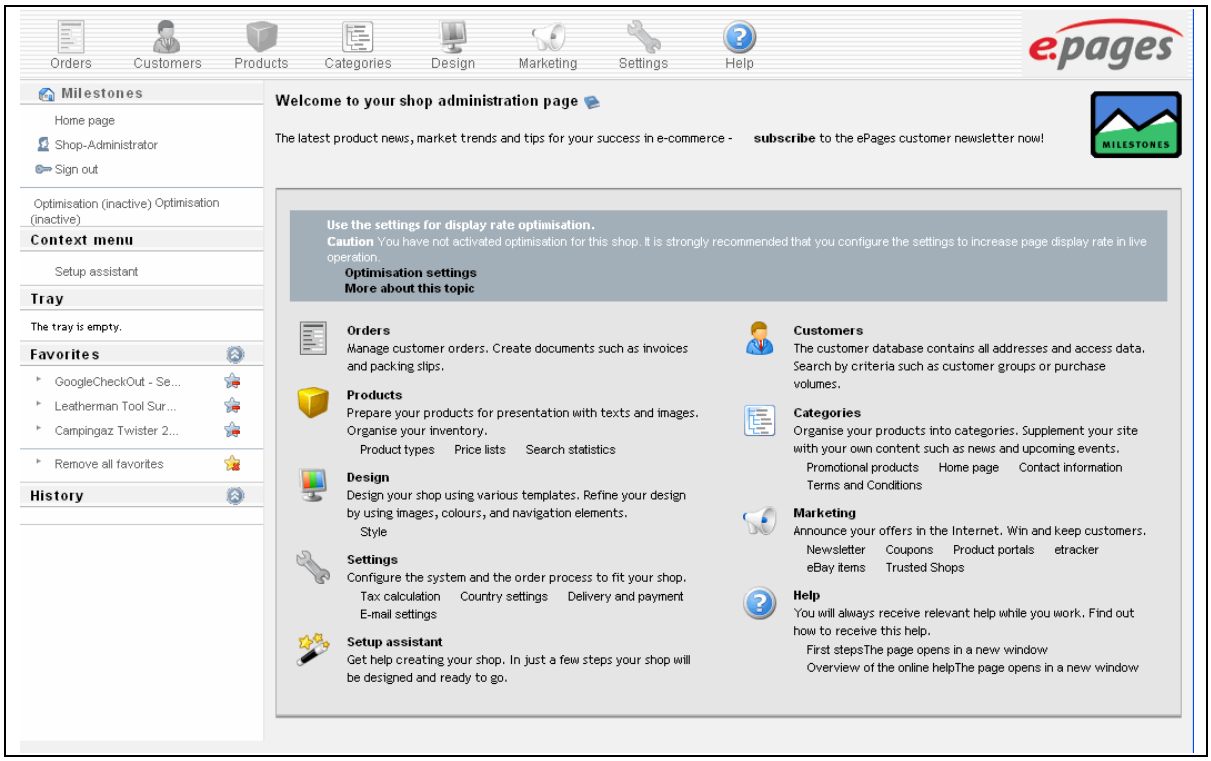

Figure 33: The back office after the design change described

The *TinyMCE* WYSIWYG editor is a platform-independent Web-based HTML application that is integrated into the ePages system. The design definitions are distributed among various HTMLS, js, and CSS files. To make design changes here, you must look for find and change the files in the *%EPAGES_WEBROOT%/Store/tinymce* directory. These changes can be overwritten by upgrades.

34. UE 5: Deactivating the Design Tool

The goal is to modify the merchant administration in such a way that the merchants have no access to the Design Tool and Setup Assistant, thereby preventing them from changing the design of their online shops. One way to do this is by overlaying certain templates. Basic information about this can be found in *Templates*, *on page 31* and *TLE*, *on page 61* and *Cartridges*, *on page 81*.

The Design Tool and the Setup Assistant are called using links in the templates. These links need to be identified and deleted from the template.

The merchant can access the design tools via the following links:

- Links in the context menu to the Setup Assistant
- Links to the Setup Assistant and the *Design* in the overview of functions on the home page
- Link to the *Design* in the main navigation bar
- Links to the *Design* in the *Related Topics* section on various pages

To deactivate the design tool, do the following:

1. Identify template

   You need to determine which template will display which information on the Web page. To do this, activate the debugging information in the source code, see *Template Debugging*, *on page 36*.
   For our example, the following applies:

   - The link to the Setup Assistant is the only entry in the context menu. Therefore, the entire context box should not be displayed. Look for the template that displays the boxes in the left navigation bar.
   - You need to remove the *Design* item from the main navigation bar. Look for the template that displays the main navigation bar.
   - The function overview on the home page contains links to the Setup Assistant and to *Design*. These links must be deleted.
   - If in *Related Topics*, the only entry is the link to the navigation bar, the entire entry may not be displayed. Otherwise, the item with the link needs to be deleted. Look for the template that displays the *Related Topics*.

   Open the merchant administration page and display the functions in the browser. In the Web page source code, get the name of the template including the cartridge information from the INCLUDE. The corresponding action is also displayed. You can use this information to look for all the templates in which this action will be executed. This is how you determine which templates you need to edit.

2. Create a cartridge

   Create a cartridge called *HideDesign*. For more details see *Creating a Cartridge Structure*, *on page 83*.

3. Create the overlay template

   Copy these templates from the original installation directories into the corresponding working directories for the cartridge. For example, copy the file *Backoffice.ContextBar.html* from

   ```
   %EPAFES_CARTRIDGES%/Cartridges/DE_EPAGES/Presentation/Templates/
   ```

   into the cartridge directory:

   ```
   /Data/Private/Templates/DE_EPAGES/Presentation/Templates/
   ```

   Edit the copied templates so that the functions mentioned above are no longer visible.

In the original *Backoffice.ContextBar.html* file, all the necessary boxes for the left navigation bar for the merchant administration page are displayed, see *Code example 115*.

```
#BLOCK("MENU","ContextBar")
  #INCLUDE(#Template)
#ENDBLOCK
```

Code example 115: Template for displaying the boxes in the left navigation bar for the MBO

You now need to change the code so that the context box with the link to the Setup Assistant is not displayed, see *Code example 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
```

Code example 116: Hiding the context box

4. Install the cartridge

   Install the cartridge as described in *Installing - nmake*, *on page 85*. Delete the cache, remove the respective ctmpl files and restart the ePages service.

The files for this example can be found in the *E5_HideDesign* directory included in the example cartridges. The templates to hide the context box and design main menu point *Design* are contained in the cartridge. You must identify the templates to hide the respective Related Topic yourself, copy them to the cartridge, and change them. Comment out the corresponding links.

35. UE 6: Design Changes using PageTypes

The focus of this example is how to work with PageTypes. We will restrict ourselves to design changes right now in order to keep this cartridge example from becoming too complex. The basis for this can be found in *Templates*, *on page 31* and *PageType Concept*, *on page 39* and *Cartridges*, *on page 81*.

Changing existing or creating new PageTypes must always be done in separate cartridges.

In this example, the display of content pages (articles) will be changed. The starting point is the article display in the demo shop. See *Figure 34*

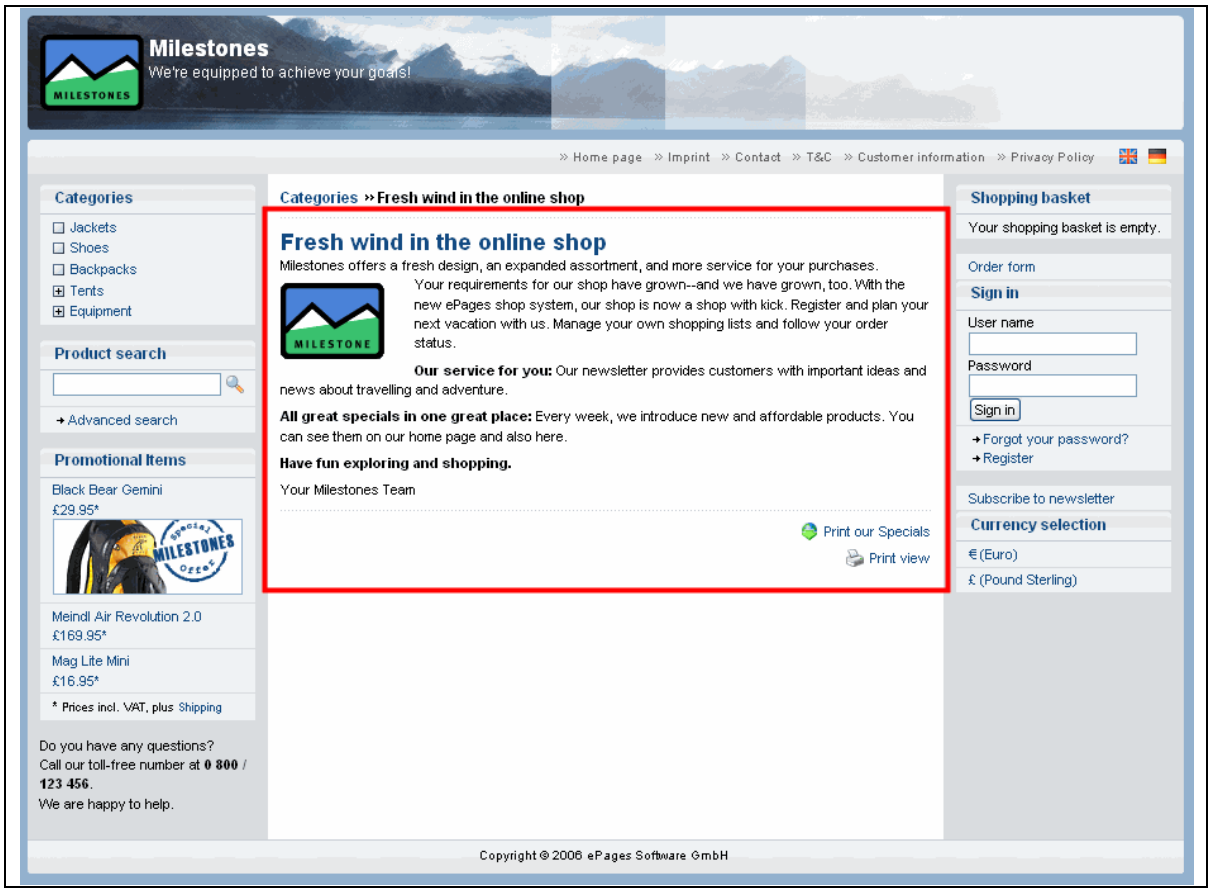

Figure 34: The home page as starting point for the example

The template for article display should be changed so that the content is included using two INCLUDES. One INCLUDE provides all data that are relevant for the text. The other connects the functions to print and display the attachment. To make the result more vivid, the functions should be placed above the article contents and the image beneath the text.

1. Identify template and PageType

   Determine which template will be used to display articles and in which PageType the assignment will occur. Using the debugging information, you will see that the content area is displayed via the following template

   ```
   %EPAGES_CARTRIDGES%/DE_EPAGES/Content/Templates/SF/SF-Article.Content.html
   ```

   Now, you must find out in which PageType this template is assigned. Template and PageType, in which the template is assigned are usually in a cartridge. The PageTypes for the *Content* cartridge are in the

/Database/XML/ directory in the *PageTypesSF.xml* file. You can see the original page in Code example 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>
…
```

Code example 117: *Content* definition in the *SF* PageType

The PageType *SF-Article* is based on the *SF* PageType and overwrites the *Content* area of the template listed there.
After the template and PageType are known, the following tasks should be performed:

- Change the template *SF-Article.Content.html*, so that the content is included using INCLUDES and provided to the overlay template
- Provide the INCLUDE template
- Extend the PageType

2. Create a cartridge

   Create a cartridge called *MyArticleDesign*. For more details see *Creating a Cartridge Structure, on page 83*.

3. Create the overlay template

   Create the following directory in the cartridge:

   ```
   /Data/Private/Templates/DE_EPAGES/Content/Templates/SF
   ```

   Create the *SF-Article.Content.html* file in this directory with the following content :

```
#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>
```

Code example 118: Change in the *Content* description in the template

Insert the source code for both includes in two new files:

- *SF-Article.Article_Content.html* and
- *SF-Article.Article_AttachmentSection.html*.

Since you are not overlaying original templates but rather introducing new templates, these new templates are created in the directory for cartridge-specific templates:

```
/Templates/SF
```

Use as the source code for the INCLUDE templates from the original template. Change the source code in *SF-Article.Article_Content.html* so that the image is shown beneath the text.

4. Extending the PageType

Through adding includes, the PageType *SF-Article* must also be changed. Both includes must be added as new areas in the PageType and the corresponding templates must be assigned.
These changes should **not** be made in the PageType definition in the *Content* cartridge. In the */Database/XML* directory of the *MyDesign* cartridge, create the *PageTypes.xml* file, see *Code example 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>
```

Code example 119: PageType definition for the *MyArticleDesign* cartridge

Using *Cartridge reference...*, indicate which cartridge the PageType definition belongs to.

In the *Class* tag and using *reference...*, create the reference to the object that the PageType is assigned to. You can find this in the original definition of the PageType.

The statement *reference="1"* determines that with the following commands, an existing PageType is extended. The PageType to be extended is indicated in the *Alias*.

The new areas *Article_Content* and *Article_AttachmentSection* are defined and assigned to the corresponding templates in the PageType element.

The correspondence between the area definition in the PageType and the usage in the template is illustrated here. The names of the areas are used as parameters of the INCLUDES.

5. Define dependencies

   The basic files have now been created. In order for the cartridge to function, one more file still needs to be created. The function of this file is explained in greater detail in *Import Files*, *on page 113,* . Look at this later so that you are not distracted from the task of creating a cartridge right now. Create the file in the corresponding location or copy it from the example source code. This file will be created in the same directory as *PageTypes.xml*:

   ```
   Dependencies.xml
   ```

   This file determines which functions the cartridge can or should use from any other specific cartridge.

   ```
   <?xml version="1.0" encoding="UTF-8"?>
   <epages>
     <Dependency Package="DE_EPAGES::Design" />
   </epages>
   ```

   Code example 120: Setting the dependencies on other cartridges

6. Install the cartridge

   Install the cartridge as described in *Installing - nmake*, *on page 85*.

After the process has completed, the articles can be seen in the shop. The template changes should be viewable in the display.

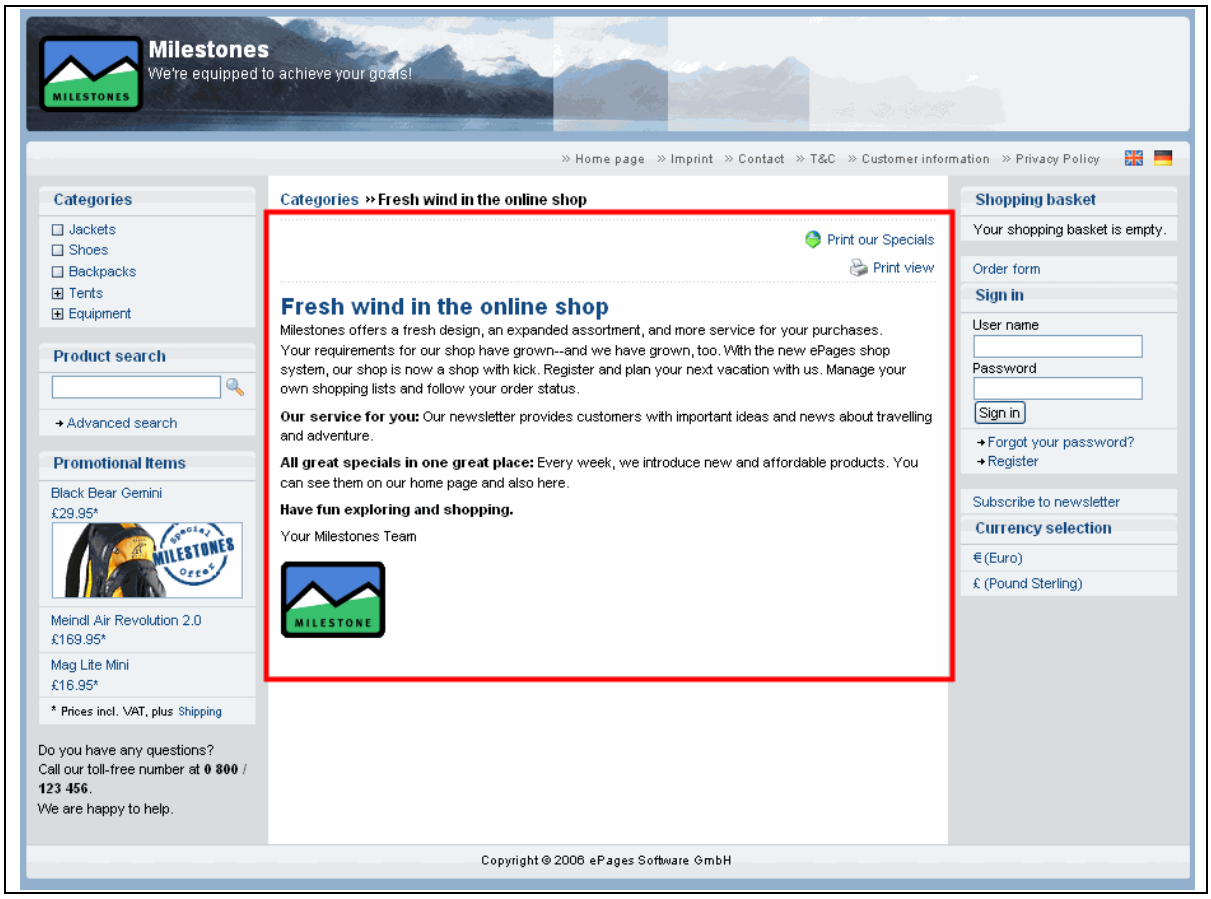

Figure 35: changed article view

Note: If the view is not visible immediately, delete the respective ctmpl files in the */STATIC* directory.

36. UE 7: New Batch Processing Commands in the MBO

Building on all the knowledge you have collected so far, you will now use a cartridge to extend the existing functions of the application in the merchant back office. The basis for this can be found in *Templates*, *on page 31* and *PageType Concept*, *on page 39* and *Cartridges*, *on page 81*.

In the MBO, there are many batch processing actions for tables that you can to process more than one entry at one time. See *Figure 36*.

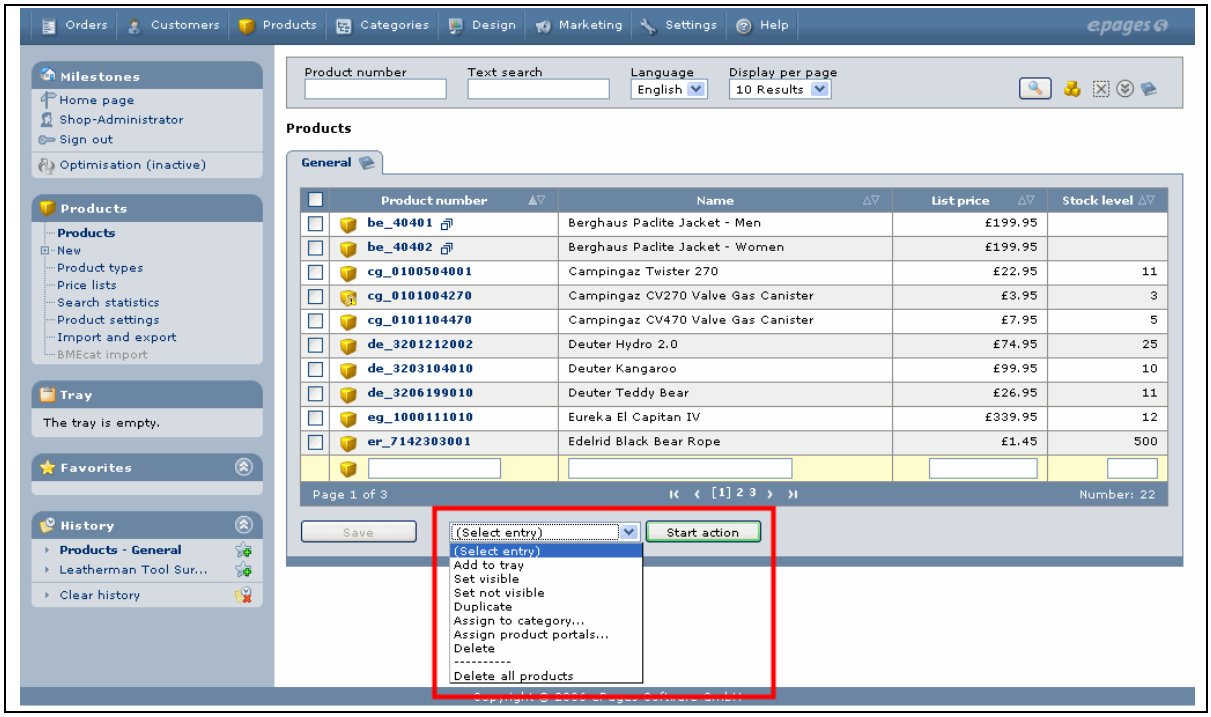

Figure 36: Batch processing commands for products

In this example, we want to add an action to the batch process for products. The action should mark the highlighted products in the product table as *New Products*. They will be given special emphasis in the shop. The new action is called *SetToNew*.

1. Identify PageType

   Determine which template will be used to display the batch processing actions and in which PageType the assignment will occur. This is the *MBO-Products* PageType. Add a new action to this page type.

2. Create a cartridge

   Create a cartridge called *AddBatchAction*. For more details see *Creating a Cartridge Structure*, *on page 83*.

3. Extending the PageType

   To add a new action, you must extend the *MBO-Products* PageType. In the

   ```
   /Database/XML
   ```

   directory, place the *PageTypesMBO.xml* file with the following source code:

```
<?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>
```

Code example 121: Extending an existing PageType

Using *reference="1"* in the *PageType* tag, indicate that you would like to extend the *MBO-Products* PageType. Here you find the *Menu* tag with the name *BatchActions*, in which different batch processes are already defined.

Now create a *Menu* tag with the same name, *BatchActions*. In this action, specify the new batch process *BatchAction-SetToNew* with the URLAction *SetToNew*. The *Position* attribute determines where the new action should be in the list of all batch process actions. At runtime, this menu entry is then processed and displayed together with the default entries, as the result in *Figure 37* shows.

4. Create a template

   In the PageType definition, you also assigned the new batch process to a template. This template is called *MBO-Products.BatchAction-SetToNew.HTML* and you put it in the */Templates/MBO* directory of your cartridge. This template describes how the batch process should be displayed on the page. In our example, only an additional entry is created for the drop-down menu for the batch process. The source code for the template appears as follows:

```
<option value="#URLAction">{SetNew}</option>
```

Code example 122: Template for a new batch process action

5. Create Dictionary files

   In the template source code, you have inserted a language tag for the name of the batch processing action called *{SetNew}* This keeps your language options flexible, see *Multiple Languages–Language Tags*, *on page 49*.
   Now for every language you want to display, you need to create the corresponding language file, in which the placeholder is replaced with an actual identifier in the respective language.
   For our example, we would like to display two languages, German and English. Create the two files *Dictionary.de.xml* and *Dictionary.en.xml*. For the basics about working with language files, see *Multiple Languages–Language Tags*, *on page 49*.
   Create the *Dictionary.de.xml* and *Dictionary.en.xml* files in the */Templates* directory of your cartridge and insert the following code:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<epages>
  <Language Language="de">
    <Translation Keyword="SetNew"  Value="Als neues Produkt anzeigen>
  </Language>
</epages>
```

Code example 123: German content for the *SetNew* language tag

or

```
<?xml version="1.0" encoding="iso-8859-1"?>
<epages>
  <Language Language="en">
    <Translation Keyword="SetNew"  Value="Display as new product>
  </Language>
</epages>
```

Code example 124: English content for the *SetNew* language tag

Depending on the display language selected, the batch processing action is displayed accordingly in the merchant back office.

6. Providing functions

   In order to identify a product as a *New Product*, a product attribute needs to be changed. In order to implement the function, a PERL module must be written and made available in the cartridge. Create the file containing the PERL code in the */UI* directory. In our example, this is the *ProductFolder.pm* file with the following source code:

```
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;
```

Code example 125: Code example for setting *IsNew* product attribute

In the function, the product attribute *IsNew* is set to *1* for all the selected products.

7. Determine Dependencies

   You must again decide which functions from which cartridge to use and whether it is even possible. In the previous example, the functions from *Design* were inherited. Certain more specific requirements are now necessary. You would like to work with your products, as well as have access to or be able to extend product-relevant functions. Now derive your cartridge from the *Products* cartridge. Define this dependency in the *Dependencies.xml* file in the */Database/XML directory.* You will find the necessary source code in *Code example 126*.

```
<?xml version="1.0" encoding="UTF-8"?>
<epages>
  <Dependency Package="DE_EPAGES::Product" />
</epages>
```

Code example 126: Setting the dependency of another cartridge

8. Register the action

   You have also defined a new action via the PageType. This action must also have been previously created in the database and made known. In addition, create the *ActionsProduct.xml* file in the */Database/XML* directory with the following content:

```
<?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>
```

   Code example 127: Registering the new action in the database

9. Assigning permissions

   For this action, you need to set who will have permission to execute it. Since the action is only available in the merchant back office, you need to assign permissions in the *Permissions.xml* file as follows:

```
<?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>
```

   Code example 128: Registering the permissions in the database

10. Install the cartridge

    Install the cartridge as described in *Installing - nmake*, *on page 85*. After the process has ended, open the merchant back office and then the product page. On the page with the product list, click the drop-down menu for the batch processing actions. You will see the new action listed there, see *Figure 37*.

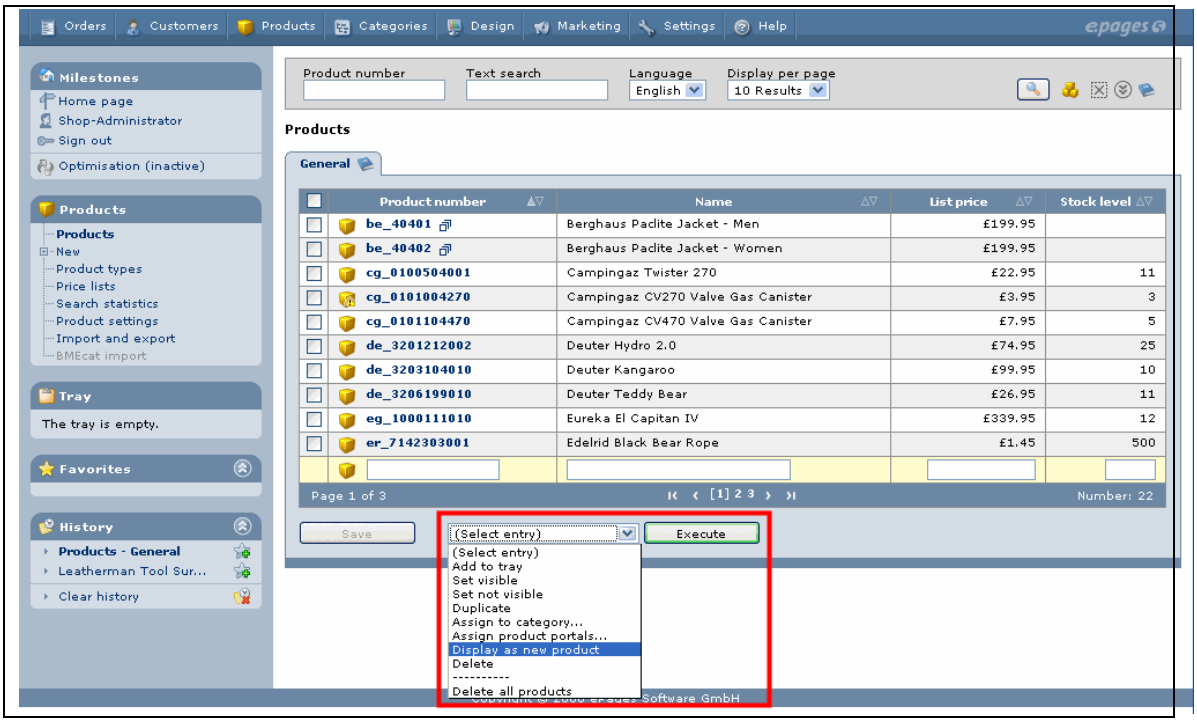

Figure 37: New batch process added

Since you have also created a language file for English, you will also see a correct entry for the new batch process action in the English display:

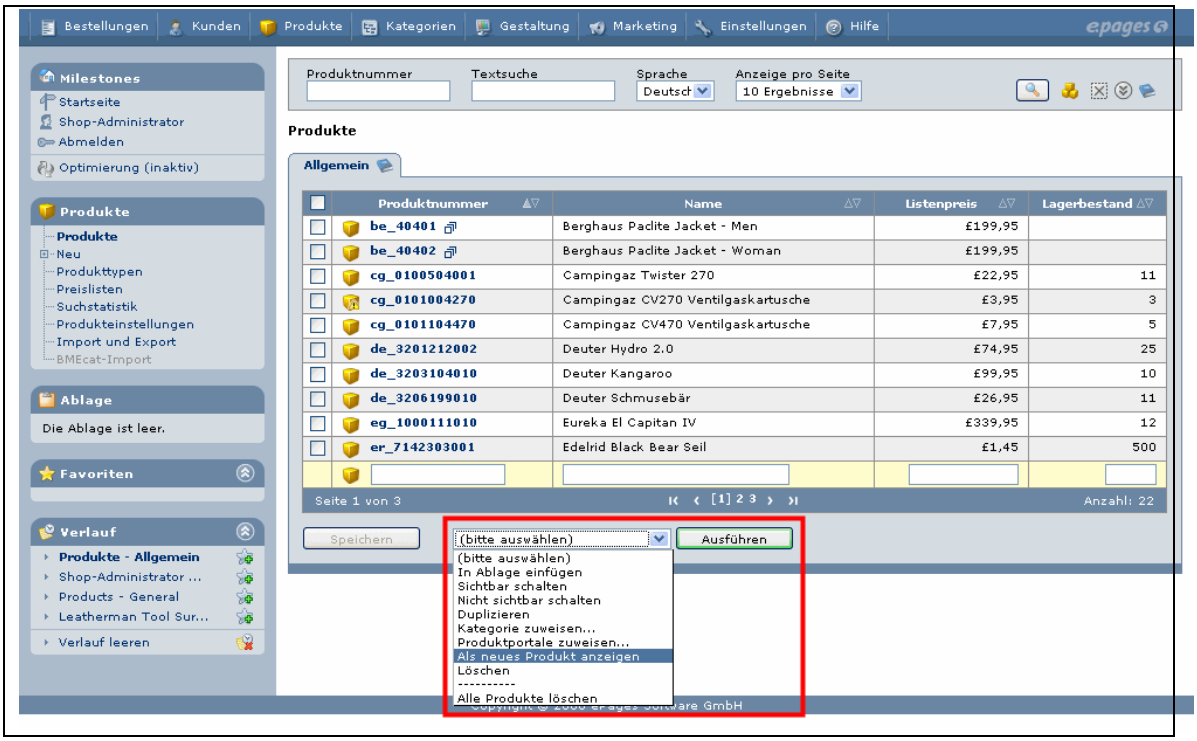

Figure 38: New batch process in the English display

Glossary

Application server
In this context, this is an instance of the ePages program. One computer can contain several application server instances. This is done by starting the ePages Windows service.
An application server can also be a server that provides a number of services or programs within a network. In our case, this is the server on which the ePages application runs.

Back office
All the administration Web pages used to help merchants manage their shops and administrators to manage how the shops are used.

Batch processing
An action that affects multiple elements at one time. This option is offered in tables where multiple instances of the same action can be combined into a multiple or batch process, for example, deleting multiple lines in a table at once.

Business unit
Unit consisting of a database and its assigned cartridges. The functions in the cartridges provide the functionality of the business unit.

Cartridge
Software module written in PERL. Every cartridge contains defined functions. Cartridges can be combined to create business units with different functions.

Data cache
Data storage cache in the application server. Data that are repeatedly needed are stored here. This prevents repeated database queries and significantly reduces response times.

Fallback
Mechanism for overlaying original files. Files for this are created in a specialized directory. These files overwrite the functions of the original files and are processed in their place.
A processing sequence for these files is defined in the system. This sequence first searches the specified directory for the required files. If they are not present, the original files are used.

Folder
A folder is a logical level in the object structure. Each object is able to save references to other objects and thereby becomes the folder for these objects. An example is orders assigned to a customer. In this case, the customer object is the folder for the objects for the individual orders.
No inheritance relationships according to class structure are required.

Language tag
HTML extension that supports multiple languages in the application. Language tags are inserted as placeholders in templates. They are replaced by the corresponding language-sensitive content when the template is processed. The language content is saved in and read from XML files.

Localization
Preparing the data and content of a Web page so it can be displayed in another language. For this, the corresponding languages, formats, and currencies must be provided in the system. Language tags are used to prepare the templates for displaying multiple languages. Some attributes can be localized, that is, you can collect different values for these attributes in various languages. They are then displayed in the respective language, for example, descriptions or names.

Shop types — Products the business administrator sells or leases to merchants or shop operators. Each shop type is offered with specific functions and usually also in various price classes. Merchants select from among these shop types to create their own shops.

Site — A) Object class in each database; *Shop* base classes
b) Name of the administration database in the default installation where the TBO and BBO are installed.

Site database — The ePages default installation database where the administration data for the application are saved. Technical and business administrators have access. Here, the databases which belong to the application and also certain basic settings, for example, for Web services and Web servers are managed. Business administrators also manage the shops and shop types here.

Store database — The ePages default installation database in which the data for the individual shops are saved. Merchants use this database to manage their shops and edit product, customer, and order data. This database provides the dynamic content for the store front.

Storefront — The "customer page" of a Web shop. All the Web pages that belong to a shop.

Style — All the data and instructions for displaying the Web pages. In addition to a .css file, information such as graphics, button icons, color design are a part of the style.

Template — HTML file for describing a Web page or certain defined areas on the Web page. In addition to the "normal" instructions for designing Web pages, ePages-specific extensions called *TLEs* and *language tags* are used.

TLE — Template Language Extension. ePages-specific language extensions that can be optionally integrated into HTML source code. They enable the integration and evaluation of dynamic content and, therefore, make programming Web pages more flexible.

Web Services — Web services allow communication between different applications. They offer the ability to link applications that run on various platforms and are implemented in different programming languages, and also to transfer data between various applications.
Web services use standard Internet protocols to transfer data such as HTTP, SMTP, and FTP. HTTP is used most often because a direct reaction to the query is possible, while SMTP and FTP only allow asynchronous data transfer.

Web shop — An Internet application that contains all the functions necessary for selling products or services. In ePages 5, the shop is created based upon a shop type defined by the ePages business administrator. The merchant generates a shop online, modifies the structure and design, and enters his products and services into the system to open his Internet sales channel.

Index

U

V

W

X