Creating a new theme

Table of Contents

How to create a design theme

Warning: ***DO NOT DUPLICATE AN EXISTING THEME****. Yes, this would make it slightly easier to create your new theme, but it will also make your new theme horribly incompatible with Magento updates. Many designers (including some major Magento design companies) did this for a long time, and left the Magento world in a bit of a mess when 1.4 was released. Instead, use the "least impact" approach, outlined in this article to maximize compatibility with updates.*

The first step will be creating the folders for the new theme. We'll call your new design new_theme. Create the following folders:

New folders:

1. /app/design/frontend/default/new_theme/ - our new theme
2. /app/design/frontend/default/new_theme/layout
3. /app/design/frontend/default/new_theme/templates
4. /skin/frontend/default/new_theme/ - our new skins folder
5. /skin/frontend/default/new_theme/css/
6. /skin/frontend/default/new_theme/images/

Also, create the following new files:

New files:

1. /app/design/frontend/default/new_theme/layout/local.xml
2. /skin/frontend/default/new_theme/css/local.css

Finally, add some boiler plate to local.xml, so that local.css will get included:

local.xml:

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

           <default>
               <!-- Remove callouts and rarely used stuff -->
               <remove name="right.poll"/>
               <remove name="right.permanent.callout"/>

               <remove name="left.permanent.callout"/>
                     <!-- add the local stylesheet -->

                     <reference name="head">
                         <action method="addCss">
                        <stylesheet>css/local.css</stylesheet>
                   </action>
                     </reference>
                 </default>

            </layout>
```

A well coded theme should have the following traits:

1. A single layout file, named local.xml, where all layout updates are placed.
2. no layout files with the same name as any layout file in the base theme
3. no css files with the same name as any css file in the default skin
4. no .phtml template files, except for those that were modified to support the new theme. (Usually this number will be very small.)

How to setup a new theme in administration area

After creating the folders we need to switch the theme from the administration area.

Step 1. Go to System > Configuration > Design via the top navigation bar
Step 2. Under “Themes” and in the “Default” text field, type: new_theme
Step 3. Press the “Save config” button in the upper right corner

Adding custom stylesheets and js libraries (part I)

Any additional external stylesheet files (css) or javascript/ajax libraries (js) that we want to include in our new project must be also copied to the proper folders.

Stylesheets Copy the files to /skin/frontend/default/new_theme/css/ folder. To link these files in, you can either modify local.xml to add the new file, or add an import in local.css, like this:

```
@import url('my_new.css');
```

Javascript / AJAX libraries Create a new folder under /js/ named new_theme and copy your files to it. If you are using javascript libraries then this is also a good place to put the library files.

The Javascript files can be added to your theme by adding the following to local.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
       <layout>
           ...
           <default>
           ...
               <reference name="head">
                   ...
                   <action method="addJs">
                       <script>varien/js.js</script>
                   </action>
                       ...
                     </reference>
                ...
                  </default>
                  ...
            </layout>
```

Using XML to change layout

With the use of XML we can change almost every aspect of our new_template. For example we can set an alternate column layout for particular pages, change META information, page encoding, block types used on each page etc. To accomplish this, you will simply add various sections to your local.xml file. A good place to look at to understand this is:

```
app/design/frontend/base/default/layout/page.xml
```

In 1.3.x and earlier, this is located at::

```
app/design/frontend/default/default/layout/page.xml
```

Changing META section

(Isn't this section horribly obsolete?) The main file used to control values for META tags and other miscelaneous details is config.xml which is located in the /app/design/frontend/new_template/default/etc/ folder.

Below is a short description of every META tag and possible values:

```
<title>Magento Commerce</title>
```

This is the name of our ecommerce site. This text will appear in the browser's title bar or the browser tab for the site. This text is also highly important for search engine keyword optimization.

```
<media_type>text/html</media_type>
```

This is default page header encoding so we should leave this as is.

```
<charset>utf8</charset>
```

UTF8 is a variable-length character encoding for Unicode. It is able to represent any character in the Unicode standard, yet the initial encoding of byte codes and character assignments for UTF-8 is backwards compatible with the ASCII table. For these reasons, it is steadily becoming the preferred encoding for e-mail, web pages, and other places where characters are stored or streamed.

Of course we can change it to any other encoding (ex. ISO-8859-1 or ISO-8859-2) but there is no need as long as we're saving our files with proper UTF8 encoding.

More information about UTF8 is in the Wiki: http://en.wikipedia.org/wiki/UTF-8

```
<description>Default Description</description>
```

The description tag controls the description meta tag and allows us to enter a short description about our site. It's often a way to get a nice description of your page to show up in the search results if your page does rank highly in a search engine. Your best bet is to write a succinct sentence or two that uses the keyword phrases that sum up the page content.

```
<keywords>Magento, Varien, E-commerce</keywords>
```

The keywords tag controls the keyword meta tag and is the place to put the most important words that refer to the site content. Best practices suggest to enter no more than 500 characters in no more than 20 words for best results.

```
<robots>*</robots>
```

The robots tag controls the robots meta directive and is a simple mechanism to indicate to visiting web bots and search engine spiders if a page should be indexed, or links on the page should be followed. The content of the Robots meta tag contains directives separated by commas. The currently defined directives are index, noindex, follow and nofollow. The two index directives specify if an indexing robot should index the page or not. The two follow directives specify if a robot is to follow links on the page or not. The defaults are index and follow. The values all and none set all directives on or off: all=index,follow and none=noindex,nofollow We can simply override Magento's default directive by placing one of the four following lines here, however it's not recommended. The options are:

- index,follow
- noindex,follow
- index,nofollow
- noindex,nofollow

The file config.xml also contains two additional tags, not connected with any meta tags but used as a default settings for every page in our shop.

```
<logo_src>images/logo.gif</logo_src>
```

The logo_src tag sets up a reference to the logo file we wish to use on our site. The image logo.gif is located in the folder /skin/frontend/new_template/default/images/ so if we want to change it we must copy a new logo file to that folder. We can also create a new folder inside the images folder (ex. new_images) and put all our new files used by new template into it and change this tag appropriately. The easiest way to customize the logo it to simply overwrite the default logo.gif file with a new logo.gif file.

```
    <logo_alt>Magento Commerce</logo_alt>
```

The logo_alt tag defines the alt attribute for our logo image and is mostly used by screen readers or browsers with images disabled. If one of our customers uses a screen reader or has images disabled he will see the alt text instead of image.

Understanding layout XML files

Introduction

Using xml instead of other methods (JSON, .ini files, include / require functions) allows us to change many aspects on our page without manually changing the .phtml files. This chapter refers to our default theme so after changing the theme (as we have done above) the paths will also change.

Layout / page structure

The core Layout is defined by page.xml which is located in

```
/app/design/frontend/base/default/layout/page.xml
```

There are two large tasks layout carries out.

First it defines the visual layout for your store. By default Magento uses a 3-column layout, so it defines use of 3columns.phtml (Located in your template/page/ folder):

```
<block type="page/html" name="root" output="toHtml" template="page/3columns.phtml">
```

If you wanted to change your store a 2-column layout, for instance, you would reference this section in local.xml, and use an action to change to the .phtml you'd like to use (in this case, 2columns-left.phtml or 2columns-right.phtml).

```
1. <reference name="root">
2.       <action method="setTemplate">
3.             <template>page/2columns-right.phtml</template>
4.             <!-- Mark root page block that template is applied -->
5.             <action method="setIsHandle">
6.                   <applied>1</applied>
7.             </action>
8.       </action>
```

You could also add a new custom layout:

```
<new_layout translate="label">
      <label>New Layout</label>
            <reference name="root">
                  <action method="setTemplate">
                        <template>page/new-layout.phtml</template>
                  </action>
            <!-- Mark root page block that template is applied -->
                  <action method="setIsHandle">
                        <applied>1</applied>
                        </action>
                  </reference>
      </new_layout>
```

More information about the action tag and associated methods is located in section d of this chapter.

Secondly it creates “block containers” filled with application data for output to your .phtml template files. First, if take a look at your standard 3column.phtml file you’ll see it calls the method (function) getChildHtml() a number of times:

(excerpt from 3columns.phtml – Starting at line 56):

```
      <?=$this->getChildHtml('header')?>
      </div><!-- [end] header --><!-- [start] middle →
      <div class="main-container">
      <div id="main" class="col-3-layout">
      <?=$this->getChildHtml('breadcrumbs')?>
      <!-- [start] left -->
      <div class="col-left side-col">
      <?=$this->getChildHtml('store')?>
      <?=$this->getChildHtml('left')?>
      </div>
      <!-- [end] left -->
```

Each of these references point towards the “block containers” defined in your page.xml and subsequent Module .xml files. You’ll notice a container for the “left” column, for the “right”, for “content”, and other standard areas. It serves up output for your .phtml template files to use. Take a look at it:

```
/app/design/frontend/default/default/layout/page.xml
```

```
1. <block type="page/html_breadcrumbs" name="breadcrumbs" as="breadcrumbs"/>
2. <block type="core/text_list" name="left" as="left"/>
3. <block type="core/messages" name="global_messages" as="global_messages"/>
4. <block type="core/messages" name="messages" as="messages"/>
5. <block type="core/text_list" name="content" as="content"/>
6. <block type="core/text_list" name="right" as="right">
```

So basically page.xml creates Data Blocks, your .phtml Outputs that data where you want it. Hence, the names for left, right and so forth in your .phtml.

To add, remove, or modify blocks in your theme, use your local.xml file, not a copy of page.xml. The reference tag allows you to define what part of the theme you are working with, then you can declare additional blocks, or use actions to modify or remove blocks.

You're aware now that page.xml contains a handle called "<default>". The XML commands nested within the <default> layout sets up the default layout for the whole site. The subsequent handles(as listed in the top of page.xml), simply updates the default layout with the according layout for the page.

```
1. <layoutUpdate>
2. <reference name="top.menu">
3. <block type="catalog/navigation" name="catalog.topnav">
4. <action method="setTemplate">
5. <template>catalog/navigation/top.phtml</template>
6. </action>
7.       </block>
8. </reference>
9.
10.      <reference name="right">
11.            <block type="catalog/product_compare_sidebar"  name="catalog.compare.sidebar">
12.            <action method="setTemplate">
13.                  <template>catalog/product/compare/sidebar.phtml</template>
14.            </action>
15.            </block>
16.      </reference>
```

If you read the code and think about whats going on it makes sense. <layoutUpdate> is UPDATING your code with new blocks. How does it know where to put the new blocks? Well remember in your default you defined "block containers" for left, right, etc. So when it says

```
<reference name = "right">
```

it's telling Magento to insert the following block into the RIGHT column when you get into "Catalog" view. (Remember, its located at layout/catalog/) so it appears once you've entered the catalog section of the shop. It also defines a TEMPLATE for the new block to use, so for the example above the compare box has its own template located at catalog/product/compare/sidebar.phtml (in your template folder).

These same handles can be used in local.xml to restrict the scope of a particular update.

For instance, let's say you want a wishlist everywhere in your store, but you don't want it in the customer account pages. You would look for the handle used on account pages (it is in customer.xml) and add that handle to local.xml. Then remove the wishlist so it does not load in the account pages.

The example below removes "Login" and adds "Register" instead, but only if the customer is not already logged in:

```
1. <customer_logged_out>
2.       <reference name="top.links">
3.             <action method="removeLinkByUrl" module="catalog">
4.                   <url helper="customer/getLoginUrl" />
5.             </action>
6.             <action method="addLink" translate="label title" module="customer">
7.                   <label>Register</label>
8.                   <url helper="customer/getRegisterUrl"/>
9.                   <title>Register</title>
10.                        <prepare/>
11.                        <urlParams/>
12.                        <position>100</position>
13.                  </action>
14.            </reference>
15.      </customer_logged_out>
```

Reference name values/attributes:

As we've seen, the references can use different attributes for displaying blocks on our page. Possible values are:

- root – we will change it mostly to set up another .phtml file as a root template ex. (1column.phtml , 2columns-left.phtml , 2columns-right.phtml etc.)
- head – as this reffers to our <HEAD> section on page, we will use this reference name to reflect our changes in this section
- top.menu – defines our content for top menu section
- left – defines our content for left column
- right – as above but for right column
- content – defines blocks placed in main content of our page
- before_body_end – is used to add a block before end of our page so before </BODY>

There are more reference names that we could use but they are more page-specific than for global use for example:

customer_account_navigation
customer_account_dashboard

are used on our clients account page

product.info.additional – sets up additional block for placing related items, shipping estimator etc.

Action methods:

Action methods allow us to change many theme settings without appending manual changes to our .phtml files. The method listed in the method attribute refers to a method in the Model that is associated with the particular block in question. The most important methods are described below.

setTemplate

Action method setTemplate allows us to change the default .phtml file used in particular block. For example by navigating to app/design/frontend/default/default/layout/catalog/product/view.xml we can see the reference:

```
<reference name="root">
      <action method="setTemplate">
            <template>page/2columns-right.phtml</template>
      </action>
</reference>
```

and by using another <template> value we are allowed to change default .phtml file used on our products page. Possible values are:

- 1column.phtml
- 2columns-left.phtml
- 2columns-right.phtml
- 3columns.phtml
- one-column.phtml
- dashboard.phtml

As we see in app/design/frontend/default/default/layout/checkout/cart.xml , there also additional 2 values for empty and non-empty cart

```
<action method="setCartTemplate">
      <value>checkout/cart.phtml</value>
</action>

<action method="setEmptyTemplate">
      <value>checkout/cart/noItems.phtml</value>
</action>

```

```
9. <action method="chooseTemplate"/>
```

The method chooseTemplate is used to set a template (setCartTemplate / setEmptyTemplate) depending on quantity of items in our cart. If we have more than 0 than the

```
1. <action method="setCartTemplate">
2.       <value>checkout/cart.phtml</value>
3. </action>
```

is used. If we have no items in cart then the following will be used.

```
1. <action method="setEmptyTemplate">
2.       <value>checkout/cart/noItems.phtml</value>
3. </action>
```

The function provided by the Model is shown below:

```
1. public function chooseTemplate()
2.
3. {
4.       if ($this->getQuote()->hasItems()) {
5.
6.             $this->setTemplate($this->getCartTemplate());
7.
8.       } else {
9.
10.                  $this->setTemplate($this->getEmptyTemplate());
11.            }
12.      }
```

That should clarify how we can use this particular switch. Depending on our needs we can write custom functions in our blocks and than assign a template depending on parameters returned by a function.

addCss

This method allows us to add an additional CSS file to our page on per-page basis or globally for our template. If we use a reference name “head” and action method addCss by using

```
1. <reference name="head">
2.       <action method="addCss"><link>style.css</link></action>
3. </reference>
```

then our page will have an additional line of code to attach the CSS file, for example:

```
1. <link rel="stylesheet" type="text/css" media="all"
   href="http://www.ourstore.com/skin/frontend/default/default/css/style.css" ></link>
```

As we can see, the <link> path refers to the /skin/frontend/default/default/css/ folder.

addCssIe

This method is very similar to the above but it will output a css file when a User-Agent (browser) is Internet Explorer or Maxthon. So using this method we can attach a specific css file for those browsers. This is very helpful if our page will requires changes in css dependant on a specific browser.

Usage:

```
1. <reference name="head">
2.       <action method="addCssIe"><link>style.css</link></action>
3. </reference>
```

Output in page source:

```
1. <!--[if IE]>
2. <link rel="stylesheet" type="text/css" media="all"
   href="http://www.ourstore.com/skin/frontend/default/default/css/style.css" ></link>
3. <![endif]-->
```

addJs

The below method allows us to attach a .js script in the same way as we attached a CSS file. The script path refers to the /js/test/ folder but you can specify any subdirectory of /js/

Usage:

```
1. <reference name="head">
2.       <action method="addJs">test/script.js</action>
3. </reference>
```

It will add a script to our page with src attribute of

```
1. <script src="http://www.ourstore.com/js/test/script.js" />
```

addJsIe – adding a .js file to head section of page and using it if User Agent (browser) is Internet Explorer.

If we can add different css files depending on User-Agent we can do the same with our .js files. Than we can use different scripts for our IE/Maxthone users and another for other browsers.

Usage:

```
1. <reference name="head">
2.       <action method="addJsIe">our/script.js</action>
3. </reference>
```

It will add a script to our page with src attribute of

```
1. <script src="http://www.ourstore.com/js/our/script.js" />
```

but also inside IE comments

```
1. <!--[if IE]><![endif]-->
```

setContentType

This method can override default headers sent by our page to the browser. So we can set a text/xml instead of text/html (or another as we wish).

Usage:

```
1. <reference name="head">
2. <action method="setContentType"><content>text/xml</content></action>
3. </reference>
```

setCharset

allows us to override default page encoding on per-page basis or globally. As long as we are saving our files with proper encoding this will not be necessary.

Usage:

```
1. <reference name="head">
2. <action method="setCharset"><charset>ISO-8859-2</charset></action>
3. </reference>
```

So in this case our page will have Central European encoding instead of default UTF-8

addLink

addLink methods can be used to set a setting to which we can refer in our .phtml template files also without manually changing the .phtml files.

Example usage :

Warning, this example requires account.xml to be duplicated, which can break things after updates. Someone should rewrite this to use local.xml instead. By adding a block which was found in app/design/frontend/default/default/layout/customer/account.xml into our <reference name=”content”> in app/design/frontend/default/default/layout/checkout/cart.xml we can allow the customer to skip to the account information directly from the cart.

```
<block type="customer/account_navigation" name="customer_account_navigation" before="-">
	<action method="setTemplate">
		<template>customer/account/navigation.phtml</template>
	</action>

	<action method="addLink" translate="label">
		<name>account</name>
		<path>customer/account/</path>
		<label>Account Dashboard</label>
	</action>

	<action method="addLink" translate="label">
		<name>address_book</name>
		<path>customer/address/</path>
		<label>Address Book</label>
	</action>

	<action method="addLink" translate="label">
		<name>account_edit</name>
		<path>customer/account/edit/</path>
		<label>Account Information</label>
</block>
```

The cart.xml file should look like below

```
<layoutUpdate>
    <reference name="root">
        <action method="setTemplate">
            <template>page/1column.phtml</template>
            </action>
    </reference>

    <reference name="content">
        <block type="customer/account_navigation" name="customer_account_navigation" before="-">
                <action method="setTemplate">
                    <template>customer/account/navigation.phtml</template>
                </action>

                <action method="addLink" translate="label">
                    <name>account</name>
                    <path>customer/account/</path>
                    <label>Account Dashboard</label>
                </action>

                <action method="addLink" translate="label">
                    <name>address_book</name>
                    <path>customer/address/</path>
                    <label>Address Book</label>
                </action>

                <action method="addLink" translate="label">
                    <name>account_edit</name>
                    <path>customer/account/edit/</path>
                    <label>Account Information</label>
                    <base>{{baseSecureUrl}}</base>
                </action>
            </block>

            <block type="checkout/cart" name="checkout.cart">
                <action method="setCartTemplate">
                    <value>checkout/cart.phtml</value>
                </action>

                <action method="setEmptyTemplate">
                    <value>checkout/cart/noItems.phtml</value>
                </action>

                <action method="chooseTemplate"/>

                <block type="checkout/cart_coupon" name="checkout.cart.coupon" as="coupon">
```

```
                    <action method="setTemplate">
                        <template>checkout/cart/coupon.phtml</template>
                        </action>
                </block>

                <block type="checkout/cart_shipping" name="checkout.cart.shipping" as="shipping">
                    <action method="setTemplate">
                        <template>checkout/cart/shipping.phtml</template>
                        </action>
                </block>

                <block type="checkout/cart_crosssell" name="checkout.cart.crosssell" as="crosssell">
                    <action method="setTemplate">
                        <template>checkout/cart/crosssell.phtml</template>
                        </action>
                </block>
            </block>
        </reference>
    </layoutUpdate>
```

Of course according to the previous references we can also change

```
<action method="setTemplate">
      <template>page/1column.phtml</template>
</action>
```

in the above code to suit our needs. I've used for example

```
<action method="setTemplate">
      <template>page/one-column.phtml</template>
</action>
```

to show only our cart without other blocks.

Layout blocks

As we've seen before, most of our xml files have a <block> tag. It defines a type of block, its name and alias "as" so we can refer to it on our page. Basic block structure looks like this:

```
1. <block type="catalog/product_view_super_config" name="product.info.config" as="super_config">
2.       <action method="setTemplate">
3.             <template>catalog/product/view/super/config.phtml</template>
4.       </action>
5. </block>
```

type="catalog/product_view_super_config" - defines the type of our block on page. This example would refer to the file /app/code/core/Mage/Catalog/Block/Product/View/Super/Config.php which defines the class Mage_Catalog_Block_Product_View_Super_Config name="product.info.config" - defines a name for our block and should be unique as="super_config" - defines a shortname for our block which can we use with getChild function on particular page

Blocks used in our XML files can change or override most every aspect of our design. We can use them to simply change used phtml files on per-page basis, to add additional scripts, stylesheets to our page , to move particular sections of page without needing to change phtml files.

Understanding .phtml files

Introduction

Phtml files are template files that handle the output to browser. They are nothing more than html files with nested php tags. We use them to style our page and the look of our site.

Changing .phtml files requires basic knowledge about XHTML, CSS and understanding how to use PHP functions on a page.

IMPORTANT: Before changing a .phtml file, it has to be copied from the base (or default) theme, into the new theme. When Magento finds one of these files in your new theme, it will ignore the .phtml file from the base theme, so it is important to ONLY copy over the files that you absolutely need to modify. This will minimize errors when updating Magento. The additional effort required to individually copy the 4-5 files you eventually modify will more than make up for itself the first time Magento needs to be updated.

Let's have look at header.phtml placed in templates/page/html.

```
1. <div class="header-top-container">
2.
3.     <div class="header-top">
4.
5.         <h1 id="logo">
6.             <a href="<?=$this->getUrl('')?>">
7.                 <img src="<?=$this->getLogoSrc()?>" alt="<?=$this->getLogoAlt()?>"/>
8.             </a>
9.          </h1>
10.             <p class="no-show"><a href="#main"><strong><?=__('Skip to main content')?> &raquo;</strong></a></p>
11.             <div class="quick-access">
12.                 <div class="account-access">
13.                     <strong><?=$this->getWelcome()?></strong> <?=$this->getChildHtml('topLeftLinks')?>
14.                 </div>
15.
16.                 <div class="shop-access">
17.                     <?=$this->getChildHtml('topRightLinks')?>
18.                 </div>
19.
20.             </div>
21.
22.          </div>
23.
24.      </div>
25.      <?=$this->getChildHtml('topMenu')?>
```

In this file we see basic XHTML tags, usage of CSS classes but most important – Magento functions used to get layout XML blocks and render it in our phtml file.

Mostly in our template we'll see <? ?> tags used for functions calls. Basing on above example: <?=$this→getUrl('')?> – used without parameters will return base path of our store <?=$this→getLogoSrc()?> – will render a logo image based on path used in etc/config.xml <logo_src>images/logo.gif</logo_src>

If we'd like to change our logo we can do this in two ways. First possibility is to change <logo_src> path to anything else. Second possibility is to hardcode the path directly in phtml file so

```
1. <img src="<?=$this->getLogoSrc()?>" alt="<?=$this->getLogoAlt()?>"/>
```

but this resolution isn't recommended since we should use core functions and learn their usage.

<?=$this→getLogoAlt()?> – this function will allow us to change the alt tag for our logo so if it will be unavailable , the alt tag will appear. Any changes we can also append by changing <logo_alt>Magentso Commerce</logo_alt> or setting it direclty in phtml file up.

<?=__(Skip to main content)?> – all tags that look like this will be used to dynamically translate page content to other languages. We should understand it as <?=__('English text to translate')?>

<?=$this→getChildHtml('topLeftLinks')?> – The getChildHtml() function is the most important function used in our template. It calls particular block defined in XML file and renders it as HTML , then outputs it to the browser. We can call blocks from everywhere and from corresponding XML files.

To use getChildHtml('topLeftLinks') we must have defined first the child "as" so take a closer look at page.xml (layout/ folder). Here's what you should see:

```
1. <block type="page/html_toplinks" name="top.left.links" as="topLeftLinks"/>
```

As we see, getChildHtml('topLeftLinks') uses its alias "as" and calls it from the XML. The getChildHtml() function only allows Magento to call a block if that block was defined in the corresponding XML file.

We can also override this mechanism by using another function call:

```
<?=$this→getLayout()→getBlock('top.left.links')→toHtml()?>
```

This structure will call the top.search block (based on its name, not its alias “as”) from anywhere in any of our templates so we do not need to define it everywhere in our XML files. Remember to use the “name” attribute instead of the “as” attribute with this workaround.

We must be aware that every phtml file and every function will always refer to the corresponding XML file or files. We can identify used phtml files simply by searching for the following:

1. `<action method="setTemplate"><template>wishlist/sidebar.phtml</template></action>`

which tells us which phtml file will be used.

Folder structure

Every View used in our application has separate folders and subfolders to store template files. Let's have a look at the folder structure:

- **callouts** – folder where are files for our callouts and ads are placed
- **catalog** – folder to store files used on our category, layered navigation, product, comparision pages
- **catalogsearch** – here we find files that are used to skin our search engine and result pages
- **checkout** – all the pages utilised during checkout and shopping in our shop. Here we'll also find the shopping cart templates and blocks for cross-selling and coupons.
- **cms** – folder for static pages templates.
- **core** – folder containing store-switching templates (not yet active)
- **customer** – all customer pages (ex. account dashboard, address forms, orders list and reviews)
- **datafeed** – folder to store our csv, txt, xml files, used for comparision engines and other external applications
- **directory** – here we find currency switcher for our shop
- **email** – here we'll find all pages that require email transport so for example order, password recovery, newsletter signup
- **install** – template files for Magento's installer
- **newsletter** – subscribe.phtml placed in this folder will allow us to change the look of newsletter signup box on our page
- **page** – the most important folder in which we'll find all main files used to style our site. More about it will be described in following subchapter.
- **payment** – templates used to style our payment forms (ex. CC payment.)
- **poll** – 2 files to change the look of our poll depending on state (didn't vote yet / show result)
- **rating** – rating block used on our products pages
- **review** – all files used to render the blocks used by reviews
- **sales** – pages for order details, invoices, recent orders
- **searchlucene** – result output for Zend_Lucene controller used in Magento
- **tag** – product tags templates are stored here
- **wishlist** – template files to handle the output of our wishlist actions.

Appending basic changes to our templates

Every file used to skin the basic look of our template is placed in template/page folder. Here we'll see

1column.phtml 2columns-left.phtml 2columns-right.phtml 3columns.phtml one-column.phtml dashboard.phtml

These are essentially the HTML skeleton files for our pages. By changing those files we can set up a new look of the page structure.

There is also an html subfolder placed under template/page in which we can change footer, header and links blocks of our template.

Every one of them uses simple function calls (ex. getChildHtml()) so we can also identify the block by searching in layout XML files .

For example when we'll see

```
1. <reference name="root">
2.       <action method="setTemplate">
3.             <template>page/2columns-right.phtml</template>
4.       </action>
5. </reference>
```

this tells us that the page will use 2columns-right.phtml as a skeleton for our page.

Calling layout blocks using Magento functions

As it was described above, there are two ways of calling blocks .

```
<block type="page/html_toplinks" name="top.left.links" as="topLeftLinks"/>
```

<?=$this→getChildHtml('as')?> - by using the block alias “as” from the XML file we can display a block on our page providing that it was defined already in the corresponding XML file

<?=$this→getLayout()→getBlock('name')→toHtml()?> - by using the block name from the XML file, we can call any block whether or not it was already defined in corresponding XML file

e. Adding custom CSS and JS files to a layout (part II)

There are 2 ways of adding custom js and css files to our template. The recommended way is by extending the head section in the default XML file. But you also have the ability to add the files directly in the particular root template file.

```
1. <script type="text/javascript" src="<?=$this->getJsUrl()?>varien/js.js" ></script>
2. <script type="text/javascript" src="<?=$this->getJsUrl()?>varien/form.js" ></script>
3. <script type="text/javascript" src="<?=$this->getJsUrl()?>varien/menu.js" ></script>
```

As we see here , we can also use getJsUrl method which adds a scripts path to our src attribute. This should output http://yoursite.com/js/ so the source will look like the following:

```
1. <script type="text/javascript" src="http://yoursite.com/js/varien/js.js" ></script>
2. <script type="text/javascript" src="http://yoursite.com/js/varien/form.js" ></script>
3. <script type="text/javascript" src="http://yoursite.com/js/varien/menu.js" ></script>
```

Adding a css file isn't harder than adding a js file. We do this also by using a function that outputs a path to our skins folder.

```
1. <link rel="stylesheet" href="<?=$this->getSkinUrl('css/styles.css')?>" type="text/css" media="all"/>
```

And the output will be

```
1. <link rel="stylesheet" href="http://yoursite.com/skin/frontend/default/default/css/styles.css" type="text/css"
   media="all"/>
```

so getSkinUrl() sets actual path to our skins folder : http://yoursite.com/skin/frontend/default/default/

Opening .phtml files in Dreamweaver

By default, Dreamweaver cannot read PHTML files. You can add the file type to the “Open in Code View” section of the preferences if you wish to have fast access, however you cannot view the file in design view if you do that. So if you use Dreamweaver (versions 4, MX, MX2004, 8, or 9, aka CS3) to design your sites, and you wish to open Magento’s Template files (they have .phtml extensions) in Dreamweaver, you can follow these steps to add support for .phtml and make Dreamweaver render PHP code (with coloring, hinting, et al) as well as allow you to see the design in code view if desired. Below are three steps to follow.*

*IMPORTANT NOTES: This guide is for Dreamweaver on Windows (XP or Vista) or Mac OS X. Note: I have excluded version numbers from the file locations shown, and if you are using a version older than Dreamweaver 9 (CS3) replace “Adobe” with “Macromedia” in the file locations shown. Some spaces have also been removed to keep the refer*ences on one line.

* Dreamweaver 4 users: if you are using the archaic Dreamweaver 4, you only need to follow step one. However, it’s highly recommended that you just upgrade to version 8 or newer for superb CSS and Web Standards support.

Step One: Add .phtml to the extension.txt configuration file

Open the following extension configuration file in a notepad and change the lines as specified below:

XP, Vista: C:Program FilesAdobeDreamweaverConfigurationExtensions.txt

Mac OS X: Applications > Adobe Dreamweaver CS3 > configuration > Extensions.txt In the first line add PHTML like so:

HTM,HTML,SHTM,SHTML, ... ,TXT,PHP,PHP3,PHP4,PHP5,PHTML,JSP,WML,TPL, ... ,MASTER:All Documents

In the PHP Files line add PHTML like so:

PHP,PHP3,PHP4,PHP5,PHTML,TPL,PHTML:PHP Files

Step Two: Add .phtml to extension.txt in your Application Data

This file is pretty much exactly like the extensions.txt file located in Dreamweaver’s configuration folder, except it is in your users Application Data folder (AppData folder for Vista users). Just as in Step One, find the file and change the lines as specified below.

XP: C:Documents and Settings[user]Application DataAdobeDreamweaverConfigurationextensions.txt

Vista: C:Users[user]AppDataRoamingAdobeDreamweaverConfigurationExtensions.txt

Mac OS X: Users > Library > Application Support > Adobe > Dreamweaver 9 > Configuration > Extensions.txt

In the first line add PHTML like so:

HTM,HTML,SHTM,SHTML, ... ,TXT,PHP,PHP3,PHP4,PHP5,PHTML,JSP,WML,TPL, ... ,MASTER:All Documents

In the PHP Files line add PHTML like so:

PHP,PHP3,PHP4,PHP5,PHTML,TPL,PHTML:PHP Files

Step Three: Add PHTML to MMDocumentTypes.xml

This file is an XML file which should be located in:

C:Program FilesAdobeDreamweaverConfigurationDocumentTypesMMDocumentTypes.XML

Mac OS X: Applications > Adobe Dreamweaver CS3 > configuration > DocumentTypes > MMDocumentTypes.XML

Add PHTML to this line (approx. line 75) twice, like so:

1. `<documenttype id="PHP_MySQL" servermodel="PHP MySQL" internaltype="Dynamic" winfileextension="php,php3,php4,php5,phtml" macfileextension="php,php3,php4,php5,phtml" file="Default.php" writebyteordermark="false" />`

Restart or Open Dreamweaver and you shouldn't have any problems with PHTML files any longer.