**Tutorial: Introduction to Magento Theme Development**

[](http://www.sitepoint.com/author/nick-jones/)

[Nick Jones](http://www.sitepoint.com/author/nick-jones/)

Published December 9, 2010

In 2008, a small company named Varien, based in Los Angeles, quietly released an open source ecommerce platform to the world. Magento, as it came to be known as, quickly gained traction with web developers wishing to move on from the days of osCommerce and Zen Cart to a more professional and robust system.

Magento has earned a reputation of being flexible and powerful, while remaining somewhat of a mystery to developers. Documentation for the platform was, and still is, scarce. In this tutorial, we’ll look at Magento’s powerful theme system, and learn how to build a simple custom theme for Magento.

**Magento Templates**

The template system that ships with Magento may seem a little complex at first, but the price we pay in complexity affords us a great deal of flexibility and power, as we’ll discover.

There are three components to the template system:

* PHP template files
* PHP block classes
* XML layout configuration

The **template files** contain what you’d expect a template system to handle, such as the HTML, JavaScript, and some PHP.

The **block classes** allow us to move the reusable functionality from the PHP template files into PHP classes, which can be used again on different template files in the future.  As a rule, all template files will have an associated block class.  Block classes are just normal PHP classes, and can be accessed in the template through the $this variable.

Various helper methods, such as getSkinUrl($path) or getUrl($path), are contained in all block classes. These methods are then used in a template file, by calling: $this->getSkinurl('images/logo.png'), for example.

Along with providing useful methods to the template files, blocks are also used to decide how to render the template file to the user. Each block has a toHtml() method for this purpose. Usually, the toHtml() method will simply parse the template file and output it to the user, but we could override that functionality and return anything we like, such as XML or JSON.

Finally, the **XML layout configuration** files are the “glue” that pulls together the entire set of template files to be rendered in the browser. In the XML, we can specify what PHP template/block combinations we’d like to load, and the order that we’d like to display them on the page.

**Folder Structure**

No doubt, one of the first things you’ll notice about Magento is that there are hundreds of folders, and thousands of files. Let’s take a look at a few of the more important ones for modifying templates and layouts:

app/

code/

{core,local,community}/

Company/

Module/

Block/

design/

frontend/

base/

default/

layout/

template/

mytheme/

default/

layout/

template/

christmas/

layout/

template/

skin/

frontend/

base/

default/

css/

images/

mytheme/

default/

css/

images/

christmas/

css/

images/

All template files will be in the .phtml files found in various folders under the app/design/frontend/{$interface}/{$theme}/template folder.  All layout XML files will be found in the app/design/frontend/{$interface}/{$theme}/layout folder.  All blocks will be in the app/code/{core,local,community}/{$company}/{$module}/Block folder.

But what are those interface and theme names in there? An interface, in Magento template terms, is a collection of themes. A theme, therefore, is just a bunch of layout and template files. Typically, in a given Magento installation, you’ll have one interface with many themes. The interface will define the overall layout of your ecommerce site, while a theme sits on top of it and gives it a particular look. Say you have an interface for your site, and a default theme. When you want to customize your shop for a holiday promotion, for example, you’d just create a new theme in the same interface and switch to that one for the duration of the sale.

**The Inclusion Hierarchy**

Much like WordPress, Magento relies on a hierarchy to locate theme files. Whenever a module specifies that a layout file needs to be loaded by the system, Magento will look in a number of places.

As an example, let’s say that the Mage\_Customer module is requesting that customer.xml be loaded to define the layout for the customer/account/login page we’re trying to display. Let’s also assume that we’ve already created an interface called “sitepoint”; have three themes called “christmas,” “5 for 1,” and “default”; and have specified that we’d like to use the “christmas” theme in the Magento administration area.

Magento will check the following folders, in order, for a file named customer.xml:

* app/design/frontend/sitepoint/christmas/layout
* app/design/frontend/sitepoint/default/layout
* app/design/frontend/base/default/layout

This strategy for locating files is also used for template files and skin files. This means that there’s no need to copy all the layout, template, and skin files across to the Christmas theme; we only copy over the files that we’re going to be changing. Any unchanged files will be located and loaded by heading down the inclusion hierarchy.

This searching strategy makes multiple themes quite manageable.  If, for example, we decide that we’d only like to change the product page’s template, then we just need to copy app/design/frontend/sitepoint/default/template/catalog/product/view.phtml into the app/design/frontend/sitepoint/christmas/template directory. But if we decided that the CSS would stay the same, then we’d leave the styles.css file in skin/frontend/sitepoint/default/css where it is, and it would still be loaded.

**An Example Layout File**

Let’s take a look at (and dissect) an example layout XML file:

<layout>

<customer\_account\_register>

<block name="wrapper">

<block type="core/template" template="page/html/header.phtml">

<block name="navigation" template="page/html/navigation.phtml" />

</block>

</block>

</customer\_account\_register>

</layout>

There are currently two sections to the file: the section definition and the block definitions.

The section we’re defining is for the /customer/account/register page on the website, which is referenced in the XML as customer\_account\_register, called the **layout handle**.

When you’re visiting any page on a website running Magento, pay attention to the URL (with SEO rewriting turned off); they’ll always start with {$module}/{$controller}/{$action}. This gives an enormous clue as to which layout handle in the XML file to edit: {$module}\_{$controller}\_{$action}.

In the section definition, we list some blocks that we’d like included on the page. As you can see, blocks can either be self-closing (ending with />) or can contain other blocks.

Blocks can have a number of attributes that help us organize and describe how we’d like them to appear on a page. A minimal block definition must contain a block type and a name.

It’s possible to define the template associated with a block by specifying a **template** attribute. Earlier, we mentioned that all templates must have a block type. This relationship is asymmetrical; we can define a block that does not have a template. That block would simply have to use its toHTML() method to generate its output.

**Diving into Template Files**

Let’s take a look at the header.phtml file mentioned in the layout XML:

<html>

<head>

<title><?php echo $this->getTitle() ?></title>

</head>

<body>

<h1><a href="<?php echo $this->getUrl() ?>"><img src="<?php echo $this->getSkinUrl('images/logo.png') ?>" /></a></h1>

<?php echo $this->getChildHtml('navigation') ?>

<p>Welcome to my shop!</p>

We can see usage of the $this variable that was previously mentioned. There are some standard methods that are made available to all blocks, such as:

* getUrl($path): using the Base URL defined in the Magento administration area, this method will generate a full URL, which is useful if you’re moving from a development environment to production, for example.  Example usage: $this->getUrl('customer/account/login') will produce http://www.example.com/customer/account/login.
* getSkinUrl($path): this is the same idea as the getUrl() method, but works with the skin directory found in “skin/frontend” and the interface and theme active in the Magento Administration.  The same hierarchical lookup is done on files in the skin directory as is done with template files.
* getChildHtml($name): this allows us to call nested or child blocks defined in the layout. We’ll be looking at this in more detail shortly.

The getTitle() function in our example layout file is only available to the core/template block—which makes sense, as its role is to output the page’s title element.

**Nesting Child Blocks**

One of the more powerful methods made available is the getChildHtml() method. In the layout XML above, you can see that the header block has another block nested inside it. Magento won’t know where to output the child’s HTML in the template file, so we need to explicitly define where to put it by calling the getChildHtml($name) method, and providing the name of the child block we want to display as the first argument. Child blocks are a great way to separate out parts of a page into reusable components.

**More on Layout Handles**

As we previously mentioned, the layout handle maps a URL to a specific area in the layout file. Magento will merge various layout handles on each page load. These layout handles can be extended and modified, but are usually along the lines of:

* default
* STORE\_{$theme}
* THEME\_frontend\_{$interface}\_{$theme}
* {$module}\_{$controller}\_{$action}

If we want to add content to all pages on the site, it would be silly to have to specify every single handle we could possibly think of, so we just use the <default> handle.

When Magento renders a page to the user, it loads all the layout XML files it can find following the previously discussed inclusion hierarchy, and looks for the above handles in turn. It follows that absolutely any layout XML file could contain a handle to any page in the system.

**Referencing Declared Blocks**

All of the global blocks are defined in the default handle.

But now we have a problem. What if we want to add something to the header block on some pages, but not others? Does that mean we need to put the header separately into each layout handle? Fortunately for us, no. Using the <reference> tag solves this problem.

The reference tag gives you access to the inside of another block: anything inside that reference tag will be placed inside the target block. How does it know which block to target? You just set the name attribute to be the same as the targeted block. This way, you can add new children to any block from any layout file:

<layout>

<default>

<block name="wrapper">

<block type="core/template" template="page/html/header.phtml">

<block name="navigation" template="page/html/navigation.phtml" />

</block>

<block type="core/template" template="page/html/footer.phtml" />

</block>

</default>

<customer\_account\_register>

<reference name="wrapper">

<block type="core/template" name="customer\_hello" template="customer/helloworld.phtml" />

</reference>

</customer\_account\_register>

</layout>

**Deleting Declared Blocks**

What if, for instance, we’ve defined a block in the default handle, but we want to remove it from the *customer\_account\_register* page only? Here’s an example where we get rid of the footer on the *customer/account/register* page:

<layout>

<default>

<block name="wrapper">

<block type="core/template" template="page/html/header.phtml">

<block name="navigation" template="page/html/navigation.phtml" />

</block>

<block type="core/template" template="page/html/footer.phtml" />

</block>

</default>

<customer\_account\_register>

<reference name="wrapper">

<block type="core/template" name="customer\_hello" template="customer/helloworld.phtml" />

<remove name="footer" />

</reference>

</customer\_account\_register>

</layout>

**The Wrapper and Block Ordering**

You may have noticed that the header and footer blocks are actually wrapped by a block called wrapper, of type text/list.  To explain how this works, we need to head back to the toHtml() method of a block.

The toHtml() method of the text/list block will inspect all its child blocks, and then, individually, call the toHtml() methods on those child blocks. The order of the blocks being output is, generally, the order in which they appear in the XML; however, we can override this behavior with the after and before attributes. So the wrapper is just a list of other blocks that will be displayed in a certain order.

Here’s the revised layout XML with some modifications to the positioning of the blocks:

<layout>

<default>

<block name="wrapper">

<block type="core/template" template="page/html/header.phtml" >

<block name="navigation" template="page/html/navigation.phtml" />

</block>

<block type="core/template" template="page/html/footer.phtml"/>

</block>

</default>

<customer\_account\_register>

<reference name="wrapper">

<block type="core/template" name="customer\_hello" template="customer/helloworld.phtml" after="header" />

</reference>

</customer\_account\_register>

</layout>

**Best Practices**

Although we’ve yet to fully explore layouts and templates to their fullest extents, I feel an obligation to start us off with some discussion about best practices. With Magento being such a difficult beast to tame, even for the most experienced developers, it’s easy to fall into the trap of inadvertently flouting best practice. With that in mind, here are a few guiding principles for your theme development.

**Hands off base/default**

As discussed, *base/default* is the theme used by Magento as a fallback for when it can’t find the template file in all the more important locations in the system. It’s for this reason that Magento places all the default templates and layout files here. When you update Magento, the default templates will likely be modified. Any such updates will be written to the *base/default* folder.

If you’ve used the *base/default* theme to contain your templates and layout changes, there’s a high probability that they’ll be lost with the update. It’s for this reason that if you’d like your client to be able to upgrade their Magento installation, **do not ever modify *base/default*!**

The quickest and easiest way to get around this problem is to create a new interface with a default theme (by simply creating a new folder in app/design/frontend/{$interface}/default); copy any file you’d like to change over to that folder (following the same directory structure as *base/default*) and make your changes there.

**Use local.xml**

You’ll notice that the layout folder of *base/default* contains tens of XML files, all relating to various different modules in the system. When creating your own theme that overrides the *base/default* functionality, it’s generally considered best practice to let Magento load the *base/default* customer.xml, so you should avoid placing your own customer.xml in your theme’s layout folder. Instead, apply your changes to a different file called local.xml.

The local.xml file lives in your app/design/frontend/{$interface}/{$theme}/layout directory, and will likely contain numerous update handles by the end of the theming process.

Magento loads the local.xml file automatically, so any changes you make will be pulled into the site.

**A Real World Example**

So we have the theory nailed, right?  Let’s get started and make the beginnings of our very own theme for Magento. Before we begin, you should [download Magento](http://www.magentocommerce.com/download) and install it on your local web server or virtual machine.

**Setting Up**

First things first, let’s create a folder to contain our own files. Create the following folder structure and files (replacing mycompany with your desired interface name):

app/design/frontend/

mycompany/

default/

layout/

local.xml

template/

skin/frontend/

mycompany/

default/

images/

css/

The local.xml file can remain empty for the moment.

Next, let’s enable our theme in the Magento administration area. Navigate to **System->Design** in the administration dashboard. Choose **Add Design Change** and select your new theme. As you can see, you can set a design change to come into effect during certain days of the year. This would be perfect for the Christmas example we used earlier on, but for now just leave those fields blank to have your design become the default.

If you refresh the front end, you’ll see that nothing has changed. This is normal, because the pages are cached. To disable caching so that you can see your changes during development, go to **System->Cache Management** and disable everything.

Back in the front end, you’ll notice that all the styles have disappeared. Take a look at the source of the page. All the URLs for style sheets and images are still pointing to base/default. This is normal: Magento is following the include hierarchy, having noticed that the files in question don’t exist in our theme yet. base/default doesn’t have a styles.css file, so all the styles have disappeared. To give ourselves a starting point, let’s take the style sheets from skin/frontend/default/default/css and put them into our skin/frontend/mycompany/default/css folder. Likewise, move the images from default/default to the images folder in your new theme. As you modify the theme, you can delete any of these that you won’t be using, but for now it’s good to have it to look at while you work.

**Removing a Block**

Magento ships with a “Back to School” advert and the PayPal logo in the right-hand column of the home page. Turns out that what we sell is irrelevant to students, and we don’t accept PayPal, so let’s remove them. We’d like to remove the blocks from every page on the site, so we want to place the removal code in the default layout handle.

First, we need to identify the names of the blocks. In order to work out the names of the blocks for our further steps, we’re going to have to enable block debugging. Go to **System->Configuration->Developer->Debug**, and enable the **Template Path Hints** and **Add Block Names to Hints** options. (If you don’t see these options, make sure you’re in anything but the global scope switcher in the top left of your screen)

Now, when you reload the front end, you’ll see the name of every block displayed over it on the page. We need to search through the layout XML files in app/design/frontend/base/default/layout/ to find the name of the block with the type and template we’re looking for. If your editor or IDE has a folder-wide search option, this is a great time to put it to use!

The “Back to School” advert has the block type Mage\_Core\_Block\_Template and the template callouts/right\_col.phtml. The PayPal logo has the block type Mage\_Paypal\_Block\_Logo and the template paypal/partner/logo.phtml.

After some detective work, we find the PayPal block defined as the following in paypal.xml:

<block name="paypal.partner.right.logo" template="paypal/partner/logo.phtml"/>

And the “Back to School” block defined as the following in catalog.xml:

<block name="right.permanent.callout" template="callouts/right\_col.phtml">

<action method="setImgSrc"><src>images/media/col\_right\_callout.jpg</src></action>

<action method="setImgAlt" translate="alt" module="catalog"><alt>Keep your eyes open for our special Back to School items and save A LOT!</alt></action>

</block>

Don’t worry about those action tags; what’s important is that we now have our names: paypal.partner.right.callout and right.permanent.callout.

The XML, which is to be placed in our local.xml, for this is as follows:

<?xml version="1.0"?>

<layout>

<default>

<remove name="paypal.partner.right.logo" />

<remove name="right.permanent.callout" />

</default>

</layout>

Refresh the page and you should find the blocks have disappeared!

**Reorganizing the Sidebar**

Now that we’ve removed those blocks, we have a sidebar containing a mini cart, “compare products” box, and a poll. We’ve only just launched, so it might be a nice idea to make our poll more prominent to attract feedback from our customers. Let’s move the poll to the top of the column.

The way we’ll do that is by deleting the blocks, and then adding them back in with the correct before and after attributes to achieve our desired layout.

Let’s take a look at the various sections that make up the sidebar. We’re looking for a <reference name="right">, because the blocks appear in the right column; we’re also looking for the block type and template names we can see in the red boxes on the front end.

We find our all-important poll definition in our right sidebar:

<default>

<reference name="right">

<block type="poll/activePoll">

<action method="setPollTemplate"><template>poll/active.phtml</template><type>poll</type></action>

<action method="setPollTemplate"><template>poll/result.phtml</template><type>results</type></action>

</block>

</reference>

</default>

Avoid being intimidated by those action tags; all they do is allow you to call a method from the block class from within the layout file. We can see that there are currently no before or after attributes defined for the block. Our local.xml file now can be updated to:

<?xml version="1.0"?>

<layout>

<default>

<remove name="paypal.partner.right.callout" />

<remove name="right.permanent.callout" />

**<reference name="right">**

**<remove name="right.poll" />**

**<block type="poll/activePoll" before="cart\_sidebar">**

**<action method="setPollTemplate"><template>poll/active.phtml</template><type>poll</type></action>**

**<action method="setPollTemplate"><template>poll/result.phtml</template><type>results</type></action>**

**</block>**

**</reference>**

</default>

</layout>

**Add an Advert**

Okay, so it’s all going great. We’ve sorted out our right-hand column, and we’re receiving a load of user feedback through the poll. *Super Big Corporation, Inc.* has approached us, and they’d like to sponsor our site in return for showing an advert in the right-hand column. Fantastic news! Let’s do it.

The first thing we need to do is make a template. We can put this in any folder in app/design/frontend/mycompany/default/template; we’ll create a new folder called sponsor, and a file inside it called superbig.phtml. That file can contain whatever HTML you want to display as the advertisement.

Next, we need to add it to the layout XML. We can use the block type core/template, as we’re not going to require any functionality that warrants the creation of a new block:

<?xml version="1.0"?>

<layout>

<default>

<remove name="paypal.partner.right.callout" />

<remove name="right.permanent.callout" />

<reference name="right">

<remove name="right.poll" />

<block name="right.poll" before="cart\_sidebar">

<action method="setPollTemplate"><template>poll/active.phtml</template><type>poll</type></action>

<action method="setPollTemplate"><template>poll/result.phtml</template><type>results</type></action>

</block>

**<block type="core/template" template="sponsor/superbig.phtml" />**

</reference>

</default>

</layout>

**Adding a New Style Sheet**

We want to add some customization to the current base/default style sheet that’s being loaded at the moment. Let’s include another CSS file, so that we can add our own tweaks.

The loading of assets on the page, such as JavaScript and CSS, is handled by Magento’s head block in the layout file.

**The head Block**

Let’s take a look at the head block as it stands. It’s located in page.xml:

<block name="head" as="head">

<action method="addJs"><script>prototype/prototype.js</script></action>

<action method="addJs" ifconfig="dev/js/deprecation"><script>prototype/deprecation.js</script></action>

<action method="addJs"><script>lib/ccard.js</script></action>

<action method="addJs"><script>prototype/validation.js</script></action>

<action method="addJs"><script>scriptaculous/builder.js</script></action>

<action method="addJs"><script>scriptaculous/effects.js</script></action>

<action method="addJs"><script>scriptaculous/dragdrop.js</script></action>

<action method="addJs"><script>scriptaculous/controls.js</script></action>

<action method="addJs"><script>scriptaculous/slider.js</script></action>

<action method="addJs"><script>varien/js.js</script></action>

<action method="addJs"><script>varien/form.js</script></action>

<action method="addJs"><script>varien/menu.js</script></action>

<action method="addJs"><script>mage/translate.js</script></action>

<action method="addJs"><script>mage/cookies.js</script></action>

<action method="addCss"><stylesheet>css/styles.css</stylesheet></action>

<action method="addItem"><type>skin\_css</type><name>css/styles-ie.css</name><params/><if>lt IE 8</if></action>

<action method="addCss"><stylesheet>css/widgets.css</stylesheet></action>

<action method="addCss"><stylesheet>css/print.css</stylesheet><params>media="print"</params></action>

<action method="addItem"><type>js</type><name>lib/ds-sleight.js</name><params/><if>lt IE 7</if></action>

<action method="addItem"><type>skin\_js</type><name>js/ie6.js</name><params/><if>lt IE 7</if></action>

</block>

As we’ve previously mentioned, the action tags allow us to call the methods associated with the block classes.

There are a number of methods defined for this block, but we’re only really interested in a couple:

* addJs: this allows us to include JavaScript that’s located in the /js directory at the root of our Magento installation.
* addItem: this allows us to include assets that we have in our skin directory for our theme.

The syntax for including a style sheet is:

<action method="addItem"><type>$type</type><name>$name</name><params>$params</params><if>$if</if></action>

The values for $type that you’ll most likely use in a career as a Magento designer/developer are:

* skin\_js
* skin\_css

As their names imply, they load JavaScript and CSS respectively, located in the skin folder of your theme, following the inclusion hierarchy if not found.

The $name refers to the location of your asset from the skin/frontend/mycompany/default/ folder, so a style sheet in skin/frontend/mycompany/default/css/updates.css will have the $name css/updates.css.

For style sheets, the $params value can be used to pass parameters to the <link /> tag, such as the media the style sheet is applicable to.

The $if value allows you to wrap a style sheet in [conditional comments](http://reference.sitepoint.com/css/conditionalcomments) for Internet Explorer.

**Adding to the head Block**

So we’ve had an introduction to the head block and how we can use it to include various assets; let’s add our own style sheet.

We’d like the style sheet to be loaded on every page, so we’ll use the default layout handle. We’re updating an already defined block, so we’ll be using a reference.

Our updated layout code now looks like:

<?xml version="1.0"?>

<layout>

<default>

<remove name="paypal.partner.right.callout" />

<remove name="right.permanent.callout" />

<reference name="right">

<remove name="right.poll" />

<block type="poll/activePoll" before="cart\_sidebar">

<action method="setPollTemplate"><template>poll/active.phtml</template><type>poll</type></action>

<action method="setPollTemplate"><template>poll/result.phtml</template><type>results</type></action>

</block>

</reference>

**<reference name="head">**

**<action method="addItem">**

**<type>skin\_css</type>**

**<name>css/updates.css</name>**

**<params />**

**<if />**

**</action>**

**</reference>**

</default>

</layout>

**Congratulations!**

There you go! You’ve learned how to add style sheets, remove blocks, add new blocks, and generally bend Magento to your will. Of course, there’s a huge amount we haven’t covered here, but I hope I’ve given you the foundations to go out and develop your own Magento themes. To learn more, check out the [Theming & Design section](http://www.magentocommerce.com/knowledge-base/categories/category/theming-design/) in the [Magento Knowledge Base](http://www.magentocommerce.com/knowledge-base), as well as the [Magento wiki](http://www.magentocommerce.com/wiki/).