

No Frills Magento Layout

Alan Storm

April 2011

Contents

0	No Frills Magento Layout: Introduction	5
0.1	Who this Book is For	6
0.2	No Frills	6
0.3	Installing Modules	7
0.4	Parting Words	7
0.5	Bugs in the Book	8
0.6	About the Author	9
0.7	Let's Go	9
1	Building Layouts Programmatically	10
1.1	Template Blocks	12
1.2	Template Files	12
1.3	Back to our Template	13
1.4	Nesting Blocks	14
1.5	Advanced Block Functionality	18
1.6	Block Methods	20
1.7	Enter the Layout	21
1.7.1	What's a Singleton!?	23
1.8	Back to the Code	23
1.9	Who's the Leader	24
1.10	Method Chaining	25
1.11	A Full Page Layout	25
1.12	Initializing the Layout and Setting Content	27
1.13	Insert vs. Set	28
1.14	Getting a Reference and Text List	28
1.15	A Recap and a Dilema	29
2	XML Page Layout Files	31
2.1	Hello World in XML	32
2.2	An Interesting use of the Word Simple	33
2.3	Adding the XML, Generating the Blocks	34
2.4	Getting a Little More Complex	35
2.5	Action Methods	38

2.6	References and the Importance of text_lists	38
2.7	Layout Updates	41
2.8	What's an Update	41
2.8.1	What's a "Model"	42
2.9	Adding our Updates	43
2.10	Fully Armed and Operational References	43
2.11	Removing Blocks	47
2.11.1	Before (<i>Figure 2.3</i>)	49
2.11.2	After (<i>Figure 2.4</i>)	49
2.12	What's Next	50
3	The Package Layout	52
3.1	The Why and Where of the Package Layout	53
3.2	Package Layout Examples	55
3.3	What is a Handle?	55
3.4	Rendering a Magento Layout	57
3.5	Getting a Handle on Handles	58
3.6	More local.xml	63
3.7	Adding Other Handles to the Page Layout	63
3.8	Package Layout Term Review	65
3.8.1	Package Layout	65
3.8.2	Page Layout	65
3.8.3	Layout Update XML Fragment	66
4	Bringing it All Together	67
4.1	How a Magento Layout is Built	67
4.2	What is the Page Layout	68
4.3	Rendering a Layout	69
5	Advanced Layout Features	71
5.1	Action Parameters	71
5.2	Translation System	72
5.3	Conditional Method Calling	73
5.4	Dynamic Parameters	73
5.5	Ordering of Blocks	75
5.6	Reordering Existing Blocks	78
5.7	Template Blocks Need Not Apply	79
5.8	Block Name vs. Block Alias	80
5.9	Skipping a Child	81
6	CMS Pages	82
6.1	Creating a Page	82
6.1.1	Page Information : Page Title	85
6.1.2	Page Information : URL Key	85
6.1.3	Page Information : Store View	85
6.1.4	Page Information : Status	85

6.1.5	Content: Content Heading	85
6.1.6	Content: Editor	86
6.1.7	Meta : Keywords	86
6.1.8	Meta : Description	86
6.1.9	Design : Layout	86
6.1.10	Design : Layout Update XML	88
6.1.11	Design : Custom Design	88
6.2	CMS Page Rendering	88
6.3	Index Page	89
6.4	What You Need to Know	90
6.5	Where's the Layout?	91
6.6	Adding the CMS Blocks	92
6.7	Setting the Page Template	93
6.8	Rendering the Content Area	94
6.9	Page Content Filtering	95
6.10	Filtering Meta Programming	96
7	Widgets	98
7.1	Widgets Overview	98
7.2	Adding a Widget to a CMS Page	100
7.3	CMS Template Directives	102
7.4	Adding Data Property UI	103
7.5	Widget Templates	105
7.6	Instance Widgets	108
7.7	Creating an Instance Widget	108
7.8	Inserting a Widget	110
7.9	Behind the Scenes	111
7.10	Restricting Blocks.	112
7.11	Per Theme Widget Config	114
7.12	Wrap Up	114
A	Magento Block Hierarchy	115
B	Class Aliases	139
B.1	Why so Complicated?	140
B.2	What Class?	140
B.3	Class Rewrites	141
C	Creating Code Modules	142
C.1	Adding a Module	143
C.2	Enabling your Module	144
C.3	Next Steps	145
D	Block Action Reference	146
E	Theme and Layout Resolution	147

E.1	Template Resolution	147
E.2	The Base Package	148
E.3	Layout Files	148
F	The Hows and Whys of Clearing Magento’s Cache	149
G	Magento Setters and Getters	151
G.1	Getter and Setter	152
G.2	Other Magic Methods	153
H	Widget Field Rendering Options	154
H.1	Creating Your Own Form Elements	156
H.2	Advanced Examples	158
I	System Configuration Variables	159
J	Magento Connect	161
J.1	What is an Extension	161
J.2	Installing Extensions: The GUI Way	162
J.3	Installing Extensions: The Command Line Way	162
J.3.1	Magento Connect CLI install for Magento 1.42	162
J.3.2	Magento Connect CLI install for Magento 1.5+	163

Chapter 0

No Frills Magento Layout: Introduction

If you're reading this intro, chances are you know something about Magento. Maybe you've chosen it for your new online store, maybe it's been chosen for you, or maybe you're just the curious type. Whatever the reason you've kicked the tires, liked what you've seen, and ran to this book for help once you opened the hood.

Magento isn't **just** a shopping cart. It's an entire system for programming web applications and performing system integrations. The PHP you see here is not your father's PHP. It's probably not even your PHP. Magento takes enterprise java patterns and applies them to the PHP language. More than any system available today, it's pushing the limits of what's possible with object oriented PHP code.

When it comes to layout engines, Most PHP MVC systems use a simple outer-shell/inner-include approach. Magento does not. At the top of the Magento view layer there's a layout object, which controls a tree of nested block objects. Magento uses a domain specific programming language, implemented in XML, to create, configure, and render this nested tree of block objects into HTML. This layer is separate from the rest of the application, allowing non-PHP developers an unprecedented level of power to change their layouts without having to touch a single line of PHP code.

If the above paragraph was greek to you don't worry, you're not alone. With all that power available there's a learning curve to Magento that can be hard to climb by yourself. This book is your guide up that learning curve. We'll tell you what you need to know to quickly become a Magento Layout master.

0.1 Who this Book is For

This book is for interactive developers and software engineers who want to fully understand Magento's XML based Layout system.

By interactive developer we mean someone who both designs online experiences, and **implements** them using a mix of HTML/CSS/Javascript and some glue/template programming in a dynamic language like PHP, Ruby, Python, or one of those language's many template systems. There are parts of the book where we'll dive in depth into how a particular system is built, but only so that you can better understand the context of where and when to use it. Designer-coders are quickly taking over the agency world, and this book seeks to give them the tools they need to succeed.

Software engineer always seemed a fancier title than most jobs entail, so substitute software developer, or even PHP developer, if you're uncomfortable with engineer. Chances are if you work for a shop that does more than just crank out web stores you're going to be asked to extend, enhance, and generally abuse Magento, including the Layout system. In teaching you the practical, this book will also teach and inform on the engineering assumptions of the Layout system. After reading through this book you'll not only understand how to use the Layout system, you'll understand why it was built the way it was, which in turn will help you make better engineering decisions on your own project.

This book assumes some basic PHP and Magento knowledge. If you haven't already done so, reviewing the Magento Knowledge Base, as well as the additional articles on the author's website will help you get where you need to with Magento.

<http://www.magentocommerce.com/knowledge-base>

<http://alanstorm.com/category/magento>

You don't need to be a Magento master, but you should be passably familiar with the application. If you aren't, you will be by the time you're done! While the main text of the Book is focused on the Layout and related systems, whenever a deeper knowledge of Magento is needed the Appendixes will give you the overview you need to keep working.

0.2 No Frills

Why No Frills? Because we tell you what you need to know, and nothing more. Mandated book lengths make sense in a physical retail environment, but with the internet being the preferred way of distributing technical prose, there's no need to pad things out.

With that in mind, lets get started!

0.3 Installing Modules

This book was distributed with an archive containing several versions of a Magento module named `Nofrills_Booklayout`. If you want to add code to a Magento system, you create a module. The `Nofrills_Booklayout` module is where the example code in this book will go. You'll be building this module up as you go along. For each chapter in the book, we've included the module as it should be at the start of the chapter, and how it should be at the end.

You'll also find a copy of each and every code example in the `code/all` folder. If you don't want to manually type in code examples from the book, copy and paste the contents of these files into your source code editor.

There are two ways to install the module. The first is manually. If you extract the files, you'll see a folder structure like

```
app/code/local/Nofrills_Magento
app/module/etc/Nofrills_Magento.xml
app/.....
```

The archive structure mirrors where the files should be placed in your system. This is the standard layout of a Magento extension. Place the files in the same location on your own installation, clear your cache, and the extension will be loaded into the system on the next page request. For more background, read the Magento Controller Dispatch and Hello World article online

http://alanstorm.com/magento_controller_hello_world

If you're not up for a manual install, each archive is also a fully valid Magento Connect package. Magento Connect is Magento Inc's online marketplace of free extensions. It's also a package management system. For background on Magento Connect and instructions for installing its packages, please see Appendix J.

0.4 Parting Words

A few last things before we start. Magento has a special operating mode called `DEVELOPER_MODE`. When running in `DEVELOPER_MODE` Magento is less tolerant of small coding errors, and will not hide fatal errors and uncaught exceptions from the end user. You'd never want to run a production store in `DEVELOPER_MODE`, but it can make working with and learning the system much easier. You'll want to turn `DEVELOPER_MODE` on while working your way through this book. You can do this by either

1. Adding `SetEnv MAGE_IS_DEVELOPER_MODE 1` to your `.htaccess` file
2. Alternately, editing `index.php`

If you choose the second option, look for lines in your `index.php` file something like


```
if (isset($_SERVER['MAGE_IS_DEVELOPER_MODE'])) {  
    Mage::setIsDeveloperMode(true);  
}
```

You'll want to make sure the `Mage::setIsDeveloperMode(true);` call is made. Also, while you're in `index.php`, it'd be a good idea to tell PHP to show errors by changing this

```
#ini_set('display_errors', 1);
```

to this

```
ini_set('display_errors', 1);
```

Seemingly invisible errors are one of the most frustrating things for a developer new to any system. By configuring Magento to fail fast we'll be setting ourselves up to better learn what needs to be done for any given task.

Magento's a fast changing platform, and while the concepts in this book will apply to all versions the specifics may change as Magento Inc changes its focus. It should go without saying you should run the exercises presented here on a development or testing server, and **not** your production environment. The following legal notice is the fancy way of saying that

```
THIS BOOK AND SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND  
CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES,  
INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF  
MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE  
DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS  
BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,  
EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED  
TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,  
DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON  
ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR  
TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF  
THE USE OF THIS BOOK AND SOFTWARE, EVEN IF ADVISED OF THE  
POSSIBILITY OF SUCH DAMAGE.
```

0.5 Bugs in the Book

If you're having trouble working your way through the examples, post a detailed question to the programming Q&A site Stack Overflow

<http://stackoverflow.com/tags/magento>

with the following tags

```
magento magento-nofrills
```

We'll be monitoring the site for any problems with code examples, and by asking your questions in a public forum you'll be helping the global Magento developer

community. Developers are often amazed when they find people across the world are having the same problems they are, and often already have a solution ready to share.

Additionally, each chapter will contain a link to a site online for discussions specific to each chapter. You're not just getting a book, you're joining a community.

0.6 About the Author

No Frills Magento Layout was written by Alan Storm. Alan's an industry veteran with over 12 years on-the-job experience, and an active member of the Magento community. He's written the go-to developer documentation for the Magento Knowledge Base, and is the author of the popular debugging extension Commerce Bug. You can read more about Alan and his Magento products at the following URLs

<http://alanstorm.com/>

<http://store.pulestorm.net/>

0.7 Let's Go

That's it for pleasantries, let's get started. In the first chapter we're going to start by creating Magento layouts using PHP code.

Visit <http://www.pulestorm.net/nofrills-layout-introduction> to join the discussion online.

Chapter 1

Building Layouts Programmatically

Before we can understand the layout system in its entirety, we need to understand its individual parts. With that in mind, we'll start with some simple examples. A Layout can be defined with the following phrase

A Layout is a collection of blocks in a tree structure

So, let's start by defining what a block is.

A Magento block is an object with a `toHtml` method defined. When this `toHtml` method is called, it returns the string which should be output to the screen. Typically "the screen" means your web browser. In addition to having a `toHtml` method, Magento blocks inherit from the `Magento_Core_Block_Abstract` class. This class contains a number of other useful block helper methods. We'll get to these eventually, but for now just think of a block as **an object** which has a `toHtml` method, and when this `toHtml` is called **output is sent to the screen**.

Let's give this a try. If you installed the `Nofrills_Booklayout` module that came with this book, you can open the following url on your system

http://magento.example.com/nofrills_booklayout/index/index

which corresponds to the controller file at

`app/code/local/Nofrills/Booklayout/controllers/IndexController.php`

We'll be adding our code examples to the `indexAction` method

```
public function indexAction()  
{  
    var_dump(__METHOD__);  
}
```

If you load the above URL with an unmodified extension, you should see a mostly blank browser screen that looks something like *Figure 1.1*

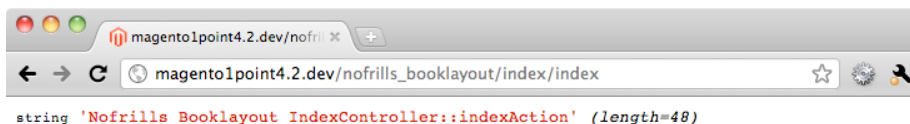


Figure 1.1

First, we'll create a simple text block.

```
public function indexAction()
{
    $block = new Mage_Core_Block_Text();
    $block->setText("Hello World");
    echo $block->toHtml();
}
```

Our first line instantiates a block object from the class `Mage_Core_Block_Text`. Our second line sets the text we want to output, and the third line calls the `toHtml` method, which returns our string and echos the output.

If you reload your browser page, you should see the following output.

```
Hello World
```

So far so good. We now have an object oriented echo statement. In our example above we instantiated a `Mage_Core_Block_Text` object. When you call this type of block's `toHtml` method, it simply outputs whatever text has been set with the `setText` method.

Magento has literally hundreds of different types of block classes for every possible need. The Magento core team subscribes to a style of development that's similar to Java and C# programming that says

When in doubt, make a new class

Each block type may have a slightly different implementation of how its `toHtml` method is implemented. Fortunately, you don't need to know what every single

block class does. In fact, you can accomplish most of what you'll ever need with the `Magento_Core_Block_Template` class.

1.1 Template Blocks

Most PHP developers quickly discover that producing HTML output by concatenating strings in PHP leads to code that's hard to debug and maintain. That's why most HTML Output/View systems break out the HTML into template files. Magento is no different. As mentioned, the majority of the blocks in the system inherit from the `Magento_Core_Block_Template` block class.

Each `Magento_Core_Block_Template` object has an associated phtml template file. When a template block's `toHtml` method is called this phtml template will be output using PHP's built-in `include` statement. Output is routed into a variable using output buffering. By including the template from a class method, the template gains access to all the parent block's public, private, and protected methods.

If that didn't quite make sense, an example should clear things up. Let's create a block object from the `Magento_Core_Block_Template` class, set a template, and then output it.

```
public function indexAction()
{
    $block = new Magento_Core_Block_Template();
    $block->setTemplate('helloworld.phtml');
    echo $block->toHtml();
}
```

With the above in your controller, reload the page and ... nothing happened. That's because we didn't create a `helloworld.phtml` file. Let's take care of that!

1.2 Template Files

Of course this raises the question, "where do template files live in the system?". Magento has a hierarchical design theming/packaging system that determines where your template files should be stored. Magento will look in a folder with the following naming conventions

```
[BASE DESIGN FOLDER]/[AREA FOLDER]/[DESIGN PACKAGE FOLDER]/[THEME FOLDER]/template
```

More recent versions of Magento have a fallback mechanism, where if a folder isn't found at one of the above locations, Magento will check a "base" design package for the same file

```
[BASE DESIGN FOLDER]/[AREA FOLDER]/base/[THEME FOLDER]/template
```

This allows you to rely on the base Magento design package, and only add files that you wish to change to your own packages and themes. See Appendix E for more information if you're interested in how this fallback system works. Here's a little trick to find out where Magento is loading any block's template from.

```
public function indexAction()
{
    $block = new Mage_Core_Block_Template();
    $block->setTemplate('helloworld.phtml');
    var_dump($block->getTemplateFile());
    //echo $block->toHtml();
}
```

By calling the block's `getTemplateFile` method, we're doing the same thing Magento will when rendering the block. Running the above will result in

```
string 'frontend/base/default/template/helloworld.phtml' (length=47)
```

As mentioned, since we haven't created a `helloworld.phtml` file, Magento falls back to the base package/theme.

1.3 Back to our Template

We're going to assume you're working on a freshly installed Magento system, which means you'll want to add your `helloworld.phtml` template to the default design package in the default theme. Create a file at the following location

```
app/design/frontend/default/default/template/helloworld.phtml
```

Add something like the following to that file

```
<?php #File: app/design/frontend/default/default/template/helloworld.phtml ?>
<h1>Hello World</h1>
<p>
Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor
incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis
nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.
Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu
fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in
culpa qui officia deserunt mollit anim id est laborum
</p>
```

As a reminder, our controller looks this

```
public function indexAction()
{
    $block = new Mage_Core_Block_Template();
    $block->setTemplate('helloworld.phtml');
    echo $block->toHtml();
}
```

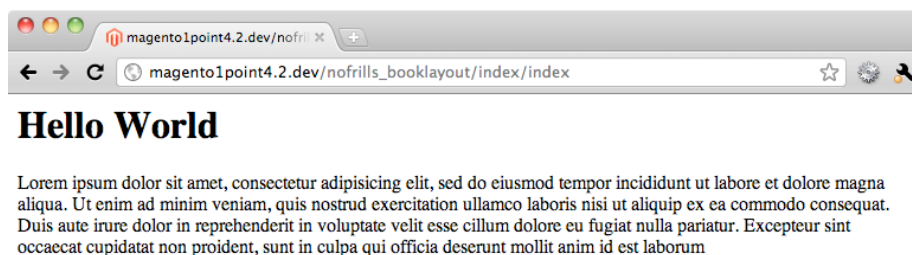


Figure 1.2

If you reload the page, you should see a hello world lorem ipsum, loaded from the template, (see *Figure 1.2*)

Congratulations, you've created your first template block!

1.4 Nesting Blocks

Let's go back to our Layout definition

A Layout is a collection of blocks in a tree structure

We've defined, very basically, what a block is, but what do we mean by "in a tree structure"?

Magento blocks are sort of like HTML nodes. For example, here

```
<p>
    <span>Lorem</span>
</p>
```

The `<p>` tag is the parent node, and the `` is the child node. All blocks share a similar relationship. Oversimplifying things a bit, this sort of parent/child relationship is known as a "Tree" in computer science circles.

Let's consider our previous template block. Alter the phtml file so it contains the following

```
<?php #File: app/design/frontend/default/default/template/helloworld.phtml
?>
<h1>Hello World</h1>
```

```
<p>
    <?php echo $this->getChildHtml('the_first'); ?>
</p>
<p>
The second paragraph is hard-coded.
</p>
```

There's a few new concepts to cover here. First, you'll notice we've dropped into PHP code

```
<?php $this->getChildHtml('the_first'); ?>
```

You may be wondering what `$this` is a reference to. If you'll remember back to our definition of a template block, we said that each template block object has a `phtml` template file. So, when you refer to `$this` within a `phtml` template, you're referring to the template's block object. If that's a little fuzzy future examples below should clear things up.

Next, we have the `getChildHtml` method. This method will fetch a child block, and call its `toHtml` method. This allows you to structure blocks and templates in a logical way. So, with the above code in our template, let's reload the page, (see *Figure 1.3*)

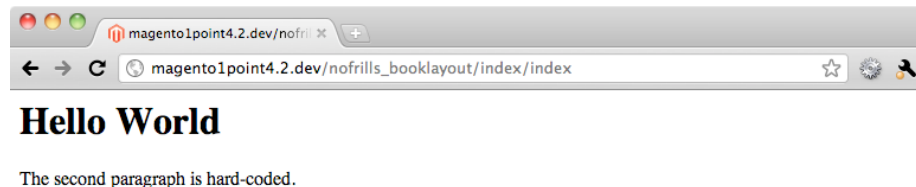


Figure 1.3

Our second hard-coded paragraph rendered, but nothing happened with our call to `getChildHtml`. That's because we failed to add a child. Let's change our controller action so it matches the following.

```
public function indexAction()
{
    $paragraph_block = new Mage_Core_Block_Text();
    $paragraph_block->setText('One paragraph to rule them all.');
```



```
$main_block = new Mage_Core_Block_Template();
$main_block->setTemplate('helloworld.phtml');

$main_block->setChild('the_first',$paragraph_block);
echo $main_block->toHtml();
}
```

We'll dissect this chunk by chunk. First, we have the following

```
$paragraph_block = new Mage_Core_Block_Text();
$paragraph_block->setText('One paragraph to rule them all..');
```

Here we've created a simple text block. We've set its text so that when the block is rendered, it will output the sentence *One paragraph to rule them all..*. Then, as we did before,

```
$main_block = new Mage_Core_Block_Template();
$main_block->setTemplate('helloworld.phtml');
```

we define a template block, and point it toward our hello world template. Finally (and here's the key)

```
$main_block->setChild('the_first',$paragraph_block);
```

Here we call a method we haven't see before, called `setChild`. Here we're telling Magento that the `$paragraph_block` is a child of the `$main_block`. We've also given that block a name (or alias) of `the_first`. This name is how we'll refer to the block later, and what we'll pass into our call to `getChildHtml`

```
<?php echo $this->getChildHtml('the_first'); ?>
```

Expressed as a generic XML tree, the relationship between blocks might look like

```
<main_block>
  <paragraph_block name="the_first"></paragraph_block>
</main_block>
```

Or maybe (getting a bit ahead of ourselves)

```
<block type="core/template" name="root" template="helloworld.phtml">
  <block type="core/text" name="the_first">
    <action name="setText">
      <text>One paragraph to rule them all</text>
    </action>
  </block>
</block>
```

A block may have an unlimited number of children, and because we're dealing with PHP 5 objects, changes made to the block after it has been appended will carry through to the final rendered object. Try the following code

```
public function indexAction()
{
    $block_1 = new Mage_Core_Block_Text();
    $block_1->setText('Original Text');

    $block_2 = new Mage_Core_Block_Text();
    $block_2->setText('The second sentence.');
```

 \$main_block = new Mage_Core_Block_Template();
 \$main_block->setTemplate('helloworld.phtml');

 \$main_block->setChild('the_first' , \$block_1);
 \$main_block->setChild('the_second' , \$block_2);

 \$block_1->setText('Wait, I want this text instead.');

 echo \$main_block->toHtml();
}

With the following template changes

```
<?php #File: app/design/frontend/default/default/template/helloworld.phtml
?>
<h1>Hello World</h1>
<p>
    <?php echo $this->getChildHtml('the_first'); ?>
    <?php echo $this->getChildHtml('the_second'); ?>
</p>
<p>
The second paragraph is hard-coded.
</p>
```

You should now see output something like

Hello World

Wait, I want this text instead. The second sentence.

The second paragraph is hard-coded.

One final trick with rendering child blocks. If you don't provide `getChildHtml` with the name of a block, **all child blocks** will be rendered. That means the following template will give us the same result as the one above

```
<?php #File: app/design/frontend/default/default/template/helloworld.phtml
?>
<h1>Hello World</h1>
<p>
    <?php echo $this->getChildHtml(); ?>
</p>
<p>
The second paragraph is hard-coded.
</p>
```

1.5 Advanced Block Functionality

There's a few more bits of block functionality we should cover before moving on.

The first thing we'll cover is creating your own block classes. There will be times where you want a block with some custom programmatic functionality. While it may be tempting to use a standard template block and then include all your logic in the phtml template, the preferred way of doing this is to create a Magento module for adding your own code to the system, and then adding your own block classes that extend the existing classes.

We're not going to cover creating a new module here, although if you're interested in learning the basics then checkout Appendix C. Instead, we'll have you create your custom block in the `NoFrills_Booklayout` module.

So, we just spent a lot of effort to create a hello world block. Let's take what we've done so far, and create a hello world block. The first thing we'll want to do is create a new class file at the following location, with the following contents

```
#File: app/code/local/NoFrills/Booklayout/Block/Helloworld.php
<?php
class NoFrills_Booklayout_Block_Helloworld extends Mage_Core_Block_Template
{
}
```

And then add the following code to the specific controller action, and load its corresponding URL in your browser

```
#http://magento.example.com/nofrills_booklayout/index/helloblock
public function helloblockAction()
{
    $block_1 = new Mage_Core_Block_Text();
    $block_1->setText('The first sentence. ');

    $block_2 = new Mage_Core_Block_Text();
    $block_2->setText('The second sentence. ');

    $main_block = new NoFrills_Booklayout_Block_Helloworld();
    $main_block->setTemplate('helloworld.phtml');

    $main_block->setChild('the_first',$block_1);
    $main_block->setChild('the_second',$block_2);

    echo $main_block->toHtml();
}
```

When you load the page in your browser, you should see your `helloworld.phtml` template rendered the same as before.

What we've done is create a new block named `NoFrills_Booklayout_Block_Helloworld`. This class extends `Mage_Core_Block_Template`, which means it automatically gains the same functionality as a standard template block.

Next, let's add the following method to our new class,

```
class Nofrills_Booklayout_Block_Helloworld extends Mage_Core_Block_Template
{
    public function _construct()
    {
        $this->setTemplate('helloworld.phtml');
        return parent::_construct();
    }
}
```

and remove the `setTemplate` class in our controller.

```
public function helloblockAction()
{
    $block_1 = new Mage_Core_Block_Text();
    $block_1->setText('The first sentence. ');

    $block_2 = new Mage_Core_Block_Text();
    $block_2->setText('The second sentence. ');

    $main_block = new Nofrills_Booklayout_Block_Helloworld();
    // $main_block->setTemplate('helloworld.phtml');

    $main_block->setChild('the_first',$block_1);
    $main_block->setChild('the_second',$block_2);

    echo $main_block->toHtml();
}
```

A page refresh should result in the same exact page.

Every block class can define an optional "pseudo-constructor". This is a method that's called whenever a new block of this type is created, but that is separate from PHP's standard constructor. What we've done is ensure that our block **always** has a template set.

```
public function _construct()
{
    $this->setTemplate('helloworld.phtml');
    return parent::_construct();
}
```

There's a few other special methods you can define in a block class. The first that we're interested in is `_beforeToHtml`. When we call `toHtml` on our block, this method is called immediately before the block content is rendered. There's also a corresponding `_afterToHtml($html)` method which is called after a block is rendered, and is passed the completed HTML string. We're going to use the `_beforeToHtml` method to automatically add our two child blocks, making everything self contained.

```
class Nofrills_Booklayout_Block_Helloworld extends Mage_Core_Block_Template
{
    public function _construct()
    {
```

```
$this->setTemplate('helloworld.phtml');
return parent::_construct();
}

public function _beforeToHtml()
{
    $block_1 = new Mage_Core_Block_Text();
    $block_1->setText('The first sentence. ');
    $this->setChild('the_first', $block_1);

    $block_2 = new Mage_Core_Block_Text();
    $block_2->setText('The second sentence. ');
    $this->setChild('the_second', $block_2);
}
}
```

This will let us remove the extraneous code from our controller

```
public function helloBlockAction()
{
    $main_block = new Nofrills_Booklayout_Block_Helloworld();
    echo $main_block->toHtml();
}
```

Again, a page refresh should result in the exact same page. We've gone from having to manually create our hello world block with 10 or so lines of code to completely encapsulating its functionality and output in 2 lines. This is a pattern you'll see over and over again in Magento.

1.6 Block Methods

The other thing we want to cover is calling, and adding, custom methods to your phtml templates. Go to your helloworld.phtml file and change the title line so it matches the following.

```
<!-- <h1>Hello World</h1> -->
<h1><?php echo $this->fetchTitle(); ?></h1>
```

If you reload your page with this in place, you'll get the following error

```
Invalid method Nofrills_Booklayout_Block_Helloworld::fetchTitle(Array
(
)
)
```

As previously mentioned, if you use the `$this` keyword in your template, you're referring to a template's parent block object. Let's add a method that returns the page title

```
class Nofrills_Booklayout_Block_Helloworld extends Mage_Core_Block_Template
{
    public function _construct()
```

```
{
    $this->setTemplate('helloworld.phtml');
    return parent::_construct();
}

public function _beforeToHtml()
{
    $block_1 = new Mage_Core_Block_Text();
    $block_1->setText('The first sentence. ');
    $this->setChild('the_first', $block_1);

    $block_2 = new Mage_Core_Block_Text();
    $block_2->setText('The second sentence. ');
    $this->setChild('the_second', $block_2);
}

public function fetchTitle()
{
    return 'Hello Fancy World';
}
}
```

Reload the page with the above `Nofrills_Booklayout_Block_Helloworld` in place, and you'll see your page with its new title.

This is the preferred way to create templates with dynamic data in Magento. Your `phtml` file should contain

1. HTML/CSS/Javascript code
2. Calls to `echo`
3. Looping and control structures
4. Calls to block methods

Any PHP more complicated than the above should be put in block methods. This includes calls to Magento models to read back data which was saved in the controller layer.

1.7 Enter the Layout

Coming back again to our definition of a Layout

A Layout is a collection of blocks in a tree structure

We now know what a block is and what a block can do. We understand how blocks are organized in a tree like structure. The only thing that leaves us to cover is the layout object itself.

A layout object, (instantiated from a `Mage_Core_Model_Layout` class)

- Is a wrapper object for interacting with your blocks.

- Provides helper methods for creating blocks
- Allows you to designate which block should start the rendering for a page
- Provides a mechanism for loading complex layouts described by XML files

Let's take a look at some layout examples. Add the following action to our controller

```
#http://magento.example.com/nofrills_booklayout/index/layout
public function layoutAction()
{
    $layout = Mage::getSingleton('core/layout');
    $block = $layout->createBlock('core/template','root');
    $block->setTemplate('helloworld-2.phtml');
    echo $block->toHtml();
}
```

Next, create a file named `helloworld-2.phtml` that's in the same location as your `helloworld.phtml` template.

```
<?php #File: app/design/frontend/default/default/template/helloworld-2.phtml
?>
<h1><?php //echo $this->fetchTitle(); ?></h1>
<h1>Hello World 2</h1>
<p>
    <?php echo $this->getChildHtml(); ?>
</p>
<p>
    The second paragraph is hard-coded.
</p>
```

Load your page and you'll see the second hello world template rendered in your browser, without any output for `getChildHtml` (as we didn't add any child nodes).

There's a lot new going on here, so let's cover things line by line.

```
$layout = Mage::getSingleton('core/layout');
```

This instantiates your layout object as a singleton model (see below). The string `core/layout` is known as a class alias. It's beyond the scope of this book to go fully into what class aliases are used for (see Appendix B: Class Alias for a better description), but from a high level; when creating a Magento model, a class alias is used as a shortcut notation for a full class name. It can be translated into a class name by the following set of transformations

Core Layout	<i>//adding a space at the slash, and capitalizing</i>
Core Model Layout	<i>//Add the word Model in between</i>
Mage Core Model Layout	<i>//Add the word Mage before</i>
Mage_Core_Model_Layout	<i>//underscore the spaces</i>

This is a bit of an over simplification, but for now when you see something like

```
$o = Mage::getModel('foo/bar');
$o = Mage::getSingleton('foo/bar');
```

just substitute

```
$o = new Mage_Foo_Model_Bar();
```

in your mind.

1.7.1 What's a Singleton!?

A singleton is a fancy object oriented programming term for an object that may only be instantiated once. The first time you instantiate it, a new object will be created. However, if you attempt to instantiate the object again, rather than create a new object, the originally created object will be returned.

A singleton is used when you only want to create a single instance of any type of object. Magento assumes you'll only want to render **one** HTML page per request (probably a safe assumption), and by using a singleton it's ensured you're always getting the same layout object.

If all that went over your head don't worry. All you need to know is whenever you want to get a reference to your layout object, use

```
$layout = Mage::getSingleton('core/layout');
```

1.8 Back to the Code

Next up we have the line

```
$block = $layout->createBlock('core/template','root');
```

This line creates a `Mage_Core_Template_Block` object named `root` (we'll get to the why of "root" in a bit) by calling the `createBlock` method on our newly instantiated layout object.

Again, in place of a class name, we have the `core/template` class alias. Because we're using the class alias to instantiate a **block**, this translates to

```
Mage_Core_Block_Template
```

Again, check Appendix B if you're interested in how class aliases are resolved. Whenever we use a class alias for the remainder of this book, we'll let you know the real PHP class.

Everything else from here on out should look familiar. The following

```
$block->setTemplate('helloworld-2.phtml');  
echo $block->toHtml();
```

sets our block template, and renders the block using its `toHtml` method. Let's use a class alias to instantiate our custom block from the previous examples

Prepared for Compaa Peruana de E-commerce S.A.; Copyright ©2011 Pulse23 Storm LLC


```
//class alias 'nofrills_booklayout/helloworld' is translated into
//the class name Nofrills_Booklayout_Block_Helloworld
public function layoutAction()
{
    $layout = Mage::getSingleton('core/layout');
    $block = $layout->createBlock('nofrills_booklayout/helloworld','root');
    echo $block->toHtml();
}
```

Reload the page, you should see our original block.

1.9 Who's the Leader

Give our next example a try

```
public function layoutAction()
{
    $layout = Mage::getSingleton('core/layout');
    $block = $layout->createBlock('nofrills_booklayout/helloworld','root');

    $layout->addOutputBlock('root');
    $layout->setDirectOutput(true);
    $layout->getOutput();
}
```

Refresh the page, and you should see the same output as your did before.

What we've done here is replace our call to the block's `toHtml` with the following

```
$layout->addOutputBlock('root');
$layout->setDirectOutput(true);
$layout->getOutput();
```

The call to `addOutputBlock` tells our layout block that **this** is the block that should start the page rendering process. Following that is a call to `getOutput`, which is the call that actually **starts** the page rendering process. Every time you use `createBlock` to create an object, the Layout object will know about that block. That's why we gave it a name earlier.

The call to `setDirectOutput` is us telling the Layout object that it should just automatically **echo** out the results of the page. If we wanted to capture the results as a string instead, we'd just use

```
$layout->addOutputBlock('root');
$layout->setDirectOutput(false);
$output = $layout->getOutput();
```

1.10 Method Chaining

Now's probably a good time to mention a PHP feature that Magento makes extensive use of called method chaining. Let's replace our code from above with the following.

```
public function layoutAction()
{
    $layout = Mage::getSingleton('core/layout');
    $block = $layout->createBlock('nofrills_booklayout/helloworld','root');
    echo $layout->addOutputBlock('root')->setDirectOutput(false)->getOutput();
}
```

You'll notice that we've trimmed a few lines from the code, but that we're using a funky syntax in that last line.

```
echo $layout->addOutputBlock('root')->setDirectOutput(false)->getOutput();
```

This is method chaining. It's not a Magento feature per se, but it's a feature of PHP that's become much more popular as applications start leveraging PHP 5 OOP capabilities.

If a call to a method returns an object, PHP lets you chain a another method call on the end for brevity. This can be repeated as long as each method call returns an object. The above is equivalent to the following

```
$block = $layout->addOutputBlock('root');
$block->setDirectOutput(false);
echo $block->getOutput();
```

You'll also see chaining syntax that spans multiple lines

```
$layout->addOutputBlock('root')
->setDirectOutput(false)
->getOutput();
```

Again, this isn't anything that's specific to Magento. It's just a pattern that's becoming more popular with PHP developers as PHP 5 style objects are used more and more. Magento **enables** this syntax by having most of its set, create, and add methods return an appropriate object. You're not required to use it, but get used to seeing it if you spend any time with core or community modules

1.11 A Full Page Layout

From here on out we're going to start using a special block we've created in the Nofrills module you installed. It's called

```
nofrills_booklayout/template
Nofrills_Booklayout_Block_Template
```

CHAPTER 1. BUILDING LAYOUTS PROGRAMMATICALLY

The block is identical to the Core template block, with one exception.

```
public function fetchView($fileName)
{
    //ignores file name, just uses a simple include with template name
    $this->setScriptPath(
        Mage::getModuleDir('', 'Nofrills_Booklayout') .
        DS .
        'design'
    );
    return parent::fetchView($this->getTemplate());
}
```

We've overridden the `fetchView` function in our template with the code above. What this does is move the base folder for templates from

`app/design`

to a folder in our local module hierarchy.

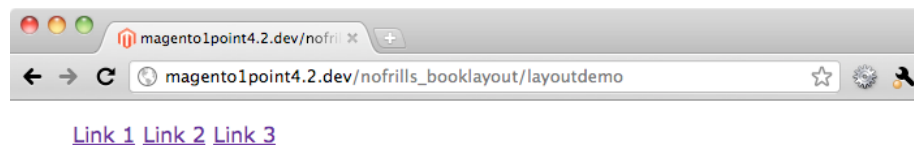
`app/code/local/Nofrills/Booklayout/design`

This has allowed us to package our template files in the same folder as our PHP files, and save you from a lot of copy/paste

Let's open up the URL that corresponds to the Layoutdemo controller

```
#URL:    http://magento.example.com/nofrills_booklayout/layoutdemo
#File:   app/code/local/Nofrills/Booklayout/controllers/LayoutdemoController.php
```

You should see a browser screen that looks like *Figure 1.4*



It was the best of times, it was the BLURST?! of times?

Figure 1.4

If you view the source of this page, you'll see we have a full (if very basic) HTML page structure. Let's take a look at the code we used to create the layout and blocks necessary to pull this off.

```
#File: app/code/local/Nofrills/Booklayout/controllers/LayoutdemoController.php
class Nofrills_Booklayout_LayoutdemoController
extends Mage_Core_Controller_Front_Action
{
    public function _initLayout()
    {
        $layout = Mage::getSingleton('core/layout');
```

```
$layout->addOutputBlock('root');

$additional_head = $layout->createBlock(
    'nofrills_booklayout/template','additional_head')
->setTemplate('simple-page/head.phtml');

$sidebar = $layout->createBlock('nofrills_booklayout/template','sidebar')
->setTemplate('simple-page/sidebar.phtml');

$content = $layout->createBlock('core/text_list', 'content');

$root = $layout->createBlock('nofrills_booklayout/template','root')
->setTemplate('simple-page/2col.phtml')
->insert($additional_head)
->insert($sidebar)
->insert($content);

return $layout;
}

public function indexAction()
{
    $layout = $this->_initLayout();

    $text = $layout->createBlock('core/text','words');
    $text->setText('It was the best of times, it was the BLURST?! of times?');

    $content = $layout->getBlock('content');
    $content->insert($text);

    $layout->setDirectOutput(true);
    $layout->getOutput();

    exit;
}
}
```

Again, we have some new concepts we'll need to cover here.

1.12 Initializing the Layout and Setting Content

The first thing you'll notice is the `_initLayout` layout method. We're using this controller method to setup a base layout object that has common components (like navigation, some `<head>` HTML, etc.) defined. This allows many different controller methods to share the same basic layout, without us having to rewrite the setup code every time. Instead, all each action would need to do is call

```
$layout = $this->_initLayout();
```

and a base/shared layout would already be created.

1.13 Insert vs. Set

You also probably noticed we're not using the `setChild` method to add blocks to our layout. Instead, we're using the `insert` to add child blocks to other blocks.

```
->insert($additional_head)
```

The `insert` method is the preferred method for adding child blocks to an existing block. There's a bit of redundancy in the `setChild` method, as it requires you to pass in a name for your block.

```
$block->setChild('block_name', $block);
```

However, the `insert` method will automatically use the name you set when you created it (the following code created a block named "sidebar")

```
$layout->createBlock('nofrills_booklayout/template','sidebar');
```

There are a few other problems with using `setChild` in a public context; as of CE 1.4.2 it still doesn't add blocks to the internal `_sortedBlocks` array, which will cause problems down the road, (see Chapter 5 for more information).

Stick with `insert` method and you'll be a happy camper.

1.14 Getting a Reference and Text List

So, the `initLayout` method serves as a central location for instantiating a base layout object. Templates are set, and a complete layout object with an empty "content" node is returned. We'll want to turn our attention to the following code.

```
$text = $layout->createBlock('core/text','words');
$text->setText('It was the best of times, it was the BLURST?! of times?');

$content = $layout->getBlock('content');
$content->insert($text);
```

The first two lines should look familiar. We're creating a simple `core/text` (`Mage_Core_Block_Text`) block that looks like it was written by an infinite number of monkeys - 1. The next set of lines is far more interesting. The `getBlock` method allows you to re-obtain a reference to any block that's been added to the layout (including those added using `createBlock`).

What this code does is get a reference to the content block that was added in `initLayout`, and then **add** our new content block to it. The `getBlock` method is what allows us to centralize the creation of a general layout, and then customize it further for any specific action's needs.

Let's look back up at the creation of our block named content.

```
$content = $layout->createBlock('core/text_list', 'content');
```

You'll notice we used the class alias `core/text_list` here, which corresponds to the class `Magento_Core_Block_Text_List`. Text list blocks have a slightly deceptive name, as you don't set their text. Instead, what a `core/text_list` block does is **automatically** render **all** child blocks that have been added to it.

This feature of the `core/text_list` block is what allows us to add a block named content and just start inserting blocks into it. Any block we add will be automatically rendered.

1.15 A Recap and a Dilema

Look back one last time at our definition of a Layout

A Layout is a collection of blocks in a tree structure

We appear to have covered everything a layout is. We know what a block is, we know how to create a nested structure of blocks, and we now understand how the Layout object provides command and control for the entire show. We've also seen how a generic layout can be built, and then added to depending on our needs.

However, by answering these questions, we've created a new one. **How** should Magento create the layouts needed for each page? Consider our example code above where we abstracted the creation of the Layout object to a `_initLayout` method. This made sense for the tutorials, but Magento core code contains over 180 controllers. If we put layout instantiation in each controller, that means anytime the core team wanted to make a change to the base layout we'd need to update over 180 files. These different `_initLayout` functions would inevitably start to differ in slight ways, eventually causing incompatibility.

The next choice would be to create a separate, centralized, master Layout object for the base layout. Core system programmers could then get a reference to the object, and add to it as need be. This solves some problems, but creates a situation where we're either relying on system programmers whenever designers need to change something, or letting designers into core system code to change highly abstracted PHP code they may not understand. While services based agencies have long used designer/coders and coder/designers, this metaphor hasn't penetrated as deeply in the computer science world, which prefers a layer of separation between the two worlds.

Magento's solution to this situation was to create a system where designers could **configure** what layout they wanted for any particular URL request in Magento. This is the Layout XML system that many of you are already familiar with, and the system that we'll be diving into in our next chapter.

CHAPTER 1. BUILDING LAYOUTS PROGRAMMATICALLY

Visit <http://www.pulsestorm.net/nofrills-layout-chapter-one> to join the discussion online.

Chapter 2

XML Page Layout Files

The core problem the Magento Layout XML system sets out to solve is

How do we allow designers and theme developers to configure a layout, but still offer them full control over the HTML output if they need/want it

So why XML? XML gets used for a lot of things. If you've been doing web development for a while you probably think of XML as a generic document format. While that's **one** of the things XML can be used for, software engineers and computer scientists have other uses for it.

The Magento XML Layout format is less a document format, and more a mini-programming language. The Magento core team created a special format of XML files that fully describes the process of creating blocks programmatically that was described in the previous chapter. There is no public schema or DTD for this dialect of XML, but don't worry, by the time we're through with this chapter you'll have the format down cold.

The Magento Layout XML System is made up of multiple independent pieces. It may seem like some of what we're doing is more of a hassle than just using PHP to create our blocks. However, once you've seen all the pieces, and how those pieces fit together, the advantages of the system should be apparent.

All of which is a fancy way of saying, "Hang in There". This is new, this is different than what you're used to, but it's no harder than any other web development you've learned before.

Finally, while not 100% necessary, the content of this chapter assumes you've been through Chapter 1. Even if you're a master at creating blocks programmatically, you may want to skim through the previous chapter before venturing on.

2.1 Hello World in XML

We'll be working in `UpdateController.php` in this chapter, which may be accessed at the following URL/file

```
http://magento.example.com/nofrills_booklayout/update  
app/code/local/Nofrills/Booklayout/controllers/UpdateController.php
```

The first type of XML tree Magento uses is called the Page Layout. We say tree instead of file, as the Page Layout XML is normally generated on the fly. We're going to create a few Page Layouts manually to get an idea of how they work.

In the previous chapter, we created a Hello World block with a class alias of `nofrills_booklayout/helloworld` (corresponding to the class `Nofrills_Booklayout_Block_Helloworld`). Let's start by creating a Page Layout that uses this block.

First, here's the XML we'll use

```
<layout>  
    <block type="nofrills_booklayout/helloworld" name="root" output="toHtml" />  
</layout>
```

We have an XML node with a root node named `layout`. Within the root node is a single block node with three attributes; `type`, `name`, and `output`.

The **type** attribute is where we specify the class alias of the block we'd like to instantiate. The **name** attribute allows us to set a name for the block which can be used later to get a reference. The `output="toHtml"` attribute/value pair tells the layout system that **this** is the block which should start output.

The Page Layout XML above is roughly equivalent to the following PHP code

```
$layout = new Mage::getSingleton('core/layout');  
$layout->createBlock('nofrills_booklayout/helloworld', 'root');  
$layout->addOutputBlock('root', 'toHtml')
```

You'll notice we've passed in a second parameter ('toHtml') to the `addOutputBlock` method. This is an optional parameter that tells the layout object **which method** on the output block should be used to kick off output. If you look at its definition, it normally defaults to `toHtml`

```
public function addOutputBlock($blockName, $method='toHtml')  
{  
    // $this->_output[] = array($blockName, $method);  
    $this->_output[$blockName] = array($blockName, $method);  
    return $this;  
}
```

In practice you'll never set this optional parameter, but we're including it here to make it more clear what the output attribute in the XML node above is doing

2.2 An Interesting use of the Word Simple

Let's load our XML into the layout object and use it to generate our output. Edit the `indexAction` method in `UpdateController.php` so it matches the following

```
public function indexAction()
{
    $layout = Mage::getSingleton('core/layout');
    $xml = simplexml_load_string('<layout>
        <block type="nofrills_booklayout/helloworld"
            name="root" output="toHtml" />
    </layout>', 'Mage_Core_Model_Layout_Element');

    $layout->setXml($xml);
    $layout->generateBlocks();
    echo $layout->setDirectOutput(true)->getOutput();
}
```

Load the code above in a browser at

`http://magento.example.com/nofrills_booklayout/update`

and you should see your Hello World block.

The first thing that may look a little unfamiliar about the code above is the fragment that creates our simple XML object.

```
$xml = simplexml_load_string('<layout>
<block type="nofrills_booklayout/helloworld" name="root" output="toHtml" />
</layout>', 'Mage_Core_Model_Layout_Element');
```

You may have never seen a SimpleXML node created with that second parameter

`Mage_Core_Model_Layout_Element`

One of SimpleXML's lesser known features is the ability to tell PHP to use a user defined class to represent the nodes. By default, a SimpleXML node is a object of type `SimpleXMLElement`, which is a PHP built-in. By using the syntax above, the Magento core code is telling PHP

Make our simple XML nodes objects of type `Mage_Core_Model_Layout_Element` instead of type `SimpleXMLElement`

If you look at the inheritance chain, you can see that the `Mage_Core_Model_Layout_Element` class has `SimpleXMLElement` as an ancestor.

```
class Mage_Core_Model_Layout_Element extends Varien_Simplexml_Element {...}
class Varien_Simplexml_Element extends SimpleXMLElement {...}
```

So, the Magento provided class name extends `SimpleXMLElement`. That means all normal SimpleXML functionality is preserved.

If you tried to use `setXml` with a normal `SimpleXMLElement`, you'd end up with an error that looks something like this

Prepared for Compaa Peruana de E-commerce S.A.; Copyright ©2011 Pulse33 Storm LLC

```
Recoverable Error: Argument 1 passed to Varien_Simplexml_Config::setXml()
must be an instance of Varien_Simplexml_Element, instance of
SimpleXMLElement given
```

That's because Magento uses PHP's type hinting features to ensure that a normal SimpleXMLElement based object can't be used.

```
//notice the Varien_Simplexml_Element type hinting
public function setXml(Varien_Simplexml_Element $node)
{
    ...
}
```

This is another example of Magento's object oriented system design. Some of you are probably thinking "**That's nuts! Why would you want to do this?**" By providing a custom class here, we gain the ability to add custom methods to any XML node. For example, if we were using the default SimpleXMLElement node, every time we wanted to grab a block's name attribute we'd need to do something like this

```
$tagName = (string)$node->getName();
if ('block'!=$tagName && 'reference'!=$tagName || empty($node['name'])) {
    $name = false;
}
$name = (string)$node['name'];
```

Using the SimpleXML custom class feature, we can define a method on our class to do this for us

```
public function getBlockName()
{
    $tagName = (string)$this->getName();
    if ('block'!=$tagName && 'reference'!=$tagName || empty($this['name'])) {
        return false;
    }
    return (string)$this['name'];
}
```

and then use it wherever we want, resulting in cleaner end-user code which is easier to read and understand

```
$name = $node->getBlockName();
```

If you're not convinced, a little paraphrased Tennyson might help you along the way

Ours is not to question why/Ours is but to do or die

2.3 Adding the XML, Generating the Blocks

So, that little foray in lesser known PHP features complete, the next bit is pretty straight forward. Our Layout object is responsible for managing our Page Layout XML

```
$layout = Mage::getSingleton('core/layout');
```

Page Layout XML is one of its jobs. After getting a reference to the Layout object, we set our newly created simple XML object

```
$layout->setXml($xml);
```

Next, we tell the Layout object to use the Page Layout XML to generate the needed block objects

```
$layout->generateBlocks();
```

This doesn't create any output. When you call the `generateBlocks` method, it goes through your Page Layout XML and creates all the PHP block objects that are needed to generate your layout. The Page Layout XML **configures** which blocks are used as well as the parent/child relationships between those blocks.

It's not until we call

```
echo $layout->setDirectOutput(true)->getOutput();
```

that the `toHtml` method is called and rendering begins.

2.4 Getting a Little More Complex

Let's take a look at a layout that's a bit more complex. Create a new action in the `UpdateController.php` file, and load its corresponding URL

```
#URL: http://magento.example.com/nofrills_booklayout/update/complex
public function complexAction()
{
    $layout = Mage::getSingleton('core/layout');
    $path   = Mage::getModuleDir('', 'Nofrills_Booklayout') . DS .
        'page-layouts' . DS . 'complex.xml';
    $xml = simplexml_load_file($path,
        Mage::getConfig()->getModelClassName('core/layout_element'));
    $layout->setXml($xml);
    $layout->generateBlocks();
    echo $layout->setDirectOutput(true)->getOutput();
}
```

Before we get into the Layout XML itself, there's two new things going on here, both related to how we're loading our XML. First,

```
$path   = Mage::getModuleDir('', 'Nofrills_Booklayout') . DS .
    'page-layouts' . DS . 'complex.xml';
$xml = simplexml_load_file($path,
    Mage::getConfig()->getModelClassName('core/layout_element'));
```

you'll notice we're loading our Page Layout XML from a file rather than passing in a string. This isn't necessary, but will make it easier for us to examine/add-to the XML.

The second thing you'll notice is we've replaced the hard coded XML element class

```
Mage_Core_Model_Layout_Element
```

with a call to

```
Mage::getConfig()->getModelClassName('core/layout_element')
```

While current versions of Magento use a `Mage_Core_Model_Layout_Element`, it's possible that a future version may change this. Because of that, Magento engineers store and read this class name from a config file. When possible, it's best to follow the same conventions you see in Magento core code to ensure maximum compatibility with future versions. Again, this is something you won't need to concern yourself with while **using** the Layout system, rather it's something you'd want to understand if you're working on extending it.

Alright! Let's take a look at the layout we just rendered and the XML that created it, (see *Figure 2.1*)

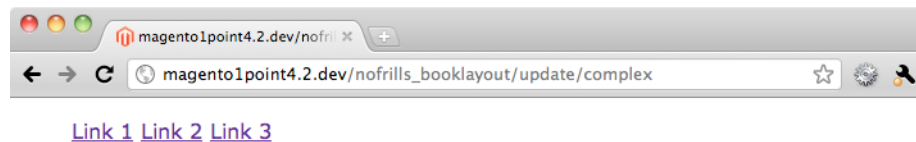


Figure 2.1

If you look at the `complex.xml` file (bundled with the Chapter 2 module code),
`app/code/local/Nofrills/Booklayout/page-layouts/complex.xml`

you'll see the following

```
<layout>
  <block type="nofrills_booklayout/template" name="root"
    template="simple-page/2col.phtml" output="toHtml">
    <block type="nofrills_booklayout/template" name="additional_head"
```

```

        template="simple-page/head.phtml" />

        <block type="nofrills_booklayout/template" name="sidebar">
            <action method="setTemplate">
                <template>simple-page/sidebar.phtml</template>
            </action>
        </block>

        <block type="core/text_list" name="content" />

    </block>
</layout>

```

Lots of new and interesting things to discuss here. The first thing you'll notice is that we've added some sub-nodes to our parent block, as well as introduced a new attribute named `template`.

```

<block type="nofrills_booklayout/template" name="root"
template="simple-page/2col.phtml" output="toHtml">
    ...
</block>

```

You'll remember that a `nofrills_booklayout/template` block is our version of Magento's `core/template` block. When your block is a template block, you can specify which template it should use in the `template` attribute

```

template="simple-page/2col.phtml"

```

When you nest blocks in a Page Layout XML tree, it's the equivalent of using the `insert` method when you're creating them programmatically. The node structure of the XML mirrors the parent/child relationships you were previously setting up programmatically.

Depending on how well you're following along (and if you've taken a few days off to digest everything and/or drink heavily), you may be wondering why it's only the top level node that has a `output` attribute. How does Magento know how to render the sub-blocks? The answer, of course, is in your `simple-page/2col.phtml` template.

```

File: app/code/local/Nofrills/Booklayout/design/simple-page/2col.phtml
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title></title>
    <?php echo $this->getChildhtml('additional_head'); ?>
</head>
<body>
    <?php echo $this->getChildhtml('sidebar'); ?>
    <section>
        <?php echo $this->getChildhtml('content'); ?>
    </section>
</body>
</html>

```

The phtml template files don't care how their parent blocks have been instantiated, they'll function the same regardless of whether they've been created with PHP code or XML code. The `simple-page/2col.phtml` template is still looking for a child block named (in this example) `additional_head`. That's why it's important that all your sub block `<block/>` elements have names

```
name="additional_head"
name="sidebar"
name="content"
```

2.5 Action Methods

Another new node is the `<action/>` node. Let's take a look at the sidebar block

```
<block type="nofrills_booklayout/template" name="sidebar">
    <action method="setTemplate">
        <template>simple-page/sidebar.phtml</template>
    </action>
</block>
```

Here you'll see we're still using a template block, but we've left off the `template` attribute. Instead, we've added a sub-node named `<action/>`.

An `<action/>` node will allow you to call methods on the block which contains it. The above node is equivalent to the following PHP code

```
$layout = Mage::getSingleton('core/layout');
$block = $layout->createBlock('nofrills_booklayout/template','sidebar');
$block->setTemplate('simple-page/sidebar.phtml');
```

You can call **any** public method on a block this way, although some methods won't have any meaning when called from XML. Here we've used it as an alternate method of setting a template, but the Magento core themes are filled with other practical examples. Consider the `page/html.head` blocks (`Mage_Core_Block_Html_Head`). They contain a number of methods for adding CSS and Javascript files to your page

```
<action method="addCss"><stylesheet>css/styles.css</stylesheet></action>
<action method="addJs"><script>lib/ccard.js</script></action>
```

We'll cover the `<action/>` node in greater depth later on in Chapter 5. You also may be interested in Appendix D, which contains a full list, in XML format, of what actions may be called from what blocks.

2.6 References and the Importance of text_lists

To review: We've rendered out our blank page template again, but this time with XML. Let's add some content to it. Edit your `complexAction` method so it matches the following

Prepared for Compaa Peruana de E-commerce S.A.; Copyright ©2011 Pulse38 Storm LLC

```

public function complexAction()
{
    $layout = Mage::getSingleton('core/layout');
    $path   = Mage::getModuleDir('', 'Nofrills_Booklayout') . DS .
        'page-layouts' . DS . 'complex.xml';
    $xml = simplexml_load_file($path,
        Mage::getConfig()->getModelClassName('core/layout_element'));
    $layout->setXml($xml);

    $text = $layout->createBlock('core/text','foxy')
        ->setText("The quick brown fox jumped over the lazy dog.");

    $layout->generateBlocks();
    $layout->getBlock('content')->insert($text);

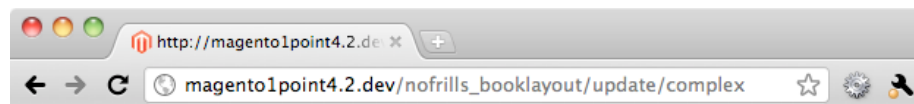
    echo $layout->setDirectOutput(true)->getOutput();
}

```

Just as we were able to in the previous chapter, we obtained a reference to the content block, and inserted a new text block for the page. It's important to note that we couldn't do this **before** we'd called `generateBlocks`. If we tried to, we'd get an error along the lines of

Call to a member function `insert()` on a non-object

because we can't get a reference to a block before it's been created. Reload the page and you'll see our new content, (see *Figure 2.2*)



[Link 1](#) [Link 2](#) [Link 3](#)

The quick brown fox jumped over the lazy dog.

Figure 2.2

Of course, now we're back to adding things to the layout via PHP. Wouldn't it be nice if there was a way to get references to blocks via the Page Layout XML? As you might have guessed by our overtly rhetorical tone, The Page Layout XML offers just such capabilities. At the top level of `complex.xml` add the node named `<reference/>` below and give your page a refresh.

```
<layout>
  <block type="nofrills_booklayout/template" name="root"
    template="simple-page/2col.phtml" output="toHtml">

    <block type="nofrills_booklayout/template" name="additional_head"
      template="simple-page/head.phtml" />

    <block type="nofrills_booklayout/template" name="sidebar">
      <action method="setTemplate">
        <template>simple-page/sidebar.phtml</template>
      </action>
    </block>

    <block type="core/text_list" name="content" />

  </block>

  <reference name="content">
    <block type="core/text" name="goodbye">
      <action method="setText">
        <text> The lazy dog was only faking it. </text>
      </action>
    </block>
  </reference>
</layout>
```

Voila! Another node added to content.

The `<reference/>` tag is the other tag that's valid at the top level of a Page Layout XML `<layout/>` node. It allows you to get a reference to an existing, named node in the layout. Placing blocks **inside** the `<reference/>` node is the equivalent of inserting them. So, the above Page Layout XML reference is equivalent to the following PHP

```
$layout      = Mage::getSingleton('core/layout');
$content     = $layout->getBlock('content');
$text       = $layout->createBlock('core/text', 'goodbye')
->setText(' The lazy dog was only faking it. ');
$content->insert($text);
```

Now's a good time to remind you that it's the `core/text_list` node that makes this insert/auto-render process work. If we were to get a reference to the top level `root` node and insert a block, that block wouldn't be rendered unless the `root` block's template explicitly rendered it. A `core/text_list` block, on the other hand, will **automatically** render any block inserted into it. This difference in

rendering between `core/text_list` and `core/template` blocks is the biggest reason for head scratching layout problems I've seen in the field.

2.7 Layout Updates

So, we've reached a waypoint in our journey to the depths of Magento's Layout XML system. We now know how to create individual XML trees which can be used to generate a page layout. However, it seems like we've swapped one problem for the another. Instead of having to worry about multiple PHP scripts for each page in our site, now we need to worry about multiple XML files. We've moved laterally, but haven't made much progress on the core problem.

And what good is that reference tag? It seems like it'd be easier just to add content directly to the block structure.

This brings us to the next piece of the Magento Layout puzzle: Layout Updates.

2.8 What's an Update

Updates are fragments of XML that are added to a layout object one at a time. These fragments are then processed for special instructions and combined into a Page Layout XML tree. The Page Layout XML tree (which we covered in the first half of this chapter) then renders the page.

By allowing us to build Page Layouts using these chunks of XML, Magento encourages splitting layouts up into logical components which can then be used to build a variety of pages. If that was a bit abstract and hard to follow, our code samples should clear things up.

We'll rely on our trusty hello world block to lead the way. Add the following action to our `UpdateController.php` file.

```
#http://magento.example.com/nofrills_booklayout/update/helloUpdates
public function helloUpdatesAction()
{
    $layout          = Mage::getSingleton('core/layout');
    $update_manager = $layout->getUpdate();
    $update_manager->addUpdate( '<block
    type="nofrills_booklayout/helloworld"
    name="root"
    output="toHtml" />');
    $layout->generateXml();
    $layout->generateBlocks();
    echo $layout->setDirectOutput(true)->getOutput();
}
```

Load the page, and you'll once again see your hello world block.

The three new lines we're interested in above are

```
$update_manager = $layout->getUpdate();
$update_manager->addUpdate('<block
type="nofrills_booklayout/helloworld"
name="root"
output="toHtml" />');
$layout->generateXml();
```

These replace the manual loading of our page layout that we did above. First, a Layout object contains a reference to a `Mage_Core_Model_Layout_Update` object. This object is responsible for managing and holding the individual XML chunks that we're calling updates.

2.8.1 What's a "Model"

You may be wondering why both the Layout and this new Update Manager objects are models, even though they don't read/write to/from a database. If you've used PHP based MVC systems in the past, you've probably become accustomed to the idea that a `Model` is an object that represents a table of data in a SQL database, or perhaps even **multiple tables**. While that's become one common understanding of the term, the original meaning of Model in MVC was the computer science term *Domain Model*.

The Domain Model is an abstract concept. It's where you describe the concepts and vocabulary of the problems you're trying to solve in code. It's sometimes referred to as business logic, or the objects that you use when writing business logic code.

The "Un-Domain Model" portions of a project are things like the code that runs your controller dispatching, or the code that renders a template. This is code you might use on any projects for any number of companies, each with their own Domain Model.

Another way of thinking about this might be a school. Teachers, students, classes, which classes are in each room; these things are all the Domain Model of a School. The non Domain Model would then be the school building itself, its plumbing and boiler, etc.

We mention this here because much of the Magento model layer can be thought of in the more recent, "Models are data in a database way". The layout and update hierarchy, however, cannot. A layout and an update object are both models in the Domain Model sense of the word. They are modeling the "business rules" of creating HTML pages. This can be particularly confusing with the update object, as a single update object will be used to manage multiple Layout Update XML fragments. That's why we're calling this object an Update Manager

```
$update_manager = $layout->getUpdate();
```

2.9 Adding our Updates

So, another little detour into Computer Science 101 out of the way, and we're left with the following two lines

```
$update_manager->addUpdate('<block
type="nofrills_booklayout/helloworld"
name="root"
output="toHtml" />');
$layout->generateXml();
```

Here we're adding a single XML update that is our hello world block. Once we've done that, we then tell our Layout object to generate its own Page Layout XML tree. You may be a little confused, as it appears we've never told our **layout object** about the the updates. Remember, this is object oriented programming. Our update object is already a part of the Layout object. When we said

```
$update_manager = $layout->getUpdate();
```

we got a reference to the update object, but it's **still a part of the layout object**. So when we add a chunk of XML via the Update object, the Layout automatically knows about it.

Our call to the `generateXml` method is roughly equivalent to our previous call that looked like

```
$layout->setXml($xml);
```

When you tell a layout object to generate its XML, it will

1. Combine all the chunks of Update XML into a single tree by concatenating them under a top level `<layout>` node
2. Do some additional processing of the nodes (see "Removing Blocks" below)
3. Set this new tree as the Layout's XML. In other words, set it as the Page Layout XML.

2.10 Fully Armed and Operational References

In this context, references start to make more sense. Let's take a look at `ReferenceController.php` to see some more examples.

```
#File: app/code/local/Nofrills/Booklayout/controllers/ReferenceController.php
#URL: http://magento.example.com/nofrills_booklayout/reference
class Nofrills_Booklayout_ReferenceController
extends Mage_Core_Controller_Front_Action
{

    /**
     * Use to set the base page structure
```

```

    */
    protected function _initLayout()
    {
        $path_page = Mage::getModuleDir('', 'Nofrills_Booklayout') . DS .
            'page-layouts' . DS . 'page.xml';
        $xml = file_get_contents($path_page);
        $layout = Mage::getSingleton('core/layout')
            ->getUpdate()
            ->addUpdate($xml);
    }

    /**
     * Use to send output
     */
    protected function _sendOutput()
    {
        $layout = Mage::getSingleton('core/layout');

        $layout->generateXml()
            ->generateBlocks();

        echo $layout->setDirectOutput(false)->getOutput();
    }

    public function indexAction()
    {
        $this->_initLayout();
        $this->_sendOutput();
    }
}

```

If you load the above URL, you'll get our basic, but complete, page layout from previous examples.

First off, let's cover a slight change in our approach. There's two protected methods on this controller

1. `_initLayout`
2. `_sendOutput`

The `_initLayout` method we've used before. This is where we'll setup a base Layout object, to which our primary controller action can add blocks. We're also loading up a new file, `page.xml` (included with the Chapter 2 module).

The `_sendOutput` method centralizes the code we've been using to render a layout object once we're done manipulating it. By centralizing these functions, all we need to do in our controller action is something like

```

public function indexAction()
{
    $this->_initLayout();
    //...add additional updates here...
    $this->_sendOutput();
}

```

Before we get deep into that, let's take a look at the code that's loading our layout in `_initLayout`

```
protected function _initLayout()
{
    $path_page = Mage::getModuleDir('', 'Nofrills_Booklayout') . DS .
        'page-layouts' . DS . 'page.xml';
    $xml = file_get_contents($path_page);

    $layout = Mage::getSingleton('core/layout')
        ->getUpdate()
        ->addUpdate($xml);
}
```

Here you can already see some of the efficiencies that updates have brought us. We no longer need to worry about creating/adding the right type of simple XML object. We can store our base XML fragment in a file,

```
<!-- #File: app/code/local/Nofrills/Booklayout/page-layouts/page.xml -->
<block type="nofrills_booklayout/template" name="root"
template="simple-page/2col.phtml" output="toHtml">
    <block type="nofrills_booklayout/template" name="additional_head"
        template="simple-page/head.phtml" />

    <block type="nofrills_booklayout/template" name="sidebar">
        <action method="setTemplate">
            <template>simple-page/sidebar.phtml</template>
        </action>
    </block>

    <block type="core/text_list" name="content" />
</block>
```

and then just pass it to the update object as a string. You'll notice there's no surrounding `<layout>` node for Layout Update XML **fragments**. Instead, we pass in the block nodes we want at the top level of our eventual Page Layout file.

So, with the basic page structure for our layout set, we're ready to add in our custom blocks. It's only now that reference blocks show their true power. Consider the `indexAction`, and then load up the controller URL

```
#URL: http://magento.example.com/nofrills_booklayout/reference
public function indexAction()
{
    $this->_initLayout();
    Mage::getSingleton('core/layout')
        ->getUpdate()
        ->addUpdate('<reference name="content">
            <block type="core/text" name="our_message">
                <action method="setText"><text>Here we go!</text></action>
            </block>
        </reference>');
    $this->_sendOutput();
}
```

You should see the content area with the text "Here we go!".

What `<reference/>` nodes allow us to do is **alter** elements that have already been added to a layout elsewhere. This allows us to write our structural Page Layout XML once, and then have different controller actions insert the different blocks they need.

Next, try adding the following methods to the controller

```
protected function _loadUpdateFile($file)
{
    $path_update = Mage::getModuleDir('', 'Nofrills_Booklayout') . DS .
        'content-updates' . DS . $file;

    $layout = Mage::getSingleton('core/layout')
        ->getUpdate()
        ->addUpdate(file_get_contents($path_update));
}

#URL: http://magento.example.com/nofrills_booklayout/reference/fox
public function foxAction()
{
    $this->_initLayout();
    $this->_loadUpdateFile('fox.xml');
    $this->_sendOutput();
}
```

The `_loadUpdateFile` method will load an XML Update from our module's "content-updates" folder. This allows us a simple three line controller action to load up content for any particular controller action/URL. Consider these other actions, `ceaser` and `dog`

```
#URL: http://magento.example.com/nofrills_booklayout/reference/dog
public function dogAction()
{
    $this->_initLayout();
    $this->_loadUpdateFile('dog.xml');
    $this->_sendOutput();
}

#URL: http://magento.example.com/nofrills_booklayout/reference/ceaser
public function ceaserAction()
{
    $this->_initLayout();
    $this->_loadUpdateFile('ceaser.xml');
    $this->_sendOutput();
}
```

We could even take this a step further. Consider the following method in place of `_loadUpdateFile`.

```
protected function _loadUpdateFileFromRequest()
{
    $path_update = Mage::getModuleDir('', 'Nofrills_Booklayout') . DS .
        'content-updates' . DS . $this->getFullActionName() . '.xml';
```

```

        $layout = Mage::getSingleton('core/layout')
        ->getUpdate()
        ->addUpdate(file_get_contents($path_update));
    }

```

and an adjustment made in the `foxAction` method.

```

#URL: http://magento.example.com/nofrills_booklayout/reference/fox
public function foxAction()
{
    $this->_initLayout();
    $this->_loadUpdateFileFromRequest();
    $this->_sendOutput();
}

```

Load the `foxAction` URL, and you'll see a warning something like this.

```

Warning: file_get_contents(/mage/path/app/code/local/Nofrills/Booklayout/content-
updates/nofrills_booklayout_reference_fox.xml)
[function.file-get-contents]: failed to open stream: No such file
or directory

```

The `_loadUpdateFileFromRequest` method attempts to load up an XML update from the file `nofrills_booklayout_reference_fox.xml`. This filename is created using controller method `$this->getFullActionName()`. The "Full Action Name" is a string that combines, via underscores, the lowercase versions of

- Module Name: `nofrills_booklayout`
- Controller Name: `reference`
- Action Name: `fox`

It's essentially a name that allows us to uniquely identify any request that comes into Magento based on these three criteria. Let's create a file for our new method to load

```

<!-- #File: app/code/local/Nofrills/Booklayout/content-updates/
nofrills_booklayout_reference_fox.xml -->
<reference name="content">
    <block type="core/text" name="our_message">
        <action method="setText"><text>
            Magento is a foxy system.
        </text></action>
    </block>
</reference>

```

Reload the page, and you'll see our new content block.

2.11 Removing Blocks

As previously mentioned, when we call the `generateXml` method on the layout object, it does the following

Prepared for Compaa Peruana de E-commerce S.A.; Copyright ©2011 Pulse47 Storm LLC

1. Combines all the chunks of Update XML into a single tree by concatenating them under a top level `<layout>` node
2. Does some additional processing of the nodes
3. Sets this new tree as the Layout's XML. In other words, set it as the Page Layout

So, in our examples above, that means we end up with a Page Layout that looks something like this

```
<layout>
  <!-- update loaded from page.xml -->
  <block type="nofrills_booklayout/template" name="root"
    template="simple-page/2col.phtml" output="toHtml">
    <block type="nofrills_booklayout/template" name="additional_head"
      template="simple-page/head.phtml" />

    <block type="nofrills_booklayout/template" name="sidebar">
      <action method="setTemplate">
        <template>simple-page/sidebar.phtml</template>
      </action>
    </block>

    <block type="core/text_list" name="content" />
  </block>

  <!-- update loaded from nofrills_booklayout_reference_fox.xml -->
  <reference name="content">
    <block type="core/text" name="our_message">
      <action method="setText"><text>
        Magento is a foxy system.
      </text></action>
    </block>
  </reference>
</layout>
```

The step we haven't covered yet is #2

Do some additional processing of the nodes

After concatenating all the updates into a single XML tree, but before assigning that tree as the Page Layout XML, Magento will process the concatenated tree for additional directives. As of Community Edition 1.4.2, the only other directive supported is `<remove/>`.

Let's give the remove directive a try. Alter your `nofrills_booklayout_reference_fox.xml` to include a `<remove/>` tag, as below.

```
<reference name="content">
  <block type="core/text" name="our_message">
    <action method="setText"><text>
      Sidebar? We don't need a sidebar!
    </text></action>
  </block>
</reference>
```

```
<remove name="sidebar" />
```

Reload your URL

```
http://magento.example.com/nofrills_booklayout/reference/fox
```

and you should see a page **without** the block named sidebar, which was rendering our navigation.

2.11.1 Before (*Figure 2.3*)

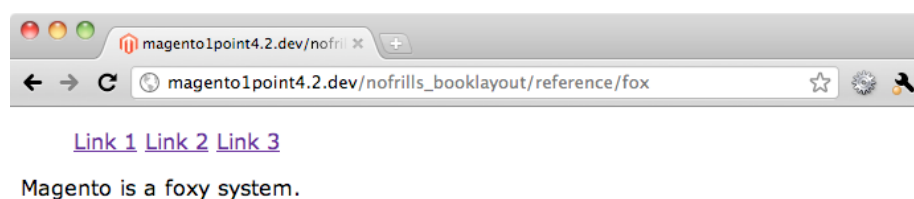


Figure 2.3

2.11.2 After (*Figure 2.4*)

Remove instructions are processed in the `Mage_Core_Model_Layout::generateXml` method. This method

1. Combines all updates with a call to `$xml = $this->getUpdate()->asSimplexml();`
2. Looks through the combined updates for any nodes named `remove`.
3. If it finds a `remove` node, it then takes that node's name and looks for any `block` or `references` nodes with the same name.
4. If it finds any `blocks` or `references`, these nodes are marked with an `ignore` attribute.

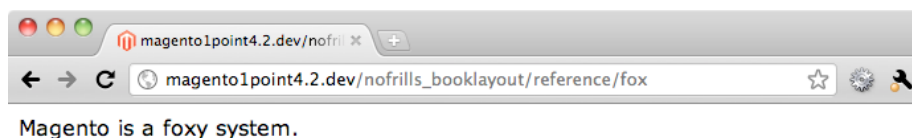


Figure 2.4

5. The `remove` blocks are ignored during the Layout rendering process. Their job is to mark which nodes should be ignored. After that, they're irrelevant

Once the remove instructions have been processed, the resulting tree is set as the Page Layout.

This means in our most recent example we ended up with a Page Layout XML tree that looked exactly the same as before, with one exception

```
<block type="nofrills_booklayout/template" name="sidebar" ignore="1">
```

When a `<block/>` or `<reference/>` has an `ignore="1"` attribute node, the Layout rendering process will **skip** that block. In this way, the block, and all its sub-blocks, are removed from the final rendered page.

2.12 What's Next

So, we've now covered how to create and manage Page Layouts via XML files. We've also explored Magento's "Update" mechanism, which allows us to build up Page Layout XML files via individual Layout Update XML fragments, allowing for modular page layouts.

The final problems we need to solve are

1. How should we store **all** the layout update for our system

2. How should we automatically **load** these layout files into the system

So far we've been using individual `_init` methods in our controllers. While this has offered us more modularity than previous methods, this will still get unwieldy as the number of controllers and actions grows. Plus, there's still the sub-problem of how to create a method of doing this that allows back-end PHP developers and front-end PHP developers the ability to go about their jobs without crossing paths.

The answer to this question, and the final large topic we need to cover, is the Package Layout.

Visit <http://www.pulsetorm.net/nofrills-layout-chapter-two> to join the discussion online.

Chapter 3

The Package Layout

We're finally getting closer to fully understanding the Magento layout rendering process. At the end of the last chapter, we stated that the final puzzle pieces were

1. How should we store **all** the layout update for our system
2. How should we **load** these layout files into the system

The approach Magento has taken is to introduce **another** XML tree, this one called the *Package Layout*. The Package Layout is an XML tree which contains any number of Layout XML Update Fragments. The Package Layout contains **all the updates fragments** that the **entire application** might want to use.

The top level node in the package layout is named `<layouts/>`.

```
<layouts>
  <!-- ... --->
</layouts>
```

Make note of the plural, `layouts`. This is different from the top level singular `<layout/>` node of the Page Layout XML tree you were building in previous chapters. That's because, as mentioned, this is a **different** XML tree.

The second level nodes of the Package Layout are not `blocks`, or `references`, or **any** tag we've seen so far. The second level nodes are something new, called *Handles*. Each handle node contains a single XML Update Fragment.

```
<layouts>
  <handle_name>
    <!-- ... XML Update Fragment --->
  </handle_name>

  <handle_name2>
    <!-- ... XML Update Fragment --->
  </handle_name2>
```

```

<handle_name3>
  <!-- ... XML Update Fragment --->
</handle_name3>

<handle_name>
  <!-- ... XML Update Fragment --->
</handle_name>

<default> <!-- default is an actual handle name from the system-->
  <!-- ... -->
</default>

<catalog_product_send> <!-- another real name -->
  <!-- ... -->
</catalog_product_send>

  <!-- etc. -->
</layouts>

```

Handle names may be repeated, but before we discuss what they mean, let's discuss how they're loaded into the system.

3.1 The Why and Where of the Package Layout

This collection of Layout Update XML nodes is called the Package Layout because it contains every possible Layout Update XML fragment that might be used in a particular design package.

Jumping back a few chapters, you'll remember that Magento stores its theme templates in the following location

```
[BASE DESIGN FOLDER]/[AREA FOLDER]/[DESIGN PACKAGE FOLDER]/[THEME FOLDER]/template
```

Magento also stores its layout files in a similar location.

```
[BASE DESIGN FOLDER]/[AREA FOLDER]/[DESIGN PACKAGE FOLDER]/[THEME FOLDER]/layout
```

Magento will look for layout files in this folder first. If it doesn't find a specific layout file here, it will check the base folder at

```
[BASE DESIGN FOLDER]/[AREA FOLDER]/base/default/layout
```

See Appendix E for more information on the `base` folder.

So, that's the **folder** where layout files are stored. Where does Magento get the name of individual layout files? Every individual code module in Magento has a `config.xml`. In this file, there's a node at

```

<frontend> <!-- frontend is the "area" name. -->
  <layout>

```

```

        <updates>
            <section>
                <file>section.xml</file>
            </section>
            <anysection>
                <file>anysection.xml</file>
            </anysection>
        </updates>
    </layout>
</frontend>

```

On each request, Magento will scan the config for any XML files located in the `<updates>` node. These filenames will be the files Magento will attempt to load up as the Package Layout. The code that does this can be found in

```

app/code/core/Mage/Core/Model/Layout/Update.php
Mage_Core_Model_Layout_Update::getFileLayoutUpdatesXml

```

The actual code in `getFileLayoutUpdatesXml` is pretty dense. However, you can approximate the code that grabs the list of files with something like this

```

#URL: http://magento.example.com/nofrills_booklayout/reference/layoutfiles
public function layoutfilesAction()
{
    $updatesRoot = Mage::app()->getConfig()->getNode('frontend/layout/updates');
    $updateFiles = array();
    foreach ($updatesRoot->children() as $updateNode) {
        if ($updateNode->file) {
            $module = $updateNode->getAttribute('module');
            if ($module &&
                Mage::getStoreConfigFlag('advanced/modules_disable_output/' .
                    $module)) {
                continue;
            }
            $updateFiles[] = (string)$updateNode->file;
        }
    }
    // custom local layout updates file - load always last
    $updateFiles[] = 'local.xml';
    var_dump($updateFiles);
}

```

Load the URL for the above action, and you'll see the list of files that Magento will load, and then combine, into the package layout.

Two additional things to note about the loading of the package layout. First, you'll see in the above code, (which was copied from `Mage_Core_Model_Layout_Update::getFileLayoutUpdatesXml`), that Magento checks for a config flag at `advanced/modules_disable_output` before loading any particular file. This corresponds to the System Config section at

```

System -> Configuration -> Advanced -> Disable Module's Output

```

If you've disabled a module's output through this config section, Magento will ignore loading that module's updates into the Package Layout.

The second thing you'll want to notice is this line

```
// custom local layout updates file - load always last
$updateFiles[] = 'local.xml';
```

After loading XML files found in the configuration, Magento will add a `local.xml` file to the end of the list. This file is where store owners can add their own Layout Update XML fragments to the Package Layout. We'll learn more about this later, but by loading `local.xml` last, Magento ensures any Layout Update XML Fragments here have the final say of what goes into the Layout.

Once Magento has determined which files should be loaded into the Package Layout, the contents of each file will be combined into a single, massive XML tree.

3.2 Package Layout Examples

As part of the module that came with this book, we've included a theme that **clears out** all most of the handles in the default Package Layout. We've done this to provide some clarity in the examples below. Unlike our examples so far, there's no public API for programmatically manipulating the Package Layout once its loaded.

You'll want to switch to this theme now. We've placed this theme in the default package. **IMPORTANT:** Doing this will make every part of your frontend cart produce a blank page. It goes without saying, but bears repeating, don't do this with a production store.

If you go to

```
System -> Configuration -> Design -> Package -> Current Package Name
```

and enter `default` (if it's not already there). Next, go to

```
System -> Configuration -> Design -> Themes -> Layout
```

and enter `nofrills_layoutbook`. Click **Save**, and you'll be set for the example in the next section, (see *Figure 3.1*)

You can find your new "zeroed out" layout files at

```
app/design/frontend/default/nofrills_layoutbook/layout/
```

If you load any page in your store, you'll encounter an empty, blank, and error-less browser screen.

3.3 What is a Handle?

Handles are used to organize the Layout Update XML fragments that your application needs. Every time an HTTP request is sent to the Magento system,

Package

Current Package Name: [STORE VIEW]

[Add Exception](#) [STORE VIEW]

▲ Match expressions in the same order as displayed in the configuration.

Themes

Translations: [STORE VIEW]

Templates: [STORE VIEW]

[Add Exception](#) [STORE VIEW]

▲ Match expressions in the same order as displayed in the configuration.

Skin (Images / CSS): [STORE VIEW]

[Add Exception](#) [STORE VIEW]

Layout: [STORE VIEW]

[Add Exception](#) [STORE VIEW]

Default: [STORE VIEW]

[Add Exception](#) [STORE VIEW]

Figure 3.1

it generates handle names for the request. There are some handle names which are produced on every request. They include the handle `default`, as well as a handle based on the controller's "Full Action Name" that was discussed in the previous chapter.

The Full Action Name, as a reminder, is a combination of the current module name, controller name, and action name.

- Module Name: `nofrills_booklayout`
- Controller Name: `reference`
- Action Name: `fox`

In the Package Layout example above, the

```
<catalog_product_send>...</catalog_product_send>
```

node is an example of a full action name handle for the **send** Action in the **Product** controller of the **Catalog** module.

In general, it's the responsibility of the controller object to set handles for any particular request. Also, you generally don't need to worry about setting your own handles. Magento's base controller methods do this for you. If you want to

see the handles set from a particular action controller, use the following code snippet

```
#URL: http://magento.example.com/nofrills_booklayout/reference/handle
public function handleAction()
{
    $this->loadLayout();
    $handles = Mage::getSingleton('core/layout')->getUpdate()->getHandles();
    var_dump($handles);
    exit;
}
```

You're probably wondering about the call to `$this->loadLayout()`; . Don't worry about it too much for now, we'll get to it soon enough. Just know that that you need to call this method before being able to get a list of handles for a particular request.

3.4 Rendering a Magento Layout

So, we've finally arrived at the point where we have the vocabulary to fully explore how a Magento layout is created and then rendered for each request. The rest of this chapter will explain that process in full. From a high level, here's what happens

1. If it's not already cached, Magento loads the entire package layout into memory (from the individual XML file already discussed)
2. In the controller, the `loadLayout` method is called
3. In `loadLayout`, Magento generates a list of "handles" for the request
4. In `loadLayout`, Magento takes this list of handles, and uses them to search the Package Layout for a list of Layout XML Update Fragments
5. In `loadLayout`, after fetching a list of Layout XML Update fragments, Magento checks those fragments for `<update name=""/>` tags. If it finds any, it checks the package layout for any additional handles which match this tag
6. In `loadLayout`, the Layout Update XML Fragments found in steps four and five are combined. This combined XML tree is now the Page Layout
7. Magento uses the Page Layout to instantiate the needed block objects
8. In the controller, the `renderLayout` method is called. This kicks off rendering of the Layout via its `getOutput` method. The resulting output is added to a Magento Response object.

Let's take a look at some concrete code examples. We'll be working in the following controller file.

```
#File: app/code/local/Nofrills/Booklayout/controllers/PackageController.php
class Nofrills_Booklayout_PackageController extends
Mage_Core_Controller_Front_Action
{
    public function loadLayout($handles=null, $generateBlocks=true,
    $generateXml=true)
    {
        $original_results = parent::loadLayout($handles,$generateBlocks,
        $generateXml);

        $handles = Mage::getSingleton('core/layout')->getUpdate()->getHandles();
        echo "<strong>Handles Generated For This Request: ",
        implode(",",$handles), "</strong>";

        return $original_results;
    }

    #http://magento.example.com/nofrills_booklayout/package/index
    public function indexAction()
    {
        $this->loadLayout();
        $this->renderLayout();
    }
}
```

You'll notice we've extended the `loadLayout` method to print out the handles generated by Magento. This is for our own debugging purposes. Load up the index URL, and you should see a blank white page with only the handles listed, (see *Figure 3.2*)

3.5 Getting a Handle on Handles

There's two handles you can always rely on being generated. Those are the handle named `default`, and the handle that's named for the "Full Action Name".

```
default
nofrills_booklayout_package_index
```

Because of this, in the layout files that ship with Magento the handles for a page's structure are kept under the `<default/>` handle. This is also where the "root" tag with the `output="toHtml"` attribute is stored. If you look at a stock `page.xml`, you can see this.

```
<!-- #File: app/design/frontend/base/default/layout/page.xml -->
<default translate="label" module="page">
    <label>All Pages</label>
    <block type="page/html" name="root" output="toHtml"
    template="page/3columns.phtml">

        <block type="page/html_head" name="head" as="head">
            <action method="addJs">
                <script>prototype/prototype.js</script>
```

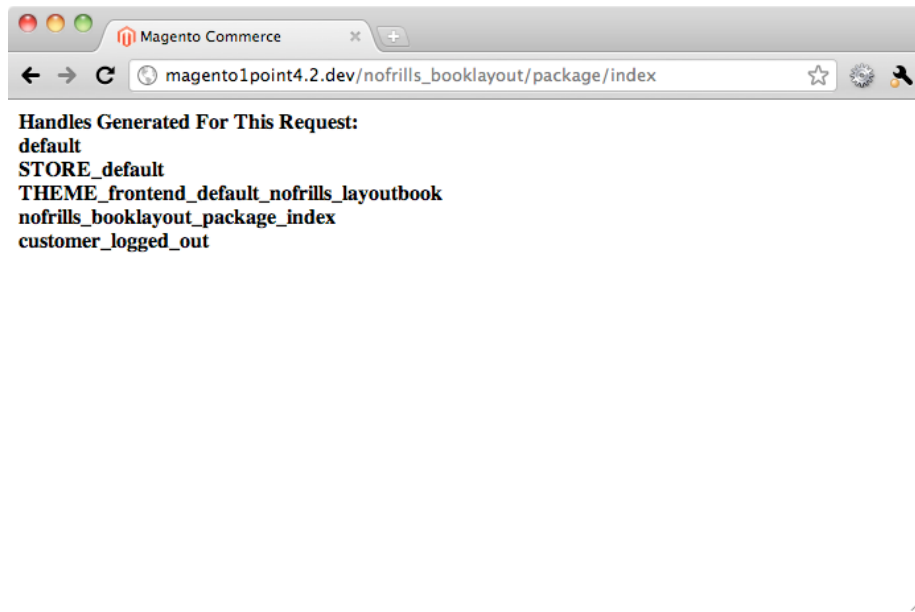


Figure 3.2

```

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

```

Layout Update XML Fragments located in the `default` handle will always be loaded.

If you look at our custom `page.xml`, you'll see we've removed all the handle tags

```

<!-- #File: app/design/frontend/default/nofrills_layoutbook/layout/page.xml -->
<?xml version="1.0"?>
<layout version="0.1.0">
</layout>

```

That's why our page is rendering blank. Let's restore those tags and see what effect it has. We'll copy the base `page.xml` over our blank one.

```

cp app/design/frontend/base/default/layout/page.xml \
app/design/frontend/default/nofrills_layoutbook/layout/page.xml

```

Clear your Magento cache and reload your page. You should now see a base Magento layout, (see *Figure 3.3*)

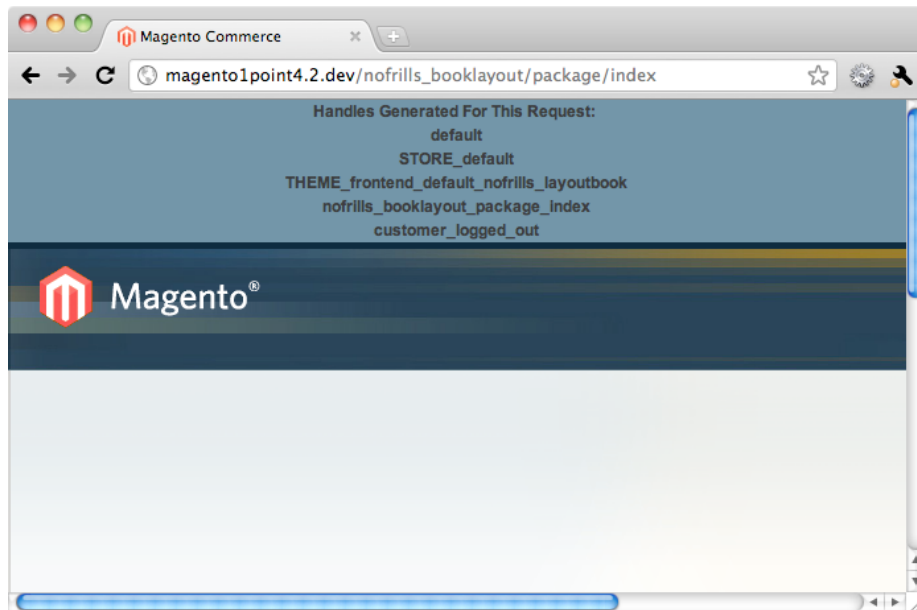


Figure 3.3

You'll want to either turn caching off or clear it between each page reload from here on out, see Appendix F for more information if you're interested in why.

Take a look at the top of `page.xml` and find the node named `root`

```
#File: app/design/frontend/default/nofrills_layoutbook/layout/page.xml
<block type="page/html" name="root" output="toHtml"
template="page/3columns.phtml">
```

Let's edit this to remove the output tag

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

Refresh your page (again, after clearing your cache). The page now renders as blank. You'd probably never do this for a production site, we've done here to demonstrate that the Magento Layout itself is built on the same concepts our simple templates from previous chapters.

Let's restore that output attribute before continuing

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

So, that's the `default` handle. Any Layout Update XML fragments inside a default handle will always be a part of the Page Layout, and that's where most of the structural blocks live. That brings us to the other handle you can always

rely on being present, the Full Action Name handle. In our examples above this is

```
nofrills_booklayout_package_index
```

As previously mentioned, this handle is generated from the module name (`nofrills_booklayout`), the controller name (`package`) and the action name (`index`). The handle will uniquely identify any request into the system. Therefore, Layout Update XML fragments located in this handle are most often used to add content to the page.

Let's add a Layout Update XML fragment using our handle. Open up the `local.xml` file, and paste in the following code.

```
<!-- #File: app/design/frontend/default/nofrills_layoutbook/layout/local.xml -->
<?xml version="1.0"?>
<layout version="0.1.0">
    <nofrills_booklayout_package_index>
        <reference name="content">
            <block type="core/text" name="our_message">
                <action method="setText"><text>Hello Mars</text></action>
            </block>
        </reference>
    </nofrills_booklayout_package_index>
</layout>
```

Inside our `nofrills_booklayout_package_index` node we've added a Layout Update XML fragment to update the content block with a little bit of text. Reload, clear cache, and you can see our simple *Hello Mars* text block has been added to the page. However, if we move to this URL/Action

```
#http://magento.example.com/nofrills_booklayout/package/second
public function secondAction()
{
    $this->loadLayout();
    $this->renderLayout();
}
```

We can see that our text block is, as expected, NOT added. We'd need to add another handle to the Package Layout with its own Layout Update XML fragment for that to happen. Let's do that now.

```
<!-- #File: app/design/frontend/default/nofrills_layoutbook/layout/local.xml -->
<layout version="0.1.0">
    <!-- ... -->
    <nofrills_booklayout_package_second>
        <reference name="content">
            <block type="core/text" name="our_message">
                <action method="setText"><text>Hello Jupiter</text></action>
            </block>
        </reference>
    </nofrills_booklayout_package_second>
    <!-- ... -->
</layout>
```

Notice the new handle's name (`nofrills_booklayout_package_second`) matches the `secondAction` method. Refresh the page (after clearing your cache) and you'll see the Hello Jupiter text.

We can also use the Full Action Handle to change which template an existing block uses. For example, to make this a one column layout, we'll get a reference to the root block and call its `setTemplate` method.

```
<nofrills_booklayout_package_second>
  <reference name="content">
    <block type="core/text" name="our_message">
      <action method="setText"><text>Hello Jupiter</text></action>
    </block>
  </reference>

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

When editing a single Layout XML file, you can either put all your additional tags changes into a single handle, or spread them out. The following would be functionally the same as the above.

```
<nofrills_booklayout_package_second>
  <reference name="content">
    <block type="core/text" name="our_message">
      <action method="setText"><text>Hello Jupiter</text></action>
    </block>
  </reference>
</nofrills_booklayout_package_second>

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

The order the update handles are placed in **is** significant. Consider multiple layout files that try to change a block's template. The last file processed (`local.xml`) will be the one that wins, just like the last method called on a PHP block wins

```
$block->setTemplate('3columns.phtml');
$block->setTemplate('6columns.phtml');
$block->setTemplate('1column.phtml');
```

There's no firm rule in place here, but try not to have your layout action in one group of handles be too dependent on what's happened in another handle.

3.6 More local.xml

Just because we're in `local.xml` doesn't mean we're limited to Full Action Handles. **Any** handle can be added to **any** Layout XML file, as all these files are combined into the Package Layout. For example, we could add a default handle that would ensure the same content always gets added to the page in `local.xml`

```
<layout>
  <!-- ... -->
  <default>
    <reference name="content">
      <block type="core/text" name="for_everyone">
        <action method="setText">
          <text>I am on all pages!</text>
        </action>
      </block>
    </reference>
  </default>

  <!-- ... -->
</layout>
```

3.7 Adding Other Handles to the Page Layout

There's one other tag you'll need to be aware of in the Package Layout. You'll often want to use the same set of blocks over and over again for different Full Action Handles, similar to the way you'd use a simple subroutine or function in a full programming language.

To handle this situation there's an additional tag that the Package Layout understands named `<update/>`.

When Magento is scanning the package layout for Layout Update XML fragments to use, it will do a secondary scan of those fragments for an `<update/>` tag. If it finds one, it will **go back to the entire package layout** and grab any Layout Fragments that match the handle attribute. Consider, based on our example above, the following `local.xml`

```
<layout version="0.1.0">
  <nofrills_booklayout_package_index>
    <reference name="content">
      <block type="core/text" name="planet_4">
        <action method="setText"><text>Hello Mars. </text></action>
      </block>
    </reference>

    <update handle="nofrills_booklayout_package_second" />
  </nofrills_booklayout_package_index>

  <nofrills_booklayout_package_second>
    <reference name="content">
```



```

        <block type="core/text" name="planet_5">
            <action method="setText"><text>Hello Jupiter. </text></action>
        </block>
    </reference>
</nofrills_booklayout_package_second>

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

```

If we loaded our index page here, the Page Layout would contain the following (sans comments)

```

<!-- from our handle -->
<reference name="content">
    <block type="core/text" name="planet_4">
        <action method="setText"><text>Hello Mars. </text></action>
    </block>
</reference>

<!-- from our [update handle="nofrills_booklayout_package_second"/] -->
<reference name="root">
    <action method="setTemplate"><template>page/1column.phtml</template></action>
</reference>
<reference name="content">
    <block type="core/text" name="planet_5">
        <action method="setText"><text>Hello Jupiter. </text></action>
    </block>
</reference>

```

That's because while processing the `nofrills_booklayout_package_index` handle, Magento encountered the `<update/>` tag.

```
<update handle="nofrills_booklayout_package_second" />
```

By including this tag, we've told Magento that we **also** want to grab Layout Update XML fragments that are included in the `nofrills_booklayout_package_second` handle.

You can think of this as a sort of "include" for Layout Update fragments. Magento itself uses this technique extensively. For example, Magento defines the blocks for the `customer_account_login` handle, and then uses those again later on when it wants to include the same login on the multi-shipping checkout page.

```

<checkout_multishipping_login>
    <update handle="customer_account_login"/>
</checkout_multishipping_login>

```

3.8 Package Layout Term Review

Phew! That was a lot of new terminology to take in. Let's close with a quick recap of the structure of our two XML trees, the Package Layout and the Page Layout

3.8.1 Package Layout

The Package Layout is an XML tree that contains all possible Layout XML Update Fragments for a design package. Fragments are organized by handle.

Top Level Node in the Package Layout

- `<layouts>`

Allowed Second Level Nodes

- Any arbitrary named node called a `handle`

Allowed Third Level Nodes

- `<block>` or `<reference>`, as the start of a Layout Update XML fragment
- `<update>`, used to include another handle's Layout XML Update Fragments

3.8.2 Page Layout

The Page Layout is the final collection of Layout Update XML Fragments used to create block objects for a request.

Top Level Node in the Page Layout

- `<layout>`

Allowed Second Level Nodes

- `<block>`
- `<reference>`
- `<remove/>`

Allowed Third Level nodes

- `<block>`'s and `<reference>`'s may contain - other blocks, or actions
- `<remove/>`

Note

- `<remove/>` is technically allowed anywhere, as the xpath expression used to parse it (`//remove`) ends up searching the entire Page Layout. Convention keeps it at the second nesting level

3.8.3 Layout Update XML Fragment

A partial XML document fragment that describes a series of PHP commands to run which may

- Create block objects
- Insert block objects into other block objects
- Reference block objects to call their methods

Visit <http://www.pulesestorm.net/nofrills-layout-chapter-three> to join the discussion online.

Chapter 4

Bringing it All Together

We've just spent the last three chapters reviewing some complicated concepts and **the interaction** of complicated concepts. We're going to pause for a moment to review what we've learned, as well as provide an overall picture of how Magento builds a Page Layout. Original drafts had this brief, mini-chapter at the start of the book, but it made people's head explode. Hopefully it's safe enough to cover now

4.1 How a Magento Layout is Built

Somewhere in a controller action, the programmer creating the controller action method tells Magento that it's time to load the layout. The end result of loading a layout is a Page Layout XML tree, (see *Figure 4.1*)

To load a Page Layout, Magento will pick and choose Layout Update XML fragments from a repository of Layout Update XML fragments. This repository of Layout Update XML fragments is known as the Package Layout. The Package Layout is loaded from disk by combining several XML files into a single tree, (see *Figure 4.2*)

Users of the Magento system can add to the Package Layout by

1. Creating and editing a `local.xml` file
2. Adding a custom XML file to the Layout via a module's `config.xml` file
3. Least desirably, but most commonly, editing or replacing existing Package Layout files in their theme's layout

The Package Layout organizes its many Layout Update XML fragments by handle. During a normal page request life cycle, various parts of the Magento system will tell the Layout Update Manager that, when the time comes, Layout

```
- <layout>
  <block name="formkey" type="core/template" template="core/formkey.phtml"/>
  <label>All Pages</label>
- <block type="page/html" name="root" output="toHtml" template="page/3columns.phtml">
+ <block type="page/html_head" name="head" as="head"></block>
- <block type="core/text_list" name="after_body_start" as="after_body_start" translate="label">
  <label>Page Top</label>
</block>
<block type="page/html_notices" name="global_notices" as="global_notices" template="page/html/notices.phtml"/>
- <block type="page/html_header" name="header" as="header">
  <block type="page/template_links" name="top.links" as="topLinks"/>
  <block type="page/switch" name="store_language" as="store_language" template="page/switch/languages.phtml"/>
- <block type="core/text_list" name="top.menu" as="topMenu" translate="label">
  <label>Navigation Bar</label>
</block>
- <block type="page/html_wrapper" name="top.container" as="topContainer" translate="label">
  <label>Page Header</label>
  - <action method="setElementClass">
    <value>top-container</value>
  </action>
</block>
</block>
<block type="page/html_breadcrumbs" name="breadcrumbs" as="breadcrumbs"/>
- <block type="core/text_list" name="left" as="left" translate="label">
  <label>Left Column</label>
</block>
```

Figure 4.1

Update XML fragments from "handle x" should be loaded. When the Controller Action developer tells Magento to load the layout, the Layout Update Manager checks this list, and asks the Package Layout for a copy of the Layout Update XML fragments contained within those particular handles.

Also, each fetched Layout Update XML fragment is processed at this time for an `<update handle="...">` node. This node can be used to tell the manager to fetch **additional nodes** based on the specified handle.

Finally, a copy of all Layout Update XML fragments in hand, the Layout Update Manager combines them into a single XML tree. This is the Page Layout.

4.2 What is the Page Layout

The Page Layout is a list of instruction for Magento. Programmer types may call it a meta-programing language, or a domain-specific language. Regardless of what you call, the last step of **loading** a Layout is for Magento to use the Page Layout to create and instantiate a nested tree of block objects. These are PHP Objects, each one ultimately responsible for rendering a chunk of HTML.

That's the layout loaded. The controller action programmer may, at this point, choose to manipulate the layout object further. This may include adding, removing, or setting properties on blocks. The Magento Admin Console application does this regularly. The Magento frontend (cart) application tends not to do this. Irrespective of how, after loading a Layout and fiddling with it if they

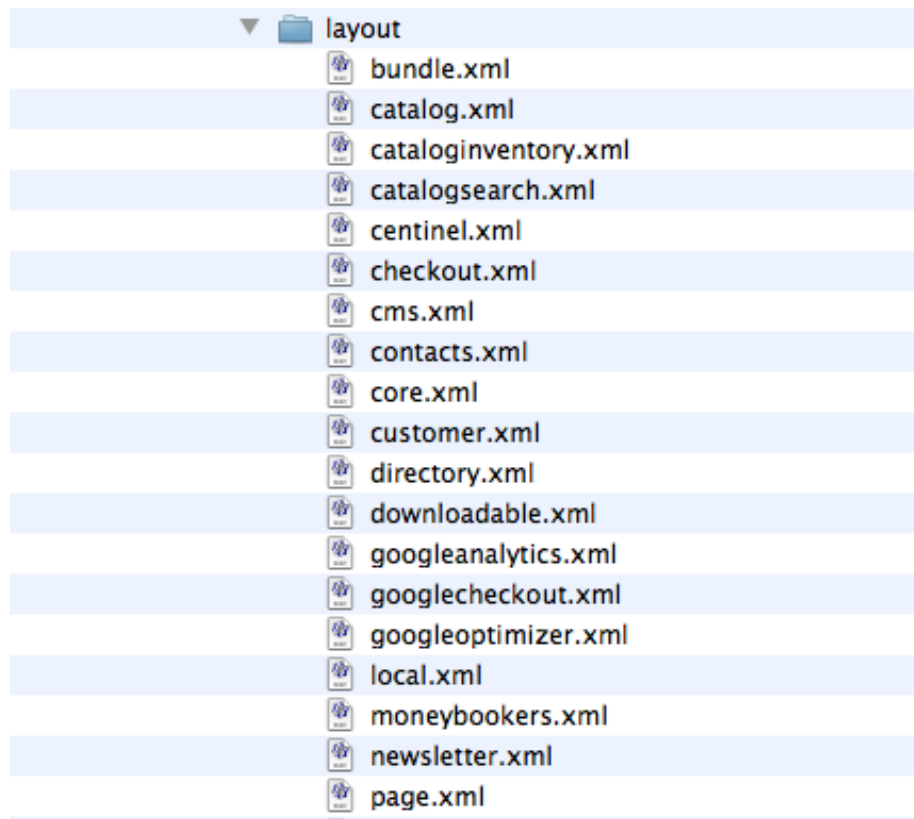


Figure 4.2

wish, the controller action developer then tells Magento it's time to render the layout object

4.3 Rendering a Layout

During the creation of Page Layout, certain Layout Update XML fragments marked certain blocks as "output blocks". When we say certain blocks, this is almost always a single block, and equally almost always this is the block named `root`. This root block renders a page template. This template, in turn, includes calls to render child blocks. Some of these child blocks render via a template file, others are `core/text_list` blocks which automatically render all their children. Others render via pure PHP in the `_toHtml` method. This blocks rendering sub-blocks, rendering sub-sub-blocks can continue many layers deep.

The end result of this rendering is a single string variable containing all the

HTML from the cascading render. The string is then passed into a Magento response object, which is responsible for outputting the HTML page.

That, in a nutshell, is the Layout system.

Visit <http://www.pulsestorm.net/nofrills-layout-chapter-four> to join the discussion online.

Chapter 5

Advanced Layout Features

If you've absorbed the previous chapters of information, you have everything you need to be a skillful practitioner of Magento layouts. Next we're going to cover some advanced features of the Layout system. We'll be moving pretty fast so if you're getting frustrated stop, take a deep breath, and remember that this isn't hard, it's just different. Once we're through, you'll be a true layout master.

Note: If you came here from Chapter 3, be sure to turn off the `nofrills_layoutbook` theme.

5.1 Action Parameters

We've already covered calling Action Methods, but let's quickly review. During the loading of the layout and instantiation of the block object, the XML below

```
<block type="some/foo" name="some_block">
    <action method="someMethod">
        <param1>a value</param1>
        <param2>27</param2>
    </action>
</block>
```

would run code something like the following

```
$block = new Mage_Some_Block_Foo();
$block->someMethod('a value','27');
```

There are, however, a few extra features you can tap into while calling methods via blocks. Let's take a look.

5.2 Translation System

Magento ships, out of the box, with a `gettext` like translation system. This system allows you to define a number of symbols (usually the English language version of a phrase), and then conditionally swap in a translated phrase. By default the action method parameters aren't run through this system, but it's possible (on an action by action basis) to tell Magento to translate certain parameters.

To indicate a parameter should be translated, you'd do something like

```
<action method="someMethod" translate="param1" module="core">
    <param1>a value</param1>
    <param2>27</param2>
</action>
```

We've added two parameters to the `<action/>` node here. The first is

```
translate="param1"
```

This tells Magento we want to run the `<param1/>` parameter through the translation engine. In this case, that's the string "a value". This is the reason each parameter is an extra node in the tree, it allows us to identify strings that need translation. If you want to translate more than one parameter, the attribute will accept multiple names

```
translate="param1 param2"
```

Next, we have

```
module="core"
```

This tells Magento which module's data helper should be used to translate our strings. Each module has (or should have) a helper class named `Data`.

```
Mage_Core_Helper_Data
Mage_Catalog_Helper_Data
```

This helper can be instantiated via a call to the static helper method on the `Mage` object

```
Mage::helper('core');           //shortcut for the one below
Mage::helper('core/data');
```

It's this helper object that has the translation function

```
$h = Mage::helper('core');
$hello = $h->__('Hello');

//in the above scenario "$hello" might contain
//the string "Hola" if the spanish locale was loaded
```

The reason you need to specify a module for the translation helper is, each module can contain **its own** translations. This allows different modules to translate their own symbols slightly differently based on context.

5.3 Conditional Method Calling

Another attribute you may see in the `<action/>` node is `ifconfig`.

```
<block type="page/html_head" name="head" as="head">
    <action method="addJs" ifconfig="dev/js/deprecation">
        <script>prototype/deprecation.js</script>
    </action>
</block>
```

This attribute can be used to tell Magento to **conditionally** call the specified method. The above XML is equivalent to PHP code something like

```
$block = new Mage_Page_Block_Html_Head();
if(Mage::getStoreConfigFlag('dev/js/deprecation'))
{
    $block->addJs('prototype/deprecation.js');
}
```

That is, when you use the `ifconfig` attribute, you're telling Magento

Only make the following method call if the following System Configuration Variable returns true

System Configuration variables can be set in the Admin Console under

System -> Configuration

See Appendix I for more information of using the System Config system.

The `ifconfig` attribute is a powerful feature you can use to allow end users to selectively turn certain layout features on or off. You can also use it to display different layout states based on the existing System Configuration values.

5.4 Dynamic Parameters

Magento also has the ability to pass **dynamic** parameters via Layout Update XML. Normally, parameter values need to be fixed values

```
<action method="someMethod" translate="param1" module="core">
    <param1>a value</param1>
    <param2>27</param2>
</action>
```

Above we're passing in the fixed values

```
a value
27
```

However, consider the following alternate syntax.

```
<action method="addLink" translate="label title" module="customer">
    <label>My Account</label>
    <url helper="customer/getAccountUrl"/>
    <title>My Account</title>
    <prepare/>
    <urlParams/>
    <position>10</position>
</action>
```

Here we're passing in three fixed values

```
<label>My Account</label>
...
<title>My Account</title>
...
...
<position>10</position>
```

We're also passing in two null values

```
...
...
...
<prepare/>
<urlParams/>
...
```

But there's one final parameter we're using with a syntax we haven't seen before

```
...
<url helper="customer/getAccountUrl"/>
...
...
...
...
```

This `url` parameter tag is fetching data dynamically using Magento's helper classes. When Magento encounters an action parameter with a helper attribute, it

1. Splits the helper by the last "/"
2. The first part of the split is used to instantiate the helper
3. The second part of the split is used as a method name
4. A helper is instantiated and the method from step #3 is called.
5. The value returned by the method is used in the action method call

So, that means the above XML translates into PHP code something like;

```
$block;      //the block object
$h           = Mage::helper('customer'); //instantiate the customer data helper
$url         = $h->getAccountUrl();
$block->addLink('My Account',$url,'My Account',null,null,10);
```

Magento examines the `helper` attribute and splits off `getAccountUrl` to use as a method, leaving `customer` to be used to instantiate the helper class. The helper is instantiated and `getAccountUrl` is called. The value returned from this method is then used as the parameter to pass to `addLink`.

The above example uses the shorthand "data" helper format, but fear not. You can use **any** helper class alias to return a value. Consider the following example

```
<action method="addLink" translate="label title" module="catalog"
ifconfig="catalog/seo/site_map">
    <label>Site Map</label>
    <url helper="catalog/map/getCategoryId" />
    <title>Site Map</title>
</action>
```

Here we're instantiating a `catalog/map` helper and calling its `getCategoryId` method. The value which `getCategoryId` returns will be used in the call to the `addLink` method.

This powerful feature is the missing link for layout programming. The ability to call into blocks with dynamic data parameters unlocks a world of potential for developers and designers alike.

5.5 Ordering of Blocks

Next up we have block ordering. We'll be working in the following controller action

```
#File: app/code/local/Nofrills/Booklayout/controllers/OrderController.php
class Nofrills_Booklayout_OrderController
extends Mage_Core_Controller_Front_Action
{
    public function indexAction()
    {
        $this->loadLayout();
        $this->renderLayout();
    }
}
```

Which corresponds to the URL

```
http://magento.example.com/nofrills_booklayout/order
```

Consider the following `<nofrills_booklayout_order_index>` update handle for our controller action. You should know how to add this to your system by now, but if you don't putting it in your `local.xml` will do. Review the previous chapters if you're unsure where `local.xml` is.

```
<nofrills_booklayout_order_index>
    <reference name="content">
        <block type="core/text" name="one">
            <action method="setText">
```

```

        <text><![CDATA[<p>One</p>]]></text>
    </action>
</block>
<block type="core/text" name="two">
    <action method="setText">
        <text><![CDATA[<p>Two</p>]]></text>
    </action>
</block>
<block type="core/text" name="three">
    <action method="setText">
        <text><![CDATA[<p>Three</p>]]></text>
    </action>
</block>
<block type="core/text" name="four">
    <action method="setText">
        <text><![CDATA[<p>Four</p>]]></text>
    </action>
</block>
<block type="core/text" name="five">
    <action method="setText">
        <text><![CDATA[<p>Five</p>]]></text>
    </action>
</block>
<block type="core/text" name="six">
    <action method="setText">
        <text><![CDATA[<p>Six</p>]]></text>
    </action>
</block>
<block type="core/text" name="seven">
    <action method="setText">
        <text><![CDATA[<p>Seven</p>]]></text>
    </action>
</block>
<block type="core/text" name="line">
    <action method="setText">
        <text><![CDATA[<hr/>]]></text>
    </action>
</block>
</reference>
</nofrills_booklayout_order_index>

```

Loading up our page with this bit of Layout Update XML in place will give us a simple ordered list of paragraphs, followed by a line, (see *Figure 5.1*)

(We’re using `<![CDATA[<hr/>]]>` nodes for our `setText` parameter. This allows us to insert HTML.)

Once you’ve got the above working, change your Layout Update XML such that an extra attribute named `before` is added to the block named `line`

```

<block type="core/text" name="line" before="two">
    <action method="setText"><text><![CDATA[<hr/>]]></text></action>
</block>

```

Refresh your page. The `<hr/>` element should now be rendered in between the “One” and “Two” paragraph, (see *Figure 5.2*)

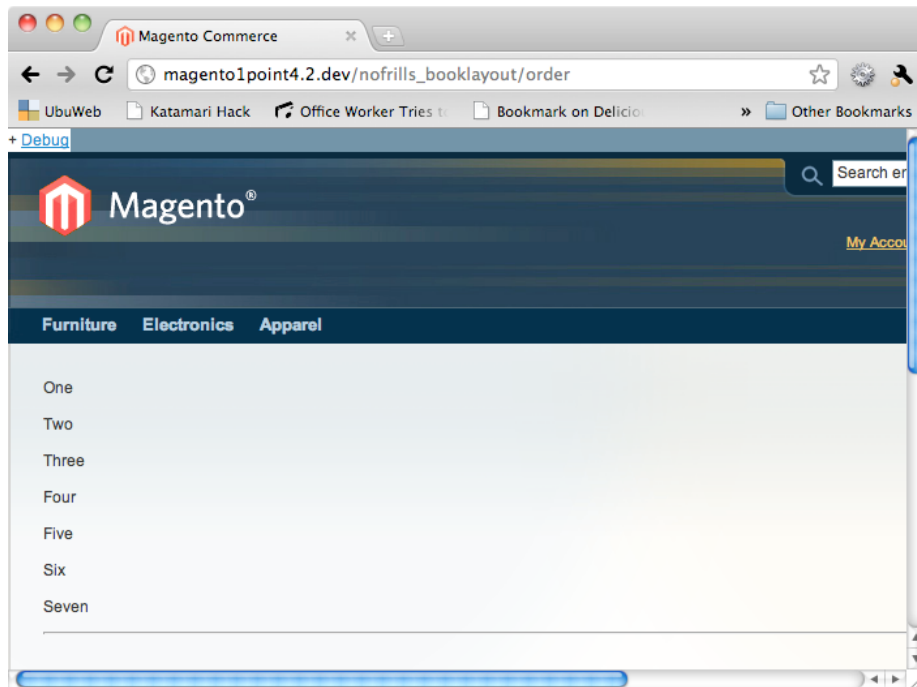


Figure 5.1

In plain english, the block `line` was inserted `before` the block `two`. There's also a corresponding `after` parameter.

```
<block type="core/text" name="line" after="six">
  <action method="setText"><text><![CDATA[<hr/>]]></text></action>
</block>
```

Reload your page with the above in place, and your line block should render between six and seven. If you want a block to be inserted last, just use

```
<block type="core/text" name="line" after="-">
```

If, however, you want your block to be inserted first, use

```
<block type="core/text" name="line" before="-">
```

The `before` and `after` attributes are most useful when you're inserting blocks into an existing set. For example, with the above in place, we might have another Layout Update XML node somewhere that looked like

```
<reference name="content">
  <block type="core/text" name="fakeline" after="four">
    <action method="setText">
      <text><![CDATA[<div style="border-color:black;
        border-style:solid;border-top:1px;width:300px;"></div>]]>
    </action>
  </block>
</reference>
```

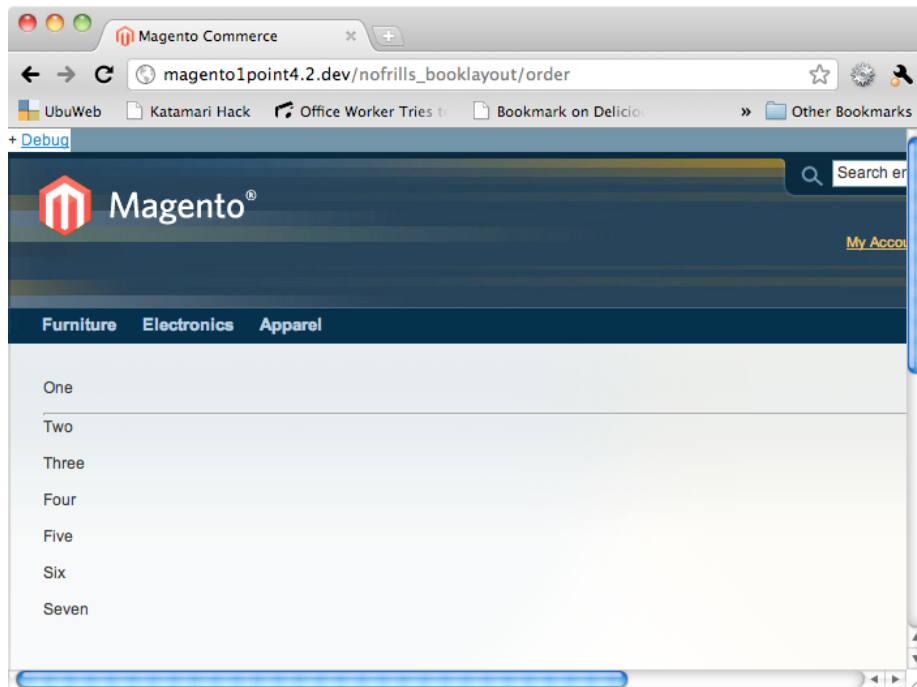


Figure 5.2

```

        </text>
      </action>
    </block>
  </reference>

```

Assuming blocks one - seven had already been inserted, this bit of Layout Update XML would ensure your new block was inserted **after** the block named **four**. This feature makes working with a package or theme's default layout blocks far easier.

5.6 Reordering Existing Blocks

One thing that trips people up when dealing with block ordering is, you can only control where an individual block is inserted **at the time of insertion**. Once you've inserted a block into the layout, it's "impossible" to change where it's rendered via the Layout XML files alone.

If you wanted to re-order a block that was already inserted, sometimes you can get away with removing it via the `unsetChild` method, and then reinserting it at the desired location

```
<reference name="content">
    <action method="unsetChild"><name>one</name></action>
    <block type="core/text" name="one" after="-">
        <action method="setText"><t>one</t></action>
    </block>
</reference>
```

While this will sometimes work, if the block you're removing had children, or has data parameters set by other parts of the layout, you'll need to reset them after reinserting the block. This makes the unset/re-insert method perilous at best, and should only be considered when all other options have been exhausted.

5.7 Template Blocks Need Not Apply

The `before` and `after` attributes work due to the way `core/text_list` blocks automatically render their children

```
class Mage_Core_Block_Text_List extends Mage_Core_Block_Text
{
    protected function _toHtml()
    {
        $this->setText('');
        foreach ($this->getSortedChildren() as $name) {
            $block = $this->getLayout()->getBlock($name);
            if (!$block) {
                Mage::throwException(Mage::helper('core')
                    ->__('Invalid block: %s', $name));
            }
            $this->addText($block->toHtml());
        }
        return parent::_toHtml();
    }
}
```

The important line here is

```
foreach ($this->getSortedChildren() as $name) {
```

This code is `foreach`ing over a list of **sorted** children. If you climb the chain back up to the `Mage_Core_Block_Abstract` class, you can see that Magento keeps track of both children blocks, as well as a sorted array of children

```
/**
 * Contains references to child block objects
 *
 * @var array
 */
protected $_children = array();

/**
 * Sorted children list
 *
 * @var array
```



```
*/  
protected $_sortedChildren = array();
```

So, while a `core/template` has this list of sorted children, the `before` and `after` attributes have no influence on a template block, as the order there is determined by where `$this->getChildHtml(...)` is called in the `phtml` template.

While it's beyond the scope of this book, an enterprising extension developer could probably create a class rewrite that would add a method to `core/text_list` blocks allowing for an explicit reorder of the `$_sortedChildren` array. I wouldn't be surprised to see the feature crop up in a future version of Magento.

5.8 Block Name vs. Block Alias

There's one last block attribute we need to talk about, and that's the `as` attribute.

```
<block type="sales/order_info" as="info" name="sales.order.info"/>
```

A block's name attribute defines its unique name in the layout object. If present, the `as` attribute will define the block's alias in the layout. If an alias is defined, you still interact with a block programmatically via its name. The only time you use an alias is when rendering the block in a template. For example, the above block's parent renders it with the following

```
<?php $this->getChildHtml('info'); ?>
```

This allows someone programming Layout XML Updates to insert a different block to be rendered **without** changing the template. If you take a look at the `insert` method

```
#File: app/code/core/Mage/Core/Block/Abstract.php  
public function insert($block, $siblingName='', $after=false, $alias='')  
{  
    //...  
    if ($block->getIsAnonymous()) {  
        $this->setChild('', $block);  
        $name = $block->getNameInLayout();  
    } elseif ('' != $alias) {  
        $this->setChild($alias, $block);  
        $name = $block->getNameInLayout();  
    } else {  
        $name = $block->getNameInLayout();  
        $this->setChild($name, $block);  
    }  
    //...  
}
```

you can see if an alias is used that's the value that will be used to set the child block (and therefore the "template name"). Otherwise, Magento defaults to using the the block's name in the layout.

Block aliases are a feature you may never personally use, but recent versions of Magento have made **heavy** use of them to overcome some earlier design decisions. You'll want to make sure you're aware of the difference between an alias and name, even if you never use an alias in your own updates.

5.9 Skipping a Child

We've already covered the `getChildHtml` method in a previous chapter. However, it has a cousin method named `getChildChildHtml`. This method is also defined on the `Mage_Core_Block_Abstract` class

```
public function getChildChildHtml($name, $childName = '', $useCache = true,
$sorted = false)
{
    if (empty($name)) {
        return '';
    }
    $child = $this->getChild($name);
    if (!$child) {
        return '';
    }
    return $child->getChildHtml($childName, $useCache, $sorted);
}
```

You use this method from a `phtml` template, and it might look something like

```
<?php echo $this->getChildChildHtml('my_child', 'foo'); ?>
```

The `getChildHtml` method will render out the specified child. The `getChildChildHtml` method obtains a reference to the first child block (`my-child` above), and then calls `getChildHtml` on it.

This method is most useful when you're editing a `phtml` template and don't want to restructure your blocks. I personally haven't found much use for it, but you will see it used in the wild and in the core, so it's worth knowing about. The most typical use is to render a `core/template` block as though it was a `core/text_list` block.

Visit <http://www.pulsestorm.net/nofrills-layout-chapter-five> to join the discussion online.

Chapter 6

CMS Pages

This chapter covers the CMS Page feature. For day-to-day Magento work most of the knowledge here is unnecessary. However, if you ever need to debug a Magento CMS page render, or are curious how CMS pages interact with the layout, this is the chapter for you. We'll also be laying the groundwork for our final chapter on Widgets.

Back in 1996, if you wanted to put a piece of content online, you just uploaded an HTML file. If you were really savvy, you'd upload an HTML **include** file that contained your content, and the HTML page itself would use server side includes.

It's weird, that in 2011, if you asked a developer how to add some content to a site or a web application, their process would be almost exactly the same. Instead of adding an HTML file, they'd add a controller and a template, and then put the HTML content in the template.


However, for **non-developers**, managing content on a website has gone completely GUI. Systems like Drupal, Concrete5, and Joomla rule the roost. Users expect to manage their sites via a user interface, and *not* via code or adding files. Magento's often overlooked CMS features allows users the control they want. Don't worry though, there's plenty in the CMS for a developer to sink their teeth into, particularly a developer who knows the layout system inside out.

6.1 Creating a Page

The CMS starts with a CMS Page entity. If you browse to

CMS -> Pages

in the Admin Console, you'll see a list of CMS pages that have already been added to the system, (see *Figure 5.1*)

 **Manage Pages**

Page of 1 pages | View per page | Total 9 records found

Title	URL Key	Layout	Store View	Status
<input type="text"/>	<input type="text"/>	<input type="text"/>	All Store Vie	<input type="text"/>
About Us	about-magento-demo-store	1 column	All Store Views	Enabled
Contact Form	contact-form-example	1 column	All Store Views	Enabled
Customer Service	customer-service	3sum columns	All Store Views	Enabled
Enable Cookies	enable-cookies	1 column	All Store Views	Enabled

Figure 6.1

If you click on "Add New Page" you'll be presented with a standard Magento editing UI, allowing you to enter information and create your page, (see *Figure 5.2*)

Page Information

Page Title *

URL Key *

▲ Relative to Website Base URL

Store View *

All Store Views
 Main Website
 Main Store
 English
 French
 German

Status *

Figure 6.2

Let's create a simple page by entering the following values. Don't worry about the specifics right now, we'll get to them down below

```
[Page Information : Page Title]    Our Hello World Page
[Page Information : URL Key]      hello-world
[Page Information : Store View]   All Store Views
```

Prepared for Compaa Peruana de E-commerce S.A.; Copyright ©2011 Pulse83 Storm LLC

[Page Information : Status]	Enabled
[Content: Content Heading]	Welcome World!
[Content: Editor]	The quick brown fox jumps over the lazy dog.

Once you're done, click on save and Next, load up the following URL in your browser

`http://magento.example.com/hello-world`

You should see your new CMS page, (see *Figure 5.3*)

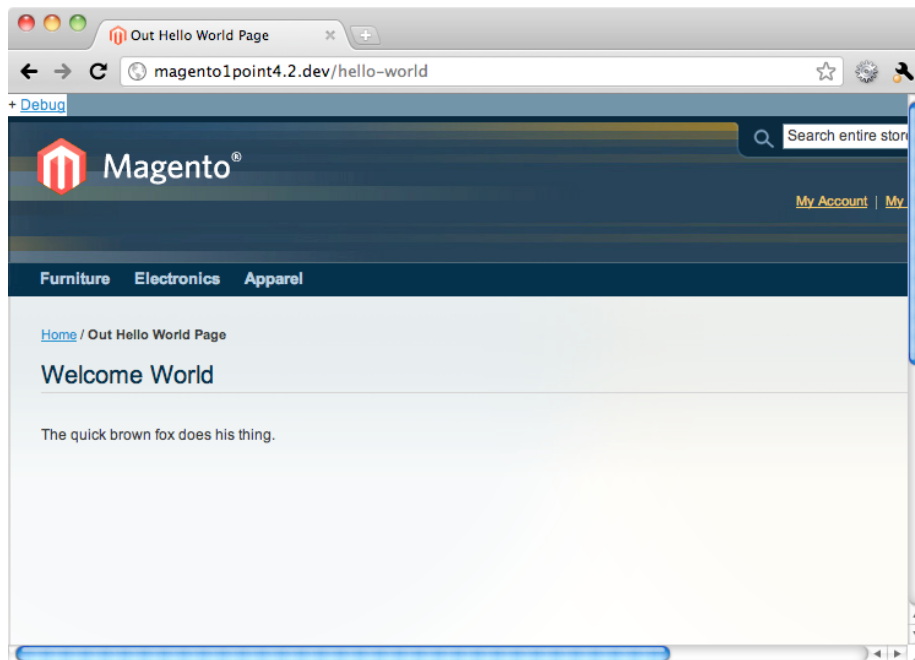


Figure 6.3

When you saved your page in the admin, Magento stored all that data as a `cms/page` model.

```
$page = Mage::getModel('cms/page');
```

When Magento identifies a URL as a CMS page, it loads this model up, reads its information, and then displays it to the page.

Let's take a look at all of the CMS Page fields and briefly describe what they do.

6.1.1 Page Information : Page Title

This is the title of your CMS page. It will display in the Admin Console grid, your page's `<title/>` tag, and the default breadcrumb display.

6.1.2 Page Information : URL Key

This is the non-server portion of your page URL. This can be any string that's valid in a URL. In our example above, we used

```
hello-world
```

If we had used

```
hello-world.html
```

our page URL would have been

```
http://magento.example.com/hello-world.html
```

6.1.3 Page Information : Store View

This setting determines which stores your page may appear in. This allows you to hide content from stores where it may not be appropriate, or provide different version of a page for different stores.

Stores here is referring to Magento's internal abstract "Store" concept.

6.1.4 Page Information : Status

The status field allows you to disable or enable a page. A disabled page will return a response with a 404 HTTP Status code. This is great for embargoed content, or for saving seasonal content to use over again.

6.1.5 Content: Content Heading

The content heading determines your page's top level `<h1/>` tag. (See rendering section below for more information).

6.1.6 Content: Editor

This is the rich text editor where you enter your page's content. In addition to the various formatting buttons common to most rich text editors, clicking the Show/Hide Editor button will toggle the raw source view. When viewing a page in raw source view, you can view and edit the actual HTML that will render your page.

Also in raw source view, you'll see a few additional buttons. Insert Widget, Insert Image, and Insert Variable. Clicking one these buttons eventually results in text something like

```
{{config path="trans_email/ident_general/email"}}
```

being added to your raw source. This is a directive tag, and we'll cover it in greater detail in just a bit.

6.1.7 Meta : Keywords

The text in this field controls the `<meta name="keywords"/>` tag on your page. The contents of this field will be added directly to the `content` attribute of the tag.

```
<meta name="keywords" content="This is a test of the keywords." />
```

6.1.8 Meta : Description

Much like **Keywords**, the **Description** field controls the contents of your page's `<meta name="description"/>` tag.

```
<meta name="description" content="Describing the meta." />
```

6.1.9 Design : Layout

This select box allows you to set which page structure template your CMS page will use. This select is populated by a call to

```
//Mage_Page_Model_Source_Layout
Mage::getSingleton('page/source_layout')->toOptionArray()
```

which, in a default installation, ultimately reads from the global config nodes at the following location

```
<config>
    <global>
        <cms>
            <page>
                ...
```

```
        </page>
    </cms>
</global>
</config>
```

You can take a look at the structure in
`app/code/core/Mage/Page/etc/config.xml`

for an idea on what Magento expects to find in there

```
<page>
    <layouts>
        <empty module="page" translate="label">
            <label>Empty</label>
            <template>page/empty.phtml</template>
            <layout_handle>page_empty</layout_handle>
        </empty>
        <one_column module="page" translate="label">
            <label>1 column</label>
            <template>page/1column.phtml</template>
            <layout_handle>page_one_column</layout_handle>
            <is_default>1</is_default>
        </one_column>
        <two_columns_left module="page" translate="label">
            <label>2 columns with left bar</label>
            <template>page/2columns-left.phtml</template>
            <layout_handle>page_two_columns_left</layout_handle>
        </two_columns_left>
        <two_columns_right module="page" translate="label">
            <label>2 columns with right bar</label>
            <template>page/2columns-right.phtml</template>
            <layout_handle>page_two_columns_right</layout_handle>
        </two_columns_right>
        <three_columns module="page" translate="label">
            <label>3sum columns</label>
            <template>page/3columns.phtml</template>
            <layout_handle>page_three_columns</layout_handle>
        </three_columns>
    </layouts>
</page>
```

If you can hold on, you'll eventually understand to the meaning of all the tags above. For now, if you take a peek at the HTML source for that select

```
<select id="page_root_template" name="root_template"
class="required-entry select">
    <option value="empty">Empty</option>
    <option value="one_column" selected="selected">1 column</option>
    <option value="two_columns_left">2 columns with left bar</option>
    <option value="two_columns_right">2 columns with right bar</option>
    <option value="three_columns">3sum columns</option>
</select>
```

you can see that Magento's taking the **tag names** from the above nodes for option values. This is what Magento will save with its page model, and will then use later to retrieve the label, template, and layout_handle values.

6.1.10 Design : Layout Update XML

The Magento CMS system still uses the layout/block mechanism for page rendering. This field will allow you to add additional Layout Update XML fragments to your Page Layout for a CMS Page request. For example, you could add an additional text content block if you liked with

```
<reference name="content">
    <block type="core/text" name="redundant">
        <action method="setText"><text>Hello Again</text></action>
    </block>
</reference>
```

6.1.11 Design : Custom Design

Fields in this section allow users to override the above values, and our default theme, for specific date ranges.

6.2 CMS Page Rendering

Overview out of the way, let's get to those seemingly confusing abstractions! You're probably wondering how Magento knows a particular request should be rendered with a CMS Page. If you're a certain kind of developer, you're wondering how Magento routes a URL to the CMS rendering routines, (which is just a different way of saying the same thing)

When a URL request comes into Magento, the first thing Magento asks itself is

Based on my current configuration, should this URL be handled by an admin controller?

If the answer is yes, Magento dispatches to the appropriate admin action controller. If not, the next thing Magento asks itself is

Based on my current configuration, should this URL be handled by a frontend controller action?

If the answer is yes, Magento dispatches to the appropriate action controller. If the answer is no, Magento asks itself one last question

Looking at that URL, is there a CMS page that matches its key/identifier?

If so, manually set **the request's** module, controller, and action. Also, add a parameter with the page ID. The Page ID is the database ID of the `cms/page` object. The code that does this looks something like

```
$request->setModuleName('cms')
->setControllerName('page')
->setActionName('view')
->setParam('page_id', $pageId);
```

By doing this, Magento will automatically dispatch to the following controller action.

```
Mage_Cms_PageController::viewAction
```

If you're interested in checking out the code that looks for a CMS page, checkout the `match` method in

```
#File: app/code/core/Mage/Cms/Controller/Router.php
public function match(Zend_Controller_Request_Http $request)
{
    ...
}
```

6.3 Index Page

The one exception to the routing scenario described above is the special root page of a site, alternately called the "index" page or the "home" page.

```
http://magento.example.com/
```

Magento URLs that lack ANY path portion (that is, they contain only a server name) will be dispatch to the following controller action.

```
Mage_Cms_IndexController::indexAction
```

This check happens **after** the check for a standard controller is instantiated, but **before** the CMS Page check is done.

You can override which controller the index page dispatches to via the System Config variable

```
System -> Configuration -> Web -> Default Pages -> Default Web Url
```

In a base install, this value is `cms`. What that means is, when you go to the root level page, Magento will treat it as though you've gone to

```
http://magento.example.com/cms
```

If you wanted to have a particular category page display as the home page, you could set this value to something like `catalog/category/view/id/8`.

6.4 What You Need to Know

That was some heavy abstract lifting back there. If you're not interested in those kind of details, all you really need to know can be summed up by the following

If Magento decides a URL needs a CMS page, it dispatches to
`Mage_Cms_PageController::viewAction`

Let's take a look at that controller

```
#File: app/code/core/Mage/Cms/controllers/PageController.php
class Mage_Cms_PageController extends Mage_Core_Controller_Front_Action
{
    /**
     * View CMS page action
     */
    public function viewAction()
    {
        $pageId = $this->getRequest()
            ->getParam('page_id', $this->getRequest()->getParam('id', false));
        if (!Mage::helper('cms/page')->renderPage($this, $pageId)) {
            $this->_forward('noRoute');
        }
    }
}
```

That's only four lines in the method body, but if you're not familiar with Magento coding conventions, it's four dense looking lines. We're going to tease apart what's going on in this method. If you're already familiar with Magento conventions you may want to skip ahead (although reviewing information never hurt anyone)

The call to `$this->getRequest()` returns the Magento request object. Rather than have you interact directly with `$_GET`, `$_POST` and `$_COOKIE`s, Magento provides a request object that allows you access to the same information. This object is a `Mage_Core_Controller_Request_Http`, which extends from a Zend class (`Zend_Controller_Request_Http`)

Next, we're chaining in a call to `getParam` in order to retrieve the value of `page_id`. This is the id of our `cms/page` model. The second parameter to `getParam` is a default value to return if `page_id` isn't found. In this case, we're calling `getParam` **again**, this time looking for value of the id parameter. If there's no id parameter, `$pageId` is set to false.

So, we now have our page id. Next,

```
if (!Mage::helper('cms/page')->renderPage($this, $pageId)) {
    $this->_forward('noRoute');
}
```

we instantiate a `cms/page` helper class, and call its `render` method. We pass in a reference to the controller, and the page id we just fetched from the request.

If this method returns false, we forward on to `noRoute`, which for our purposes we'll call the 404 Page.

6.5 Where's the Layout?

Earlier we mentioned the CMS system used the same layout rendering engine as the rest of Magento. However, you're probably wondering where the calls to `$this->loadLayout()` and `$this->renderLayout()` are. You may also be wondering why we're doing something weird like passing a reference to the Controller (`$this`) to our `cms/page` helper.

The answers to both questions lies within the `renderPage` method, so lets take a look

```
#File: app/code/core/Mage/Cms/Helper/Page.php
class Mage_Cms_Helper_Page extends Mage_Core_Helper_Abstract
{
    public function renderPage(Mage_Core_Controller_Front_Action $action,
        $pageId = null)
    {
        return $this->_renderPage($action, $pageId);
    }

    ...

    protected function _renderPage(
        Mage_Core_Controller_Varien_Action $action, $pageId = null,
        $renderLayout = true)
    {
        ...

        $action->loadLayoutUpdates();
        $layoutUpdate = ($page->getCustomLayoutUpdateXml() && $inRange)
            ? $page->getCustomLayoutUpdateXml() : $page->getLayoutUpdateXml();
        $action->getLayout()->getUpdate()->addUpdate($layoutUpdate);
        $action->generateLayoutXml()->generateLayoutBlocks();

        ...

        if ($renderLayout) {
            $action->renderLayout();
        }
    }
}
```

We've truncated much of the actual code (...) to focus on the specific lines above. You'll see that the `renderPage` method wraps a call to the internal, protected `_renderPage` method. Notice that the controller we've passed in is (locally) known

as `$action`. Without going into too much detail, the code above replaces your calls to `$this->loadLayout()`.

In fact, if you looked at the implementation of the `loadLayout` method in the base action controller, you'd see code similar to what's above. The only difference here is, after loading the layout update handles from the package layout files, we then add any additional layout handles from our CMS Page. (You'll recall that Admin Console allowed us to add layout update handles for specific CMS pages)

We won't go into every little detail of the page rendering process, but we will highlight a few other chunks of code that should shed some light on what we were doing in the Admin Console GUI.

6.6 Adding the CMS Blocks

Take a look at the following line

```
$action->getLayout()->getUpdate()  
    ->addHandle('default')  
    ->addHandle('cms_page');
```

Here, the handle `cms_page` is being issued. This means when we're pulling Layout Update XML from the package layout, the following will be included.

```
<!-- File: app/design/frontend/base/default/layout/cms.xml -->  
<layout>  
    <!-- ... -->  
    <cms_page translate="label">  
        <label>CMS Pages (All)</label>  
        <reference name="content">  
            <block type="core/template" name="page_content_heading"  
                template="cms/content_heading.phtml"/>  
            <block type="page/html_wrapper" name="cms.wrapper" translate="label">  
                <label>CMS Content Wrapper</label>  
                <action method="setElementClass"><value>std</value></action>  
                <block type="cms/page" name="cms_page"/>  
            </block>  
        </reference>  
    </cms_page>  
    <!-- ... --->  
</layout>
```

This is the key Layout Update XML for CMS pages. It adds the blocks for the content heading, and the page content itself, to the page layout. Later on in the render method we set the page content header by grabbing the saved content header values from our page model

```
$contentHeadingBlock = $action->getLayout()->getBlock('page_content_heading');  
if ($contentHeadingBlock) {  
    $contentHeadingBlock->setContentHeading($page->getContentHeading());  
}
```

This value is then referenced in the content heading block's template `cms/content_heading.phtml`.

6.7 Setting the Page Template

You may remember configuring a page template in the GUI. This template is stored in the `root_template` property, and Magento uses it here

```
if ($page->getRootTemplate()) {  
    $action->getLayout()->helper('page/layout')  
        ->applyTemplate($page->getRootTemplate());  
}
```

The `Layout` object's `applyTemplate` method takes the saved value (say, `two_columns_left`), jumps back into the config to find the name of the template it should set, and then sets it.

```
public function applyTemplate($pageLayout = null)  
{  
    if ($pageLayout === null) {  
        $pageLayout = $this->getCurrentPageLayout();  
    } else {  
        $pageLayout = $this->_getConfig()->getPageLayout($pageLayout);  
    }  
  
    if (!$pageLayout) {  
        return $this;  
    }  
  
    if ($this->getLayout()->getBlock('root') &&  
        !$this->getLayout()->getBlock('root')->getIsHandle()) {  
        // If not applied handle  
        $this->getLayout()  
            ->getBlock('root')  
            ->setTemplate($pageLayout->getTemplate());  
    }  
  
    return $this;  
}
```

You'll remember that the `two_columns_left` config node looked something like this

```
<two_columns_left module="page" translate="label">  
    <label>2 columns with left bar</label>  
    <template>page/2columns-left.phtml</template>  
    <layout_handle>page_two_columns_left</layout_handle>  
</two_columns_left>
```

You can see we're using the `<template/>` node above, but we don't seem to be using the `<layout_handle/>` anywhere. Plus, there's the `getIsHandle` method call above. What's that all about?

Other parts of the system will add a layout handle named after the values in the `<layout_handle>` tag. If you look at one of these handles

```
<page_two_columns_left translate="label">
  <label>All Two-Column Layout Pages (Left Column)</label>
  <reference name="root">
    <action method="setTemplate">
      <template>page/2columns-left.phtml</template>
    </action>
    <!-- Mark root page block that template is applied -->
    <action method="setIsHandle"><applied>1</applied></action>
  </reference>
</page_two_columns_left>
```

you can see it's applying a template via the `setTemplate` method, and also setting a `IsHandle` flag on the object. This flag is used internally by the block to prevent multiple handles from setting the root template. This isn't done by the CMS Page render, but it's good to know about.

6.8 Rendering the Content Area

So that covers some of the ancillary items around rendering a CMS page, but what about the page content itself? We've added a `cms/page` block named `cms_page` using the `cms_page` handle, but we don't seem to do anything with it.

That's because the CMS block itself does the rendering. If you take a look at its `_toHtml` method

```
class Mage_Cms_Block_Page extends Mage_Core_Block_Abstract
{
    //...
    protected function _toHtml()
    {
        /* @var $helper Mage_Cms_Helper_Data */
        $helper = Mage::helper('cms');
        $processor = $helper->getPageTemplateProcessor();
        $html = $processor->filter($this->getPage()->getContent());
        $html = $this->getMessagesBlock()->getGroupedHtml() . $html;
        return $html;
    }
    //...
}
```

we can see we're calling `$this->getPage()->getContent()`, which looks like a likely candidate. But how is `getPage()` obtaining a reference to our CMS Page object?

```
public function getPage()
{
    if (!$this->hasData('page')) {
        if ($this->getPageId()) {
            $page = Mage::getModel('cms/page')
                ->setStoreId(Mage::app()->getStore()->getId())
                ->load($this->getPageId(), 'identifier');
        } else {
            $page = Mage::getSingleton('cms/page');
        }
    }
}
```

```
        $this->setData('page', $page);
    }
    return $this->getData('page');
}
```

It looks like this method will

1. Look for a data member named 'page'. If it finds it, return it
2. If not, look for a data member named page_id (getPageId). If we find it, use it to instantiate a page object
3. If there's no page_id data member, get a reference to the cms/page singleton.

It's #3 that's the key here. We didn't set any data properties named page or page_id. However, when we originally instantiated our page object

```
#File: app/code/core/Mage/Cms/Helper/Page.php
$page = Mage::getSingleton('cms/page');
if (!is_null($pageId) && $pageId!=$page->getId()) {
    $delimiterPosition = strrpos($pageId, '|');
    if ($delimiterPosition) {
        $pageId = substr($pageId, 0, $delimiterPosition);
    }

    $page->setStoreId(Mage::app()->getStore()->getId());
    if (!$page->load($pageId)) {
        return false;
    }
}
```

we created a singleton instance. This means that we'll only ever have one reference to this object during the PHP request lifecycle, which is why our call to

```
$page = Mage::getSingleton('cms/page');
```

returns the same page we were dealing with in the helper class

6.9 Page Content Filtering

Our mystery of the CMS Page object solved, let's examine the `_toHtml` method of our block again

```
protected function _toHtml()
{
    /* @var $helper Mage_Cms_Helper_Data */
    $helper = Mage::helper('cms');
    $processor = $helper->getPageTemplateProcessor();
    var_dump
    $html = $processor->filter($this->getPage()->getContent());
    $html = $this->getMessagesBlock()->getGroupedHtml() . $html;
    return $html;
}
```


Rather than just return the contents of `$this->getPage()->getContent()`, this code instantiates a filtering object and passes our content through it to get the final HTML.

This is the code that's responsible for swapping out the template **directive** tags we mentioned earlier.

```
{{config path="trans_email/ident_general/email"}}
{{media url="/workforfree.jpg"}}
```

The object returned by the call to

```
$helper->getPageTemplateProcessor()
```

contains all the code that will process these template directives. Like a lot of Magento, this is a configuration based class instantiation. If you look at the implementation of `getPageTemplateProcessor`

```
#File: app/code/core/Mage/Cms/Helper/Data.php
class Mage_Cms_Helper_Data extends Mage_Core_Helper_Abstract
{
    const XML_NODE_PAGE_TEMPLATE_FILTER = 'global/cms/page/template_filter';
    const XML_NODE_BLOCK_TEMPLATE_FILTER = 'global/cms/block/template_filter';
    public function getPageTemplateProcessor()
    {
        $model = (string)Mage::getConfig()
            ->getNode(self::XML_NODE_PAGE_TEMPLATE_FILTER);
        return Mage::getModel($model);
    }
    <!-- ... -->
```

You can see we look for our directive filtering class at the global configuration node `global/cms/page/template.filter`. As of Magento 1.4.2, this node contains the class alias `widget/template.filter`, which translates into the class `Mage_Widget_Model_Template_Filter`. However, this may have changed by the time you're reading this, as whenever Magento adds new template directives a new filtering class is created that extends the old one, add adds the new filtering methods.

6.10 Filtering Meta Programming

If you follow the inheritance chain of the Template Filter far enough back, you eventually reach `Varien_Filter_Template`

```
Mage_Widget_Model_Template_Filter extends
Mage_Cms_Model_Template_Filter extends
Mage_Core_Model_Email_Template_Filter extends
Mage_Core_Model_Email_Template_Filter extends
Varien_Filter_Template
```

This class defines an object which contains parsing code which will

1. Look for any `{{foo path="trans_email/ident_general/email"}}` style strings
2. Parse the token for the directive name (`foo` above)
3. Create a method name based on the directive name. In the above example, the directive name is `foo`, which means the method name would be `fooDirective`
4. Use `call_user_func` to call this method on itself, passing in the an array containing a tokenized version of the directive string.

It's beyond the scope of this book to cover the implementation details of each specific directive. We mention mainly to let you know that, if you're trying to debug a particular template directive, say

```
{{media url="/workforfree.jpg"}}
```

you can find its implementation method by taking the directive name (`media`), and adding the word `Directive`. A quick search through the code base should turn up the implementation

```
#File: app/code/core/Mage/Core/Model/Email/Template/Filter.php
public function mediaDirective($construction)
{
    $params = $this->_getIncludeParameters($construction[2]);
    return Mage::getBaseUrl('media') . $params['url'];
}
```

In this specific case we can see that the `{{media ...}}` directive simply grabs the base media URL using `Mage::getBaseUrl('media')`, and appends the `url` parameter to it.

Visit <http://www.pulsetorm.net/nofrills-layout-chapter-six> to join the discussion online.

Chapter 7

Widgets

Consider the following situation. You're a developer. You have a deep knowledge of the Magento system. The corporate VP in charge of giving you things to do runs into your work area and says

I want to add a YouTube video to the sidebar?!

You start explaining layouts, and blocks, and pages, and how they render, and which XML file he'll need to edit, or maybe you could add it as a page update o...

Your boss then gives you that steely, bossy look and says again

I want to add a YouTube video to the sidebar

Most people don't work on their own cars. Most people don't harvest or hunt their own food. And most people don't want to code their own websites. That's the problem widgets set out to solve. In this chapter we'll give you a full overfull of the Magento widget system. From using the widgets that ship with Magento, to creating your own widgets, to understanding how widgets are inserted into the flow of the Layout.

7.1 Widgets Overview

So, what are widgets?

1. Widgets are Magento Template Blocks
2. Widgets Contain Structured Data
3. Widgets Contain Rules for Building User Interfaces
4. Widgets are formally associated with a number of phtml template files

5. Widgets contain rules that say which blocks in the layout system are allowed to contain them

Let's start by building ourselves a minimum viable widget, and inserting it into a CMS page. We'll be building our widget in the `Nofrills_Booklayout` module. You, of course, are free to add widgets to **any** module you create.

To start with, we need to create a configuration file that will let Magento know about our widget. Being a newer subsystem of Magento, widgets have their own custom XML config file which will be merged with the Magento config as needed. Widget config file are named `widget.xml`, and should be placed in your module's `etc` folder

```
<!-- #File: app/code/local/Nofrills/Booklayout/etc/widget.xml -->
<widgets>
</widgets>
```

There are times where Magento will load the widget config from cache, and there's other times where the config will always be loaded from disk. Because of that, it's best to always clear the cache when making changes to this file.

We now have an empty widget config. Next, let's add a node to hold our widget definition

```
<!-- #File: app/code/local/Nofrills/Booklayout/etc/widget.xml -->
<widgets>
    <nofrills_layoutbook_youtube type="nofrills_booklayout/youtube">
        <name>YouTube Example Widget</name>
        <description type="desc">
            This widget displays a YouTube video.
        </description>
    </nofrills_layoutbook_youtube>
</widgets>
```

Each second level node in this file tells Magento about a single widget that's available to the system. You should take steps to ensure this node's name is unique to avoid possible collisions with other widgets that are loaded in the system from other modules. In this case, the name `nofrills_layoutbook_youtube` should suffice.

It's the `type="nofrills_booklayout/youtube"` attribute we're interested in. This defines a block class alias for our widget. We're telling Magento that the block class

`Nofrills_Booklayout_Block_Youtube`

should be used for rendering this widget. The `<name/>` and `<description/>` tags are used for text display in the Magento Admin Console.

Let's create that class. Add the following file

```
#File: app/code/local/Nofrills/Booklayout/Block/Youtube.php
<?php
class Nofrills_Booklayout_Block_Youtube extends Mage_Core_Block_Abstract
```

Prepared for Compaa Peruana de E-commerce S.A.; Copyright ©2011 Pulse99 Storm LLC

```
implements Mage_Widget_Block_Interface
{
    protected function _toHtml()
    {
        return '<object width="640" height="505">
            <param name="movie"
            value="http://www.youtube.com/v/dQw4w9WgXcQ?fs=1&hl=en_US">
            </param>
            <param name="allowFullScreen" value="true"></param>
            <param name="allowscriptaccess" value="always"></param>
            <embed src="http://www.youtube.com/v/dQw4w9WgXcQ?fs=1&hl=en_US"
            type="application/x-shockwave-flash"
            allowscriptaccess="always" allowfullscreen="true"
            width="640" height="505"></embed></object>';
    }
}
```

This class is *mostly* a standard block class. It extends from the `Mage_Core_Block_Abstract` class, and we've overridden the base `_toHtml` method to have this block return the embed code for a specific YouTube video. The one difference you'll notice is the class definition also has this

```
implements Mage_Widget_Block_Interface
```

This line is important. It tells PHP that our class is implementing the widget interface. If you don't understand the concept of PHP OOP interfaces, don't worry. Just include this line with your widget class. Without it, Magento won't be able to fully identify your block as a widget class.

That's it! We now have a super simple widget. Let's take it for a spin!

7.2 Adding a Widget to a CMS Page

We'll need to setup a new CMS Page for our widget. Complete the following steps

1. Go to **CMS** -> **Pages** in the Admin Console
2. Click on **Add New Page**
3. Enter **YouTube Video** in the Page Title field
4. Enter **example-youtube** in the URL Key field
5. Select **All Store Views**
6. Ensure that **Enabled** is selected for status
7. Click on the **Content** tab, and enter a Content Heading, as well as some text in the editor
8. Click on **Save and Continue Edit** button

9. Load your new page in a browser, at <http://magento.example.com/example-youtube>

Now that we've got our new page setup, let's add the widget. Choose the **Content Tab** in the CMS Page editing interface, and click on the Show/Hide Editor (see *Figure 7.1*)



Figure 7.1

The WYSIWYG editing will disappear and be replaced by an HTML source editor. More importantly, you'll have a new list of buttons, one of which is **Insert Widget**. Click on this button, and a modal window will come up (see *Figure 7.2*)

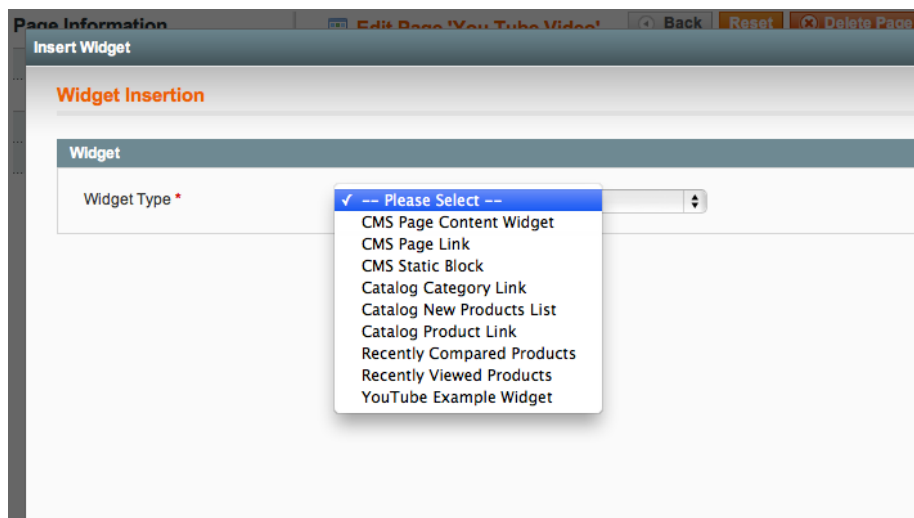


Figure 7.2

If you click on the **Widget Type** drop-down, you'll see a list of standard Magento widgets, with your **YouTube Example Widget** widget listed last.

Select your widget from the menu and click in **Insert Widget**. You should notice the following text has been added to your HTML source

```
{{widget type="nofrills_booklayout/youtube"}}
```

Save your CMS page, and then load the page

```
http://magento.example.com/example-youtube
```

in your web browser. You should see your embedded YouTube video.

7.3 CMS Template Directives

The `{{curly braces}}` text is a template directive. When Magento encounters these, a template engine will kick in. If your widget isn't displaying correctly and you want to debug this template engine, hop to the following file

```
#File: app/code/core/Mage/Widget/Model/Template/Filter.php
...
class Mage_Widget_Model_Template_Filter extends Mage_Cms_Model_Template_Filter
{
    ...
    public function widgetDirective($construction)
    {
        ...widget directives are rendered here...
    }
    ...
}
```

Every directive in a CMS page works this way. Just look for the method name that matches the directive name, followed by the word directive.

```
widgetDirective
templateDirective
foobazbarDirective
```

The `{{widget}}` directive has a useful feature. You can use it to set properties on your widget block object (see Appendix G: Magento Magic setters and getters). We can use this to make our widget a bit more useful.

Change your block code so it matches the following, and refresh the CMS page.

```
<?php
class Nofrills_Booklayout_Block_Youtube extends Mage_Core_Block_Abstract
implements Mage_Widget_Block_Interface
{
    protected function _toHtml()
    {
        $this->setVideoId('dQw4w9WgXcQ');
        return '
    }
}
```

```

<object width="640" height="505">
  <param name="movie" value="http://www.youtube.com/v/' .
    $this->getVideoId() .
    '?fs=1&hl=en_US"></param>
  <param name="allowFullScreen" value="true"></param>
  <param name="allowscriptaccess" value="always"></param>
  <embed src="http://www.youtube.com/v/' .
    $this->getVideoId() .
    '?fs=1&hl=en_US"
    type="application/x-shockwave-flash" allowscriptaccess="always" ' .
    'allowfullscreen="true" width="640" height="505"></embed>
</object>
';
}
}

```

Your CMS page will remain unchanged. We've altered the code above to set a `video_id` data property on the block object, and then used that property in rendering the YouTube embed code. (Remember, data properties are stored with `underscore_notation`, but the magic methods to fetch them are `CamelCased`)

Next, remove the following line from your block and reload the CMS page.

```
$this->setVideoId('dQw4w9WgXcQ');
```

Without setting this property, the video will fail to render. So far that's all pretty obvious. Next, edit the widget directive so it looks like the following

```
{{widget type="nofrills_booklayout/youtube" video_id="dQw4w9WgXcQ"}}
```

Save the CMS page, and reload the frontend page in your browser. Your video is back!

The `widgetDirective` method will parse the directive text for attributes, and if it finds any they'll be assigned as data attributes to the widget object. With this feature, your widgets go from static content renderers to dynamic content renderers.

7.4 Adding Data Property UI

Of course, the whole point of widgets is that they're meant as a method of codeless block adding. While it's good to **know** you can edit the widget directives directly, something more is needed if this feature is going to fulfill its promise.

In your widget config, add a `<parameters/>` node as defined below.

```

<!-- #File: app/code/local/Nofrills/Booklayout/etc/widget.xml -->
<widgets>
  <nofrills_layoutbook_youtube type="nofrills_booklayout/youtube">
    <name>YouTube Example Widget</name>
    <description type="desc">
      This widget displays a YouTube video.
    </description>
  </nofrills_layoutbook_youtube>
</widgets>

```



```

</description>

<!-- START new section -->
<parameters>
  <video_id>
    <required>1</required>
    <visible>1</visible>
    <value>Enter ID Here</value>
    <label>YouTube Video ID</label>
    <type>text</type>
  </video_id>
</parameters>
<!-- END new section -->

</nofrills_layoutbook_youtube>
</widgets>

```

Clear your cache, and then click on the **Insert Widget** button again. Select your widget from the drop-down, and you will now see a UI for entering a video ID, (see *Figure 7.3*)

Widget Insertion

Widget

Widget Type * YouTube Example Widget
This widget displays a YouTube video.

Widget Options

YouTube Video ID *

Figure 7.3

Enter an ID (we recommend `qYkbTyHXwbs` to keep with the theme) and click on **Insert Widget**. The following directive code should be inserted into the content area.

```
{{widget type="nofrills_booklayout/youtube" video_id="qYkbTyHXwbs"}}
```

Easy as that, you now have a widget for inserting any YouTube video into any page. Let's take a look at the XML we added to our widget config

```

<parameters>
  <video_id>
    <required>1</required>
    <visible>1</visible>
    <value>Enter ID Here</value>

```

```
<label>YouTube Video ID</label>
<type>text</type>
</video_id>
</parameters>
```

This node will *formally* add data parameters to our widget, and allow us to specify a field type for data entry. The `<video_id>` tag here **does have** semantic value, it's the name of the attribute that will be added to the directive tag

```
{{widget type="nofrills_booklayout/youtube" video_id="[VALUE]"}}
```

The `<required>` tag allows a level of data validation, setting this to "1" will force the Admin Console user to enter a value before inserting the widget.

The `<visible/>` node allows you to hide the input field for this data parameter, and have the inserted widget directive tag automatically include an attribute every time its used, with a value provided by the `<value/>` tag. When `<visible/>` is set to 1 the `<value/>` tag will be used as a default ID.

The value in `<label>` will be used to provide your rendered HTML form with a label, and `<type/>` controls what sort of form element is rendered. See Appendix G for a full list and explanation of form rendering configurations.

Important: Be careful changing data parameters of a deployed widget. Once a `{{widget...}}` directive tag has been added to a CMS page, it become "detached" from its definition. That is, if we changed the `<video_id/>` above to be `<youtube_id/>`, our CMS page would still have the

```
{{widget type="nofrills_booklayout/youtube" video_id="[VALUE]"}}
```

widget tag. While this isn't necessarily a problem, it may cause confusion while further developing the widget or debugging rendering issues.

7.5 Widget Templates

Looking back at our five defining widget properties

1. Widgets are Magento Template Blocks
2. Widgets Contain Structured Data
3. Widgets Contain Rules for Building User Interfaces
4. Widgets are formally associated with a number of phtml template files
5. Widgets contain rules that say which blocks in the layout system are allowed to contain them

we can see that we've covered 1 - 3. Next up is widget templates.

Just like an ordinary block, a widget can be rendered using a phtml template. Additionally, using the UI rendering features, we can make templates a **customizable** feature of our widget.

Let's make our YouTube widget a template block. First, we'll alter our class so it inherits from the core template block and we'll remove the hard coded `toHtml` method.

```
#File: app/code/local/Nofrills/Booklayout/Block/Youtube.php
<?php
class Nofrills_Booklayout_Block_Youtube extends Mage_Core_Block_Template
implements Mage_Widget_Block_Interface
{
}
}
```

Next, we'll add the following parameter to our widget config

```
<parameters>
  <!-- ... -->
  <template>
    <required>1</required>
    <visible>0</visible>
    <value>youtube.phtml</value>
    <label>Frontend Template</label>
    <type>text</type>
  </template>
  <!-- ... -->
</parameters>
```

Finally, we'll add the `youtube.phtml` to our theme's template folder. We're adding it to the default/default theme here, but if your site's using a different theme, make sure you put it in the appropriate location

```
<!-- #File: app/design/frontend/default/default/template/youtube.phtml -->
<h2>Rick</h2>
<object width="640" height="505">
  <param name="movie" value="http://www.youtube.com/v/'<?php
echo $this->getVideoId();?>?fs=1&hl=en_US"></param>
  <param name="allowFullScreen" value="true"></param>
  <param name="allowscriptaccess" value="always"></param>
  <embed src="http://www.youtube.com/v/'<?php
echo $this->getVideoId();?>?fs=1&hl=en_US"
  type="application/x-shockwave-flash" allowscriptaccess="always"
  allowfullscreen="true" width="640" height="505"></embed>
</object>
```

With all of the above in place (and a cache clear), re-insert your widget. You should get a widget tag with a template attribute

```
{<?php echo $this->getVideoId();?>?fs=1&hl=en_US"
  type="application/x-shockwave-flash" allowscriptaccess="always"
  allowfullscreen="true" width="640" height="505"></embed>
</object>
}
```

Reload your frontend page and your configured YouTube video should render the same as before.

Because template blocks store their template as a regular block data parameter, all we're really doing here is adding a new widget data parameter named `<template/>`. We hard coded a value (by using an invisible data field), but there's no reason we couldn't make it a truly configurable value. Give the following a try in your widget config

```
<template>
  <required>1</required>
  <visible>1</visible>
  <value>youtube.phtml</value>
  <label>Frontend Template</label>
  <type>select</type>

  <values>
    <as_video>
      <value>youtube.phtml</value>
      <label>Embed Video</label>
    </as_video>
    <as_link>
      <value>youtube-as-link.phtml</value>
      <label>Link Video</label>
    </as_link>
  </values>
</template>
```

Don't forget to add the new template to your theme

```
<?php
#File: app/design/frontend/default/default/template/youtube-as-link.phtml
?>
<a href="http://www.youtube.com/watch?v=?php
    echo $this->getVideoId();?>">Watch this!</a>
```

Clear your cache and reinsert your widget. You should now see a new drop-down menu allowing you to pick which template your widget should use, (see *Figure 7.4*)

The screenshot displays the Magento Admin configuration for a widget. It is divided into two main sections: 'Widget' and 'Widget Options'. In the 'Widget' section, the 'Widget Type' is set to 'YouTube Example Widget', and a description states 'This widget displays a YouTube video.' The 'Widget Options' section contains two fields: 'YouTube Video ID' with a text input field containing 'Enter ID Here', and 'Frontend Template' which is a dropdown menu. The dropdown menu is currently open, showing two options: 'Embed Video' (which is selected, indicated by a checkmark) and 'Link Video'.

Figure 7.4

While it may appear that the template tag is being treated as just another widget property, when we move outside of CMS based widgets and into Instance Widgets, we'll see that the Instance Widget engine treats this parameter specially.

7.6 Instance Widgets

If we look back on our list of five things that make a widget

1. Widgets are Magento Template Blocks
2. Widgets Contain Structured Data
3. Widgets Contain Rules for Building User Interfaces
4. Widgets are formally associated with a number of phtml template files
5. Widgets contain rules that say which blocks in the layout system are allowed to contain them

we can see our explorations have completely ignored number five. So far all we've done is insert a widget into a CMS content area. We also haven't met our core widget requirement, which is to allow a non-programming user to add a widget to **any** page on the site. This is where Instance Widgets enter the picture.

So far we've been creating one off widgets that can't be reused. For example, if we wanted to add the same video to multiple CMS pages, we'd need to manually insert it into each page. Then, if we wanted to **change** something about each widget (say, the ID of that video), we'd need to go to each individual page and edit the template directive tag

```
{{widget type="nofrills_booklayout/youtube" video_id="dQw4w9WgXcQ"}}
```

With Instance Widgets, we can create **and save** a widget with a specific set of data, and then insert that widget into multiple locations on the site. Then, if we later change the definition of that specific widget, it will be automatically updated throughout the site.

7.7 Creating an Instance Widget

Navigate to

```
CMS -> Widgets
```

in the Admin Console to see a list of all the widgets in your system. We're going to add a new one, so click on the **Add New Widget Instance** button

Instance Widget creation is a two step process. First, we need to select the widget type we're going to create, as well as which theme the widget will be added to. Select our **YouTube** example widget from the drop down menu, and pick the currently configured theme. We'll be assuming `default/default` for the following examples, (see *Figure 7.5*)

Settings	
Type	YouTube Example Widget
Design Package/Theme	default / default
<input type="button" value="Continue"/>	

Figure 7.5

Once you've done this, click on the **Continue** button.

You should now see a two tab editing interface; **Frontend Properties**, and **Widget Options**. Widget Options contains an editing form for all the data properties for a particular widget, (with the exception of templates). Click on this tab and add a video id, and then return to the Frontend Properties tab, (see *Figure 7.6*)

Widget Instance

- Frontend Properties
- Widget Options**

New Widget Instance

Widget Options

YouTube Video ID *

Figure 7.6

In Frontend Properties you have two option groups. The first allows you to select a Widget Instance Title, Assign a Store View, and set a Sort Order for the widget. The Widget Instance Title is used in the Admin Console when displaying information about the widget (i.e. the listing page), Store View allows you to specify which Magento Stores a widget appears in.

Let's save our widget with a title, and select *All Store Views*. Click on the **Save** button, and you'll be returned to the widget listing page. You should see your widget listed along with any others that have been added to your Magento system. Click on the widget row to edit it. You'll notice you've been brought directly to the second stage, and that the **Widget Type** and **Design Package/Theme** options are un-editable. Once you select these during widget creation they **cannot** be changed, (see *Figure 7.7*)

The screenshot shows the 'Frontend Properties' panel for a 'YouTube Example Widget'. It includes the following fields:

- Type:** A dropdown menu set to 'YouTube Example Widget'.
- Design Package/Theme:** A dropdown menu set to 'default / default'.
- Widget Instance Title ***: A text input field containing 'Rick Roll'.
- Assign to Store Views ***: A list box showing a hierarchy of store views: 'All Store Views', 'Main Website', 'Main Store', 'English', 'French', and 'German'.
- Sort Order:** A text input field containing '0'.

Below the Sort Order field, there is a small triangle icon and the text: 'Sort Order of widget instances in the same block reference'.

Figure 7.7

7.8 Inserting a Widget

Here's where Instance Widgets get interesting. At the bottom of the Instance Widget editing page, there's an empty option group named **Layout Update**. Click the **Add Layout Update** button, (see *Figure 7.8*)

The screenshot shows the 'Layout Updates' section. It features an orange button labeled 'Add Layout Update' in the top right corner. Below this, there is a 'Display On *' label followed by a dropdown menu currently showing '-- Please Select --'. To the right of the dropdown is a red button with a trash icon labeled 'Remove Layout Update'.

Figure 7.8

This drop down menu contains several options, each one describing a particular set of, or a specific, Magento page. What we're configuring here is the page or pages we want to add our Widget Instance to. Select All Pages from this menu, (see *Figure 7.9*)

Two more menus have appeared. The first is **Block Reference**, the second is **Template**.

The first menu is defining **which block** you want to add your Widget Instance to. Select **Main Content Area**. The values in the second menu should look familiar to you. They're the templates we defined earlier. Select "Embed Video", and then Save you Widget Instance.

Figure 7.9

At this point you may receive a message at the top of your Magento admin that looks something like *Figure 7.10*.

Figure 7.10

This is Magento telling you that it has detected a change to the system that requires you to clear your cache. Do this, and then load any page in your site. You should now see your YouTube video added to the main content area.

7.9 Behind the Scenes

Open up your favorite MySQL browser, and run the following query against your database

```
select * from core_layout_update;
+-----+-----+-----+-----+
| layout_update_id | handle          | xml      | sort_order |
+-----+-----+-----+-----+
| 1                | default         | [...]    | 0          |
+-----+-----+-----+-----+
```

This table contains a list of Layout Update XML fragments, organized by handle. When building the Page Layout for any request, Magento will check this table **after** checking the loaded package layout. If it finds any matching handles, they'll be added to the Page Layout. When you select a value from the **Display On** menu, you're actually telling Magento **which** handles should be applied. When you save your Widget Instance, this table is updated. Because these updates add blocks to other block's that inherit from `core/text_list`, the widget blocks are automatically rendered.

If you take a look at the `Mage_Core_Model_Layout_Update::merge` method, you can see the additional call to `fetchDbLayoutUpdates`

```
public function merge($handle)
{
    $packageUpdatesStatus = $this->fetchPackageLayoutUpdates($handle);
    if (Mage::app()->isInstalled()) {
        $this->fetchDbLayoutUpdates($handle);
    }
    return $this;
}
```

Without an abstract Layout system, adding a feature like widgets would have required (at minimum) editing every single controller action, and inserting blocks into an unknown layout structure. This is the kind of power that sort of abstraction enables.

Similarly, the list of blocks which you insert a widget into is **not** hardcoded into a configuration system. It's generated automatically. Magento takes the handle indicated by the **Display On** drop down, and applies it to the Package Layout to create a temporary Page Layout. Then, rather than render a page, it looks at the top level body blocks for that layout to get a list of eligible blocks to display in the drop-down menu. This means if you add additional structural blocks to a page via means of custom XML layout files or `local.xml`, those blocks will show up in this menu. Again, this sort of thing becomes much easier to implement when using an abstract layout system.

7.10 Restricting Blocks.

Widget Instances have one more interesting feature. You can actually restrict **which** blocks a Widget Instance may be inserted into. Head back to your `widget.xml` file, and add the following to your widget's node

```
<widgets>
    <nofrills_layoutbook_youtube>
        <!-- ... -->
        <supported_blocks>
            <uniquely_named_node>
                <block_name>content</block_name>
                <template>
                    <unique_name_one>as_video</unique_name_one>
                    <unique_name_two>as_link</unique_name_two>
                </template>
            </uniquely_named_node>

            <another_uniquely_named_node>
                <block_name>left</block_name>
                <template>
                    <unique_name_one>as_video</unique_name_one>
                    <unique_name_two>as_link</unique_name_two>
                </template>
            </another_uniquely_named_node>
        </supported_blocks>
    </nofrills_layoutbook_youtube>
</widgets>
```

```
        </another_uniquely_named_node>

        </supported_blocks>
        <!-- ... -->
    </nofrills_layoutbook_youtube>
</widgets>
```

Clear your cache and reload the Widget Instance editing page. Your (formerly) long block menu now only allows you the choice of

```
Left Column
Main Content Area
```

In the absence of a `<supported_blocks/>` tag, Magento will display all eligible blocks for any particular page. However, with this node in place, it will scan each top level node for a sub-node named `<block_name>` and restrict your choices to those it finds. In our case above, the blocks are `content` and `left`. These names are the block's name as defined in the Layout Update XML fragment

```
<block type="core/text_list" name="content" as="content" translate="label">
```

You're also required to specify which, if any, templates are valid for a particular block. This context sensitive template is a powerful feature. Consider and add the following change to your `widget.xml` file

```
<uniquely_named_node>
    <block_name>content</block_name>
    <template>
        <unique_name_one>as_video</unique_name_one>
        <unique_name_two>as_link</unique_name_two>
    </template>
</uniquely_named_node>

<another_uniquely_named_node>
    <block_name>left</block_name>
    <template>
        <unique_name_two>as_link</unique_name_two>
    </template>
</another_uniquely_named_node>
```

Clear your cache and reload the widget editor. You'll notice that switching between the `content` and `left` block will result in your template choice being restricted. By using this technique, we've prevented a user from accidentally inserting a full video into the left hand column by restricting the templates they can use. In essence, each widget definition is an abstract content type, and you can control how it displays in each section of the site. This is only a few steps away from some of the advanced content management features of systems like Drupal.

The values being supplied for the templates (`as_link` and `as_video`) are the names of the nodes in the `<templates/>` block up in the `<parameters/>` section. This is what we've meant when we said Magento treats this node differently.

7.11 Per Theme Widget Config

There's another feature of the widget engine, in relation to Instances, that you should be aware of. It's possible to create fall back configurations for your widgets on a **per theme** basis. You've probably noticed the default themes each ship with a widget file.

```
app/design/frontend/default/default/etc/widget.xml
```

This file has the same format as the `widget.xml` in your module. Values in these files can be used to **override** the values for Instance Widgets. They **do not** apply to widgets inserted into CMS Pages or Static Blocks. In practice, this is done primarily for the supported blocks feature. Keeping with the generate principle of separating concerns, a general code module doesn't, technically, know which blocks or templates are going to be available for it. By keeping this information in each theme (Magento's default widgets ship with all the `<supported.blocks/>` information in the theme configs), Magento ensures that any themes which add custom `core/text_list` blocks also have the ability to allow or deny widgets access to these blocks.

7.12 Wrap Up

And that, in a nutshell, is widgets. We chose to end this book with widgets, because they appear to be the path forward for Magento content and layout management. The abstract layout system described in this book is stepping stone towards larger, more robust content and layout management for Magento. Less than four years old, Magento is dominating the ecommerce landscape like no other system. We hope the knowledge and techniques provided here will help you tame your Magento systems, and allow you to spend less time being confused by code, and more time serving your customers and building your businesses.

Visit <http://www.pulsestorm.net/nofrills-layout-chapter-seven> to join the discussion online.

Appendix A

Magento Block Hierarchy

This tree is a directory style hierarchy of every block class in Magento CE 1.4.2.0.

```
-- Varien_Object
|-- Mage_Core_Block_Abstract
|   |-- Mage_Adminhtml_Block_Urlrewrite_Link
|   |-- Mage_CatalogSearch_Block_Autocomplete
|   |-- Mage_Catalog_Block_Product_Price_Template
|   |-- Mage_Cms_Block_Block
|   |-- Mage_Cms_Block_Page
|   |-- Mage_Core_Block_Flush
|   |-- Mage_Core_Block_Html_Select
|   |   |-- Mage_Adminhtml_Block_Html_Select
|   |   |-- Mage_CatalogInventory_Block_Adminhtml_Form_Field_Customergroup
|   |-- Mage_Core_Block_Html_Select
|   |-- Mage_Core_Block_Profiler
|   |-- Mage_Core_Block_Template
|   |   |-- Mage_Adminhtml_Block_Abstract
|   |   |   |-- Mage_Adminhtml_Block_Catalog_Product_Frontend_Product_Watermark
|   |   |   |-- Mage_Adminhtml_Block_Customer_Edit_Renderer_Newpass
|   |   |   |-- Mage_Adminhtml_Block_Customer_Edit_Renderer_Region
|   |   |   |-- Mage_Adminhtml_Block_Extensions_Custom_Edit_Tab_Load
|   |   |   |-- Mage_Adminhtml_Block_Promo_Widget_Chooser_Daterange
|   |   |   |-- Mage_Adminhtml_Block_System_Config_Form_Field
|   |   |   |   |-- Mage_Adminhtml_Block_Catalog_Form_Renderer_Config_DateFieldsOrder
|   |   |   |   |-- Mage_Adminhtml_Block_Catalog_Form_Renderer_Config_YearRange
|   |   |   |   |-- Mage_Adminhtml_Block_Report_Config_Form_Field_MtdStart
|   |   |   |   |-- Mage_Adminhtml_Block_Report_Config_Form_Field_YtdStart
|   |   |   |   |-- Mage_Adminhtml_Block_System_Config_Form_Field_Array_Abstract
|   |   |   |   |   |-- Mage_Adminhtml_Block_System_Config_Form_Field_Regexceptions
|   |   |   |   |   |-- Mage_CatalogInventory_Block_Adminhtml_Form_Field_Minsaleqty
|   |   |   |   |-- Mage_Adminhtml_Block_System_Config_Form_Field_Array_Abstract
|   |   |   |   |-- Mage_Adminhtml_Block_System_Config_Form_Field_Datetime
|   |   |   |   |-- Mage_Adminhtml_Block_System_Config_Form_Field_Notification
|   |   |   |   |-- Mage_Adminhtml_Block_System_Config_Form_Field_Select_Flatcatalog
|   |   |   |   |-- Mage_Adminhtml_Block_System_Config_Form_Field_Select_Flatproduct
|   |   |   |-- Mage_Directory_Block_Adminhtml_Frontend_Currency_Base
```


APPENDIX A. MAGENTO BLOCK HIERARCHY

```
| | | | | -- Mage_Adminhtml_Block_Report_Product_Downloads_Renderer_Purchases
| | | | | -- Mage_Adminhtml_Block_Review_Grid_Renderer_Type
| | | | | -- Mage_Adminhtml_Block_Sales_Reorder_Renderer_Action
| | | | | -- Mage_Adminhtml_Block_Sitemap_Grid_Renderer_Link
| | | | | -- Mage_Adminhtml_Block_Sitemap_Grid_Renderer_Time
| | | | | -- Mage_Adminhtml_Block_System_Convert_Profile_Edit_Renderer_Action
| | | | | -- Mage_Adminhtml_Block_System_Email_Template_Grid_Renderer_Sender
| | | | | -- Mage_Adminhtml_Block_System_Email_Template_Grid_Renderer_Type
| | | | | -- Mage_Adminhtml_Block_System_Store_Grid_Render_Group
| | | | | -- Mage_Adminhtml_Block_System_Store_Grid_Render_Store
| | | | | -- Mage_Adminhtml_Block_System_Store_Grid_Render_Website
| | | | | -- Mage_Adminhtml_Block_Tax_Rate_Grid_Renderer_Data
| | | | | -- Mage_Adminhtml_Block_Widget_Grid_Column_Renderer_Checkbox
| | | | | | -- Mage_Adminhtml_Block_Catalog_Product_Edit_Tab_Super_Config_Grid_Renderer
| | | | | | -- Mage_Adminhtml_Block_Sales_Order_Create_Search_Grid_Renderer_Giftmessage
| | | | | | -- Mage_Adminhtml_Block_Widget_Grid_Column_Renderer_Massaction
| | | | | -- Mage_Adminhtml_Block_Widget_Grid_Column_Renderer_Checkbox
| | | | | -- Mage_Adminhtml_Block_Widget_Grid_Column_Renderer_Concat
| | | | | -- Mage_Adminhtml_Block_Widget_Grid_Column_Renderer_Country
| | | | | | -- Mage_Adminhtml_Block_Tax_Rate_Grid_Renderer_Country
| | | | | -- Mage_Adminhtml_Block_Widget_Grid_Column_Renderer_Country
| | | | | -- Mage_Adminhtml_Block_Widget_Grid_Column_Renderer_Currency
| | | | | | -- Mage_Adminhtml_Block_Report_Grid_Column_Renderer_Currency
| | | | | -- Mage_Adminhtml_Block_Widget_Grid_Column_Renderer_Currency
| | | | | -- Mage_Adminhtml_Block_Widget_Grid_Column_Renderer_Date
| | | | | | -- Mage_Adminhtml_Block_Report_Sales_Grid_Column_Renderer_Date
| | | | | -- Mage_Adminhtml_Block_Widget_Grid_Column_Renderer_Date
| | | | | -- Mage_Adminhtml_Block_Widget_Grid_Column_Renderer_Datetime
| | | | | -- Mage_Adminhtml_Block_Widget_Grid_Column_Renderer_Input
| | | | | -- Mage_Adminhtml_Block_Widget_Grid_Column_Renderer_Ip
| | | | | -- Mage_Adminhtml_Block_Widget_Grid_Column_Renderer_Longtext
| | | | | -- Mage_Adminhtml_Block_Widget_Grid_Column_Renderer_Number
| | | | | | -- Mage_Adminhtml_Block_Report_Grid_Column_Renderer_Blanknumber
| | | | | -- Mage_Adminhtml_Block_Widget_Grid_Column_Renderer_Number
| | | | | -- Mage_Adminhtml_Block_Widget_Grid_Column_Renderer_Price
| | | | | | -- Mage_Adminhtml_Block_Sales_Order_Create_Search_Grid_Renderer_Price
| | | | | -- Mage_Adminhtml_Block_Widget_Grid_Column_Renderer_Price
| | | | | -- Mage_Adminhtml_Block_Widget_Grid_Column_Renderer_Radio
| | | | | -- Mage_Adminhtml_Block_Widget_Grid_Column_Renderer_Select
| | | | | -- Mage_Adminhtml_Block_Widget_Grid_Column_Renderer_Store
| | | | | -- Mage_Adminhtml_Block_Widget_Grid_Column_Renderer_Text
| | | | | | -- Mage_Adminhtml_Block_Widget_Grid_Column_Renderer_Action
| | | | | | | -- Mage_Adminhtml_Block_Extensions_Local_Grid_Renderer_Action
| | | | | | | -- Mage_Adminhtml_Block_Extensions_Remote_Grid_Renderer_Action
| | | | | | | -- Mage_Adminhtml_Block_Newsletter_Queue_Grid_Renderer_Action
| | | | | | | -- Mage_Adminhtml_Block_Newsletter_Template_Grid_Renderer_Action
| | | | | | | -- Mage_Adminhtml_Block_Sitemap_Grid_Renderer_Action
| | | | | | | -- Mage_Adminhtml_Block_System_Email_Template_Grid_Renderer_Action
| | | | | | -- Mage_Adminhtml_Block_Widget_Grid_Column_Renderer_Action
| | | | | | -- Mage_Adminhtml_Block_Widget_Grid_Column_Renderer_Options
| | | | | -- Mage_Adminhtml_Block_Widget_Grid_Column_Renderer_Text
| | | | | -- Mage_Adminhtml_Block_Widget_Grid_Column_Renderer_Theme
| | | | | -- Mage_Adminhtml_Block_Widget_Grid_Column_Renderer_Wrapline
| | | | | -- Mage_GoogleBase_Block_Adminhtml_Items_Renderer_Id
| | | | | | -- Mage_GoogleBase_Block_Adminhtml_Types_Renderer_Country
| | | | | -- Mage_Adminhtml_Block_Widget_Grid_Column_Renderer_Abstract
| | | | | -- Mage_Paypal_Block_Adminhtml_System_Config_Fieldset_Global
```

APPENDIX A. MAGENTO BLOCK HIERARCHY

```
| | | | -- Mage_Paypal_Block_Adminhtml_System_Config_Fieldset_Hint
| | | | -- Mage_Sales_Block_Adminhtml_Billing_Agreement_View_Tab_Info
| | | | -- Mage_Sales_Block_Adminhtml_Recurring_Profile_Edit_Form
| | | | -- Mage_Adminhtml_Block_Abstract
| | | | -- Mage_Adminhtml_Block_Catalog_Product_Edit_Tab_Ajax_Serializer
| | | | -- Mage_Adminhtml_Block_Sales_Order_Create_Load
| | | | -- Mage_Adminhtml_Block_Tax_Rate_Title
| | | | -- Mage_Adminhtml_Block_Template
| | | | | -- Mage_Adminhtml_Block_Api_Buttons
| | | | | -- Mage_Adminhtml_Block_Api_Roles
| | | | | -- Mage_Adminhtml_Block_Api_Users
| | | | | -- Mage_Adminhtml_Block_Backup
| | | | | -- Mage_Adminhtml_Block_Cache_Additional
| | | | | -- Mage_Adminhtml_Block_Cache_Notifications
| | | | | -- Mage_Adminhtml_Block_Catalog
| | | | | -- Mage_Adminhtml_Block_Catalog_Category_Abstract
| | | | | | -- Mage_Adminhtml_Block_Catalog_Category_Edit_Form
| | | | | | -- Mage_Adminhtml_Block_Catalog_Category_Tree
| | | | | | | -- Mage_Adminhtml_Block_Catalog_Category_Checkboxes_Tree
| | | | | | | -- Mage_Adminhtml_Block_Catalog_Category_Widget_Chooser
| | | | | | | -- Mage_Adminhtml_Block_Catalog_Product_Edit_Tab_Categories
| | | | | | | -- Mage_Adminhtml_Block_Catalog_Category_Tree
| | | | | | | -- Mage_Adminhtml_Block_Urlrewrite_Category_Tree
| | | | | -- Mage_Adminhtml_Block_Catalog_Category_Abstract
| | | | | -- Mage_Adminhtml_Block_Catalog_Product_Attribute_Set_Main
| | | | | -- Mage_Adminhtml_Block_Catalog_Product_Attribute_Set_Main_Tree_Attribute
| | | | | -- Mage_Adminhtml_Block_Catalog_Product_Attribute_Set_Main_Tree_Group
| | | | | -- Mage_Adminhtml_Block_Catalog_Product_Attribute_Set_Toolbar_Add
| | | | | -- Mage_Adminhtml_Block_Catalog_Product_Attribute_Set_Toolbar_Main
| | | | | -- Mage_Adminhtml_Block_Catalog_Product_Edit_Js
| | | | | -- Mage_Adminhtml_Block_Catalog_Product_Edit_Tab_Alerts
| | | | | -- Mage_Adminhtml_Block_Catalog_Product_Widget_Chooser_Container
| | | | | -- Mage_Adminhtml_Block_Cms_Wysiwyg_Images_Content_Files
| | | | | -- Mage_Adminhtml_Block_Cms_Wysiwyg_Images_Content_Newfolder
| | | | | -- Mage_Adminhtml_Block_Cms_Wysiwyg_Images_Tree
| | | | | -- Mage_Adminhtml_Block_Customer_Edit_Tab_Carts
| | | | | -- Mage_Adminhtml_Block_Customer_Edit_Tab_View
| | | | | -- Mage_Adminhtml_Block_Customer_Edit_Tab_View_Sales
| | | | | -- Mage_Adminhtml_Block_Customer_Online
| | | | | -- Mage_Adminhtml_Block_Dashboard
| | | | | -- Mage_Adminhtml_Block_Denied
| | | | | -- Mage_Adminhtml_Block_Newsletter_Problem
| | | | | -- Mage_Adminhtml_Block_Newsletter_Queue
| | | | | -- Mage_Adminhtml_Block_Newsletter_Queue_Edit
| | | | | -- Mage_Adminhtml_Block_Newsletter_Subscriber
| | | | | -- Mage_Adminhtml_Block_Newsletter_Template
| | | | | -- Mage_Adminhtml_Block_Notification_Baseurl
| | | | | -- Mage_Adminhtml_Block_Notification_Security
| | | | | -- Mage_Adminhtml_Block_Notification_Survey
| | | | | -- Mage_Adminhtml_Block_Notification_Toolbar
| | | | | | -- Mage_Adminhtml_Block_Notification_Window
| | | | | -- Mage_Adminhtml_Block_Notification_Toolbar
| | | | | -- Mage_Adminhtml_Block_Page
| | | | | -- Mage_Adminhtml_Block_Page_Footer
| | | | | -- Mage_Adminhtml_Block_Page_Header
| | | | | -- Mage_Adminhtml_Block_Page_Menu
| | | | | -- Mage_Adminhtml_Block_Page_Notices
```

APPENDIX A. MAGENTO BLOCK HIERARCHY

```
| | | |-- Mage_Adminhtml_Block_Permissions_Buttons
| | | |-- Mage_Adminhtml_Block_Permissions_Roles
| | | |-- Mage_Adminhtml_Block_Permissions_Usernroles
| | | |-- Mage_Adminhtml_Block_Permissions_Users
| | | |-- Mage_Adminhtml_Block_Poll_Edit_Tab_Answers_List
| | | |-- Mage_Adminhtml_Block_Report_Wishlist
| | | |-- Mage_Adminhtml_Block_Review_Rating_Detailed
| | | |-- Mage_Adminhtml_Block_Review_Rating_Summary
| | | |-- Mage_Adminhtml_Block_Sales
| | | |-- Mage_Adminhtml_Block_Sales_Items_Abstract
| | | | |-- Mage_Adminhtml_Block_Sales_Items_Renderer_Configurable
| | | | |-- Mage_Adminhtml_Block_Sales_Items_Renderer_Default
| | | | | |-- Mage_Bundle_Block_Adminhtml_Sales_Order_Items_Renderer
| | | | |-- Mage_Adminhtml_Block_Sales_Items_Renderer_Default
| | | | |-- Mage_Adminhtml_Block_Sales_Order_Creditmemo_Create_Items
| | | | |-- Mage_Adminhtml_Block_Sales_Order_Creditmemo_View_Items
| | | | |-- Mage_Adminhtml_Block_Sales_Order_Invoice_Create_Items
| | | | |-- Mage_Adminhtml_Block_Sales_Order_Invoice_View_Items
| | | | |-- Mage_Adminhtml_Block_Sales_Order_Shipment_Create_Items
| | | | |-- Mage_Adminhtml_Block_Sales_Order_Shipment_View_Items
| | | | |-- Mage_Adminhtml_Block_Sales_Order_View_Items
| | | | | |-- Mage_Adminhtml_Block_Sales_Order_View_Items_Renderer_Default
| | | | | | |-- Mage_Bundle_Block_Adminhtml_Sales_Order_View_Items_Renderer
| | | | | |-- Mage_Adminhtml_Block_Sales_Order_View_Items_Renderer_Default
| | | |-- Mage_Adminhtml_Block_Sales_Items_Abstract
| | | |-- Mage_Adminhtml_Block_Sales_Items_Column_Default
| | | | |-- Mage_Adminhtml_Block_Sales_Items_Column_Name
| | | | | |-- Mage_Adminhtml_Block_Sales_Items_Column_Name_Grouped
| | | | | | |-- Mage_Downloadable_Block_Adminhtml_Sales_Items_Column_Downloadable_Name
| | | | |-- Mage_Adminhtml_Block_Sales_Items_Column_Name
| | | | |-- Mage_Adminhtml_Block_Sales_Items_Column_Qty
| | | |-- Mage_Adminhtml_Block_Sales_Items_Column_Default
| | | |-- Mage_Adminhtml_Block_Sales_Order_Comments_View
| | | |-- Mage_Adminhtml_Block_Sales_Order_Create_Totals_Table
| | | |-- Mage_Adminhtml_Block_Sales_Order_Creditmemo_Create_Adjustments
| | | |-- Mage_Adminhtml_Block_Sales_Order_Invoice_Create_Tracking
| | | |-- Mage_Adminhtml_Block_Sales_Order_Payment
| | | |-- Mage_Adminhtml_Block_Sales_Order_Shipment_Create_Tracking
| | | |-- Mage_Adminhtml_Block_Sales_Order_Shipment_Tracking_Info
| | | |-- Mage_Adminhtml_Block_Sales_Order_Shipment_View_Tracking
| | | |-- Mage_Adminhtml_Block_Sales_Order_View_Form
| | | |-- Mage_Adminhtml_Block_Sales_Order_View_History
| | | |-- Mage_Adminhtml_Block_Sales_Order_View_Tab_History
| | | |-- Mage_Adminhtml_Block_Store_Switcher
| | | | |-- Mage_Adminhtml_Block_Catalog_Product_Edit_Tab_Websites
| | | | |-- Mage_Adminhtml_Block_Sales_Order_Create_Store_Select
| | | | |-- Mage_Adminhtml_Block_Tag_Store_Switcher
| | | | |-- Mage_GoogleBase_Block_Adminhtml_Store_Switcher
| | | |-- Mage_Adminhtml_Block_Store_Switcher
| | | |-- Mage_Adminhtml_Block_System_Config_Switcher
| | | |-- Mage_Adminhtml_Block_System_Convert_Profile_Edit_Tab_Run
| | | |-- Mage_Adminhtml_Block_System_Currency
| | | |-- Mage_Adminhtml_Block_System_Currency_Rate_Matrix
| | | |-- Mage_Adminhtml_Block_System_Currency_Rate_Services
| | | |-- Mage_Adminhtml_Block_System_Design
| | | |-- Mage_Adminhtml_Block_System_Email_Template
| | | |-- Mage_Adminhtml_Block_System_Store_Delete_Group
```


APPENDIX A. MAGENTO BLOCK HIERARCHY

```
| | | | -- Mage_Adminhtml_Block_System_Store_Delete_Website
| | | | -- Mage_Adminhtml_Block_Tag
| | | | -- Mage_Adminhtml_Block_Tag_Pending
| | | | -- Mage_Adminhtml_Block_Tax_Rate_Toolbar_Add
| | | | -- Mage_Adminhtml_Block_Tax_Rate_Toolbar_Save
| | | | -- Mage_Adminhtml_Block_Widget
| | | | |-- Mage_Adminhtml_Block_Catalog_Product_Attribute_New_Product_Created
| | | | |-- Mage_Adminhtml_Block_Catalog_Product_Created
| | | | |-- Mage_Adminhtml_Block_Catalog_Product_Edit
| | | | |-- Mage_Adminhtml_Block_Catalog_Product_Edit_Action_Attribute
| | | | |-- Mage_Adminhtml_Block_Catalog_Product_Edit_Action_Attribute_Tab_Inventory
| | | | |-- Mage_Adminhtml_Block_Catalog_Product_Edit_Action_Attribute_Tab_Websites
| | | | |-- Mage_Adminhtml_Block_Catalog_Product_Edit_Tab_Inventory
| | | | |-- Mage_Adminhtml_Block_Catalog_Product_Edit_Tab_Options
| | | | |-- Mage_Adminhtml_Block_Catalog_Product_Edit_Tab_Options_Option
| | | | |-- Mage_Adminhtml_Block_Catalog_Product_Edit_Tab_Options_Type_Abstract
| | | | |-- Mage_Adminhtml_Block_Catalog_Product_Edit_Tab_Price_Tier
| | | | |-- Mage_Adminhtml_Block_Catalog_Product_Edit_Tab_Super_Config
| | | | |-- Mage_Adminhtml_Block_Catalog_Product_Helper_Form_Gallery_Content
| | | | |-- Mage_Adminhtml_Block_Dashboard_Abstract
| | | | | |-- Mage_Adminhtml_Block_Dashboard_Bar
| | | | | | |-- Mage_Adminhtml_Block_Dashboard_Sales
| | | | | | |-- Mage_Adminhtml_Block_Dashboard_Totals
| | | | | |-- Mage_Adminhtml_Block_Dashboard_Bar
| | | | | | |-- Mage_Adminhtml_Block_Dashboard_Graph
| | | | | | |-- Mage_Adminhtml_Block_Dashboard_Tab_Amounts
| | | | | | |-- Mage_Adminhtml_Block_Dashboard_Tab_Orders
| | | | | |-- Mage_Adminhtml_Block_Dashboard_Graph
| | | | |-- Mage_Adminhtml_Block_Dashboard_Abstract
| | | | |-- Mage_Adminhtml_Block_Extensions_Config_Edit
| | | | |-- Mage_Adminhtml_Block_Media_Editor
| | | | |-- Mage_Adminhtml_Block_Media_Uploader
| | | | | |-- Mage_Adminhtml_Block_Cms_Wysiwyg_Images_Content_Uploader
| | | | |-- Mage_Adminhtml_Block_Media_Uploader
| | | | |-- Mage_Adminhtml_Block_Newsletter_Queue_Preview
| | | | |-- Mage_Adminhtml_Block_Newsletter_Template_Edit
| | | | |-- Mage_Adminhtml_Block_Newsletter_Template_Preview
| | | | |-- Mage_Adminhtml_Block_Sales_Order_Abstract
| | | | | |-- Mage_Adminhtml_Block_Sales_Order_Creditmemo_Create_Form
| | | | | |-- Mage_Adminhtml_Block_Sales_Order_Creditmemo_View_Form
| | | | | |-- Mage_Adminhtml_Block_Sales_Order_Invoice_Create_Form
| | | | | |-- Mage_Adminhtml_Block_Sales_Order_Invoice_View_Form
| | | | | |-- Mage_Adminhtml_Block_Sales_Order_Shipment_Create_Form
| | | | | |-- Mage_Adminhtml_Block_Sales_Order_Shipment_View_Form
| | | | | |-- Mage_Adminhtml_Block_Sales_Order_Totalbar
| | | | | |-- Mage_Adminhtml_Block_Sales_Order_View_Info
| | | | | |-- Mage_Adminhtml_Block_Sales_Order_View_Tab_Info
| | | | |-- Mage_Adminhtml_Block_Sales_Order_Abstract
| | | | |-- Mage_Adminhtml_Block_Sales_Order_Create_Abstract
| | | | | |-- Mage_Adminhtml_Block_Sales_Order_Create_Billing_Method
| | | | | |-- Mage_Adminhtml_Block_Sales_Order_Create_Comment
| | | | | |-- Mage_Adminhtml_Block_Sales_Order_Create_Coupons
| | | | | |-- Mage_Adminhtml_Block_Sales_Order_Create_Coupons_Form
| | | | | |-- Mage_Adminhtml_Block_Sales_Order_Create_Customer
| | | | | |-- Mage_Adminhtml_Block_Sales_Order_Create_Data
| | | | | |-- Mage_Adminhtml_Block_Sales_Order_Create_Form
| | | | | |-- Mage_Adminhtml_Block_Sales_Order_Create_Form_Abstract
```


Prepared for Compaa Peruana de E-commerce S.A.; Copyright ©2011 Pulsar
Storm LLC

Prepared for Compaa Peruana de E-commerce S.A.; Copyright ©2011 Pulse29 Storm LLC

APPENDIX A. MAGENTO BLOCK HIERARCHY

```
| | | | | -- Mage_Adminhtml_Block_Catalog_Form_Renderer_Fieldset_Element
| | | | | -- Mage_Adminhtml_Block_Catalog_Form_Renderer_Googleoptimizer_Import
| | | | | -- Mage_Adminhtml_Block_System_Variable_Form_Renderer_Fieldset_Element
| | | | | -- Mage_GoogleBase_Block_Adminhtml_Types_Edit_Attributes
| | | | | -- Mage_GoogleOptimizer_Block_Adminhtml_Cms_Page_Edit_Renderer_Conversion
| | | | | -- Mage_Adminhtml_Block_Widget_Form_Renderer_Fieldset_Element
| | | | | -- Mage_Compiler_Block_Process
| | | | | -- Mage_Downloadable_Block_Adminhtml_Catalog_Product_Edit_Tab_Downloadable_Links
| | | | | -- Mage_Eav_Block_Adminhtml_Attribute_Edit_Js
| | | | | -- Mage_GoogleBase_Block_Adminhtml_Captcha
| | | | | -- Mage_GoogleOptimizer_Block_Adminhtml_Cms_Page_Edit_Enable
| | | | | -- Mage_GoogleOptimizer_Block_Js
| | | | | -- Mage_Index_Block_Adminhtml_Notifications
| | | | | -- Mage_Sales_Block_Adminhtml_Billing_Agreement_View_Form
| | | | | -- Mage_Widget_Block_Adminhtml_Widget_Chooser
| | | | | -- Mage_Widget_Block_Adminhtml_Widget_Instance_Edit_Tab_Main_Layout
| | | | | -- Mage_Adminhtml_Block_Template
| | | | | -- Mage_Adminhtml_Block_Urlrewrite_Selector
| | | | | -- Mage_Adminhtml_Block_Widget_Grid_Serializer
| | | | | -- Mage_CatalogInventory_Block_Qtyincrements
| | | | | -- Mage_CatalogInventory_Block_Stockqty_Abstract
| | | | | | -- Mage_CatalogInventory_Block_Stockqty_Default
| | | | | | | -- Mage_CatalogInventory_Block_Stockqty_Composite
| | | | | | | | -- Mage_CatalogInventory_Block_Stockqty_Type_Configurable
| | | | | | | | -- Mage_CatalogInventory_Block_Stockqty_Type_Grouped
| | | | | | | | -- Mage_CatalogInventory_Block_Stockqty_Composite
| | | | | | | | -- Mage_CatalogInventory_Block_Stockqty_Default
| | | | | -- Mage_CatalogInventory_Block_Stockqty_Abstract
| | | | | -- Mage_CatalogSearch_Block_Advanced_Form
| | | | | -- Mage_CatalogSearch_Block_Advanced_Result
| | | | | -- Mage_CatalogSearch_Block_Result
| | | | | -- Mage_CatalogSearch_Block_Term
| | | | | -- Mage_Catalog_Block_Breadcrumbs
| | | | | -- Mage_Catalog_Block_Category_View
| | | | | -- Mage_Catalog_Block_Layer_Filter_Abstract
| | | | | | -- Mage_Catalog_Block_Layer_Filter_Attribute
| | | | | | | -- Mage_CatalogSearch_Block_Layer_Filter_Attribute
| | | | | | | -- Mage_Catalog_Block_Layer_Filter_Attribute
| | | | | | | -- Mage_Catalog_Block_Layer_Filter_Category
| | | | | | | -- Mage_Catalog_Block_Layer_Filter_Decimal
| | | | | | | -- Mage_Catalog_Block_Layer_Filter_Price
| | | | | -- Mage_Catalog_Block_Layer_Filter_Abstract
| | | | | -- Mage_Catalog_Block_Layer_State
| | | | | -- Mage_Catalog_Block_Layer_View
| | | | | | -- Mage_CatalogSearch_Block_Layer
| | | | | -- Mage_Catalog_Block_Layer_View
| | | | | -- Mage_Catalog_Block_Navigation
| | | | | -- Mage_Catalog_Block_Product
| | | | | -- Mage_Catalog_Block_Product_Abstract
| | | | | | -- Mage_Bundle_Block_Catalog_Product_List_Partof
| | | | | | -- Mage_Catalog_Block_Product_Compare_Abstract
| | | | | | | -- Mage_Catalog_Block_Product_Compare_List
| | | | | | | | -- Mage_Catalog_Block_Product_Compare_Sidebar
| | | | | | | -- Mage_Catalog_Block_Product_Compare_Abstract
| | | | | | | -- Mage_Catalog_Block_Product_List
| | | | | | | -- Mage_Catalog_Block_Product_List_Promotion
| | | | | | | -- Mage_Catalog_Block_Product_List_Random
```

APPENDIX A. MAGENTO BLOCK HIERARCHY

```
| | | | -- Mage_Catalog_Block_Product_List
| | | | -- Mage_Catalog_Block_Product_List_Crosssell
| | | | -- Mage_Catalog_Block_Product_List_Related
| | | | -- Mage_Catalog_Block_Product_List_Upsell
| | | | -- Mage_Catalog_Block_Product_New
| | | | ' -- Mage_Catalog_Block_Product_Widget_New
| | | | -- Mage_Catalog_Block_Product_New
| | | | -- Mage_Catalog_Block_Product_Send
| | | | -- Mage_Catalog_Block_Product_View
| | | | | -- Mage_Bundle_Block_Catalog_Product_View
| | | | | -- Mage_Review_Block_Product_View
| | | | | ' -- Mage_Review_Block_Product_View_List
| | | | ' -- Mage_Review_Block_Product_View
| | | | -- Mage_Catalog_Block_Product_View
| | | | -- Mage_Catalog_Block_Product_View_Abstract
| | | | | -- Mage_Bundle_Block_Catalog_Product_View_Type_Bundle
| | | | | -- Mage_Catalog_Block_Product_View_Type_Media
| | | | | -- Mage_Catalog_Block_Product_View_Type_Configurable
| | | | | -- Mage_Catalog_Block_Product_View_Type_Grouped
| | | | | -- Mage_Catalog_Block_Product_View_Type_Simple
| | | | | -- Mage_Catalog_Block_Product_View_Type_Virtual
| | | | | ' -- Mage_Downloadable_Block_Catalog_Product_View_Type
| | | | ' -- Mage_Catalog_Block_Product_View_Type_Virtual
| | | | -- Mage_Catalog_Block_Product_View_Abstract
| | | | -- Mage_Checkout_Block_Cart_Crosssell
| | | | -- Mage_Downloadable_Block_Catalog_Product_Links
| | | | -- Mage_Downloadable_Block_Catalog_Product_Samples
| | | | -- Mage_Reports_Block_Product_Abstract
| | | | | -- Mage_Reports_Block_Product_Compared
| | | | | ' -- Mage_Reports_Block_Product_Widget_Compared
| | | | | -- Mage_Reports_Block_Product_Compared
| | | | | -- Mage_Reports_Block_Product_Viewed
| | | | | ' -- Mage_Reports_Block_Product_Widget_Viewed
| | | | ' -- Mage_Reports_Block_Product_Viewed
| | | | -- Mage_Reports_Block_Product_Abstract
| | | | -- Mage_Review_Block_Customer_View
| | | | -- Mage_Review_Block_View
| | | | -- Mage_Tag_Block_Customer_View
| | | | -- Mage_Tag_Block_Product_Result
| | | | -- Mage_Wishlist_Block_Abstract
| | | | | -- Mage_Rss_Block_Wishlist
| | | | | -- Mage_Wishlist_Block_Customer_Sidebar
| | | | | -- Mage_Wishlist_Block_Customer_Wishlist
| | | | | -- Mage_Wishlist_Block_Share_Email_Items
| | | | | ' -- Mage_Wishlist_Block_Share_Wishlist
| | | | ' -- Mage_Wishlist_Block_Abstract
| | | | -- Mage_Catalog_Block_Product_Abstract
| | | | -- Mage_Catalog_Block_Product_Gallery
| | | | -- Mage_Catalog_Block_Product_List_Toolbar
| | | | -- Mage_Catalog_Block_Product_Price
| | | | | -- Mage_Bundle_Block_Catalog_Product_Price
| | | | | | -- Mage_Bundle_Block_Catalog_Product_View_Type_Bundle_Option
| | | | | | | -- Mage_Bundle_Block_Catalog_Product_View_Type_Bundle_Option_Checkbox
| | | | | | | -- Mage_Bundle_Block_Catalog_Product_View_Type_Bundle_Option_Multi
| | | | | | | -- Mage_Bundle_Block_Catalog_Product_View_Type_Bundle_Option_Radio
| | | | | | ' -- Mage_Bundle_Block_Catalog_Product_View_Type_Bundle_Option_Select
| | | | ' -- Mage_Bundle_Block_Catalog_Product_View_Type_Bundle_Option
```

APPENDIX A. MAGENTO BLOCK HIERARCHY

```
| | | '-- Mage_Bundle_Block_Catalog_Product_Price
| | | |-- Mage_Catalog_Block_Product_Price
| | | |-- Mage_Catalog_Block_Product_View_Additional
| | | |-- Mage_Catalog_Block_Product_View_Attributes
| | | |-- Mage_Catalog_Block_Product_View_Description
| | | |-- Mage_Catalog_Block_Product_View_Options
| | | |-- Mage_Catalog_Block_Product_View_Options_Abstract
| | | | |-- Mage_Catalog_Block_Product_View_Options_Type_Date
| | | | |-- Mage_Catalog_Block_Product_View_Options_Type_Default
| | | | |-- Mage_Catalog_Block_Product_View_Options_Type_File
| | | | |-- Mage_Catalog_Block_Product_View_Options_Type_Select
| | | | '-- Mage_Catalog_Block_Product_View_Options_Type_Text
| | | |-- Mage_Catalog_Block_Product_View_Options_Abstract
| | | |-- Mage_Catalog_Block_Product_View_Price
| | | |-- Mage_Catalog_Block_Product_View_Tabs
| | | |-- Mage_Catalog_Block_Seo_Sitemap_Abstract
| | | | |-- Mage_Catalog_Block_Seo_Sitemap_Category
| | | | | '-- Mage_Catalog_Block_Seo_Sitemap_Tree_Category
| | | | |-- Mage_Catalog_Block_Seo_Sitemap_Category
| | | | '-- Mage_Catalog_Block_Seo_Sitemap_Product
| | | |-- Mage_Catalog_Block_Seo_Sitemap_Abstract
| | | |-- Mage_Centinel_Block_Authentication
| | | |-- Mage_Centinel_Block_Authentication_Complete
| | | |-- Mage_Centinel_Block_Authentication_Start
| | | |-- Mage_Centinel_Block_Logo
| | | |-- Mage_Checkout_Block_Agreements
| | | |-- Mage_Checkout_Block_Cart_Abstract
| | | | |-- Mage_Checkout_Block_Cart
| | | | |-- Mage_Checkout_Block_Cart_Coupon
| | | | |-- Mage_Checkout_Block_Cart_Shipping
| | | | |-- Mage_Checkout_Block_Cart_Sidebar
| | | | |-- Mage_Checkout_Block_Cart_Totals
| | | | | |-- Mage_Checkout_Block_Total_Default
| | | | | | |-- Mage_Checkout_Block_Total_Nominal
| | | | | | |-- Mage_Checkout_Block_Total_Tax
| | | | | | |-- Mage_Tax_Block_Checkout_Discount
| | | | | | |-- Mage_Tax_Block_Checkout_Grandtotal
| | | | | | |-- Mage_Tax_Block_Checkout_Shipping
| | | | | | |-- Mage_Tax_Block_Checkout_Subtotal
| | | | | | '-- Mage_Tax_Block_Checkout_Tax
| | | | | |-- Mage_Checkout_Block_Total_Default
| | | | | '-- Mage_Paypal_Block_Express_Review_Details
| | | | '-- Mage_Checkout_Block_Cart_Totals
| | | |-- Mage_Checkout_Block_Cart_Abstract
| | | |-- Mage_Checkout_Block_Cart_Item_Renderer
| | | | |-- Mage_Bundle_Block_Checkout_Cart_Item_Renderer
| | | | |-- Mage_Checkout_Block_Cart_Item_Renderer_Configurable
| | | | |-- Mage_Checkout_Block_Cart_Item_Renderer_Grouped
| | | | '-- Mage_Downloadable_Block_Checkout_Cart_Item_Renderer
| | | |-- Mage_Checkout_Block_Cart_Item_Renderer
| | | |-- Mage_Checkout_Block_Links
| | | |-- Mage_Checkout_Block_Multishipping_Abstract
| | | | |-- Mage_Checkout_Block_Multishipping_Address_Select
| | | | '-- Mage_Checkout_Block_Multishipping_Success
| | | |-- Mage_Checkout_Block_Multishipping_Abstract
| | | |-- Mage_Checkout_Block_Multishipping_Link
| | | |-- Mage_Checkout_Block_Multishipping_State
```

APPENDIX A. MAGENTO BLOCK HIERARCHY

```
| | | -- Mage_Checkout_Block_Onepage_Abstract
| | | | -- Mage_Checkout_Block_Onepage
| | | | | -- Mage_Checkout_Block_Onepage_Billing
| | | | | -- Mage_Checkout_Block_Onepage_Login
| | | | | -- Mage_Checkout_Block_Onepage_Payment
| | | | | -- Mage_Checkout_Block_Onepage_Progress
| | | | | -- Mage_Checkout_Block_Onepage_Review
| | | | | -- Mage_Checkout_Block_Onepage_Shipping
| | | | | -- Mage_Checkout_Block_Onepage_Shipping_Method
| | | | | -- Mage_Checkout_Block_Onepage_Shipping_Method_Additional
| | | | | -- Mage_Checkout_Block_Onepage_Shipping_Method_Available
| | | | -- Mage_Checkout_Block_Onepage_Abstract
| | | | -- Mage_Checkout_Block_Onepage_Failure
| | | | -- Mage_Checkout_Block_Onepage_Link
| | | | -- Mage_Checkout_Block_Onepage_Success
| | | | | -- Mage_Downloadable_Block_Checkout_Success
| | | | -- Mage_Checkout_Block_Onepage_Success
| | | | -- Mage_Checkout_Block_Success
| | | | -- Mage_Cms_Block_Widget_Block
| | | | -- Mage_Core_Block_Html_Calendar
| | | | -- Mage_Core_Block_Html_Date
| | | | | -- Mage_Adminhtml_Block_Html_Date
| | | | -- Mage_Core_Block_Html_Date
| | | | -- Mage_Core_Block_Html_Link
| | | | | -- Mage_Catalog_Block_Widget_Link
| | | | | | -- Mage_Catalog_Block_Category_Widget_Link
| | | | | | -- Mage_Catalog_Block_Product_Widget_Link
| | | | | -- Mage_Catalog_Block_Widget_Link
| | | | | -- Mage_Cms_Block_Widget_Page_Link
| | | | -- Mage_Core_Block_Html_Link
| | | | -- Mage_Core_Block_Messages
| | | | | -- Mage_Adminhtml_Block_Messages
| | | | | | -- Mage_Adminhtml_Block_Sales_Order_Create_Messages
| | | | | | -- Mage_Adminhtml_Block_Sales_Order_View_Messages
| | | | | -- Mage_Adminhtml_Block_Messages
| | | | -- Mage_Core_Block_Messages
| | | | -- Mage_Core_Block_Store_Switcher
| | | | -- Mage_Core_Block_Template_Facade
| | | | -- Mage_Core_Block_Template_Smarty
| | | | -- Mage_Core_Block_Template_Zend
| | | | -- Mage_Customer_Block_Account
| | | | -- Mage_Customer_Block_Account_Dashboard
| | | | | -- Mage_Customer_Block_Form_Edit
| | | | | -- Mage_Customer_Block_Newsletter
| | | | | -- Mage_Review_Block_Customer_List
| | | | | -- Mage_Tag_Block_Customer_Tags
| | | | -- Mage_Customer_Block_Account_Dashboard
| | | | -- Mage_Customer_Block_Account_Dashboard_Address
| | | | -- Mage_Customer_Block_Account_Dashboard_Block
| | | | -- Mage_Customer_Block_Account_Dashboard_Hello
| | | | -- Mage_Customer_Block_Account_Dashboard_Info
| | | | -- Mage_Customer_Block_Account_Dashboard_Newsletter
| | | | -- Mage_Customer_Block_Account_Dashboard_Sidebar
| | | | -- Mage_Customer_Block_Account_Forgotpassword
| | | | -- Mage_Customer_Block_Account_Navigation
| | | | -- Mage_Customer_Block_Address_Book
| | | | -- Mage_Customer_Block_Form_Login
```

APPENDIX A. MAGENTO BLOCK HIERARCHY

```
| | | -- Mage_Customer_Block_Widget_Abstract
| | | | -- Mage_Customer_Block_Widget_Dob
| | | | -- Mage_Customer_Block_Widget_Gender
| | | | -- Mage_Customer_Block_Widget_Name
| | | | -- Mage_Customer_Block_Widget_Taxvat
| | | -- Mage_Customer_Block_Widget_Abstract
| | | -- Mage_Directory_Block_Currency
| | | -- Mage_Directory_Block_Data
| | | | -- Mage_Customer_Block_Address_Edit
| | | | -- Mage_Customer_Block_Form_Register
| | | -- Mage_Directory_Block_Data
| | | -- Mage_Downloadable_Block_Customer_Products_List
| | | -- Mage_GiftMessage_Block_Message_Form
| | | -- Mage_GiftMessage_Block_Message_Helper
| | | -- Mage_GiftMessage_Block_Message_Inline
| | | -- Mage_GoogleCheckout_Block_Link
| | | -- Mage_GoogleOptimizer_Block_Code
| | | | -- Mage_GoogleOptimizer_Block_Code_Category
| | | | -- Mage_GoogleOptimizer_Block_Code_Conversion
| | | | -- Mage_GoogleOptimizer_Block_Code_Page
| | | | -- Mage_GoogleOptimizer_Block_Code_Product
| | | -- Mage_GoogleOptimizer_Block_Code
| | | -- Mage_Install_Block_Abstract
| | | | -- Mage_Install_Block_Admin
| | | | -- Mage_Install_Block_Begin
| | | | -- Mage_Install_Block_Config
| | | | -- Mage_Install_Block_Download
| | | | -- Mage_Install_Block_End
| | | | -- Mage_Install_Block_Locale
| | | -- Mage_Install_Block_Abstract
| | | -- Mage_Install_Block_State
| | | -- Mage_Newsletter_Block_Subscribe
| | | -- Mage_Page_Block_Html
| | | -- Mage_Page_Block_Html_Breadcrumbs
| | | -- Mage_Page_Block_Html_Footer
| | | -- Mage_Page_Block_Html_Head
| | | | -- Mage_Adminhtml_Block_Page_Head
| | | -- Mage_Page_Block_Html_Head
| | | -- Mage_Page_Block_Html_Header
| | | -- Mage_Page_Block_Html_Notices
| | | -- Mage_Page_Block_Html_Pager
| | | | -- Mage_Catalog_Block_Product_List_Toolbar_Pager
| | | | -- Mage_Catalog_Block_Seo_Sitemap_Tree_Pager
| | | -- Mage_Page_Block_Html_Pager
| | | -- Mage_Page_Block_Html_Toplinks
| | | -- Mage_Page_Block_Html_Welcome
| | | -- Mage_Page_Block_Js_Cookie
| | | -- Mage_Page_Block_Js_Translate
| | | -- Mage_Page_Block_Redirect
| | | | -- Mage_GoogleCheckout_Block_Redirect
| | | -- Mage_Page_Block_Redirect
| | | -- Mage_Page_Block_Switch
| | | -- Mage_Page_Block_Template_Container
| | | -- Mage_Page_Block_Template_Links
| | | -- Mage_Page_Block_Template_Links_Block
| | | | -- Mage_Wishlist_Block_Links
| | | -- Mage_Page_Block_Template_Links_Block
```


APPENDIX A. MAGENTO BLOCK HIERARCHY

```
| | | |-- Mage_Rss_Block_Catalog_NotifyStock
| | | |-- Mage_Rss_Block_Catalog_Review
| | | |-- Mage_Rss_Block_Catalog_Salesrule
| | | '-- Mage_Rss_Block_Catalog_Special
| | |-- Mage_Rss_Block_Abstract
| | |-- Mage_Rss_Block_List
| | |-- Mage_Rss_Block_Order_Details
| | |-- Mage_Rss_Block_Order_New
| | |-- Mage_Rss_Block_Order_Status
| | |-- Mage_Sales_Block_Billing_Agreement_View
| | |-- Mage_Sales_Block_Billing_Agreements
| | |-- Mage_Sales_Block_Items_Abstract
| | | |-- Mage_Checkout_Block_Multishipping_Addresses
| | | |-- Mage_Checkout_Block_Multishipping_Billing_Items
| | | |-- Mage_Checkout_Block_Multishipping_Overview
| | | |-- Mage_Checkout_Block_Multishipping_Shipping
| | | |-- Mage_Checkout_Block_Onepage_Review_Info
| | | |-- Mage_Sales_Block_Order_Creditmemo_Items
| | | '-- Mage_Sales_Block_Order_Creditmemo
| | | |-- Mage_Sales_Block_Order_Creditmemo_Items
| | | |-- Mage_Sales_Block_Order_Email_Creditmemo_Items
| | | |-- Mage_Sales_Block_Order_Email_Invoice_Items
| | | |-- Mage_Sales_Block_Order_Email_Items
| | | |-- Mage_Sales_Block_Order_Email_Shipment_Items
| | | |-- Mage_Sales_Block_Order_Invoice_Items
| | | '-- Mage_Sales_Block_Order_Invoice
| | | |-- Mage_Sales_Block_Order_Invoice_Items
| | | |-- Mage_Sales_Block_Order_Items
| | | |-- Mage_Sales_Block_Order_Print
| | | |-- Mage_Sales_Block_Order_Print_Creditmemo
| | | |-- Mage_Sales_Block_Order_Print_Invoice
| | | |-- Mage_Sales_Block_Order_Print_Shipment
| | | '-- Mage_Sales_Block_Order_Shipment_Items
| | |-- Mage_Sales_Block_Items_Abstract
| | |-- Mage_Sales_Block_Order_Comments
| | |-- Mage_Sales_Block_Order_Details
| | |-- Mage_Sales_Block_Order_Email_Items_Default
| | | '-- Mage_Downloadable_Block_Sales_Order_Email_Items_Downloadable
| | |-- Mage_Sales_Block_Order_Email_Items_Default
| | |-- Mage_Sales_Block_Order_Email_Items_Order_Default
| | | |-- Mage_Downloadable_Block_Sales_Order_Email_Items_Order_Downloadable
| | | '-- Mage_Sales_Block_Order_Email_Items_Order_Grouped
| | |-- Mage_Sales_Block_Order_Email_Items_Order_Default
| | |-- Mage_Sales_Block_Order_History
| | |-- Mage_Sales_Block_Order_Info
| | |-- Mage_Sales_Block_Order_Item_Renderer_Default
| | | |-- Mage_Bundle_Block_Sales_Order_Items_Renderer
| | | |-- Mage_Downloadable_Block_Sales_Order_Item_Renderer_Downloadable
| | | '-- Mage_Sales_Block_Order_Item_Renderer_Grouped
| | |-- Mage_Sales_Block_Order_Item_Renderer_Default
| | |-- Mage_Sales_Block_Order_Recent
| | |-- Mage_Sales_Block_Order_Shipment
| | |-- Mage_Sales_Block_Order_Tax
| | |-- Mage_Sales_Block_Order_Totals
| | | |-- Mage_Adminhtml_Block_Sales_Totals
| | | |-- Mage_Adminhtml_Block_Sales_Order_Creditmemo_Totals
| | | |-- Mage_Adminhtml_Block_Sales_Order_Invoice_Totals
```

APPENDIX A. MAGENTO BLOCK HIERARCHY

```
| | | | | |-- Mage_Adminhtml_Block_Sales_Order_Totals
| | | | | | '-- Mage_Adminhtml_Block_Sales_Order_Totals_Item
| | | | | '-- Mage_Adminhtml_Block_Sales_Order_Totals
| | | | |-- Mage_Adminhtml_Block_Sales_Totals
| | | | |-- Mage_Sales_Block_Order_Creditmemo_Totals
| | | | '-- Mage_Sales_Block_Order_Invoice_Totals
| | | |-- Mage_Sales_Block_Order_Totals
| | | |-- Mage_Sales_Block_Order_View
| | | |-- Mage_Sales_Block_Recurring_Profile_View
| | | |-- Mage_Sales_Block_Recurring_Profiles
| | | |-- Mage_Sales_Block_Reorder_Sidebar
| | | |-- Mage_Sendfriend_Block_Send
| | | |-- Mage_Shipping_Block_Tracking_Ajax
| | | |-- Mage_Shipping_Block_Tracking_Popup
| | | |-- Mage_Tag_Block_All
| | | |-- Mage_Tag_Block_Customer_Edit
| | | |-- Mage_Tag_Block_Customer_Recent
| | | |-- Mage_Tag_Block_Popular
| | | |-- Mage_Tag_Block_Product_List
| | | |-- Mage_Tax_Block_Sales_Order_Tax
| | | | '-- Mage_Adminhtml_Block_Sales_Order_Totals_Tax
| | | |-- Mage_Tax_Block_Sales_Order_Tax
| | | |-- Mage_Wishlist_Block_Customer_Sharing
| | | '-- Mage_Wishlist_Block_Share_Email_Rss
|-- Mage_Core_Block_Template
|-- Mage_Core_Block_Text
| | |-- Mage_Core_Block_Text_List
| | | |-- Mage_Adminhtml_Block_Text_List
| | | | |-- Mage_Adminhtml_Block_Poll_Edit_Tab_Answers
| | | | |-- Mage_Adminhtml_Block_Sales_Order_Creditmemo_View_Comments
| | | | |-- Mage_Adminhtml_Block_Sales_Order_Invoice_View_Comments
| | | | '-- Mage_Adminhtml_Block_Sales_Order_Shipment_View_Comments
| | | '-- Mage_Adminhtml_Block_Text_List
| | |-- Mage_Core_Block_Text_List
| | |-- Mage_Core_Block_Text_List_Item
| | |-- Mage_Core_Block_Text_List_Link
| | |-- Mage_Core_Block_Text_Tag
| | | |-- Mage_Core_Block_Text_Tag_Css
| | | | '-- Mage_Core_Block_Text_Tag_Css_Admin
| | | |-- Mage_Core_Block_Text_Tag_Css
| | | |-- Mage_Core_Block_Text_Tag_Debug
| | | '-- Mage_Core_Block_Text_Tag_Js
| | |-- Mage_Core_Block_Text_Tag
| | |-- Mage_Core_Block_Text_Tag_Meta
| | '-- Mage_GoogleAnalytics_Block_Ga
|-- Mage_Core_Block_Text
|-- Mage_Customer_Block_Address_Renderer_Default
|-- Mage_Page_Block_Html_Wrapper
|-- Mage_Paypal_Block_Standard_Redirect
|-- Mage_Rule_Block_Editable
|-- Mage_Rule_Block_Newchild
| '-- Mage_Rule_Block_Rule
'-- Mage_Core_Block_Abstract
```

Visit <http://www.pulsestorm.net/nofrills-layout-appendix-a> to join the discussion online.

Appendix B

Class Aliases

Magento uses a factory pattern for instantiating certain objects. Don't let the design pattern name scare you though, it's not that complicated.

In raw PHP, if you wanted to instantiate an object from a class, you'd say something like

```
$customer = new Product();
```

There's nothing in Magento stopping you from doing this. However, most of the Magento core code and its various sub-systems do things a little differently.

In Magento, when you want to instantiate an object from a class, you use code like this

```
$customer = Mage::getModel('catalog/product');
```

This is calling a static method on the `Mage` class named `getModel`. This method will examine Magento's configuration, and ask

What model class does the string `catalog/product` associate with.

Magento will answer back `"Mage_Catalog_Model_Product"`, and then a `"Mage_Catalog_Model_Product"` will be instantiated. This `catalog/product` string is known as the class alias.

Magento uses this instantiation method for

1. Block classes: `$layout->createBlock('foo/bar')`
2. Helper classes: `Mage::helper('foo/bar')`
3. Model classes: `Mage::getModel('foo/bar'), Mage::getModel('foo/bar')`

The `createBlock`, `helper`, and `getModel` methods are all factories. They make objects of a particular type.

B.1 Why so Complicated?

This may seem like a lot of misdirection for something as simple as a class declaration, but that misdirection brings some benefits along for the ride. It helps create a type system around classes, Magento itself knows what classes have or have not been declared at any one time, the shorthand saves some verbosity in typing, and it helps enable one of Magento's unique PHP feature, class rewrites (similar to duck-typing or monkey-patching in the Ruby and Python communities)

B.2 What Class?

This is all well and good, but can sometimes leave you wondering what class alias corresponds to what class definition. The easiest thing to do is use the free, online demo of Commerce Bug

`http://commercebugdemo.pulsestorm.net/`

The class URI lookup tab will let you lookup which class aliases correspond to which PHP classes for a core system.

The way Magento actually looks up class definitions is via its configuration system. All the `config.xml` files in a Magento install are merged into one, large, global config. This giant tree contains a top level `<global/>` node that looks something like this

```
<config>
  <global>
    <models>...</models>
    <helpers>...</helpers>
    <blocks>...</blocks>
  </global>
</config>
```

The first thing Magento does when you use a class alias to instantiate a class is determine the context (model, helper, block), and then look in an appropriate node (`<models>`, `<helpers>`, and `<blocks>`).

Next, each of the `<models>`, `<helpers>`, and `<blocks>` contains a number of "group" nodes

```
<models>
  <catalog>...</catalog>
  <core>...</core>
  <page>...</page>
</models>
```

If you look at a class alias

`catalog/product`

The portion to the left of the / is the group name. Magento will use this to determine which of the group nodes it should look in next.

Finally, each group node contains, at minimum, a class node `<class>`

```
<models>
  <catalog>
    <class>Mage_Catalog_Model</class>
  </catalog>
</model>
```

This node contains the **base** PHP class name for the model (or helper, or block) group. This base name in place, the non-group portion of the class alias is appended to the base class name, with the first letter of each underscored word uppercased

```
catalog/product
Mage_Catalog_Model_Product

catalog/product_review
Mage_Catalog_Model_Product_Review
```

That's how Magento resolves which PHP class to use for a class alias.

B.3 Class Rewrites

There's one additional node in the config that Magento will check while looking up a class name. End users of the system (that means you) may provide a `<rewrite/>` node that will tell Magento to replace one class with another. This is Magento's famous class rewrite system. Using the following

```
<models>
  <catalog>
    <rewrite>
      <product_review>Yourpackage_Yourmodule_Model_Someclass</product_review>
    </rewrite>
  </catalog>
</model>
```

would tell Magento that whenever a `catalog/product_review` is instantiated, it should use a `Yourpackage_Yourmodule_Model_Someclass`.

Visit <http://www.pulsestorm.net/nofrills-layout-appendix-b> to join the discussion online.

Appendix C

Creating Code Modules

The word **module** has come to be one of the most abused in software development. If a designer's adding a table to a side bar, they call it a module. If a developer is adding a class to a system, they call it a module. If the project manager wants to sound tech savvy, they call everything a module.

The word **module** has a very specific meaning in Magento. It refers to a particular organization of code, such that it may be loaded into an existing Magento system in a defined way, with the loading requiring no knowledge of what other Magento modules are doing. In layman's terms, everyone keeps their code separate, and Magento is smart enough to know where to look for it.

If you look in

```
app/code/core/Mage
```

you'll see around 50 - 60 different folders. Each of these is a single module. A module may contain controllers, models, helpers, blocks, SQL Install files, extra configuration files for changing Magento system behavior, new classes for Extending the SOAP and RPC APIs, the list goes on and on. Rather than have a single folder with, say, 200 controller files, Magento uses code modules to organize them by functionality.

When you want to add code to Magento, either to change existing functionality or add new functionality, you'll also add a new module. However, your module will go in

```
app/code/local/*
```

instead of `app/code/core`. This is part of Magento's Code Pool feature, which is separate from the module feature. The `local` code pool is where you can put your own code, such that it won't be overridden by Magento's system updates. Magento also has a

```
app/code/community/*
```

code pool, which is meant for installation of modules from third-parties.

C.1 Adding a Module

The first step to creating a module is picking a *Package Name* or *Namespace*. If Bill Gates was making a Magento module, he might pick the name *Microsoft*. Once you've selected your name, create a folder in local

```
mkdir app/code/local/Packagename
```

This package name can contain multiple code modules. Consider Magento Inc. They use the package name **mage** (short for Magento). While not necessary, the general consensus is that the package name should contain only alphanumeric characters, and be single word cased. This helps avoid autoload problems when developing on case insensitive file systems (Windows, OS X sort of) that deploy to case sensitive systems (Linux). The **Packagename** **will** be used as part of PHP class names, so it also must meet those naming conventions as well.

Next, pick a name for your module. Strive for something simple that describes what the module is for. **Important:** There's many tutorials that recommend you use names that are the same as Magento's module names if you're rewriting or changing the functionality of a core Magento class. While there's nothing stopping you from doing this, it's not required and can actually cause mass confusion to developers when they're new to the system.

When you've picked a name, create a folder inside your package name folder

```
mkdir app/code/local/Packagename/Modulename
```

Finally, every module in Magento needs one more file, a configuration file. This file will contain information about the module's features, and will be merged into Magento's main config, along with all the other modules. Create the following folder

```
mkdir app/code/local/Packagename/Modulename/etc
```

and then create the following file

```
<!-- #File: app/code/local/Packagename/Modulename/etc/config.xml -->
<config>
    <modules>
        <Packagename_Modulename>
            <version>1.0.0</version>
        </Packagename_Modulename>
    </modules>
</config>
```


The `<Packagename_ModuleName/>` node should be named using the package name and module name you chose. This unique string will be used to identify your modules. It will also, (and should also), be used as the base name for any classes in your module

```
class Packagename_ModuleName_IndexController {}
class Packagename_ModuleName_Block_Myblock {}
```

C.2 Enabling your Module

There's one last step you'll need to take if you want to let Magento know about your module. If you browse to

```
app/etc/modules/
```

you'll see a number of XML files. Think of the `etc` folder in Magento the same way you would on a *nix system. It contains configuration files for your store's core systems. These XML files tell Magento that you'd like to "turn on" a particular module. Create an XML file using the unique `Packagename_ModuleName` string with the following contents.

```
<!-- #File: app/etc/modules/Packagename_ModuleName.xml -->
<?xml version="1.0" encoding="UTF-8"?>
<config>
    <modules>
        <Packagename_ModuleName>
            <active>true</active>
            <codePool>local</codePool>
        </Packagename_ModuleName>
    </modules>
</config>
```

Again, the `<Packagename_ModuleName/>` node should use the unique string that identifies your own module. The `<active>true</active>` node determines if Magento loads this particular module's `config.xml` into the system. You can use this to completely shut off a module (although, if other module's attempt to use that module's functionality, object instantiations will fail). The `<codePool>local</codePool>` node lets the system know where it can find your module files. The three valid values are `core`, `community`, and `local`

```
app/code/core
app/code/community
app/code/local
```

With all of the above in place, clear your cache and load up the Admin Console. Head over to

```
System -> Configuration -> Advanced -> Disable Module's Output
```

This configuration panel is one of the few areas of Magento where you can see a list of all the installed modules. If you followed the above steps correctly, you should see your module listed. Congratulations, you've added a module to the system!

C.3 Next Steps

Of course, a module is useless without additional code. Going into everything you can do with a module would be a book in and of itself. However, the general pattern is, before you can add a type of class to your module (model, helper, etc.) you need to add some code to your `config.xml`. This lets the system know that "hey, this module has feature X and use these classes". This is what makes Magento a **configuration** based MVC system, rather than a convention based one.

Visit <http://www.pulsestorm.net/nofrills-layout-appendix-c> to join the discussion online.

Appendix D

Block Action Reference

The Block Action reference ended up being ludicrously huge. See the `d_block_action_reference.html` file, distributed with this book.

Visit <http://www.pulsetorm.net/nofrills-layout-appendix-d> to join the discussion online.

Appendix E

Theme and Layout Resolution

The default Magento design package ships with a default Magento theme.

```
#Location of default theme  
app/design/frontend/default/default/*
```

You could (and can) change the location of this default theme in the Admin Console

```
System -> Configuration -> Design -> Themes -> Default
```

but even with a new folder name this theme is still your **default** theme. It's meant to contain the main design of your store.

If you navigate to

```
System -> Design
```

Magento will allow you select a **custom** theme for a particular date range. The most obvious use case of this is the holiday special. Put Santa on your store, watch those sales soar.

E.1 Template Resolution

Prior to Magento 1.4.1, when Magento went looking for a block's template file, it would check if a custom design was set, and

1. If no custom design was set, the default theme folder was used
2. If a custom design **was** set, Magento was first look for a template there. If it didn't find one, it would fallback to the default theme

This system worked, but had a small problem. Many stores would never change the default theme. They'd leave it

`default/default`

and just modify those files to skin their store. When they upgraded their system, the `default/default` theme would also be updated, wiping out their changes. This was viewed as a untenable state of affairs, and the concept of the `base` design package and theme was created.

E.2 The Base Package

The Magento theming system still operates as described above. However, if a template isn't found in the default theme rather than render nothing Magento will look in one final place for the template file in **the base package's default theme**

`app/design/frontend/base/default/*`

This is the **final** fallback. Currently, the default theme that ships with Magento is mostly implemented in the base design package. This allows designers and developers to **selectively** update `phtml` in the **default** theme if they want to change the behavior of something. The intent is that you **never** edit the base design package. If you want to change a particular template you add it to your default theme.

E.3 Layout Files

Layout files follow the same cascading loading rules. First the current custom design is checked, then the default theme, and the default theme in the base design package. Layout files were included in this grand design of the base theme system, as many developers ignore, or are unaware of, `local.xml` for layout updates. This led many system owners to edit the default Package Layout files to achieve their final design goals, resulting in the same upgrade problems mentioned above.

Visit <http://www.pulsestorm.net/nofrills-layout-appendix-e> to join the discussion online.

Appendix F

The Hows and Whys of Clearing Magento's Cache

In layman's terms, caching in computer science/software engineering refers to the practice of doing something that's resource intensive once, storing the results somewhere else, and then the next time someone wants it you hand them the stored result. The most common example of this for web developers is browser caching. Asset files (images, CSS, Javascript) will be downloaded once, and then stored locally for a period of time. This results in better network performance, and hair pulling by web developers at 2am wondering why their CSS files aren't being updated.

Magento, like most modern web frameworks, heavily utilizes a caching system to improve performance. For example, certain configuration files are loaded once from disk once, combined, and then the combined config is stored on disk for later use. This includes the Layout XML files. This means if you have caching turned on (CE ships with caching on out of the box), you'll need to clear your cache after making any changes to the layout files. Otherwise, the old, cached version of the Package Layout will be loaded and Magento won't see your changes, and you'll be left wondering why your new block isn't showing up.

You can control cache settings by navigating to

`System -> Cache Managment`

in the Admin Console, (see *Figure F.1*)

From this page you can clear out the cache, or turn it off entirely.

Occasionally, a cached configuration may prevent you from getting to the Magento Admin Console. For example, consider an event observer with an invalid class name. If this happens, you'll want to look in the `var/cache` folder

APPENDIX F. THE HOWS AND WHYS OF CLEARING MAGENTO'S CACHE

Cache Storage Management

Select All | Unselect All | Select Visible | Unselect Visible | 0 items selected

Cache Type	Description	Associated
<input type="checkbox"/> Configuration	System(config.xml, local.xml) and modules configuration files(config.xml).	CONFIG
<input type="checkbox"/> Layouts	Layout building instructions.	LAYOUT_C
<input type="checkbox"/> Blocks HTML output	Page blocks HTML.	BLOCK_H'
<input type="checkbox"/> Translations	Translation files.	TRANSLA'
<input type="checkbox"/> Collections Data	Collection data files.	COLLECTI
<input type="checkbox"/> EAV types and attributes	Entity types declaration cache.	EAV
<input type="checkbox"/> Web Services Configuration	Web Services definition files (api.xml).	CONFIG_A

Additional Cache Management

Flush Catalog Images Cache | Pregenerated product images files.

Flush JavaScript/CSS Cache | Themes JavaScript and CSS files combined to one file.

Figure F.1

```
ls -l var/cache
total 0
drwxrwxrwx 10 _www staff 340 Mar 15 16:10 mage--0
drwxrwxrwx 10 _www staff 340 Mar 15 16:10 mage--1
drwxrwxrwx 6 _www staff 204 Mar 15 16:02 mage--2
drwxrwxrwx 8 _www staff 272 Mar 15 16:02 mage--3
drwxrwxrwx 8 _www staff 272 Mar 15 16:10 mage--4
drwxrwxrwx 12 _www staff 408 Mar 15 16:02 mage--5
drwxrwxrwx 10 _www staff 340 Mar 15 16:02 mage--6
drwxrwxrwx 8 _www staff 272 Mar 15 16:02 mage--7
drwxrwxrwx 12 _www staff 408 Mar 15 16:02 mage--8
drwxrwxrwx 10 _www staff 340 Mar 15 16:02 mage--9
drwxrwxrwx 6 _www staff 204 Mar 15 16:02 mage--a
drwxrwxrwx 10 _www staff 340 Mar 15 16:02 mage--b
drwxrwxrwx 164 _www staff 5576 Mar 15 16:02 mage--c
drwxrwxrwx 172 _www staff 5848 Mar 15 16:02 mage--d
drwxrwxrwx 4 _www staff 136 Mar 15 16:10 mage--e
drwxrwxrwx 12 _www staff 408 Mar 15 16:10 mage--f
```

This folder is where Magento stores its cached data. Delete everything in this folder to manually clear the Magento cache and restore you store's functionality.

Visit <http://www.pulsetorm.net/nofrills-layout-appendix-f> to join the discussion online.

Appendix G

Magento Setters and Getters

In most MVC model systems a common pattern develops. Developers find they need to store two general types of data

1. "Business Logic" data (ex. a product's SKU)
2. Data that allows the model to function (ex. the database table name)

It's very common to see developers use a single `array` (or similar, hash table like structure) property to store the business logic data, allowing the other class/object properties to be used for system functionality. Magento is no different. Any object that comes from a class that inherits from `Varien_Object` (which includes both models and blocks) has a protected `$_data` property

```
/**
 * Object attributes
 *
 * @var array
 */
protected $_data = array();
```

This array holds all the object's business logic data. You can get an array of key/value pairs for this data with the `getData` method.

```
var_dump($object->getData());
```

If you want to **set** a specific data field, use

```
$object->setData('the_key', 'value');
```

similarly, if you want a specific field back from an object, you can use

```
$value = $object->getData('the_key');
```


and you can set multiple keys at once by using `setData` with an array.

```
$value = $object->setData(array(
    'the_key'=>'value'
    'the_thing'=>$thing,
));
```

You've probably noticed we're naming our keys using an all lower-case, underscore-for-spaces convention. While nothing enforces this, it *is* the standard Magento convention. Beyond consistency, this also helps when it comes to Magento's magic getter and setter methods.

G.1 Getter and Setter

In addition to the data getting and setting methods mentioned above, there's also a more "magic" syntax.

```
$key = $object->getTheKey();
$object->setTheKey('value');
```

Using PHP's `__call` method, Magento has implemented their own `get` and `set` methods. If you call a method on an object (with `Varien_Object` in the inheritance chain) whose name begins with `get` or `set`, **and** there isn't an existing method already with the same name, Magento will use the remainder of the method name to create a data property key, and either get or set the value. That means this

```
$object->setTheKey('value');
```

is equivalent to this

```
$object->setData('the_key','value');
```

That's why it's important to keep with the lowercase/underscore key convention. Magento will convert the leading-camel-case

`TheKey`

into a key named

`the_key`

Another neat feature here is that the `set` method will always return an instance of the object being set, which enables method chaining

```
$object->setFoo('bar')->setBaz('hola')->save();
```

After using this style interface for a few weeks you'll be loath to return to typing out array brackets.

G.2 Other Magic Methods

Magento also has magic methods for unsetting, and checking for the existence of a property

```
$this->unTheKey();  
$this->hasTheKey();
```

Checkout the source of `Varien_Object` for more information

```
public function __call($method, $args)  
{  
    switch (substr($method, 0, 3)) {  
        case 'get' :  
            //Varien_Profiler::start('GETTER: ' . get_class($this) . '::' . $method);  
            $key = $this->_underscore(substr($method,3));  
            $data = $this->getData($key, isset($args[0]) ? $args[0] : null);  
            //Varien_Profiler::stop('GETTER: ' . get_class($this) . '::' . $method);  
            return $data;  
  
        case 'set' :  
            //Varien_Profiler::start('SETTER: ' . get_class($this) . '::' . $method);  
            $key = $this->_underscore(substr($method,3));  
            $result = $this->setData($key, isset($args[0]) ? $args[0] : null);  
            //Varien_Profiler::stop('SETTER: ' . get_class($this) . '::' . $method);  
            return $result;  
  
        case 'uns' :  
            //Varien_Profiler::start('UNS: ' . get_class($this) . '::' . $method);  
            $key = $this->_underscore(substr($method,3));  
            $result = $this->unsetData($key);  
            //Varien_Profiler::stop('UNS: ' . get_class($this) . '::' . $method);  
            return $result;  
  
        case 'has' :  
            //Varien_Profiler::start('HAS: ' . get_class($this) . '::' . $method);  
            $key = $this->_underscore(substr($method,3));  
            //Varien_Profiler::stop('HAS: ' . get_class($this) . '::' . $method);  
            return isset($this->_data[$key]);  
    }  
    throw new Varien_Exception("Invalid method " . get_class($this) . "::" .  
        $method . "(" . print_r($args,1) . ")");  
}
```

Visit <http://www.pulsetorm.net/nofrills-layout-appendix-g> to join the discussion online.

Appendix H

Widget Field Rendering Options

The simplest configuration for a Magento widget data parameter is

```
<parameters>
  <our_parameter>
    <visible>0</visible>
    <required>1</required>
    <value>foobazbar</value>
    <type>text</type>
  </our_parameter>
</parameters>
```

This creates a hidden field (`<visible>0</visible>`) that will always be populated with the value 'foobazbar' (`<value>foobazbar</value>`).

While hidden fields are useful for widgets entered via a CMS content area, it's far more common for parameters to have a visible user interface element that allows end-system-users to enter data. That is to say, a visible text field is more common

```
<parameters>
  <our_parameter>
    <visible>1</visible>
    <required>1</required>
    <label>Label for our Parameter</label>
    <type>text</type>
    <value>bazbarfoo</value>
    <sort_order>10</sort_order>
  </our_parameter>
</parameters>
```

We've changed the `<visible/>` tag so it contains the value "1" (boolean for true). We've also added a `<label>` tag which will be used as the text label

APPENDIX H. WIDGET FIELD RENDERING OPTIONS

which describes the field, and a `<sort_order>` field which control where (above or below) a particular UI element will show up compared to others. The `<value>bazbarfoo</value>` tag will set the **default** value for the UI element.

Widget Options	
Label for our Parameter*	<input type="text" value="bazbarfoo"/>

Figure H.1

You can also augment your fields with some instructional text by using the `<description/>` node.

```
<parameters>
  <our_parameter>
    <visible>1</visible>
    <required>1</required>
    <label>Label for our Parameter</label>
    <type>text</type>
    <value>bazbarfoo</value>
    <sort_order>10</sort_order>
    <description>
      This is the field where we put the thing
    </description>
  </our_parameter>
</parameters>
```

Sometimes a free form text field gives users too much control over what values they enter in a widget. For cases where we want to restrict a user's choices, we can use a `select` or `multiselect`.

```
<parameters>
  <our_parameter>
    <visible>1</visible>
    <required>1</required>
    <label>Should I Stay or Should I Go?</label>
    <type>select</type>
    <value>stay</value>
    <values>
      <staying>
        <value>stay</value>
        <label>There will be Trouble</label>
      </staying>
      <going>
        <value>go</value>
        <label>There will be Double</label>
      </going>
    </values>
    <sort_order>10</sort_order>
  </our_parameter>
</parameters>
```

The important changes here are we've changed our `<type/>` tag to `select`, and added a new `<values/>` node. The sub-nodes of the `<values/>` node will be used to create the label/value pairs for the HTML `<select>` elements generated for the front end. Alternatly, you can provide the name of a source model.

```
<parameters>
  <our_parameter>
    <visible>1</visible>
    <required>1</required>
    <label>Should I Stay or Should I Go?</label>
    <type>select</type>
    <value>stay</value>
    <source_model>adminhtml/system_config_source_yesno</source_model>
    <sort_order>10</sort_order>
  </our_parameter>
</parameters>
```

The string `adminhtml/system_config_source_yesno` is a class alias for a Magento model (in this case `Mage_Adminhtml_Model_System_Config_Source_Yesno`). Source models are special model classes with a `toOptionArray` method.

```
class Mage_Adminhtml_Model_System_Config_Source_Yesno
{
    /**
     * Options getter
     *
     * @return array
     */
    public function toOptionArray()
    {
        return array(
            array('value' => 1, 'label'=>Mage::helper('adminhtml')->__('Yes')),
            array('value' => 0, 'label'=>Mage::helper('adminhtml')->__('No')),
        );
    }
}
```

This method returns a set of key/value pairs for your select. You may create your own source models, or use one the models that ships with Magento. Checkout the PHP files in

```
app/code/core/Mage/Adminhtml/Model/System/Config/Source
```

for a list of the source models that ship with Magento.

H.1 Creating Your Own Form Elements

Sometimes you're going to want a form element that's more interactive than a single text or a select. The widget system has a mechanism that allows you to build your own form elements for the widget UI.

APPENDIX H. WIDGET FIELD RENDERING OPTIONS

First, your parameter configuration should look like the following

```
<our_parameter>
    <visible>1</visible>
    <required>1</required>
    <label>Should I Stay or Should I Go?</label>
    <type>label</type>
    <helper_block>
        <type>yourpackage_yourmodule/widgettest</type>
    </helper_block>
    <sort_order>10</sort_order>
    <description>This is the field where you'll put the synergy.</description>
</our_parameter>
```

The key nodes here are `<type>` and `<helper_block>`. Our field type here is `label`. Normally, a field type of `label` will render the label for a field without any form element. In other words, a form element with no functional value, only instructional/branding/experience. However, we've also included a `<helper_block>` node. This node (via the `<type/>` sub-node) configures a block class that will render our form.

After configuring this parameter, you'll need to define your block class. Despite living in the standard block hierarchy, we **do not** want to implement our rendering in its `toHtml` method. Instead, this class needs a special method named `prepareElementHtml`

```
#File: app/code/local/Yourpackage/Yourmodule/Block/Widgettest.php
class Yourpackage_Yourmodule_Block_Widgettest extends Mage_Core_Block_Abstract
{
    /**
     * Overly simple example
     */
    public function prepareElementHtml(
        Varien_Data_Form_Element_Abstract $element)
    {
        $simple_input = '<input type="text" name="' .
            strip_tags($element->getName()) .
            '" value="' .
            strip_tags($element->getValue()) .
            '" />';

        $element->setData('after_element_html', $simple_input);

        $element->setValue(''); //blank out value
        return $element;
    }
}
```

During the rendering of your parameter's UI, Magento will call the `prepareElementHtml` method of your `helper_block`, passing in a `Varien_Data_Form_Element_Abstract` object. This `$element` is the object that Magento will use to render out your form element. To implement a custom form element, your job is

1. Grab the form element's name and value from `$element`. The value will contain previously saved values, and the name will be the correct name

for the HTML form element to ensure the form data is saved on post

2. Add HTML to the form element's rendering process to implement your custom element
3. Optionally, if you don't want the default value rendering to take place, clear the value from `$element` before returning it.

In our example above, we've created a ludicrously simple example to demonstrate how you might achieve this. We've

1. Created a HTML element (`$simple_input`) using string concatenation.
2. Added this HTML to the element with `$element->setData('after_element.html', $simple_input);`
3. Zeroed out the value of `$element`

H.2 Advanced Examples

As mentioned, our example above is ludicrously simple. However, with the power to create any arbitrary HTML, CSS or Javascript for your form, the possibilities are endless. Look to a few of Magento's `<block.helper>`s for inspiration

```
app/code/core/Mage/Adminhtml/Block/Catalog/Category/Widget/Chooser.php
app/code/core/Mage/Adminhtml/Block/Catalog/Product/Widget/Chooser.php
app/code/core/Mage/Adminhtml/Block/Cms/Block/Widget/Chooser.php
app/code/core/Mage/Adminhtml/Block/Cms/Page/Widget/Chooser.php
```

Visit <http://www.pulsetorm.net/nofrills-layout-appendix-h> to join the discussion online.

Appendix I

System Configuration Variables

A simple system configuration system might store a set of key/value pairs something like this.

```
$config['db_name']      = 'localhost';
$config['db_password']  = '12345';
$config['logo']         = 'awesomelogo.gif';
etc.
```

However, in keeping with its core philosophy of "When in doubt, use XML", Magento stores its system configuration values in *tree* format. The above might be represented something like this

```
<system_config>
  <store>
    <database>
      <name>localhost</name>
      <password>12345</password>
    </databases>
    <design>
      <logo>awesomelogo.gif</logo>
    </design>
  </store>
</system_config>
```

This is a common approach to modern configuration systems, as it allows you to develop a hierarchy of organized values as more and more sections of your system or application become configurable. What's really interesting is, using xpath-like expressions, you can still treat a node-based configuration system as a set of key/value pairs. That's exactly how you fetch a Magento System Configuration value

```
Mage::getStoreConfig('store/database/name');
```


APPENDIX I. SYSTEM CONFIGURATION VARIABLES

```
Mage::getStoreConfig('store/database/password');  
Mage::getStoreConfig('store/design/logo');
```

Magento allows each module to define new nodes for this configuration tree, as well as user interfaces for store owners to enter configuration values in the Admin Console under

System -> Configuration

This system is beyond the scope of this book, but there's plenty of information online. If you're interested, you can start reading here

<http://alansstorm.com/custom-magento-system-configuration>

Visit <http://www.pulsestorm.net/nofrills-layout-appendix-i> to join the discussion online.

Appendix J

Magento Connect

Magento Connect is a lot of things. First and foremost, it's Magento Inc's online repository for free, downloadable extensions. It's also a package management system that was, originally, based on the PHP PEAR packaging format. Magento Connect 2.0 was released along with Magento CE 1.5. This means there's two separate package file formats, which is why we've included two different sets of modules.

J.1 What is an Extension

As you may already know, Magento separates its "backend" code into formal code modules. A Magento Connect extension may **contain** modules, but a Magento Connect extension is not **just** a code module.

A Magento Connect extension is a packaged collection of files that Magento will install into your system. Each file in the package has a Magento Connect type, which will control where Magento installs the file. For example, a **local module file** knows to install itself in `./app/code/local`, whereas a **PHP Library file** installs itself in `./lib`. See

System ->Connect ->Package Extensions

for a full list of types.

There's one type in particular you'll want to be aware of. That's the **other** type. This type's base folder is Magento's base installation folder, which gives a Magento extension the ability to install a file **anywhere** in your system, and in turn you can create a package that includes files from anywhere.

J.2 Installing Extensions: The GUI Way

There's a GUI admin for Magento Connect. You can reach it from the Admin Console by navigating to

```
System -> Magento Connect -> Magento Connect Manager
```

You'll need to reauthorize your session as the admin user, (or any user with Magento Connect ACL rights). The code that bootstraps the Magento Connect Manager is separate from the source code of your Magento system proper.

Installing extensions that have been uploaded to Magento Inc's central server is as easy as entering the extension key into the installation field.

If you've downloaded a .tgz package file from the internet, Magento 1.5 also offers a handy upload form, allows you to directly upload an extension

J.3 Installing Extensions: The Command Line Way

Both the 1.4x and 1.5x branches of Magento offer the ability to install extension from the command line. However, the tools used for each version differ slightly.

J.3.1 Magento Connect CLI install for Magento 1.42

In the root folder of Magento 1.4.2 there's a shell script named `pear`. This shell script is **not** the standard PEAR installer. It's a customized installer you may use to install Magento Connect extensions. To use it, you'll need to tell your operating system its allowed to execute it as a program

```
chmod +x pear
```

After that, you'll need to run

```
./pear mage-setup
```

After setting a number of configuration variables and initializing two channels

```
connect.magentocommerce.com/core
connect.magentocommerce.com/community
```

the script will exit. You're now ready to install and uninstall packages using the command line installer

```
./pear install No_Frills_Magento_Layout_1_start-1.0.0.tgz
./pear uninstall \
channel://connect.magentocommerce.com/community/No_Frills_Magento_Layout_1_sta...
```

J.3.2 Magento Connect CLI install for Magento 1.5+

Magento 1.5 removed the `pear` installer, and introduced a new command line script (`mage`) that offers a similar function. Again, you'll need to give it executable permissions

```
chmod +x mage
```

and then initialize it with

```
./mage mage-setup
```

After the setup script finishes running, you'll be able to install extensions from a file. **NOTE:** The command has changed to `install-file`, and the arguments to `uninstall` have changed as well

```
./mage install-file No_Frills_Magento_Layout_3_start-1.0.0.tgz  
./mage uninstall community No_Frills_Magento_Layout_3_start
```

Visit <http://www.pulsetorm.net/nofrills-layout-appendix-j> to join the discussion online.