No Frills Magento Layout

Alan Storm

April 2011

Contents

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 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, of 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 instal, 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 frusting 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 books 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 ready more about Alan and his Magento products at the following URLs

http://alanstorm.com/

http://store.pulsestorm.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.pulsestorm.net/nofrills-layout-introduction to join the discussion online.

Chapter 1

Building Layouts Programmatically

1.1 Template Blocks

1.2 Template Files

1.3 Back to our Template

1.4 Nesting Blocks

1.5 Advanced Block Functionality

1.6 Block Methods

1.7 Enter the Layout

1.7.1 What's a Singleton!?

1.8 Back to the Code

1.9 Who's the Leader

1.10 Method Chaining

1.11 A Full Page Layout

1.12 Initializing the Layout and Setting Content

1.13 Insert vs. Set

1.14 Getting a Reference and Text List

1.15 A Recap and a Dilema

Chapter 2

XML Page Layout Files

2.1 Hello World in XML

2.2 An Interesting use of the Word Simple

2.3 Adding the XML, Generating the Blocks

2.4 Getting a Little More Complex

2.5 Action Methods

2.6 References and the Importance of text_lists

2.7 Layout Updates

2.8 What's an Update

2.8.1 What's a "Model"

2.9 Adding our Updates

2.10 Fully Armed and Operational References

2.11 Removing Blocks

2.11.1 Before (*Figure 2.3*)

2.11.2 After (*Figure 2.4*)

2.12 What's Next

Chapter 3

The Package Layout

3.1 The Why and Where of the Package Layout

3.2 Package Layout Examples

3.3 What is a Handle?

3.4 Rendering a Magento Layout

3.5 Getting a Handle on Handles

3.6 More local.xml

3.7 Adding Other Handles to the Page Layout

3.8 Package Layout Term Review

3.8.1 Package Layout

3.8.2 Page Layout

3.8.3 Layout Update XML Fragment

Chapter 4

Bringing it All Together

4.1 How a Magento Layout is Built

4.2 What is the Page Layout

4.3 Rendering a Layout

Chapter 5

Advanced Layout Features

5.1 Action Paramaters

5.2 Translation System

5.3 Conditional Method Calling

5.4 Dynamic Paramaters

5.5 Ordering of Blocks

5.6 Reordering Existing Blocks

5.7 Template Blocks Need Not Apply

5.8 Block Name vs. Block Alias

5.9 Skipping a Child

Chapter 6

CMS Pages

6.1 Creating a Page

6.1.1 Page Information : Page Title

6.1.2 Page Information : URL Key

6.1.3 Page Information : Store View

6.1.4 Page Information : Status

6.1.5 Content: Content Heading

6.1.6 Content: Editor

6.1.7 Meta : Keywords

6.1.8 Meta : Description

6.1.9 Design : Layout

6.1.10 Design : Layout Update XML

6.1.11 Design : Custom Design

6.2 CMS Page Rendering

6.3 Index Page

6.4 What You Need to Know

6.5 Where's the Layout?

6.6 Adding the CMS Blocks

6.7 Setting the Page Template

6.8 Rendering the Content Area

6.9 Page Content Filtering

6.10 Filtering Meta Programming

Chapter 7

Widgets

Consder 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 wiget 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
```

```
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&amp;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&amp;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!

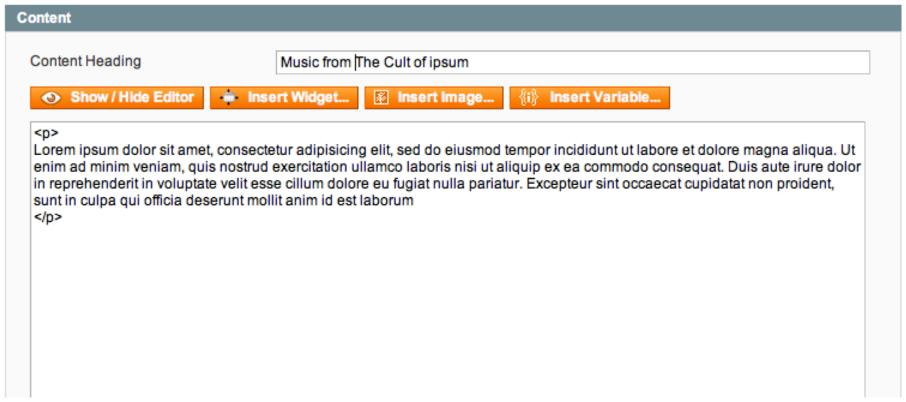
7.2 Adding a Widget to a CMS Page

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*)



Content Heading: Music from The Cult of ipsum

```
<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>
```

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*)

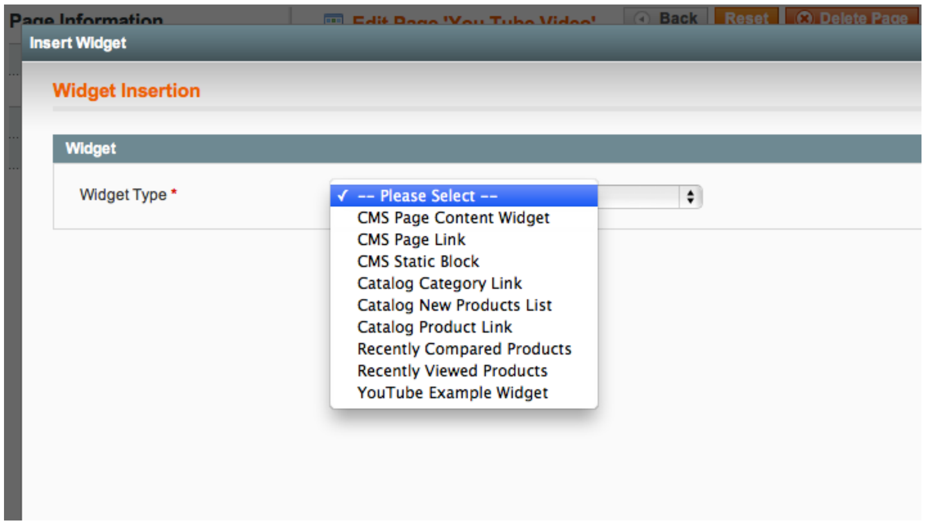


Figure 7.2

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 a 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&amp;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&amp;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.

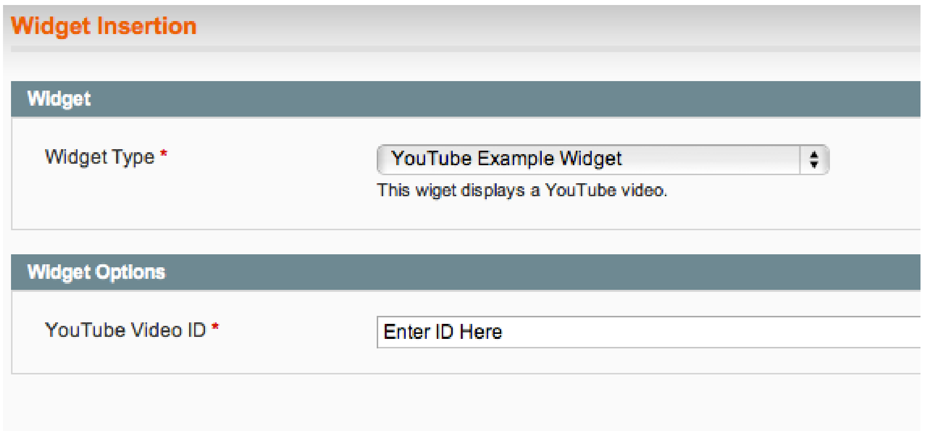
In your widget config, add a `<paramaters/>` node as defined below.

```
<!-- #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 wiget displays a YouTube video.
        </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 Type *
YouTube Example Widget
This wiget displays a YouTube video.

YouTube Video ID *
Enter ID Here

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 paramaters 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 paramater, 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 paramaters 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 removing 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&amp;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&amp;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

```
{{widget type="nofrills_booklayout/youtube" video_id="qYkbTyHXwbs"
template="youtube.phtml"}}
```

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 paramater, all we're really doing here is adding a new widget data paramater 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*)

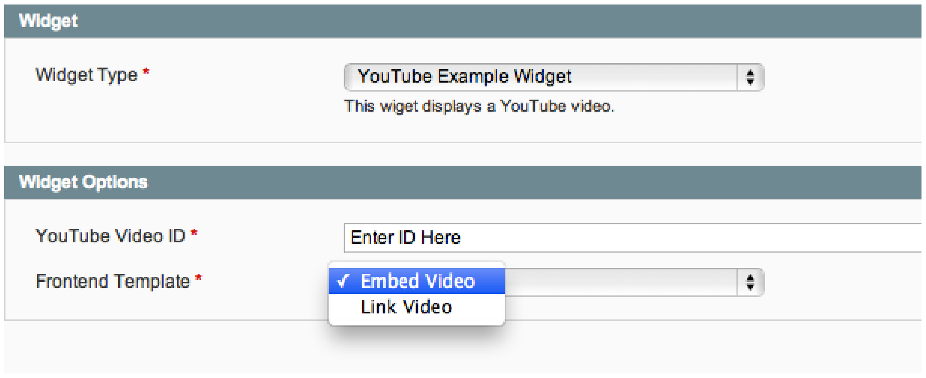


Figure 7.4

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 paramater 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.

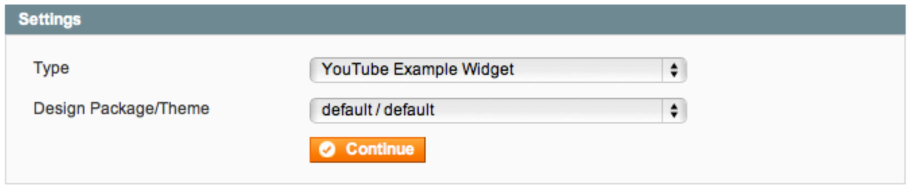
7.7 Creating an Instance Widget

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*)

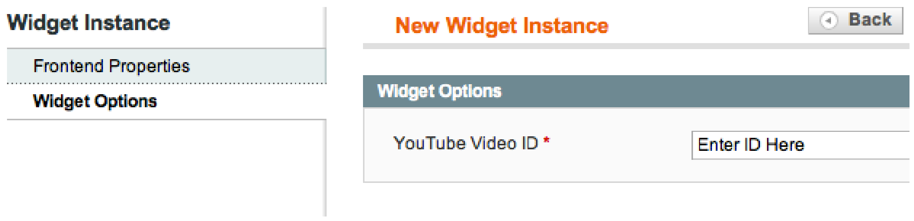


Type: YouTube Example Widget
Design Package/Theme: default / default

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*)

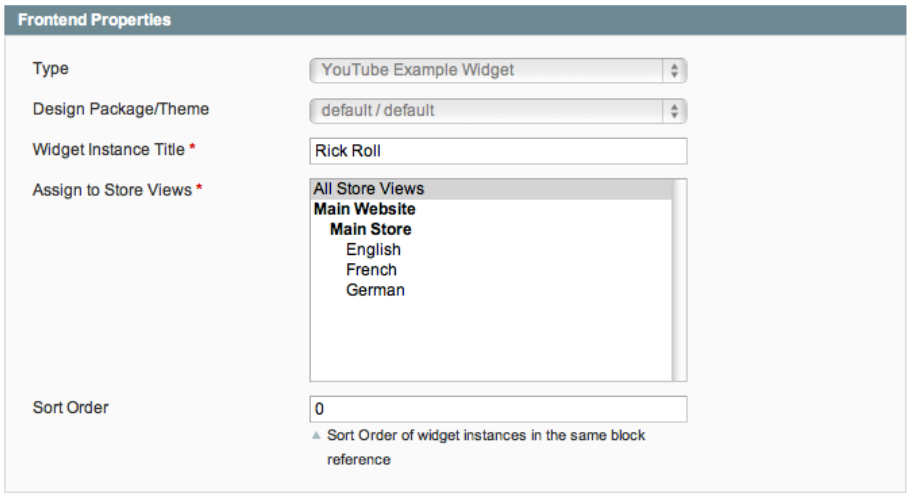


YouTube Video ID *: Enter ID Here

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*)



Frontend Properties

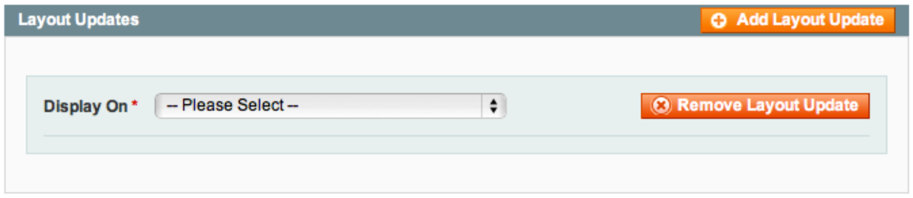
Type	YouTube Example Widget
Design Package/Theme	default / default
Widget Instance Title *	Rick Roll
Assign to Store Views *	All Store Views Main Website Main Store English French German
Sort Order	0

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*)



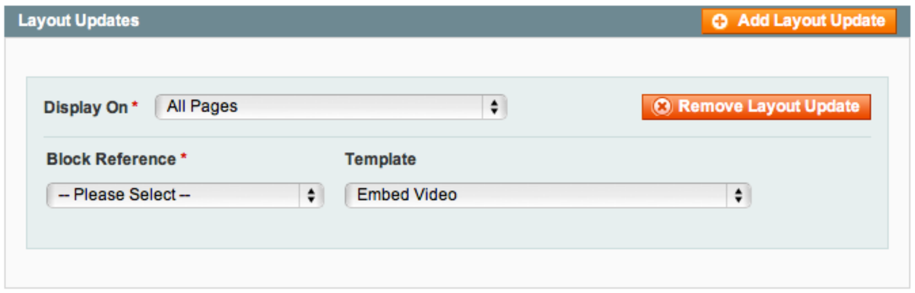
Display On * -- Please Select --

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**.

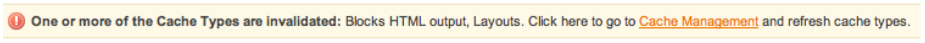
The first menu is defining **which block** you want to add your Widget Instance to. Select **Main Content Area**. The values in the second menu should look familiar to you. They're the templates we defined earlier. Select "Embed Video", and then Save you Widget Instance.



Display On * All Pages
Block Reference * -- Please Select --
Template Embed Video

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*.



One or more of the Cache Types are invalidated: Blocks HTML output, Layouts. Click here to go to Cache Management and refresh cache types.

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 the 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>
```

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 `<paramaters/>` 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/test_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 books 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 knowedge 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

Appendix B

Class Aliases

Magento uses a factory pattern for instantiating certain objects. Don't let the design patterny 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_Block_Product"`, and then a `"Mage_Catalog_Block_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 or 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, is 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

C.1 Adding a Module

C.2 Enabling your Module

C.3 Next Steps

Appendix D

Block Action Reference

Appendix E

Theme and Layout Resolution

E.1 Template Resolution

E.2 The Base Package

E.3 Layout Files

Appendix F

The Hows and Whys of Clearing Magento's Cache

Appendix G

Magento Setters and Getters

G.1 Getter and Setter

G.2 Other Magic Methods

Appendix H

Widget Field Rendering Options

H.1 Creating Your Own Form Elements

H.2 Advanced Examples

Appendix I

System Configuration Variables

Appendix J

Magento Connect

J.1 What is an Extension

J.2 Installing Extensions: The GUI Way

J.3 Installing Extensions: The Command Line Way

J.3.1 Magento Connect CLI install for Magento 1.42

J.3.2 Magento Connect CLI install for Magento 1.5+