Magento for Developers: Part 1 - Introduction to Magento

What is Magento? It's the most powerful online eCommerce platform in the universe and is changing the face of eCommerce forever. 

Of course, you already know that. What you may not realize is Magento's also an object-oriented PHP Framework that can be used to develop modern, dynamic web applications that tap into Magento's powerful eCommerce features.

This is the first in a series of articles in which we're going to go on a whirlwind tour of Magento's programming framework features. Don't worry if you don't follow everything immediately. As you study the system more everything in this article will start to make sense, and you'll soon be the envy of your colleagues stuck working with more primitive PHP systems.

In this article...

Or for the more visually oriented Magento_MVC.pdf.

Code Organized in Modules

Magento organizes its code into individual Modules. In a typical PHP **Model-View-Controller (MVC)** application, all the Controllers will be in one folder, all the Models in another, etc. In Magento, files are grouped together based on functionality, which are called **modules** in Magento.

Magento's Code

For example, you'll find Controllers, Models, Helpers, Blocks, etc. related to Magento's checkout functionality in

```
app/code/core/Mage/Checkout
```

You'll find Controllers, Models, Helpers, Blocks, etc. related to Magento's Google Checkout functionality in

```
app/code/core/Mage/GoogleCheckout
```

Your Code

When you want to customize or extend Magento, rather than editing core files directly, or even placing you new Controllers, Models, Helpers, Blocks, etc. next to Magento code, you'll create your own Modules in

```
app/code/local/Package/Modulename
```

Package (also often referred to as a **Namespace**) is a unique name that identifies your company or organization. The intent is that each member of the world-wide Magento community will use their own Package name when creating modules in order to avoid colliding with another user's code.

When you create a new Module, you need to tell Magento about it. This is done by adding an XML file to the folder:

```
app/etc/modules
```

There are two kinds of files in this folder, the first enables an individual Module, and is named in the form: `Packagename_Modulename.xml`

The second is a file that will enable multiple Modules from a Package/Namespace, and is named in the form: `Packagename_All.xml`

Configuration-Based MVC

Magento is a **configuration-based** MVC system. The alternative to this would a **convention-based** MVC system.

In a convention-based MVC system, if you wanted to add, say, a new Controller or maybe a new Model, you'd just create the file/class, and the system would pick it up automatically.

In a configuration-based system, like Magento, in addition to adding the new file/class to the codebase, you often need to explicitly tell the system about the new class, or new group of classes. In Magento, each Module has a file named config.xml. This file contains all the relevant configuration for a Magento Module. At runtime, all these files are loaded into one large configuration tree.

For example, want to use Models in your custom Module? You'll need to add some code to config.xml that tells Magento you want to use Models, as well as what the base class name for all your Models should be.

```
<models>
     <packagename>
          <class>Packagename_Modulename_Model</class>
    <packagename>
</models>
```

The same goes for Helpers, Blocks, Routes for your Controllers, Event Handlers, and more. Almost anytime you want to tap into the power of the Magento system, you'll need to make some change or addition to your config file.

Controllers

In any PHP system, the main PHP entry point remains a PHP file. Magento is no different, and that file is `index.php`.

However, you never CODE in index.php. In an MVC system, index.php will contains code/calls to code that does the following:

1. Examines the URL
2. Based on some set of rules, turns this URL into a Controller class and an Action method (called Routing)
3. Instantiates the Controller class and calls the Action method (called dispatching)

This means the **practical** entry point in Magento (or any MVC-based system) is a method in a Controller file. Consider the following URL:

```
http://example.com/catalog/category/view/id/25
```

Each portion of the path after the server name is parsed as follows.

Front Name - `catalog`

The first portion of the URL is called the front name. This, more or less, tells magento which Module it can find a Controller in. In the above example, the front name is *catalog*, which corresponds to the Module located at:

```
app/code/core/Mage/Catalog
```

Controller Name - `category`

The second portion of the URL tells Magento which Controller it should use. Each Module with Controllers has a special folder named 'controllers' which contains all the Controllers for a module. In the above example, the URL portion category is translated into the Controller file

```
app/code/core/Mage/Catalog/controllers/CategoryController.php
```

Which looks like

```
class Mage_Catalog_CategoryController extends Mage_Core_Controller_Front_Acti
on
{
}
```

All Controllers in the Magento cart application extend from Mage_Core_Controller_Front_Action.

Action Name - `view`

Third in our URL is the action name. In our example, this is "view". The word "view" is used to create the Action Method. So, in our example, "view" would be turned into "viewAction"

```
class Mage_Catalog_CategoryController extends Mage_Core_Controller_Front_Acti
on
{
    public function viewAction()
    {
        //main entry point
    }
}
```

People familiar with the Zend Framework will recognize the naming convention here.

Paramater/Value - `id/25`

Any path portions after the action name will be considered key/value GET request variables. So, in our example, the "id/25" means there will get a GET variable named "id", with a value of "25".

As previously mentioned, if you want your Module to use Controllers, you'll need to configure them. Below is the configuration chunk that enables Controllers for the Catalog Module

```
<frontend>
    <routers>
        <catalog>
            <use>standard</use>
            <args>
                <module>Mage_Catalog</module>
                <frontName>catalog</frontName>
            </args>
        </catalog>
    </routers>
</frontend>
```

Don't worry too much about the specifics right now, but notice the `<frontName>catalog</frontName>`

This is what links a Module with a URL frontname. Most Magento core Modules choose a frontname that is the same as their Module name, but this is not required.

Multiple Routers

The routing described above is for the Magento cart application (often called the frontend). If Magento doesn't find a valid Controller/Action for a URL, it tries again, this time using a second set of Routing rules for the Admin application. If Magento doesn't find a valid **Admin** Controller/Action, it uses a special Controller named `Mage_Cms_IndexController`.

The CMS Controller checks Magento's content Management system to see if there's any content that should be loaded. If it finds some, it loads it, otherwise the user will be presented with a 404 page.

For example, the main magento "index" page is one that uses the CMS Controller, which can often throw newcomers for a loop.

Context-Based URI Model Loading

Now that we're in our Action method entry point, we'll want to start instantiating classes that do things. Magento offers a special way to instantiate Models, Helpers and Blocks using static factory methods on the global Mage class. For example:

```
Mage::getModel('catalog/product');
Mage::helper('catalog/product');
```

The string 'catalog/product' is called a Grouped Class Name. It's also often called a URI. The first portion of any Grouped Class Name (in this case, catalog), is used to lookup which Module the class resides in. The second portion ('product' above) is used to determine which class should be loaded.

So, in both of the examples above, 'catalog' resolves to the Module `app/code/core/Mage/Catalog`.

Meaning our class name will start with `Mage_Catalog`.

Then, product is added to get the final class name

```
Mage::getModel('catalog/product');
Mage_Catalog_Model_Product

Mage::helper('catalog/product');
Mage_Catalog_Helper_Product
```

These rules are bound by what's been setup in each Module's config file. When you create your own custom Module, you'll have your own grouped classnames to work with `Mage::getModel('myspecialprefix/modelname');`.

You don't **have** to use Groupd Class Names to instantiate your classes. However, as we'll learn later, there are certain advantages to doing so.

Magento Models

Magento, like most frameworks these days, offers an Object Relational Mapping (ORM) system. ORMs get you out of the business of writing SQL and allow you to manipulate a datastore purely through PHP code. For example:

```
$model = Mage::getModel('catalog/product')->load(27);
$price = $model->getPrice();
$price += 5;
$model->setPrice($price)->setSku('SK83293432');
$model->save();
```

In the above example we're calling the methods "getPrice" and "setPrice" on our Product. However, the Mage_Catalog_Model_Product class has no methods with these names. That's because Magento's ORM uses PHP's *_get and* _set magic methods.

Calling the method `$product->getPrice();` will "get" the Model attribute "price".

Calling `$product->setPrice();` will "set" the Model attribute "price". All of this assumes the Model class doesn't already have methods named getPrice or setPrice. If it does, the magic methods will be bypassed. If you're interested in the implementation of this, checkout the Varien_Object class, which all Models inherit from.

If you wanted to get all the available data on a Model, call `$product->getData();` to get an array of all the attributes.

You'll also notice it's possible to chain together several calls to the set method:
`$model->setPrice($price)->setSku('SK83293432');`

That's because each set method returns and instance of the Model. This is a pattern you'll see used in much of the Magento codebase.

Magento's ORM also contains a way to query for multiple Objects via a Collections interface. The following would get us a collection of all products that cost $5.00

```
$products_collection = Mage::getModel('catalog/product')
->getCollection()
->addAttributeToSelect('*')
->addFieldToFilter('price','5.00');
```

Again, you'll notice Magento's implemented a chaining interface here. Collections use the PHP Standard Library to implement Objects that have array like properties.

```
foreach($products_collection as $product)
{
    echo $product->getName();
}
```

You may be wondering what the "addAttributeToSelect" method is for. Magento has two broad types of Model objects. One is a traditional "One Object, One Table" Active Record style Model. When you instantiate these Models, all attributes are automatically selected.

The second type of Model is an Entity Attribute Value (EAV) Model. EAV Models spread data across several different tables in the database. This gives the Magento system the flexibility to offer its flexible product attribute system without having to do a schema change each time you add an attribute. When creating a collection of EAV objects, Magento is conservative in the number of columns it will query for, so you can use addAttributeToSelect to get the columns you want, or addAttributeToSelect('*') to get all columns.

Helpers

Magento's Helper classes contain utility methods that will allow you to perform common tasks on objects and variables. For example:

```
$helper = Mage::helper('catalog');
```

You'll notice we've left off the second part of the grouped class name. Each Module has a default Data Helper class. The following is equivalent to the above:

```
$helper = Mage::helper('catalog/data');
```

Most Helpers inherit form Mage_Core_Helper_Abstract, which gives you several useful methods by default.

```
$translated_output =  $helper-
>__('Magento is Great'); //gettext style translations
if($helper->isModuleOutputEnabled()): //is output for this module on or off?
```

Layouts

So, we've seen Controllers, Models, and Helpers. In a typical PHP MVC system, after we've manipulated our Models we would

1. Set some variables for our view
2. The system would load a default "outer" HTML layout>
3. The system would then load our view inside that outer layout

However, if you look at a typical Magento Controller action, you don't see any of this:

```
/**
 * View product gallery action
 */
public function galleryAction()
{
    if (!$this->_initProduct()) {
        if (isset($_GET['store']) &amp;&amp; !$this->getResponse()-
>isRedirect()) {
            $this->_redirect('');
```

```
        } elseif (!$this->getResponse()->isRedirect()) {
            $this->_forward('noRoute');
        }
        return;
    }
    $this->loadLayout();
    $this->renderLayout();
}
```

Instead, the Controller action ends with two calls

```
$this->loadLayout();
$this->renderLayout();
```

So, the "V" in Magento's MVC already differs from what you're probably used to, in that you need to explicitly kick off rendering the layout.

The layout itself also differs. A Magento Layout is an object that contains a nested/tree collection of "Block" objects. Each Block object will render a specific bit of HTML. Block objects do this through a combination of PHP code, and including PHP .phtml template files.

Blocks objects are meant to interact with the Magento system to retrieve data from Models, while the phtml template files will produce the HTML needed for a page.

For example, the page header Block `app/code/core/Mage/Page/Block/Html/Head.php` uses the head.phtml file `page/html/head.phtml`.

Another way of thinking about it is the Block classes are almost like little mini-controllers, and the .phtml files are the view.

By default, when you call

```
$this->loadLayout();
$this->renderLayout();
```

Magento will load up a Layout with a skeleton site structure. There will be Structure Blocks to give you your `html`, `head`, and `body`, as well as HTML to setup single or multiple columns of Layout. Additionally, there will be a few Content Blocks for the navigation, default welcome message, etc.

"Structure" and "Content" are arbitrary designations in the Layout system. A Block doesn't programmatically know if it's Structure or Content, but it's useful to think of a Block as one or the other.

To add Content to this Layout you need to tell the Magento system something like

```
"Hey, Magento, add these additional Blocks under the "content" Block of the
skeleton"
```

or

```
"Hey, Magento, add these additional Blocks under the "left column" Block of
the skeleton"
```

This can be done programmatically in a Controller action

```
public function indexAction()
{
    $block = $this->getLayout()->createBlock('adminhtml/system_account_edit')
    $this->getLayout()->getBlock('content')->append($block);
}
```

but more commonly (at least in the frontend cart application), is use of the XML Layout system.

The Layout XML files in a theme allow you, on a per Controller basis, to remove Blocks that would normally be rendered, or add Blocks to that default skeleton areas. For example, consider this Layout XML file:

```
<catalog_category_default>
    <reference name="left">
        <block type="catalog/navigation" name="catalog.leftnav" after="curren
cy" template="catalog/navigation/left.phtml"/>
    </reference>
</catalog_category_default>
```

It's saying in the catalog Module, in the category Controller, and the default Action, insert the the catalog/navigation Block into the "left" structure Block, using the catalog/navigation/left.phtml template.

One last important thing about Blocks. You'll often see code in templates that looks like this:
`$this->getChildHtml('order_items')`

This is how a Block renders a nested Block. However, a Block can only render a child Block if the child Block **is included as a nested Block in the Layout XML file**. In the example above our catalog/navigation Block has no nested Blocks. This means any call to $this->getChildHtml() in left.phtml will render as blank.

If, however, we had something like:

```
<catalog_category_default>
    <reference name="left">
        <block type="catalog/navigation" name="catalog.leftnav" after="curren
cy" template="catalog/navigation/left.phtml">
            <block type="core/template" name="foobar" template="foo/baz/bar.p
html"
        </block>
    </reference>
</catalog_category_default>
```

From the catalog/navigation Block, we'd be able to call
`$this->getChildHtml('foobar');`

Observers

Like any good object-oriented system, Magento implements an Event/Observer pattern for end users to hook into. As certain actions happen during a Page request (a Model is saved, a user logs in, etc.), Magento will issue an event signal.

When creating your own Modules, you can "listen" for these events. Say you wanted to get an email every time a certain customer logged into the store. You could listen for the "customer_login" event (setup in config.xml)

```
<events>
    <customer_login>
        <observers>
            <unique_name>
                <type>singleton</type>
                <class>mymodule/observer</class>
                <method>iSpyWithMyLittleEye</method>
            </unique_name>
        </observers>
    </customer_login>
</events>
```

and then write some code that would run whenever a user logged in:

```
<pre><code>class Packagename_Mymodule_Model_Observer
{
    public function iSpyWithMyLittleEye($observer)
    {
        $data = $observer->getData();
        //code to check observer data for out user,
        //and take some action goes here
    }
}
```

Class Overrides

Finally, the Magento System offers you the ability to replace Model, Helper and Block classes from the core modules with your own. This is a feature that's similar to "Duck Typing" or "Monkey Patching" in a language like Ruby or Python.

Here's an example to help you understand. The Model class for a product is `Mage_Catalog_Model_Product`.

Whenever the following code is called, a Mage_Catalog_Model_Product object is created

```
$product = Mage::getModel('catalog/product');
```

This is a factory pattern.

What Magento's class override system does is allow you to tell the system

```
"Hey, whenever anyone asks for a catalog/product, instead of giving them a
Mage_Catalog_Model_Product,
give them a Packagename_Modulename_Model_Foobazproduct instead".
```

Then, if you want, your Packagename_Modulename_Model_Foobazproduct class can extend the original product class

```
class Packagename_Modulename_Model_Foobazproduct extends Mage_Catalog_Model_P
roduct
{
}
```

Which will allow you to change the behavior of any method on the class, but keep the functionality of the existing methods.

```
class Packagename_Modulename_Model_Foobazproduct extends Mage_Catalog_Model_P
roduct
{
    public function validate()
    {
        //add custom validation functionality here
        return $this;
    }

}
```

As you might expect, this overriding (or rewriting) is done in the config.xml file.

```
<models>
    <!-- tells the system this module has models -->
    <modulename>
        <class>Packagename_Modulename_Model</class>
    </modulename>

    <!-- does the override for catalog/product-->
    <catalog>
        <rewrite>
            <product>Packagename_Modulename_Model_Foobazproduct</product>
        </rewrite>
    </catalog>
</models>
```

One thing that's important to note here. Individual classes in **your** Module are overriding individual classes in **other** Modules. You are not, however, overriding the entire Module. This allows you to change specific method behavior without having to worry what the rest of the Module is doing.

Wrap Up

We hope you've enjoyed this whirlwind tour of some of the features the Magento eCommerce system offers to developers. It can be a little overwhelming at first, especially if this is your first experience with a modern, object-oriented PHP system. If you start to get frustrated, take a deep breath, remind yourself that this is new, and new things are hard, but at the end of the day it's just a different way of coding. Once you get over the learning curve you'll find yourself loath to return to other, less powerful systems.

Magento for Developers: Part 2 - The Magento Config

The config is the beating heart of the Magento System. It describes, in whole, almost any module, model, class, template, etc. than you'll need to access. It's a level of abstraction that most PHP developers aren't used to working with, and while it adds development time in the form of confusion and head scratching, it also allows you an unprecedented amount of flexibility as far as overriding default system behaviors go.

To start with, we're going to create a Magento module that will let us view the system config in our web browser. Follow along by copying and pasting the code below, it's worth going through on your own as a way to start getting comfortable with things you'll be doing while working with Magento, as well as learning key terminology.

In this article...

Setting up a Module Directory Structure

We're going to be creating a Magento module. A module is a group of php and xml files meant to extend the system with new functionality, or override core system behavior. This may meaning adding additional data models to track sales information, changing the behavior of existing classes, or adding entirely new features.

It's worth noting that most of the base Magento system is built using the same module system you'll be using. If you look in

```
app/code/core/Mage
```

each folder is a separate module built by the Magento team. Together, these modules form the community shopping cart system you're using. Your modules should be placed in the following folder

```
app/code/local/Packagename
```

"Packagename" should be a unique string to Namespace/Package your code. It's an unofficial convention that this should be the name of your company. The idea is to pick a string that no one else if the world could possibly be using.

```
app/code/local/Microsoft
```

We'll use "Magentotutorial".

So, to add a module to your Magento system, create the following directory structure

```
app/code/local/Magentotutorial/Configviewer/Block
app/code/local/Magentotutorial/Configviewer/controllers
app/code/local/Magentotutorial/Configviewer/etc
app/code/local/Magentotutorial/Configviewer/Helper
app/code/local/Magentotutorial/Configviewer/Model
app/code/local/Magentotutorial/Configviewer/sql
```

You won't need all these folder for every module, but setting them all up now is a smart idea.

Next, there's two files you'll need to create. The first, config.xml, goes in the etc folder you just created.

```
app/code/local/Magentotutorial/Configviewer/etc/config.xml
```

The second file should be created at the following location

```
app/etc/modules/Magentotutorial_Configviewer.xml
```

The naming convention for this files is Packagename_Modulename.xml.

The config.xml file should contain the following XML. Don't worry too much about what all this does for now, we'll get there eventually

```
<config>
    <modules>
        <Magentotutorial_Configviewer>
            <version>0.1.0</version>
        </Magentotutorial_Configviewer>
    </modules>
</config>
```

Finally, Magentotutorial_Configviewer.xml should contain the following xml.

```
<config>
    <modules>
        <Magentotutorial_Configviewer>
            <active>true</active>
            <codePool>local</codePool>
        </Magentotutorial_Configviewer>
    </modules>
</config>
```

That's it. You now have a bare bones module that won't do anything, but that Magento will be aware of. To make sure you've done everything right, do the following:

1. Clear your Magento cache
2. In the Magento Admin, go to **System->Configuration->Advanced**
3. In the "Disable modules output" panel verify that Magentotutorial_Configviewer shows up

Congratulations, you've built your first Magento module!

Creating a Module Config

Of course, this module doesn't do anything yet. When we're done, our module will

1. Check for the existence of a "showConfig" query string variable
2. If showConfig is present, display our Magento config and halt normal execution
3. Check for the existence of an additional query string variable, showConfigFormat that will let us specify text or xml output.

First, we're going to add the following <global> section to our config.xml file.

```
<config>
    <modules>...</modules>
    <global>
        <events>
            <controller_front_init_routers>
                <observers>
                    <Magentotutorial_configviewer_model_observer>
                        <type>singleton</type>
                        <class>Magentotutorial_Configviewer_Model_Observer</c
lass>
                        <method>checkForConfigRequest</method>
                    </Magentotutorial_configviewer_model_observer>
                </observers>
            </controller_front_init_routers>
        </events>
    </global>
</config>
```

Then, create a file at

```
Magentotutorial/Configviewer/Model/Observer.php
```

and place the following code inside

```
<?php
    class Magentotutorial_Configviewer_Model_Observer {
        const FLAG_SHOW_CONFIG = 'showConfig';
        const FLAG_SHOW_CONFIG_FORMAT = 'showConfigFormat';
```

```
        private $request;

        public function checkForConfigRequest($observer) {
            $this->request = $observer->getEvent()->getData('front')-
>getRequest();
            if($this->request->{self::FLAG_SHOW_CONFIG} === 'true'){
                $this->setHeader();
                $this->outputConfig();
            }
        }

        private function setHeader() {
            $format = isset($this->request-
>{self::FLAG_SHOW_CONFIG_FORMAT}) ?
            $this->request-
>{self::FLAG_SHOW_CONFIG_FORMAT} : 'xml';
            switch($format){
                case 'text':
                    header("Content-Type: text/plain");
                    break;
                default:
                    header("Content-Type: text/xml");
            }
        }

        private function outputConfig() {
            die(Mage::app()->getConfig()->getNode()->asXML());
        }
    }
```

That's it. Clear your Magento cache again and then load any Magento URL with a `showConfig=true` query string

```
http://magento.example.com/?showConfig=true
```

What am I looking at?

You should be looking at a giant XML file. This describes the state of your Magento system. It lists all modules, models, classes, event listeners or almost anything else you could think of.

For example, consider the config.xml file you created above. If you search the XML file in your browser for the text `Configviewer_Model_Observer` you'll find your class listed. Every module's config.xml file is parsed by Magento and included in the global config.

Why Do I Care?

Right now this may seem esoteric, but this config is key to understanding Magento. Every module you'll be creating will add to this config, and anytime you need to access a piece of core system functionality, Magento will be referring back to the config to look something up.

A quick example: As an MVC developer, you've likely worked with some kind of helper class, instantiated something like

```
$helper_sales = new HelperSales();
```

One of the things Magento has done is abstract away PHP's class declaration. In Magento, the above code looks something like

```
$helper_sales = Mage::helper('sales');
```

In plain english, the static helper method will:

1. Look in the <helpers /> section of the Config.
2. Within <helpers />, look for a <sales /> section
3. Within the <sales /> section look for a <class /> section
4. Instantiate the class found in #3 (Mage_SalesRule_Helper)

While this seems like a lot of work (and it is), the key advantage is by always looking to the config file for class names, we can override core Magento functionality **without** changing or adding to the core code. This level of meta programming, not usually found in PHP, allows you to cleanly extend only the parts of the system you need to.

Magento for Developers: Part 3 - Magento Controller Dispatch

The Model-View-Controller (MVC) architecture traces its origins back to the Smalltalk Programming language and Xerox Parc. Since then, there have been many systems that describe their architecture as MVC. Each system is slightly different, but all have the goal of separating data access, business logic, and user-interface code from one another.

The architecture of most PHP MVC frameworks will looks something like this.

1. A URL is intercepted by a single PHP file (usually called a Front Controller).
2. This PHP file will examine the URL, and derive a Controller name and an Action name (a process that's often called routing).
3. The derived Controller is instantiated.
4. The method name matching the derived Action name is called on the Controller.
5. This Action method will instantiate and call methods on models, depending on the request variables.
6. The Action method will also prepare a data structure of information. This data structure is passed on to the view.
7. The view then renders HTML, using the information in the data structure it has received from the Controller.

While this pattern was a great leap forward from the "each php file is a page" pattern established early on, for some software engineers, it's still an ugly hack. Common complaints are:

- The Front Controller PHP file still operates in the global namespace.
- Convention over configuration leads to less modularity.
    - URLs routing is often inflexible.
    - Controllers are often bound to specific views.
    - Even when a system offers a way to override these defaults, the convention leads to applications where it's difficult/impossible to drop in new a new model, view, or Controller implementation without massive re-factoring.

As you've probably guessed, the Magento team shares this world view and has created a more abstract MVC pattern that looks something like this:.

1. A URL is intercepted by a single PHP file.
2. This PHP file instantiates a Magento application.
3. The Magento application instantiates a Front Controller object.
4. Front Controller instantiates any number of Router objects (specified in global config).
5. Routers check the request URL for a "match".
6. If a match is found, an Action Controller and Action are derived.
7. The Action Controller is instantiated and the method name matching the Action Name is called.
8. This Action method will instantiate and call methods on models, depending on the request.
9. This Action Controller will then instantiate a Layout Object.
10. This Layout Object will, based some request variables and system properties (also known as "handles"), create a list of Block objects that are valid for this request.
11. Layout will also call an output method on certain Block objects, which start a nested rendering (Blocks will include other Blocks).
12. Each Block has a corresponding Template file. Blocks contain PHP logic, templates contain HTML and PHP output code.
13. Blocks refer directly back to the models for their data. In other words, **the Action Controller does not pass them a data structure.**

We'll eventually touch on each part of this request, but for now we're concerned with the **Front Controller -> Routers -> Action Controller** section.

Hello World

Enough theory, it's time for Hello World. We're going to

1. Create a Hello World module in the Magento system
2. Configure this module with routes
3. Create Action Controller(s) for our routes

Create Hello World Module

First, we'll create a directory structure for this module. Our directory structure should look as follows:

```
app/code/local/Magentotutorial/Helloworld/Block
app/code/local/Magentotutorial/Helloworld/controllers
app/code/local/Magentotutorial/Helloworld/etc
app/code/local/Magentotutorial/Helloworld/Helper
app/code/local/Magentotutorial/Helloworld/Model
app/code/local/Magentotutorial/Helloworld/sql
```

Then create a configuration file for the module (at path `app/code/local/Magentotutorial/Helloworld/etc/config.xml`):

```
<config>
    <modules>
        <Magentotutorial_Helloworld>
            <version>0.1.0</version>
        </Magentotutorial_Helloworld>
    </modules>
</config>
```

Then create a file to activate the module (at path `app/etc/modules/Magentotutorial_Helloworld.xml`):

```
<config>
    <modules>
        <Magentotutorial_Helloworld>
            <active>true</active>
            <codePool>local</codePool>
        </Magentotutorial_Helloworld>
    </modules>
</config>
```

Finally, we ensure the module is active:

1. Clear your Magento cache.
2. In the Magento Admin, go to **System->Configuration->Advanced**.
3. Expand "Disable Modules Output" (if it isn't already).
4. Ensure that Magentotutorial_Helloworld shows up.

Configuring Routes

Next, we're going to configure a route. A route will turn a URL into an Action Controller and a method. Unlike other convention based PHP MVC systems, with Magento you need to explicitly define a route in the global Magento config.

In your config.xml file, add the following section:

```
<config>
    ...
    <frontend>
        <routers>
            <helloworld>
                <use>standard</use>
                <args>
                    <module>Magentotutorial_Helloworld</module>
                    <frontName>helloworld</frontName>
                </args>
            </helloworld>
        </routers>
    </frontend>
    ...
</config>
```

We have a lot of new terminology here, let's break it down.

What is a <frontend>?

The <frontend> tag refers to a Magento Area. For now, think of Areas as individual Magento applications. The "frontend" Area is the public facing Magento shopping cart application. The "admin" Area is the the private administrative console application. The "install" Area is the application you use to run though installing Magento the first time.

Why a <routers> tags if we're configuring individual routes?

There's a famous quote about computer science, often attributed to Phil Karlton:

"There are only two hard things in Computer Science: cache invalidation and naming things"

Magento, like all large systems, suffers from the naming problem in spades. You'll find there are are many places in the global config, and the system in general, where the naming conventions seem unintuitive or even ambiguous. This is one of those places. Sometimes the `<routers>` tag will enclose configuration information about routers, other times it will enclose configuration information about the actual router objects that do the routing. This is going to seem counter intuitive at first, but as you start to work with Magento more and more, you'll start to understand its world view a little better. (Or, in the words of Han Solo, "Hey, trust me!").

What is a <frontName>?

When a router parses a URL, it gets separated as follows

```
http://example.com/frontName/actionControllerName/actionMethod/
```

So, by defining a value of "helloworld" in the <frontName> tags, we're telling Magento that we want the system to respond to URLs in the form of

```
http://example.com/helloworld/*
```

Many developers new to Magento confuse this frontName with the Front Controller object. They are **not** the same thing. The frontName belongs solely to routing.

What's the <helloworld> tag for?

This tag should be the lowercase version of you module name. Our module name is Helloworld, this tag is helloworld.

You'll also notice our frontName matches our module name. It's a loose convention to have frontNames match the module names, but it's not a requirement. In your own modules, it's probably better to use a route name that's a combination of your module name and package name to avoid possible namespace collisions.

What's <module>Magentotutorial_Helloworld</module> for?

This module tag should be the full name of your module, including its package/namespace name. This will be used by the system to locate your Controller files.

Create Action Controller(s) for our Routes

One last step to go, and we'll have our Action Controller. Create a file at

```
app/code/local/Magentotutorial/Helloworld/controllers/IndexController.php
```

That contains the following

```
class Magentotutorial_Helloworld_IndexController extends Mage_Core_Controller
_Front_Action {

    public function indexAction() {

        echo 'Hello Index!';

    }

}
```

Clear your config cache, and load the following URL

```
http://exmaple.com/helloworld/index/index
```

You should also be able to load

http://exmaple.com/helloworld/index/
http://exmaple.com/helloworld/

You should see a blank page with the text "Hello World". Congratulations, you've setup your first Magento Controller!

Where do Action Controllers go?

Action Controllers should be placed in a module's `controllers` (lowercase c) folder. This is where the system will look for them.

How should Action Controllers be named?

Remember the <module> tag back in config.xml?

```
<module>Magentotutorial_Helloworld</module>
```

An Action Controller's name will

1. Start with this string specified in config.xml (`Magentotutorial_Helloworld`)
2. Be followed by an underscore (`Magentotutorial_Helloworld_`)
3. Which will be followed by the Action Controller's name (`Magentotutorial_Helloworld_Index`)
4. And finally, the word "Controller" (`Magentotutorial_Helloworld_IndexController`)

All Action Controllers need `Mage_Core_Controller_Front_Action` as an ancestor.

What's that index/index nonsense?

As we previously mentioned, Magento URLs are routed (by default) as follows

```
http://example.com/frontName/actionControllerName/actionMethod/
```

So in the URL

```
http://example.com/helloworld/index/index
```

the URI portion "helloworld" is the frontName, which is followed by index (The Action Controller name), which is followed by another index, which is the name of the Action Method that will be called. (an Action of index will call the method `public function indexAction(){...}`.

If a URL is incomplete, Magento uses "index" as the default, which is why the following URLs are equivalent.

http://example.com/helloworld/index
http://example.com/helloworld

If we had a URL that looked like this

```
http://example.com/checkout/cart/add
```

Magento would

1. Consult the global config to find the module to use for the frontName checkout (`Mage_Checkout`)
2. Look for the cart Action Controller (`Mage_Checkout_CartController`)
3. Call the `addAction` method on the cart Action Controller

Other Action Controller Tricks

Let's try adding a non-default method to our Action Controller. Add the following code to IndexController.php

```
public function goodbyeAction() {
    echo 'Goodbye World!';
}
```

And then visit the URL to test it out:

```
http://example.com/helloworld/index/goodbye
```

Because we're extending the `Mage_Core_Controller_Front_Action` class, we get some methods for free. For example, additional URL elements are automatically parsed into key/value pairs for us. Add the following method to your Action Controller.

```
public function paramsAction() {
    echo '<dl>';
    foreach($this->getRequest()->getParams() as $key=>$value) {
        echo '<dt><strong>Param: </strong>'.$key.'</dt>';
        echo '<dt><strong>Value: </strong>'.$value.'</dt>';
    }
    echo '</dl>';
}
```

and visit the following URL

```
http://example.com/helloworld/index/params?foo=bar&baz=eof
```

You should see each parameter and value printed out.

Finally, what would we do if we wanted a URL that responded at

```
http://example.com/helloworld/messages/goodbye
```

Here our Action Controller's name is messages, so we'd create a file at

```
app/code/local/Magentotutorial/Helloworld/controllers/MessagesController.php
```

with an Action Controller named `Magentotutorial_Helloworld_MessagesController` and an Action Method that looked something like

```
public function goodbyeAction()
{
    echo 'Another Goodbye';
}
```

And that, in a nutshell, is how Magento implements the Controller portion of MVC. While it's a little more complicated than other PHP MVC framework's, it's a highly flexible system that will allow you build almost any URL structure you want.

Magento for Developers: Part 4 - Magento Layouts, Blocks and Templatese

Developers new to Magento are often confused by the Layout and View system. This article will take a look at Magento's Layout/Block approach, and show you how it fits into Magento MVC worldview.

Unlike many popular MVC systems, Magento's Action Controller does **not** pass a data object to the view or set properties on the view object. Instead, the View component directly references system models to get the information it needs for display.

One consequence of this design decision is that the View has been separated into Blocks and Templates. Blocks are PHP objects, Templates are "raw" PHP files (with a .phtml extension) that contain a mix of HTML and PHP (where PHP is used as a templating language). Each Block is tied to a single Template file. Inside a phtml file, PHP's $this keyword will contain a reference to the Template's Block object.

A quick example

Take a look a the default product Template at the file at

`app/design/frontend/base/default/template/catalog/product/list.phtml`

You'll see the following PHP template code.

```
<?php $_productCollection=$this->getLoadedProductCollection() ?>
    <?php if(!$_productCollection->count()): ?> <div class="note-msg">
        <?php echo $this-
>__("There are no products matching the selection.") ?>    </div>
    <?php else: ?>
```

The `getLoadedProductCollection` method can be found in the Template's Block, `Mage_Catalog_Block_Product_List` as shown:

```
File: app/code/core/Mage/Catalog/Block/Product/List.php
...
public function getLoadedProductCollection()
{
    return $this->_getProductCollection();
}
...
```

The block's `_getProductCollection` then instantiates models and reads their data, returning a result to the template.

Nesting Blocks

The real power of Blocks/Templates come with the `getChildHtml` method. This allows you to include the contents of a secondary Block/Template inside of a primary Block/Template.

Blocks calling Blocks calling Blocks is how the entire HTML layout for your page is created. Take a look at the one column layout Template.

```
File: app/design/frontend/base/default/template/page/one-column.phtml
<!DOCTYPE html PUBLIC "-
//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="?php echo $this-
>getLang() ?>" lang="<?php echo $this->getLang() ?>">
<head>
<?php echo $this->getChildHtml('head') ?>
</head>
<body class="page-popup <?php echo $this->getBodyClass()?$this-
>getBodyClass():'' ?>">
    <?php echo $this->getChildHtml('content') ?>
    <?php echo $this->getChildHtml('before_body_end') ?>
    <?php echo $this->getAbsoluteFooter() ?>
</body>
```

The template itself is only 11 lines long. However, each call to `$this->getChildHtml(...)` will include and render another Block. These Blocks will, in turn, use `getChildHtml` to render other Blocks. It's Blocks all the way down.

The Layout

So, Blocks and Templates are all well and good, but you're probably wondering

1. How do I tell Magento which Blocks I want to use on a page?
2. How do I tell Magento which Block I should start rendering with?
3. How do I specify a particular Block in `getChildHtml(...)`? Those argument strings don't look like Block names to me.

This is where the Layout Object enters the picture. The Layout Object is an XML object that will define which Blocks are included on a page, and which Block(s) should kick off the rendering process.

Last time we were echoing content directly from out Action Methods. This time let's create a simple HTML template for our Hello World module.

First, create a file at

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

with the following contents

```
<layout version="0.1.0">
    <default>
        <reference name="root">
            <block type="page/html" name="root" output="toHtml" template="../
../../../../code/local/Magentotutorial/Helloworld/simple_page.phtml" />
        </reference>
    </default>
</layout>
```

Then, create a file at

```
app/code/local/Magentotutorial/Helloworld/simple_page.phtml
```

with the following contents

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title>Untitled</title>
    <meta name="generator" content="BBEdit 9.2" />
    <style type="text/css">
        body {
            background-color:#f00;
        }
    </style>
</head>
<body>

</body>
</html>
```

Finally, each Action Controller is responsible for kicking off the layout process. We'll need to add two method calls to the Action Method.

```
public function indexAction() {
    //remove our previous echo
    //echo 'Hello Index!';
    $this->loadLayout();
```

```
    $this->renderLayout();
}
```

Clear your Magento cache and reload your Hello World controller page. You should now see a website with a bright red background and an HTML source that matches what's in simple_page.phtml.

What's Going On

So, that's a lot of voodoo and cryptic incantations. Let's take a look at what's going on.

First, you'll want to install the Layoutviewer module. This is a module similar to the Configviewer module you built in the Hello World article that will let us peek at some of Magento's internals.

Once you've installed the module (similar to how you setup the Configviewer module), go to the following URL

```
http://example.com/helloworld/index/index?showLayout=page
```

This is the layout xml for your page/request. It's made up of <block />, <reference /> and <remove /> tags. When you call the `loadLayout` method of your Action Controller, Magento will

1. Generate this Layout XML
2. Instantiate a Block class for each <block /> and <reference /> tag, looking up the class using the tag's name attribute as a global config path and store in the internal `_blocks` array of the layout object.
3. If the <block /> tag contains an output attribute, its value is added to the internal `_output` array of the layout object.

Then, when you call the `renderLayout` method in your Action Controller, Magento will iterate over all the Blocks in the `_blocks` array, using the value of the output attribute as a callback method. This is almost always toHtml, and means the starting point for output will be that Block's Template.

The following sections will cover how Blocks are instantiated, how this layout file is generated, and finishes up with kicking off the output process.

Block Instantiation

So, within a Layout XML file, a <block /> or <reference /> has a "type" that's actually a Grouped Class Name URI

```
<block type="page/html" ...
<block type="page/template_links"
```

The URI references a location in the (say it with me) global config. The first portion of the URI (in the above examples `page`) will be used to query the global config to find the page class name. The second portion of the URI (in the two examples above, `html` and `template_links`) will be appended to the base class name to create the class Magento should instantiate.

We'll go through `page/html` as an example. First, Magento looks for the global config node at

```
/global/blocks/page
```

and finds

```
<page>
    <class>
        Mage_Page_Block
    </class>
</page>
```

This gives us our base class name `Mage_Page_Block`. Then, the second part of the URI (`html`) is appended to the class name to give us our final Block class name `Mage_Page_Block_Html`. This is the class that will be instantiated.

Remember, Blocks are one of the **Grouped Class Names** in Magento, all which share a similar instantiation method.

The Difference Between <block /> and <reference />

We mentioned that both <blocks /> and <refernces /> will instantiate Block classes, and you're probably wondering what the difference is.

<reference />'s are used to **replace** existing Blocks in a layout file. For example, consider the following layout snippet.

```
<block type="page/html" name="root" output="toHtml" template="page/2columns-
left.phtml">
    <!-- ... sub blocks ... -->
</block>
<!-- ... -->
<reference name="root">
    <block type="page/someothertype" name="root" template="path/to/some/other
/template" />
    <!-- ... sub blocks ... -->
    </block>
</reference>
```

Magento initially creates a `page/html` Block named `root`. Then, when it later encounters the reference with the same name (`root`), it will replace the original root <block /> with the <block /> enclosed in the <reference />.

This is what we've done in our local.xml file from above.

```
<layout version="0.1.0">
    <default>
        <reference name="root">
            <block type="page/html" name="root" output="toHtml" template="../
../../../../code/local/Magentotutorial/Helloworld/simple_page.phtml" />
        </reference>
    </default>
</layout>
```

The Block named `root` has been replaced with our Block, which points at a different phtml Template file.

How Layout Files are Generated

So, we have a slightly better understanding of what's going on with the Layout XML, but where is this XML file coming from? To answer that question, we need to introduce two new concepts; Handles and the Package Layout.

Handles

Each page request in Magento will generate several unique Handles. The Layoutview module can show you these Handles by using a URL something like

```
http://example.com/helloworld/index/index?showLayout=handles
```

You should see a list similar to the following (depending on your configuration)

1. `default`
2. `STORE_bare_us`
3. `THEME_frontend_default_default`
4. `helloworld_index_index`
5. `customer_logged_out`

Each of these is a Handle. Handles are set in a variety of places within the Magento system. The two we want to pay attention to are `default` and `helloworld_index_index`. The `default` Handle is present in **every** request into the Magento system. The `helloworld_index_index` Handle is created by combining the frontName (helloworld), Action Controller name (index), and Action Controller Action Method (index) into a single string. This means each possible method on an Action Controller has a Handle associated with it.

Remember that "index" is the Magento default for both Action Controllers and Action Methods, so the following request

```
http://example.com/helloworld/?showLayout=handles
```

Will also produce a Handle named `helloworld_index_index`

Package Layout

You can think of the Package Layout similar to the global config. It's a large XML file that contains **every possible layout configuration** for a particular Magento install. Let's take a look at it using the Layoutview module

```
http://example.com/helloworld/index/index?showLayout=package
```

This may take a while to load. If your browser is choking on the XML rendering, try the text format

```
http://example.com/helloworld/index/index?showLayout=package&showLayoutFormat
=text
```

You should see a very large XML file. This is the Package Layout. This XML file is created by combining the contents of all the XML layout files for the current theme (or package). For the default install, this is at

```
app/design/frontend/base/default/layout/
```

Behind the scenes there's an **<updates />** section of the global config that contains nodes with all the file names to load. Once the files listed in the config have been combined, Magento will merge in one last xml file, local.xml. This is the file where you're able to add your customizations to your Magento install.

Combining Handles and The Package Layout

So, if you look at the Package Layout, you'll see some familiar tags such as <block /> and <reference />, but they're all surrounded by tags that look like

```
<default />
<catalogsearch_advanced_index />
etc...
```

These are all Handle tags. The Layout for an individual request is generated by grabbing all the sections of the Package Layout that match any Handles for the request. So, in our example above, our layout is being generated by grabbing tags from the following sections

```
<default />
<STORE_bare_us />
<THEME_frontend_default_default />
<helloworld_index_index />
<customer_logged_out />
```

There's one additional tag you'll need to be aware of in the Package Layout. The <update /> tag allows you to include another Handle's tags. For example

```
<customer_account_index>
    <!-- ... -->
    <update handle="customer_account"/>
    <!-- ... -->
</customer_account_index>
```

Is saying that requests with a customer_account_index Handle should include <reference />s and <blocks />s from the <customer_account /> Handle.

Applying What We've Learned

OK, that's a lot of theory. Lets get back to what we did earlier. Knowing what we know now, adding

```
<layout version="0.1.0">
    <default>
        <reference name="root">
            <block type="page/html" name="root" output="toHtml" template="../
../../../../code/local/Magentotutorial/Helloworld/simple_page.phtml" />
        </reference>
    </default>
</layout>
```

to `local.xml` means we've overridden the "root" tag. with a different Block. By placing this in the <default /> Handle we've ensured that this override will happen for **every page request** in the system. That's probably not what we want.

If you go to any other page in your Magento site, you'll notice they're either blank white, or have the same red background that your hello world page does. Let's change your `local.xml` file so it only applies to the hello world page. We'll do this by changing default to use the full action name handle (`helloworld_index_index`).

```
<layout version="0.1.0">
    <helloworld_index_index>
        <reference name="root">
            <block type="page/html" name="root" output="toHtml" template="../
../../../../code/local/Magentotutorial/Helloworld/simple_page.phtml" />
        </reference>
    </helloworld_index_index>
</layout>
```

Clear your Magento cache, and the rest of your pages should be restored.

Right now this only applies to our index Action Method. Let's add it to the goodbye Action Method as well. In your Action Controller, modify the goodbye action so it looks like

```
public function goodbyeAction() {
    $this->loadLayout();
    $this->renderLayout();
}
```

If you load up the following URL, you'll notice you're still getting the default Magento layout.

```
http://example.com/helloworld/index/goodbye
```

We need to add a Handle for the full action name (`helloworld`*`index`*`goodbye`) to our local.xml file. Rather than specify a new <reference />, lets use the update tag to match the `helloworld_index_index` Handle.

```
<layout version="0.1.0">
    <!-- ... -->
    <helloworld_index_goodbye>
        <update handle="helloworld_index_index" />
    </helloworld_index_goodbye>
</layout>
```

Loading the following pages (after clearing your Magento cache) should now produce identical results.

```
http://example.com/helloworld/index/index
http://example.com/helloworld/index/goodbye
```

Starting Output and getChildHtml

In a standard configuration, output starts on the Block named root (because it has an output attribute). We've overridden root's Template with our own

```
template="../../../../../code/local/Magentotutorial/Helloworld/simple_page.phtml"
```

Templates are referenced from the root folder of the current theme. In this case, that's

```
app/design/frontend/base/default
```

so we need to climb five directories (../../../../../) then drill down to our custom page. Most Magento Templates are stored in

```
app/design/frontend/base/default/templates
```

Adding Content Blocks

A simple red page is pretty boring. Let's add some content to this page. Change your `<helloworld_index_index />` Handle in local.xml so it looks like the following

```
<helloworld_index_index>
    <reference name="root">
        <block type="page/html" name="root" template="../../../../../code/local/Magentotutorial/Helloworld/simple_page.phtml">
            <block type="customer/form_register" name="customer_form_register" template="customer/form/register.phtml"/>
```

```
        </block>
    </reference>
</helloworld_index_index>
```

We're adding a new Block nested within our root. This is a Block that's distributed with Magento, and will display a customer registration form. By nesting this Block within our root Block, we've made it available to be pulled into our `simple_page.html` Template. Next, we'll use the Block's getChildHtml method in our `simple_page.phtml` file. Edit `simple_page.html` so it looks like this

```
<body>
    <?php echo $this->getChildHtml('customer_form_register'); ?>
</body>
```

Clear your Magento cache and reload the page and you should see the customer registration form on your red background. Magento also has a Block named top.links. Let's try including that. Change your simple_page.html file so it reads

```
<body>
    <h1>Links</h1>
    <?php echo $this->getChildHtml('top.links'); <?php echo '?';?>>
</body>
```

When you reload the page, you'll notice that your <h1>Links</h1> title is rendering, but nothing is rendering for top.links. That's because we didn't add it to local.xml. The `getChildHtml` method can only include Blocks that are specified as sub-Blocks in the Layout. This allows Magento to only instantiate the Blocks it needs, and also allows you to set difference Templates for Blocks based on context.

Let's add the top.links Block to our local.xml

```
<helloworld_index_index>
    <reference name="root">
        <block type="page/html" name="root" template="../../../../../code/loc
al/Magentotutorial/Helloworld/simple_page.phtml">
            <block type="page/template_links" name="top.links"/>
            <block type="customer/form_register" name="customer_form_register
" template="customer/form/register.phtml"/>
        </block>
    </reference>
</helloworld_index_index>
```

Clear your cache and reload the page. You should now see the top.links module.

Wrapup

That covers Layout fundamentals. If you found it somewhat daunting, don't worry, you'll rarely need to work with layouts on such a fundamental level. Magento provides a number of pre-built layouts which can be modified and skinned to meet the needs of your store. Understanding how

the entire Layout system works can be a great help when you're trouble shooting Layout issues, or adding new functionality to an existing Magento system.

Magento for Developers: Part 5 - Magento Models and ORM Basics

The implementation of a "Models Tier" is a huge part of any MVC framework. It represents the data of your application, and most applications are useless without data. Magento Models play an even bigger role, as they typically contain the "Business Logic" that's often relegated to the Controller or Helper methods in other PHP MVC frameworks.

Traditional PHP MVC Models

If the definition of MVC is somewhat fuzzy, the definition of a Model is even fuzzier. Prior to the wide adoption of the MVC pattern by PHP developers, data access was usually raw SQL statements and/or a SQL abstraction layer. Developers would write queries and not think too much about what objects they were modeling.

In this day and age, raw SQL is mostly frowned upon, but many PHP frameworks are still SQL centric. Models will be objects that provide some layer of abstraction, but behind the scenes developers are still writing SQL and/or calling SQL like abstraction methods to read and write-down their data.

Other frameworks eschew SQL and take the Object Relational Mapping (ORM) approach. Here, a developer is dealing strictly with Objects. Properties are set, and when a save method is called on the Object, the data is automatically written to the database. Some ORMs will attempt to divine object properties from the database, others require the user to specify them in some way, (usually in an abstract data language such as YAML). One of the most famous and popular implementations of this approach is ActiveRecord.

This definition of ORM should suffice for now, but like everything Computer Science these days, the strict definition of ORM has blurred over the years. It's beyond the scope of this article to settle that dispute, but suffice it say we're generalizing a bit.

Magento Models

It should be no surprise that Magento takes the ORM approach. While the Zend Framework SQL abstractions are available, most of your data access will be via the built in Magento Models, and Models you build yourself. It should also come as no surprise that Magento has a highly flexible, highly abstract, concept of what a Model is.

Anatomy of a Magento Model

Most Magento Models can categorized in one of two ways. There's a basic, ActiveRecord-like/one-object-one-table Model, and there's also an Entity Attribute Value (EAV) Model. Each Model also gets a Model Collection. Collections are PHP objects used to hold a number of individual Magento Model instances. The Magento team has implemented the PHP Standard Library interfaces of IteratorAggregate and Countable to allow each Model type to have it's own collection type. If you're not familiar with the PHP Standard Library, think of Model Collections as arrays that also have methods attached.

Magento Models don't contain any code for connecting to the database. Instead, each Model uses two `modelResource` classes, (one read, one write), that are used to communicate with the database server (via read and write adapter objects). By decoupling the logical Model and the code that talks to the database, it's theoretically possible to write new resource classes for a different database platform while keeping your Models themselves untouched.

Creating a Basic Model

To begin, we're going to create a basic Magento Model. PHP MVC tradition insists we model a weblog post. The steps we'll need to take are

1. Crete a new "Weblog" module
2. Create a database table for our Model
3. Add Model information to the config for a Model named Blogpost
4. Add Model Resource information to the config for the Blogpost Model
5. Add A Read Adapter to the config for the Blogpost Model
6. Add A Write Adapter to the config for the Blogpost Model
7. Add a PHP class file for the Blogpost Model
8. Add a PHP class file for the Blogpost Model
9. Instantiate the Model

Create a Weblog Module

You should be an old hat at creating empty modules at this point, so we'll skip the details and assume you can create an empty module named Weblog. After you've done that, we'll setup a route for an index Action Controller with an action named "testModel". As always, the following examples assume a Package Name of "Magentotutorial".

In Magentotutorial/Weblog/etc/config.xml, setup the following route

```
<frontend>
    <routers>
        <weblog>
            <use>standard</use>
            <args>
                <module>Magentotutorial_Weblog</module>
                <frontName>weblog</frontName>
            </args>
        </weblog>
```

```
    </routers>
</frontend>
```

And then add the following Action Controller in

```
class Magentotutorial_Weblog_IndexController extends Mage_Core_Controller_Fro
nt_Action {
    public function testModelAction() {
        echo 'Setup!';
    }
}
```

at `Magentotutorial/Weblog/controllers/IndexController.php`. Clear your Magento cache and load the following URL to ensure everything's been setup correctly.

```
http://example.com/weblog/index/testModel
```

You should see the word "Setup" on a white background.

Creating the Database Table

Magento has a system for automatically creating and changing your database schemas, but for the time being we'll just manually create a table for our Model.

Using the command-line or your favorite MySQL GUI application, create a table with the following schema

```
CREATE TABLE `blog_posts` (
  `blogpost_id` int(11) NOT NULL auto_increment,
  `title` text,
  `post` text,
  `date` datetime default NULL,
  `timestamp` timestamp NOT NULL default CURRENT_TIMESTAMP,
  PRIMARY KEY  (`blogpost_id`)
)
```

And then populate it with some data

```
INSERT INTO `blog_posts` VALUES (1,'My New Title','This is a blog post','2010
-07-01 00:00:00','2010-07-02 23:12:30');
```

The Global Config and Creating The Model

There are five individual things we need to setup for a Model in our config.

1. Enabling Models in our Module
2. Enabling Model Resources in our Module
3. Add an "entity" to our Model Resource. For simple Models, this is the name of the table
4. Specifying a Read Adapter for our specific Model Resource

5. Specifying a Write Adapter for our specific Model Resource

When you instantiate a Model in Magento, you make a call like this

```
$model = Mage::getModel('weblog/blogpost');
```

The first part of the URI you pass into get Model is the Model Group Name. Because of the way Magento uses `__autoload` for classes, this also has to be the (lowercase) name of your module. The second part of the URI is the lowercase version of your Model name.

So, let's add the following XML to our module's config.xml.

```
<global>
    <!-- ... -->
    <models>
        <weblog>
            <class>Magentotutorial_Weblog_Model</class>
            <!--
            need to create our own resource, cant just
            use core_mysql4
            -->
            <resourceModel>weblog_mysql4</resourceModel>
        </weblog>
    </models>
    <!-- ... -->
</global>
```

The outer <weblog /> tag is your Group Name, which should match your module name. <class /> is the BASE name all Models in the weblog group will have. The <resourceModel /> tag indicates which Resource Model that weblog group Models should use. There's more on this below, but for now be content to know it's your Group Name, followed by a the literal string "mysql4".

So, we're not done yet, but let's see what happens if we clear our Magento cache and attempt to instantiate a blogpost Model. In your testModelAction method, use the following code

```
public function testModelAction() {
        $blogpost = Mage::getModel('weblog/blogpost');
        echo get_class($blogpost);
    }
```

and reload your page. You should see an exception that looks something like this (be sure you've turned on developer mode).

```
include(Magentotutorial/Weblog/Model/Blogpost.php) [function.include]: failed
 to open stream: No such file or directory
```

By attempting to retrieve a `weblog/blogpost` Model, you told Magento to instantiate a class with the name

```
Magentotutorial_Weblog_Model_Blogpost
```

Magento is trying to __autoload include this Model, but can't find the file. Let's create it! Create the following class at the following location

```
File: app/code/local/Magentotutorial/Weblog/Model/Blogpost.php
class Magentotutorial_Weblog_Model_Blogpost extends Mage_Core_Model_Abstract
{
    protected function _construct()
    {
        $this->_init('weblog/blogpost');
    }
}
```

Reload your page, and the exception should be replaced with the name of your class.

All basic Models should extend the `Mage_Core_Model_Abstract` class. This abstract class forces you to implement a single method named `_construct` (**NOTE:** this is not PHP's constructor `__construct`). This method should call the class's _init method with the same identifying URI you'll be using in the `Mage::getModel` method call.

The Global Config and Resources

So, we've setup our Model. Next, we need to setup our Model Resource. Model Resources contain the code that actually talks to our database. In the last section, we included the following in our config.

```
<resourceModel>weblog_mysql4</resourceModel>
```

The value in <resourceModel /> will be used to instantiate a Model Resource class. Although you'll never need to call it yourself, when any Model in the weblog group needs to talk to the database, Magento will make the following method call to get the Model resource

```
Mage::getResourceModel('weblog/blogpost');
```

Again, weblog is the Group Name, and blogpost is the Model. The `Mage::getResourceModel` method will use the `weblog/blogpost` URI to inspect the global config and pull out the value in <resourceModel> (in this case, `weblog_mysql4`). Then, a model class will be instantiated with the following URI

```
weblog_mysql4/blogpost
```

So, if you followed that all the way, what this means is, **resource models are configured in the same section of the XML config as normal Models**. This can be confusing to newcomers and old-hands alike.

So, with that in mind, let's configure our resource. In our <models> section add

```
<global>
    <!-- ... -->
    <models>
        <!-- ... -->
        <weblog_mysql4>
            <class>Magentotutorial_Weblog_Model_Mysql4</class>
        </weblog_mysql4>
    </models>
</global>
```

You're adding the <weblog_mysql4 /> tag, which is the value of the <resourceModel /> tag you just setup. The value of <class /> is the base name that all your resource modes will have, and should be named with the following format

```
Packagename_Modulename_Model_Mysql4
```

So, we have a configured resource, let's try loading up some Model data. Change your action to look like the following

```
public function testModelAction() {
    $params = $this->getRequest()->getParams();
    $blogpost = Mage::getModel('weblog/blogpost');
    echo("Loading the blogpost with an ID of ".$params['id']);
    $blogpost->load($params['id']);
    $data = $blogpost->getData();
    var_dump($data);
}
```

And then load the following URL in your browser (after clearing your Magento cache)

```
http://example.com/weblog/index/testModel/id/1
```

You should see an exception something like the following

```
Warning: include(Magentotutorial/Weblog/Model/Mysql4/Blogpost.php)
[function.include]: failed to open stream: No such file ....
```

As you've likely intuited, we need to add a resource class for our Model. **Every** Model has its own resource class. Add the following class at at the following location

```
File: app/code/local/Magentotutorial/Weblog/Model/Mysql4/Blogpost.php
class Magentotutorial_Weblog_Model_Mysql4_Blogpost extends Mage_Core_Model_My
sql4_Abstract{
    protected function _construct()
    {
        $this->_init('weblog/blogpost', 'blogpost_id');
    }
}
```

Again, the first parameter of the init method is the URL used to identify the **Model**. The second parameter is the database field that uniquely identifies any particular column. In most cases, this should be the primary key. Clear your cache, reload, and you should see

```
Loading the blogpost with an ID of 1
array
empty
```

So, we've gotten things to the point where there's no exception, but we're still not reading from the database. What gives?

Each Model group has a read adapter (for reading from the database) and a write adapter (for updating the database). Magento allows a Model to use the default adapter, or for developers to write their own. Either way, we need tell Magento about it. We'll be adding a new tag section named <resources /> to the <global /> section of our module config.

```
<global>
    <!-- ... -->
    <resources>
        <weblog_write>
            <connection>
                <use>core_write</use>
            </connection>
        </weblog_write>
        <weblog_read>
            <connection>
                <use>core_read</use>
            </connection>
        </weblog_read>
    </resources>
</global>
```

We're adding two sub-sections to <resources />. One for writing, and one for reading. The tag names (<weblog_write /> and <weblog_read />) are based on the Group Name we defined above ("weblog").

OK, **surely** with our adapter's in place we're ready. Let's clear our cache, reload the the page, and ...

```
Can't retrieve entity config: weblog/blogpost
```

Another exception! When we use the Model URI `weblog/blogpost`, we're telling Magento we want the Model Group `weblog`, and the `blogpost` *Entity*. In the context of simple Models that extend `Mage_Core_Model_Mysql4_Abstract`, an entity corresponds to a table. In this case, the table named `blog_post` that we created above. Let's add that entity to our XML config.

```
<models>
        <!-- ... --->
        <weblog_mysql4>
            <class>Magentotutorial_Weblog_Model_Mysql4</class>
            <entities>
```

```
                <blogpost>
                    <table>blog_posts</table>
                </blogpost>
            </entities>
        </weblog_mysql4>
    </models>
```

We've added a new <entities /> section to the resource Model section of our config. This, in turn, has a section named after our entity (<blogpost />) that specifies the name of the database table we want to use for this Model.

Clear your Magento cache, cross your fingers, reload the page and ...

```
Loading the blogpost with an ID of 2

Loading the blogpost with an ID of 1

array
  'blogpost_id' => string '1' (length=1)
  'title' => string 'My New Title' (length=12)
  'post' => string 'This is a blog post' (length=19)
  'date' => string '2009-07-01 00:00:00' (length=19)
  'timestamp' => string '2009-07-02 16:12:30' (length=19)
```

Eureka! We've managed to extract our data and, more importantly, completely configure a Magento Model.

Basic Model Operations

All Magento Models inherit from the the Varien_Object class. This class is part of the Magento system library and **not** part of any Magento core module. You can find this object at

```
lib/Varien/Object.php
```

Magento Models store their data in a protected _data property. The Varien_Object class gives us several methods we can use to extract this data. You've already seen `getData`, which will return an array of key/value pairs. This method can also be passed a string key to get a specific field.

```
$model->getData();
$model->getData('title');
```

There's also a getOrigData method, which will return the Model data as it was when the object was initially populated, (working with the protected _origData method).

```
$model->getOrigData();
$model->getOrigData('title');
```

The Varien_Object also implements some special methods via PHP's magic `__call` method. You can get, set, unset, or check for the existence of any property using a method that begins with the word get, set, unset or has and is followed by the camel cased name of a property.

```
$model->getBlogpostId();
$model->setBlogpostId(25);
$model->unsetBlogpostId();
if($model->hasBlogpostId()){...}
```

For this reason, you'll want to name all your database columns with lower case characters and use underscores to separate characters. More recent version of Magento have deprecated this syntax in favor of implementing the PHP ArrayAccess interface

```
$id = $model->['blogpost_id'];
$model->['blogpost_id'] = 25;
 //etc...
```

That said, you're likely to see both techniques used throughout the Magento code base, as well as third party extensions.

CRUD, the Magento Way

Magento Models support the basic Create, Read, Update, and Delete functionality of CRUD with `load`, `save`, and `delete` methods. You've already seen the load method in action. When passed a single parameter, the load method will return a record whose id field (set in the Model's resource) matches the passed in value.

```
$blogpost->load(1);
```

The save method will allow you to both INSERT a new Model into the database, or UPDATE an existing one. Add the following method to your Controller

```
public function createNewPostAction() {
    $blogpost = Mage::getModel('weblog/blogpost');
    $blogpost->setTitle('Code Post!');
    $blogpost->setPost('This post was created from code!');
    $blogpost->save();
    echo 'post created';
}
```

and then execute your Controller Action by loading the following URL

```
http://example.com/weblog/index/createNewPost
```

You should now see an additional saved post in you database table. Next, try the following to edit your post

```
public function editFirstPostAction() {
    $blogpost = Mage::getModel('weblog/blogpost');
```

```
    $blogpost->load(1);
    $blogpost->setTitle("The First post!");
    $blogpost->save();
    echo 'post edited';
}
```

And finally, you can delete your post using very similar syntax.

```
public function deleteFirstPostAction() {
    $blogpost = Mage::getModel('weblog/blogpost');
    $blogpost->load(1);
    $blogpost->delete();
    echo 'post removed';
}
```

Model Collections

So, having a single Model is useful, but sometimes we want to grab list of Models. Rather than returning a simple array of Models, each Magento Model type has a unique collection object associated with it. These objects implement the PHP IteratorAggregate and Countable interfaces, which means they can be passed to the `count` function, and used in `for each` constructs.

We'll cover Collections in full in a later article, but for now let's look at basic setup and usage. Add the following action method to your Controller, and load it in your browser.

```
public function showAllBlogPostsAction() {
    $posts = Mage::getModel('weblog/blogpost')->getCollection();
    foreach($posts as $blog_post){
        echo '<h3>'.$blog_post->getTitle().'</h3>';
        echo nl2br($blog_post->getPost());
    }
}
```

Load the action URL,

```
http://example.com/weblog/index/showAllBlogPosts
```

and you should see a (by now) familiar exception.

```
Warning: include(Magentotutorial/Weblog/Model/Mysql4/Blogpost/Collection.php)
[function.include]: failed to open stream
```

You're not surprised, are you? We need to add a PHP class file that defines our Blogpost collection. Every Model has a protected property named `_resourceCollectionName` that contains a URI that's used to identify our collection.

```
protected '_resourceCollectionName' => string 'weblog/blogpost_collection'
```

By default, this is the same URI that's used to identify our Resource Model, with the string "_collection" appended to the end. Magento considers Collections part of the Resource, so this URI is converted into the class name

```
Magentotutorial_Weblog_Model_Mysql4_Blogpost_Collection
```

Add the following PHP class at the following location

```
File:
app/code/local/Magentotutorial/Weblog/Model/Mysql4/Blogpost/Collection.php
class Magentotutorial_Weblog_Model_Mysql4_Blogpost_Collection extends Mage_Co
re_Model_Mysql4_Collection_Abstract {
    protected function _construct()
    {
            $this->_init('weblog/blogpost');
    }
}
```

Just as with our other classes, we need to init our Collection with the Model URI. (weblog/blogpost). Rerun your Controller Action, and you should see your post information.

Wrapup and a Quick Note on Core Magento Models

Congratulations, you've created and configured you first Magento Model. In a later article we'll take a look at Magento's Entity Attribute Value Models (EAV), which expand on what we've learned here.

Also, there's a little fib we had to tell you earlier. In this article we indicated that all basic Magento Models inherit from `Mage_Core_Model_Abstract`. This isn't 100% true. This Abstract method hasn't existed for Magento's entire life, and there are still many Models in the system that inherit directly from `Varien_Object`. While this shouldn't affect any Models you create, it is something you'll want to be aware of while working with Magento code.

Magento for Developers: Part 6 - Magento Setup Resources

On any fast paced software development project, the task of keeping the development and production databases in sync become a sticky wicket. Magento offers a system to create versioned resource migration scripts that can help your team deal with this often contentious part of the development process.

In the ORM article we created a model for a weblog post. At the time, we ran our `CREATE TABLE` statements directly against the database. This time, we'll create a Setup Resource for our module that will create the table for us. We'll also create an upgrade script for our module that will update an already installed module. The steps we'll need to take are

1. Add the Setup Resource to our config

2. Create our resource class file
3. Create our installer script
4. Create our upgrade script

Adding the Setup Resource

So, let's continue with the weblog module we created last time. In our <resources /> section, add the following

```
<resources>
    <!-- ... -->
    <weblog_setup>
        <setup>
            <module>Magentotutorial_Weblog</module>
            <class>Magentotutorial_Weblog_Model_Resource_Mysql4_Setup</class>
        </setup>
        <connection>
            <use>core_setup</use>
        </connection>
    </weblog_setup>
    <!-- ... -->
</resources>
```

The `<weblog_setup>` tag will be used to uniquely identify this Setup Resource. It's encouraged, but not necessary, that you use the `modelname_setup` naming convention. The `<module>Magentotutorial_Weblog</module>` tag block should contain the `Packagename_Modulename` of your module. Finally, `<class>Magentotutorial_Weblog_Model_Resource_Mysql4_Setup</class>` should contain the name of the class we'll be creating for our Setup Resource. For basic setup scripts it's not necessary to create a custom class, but by doing it now you'll give yourself more flexibility down the line.

After adding the above section to your config, clear your Magento cache and try to load any page of your Magento site. You'll see an exception something like

```
Fatal error: Class 'Magentotutorial_Weblog_Model_Resource_Mysql4_Setup' not
found in
```

Magento just tried to instantiate the class you specified in your config, but couldn't find it. You'll want to create the following file, with the following contents.

File: app/code/local/Magentotutorial/Weblog/Model/Resource/Mysql4/Setup.php

```
class Magentotutorial_Weblog_Model_Resource_Mysql4_Setup extends Mage_Core_Mo
del_Resource_Setup {
}
```

Now, reload any page of your Magento site. The exception should be gone, and your page should load as expected.

Creating our Installer Script

Next, we'll want to create our installer script. This is the script that will contain any `CREATE TABLE` or other SQL code that needs to be run to initialize our module.

First, take a look at your config.xml file

```
<modules>
    <Magentotutorial_Weblog>
        <version>0.1.0</version>
    </Magentotutorial_Weblog>
</modules>
```

This section is required in all config.xml files, and identifies the module as well as the its version number. Your installer script's name will be based on this version number. The following assumes the current version of your module is 0.1.0.

Create the following file at the following location

File: app/code/local/Magentotutorial/Weblog/sql/weblog_setup/mysql4-install-0.1.0.php

```
echo 'Running This Upgrade: '.get_class($this)."\n <br /> \n";
die("Exit for now");
```

The `weblog_setup` portion of the path should match the tag you created in your config.xml file (`<weblog_setup />`). The `0.1.0` portion of the filename should match the starting version of your module. Clear your Magento cache and reload any page in your Magento site and you should see something like

```
Running This Upgrade: Magentotutorial_Weblog_Model_Resource_Mysql4_Setup
Exit for now
 ...
```

Which means your update script ran. Eventually we'll put our SQL update scripts here, but for now we're going to concentrate on the setup mechanism itself. Remove the "die" statement from your script so it looks like the following

```
echo 'Running This Upgrade: '.get_class($this)."\n <br /> \n";
```

Reload your page. You should see your upgrade message displayed at the top of the page. Reload again, and your page should be displayed as normal.

Resource Versions

Magento's Setup Resources allow you to simply drop your install scripts (and upgrade scripts, which we'll get to in a bit) onto the server, and have the system automatically run them. This allows you to have all your database migrations scripts stored in the system in a consistent format.

Using your favorite database client, take a look at the the core_setup table

```
mysql> select * from core_resource;
+--------------------------+---------+
| code                     | version |
+--------------------------+---------+
| adminnotification_setup | 1.0.0   |
| admin_setup              | 0.7.1   |
| amazonpayments_setup     | 0.1.2   |
| api_setup                | 0.8.1   |
| backup_setup             | 0.7.0   |
| bundle_setup             | 0.1.7   |
| catalogindex_setup       | 0.7.10  |
| cataloginventory_setup   | 0.7.5   |
| catalogrule_setup        | 0.7.7   |
| catalogsearch_setup      | 0.7.6   |
| catalog_setup            | 0.7.69  |
| checkout_setup           | 0.9.3   |
| chronopay_setup          | 0.1.0   |
| cms_setup                | 0.7.8   |
| compiler_setup           | 0.1.0   |
| contacts_setup           | 0.8.0   |
| core_setup               | 0.8.13  |
| cron_setup               | 0.7.1   |
| customer_setup           | 0.8.11  |
| cybermut_setup           | 0.1.0   |
| cybersource_setup        | 0.7.0   |
| dataflow_setup           | 0.7.4   |
| directory_setup          | 0.8.5   |
| downloadable_setup       | 0.1.14  |
| eav_setup                | 0.7.13  |
| eway_setup               | 0.1.0   |
| flo2cash_setup           | 0.1.1   |
| giftmessage_setup        | 0.7.2   |
| googleanalytics_setup    | 0.1.0   |
| googlebase_setup         | 0.1.1   |
| googlecheckout_setup     | 0.7.3   |
| googleoptimizer_setup    | 0.1.2   |
| ideal_setup              | 0.1.0   |
| log_setup                | 0.7.6   |
| newsletter_setup         | 0.8.0   |
| oscommerce_setup         | 0.8.10  |
| paybox_setup             | 0.1.3   |
| paygate_setup            | 0.7.0   |
| payment_setup            | 0.7.0   |
| paypaluk_setup           | 0.7.0   |
| paypal_setup             | 0.7.2   |
| poll_setup               | 0.7.2   |
| productalert_setup       | 0.7.2   |
| protx_setup              | 0.1.0   |
```

```
| rating_setup            | 0.7.2   |
| reports_setup           | 0.7.7   |
| review_setup            | 0.7.4   |
| salesrule_setup         | 0.7.7   |
| sales_setup             | 0.9.38  |
| sendfriend_setup        | 0.7.2   |
| shipping_setup          | 0.7.0   |
| sitemap_setup           | 0.7.2   |
| strikeiron_setup        | 0.9.1   |
| tag_setup               | 0.7.2   |
| tax_setup               | 0.7.8   |
| usa_setup               | 0.7.0   |
| weblog_setup            | 0.1.0   |
| weee_setup              | 0.13    |
| wishlist_setup          | 0.7.4   |
+-------------------------+---------+
59 rows in set (0.00 sec)
```

This table contains a list of all the installed modules, along with the installed version number. You can see our module near the end

```
| weblog_setup            | 0.1.0   |
```

This is how Magento knows not to re-run your script on the second, and on all successive, page loads. The `weblog_setup` is already installed, so it won't be updated. If you want to re-run your installer script (useful when you're developing), just delete the row for your module from this table. Let's do that now, and actually add the SQL to create our table. So first, run the following SQL.

```
DELETE from core_resource where code = 'weblog_setup';
```

We'll also want to drop the table we manually created in the ORM article.

```
DROP TABLE blog_posts;
```

Then, add the following code to your setup script.

```
$installer = $this;
$installer->startSetup();
$installer->run("
    CREATE TABLE `{$installer->getTable('weblog/blogpost')}` (
      `blogpost_id` int(11) NOT NULL auto_increment,
      `title` text,
      `post` text,
      `date` datetime default NULL,
      `timestamp` timestamp NOT NULL default CURRENT_TIMESTAMP,
      PRIMARY KEY  (`blogpost_id`)
    ) ENGINE=InnoDB DEFAULT CHARSET=utf8;

    INSERT INTO `{$installer->getTable('weblog/blogpost')}` VALUES (1,'My New
Title','This is a blog post','2009-07-01 00:00:00','2009-07-02 23:12:30');
");
$installer->endSetup();
```

Clear your Magento cache and reload any page in the system. You should have a new blog_posts table with a single row.

Anatomy of a Setup Script

So, let's go over the script line-by-line. First, there's this (or is that $this?)

```
$installer = $this;
```

Each installer script is run from the context of a Setup Resource class, the class you created above. That means any reference to `$this` from within the script will be a reference to an object instantiated from this class. While not necessary, most setup scripts in the core modules will alias `$this` to a variable called installer, which is what we've done here. While not necessary, it is the convention and it's always best to follow the convention unless you have a good reason for breaking it.

Next, you'll see our queries are bookended by the following two method calls.

```
$installer->startSetup();
//...
$installer->endSetup();
```

If you take a look at the `Mage_Core_Model_Resource_Setup` class in `app/code/core/Mage/Core/Model/Resource/Setup.php` (which your setup class inherits from) you can see that these methods do some basic SQL setup

```
public function startSetup()
    {
        $this->_conn->multi_query("SET SQL_MODE='';
SET @OLD_FOREIGN_KEY_CHECKS=@@FOREIGN_KEY_CHECKS, FOREIGN_KEY_CHECKS=0;
SET @OLD_SQL_MODE=@@SQL_MODE, SQL_MODE='NO_AUTO_VALUE_ON_ZERO';
");

        return $this;
    }

    public function endSetup()
    {
        $this->_conn->multi_query("
SET SQL_MODE=IFNULL(@OLD_SQL_MODE,'');
SET FOREIGN_KEY_CHECKS=IFNULL(@OLD_FOREIGN_KEY_CHECKS,0);
");
        return $this;
    }
```

Finally, there's the call to the `run` method

```
$installer->run(...);
```

which accepts a string containing the SQL needed to setup your database table(s). You may specify any number of queries, separated by a semi-colon. You also probably noticed the following

```
$installer->getTable('weblog/blogpost')
```

The `getTable` method allows you to pass in a Magento Model URI and get its table name. While not necessary, using this method ensure that your script will continue to run, even if someone changes the name of their table in the config file. The `Mage_Core_Model_Resource_Setup` class contains many useful helper methods like this. The best way to become familiar with everything that's possible is to study the installer scripts used by the core Magento modules.

Module Upgrades

So, that's how you create a script that will setup your initial database tables, but what if you want to alter the structure of an existing module? Magento's Setup Resources support a simple versioning scheme that will let you automatically run scripts to **upgrade** your modules.

Once Magento runs an *installer* script for a module, it will **never run another installer for that module again** (short of manually deleting the reference in the `core_resource` table). Instead, you'll need to create an upgrade script. Upgrade scripts are very similar to installer scripts, with a few key differences.

To get started, we'll create a script at the following location, with the following contents

File: app/code/local/Magentotutorial/Weblog/sql/weblog_setup/mysql4-upgrade-0.1.0-0.2.0.php:

```
echo 'Testing our upgrade script (mysql4-upgrade-0.1.0-
0.2.0.php) and halting execution to avoid updating the system version number
<br />';
die();
```

Upgrade scripts are placed in the same folder as your installer script, but named slightly differently. First, and most obviously, the file name contains the word upgrade. Secondly, you'll notice there are **two** version numbers, separated by a "-". The first (`0.1.0`) is the module version that we're upgrading **from**. The second (`0.2.0`) is the module version we're upgrading **to**.

If we cleared our Magento cache and reloaded a page, our script wouldn't run. We need to update the the version number in our module's config.xml file to trigger the upgrade

```
<modules>
    <Magentotutorial_Weblog>
        <version>0.2.0</version>
    </Magentotutorial_Weblog>
</modules>
```

With the new version number in place, we'll need to clear our Magento cache and load any page in our Magento site. You should now see output from your upgrade script.

Before we continue and actually implement the upgrade script, there's one important piece of behavior you'll want to be aware of. Create another upgrade file at the following location with the following contents.

File: app/code/local/Magentotutorial/Weblog/sql/weblog_setup/mysql4-upgrade-0.1.0-0.1.5.php:

```
echo 'Testing our upgrade script (mysql4-upgrade-0.1.0-
0.1.5.php) and NOT halting execution <br />';
```

If you reload a page, you'll notice you see BOTH messages. When Magento notices the version number of a module has changed, it will run through **all** the setup scripts needed to bring that version up to date. Although we never really created a version 0.1.5 of the Weblog module, Magento sees the upgrade script, and will attempt to run it. Scripts will be run in order from lowest to highest. If you take a peek at the `core_resource` table,

```
mysql> select * from core_resource where code = 'weblog_setup';
+--------------+---------+
| code         | version |
+--------------+---------+
| weblog_setup | 0.1.5   |
+--------------+---------+
1 row in set (0.00 sec)
```

you'll notice Magento considers the version number to be 1.5. That's because we completed executing the 1.0 to 1.5 upgrade, but did not complete execution of the 1.0 to 2.0 upgrade.

So, with all that out of the way, writing our actual upgrade script is identical to writing an installer script. Let's change the `0.1.0-0.2.0` script to read

```
$installer = $this;
$installer->startSetup();
$installer->run("
    ALTER TABLE `{$installer->getTable('weblog/blogpost')}`
    CHANGE post post text not null;
");
$installer->endSetup();
die("You'll see why this is here in a second");
```

Try refreshing a page in your Magento site and ... nothing. The upgrade script didn't run. The post field in our table still allows null values, and more importantly, the call to `die()` did not halt execution. Here's what happened

1. The weblog_setup resource was at version `0.1.0`
2. We upgraded our module to version `0.2.0`
3. Magento saw the upgraded module, and saw there were two upgrade scripts to run; `0.1.0` to `0.1.5` and `0.1.0` to `0.2.0`

4. Magento queued up both scripts to run
5. Magento ran the `0.1.0` to `0.1.5` script
6. The weblog_setup resource is now at version `0.1.5`
7. Magento ran the `0.1.0` to `0.2.0` script, execution was halted
8. On the next page load, Magento saw weblog_setup at version `0.1.5` and did not see any upgrade scripts to run since both scripts indicated they should be run **from** `0.1.0`

The correct way to achieve what we wanted would have been to name our scripts as follows

```
mysql4-upgrade-0.1.0-0.1.5.php #This goes from 0.1.0 to 0.1.5
mysql4-upgrade-0.1.5-0.2.0.php #This goes 0.1.5 to 0.2.0
```

Magento is smart enough to run both scripts on a single page load. You can go back in time and give this a try by updating the `core_resource` table

```
update core_resource set version = '0.1.0' where code = 'weblog_setup';
...
```

It's one of the odd quirks of the Magento system that the updates will run as previously configured. This means you'll want to be careful with multiple developers adding update scripts to the system. You'll either want a build-meister/deployment-manager type in charge of the upgrade scripts or (heaven forbid) developers will need to talk to one another.

Wrap-up

You should now know how to use Magento Setup Resources to create versioned database migration scripts, as well as understand the scripts provided in the core modules. Beyond having a standard way for developers to write migration scripts, Setup Resources become much more important when creating and modifying Entity Attribute Value models.

Magento for Developers: Part 7 - Advanced ORM - Entity Attribute Value

In the first ORM article we told you there were two kinds of Models in Magento. Regular, or "simple" Models, and Entity Attribute Value (or EAV) Models. We also told you this was a bit of a fib. Here's where we come clean.

ALL Magento Models inherit from the `Mage_Core_Model_Abstract` / `Varien_Object` chain. What makes something either a simple Model or an EAV Model is its **Model Resource**. While all resources extend the base `Mage_Core_Model_Resource_Abstract` class, simple Models have a resource that inherits from `Mage_Core_Model_Mysql4_Abstract`, and EAV Models have a resource that inherits from `Mage_Eav_Model_Entity_Abstract`

If you think about it, this makes sense. As the end-programmer-user of the system you want a set of methods you can use to talk to and manipulate your Models. You don't care what the back-end storage looks like, you just want to get properties and invoke methods that trigger business rules.

What is EAV

Wikipedia defines EAV as

Entity-Attribute-Value model (EAV), also known as object-attribute-value model and open schema is a data model that is used in circumstances where the number of attributes (properties, parameters) that can be used to describe a thing (an "entity" or "object") is potentially very vast, but the number that will actually apply to a given entity is relatively modest. In mathematics, this model is known as a sparse matrix.

Another metaphor that helps me wrap my head around it is "EAV brings some aspects of normalization to the database **table schema**". In a traditional database, tables have a fixed number of columns

```
+------------------+
| products         |
+------------------+
| product_id       |
| name             |
| price            |
| etc..            |
+------------------+
```

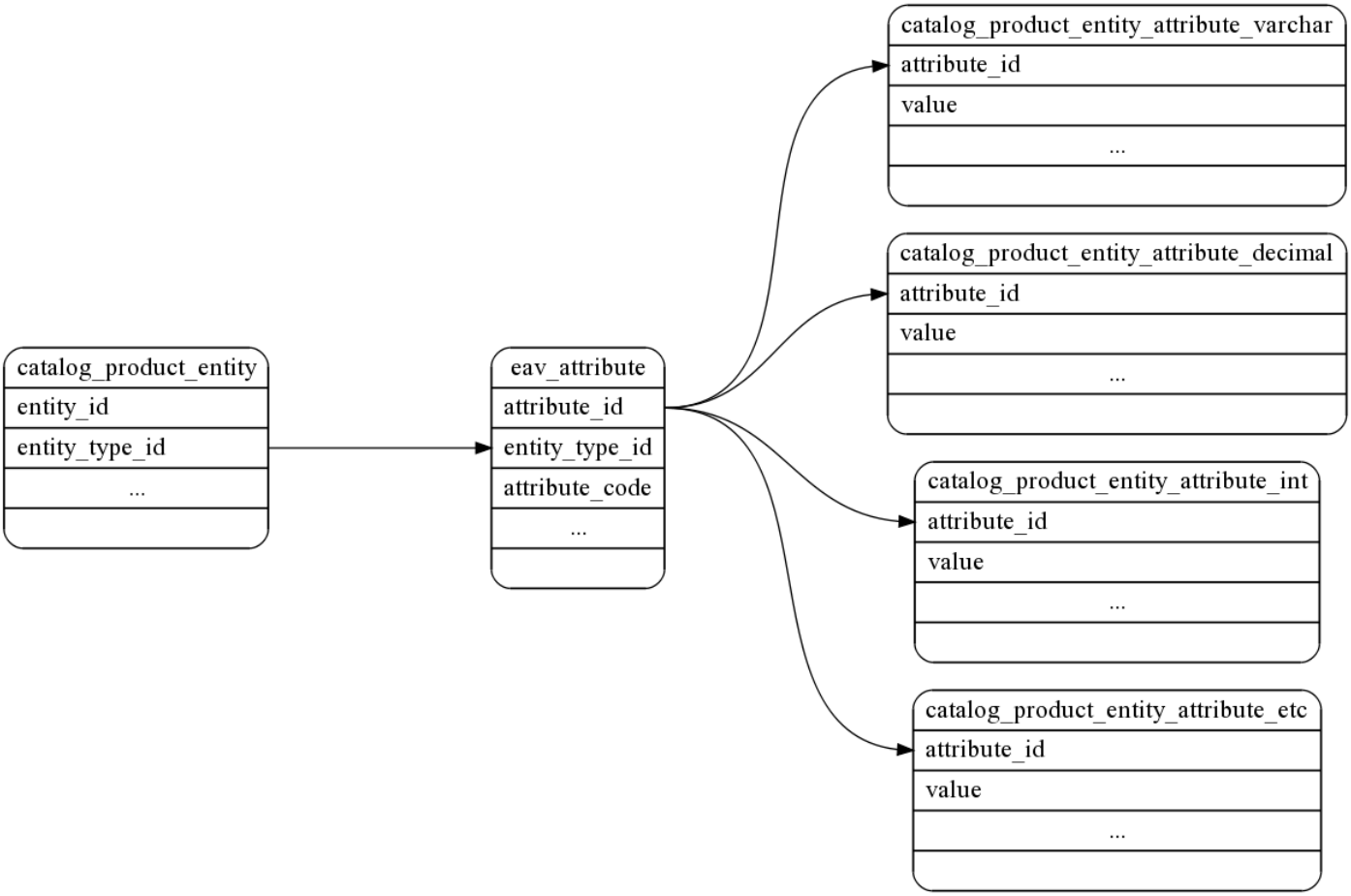
```
+------------+----------------+------------------+---------+
| product_id | name           | price            | etc...  |
+------------+----------------+------------------+---------+
| 1          | Widget A       | 11.34            | etc...  |
+------------+----------------+------------------+---------+
| 2          | Dongle B       | 6.34             | etc...  |
+------------+----------------+------------------+---------+
```

Every product has a name, every product has a price, etc.

In an EAV Model, each "entity" (product) being modeled has a **different** set of attributes. EAV makes a lot of sense for a generic eCommerce solution. A store that sells laptops (which have a CPU speed, color, ram amount, etc) is going to have a different set of needs than a store that sells yarn (yarn has a color, but no CPU speed, etc.). Even within our hypothetical yarn store, some products will have length (balls of yarn), and others will have diameter (knitting needles).

There aren't many open source or commercial databases that use EAV by default. There are none that are available on a wide variety of web hosting platforms. Because of that, the Magento engineers have built an EAV system out of PHP objects that use MySQL as a data-store. In other words, they've built an EAV database system **on top of** a traditional relational database.

In practice this means any Model that uses an EAV resource has its attributes spread out over a number of MySQL tables.



The above diagram is a rough layout of the database tables Magento consults when it looks up an EAV record for the `catalog_product` entity. Each individual product has a row in `catalog_product_entity`. All the available attributes in the **entire** system (not just for products) are stored in `eav_attribute`, and the actual attribute values are stored in tables with names like `catalog_product_entity_attribute_varchar`, `catalog_product_entity_attribute_decimal`, `catalog_product_entity_attribute_etc`.

Beyond the mental flexibility an EAV system gives you, there's also the practical benefit of avoiding ALTER TABLE statements. When you add a new attribute for your products, a new row is inserted into `eav_attribute`. In a traditional relational database/single-table system, you'd need to ALTER the actual database structure, which can be a time consuming/risky proposition for tables with large data-sets.

The downside is there's no one single simple SQL query you can use to get at all your product data. Several single SQL queries or one large join need to be made.

Implementing EAV

That's EAV in a nutshell. The rest of this articles is a run-through of what's needed to create a new EAV Model in Magento. It's the hairiest thing you'll read about Magento and it's something that 95% of people working with the system will never need to do. However, understanding what it takes to build an EAV Model Resource will help you understand what's going on with the EAV Resources that Magento uses.

Because the EAV information is so dense, we're going to assume you've studied up and are already very familiar with Magento's MVC and grouped class name features. We'll help you along the way, but training wheels are off.

Weblog, EAV Style

We're going to create another Model for a weblog post, but this time using an EAV Resource. To start with, setup and create a new module which responds at the the following URL

```
http://example.com/complexworld
```

If you're unsure how to do this, be sure you've mastered the concepts in the previous tutorials.

Next, we'll create a new Model named Weblogeav. Remember, it's the **Resource** that's considered EAV. We design and configure our Model the exact same way, so let's configure a Model similar to one we created in the first ORM article.

```
<global>
    <!-- ... -->
    <models>
        <!-- ... -->
        <complexworld>
            <class>Magentotutorial_Complexworld_Model</class>
            <resourceModel>complexworld_resource_eav_mysql4</resourceModel>
        </complexworld>
        <!-- ... -->
    </models>
    <!-- ... -->
</global>
```

You'll notice one difference to setting up a regular Model is the `<resourceModel/>` name looks a bit more complex (`weblog_resource_eav_mysql4`).

We'll still need to let Magento know about this resource. Similar to basic Models, EAV Resources are configured in the same `<model/>` node with everything else.

```
<global>
    <!-- ... -->
    <models>
        <!-- ... -->
        <complexworld_resource_eav_mysql4>
            <class>Magentotutorial_Complexworld_Model_Resource_Eav_Mysql4</cl
ass>
```

```
            <entities>
                <eavblogpost>
                    <table>eavblog_posts</table>
                </eavblogpost>
            </entities>
        </complexworld_resource_eav_mysql4>
        <!-- ... -->
    </models>
    <!-- ... -->
</global>
```

Again, so far this is setup similar to our regular Model Resource. We provide a `<class/>` that configures a PHP class, as well as an `<entities/>` section that will let Magento know the base table for an individual Model we want to create. The `<eavblogpost/>` tag is the name of the specific Model we want to create, and its inner `<table/>` tag specifies the base table this Model will use (more on this later).

We're also going to need `<resources/>`. Again, this is identical to the setup of a regular Model. The resources are the classes that Magento will use to interact with the database back-end.

```
<global>
    <!-- ... -->
    <resources>
        <complexworld_write>
            <connection>
                <use>core_write</use>
            </connection>
        </complexworld_write>
        <complexworld_read>
            <connection>
                <use>core_read</use>
            </connection>
        </complexworld_read>
    </resources>
    <!-- ... -->
</global>
```

Where Does That File Go?

Until wide adoption of PHP 5.3 and namespaces, one of the trickier (and tedious) parts of Magento will remain remembering how `<classname/>`s relate to file paths, and then ensuring you create the correctly named directory structure and class files. After configuring any `<classname/>`s or URIs, you may find it useful to attempt to instantiate an instance of the class in a controller **without** first creating the class files. This way PHP will throw an exception telling me it can't find a file, along with the file location. Give the following a try in your Index Controller.

```
public function indexAction() {
    $weblog2 = Mage::getModel('complexworld/eavblogpost');
    $weblog2->load(1);
```

```
    var_dump($weblog2);
}
```

As predicted, a warning should be thrown

```
Warning: include(Magentotutorial/Complexworld/Model/Eavblogpost.php) [functio
n.include]: failed to open stream: No such file or directory  in /Users/usern
ame/Sites/magento.dev/lib/Varien/Autoload.php on line 93
```

In addition to telling us the path where we'll need to define the new resource class this also serves as a configuration check. If we'd been warned with the following

```
Warning: include(Mage/Complexworld/Model/Eavblogpost.php) [function.include]:
 failed to open stream: No such file or directory  in /Users/username/Sites/m
agento.dev/lib/Varien/Autoload.php on line 93
```

we'd know our Model was misconfigured, as Magento was looking for the Model in `code/core/Mage` instead of `code/local/Magentotutorial`.

So, lets create our Model class

File: app/code/local/Magentotutorial/Complexworld/Model/Eavblogpost.php:

```
class Magentotutorial_Complexworld_Model_Eavblogpost extends Mage_Core_Model_
Abstract {
    protected function _construct()
    {
        $this->_init('complexworld/eavblogpost');
    }
}
```

Remember, the Model itself is resource independent. A regular Model and an EAV Model both extend from the same class. It's the resource that makes them different.

Clear your Magento cache, reload your page, and you should see a **new** warning.

```
Warning: include(Magentotutorial/Complexworld/Model/Resource/Eav/Mysql4/Eavbl
ogpost.php)
```

As expected, we need to create a class for our Model's resource. Let's do it!

File:
app/code/local/Magentotutorial/Complexworld/Model/Resource/Eav/Mysql4/Eavblogpost.php:

```
class Magentotutorial_Complexworld_Model_Resource_Eav_Mysql4_Eavblogpost exte
nds Mage_Eav_Model_Entity_Abstract
{
    public function _construct()
    {
        $resource = Mage::getSingleton('core/resource');
        $this->setType('complexworld_eavblogpost');
```

```
            $this->setConnection(
                  $resource->getConnection('complexworld_read'),
                  $resource->getConnection('complexworld_write')
              );
    }
}
```

So, already we're seeing a few differences between a simple Model Resource and an EAV Model Resource. First off, we're extending the `Mage_Eav_Model_Entity_Abstract` class. While `Mage_Eav_Model_Entity_Abstract` uses the same `_construct` concept as a regular Model Resource, there's no `_init` method. Instead, we need to handle the init ourselves. This means telling the resource what connection-resources it should use, and passing a unique identifier into the `setType` method of our object.

Another difference in `Mage_Eav_Model_Entity_Abstract` is `_construct` is **not** an abstract method, primarily for reasons of backwards compatability with older versions of the system.

So, with that, let's clear the Magento cache and reload the page. You should see a new exception which reads

```
Invalid entity_type specified: complexworld_eavblogpost
```

Magento is complaining that it can't find a entity_type named `complexworld_eavblogpost`. This is the value you set above

```
$this->setType('complexworld_eavblogpost');
```

Every entity has a type. Types will, among other things, let the EAV system know which attributes a Model uses, and allow the system to link to tables that store the values for attributes. We'll need to let Magento know that we're adding a new entity type. Take a look in the MySQL table named `eav_entity_type`.

```
mysql> select * from eav_entity_type\G
*************************** 1. row ***************************
          entity_type_id: 1
        entity_type_code: customer
            entity_model: customer/customer
         attribute_model:
            entity_table: customer/entity
      value_table_prefix:
         entity_id_field:
         is_data_sharing: 1
        data_sharing_key: default
default_attribute_set_id: 1
         increment_model: eav/entity_increment_numeric
     increment_per_store: 0
    increment_pad_length: 8
      increment_pad_char: 0
*************************** 2. row ***************************
          entity_type_id: 2
        entity_type_code: customer_address
```

```
            entity_model: customer/customer_address
          attribute_model:
            entity_table: customer/address_entity
       value_table_prefix:
          entity_id_field:
          is_data_sharing: 1
        data_sharing_key: default
default_attribute_set_id: 2
          increment_model:
     increment_per_store: 0
    increment_pad_length: 8
      increment_pad_char: 0
```

This table contains a list of all the entity_types in the system. The unique identifier `complexworld_eavblogpost` corresponds to the `entity_type_code` column.

Systems and Applications

This illustrates the single most important Magento concept, one that many people struggle to learn.

Consider the computer in front of you. The OS (Mac OS X, Windows, Linux, etc.) is the software system. Your web browser (Firefox, Safari, IE, Opera) is the application. Magento **is a system** first, and an application second. You build eCommerce applications using the Magneto system. What gets confusing is, there's a lot of places in Magento where the system code is exposed in a really raw form to the application code. The EAV system configuration living in the same database as your store's data is an example of this.

If you're going to get deep into Magento, you need to treat it like it's an old Type 650 machine. That is to say, it's the kind of thing you can't effectively program applications in unless unless you have a deep understanding of the system itself.

Creating a Setup Resource

So, it's theoretically possible to manually insert the rows you'll need into the Magento database to get your Model working, but it's not recommended. Fortunately, Magento provides a specialized Setup Resource that provides a number of helper method that will automatically create the needed records to get the system up and running.

So, for starters, configure the Setup Resource like you would any other.

```
<resources>
      <!-- ... -->
        <complexworld_setup>
            <setup>
                <module>Magentotutorial_Complexworld</module>
                <class>Magentotutorial_Complexworld_Entity_Setup</class>
            </setup>
            <connection>
```

```
            <use>core_setup</use>
        </connection>
    </complexworld_setup>
  <!-- ... -->
</resources>
```

Next, create its class file.

File: app/code/local/Magentotutorial/Complexworld/Entity/Setup.php:

```
class Magentotutorial_Complexworld_Entity_Setup extends Mage_Eav_Model_Entity
_Setup {
}
```

Take note that we're extending from `Mage_Eav_Model_Entity_Setup` rather than `Mage_Core_Model_Resource_Setup`.

Finally, we'll set up our installer script. If you're not familiar with the naming conventions here, you'll want to review the setup resource tutorial on Setup Resources.

File: app/code/local/Magentotutorial/Complexworld/sql/complexworld_setup/mysql4-install-0.1.0.php:

```
<?php echo '<?php';?>
$installer = $this;
throw new Exception("This is an exception to stop the installer from completi
ng");
```

Clear your Magento Cache, reload you page, and the above exception should be thrown, meaning you've correctly configured your Setup Resource.

NOTE: We'll be building up our install script piece by piece. If you've read the previous tutorial, you'll know you need to remove the setup's row from the `core_resource` table and clear your cache to make an installer script re-run. For the remainder of this tutorial, please remember that anytime we add or remove an item from our installer and re-run it, you'll need to remove this row from the database and clear your Magento cache. Normally you would create this file and run it once, a tutorial is something of an edge case.

Adding the Entity Type

To begin, add the following to your Setup Resource installer script, and then run the script by loading any page (after removing the above exception)

```
$installer = $this;
$installer->addEntityType('complexworld_eavblogpost',Array(
//entity_mode is the URL you'd pass into a Mage::getModel() call
'entity_model'          =>'complexworld/eavblogpost',
//blank for now
'attribute_model'       =>'',
```

```
//table refers to the resource URI complexworld/eavblogpost
//<complexworld_resource_eav_mysql4>...<eavblogpost><table>eavblog_posts</tab
le>
'table'         =>'complexworld/eavblogpost',
//blank for now, but can also be eav/entity_increment_numeric
'increment_model'       =>'',
//appears that this needs to be/can be above "1" if we're using eav/entity_in
crement_numeric
'increment_per_store'   =>'0'
));
```

We're calling the `addEntityType` method on our installer object. This method allows us to pass in the entity type (`complexworld_eavblogpost`) along with a list of parameters to set its default values. If you've run this script, you'll notice new rows in the `eav_attribute_group`, `eav_attribute_set`, and `eav_entity_type` tables.

So, with that in place, if we reload our complexworld page, we'll get a new error.

```
SQLSTATE[42S02]: Base table or view not found: 1146 Table 'magento.eavblog_po
sts' doesnt exist
```

Creating the Data Tables

So, we've told Magento about our new entity type. Next, we need to add the mysql tables that will be used to store all the entity values, as well as configure the system so it knows about these tables.

If you've spent any amount of time looking at the stock Magento installer files, you've seen a lot of manual SQL being used to create tables for EAV Models. Fortunately for us, this is no longer necessary. Our Setup Resource has a method named `createEntityTables` which will automatically setup the tables we need, as well as add some configuration rows to the system. Let's add the following line to our setuup resource.

```
$installer->createEntityTables(
$this->getTable('complexworld/eavblogpost')
);
```

The `createEntityTables` method accepts two parameters. The first is the base table name, the second is a list of options. We're using the Setup Resource's `getTable` method to pull the table name from our config. If you've been following along, you know this should resolve to the string `eavblog_posts`. We've omitted the second parameter which is an array of options you'll only need to used it for advanced situations that are beyond the scope of this tutorial.

After running the above script, you should have the following new tables in your database

```
eavblog_posts
eavblog_posts_datetime
eavblog_posts_decimal
eavblog_posts_int
```

```
eavblog_posts_text
eavblog_posts_varchar
```

You'll also have an additional row in the `eav_attribute_set` table

```
mysql> select * from eav_attribute_set order by attribute_set_id DESC LIMIT 1
\G
*************************** 1. row ***************************
  attribute_set_id: 65
    entity_type_id: 37
attribute_set_name: Default
        sort_order: 6
```

So, let's go back to our page and reload.

```
http://example.com/complexworld
```

Success! You should see no errors or warnings, and and a dumped `Magentotutorial_Complexworld_Model_Eavblogpost` --- with no data.

Adding Attributes

The last step we need to take in our Setup Resource is telling Magento what attributes we want our Model to have. This would be equivalent to adding new columns in a single database table setup. Again, the Setup Resource will help us. The two methods we're interested in are `installEntities` and `getDefaultEntites`.

The naming here can be a little confusing. After all, we just ran some code that told Magento about our entities, but now we need to do it again?

The code from the previous section was simply telling Magento about a **type** of entity that we **may** add to the system. These next bits of code are what will actually add a single entity of your type to the system. If you wanted to, I'm pretty sure you could create multiple entities of the same type. You can also get tattoos on the inside of your eyelids. If you're not sure if either is for you, they're not.

So, let's add the following method to our Setup Resource.

```
class Magentotutorial_Complexworld_Entity_Setup extends Mage_Eav_Model_Entity
_Setup {
    public function getDefaultEntities()
    {
        die("Calling ".__METHOD__);
    }
}
```

And then add the following call to the end of our setup script.

```
$installer->installEntities();
```

Reload your page and you should see the die/exit message specified above.

```
Calling Magentotutorial_Complexworld_Entity_Setup::getDefaultEntities
```

You **may** see an exception something like the following

```
[message:protected] => SQLSTATE[23000]: Integrity constraint violation: 1217
Cannot delete or update a parent row: a foreign key constraint fails
```

If that's the case, it's because you're re-running your setup script and calling the `createEntityTables` method again. Magento generates SQL statements for the `DROP`ing and `CREATE`ing your tables. Unfortunately, it doesn't take into account the `FOREIGN KEY` relationships, and tries to `DROP/CREATE` the primary entity table first.

For the purposes of this tutorial, just comment out the line(s) calling `createEntityTables`. Again, this installer would normally run once and `createEntityTables` would only be called once.

Configuring our New Entity

When you call `installEntites`, Magento will need to do several things to install your entities. Before it can do this, it needs to know what your entities are. The contract here is for you to have `getDefaultEntities` return the entities. (It's also possible to pass installEntities a list of entities to install, but I'm trying to stick to the Magento core conventions).

Side Note: Strangely, Magento entities are configured using a simple nested `array` structure. It seems an odd choice in a system that's been described by some as OO'd to death.

To start with, we'll add our Eavblogpost entity and give it a single attribute named title.

```
class Magentotutorial_Complexworld_Entity_Setup extends Mage_Eav_Model_Entity
_Setup {
    public function getDefaultEntities()
    {
        return array (
            'complexworld_eavblogpost' => array(
                'entity_model'      => 'complexworld/eavblogpost',
                'attribute_model'   => '',
                'table'             => 'complexworld/eavblogpost',
                'attributes'        => array(
                    'title' => array(
                        //the EAV attribute type, NOT a mysql varchar
                        'type'              => 'varchar',
                        'backend'           => '',
                        'frontend'          => '',
                        'label'             => 'Title',
                        'input'             => 'text',
                        'class'             => '',
                        'source'            => '',
                        // store scope == 0
```

```
                        // global scope == 1
                        // website scope == 2
                        'global'            => 0,
                        'visible'           => true,
                        'required'          => true,
                        'user_defined'      => true,
                        'default'           => '',
                        'searchable'        => false,
                        'filterable'        => false,
                        'comparable'        => false,
                        'visible_on_front'  => false,
                        'unique'            => false,
                    ),
                ),
            )
        );
    }
}
```

All right, that's a pile of code. Let's break it apart.

Expected Return Value

The `getDefaultEntities` method should return a php array of key/value pairs. Each key should be the name of the entity type (setup with `$installer->addEntityType('complexworld_eavblogpost',...`, each value should be an array that describes the entity.

Array that Describes the Entity

The array that describes the entity is also a list of key/value pairs. Some of these should look familiar

```
'entity_model'      => 'complexworld/eavblogpost',
'attribute_model'   => '',
'table'             => 'complexworld/eavblogpost',
'attributes'        => array(
```

These should match the values you used in the call to `$installer->addEntityType(...`. The final key, `attributes`, should contain yet another array that describes the attributes themselves.

Yet Another Array that Describes the Attributes Themselves

This next array is, yet again, an array of key value pairs. This time the key is the attribute name (`title`) and the values are a final array of key value pairs that define the attribute. For the sake of simplicity we've chose to define a single attribute, but you could go on to define as many as you'd like.

Final Array of Key Value Pairs that Define the Attribute

Finally, we have a long list of attribute properties.

```
//the EAV attribute type, NOT a mysql varchar
'type'              => 'varchar',
'backend'           => '',
'frontend'          => '',
'label'             => 'Title',
'input'             => 'text',
'class'             => '',
'source'            => '',
// store scope == 0
// global scope == 1
// website scope == 2
'global'            => 0,
'visible'           => true,
'required'          => true,
'user_defined'      => true,
'default'           => '',
'searchable'        => false,
'filterable'        => false,
'comparable'        => false,
'visible_on_front'  => false,
'unique'            => false,
```

Unfortunately, this is where your author has to 'fess up and tell you he's unsure what most of these do. Many involve driving features of the Magento back-end UI, such as `label` and `input`. Magento engineers have chosen to tightly bind their UI implementation with their back-end Model structure. This allows them certain advantages, but it means there are large parts of the system that remain opaque to outsiders, particularly web developers who've been chanting the mantra of back-end/front-end separation for near on a decade.

That said, the one important property you'll want to make note of is

```
'type' => 'varchar'
```

This defines the type of the value that the attribute will contain. You'll recall that we added table for each attribute type

```
eavblog_posts_datetime
eavblog_posts_decimal
eavblog_posts_int
eavblog_posts_text
eavblog_posts_varchar
```

While these do not refer to the MySQL column types, (but instead the EAV attribute types), their names (varchar, datetime, etc.) are indicative of the values they'll hold.

So, now that we have everything in place, lets refresh things one last time to run our installer script. After calling `installEntities`, we should have

1. A new row in `eav_attribute` for the title attribute
2. A new row in `eav_entity_attribute`

Tying it all Together

This is clearly the lamest.blogmodel.ever, but lets try adding some rows and iterating through a collection and get the heck out of here before our heads explode. Add the following two actions to your Index Controller.

```
public function populateEntriesAction() {
    for($i=0;$i<10;$i++) {
        $weblog2 = Mage::getModel('complexworld/eavblogpost');
        $weblog2->setTitle('This is a test '.$i);
        $weblog2->save();
    }

    echo 'Done';
}

public function showcollectionAction() {
    $weblog2 = Mage::getModel('complexworld/eavblogpost');
    $entries = $weblog2->getCollection()-
>addAttributeToSelect('title');
    $entries->load();
    foreach($entries as $entry)
    {
        // var_dump($entry->getData());
        echo '<h1>'.$entry->getTitle().'</h1>';
    }
    echo '<br>Done<br>';
}
```

Let's populate some entries! Load up the following URL

```
http://magento.dev/index.php/complexworld/index/populateEntries
```

If you take a look at your database, you should see 10 new rows in the `eavblog_posts` table.

```
mysql> select * from eavblog_posts order by entity_id DESC;
+-----------+----------------+------------------+--------------+-----------+-
---------+---------------------+---------------------+-----------+
| entity_id | entity_type_id | attribute_set_id | increment_id | parent_id |
store_id | created_at          | updated_at          | is_active |
+-----------+----------------+------------------+--------------+-----------+-
---------+---------------------+---------------------+-----------+
|        10 |             31 |                0 |              |         0 |
       0 | 2009-12-06 08:36:41 | 2009-12-06 08:36:41 |         1 |
|         9 |             31 |                0 |              |         0 |
       0 | 2009-12-06 08:36:41 | 2009-12-06 08:36:41 |         1 |
|         8 |             31 |                0 |              |         0 |
       0 | 2009-12-06 08:36:41 | 2009-12-06 08:36:41 |         1 |
|         7 |             31 |                0 |              |         0 |
       0 | 2009-12-06 08:36:41 | 2009-12-06 08:36:41 |         1 |
```

```
|         6 |              31 |                 0 |              |          0 |
       0 | 2009-12-06 08:36:41 | 2009-12-06 08:36:41 |          1 |
|         5 |              31 |                 0 |              |          0 |
       0 | 2009-12-06 08:36:41 | 2009-12-06 08:36:41 |          1 |
|         4 |              31 |                 0 |              |          0 |
       0 | 2009-12-06 08:36:41 | 2009-12-06 08:36:41 |          1 |
|         3 |              31 |                 0 |              |          0 |
       0 | 2009-12-06 08:36:41 | 2009-12-06 08:36:41 |          1 |
|         2 |              31 |                 0 |              |          0 |
       0 | 2009-12-06 08:36:41 | 2009-12-06 08:36:41 |          1 |
|         1 |              31 |                 0 |              |          0 |
       0 | 2009-12-06 08:36:41 | 2009-12-06 08:36:41 |          1 |
+-----------+-----------------+-------------------+--------------+-----------+-
---------+---------------------+---------------------+-----------+
```

as well as 10 new rows in the `eavblog_posts_varchar` table.

```
mysql> select * from eavblog_posts_varchar order by value_id DESC;
+----------+----------------+--------------+----------+-----------+----------
--------+
| value_id | entity_type_id | attribute_id | store_id | entity_id | value
        |
+----------+----------------+--------------+----------+-----------+----------
--------+
|       10 |             31 |          933 |        0 |        10 | This is a
 test 9 |
|        9 |             31 |          933 |        0 |         9 | This is a
 test 8 |
|        8 |             31 |          933 |        0 |         8 | This is a
 test 7 |
|        7 |             31 |          933 |        0 |         7 | This is a
 test 6 |
|        6 |             31 |          933 |        0 |         6 | This is a
 test 5 |
|        5 |             31 |          933 |        0 |         5 | This is a
 test 4 |
|        4 |             31 |          933 |        0 |         4 | This is a
 test 3 |
|        3 |             31 |          933 |        0 |         3 | This is a
 test 2 |
|        2 |             31 |          933 |        0 |         2 | This is a
 test 1 |
|        1 |             31 |          933 |        0 |         1 | This is a
 test 0 |
+----------+----------------+--------------+----------+-----------+----------
--------+
```

Notice that `eavblog_posts_varchar` is indexed to `eavblog_posts` by `entity_id`.

Finally, let's pull our Models back out. Load the following URL in your browser

`http://magento.dev/index.php/complexworld/index/showCollection`

This should give us a

```
Warning:
include(Magentotutorial/Complexworld/Model/Resource/Eav/Mysql4/Eavblogpost/Co
llection.php) [function.include]: failed to open stream: No such file or
directory  in /Users/username/Sites/magento.dev/lib/Varien/Autoload.php on
line 93
```

So Close! We didn't make a class for our collection object! Fortunately, doing so is just as easy as with a regular Model Resource. Add the following file with the following contents

File: Magentotutorial/Complexworld/Model/Resource/Eav/Mysql4/Eavblogpost/Collection.php:

```
class Magentotutorial_Complexworld_Model_Resource_Eav_Mysql4_Eavblogpost_Coll
ection extends Mage_Eav_Model_Entity_Collection_Abstract
{

    protected function _construct()
    {
        $this->_init('complexworld/eavblogpost');
    }
}
```

This is just a standard Magento `_construct` method to initialize the Model. With this in place, reload the page, and we'll see all the titles outputted.

Which Attributes?

Those of you with sharp eyes may have noticed something slightly different about the collection loading.

```
$entries = $weblog2->getCollection()->addAttributeToSelect('title');
```

Because querying for EAV data can be SQL intensive, you'll need to specify which attributes it is you want your Models to fetch for you. This way the system can make only the queries it needs. If you're willing to suffer the performance consequences, you can use a wild card to grab **all** the attributes

```
$entries = $weblog2->getCollection()->addAttributeToSelect('*');
```

Jumping Off

So, that should give you enough information to be dangerous, or at least enough information so you're not drowning the next time you're trying to figure out why the yellow shirts aren't showing up in your store. There's still plenty to learn about EAV; here's a few topics I would have liked to cover in greater detail, and may talk about in future articles

1. EAV Attributes: Attributes aren't limited to datetime, decimal, int, text and varchar. You can create your own class files to model different attributes. This is what the blank `attribute_model` is for.

2. Collection Filtering: Filtering on EAV collections can get tricky, especially when you're dealing with the above mentioned non-simple attributes. You need to use the `addAttributeToFilter` method on your collection before loading.
3. The Magento EAV Hierarchy: Magento has taken their basic EAV Model and built up a hierarchy that's very tied to store functionality, as well as including strategies to reduce the number of queries an EAV Model generates (the concept of a **flat** Model, for example)

EAV Models are, without a doubt, the most complicated part of the Magento system that an ecommerce web developer will need to deal with. Remember to take deep breaths and that, at the end of the day, its just programming. Everything happens for a concrete reason, you just need to figure out why.

Magento for Developers: Part 8 - Varien Data Collections

Originally, as a PHP programmer, if you wanted to collect together a group of related variables you had one choice, the venerable Array. While it shares a name with C's array of memory addresses, a PHP array is a general purpose dictionary like object combined with the behaviors of a numerically indexed mutable array.

In other languages the choice isn't so simple. You have multiple data structures to chose from, each offering particular advantages in storage, speed and semantics. The PHP philosophy was to remove this choice from the client programmer and give them one useful data structure that was "good enough".

All of this is galling to a certain type of software developer, and PHP 5 set out to change the status quo by offering built-in classes and interfaces that allow you to create your own data structures.

```
$array = new ArrayObject();
class MyCollection extends ArrayObject{...}
$collection = new MyCollection();
$collection[] = 'bar';
```

While this is still galling to a certain type of software developer, as you don't have access to low level implementation details, you do have the ability to create array-like Objects with methods that encapsulate specific functionality. You can also setup rules to offer a level of type safety by only allowing certain kinds of Objects into your Collection.

It should come as no surprise that Magento offers you a number of these Collections. In fact, every Model object that follows the Magento interfaces gets a Collection type for free. Understanding how these Collections work is a key part to being an effective Magento programmer. We're going to take a look at Magento Collections, starting from the bottom and working our way up. Setup a controller action where you can run arbitrary code, and let's get started.

A Collection of Things

First, we're going to create a few new Objects.

```
$thing_1 = new Varien_Object();
$thing_1->setName('Richard');
$thing_1->setAge(24);

$thing_2 = new Varien_Object();
$thing_2->setName('Jane');
$thing_2->setAge(12);

$thing_3 = new Varien_Object();
$thing_3->setName('Spot');
$thing_3->setLastName('The Dog');
$thing_3->setAge(7);
```

The `Varien_Object` class defines the object all Magento Models inherit from. This is a common pattern in object oriented systems, and ensures you'll always have a way to easily add methods/functionally to **every** object in your system without having to edit every class file.

Any Object that extends from `Varien_Object` has magic getter and setters that can be used to set data properties. Give this a try

```
var_dump($thing_1->getName());
```

If you don't know what the property name you're after is, you can pull out all the data as an array

```
var_dump($thing_3->getData());
```

The above will give you an array something like

```
array
'name' => string 'Spot' (length=4)
'last_name' => string 'The Dog' (length=7)
'age' => int 7
```

Notice the property named "last_name"? If there's an underscore separated property, you camel case it if you want to use the getter and setter magic.

```
$thing_1->setLastName('Smith');
```

In more recent versions of Magento, use can use Array style bracket to access properties

```
var_dump($thing_3["last_name"]);
```

The ability to do these kinds of things is part of the power of PHP5, and the development style a certain class of people mean when they say "Object Oriented Programming".

So, now that we have some Objects, let's add them to a Collection. Remember, a Collection is like an Array, but is defined by a PHP programmer.

```
$collection_of_things = new Varien_Data_Collection();
$collection_of_things
->addItem($thing_1)
->addItem($thing_2)
->addItem($thing_3);
```

The `Varien_Data_Collection` is the Collection that most Magento data Collections inherit from. Any method you can call on a `Varien_Data_Collection` you can call on Collections higher up the chain (We'll see more of this later)

What can we do with a Collection? For one, with can use `foreach` to iterate over it

```
foreach($collection_of_things as $thing)
{
    var_dump($thing->getData());
}
```

There are also shortcuts for pulling out the first and last items

```
var_dump($collection_of_things->getFirstItem());
var_dump($collection_of_things->getLastItem()->getData());
```

Want your Collection data as XML? There's a method for that

```
var_dump( $collection_of_things->toXml() );
```

Only want a particular field?

```
var_dump($collection_of_things->getColumnValues('name'));
```

The team at Magento have even given us some rudimentary filtering capabilities.

```
var_dump($collection_of_things->getItemsByColumnValue('name','Spot'));
```

Neat stuff.

Model Collections

So, this is an interesting exercise, but why do we care?

We care because all of Magento's built in data Collections inherit from this object. That means if you have, say, a product Collection you can do the same sort of things. Let's take a look

```
public function testAction()
{
    $collection_of_products = Mage::getModel('catalog/product')-
```

```
>getCollection();
    var_dump($collection_of_products->getFirstItem()->getData());
}
```

Most Magento Model objects have a method named `getCollection` which will return a collection that, by default, is initialized to return every Object of that type in the system.

A Quick Note: Magento's Data Collections contain a lot of complicated logic that handles when to use an index or cache, as well as the logic for the EAV entity system. Successive method calls to the same Collection over its life can often result in unexpected behavior. Because of that, all the of the following examples are wrapped in a single method action. I'd recommend doing the same while you're experimenting. Also, XDebug's `var_dump` is a godsend when working with Magento Objects and Collections, as it will (usually) intelligently short circuit showing hugely recursive Objects, but still display a useful representation of the Object structure to you.

The products Collection, as well as many other Magento Collections, also have the `Varien_Data_Collection_Db` class in their ancestor chain. This gives us a lot of useful methods. For example, if you want to see the select statement your Collection is using

```
public function testAction()
{
    $collection_of_products = Mage::getModel('catalog/product')-
>getCollection();
    var_dump($collection_of_products-
>getSelect()); //might cause a segmentation fault
}
```

The output of the above will be

```
object(Varien_Db_Select)[94]
  protected '_bind' =>
    array
      empty
  protected '_adapter' =>
...
```

Whoops! Since Magento is using the `Zend` database abstraction layer, your Select is also an Object. Let's see that as a more useful string.

```
public function testAction()
{
    $collection_of_products = Mage::getModel('catalog/product')-
>getCollection();
    //var_dump($collection_of_products-
>getSelect()); //might cause a segmentation fault
    var_dump(
        (string) $collection_of_products->getSelect()
    );
}
```

Sometimes this is going to result in a simple select

```
'SELECT `e`.* FROM `catalog_product_entity` AS `e`'
```

Other times, something a bit more complex

```
string 'SELECT `e`.*, `price_index`.`price`, `price_index`.`final_price`,
IF(`price_index`.`tier_price`, LEAST(`price_index`.`min_price`,
`price_index`.`tier_price`), `price_index`.`min_price`) AS `minimal_price`,
`price_index`.`min_price`, `price_index`.`max_price`,
`price_index`.`tier_price` FROM `catalog_product_entity` AS `e`
INNER JOIN `catalog_product_index_price` AS `price_index` ON
price_index.entity_id = e.entity_id AND price_index.website_id = '1' AND
price_index.customer_group_id = 0'
```

The discrepancy depends on which attributes you're selecting, as well as the aforementioned indexing and cache. If you've been following along with the other articles in this series, you know that many Magento models (including the Product Model) use an EAV system. By default, a EAV Collection will not include all of an Object's attributes. You can add them all by using the addAttributeToSelect method

```
$collection_of_products = Mage::getModel('catalog/product')
->getCollection()
->addAttributeToSelect('*');  //the asterisk is like a SQL SELECT *
```

Or, you can add just one

```
//or just one
$collection_of_products = Mage::getModel('catalog/product')
->getCollection()
->addAttributeToSelect('meta_title');
```

or chain together several

```
//or just one
$collection_of_products = Mage::getModel('catalog/product')
->getCollection()
->addAttributeToSelect('meta_title')
->addAttributeToSelect('price');
```

Lazy Loading

One thing that will trip up PHP developers new to Magento's ORM system is **when** Magento makes its database calls. When you're writing literal SQL, or even when you're using a basic ORM system, SQL calls are often made immediately when instantiating an Object.

```
$model = new Customer();
//SQL Calls being made to Populate the Object
echo 'Done'; //execution continues
```

Magento doesn't work that way. Instead, the concept of Lazy Loading is used. In simplified terms, Lazy loading means that no SQL calls are made until the client-programmer needs to access the data. That means when you do something something like this

```
$collection_of_products = Mage::getModel('catalog/product')
->getCollection();
```

Magento actually hasn't gone out to the database yet. You can safely add attributes later

```
$collection_of_products = Mage::getModel('catalog/product')
->getCollection();
$collection_of_products->addAttributeToSelect('meta_title');
```

and not have to worry that Magento is making a database query each time a new attribute is added. The database query will not be made until you attempt to access an item in the Collection.

In general, try not to worry too much about the implementation details in your day to day work. It's good to know that there's s SQL backend and Magento is doing SQLy things, but when you're coding up a feature try to forget about it, and just treat the objects as block boxes that do what you need.

Filtering Database Collections

The most important method on a database Collection is `addFieldToFilter`. This adds your `WHERE` clauses to the SQL query being used behind the scenes. Consider this bit of code, run against the sample data database (substitute your own SKU is you're using a different set of product data)

```
public function testAction()
{
    $collection_of_products = Mage::getModel('catalog/product')
    ->getCollection();
    $collection_of_products->addFieldToFilter('sku','n2610');

    //another neat thing about collections is you can pass them into the coun
t       //function.  More PHP5 powered goodness
    echo "Our collection now has " . count($collection_of_products) . ' item(
s)';
    var_dump($collection_of_products->getFirstItem()->getData());
}
```

The first parameter of `addFieldToFilter` is the attribute you wish to filter by. The second is the value you're looking for. Here's we're adding a `sku` filter for the value `n2610`.

The second parameter can also be used to specify the **type** of filtering you want to do. This is where things get a little complicated, and worth going into with a little more depth.

So by default, the following

```
$collection_of_products->addFieldToFilter('sku','n2610');
```

is (essentially) equivalent to

```
WHERE sku = "n2610"
```

Take a look for yourself. Running the following

```
public function testAction()
{
    var_dump(
    (string)
    Mage::getModel('catalog/product')
    ->getCollection()
    ->addFieldToFilter('sku','n2610')
    ->getSelect());
}
```

will yield

```
SELECT `e`.* FROM `catalog_product_entity` AS `e` WHERE (e.sku = 'n2610')
```

Keep in mind, this can get complicated fast if you're using an EAV attribute. Add an attribute

```
var_dump(
(string)
Mage::getModel('catalog/product')
->getCollection()
->addAttributeToSelect('*')
->addFieldToFilter('meta_title','my title')
->getSelect()
);
```

and the query gets gnarly.

```
SELECT `e`.*, IF(_table_meta_title.value_id>0, _table_meta_title.value, _tabl
e_meta_title_default.value) AS `meta_title`
FROM `catalog_product_entity` AS `e`
INNER JOIN `catalog_product_entity_varchar` AS `_table_meta_title_default`
    ON (_table_meta_title_default.entity_id = e.entity_id) AND (_table_meta_t
itle_default.attribute_id='103')
    AND _table_meta_title_default.store_id=0
LEFT JOIN `catalog_product_entity_varchar` AS `_table_meta_title`
    ON (_table_meta_title.entity_id = e.entity_id) AND (_table_meta_title.att
ribute_id='103')
    AND (_table_meta_title.store_id='1')
WHERE (IF(_table_meta_title.value_id>0, _table_meta_title.value, _table_meta_
title_default.value) = 'my title')
```

Not to belabor the point, but try not to think too much about the SQL if you're on deadline.

Other Comparison Operators

I'm sure you're wondering "what if I want something other than an equals by query"? Not equal, greater than, less than, etc. The `addFieldToFilter` method's second parameter has you covered there as well. It supports an alternate syntax where, instead of passing in a string, you pass in a single element Array.

The key of this array is the **type** of comparison you want to make. The value associated with that key is the value you want to filter by. Let's redo the above filter, but with this explicit syntax

```
public function testAction()
{
    var_dump(
    (string)
    Mage::getModel('catalog/product')
    ->getCollection()
    ->addFieldToFilter('sku',array('eq'=>'n2610'))
    ->getSelect()
    );
}
```

Calling out our filter

```
addFieldToFilter('sku',array('eq'=>'n2610'))
```

As you can see, the second parameter is a PHP Array. Its key is `eq`, which stands for *equals*. The value for this key is `n2610`, which is the value we're filtering on.

Magento has a number of these english language like filters that will bring a [tear of remembrance] (and perhaps pain) to any old perl developers in the audience.

Listed below are all the filters, along with an example of their SQL equivalents.

```
array("eq"=>'n2610')
WHERE (e.sku = 'n2610')

array("neq"=>'n2610')
WHERE (e.sku != 'n2610')

array("like"=>'n2610')
WHERE (e.sku like 'n2610')

array("nlike"=>'n2610')
WHERE (e.sku not like 'n2610')

array("is"=>'n2610')
WHERE (e.sku is 'n2610')

array("in"=>array('n2610'))
WHERE (e.sku in ('n2610'))

array("nin"=>array('n2610'))
WHERE (e.sku not in ('n2610'))
```

```
array("notnull"=>'n2610')
WHERE (e.sku is NOT NULL)

array("null"=>'n2610')
WHERE (e.sku is NULL)

array("gt"=>'n2610')
WHERE (e.sku > 'n2610')

array("lt"=>'n2610')
WHERE (e.sku < 'n2610')

array("gteq"=>'n2610')
WHERE (e.sku >= 'n2610')

array("moreq"=>'n2610') //a weird, second way to do greater than equal
WHERE (e.sku >= 'n2610')

array("lteq"=>'n2610')
WHERE (e.sku <= 'n2610')

array("finset"=>array('n2610'))
WHERE (find_in_set('n2610',e.sku))

array('from'=>'10','to'=>'20')
WHERE e.sku >= '10' and e.sku <= '20'
```

Most of these are self explanatory, but a few deserve a special callout

in, nin, find_in_set

The `in` and `nin` conditionals allow you to pass in an Array of values. That is, the value portion of your filter array is itself allowed to be an array.

```
array("in"=>array('n2610','ABC123')
WHERE (e.sku in ('n2610','ABC123'))
```

notnull, null

The keyword NULL is special in most flavors of SQL. It typically won't play nice with the standard equality (=) operator. Specifying `notnull` or `null` as your filter type will get you the correct syntax for a NULL comparison while ignoring whatever value you pass in

```
array("notnull"=>'n2610')
WHERE (e.sku is NOT NULL)
```

from - to filter

This is another special format that breaks the standard rule. Instead of a single element array, you specify a two element array. One element has the key `from`, the other element has the key `to`. As

the keys indicated, this filter allows you to construct a from/to range without having to worry about greater than and less than symbols

```
public function testAction
{
        var_dump(
        (string)
        Mage::getModel('catalog/product')
        ->getCollection()
        ->addFieldToFilter('price',array('from'=>'10','to'=>'20'))
        ->getSelect()
        );
}
```

The above yields

```
WHERE (_table_price.value >= '10' and _table_price.value <= '20')
```

AND or OR, or is that OR and AND?

Finally, we come to the boolean operators. It's the rare moment where we're only filtering by one attribute. Fortunately, Magento's Collections have us covered. You can chain together multiple calls to `addFieldToFilter` to get a number of "AND" queries.

```
function testAction()
{
        echo(
        (string)
        Mage::getModel('catalog/product')
        ->getCollection()
        ->addFieldToFilter('sku',array('like'=>'a%'))
        ->addFieldToFilter('sku',array('like'=>'b%'))
        ->getSelect()
        );
}
```

By chaining together multiple calls as above, we'll produce a where clause that looks something like the the following

```
WHERE (e.sku like 'a%') AND (e.sku like 'b%')
```

To those of you that just raised your hand, yes, the above example would always return 0 records. No sku can begin with BOTH an `a` and a `b`. What we probably want here is an `OR` query. This brings us to another confusing aspect of `addFieldToFilter`'s second parameter.

If you want to build an `OR` query, you need to pass an Array of filter Arrays in as the second parameter. I find it's best to assign your individual filter Arrays to variables

```
public function testAction()
{
```

```
        $filter_a = array('like'=>'a%');
        $filter_b = array('like'=>'b%');
}
```

and then assign an array of all my filter variables

```
public function testAction()
{
        $filter_a = array('like'=>'a%');
        $filter_b = array('like'=>'b%');
        echo(
        (string)
        Mage::getModel('catalog/product')
        ->getCollection()
        ->addFieldToFilter('sku',array($filter_a,$filter_b))
        ->getSelect()
        );
}
```

In the interest of being explicit, here's the aforementioned Array of filter Arrays.

```
array($filter_a,$filter_b)
```

This will gives us a WHERE clause that looks something like the following

```
WHERE (((e.sku like 'a%') or (e.sku like 'b%')))
```

Wrap Up

You're now a Magento developer walking around with some serious firepower. Without having to write a single line of SQL you now know how to query Magento for any Model your store or application might need.