In the world of e-commerce, there are dozens of tools available to use in custom web design and development. Out of all of these options, Magento is the web's fastest growing e-commerce platform. Its architecture makes it possible to extend the functionalities using plugins, which are shared by the community, extensively.

This book is a practical guide packed with examples so developers can understand its fundamental concepts and then start developing and testing Magento code. The book starts by building your knowledge of Magento, providing you with the techniques and tools you require to start your first Magento development. You will then learn to work with data collections and the EAV system, before gradually moving on to extending the frontend and backend. You will learn all about deploying and distributing custom modules.

This book will help you navigate your way around your first Magento developments, helping you to avoid all of the most common headaches that new developers face when first getting started.

Who this book is written for
This book is written for PHP developers who are familiar with frameworks such as Zend or CakePHP and want to start developing and working with Magento. No prior knowledge of Magento is required.

What you will learn from this book
- Build your very first Magento extension
- Deploy and distribute custom modules
- Extend Magento's core functionality
- Use development techniques and tools for Magento
- Implement the design patterns that form Magento
- Test Magento with different tools and frameworks

Get up and running with the highly customizable and powerful e-commerce solution, Magento
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 ‘Magento Fundamentals for Developers’
- A synopsis of the book’s content
Allan MacGregor is a Magento Certified Developer Plus with 4 years of Magento experience. He also has a certification in Linux System Administration from IBM. He started working with Magento as a freelancer, looking for a better framework to build e-commerce solutions with, and he is now the Magento lead developer at Demac Media (http://www.demacmedia.com). At Demac Media, he has participated in building core solutions for a wide range of clients; this has given him the experience and knowledge to solve many Magento challenges. As part of an internal project at Demac Media, he worked on Triplecheck.io (http://www.triplecheck.io), a unique service to monitor and audit the code health of a Magento store. He's very passionate about software development in general. He is constantly working with new technologies and frameworks. You can also follow him on Twitter at http://www.twitter.com/allanmacgregor.
Preface

This book will help new and not-so-new developers understand and work with the Magento fundamental concepts and standard practices in order to develop and test Magento code.

This book is my attempt at writing a book that answers questions that many developers, including myself, had when we started to develop for Magento, for example, what is EAV? How does the ORM in Magento work? What are observers and events? What were the design patterns used to create Magento?

More importantly, it will also answer questions that many developers still have, for example, what is the standard to develop modules and extend the frontend and backend? How can I properly test my code? What is the best method to deploy and distribute custom modules?

What this book covers

Chapter 1, *Magento Fundamentals for Developers*, discusses Magento's fundamental concepts, such as the system architecture and the MVC implementation and its relation with the Zend Framework. All the concepts in this chapter will set the foundation for developers starting with Magento.

Chapter 2, *ORM and Data Collections*, introduces you to the Magento ORM system. It explains how collections and models are the bread and butter of everyday Magento development. We will learn how to properly work with data collections and the EAV system.

Chapter 3, *Frontend Development*, discusses the practical use of the skills and knowledge we have acquired so far, and we'll build a fully functional Magento module step by step. The custom module will allow readers to apply a variety of important concepts, such as working with collections, routing, sessions, and caching.
Chapter 4, *Backend Development*, extends on what we built in the previous chapter—that is, the frontend part of our module—and create an interface in the Magento backend for interaction with our application data. We will learn about extending the backend and the adminhtml theme, setting data sources, and controlling our extension behavior through configuration.

Chapter 5, *The Magento API*, discusses the Magento API and how we can extend it to provide access to the custom data that we captured using our extension.

Chapter 6, *Testing and Quality Assurance*, teaches us the critical skills required to test our custom Magento modules; this is an integral part of development. We will learn about the different types of tests and the tools available for each particular type of test.

Chapter 7, *Deployment and Distribution*, discusses the multiple tools available in order to deploy our code to a production environment, and we will also learn how to properly pack our extensions for distribution through channels such as Magento Connect.

Appendix A, *Hello Magento!*, gives new developers a quick and easy-to-follow introduction to create our first Magento extension.

Appendix B, *Understanding and Setting Up Our Development Environment*, discusses steps to set up a complete environment for Magento development with MySQL and Apache. Additionally, we will go over the tools available to facilitate the development, several IDEs, and version control systems.
Magento Fundamentals for Developers

In this chapter, we will cover the fundamental concepts of working with Magento. We will learn how Magento is structured, and go over the source of Magento’s flexibility, that is, its modular architecture.

Magento is a flexible and powerful system. Unfortunately, this adds some level of complexity as well. Currently, a clean installation of Magento has around 20,000 files and over 1.2 million lines of code.

With all that power and complexity, Magento can be daunting for new developers; but don’t worry. This chapter is designed to give new developers all the fundamental concepts and tools they need to use and extend Magento, and in the next chapter, we will dive deep into Magento models and data collection.

Zend Framework – the base of Magento

As you probably know, Magento is the most powerful e-commerce platform in the market. What you might not know about Magento is that it is also an object-oriented (OO) PHP framework developed on top of Zend Framework.

Zend Framework 2 has been available since 2013, but Magento still relies on Zend Framework 1.11.
Here's how Zend's official site describes the framework:

"We designed Zend Framework with simplicity in mind. To provide a lightweight, loosely-coupled component library simplified to provide 4/5s of the functionality everyone needs and that lets you customize the other 20% to meet your specific business needs. By focusing on the most commonly needed functionality, we retain the simplified spirit of PHP programming, while dramatically lower the learning curve – and your training costs – so developers get up-to-speed quickly."


What exactly is Zend Framework? Zend Framework is an OO framework developed on PHP that implements the Model-View-Controller (MVC) paradigm. When Varien (now Magento Inc.) started developing Magento, they decided to do it on top of Zend because of some components, some of which are as follows:

- Zend_Cache
- Zend_Acl
- Zend_Locale
- Zend_DB
- Zend_Pdf
- Zend_Currency
- Zend_Date
- Zend_Soap
- Zend_Http

In total, Magento uses around 15 different Zend components. The Varien library extends several of the Zend components mentioned before directly. For example, Varien_Cache_Core extends from Zend_Cache_Core.

Using Zend Framework, Magento was built with the following principles in mind:

- **Maintainable**: By using code pools to keep the core code separate from local customizations and third-party modules
- **Upgradable**: Magento modularity allows extensions and third-party modules to be updated independently from the rest of the system
- **Flexible**: Allows seamless customization and simplifies the development of new features
Although experience of using Zend Framework or even understanding it are not requirements to develop Magento, having at least some basic understanding of the Zend components, usage, and interaction can be invaluable information as we start digging deeper into the core of Magento.

You can learn more about Zend Framework at http://framework.zend.com/.

The Magento folder structure

The Magento folder structure is slightly different from other MVC applications. Let’s take a look at the directory tree, and each directory and its functions:

- **app**: This folder is the core of Magento and is subdivided into three important directories:
  - code: This contains all our application code divided into three code pools, namely core, community, and local
  - design: This contains all the templates and layouts for our application
  - locale: This contains all the translation and e-mail template files used for the store

- **js**: This contains all the JavaScript libraries that are used in Magento

- **media**: This contains all the images and media files for our products and CMS pages, as well the product image cache

- **lib**: This contains all the third-party libraries used in Magento (such as Zend and PEAR) as well as the custom libraries developed by Magento, which reside under the Varien and Mage directories

- **skin**: This contains all CSS, images and JavaScript used by the corresponding theme

- **var**: This contains our temporary data, such as the cache file, index lock files, sessions, import/export files, and in the case of the Enterprise edition, the full page cache folders
Magento is a modular system. This means the application, including the core, is divided into smaller modules. For this reason, the folder structure plays a key role in the organization of each module. A typical Magento module folder structure would look something like this:

Let's review each folder further:

- **Block**: In Magento, blocks form an additional layer of logic between the controllers and views
- **controllers**: These are formed by actions that process webserver requests
- **Controller**: Classes in this folder can be abstract classes and they can be extended by the controller class under the controllers folder
- **etc**: In this we can find the module-specific configuration in the form of XML files such as config.xml and system.xml
- **Helper**: This contains auxiliary classes that encapsulate a common module functionality and make them available to a class of the same module and to other modules classes as well
- **Model**: This contains models that support the controllers in the module to interact with data
- **sql**: These contain the installation and upgrade files for each specific module
- **data**: This folder was introduced in Magento 1.6 CE and it is used in a manner similar to SQL scripts, but data scripts are only concerned about inserting data

As we will see later in this chapter, Magento makes heavy use of factory names and factory methods. This is why the folder structure is so important.
Modular architecture

Rather than being a large application, Magento is built by smaller modules, each adding specific functionality to Magento.

One of the advantages of this approach is the ability to enable and disable specific module functionality with ease as well as adding new functionality by adding new modules.

Autoloader

Magento is a huge framework composed by close to 20,000 files. Requiring every single file when the application starts would make it incredibly slow and resource intensive. Hence, Magento makes use of an autoloader class to require the files each time a factory method is called.

So what exactly is an autoloader? PHP5 includes a function called \_autoload(). When instantiating a class, the \_autoload() function is automatically called. Inside this function, the custom logic is defined to parse the class name and require the file.

Let’s take a closer look at the Magento Bootstrap code located in app/Mage.php:

```php
...
Mage::register('original_include_path', get_include_path());
if (defined('COMPILER_INCLUDE_PATH')) {
    $appPath = COMPILER_INCLUDE_PATH;
    set_include_path($appPath . PS . Mage::registry('original_include_path'));
    include_once "Mage_Core_functions.php";
    include_once "Varien_Autoload.php";
} else {
    /**
     * Set include path
     */
    $paths[] = BP . DS . 'app' . DS . 'code' . DS . 'local';
    $paths[] = BP . DS . 'app' . DS . 'code' . DS . 'community';
    $paths[] = BP . DS . 'app' . DS . 'code' . DS . 'core';
    $paths[] = BP . DS . 'lib';

    $appPath = implode(PS, $paths);
    set_include_path($appPath . PS . Mage::registry('original_include_path'));
    include_once "Mage/Core/functions.php";
    include_once "Varien/Autoload.php";
}

Varien_Autoload::register();
```
The Bootstrap file takes care of defining the include paths and initializing the Varien autoloader, which will in turn define its own autoload function as the default function to call. Let's take a look under the hood and see what the Varien's autoload function is doing:

```php
/**
 * Load class source code
 *
 * @param string $class
 */
public function autoload($class)
{
    if ($this->_collectClasses) {
        $this->_arrLoadedClasses[self::$scope][] = $class;
    }
    if ($this->_isIncludePathDefined) {
        $classFile = COMPILER_INCLUDE_PATH . DIRECTORY_SEPARATOR . $class;
    } else {
        $classFile = str_replace(' ', DIRECTORY_SEPARATOR, ucwords(str_replace('_', ' ', $class)));
    }
    $classFile.= '.php';
    //echo $classFile;die();
    return include $classFile;
}
```

The `autoload` class takes a single parameter called `$class`, which is an alias provided by the factory method. This alias is processed to generate a matching class name that is then included.

As we mentioned before, Magento's directory structure is important due to the fact that Magento derives its class names from the directory structure. This convention is the core principle behind factory methods, which we will be reviewing later in this chapter.

**Code pools**

As mentioned before, inside our `app/code` folder, we have our application code divided into the following three different directories known as code pools:

- **core**: This is where the Magento Core modules that provide the base functionality reside. The golden rule among Magento developers is that you should never, under any circumstance, modify any files under the core code pool.
Community: This is the location where third-party modules are placed. They are either provided by third parties or installed through Magento Connect.

Local: This is where all the modules and code developed specifically for this instance of Magento reside.

The code pools identify where the module came from and in which order they should be loaded. If we take another look at the Mage.php Bootstrap file, we can see the order in which code pools are loaded:

```php
$paths[] = BP . DS . 'app' . DS . 'code' . DS . 'local';
$paths[] = BP . DS . 'app' . DS . 'code' . DS . 'community';
$paths[] = BP . DS . 'app' . DS . 'code' . DS . 'core';
$paths[] = BP . DS . 'lib';
```

This means, for each class request, Magento will look in the local folder, then in the community and core folders, and finally inside the lib folder.

This also produces an interesting behavior that can easily be used to override core and community classes by just copying the directory structure and matching the class name.

Needless to say, this is a terrible practice, but it is still useful to know about, just in case someday you have to take care of a project that exploits this behavior.

Routing and request flow

Before going into more detail about the different components that form a part of Magento, it is important that we understand how these components interact together and how Magento processes requests coming from the web server.

As with any other PHP application, we have a single file as an entry point for every request. In the case of Magento, this file is index.php, which is in charge of loading the Mage.php Bootstrap class and starting the request cycle.

1. The web server receives the request and Magento is instantiated by calling the Bootstrap file Mage.php.
2. The frontend controller is instantiated and initialized. During this controller initialization, Magento searches for the web routes and instantiates them.
3. Magento then iterates through each of the routers and calls the match. The match method is responsible for processing the URL and generating the corresponding controller and action.
4. Instantiates the matching controller and corresponding action.
Routers are especially important in this process. Router objects are used by the frontend controller to match a requested URL (route) to a module controller and action. By default, Magento comes with the following routers:

- Mage_Core_Controller_Varien_Router_Admin
- Mage_Core_Controller_Varien_Router_Standard
- Mage_Core_Controller_Varien_Router_Cms
- Mage_Core_Controller_Varien_Router_Default

The action controller will then load and render the layout, which in turn will load the corresponding blocks, models, and templates.

Let's analyze how Magento will handle a request to a category page. We will use http://localhost/catalog/category/view/id/10 as an example. The Magento URI comprises three parts, namely FrontName/ControllerName/ActionName.

Hence, for our example URL, the breakdown is as follows:

- FrontName: This is a catalog
- ControllerName: This is a category
- ActionName: This is a view

Let's take a look at the Magento router class Mage_Core_Controller_Varien_Router_Standard match function:

```php
public function match(Zend_Controller_Request_Http $request) {
    ...
    $path = trim($request->getPathInfo(), '/');
    if ($path) {
        $p = explode('/', $path);
    } else {
        $p = explode('/', $this->_getDefaultPath());
    }
    ...
}
```
From the previous code, we can see that the first thing the router tries to do is parse the URI into an array. Based on our example URL, the corresponding array will be similar to the following code:

```
$p = Array
(  
    [0] => catalog
    [1] => category
    [2] => view
  )
```

The next part of the function will first try to check if the request has the module name specified. If not, then it tries to determine the module name, based on the first element of our array. If a module name can't be provided, then the function will return false. Let's take a look at this part of the code:

```
// get module name
if ($request->getModuleName()) {
    $module = $request->getModuleName();
} else {
    if (!empty($p[0])) {
        $module = $p[0];
    } else {
        $module = $this->getFront()->getDefault('module');
        $request->setAlias(Mage_Core_Model_Url_Rewrite
            ::REWRITE_REQUEST_PATH_ALIAS, '');
    }
} if (!$module) {
    if (Mage::app()->getStore()->isAdmin()) {
        $module = 'admin';
    } else {
        return false;
    }
}
```

Next, the `match` function will iterate through each of the available modules and try to match the controller and action using the following code:

```
... foreach ($modules as $realModule) {
    $request->setRouteName(
        $this->getRouteByFrontName($module));
```
Magento Fundamentals for Developers

```php
// get controller name
if ($request->getControllerName()) {
    $controller = $request->getControllerName();
} else {
    if (!$empty($p[1])) {
        $controller = $p[1];
    } else {
        $controller =
        $front->getDefault('controller');
        $request->setAlias(
            Mage_Core_Model_Url_Rewrite::REWRITE_REQUEST_PATH_ALIAS,
            ltrim($request->getOriginalPathInfo(), '/')
        );
    }
}

// get action name
if (empty($action)) {
    if ($request->getActionName()) {
        $action = $request->getActionName();
    } else {
    }
}

// checking if this place should be secure
$this->_checkShouldBeSecure($request, '/'.$module.'/'.$controller.'/'.$action);

$controllerClassName = $this->_validateControllerClassName($realModule, $controller);
if (!$controllerClassName) {
    continue;
}
```
// instantiate controller class
$controllerInstance = Mage::getControllerInstance($controllerClassName,
$request, $front->getResponse());

if (!$controllerInstance->hasAction($action)) {
    continue;
}

$found = true;
break;
...

Now that looks like an awful lot of code! Let's break it down further. The first part
of the loop will check if the request has a controller name. If it is not set, it will check
our parameter array's ($p) second value and try to determine the controller name. Then, it will try to do the same for the action name.

If we get this far in the loop, we should have a module name, a controller name,
and an action name. Magento will now use these to try and get matched with the
Controller class name by calling the following code:

    $controllerClassName =
    $this->_validateControllerClassName($realModule, $controller);

This function will not only generate a matching class name, but it will also
validate its existence. In our example case, this function should return
Mage_Catalog_CategoryController.

As we now have a valid class name, we can proceed to instantiate our controller
object. You may probably notice that so far we haven't done anything with our
action yet, and that's precisely the next step on our loop.

Now, our instantiated controller comes with a very handy function called
hasAction(). In essence, all this function does is call a PHP function called
is_callable(), which will check if our current controller has a public function
matching the action name. In our case this will be viewAction().
The reason behind this elaborate matching process and the use of a `foreach` loop is that it is possible for several modules to use the same `frontName`:

Now, `http://localhost/catalog/category/view/id/10` is not a very user-friendly URL. Fortunately, Magento has its own URL rewrite system that allows us to use `http://localhost/books.html`.

Let's dig a little deeper into the URL rewrite system and see how Magento gets the controller and action names from our URL alias. Inside our `Varien/Front.php` controller dispatch function, Magento will call the following action:

```php
Mage::getModel('core/url_rewrite')->rewrite();
```

Before actually looking into the inner working of the rewrite function, let's take a look at the structure of the `core/url_rewrite` model:

```php
Array ( 
    ["url_rewrite_id"] => "10"
    ["store_id"] => "1"
    ["category_id"] => "10"
    ["product_id"] => NULL
    ["id_path"] => "category/10"
    ["request_path"] => "books.html"
    ["target_path"] => "catalog/category/view/id/10"
    ["is_system"] => "1"
    ["options"] => NULL
    ["description"] => NULL
)
As we can see, the rewrite module comprises several properties, but only two of them are of particular interest to us, namely request_path and target_path. Simply put, the job of the rewrite module is to modify the request object path information with the matching values of target_path.

The Magento version of MVC

If you are familiar with traditional MVC implementations such as CakePHP or Symfony, you may know that the most common implementation is called a convention-based MVC. With a convention-based MVC to add a new Model, or let's say a Controller, you only need to create the file/class (following the framework conventions); the system will pick it up automatically.

Magento on the other hand uses a configuration-based MVC pattern, which means creating our file/class is not enough; we have to tell Magento explicitly that we have added a new class using configuration files written in XML.

Each Magento module has a config.xml file that is located under the module's etc/ directory and contains all the relevant module configurations. For example, if we want to add a new module that includes a new model, we need to define a node in the configuration file that tells Magento where to find our model. Here's an example:

```
<global>
  ...
  <models>
    <group_classname>
      <class>Namespace_Modulename_Model</class>
    </group_classname>
    <group_classname>
      ...
    </models>
  ...
</global>
```

Although this might look like additional work, it also gives us a huge amount of flexibility and power. For example, we can rewrite another class by using the rewrite node:

```
<global>
  ...
  <models>
    <modulename>modulename</modulename>
      <rewrite>
        <group>group</group>
          <classgroup_classname>Namespace_Modulename_Model</classgroup_classname>
      </rewritegroup_classname>
    </modulename>
  ...
</models>
```
Magento will then load all the config.xml files and merge them at runtime, creating a single configuration tree.

Additionally, modules can also have a system.xml file that is used to specify configuration options in the Magento backend, which end users can in turn use to configure the module functionality. A snippet of a system.xml file will look like this:

```xml
<config>
    <sections>
        <section_name translate="label">
            <label>Section Description</label>
            <tab>general</tab>
            <frontend_type>text</frontend_type>
            <sort_order>1000</sort_order>
            <show_in_default>1</show_in_default>
            <show_in_website>1</show_in_website>
            <show_in_store>1</show_in_store>
        </section_name>
        <group_name translate="label">
            <label>Demo Of Config Fields</label>
            <frontend_type>text</frontend_type>
            <sort_order>1</sort_order>
            <show_in_default>1</show_in_default>
            <show_in_website>1</show_in_website>
            <show_in_store>1</show_in_store>
        </group_name>
        <fields>
            <field_name translate="label comment">
                <label>Enabled</label>
                <comment><![CDATA[Comments can contain
                    strong HTML]]></comment>
                <frontend_type>select</frontend_type>
                <source_model>adminhtml/system_config_source_yesno</source_model>
                <sort_order>10</sort_order>
                <show_in_default>1</show_in_default>
                <show_in_website>1</show_in_website>
                <show_in_store>1</show_in_store>
            </field_name>
        </fields>
    </sections>
</config>
```
Let's break down each node function:

- **section_name**: This is just an arbitrary name that we use to identify our configuration section. Inside this node, we will specify all the fields and groups for the configuration section.
- **group**: Groups, as the name implies, are used to group configuration options and display them inside an accordion section.
- **label**: This defines the title or label to be used on the field/section/group.
- **tab**: This defines the tab on which the section should be displayed.
- **frontend_type**: This node allows us to specify which renderer to use for our custom option field. Some of the available options are as follows:
  - Button
  - Checkboxes
  - Checkbox
  - Date
  - File
  - Hidden
  - Image
  - Label
  - Link
  - Multiline
  - Multiselect
  - Password
  - Radio
  - Radios
  - Select
  - Submit
  - Textarea
  - Text
  - Time
• **sort_order**: This specifies the position of the field, group, or section.

• **source_model**: Certain type of fields, such as a select field, can take options from a source model. Magento already provides several useful classes under `Mage/Adminhtml/Model/System/Config/Source`. Some of the classes we can find are as follows:

  - YesNo
  - Country
  - Currency
  - AllRegions
  - Category
  - Language

Just by using XML, we can build complex configuration options for our modules right on the Magento backend without having to worry about setting up templates, populating fields, or validating data.

Magento is also kind enough to provide a comprehensive amount of form field validation models that we can use with the `<validate>` tag. Among the field validators we have the following options:

- validate-email
- validate-length
- validate-url
- validate-select
- validate-password

As with any other part of Magento, we can extend `source_models`, `frontend_types`, and validators, and even create new ones. We will be tackling this task in a later chapter, where we will create a new type of each. For now, we will explore the concepts of models, views, file layouts, and controllers.

**Models**

Magento makes use of the ORM approach, although we can still use `Zend_Db` to access the database directly. We will be using models to access our data most of the time. For this type of task, Magento provides the following two types of models:

- **Simple models**: These model implementations are a simple mapping of one object to one table, meaning our object attributes match each field and our table structure
Entity Attribute Value (EAV) models: These type of models are used to describe entities with a dynamic number of attributes

Magento splits the model layer in two parts: a model handling the business logic and a resource handling the database interaction. This design decision allows Magento to support multiple database platforms without having to change any of the logic inside the models.

Magento ORM uses one of PHP’s magic class methods to provide dynamic access to object properties. In the next chapter, we will look into models, the Magento ORM, and the data collections in more detail.

Magento models don’t necessarily have to be related to any type table in the database or an EAV entity. Observers, which we will be reviewing later, are perfect examples of these type of Magento models.

Views

The view layer is one of the areas where Magento truly sets itself apart from other MVC applications. Unlike traditional MVC systems, Magento’s view layer is divided into three different components:

- **Layouts**: These are XML files that define block structures and properties, such as name and which template file to use. Each Magento module has its own set of layout files.
- **Blocks**: These are used in Magento to reduce the burden on the controller by moving most of the logic into blocks.
- **Templates**: These are PHTML files that contain the HTML code and PHP tags required.

Layouts give the Magento frontend an amazing amount of flexibility. Each module has its own layout XML files that tell Magento what to include and render on each page request. By using the layouts, we can move, add, or remove blocks from our store, without worrying about changing anything else other than our XML files.

Dissecting a layout file

Let’s examine one of the Magento core layout files, in this case, the catalog.xml file:

```xml
<layout version="0.1.0">
<default>
  <reference name="left"/>
```

Layout blocks comprise three main XML nodes:

- **Handle**: Each page request will have several unique handles. The layout uses these handles to tell Magento which blocks to load and render on a per page basis. The most commonly used handles are the default handle and the \([\text{frontname}]_.\_\_\_\_\_\_\_[\text{controller}]_.\_\_\_\_\_\_\_[\text{action}]\) handle.
- The default handle is especially useful to set global blocks, for example, adding CSS or JavaScript to all pages on the header block.
• **Reference**: A `<reference>` node is used to make references to a block. It is useful for the specification of nested blocks or modifying an already existing block. In our example, we can see how a new child block is being specified inside `<reference name="left">`.

• **Block**: The `<block>` node is used to load our actual blocks. Each block node can have the following properties:
  
  ° *type*: This is the identifier for the actual block class. For example, `catalog/product_list` makes reference to the Mage_Catalog_Block_Product_List.
  
  ° *name*: This is the name used by other blocks to make a reference to this block.
  
  ° *before/after*: These properties can be used to position the blocks relative to other block position. Both properties can use a hyphen as value to specify if the module should appear at the very top or the very bottom.
  
  ° *template*: This property determines the .phtml template file that will be used to render the block.
  
  ° *action*: Each block type has specific actions that affect the frontend functionality. For instance, the page/html_head block has actions to add CSS and js (addJs and addCss).
  
  ° *as*: This is used to specify the unique identifier that we will be using to call the block from the template. For example, calling a child block by using `getChildHtml('block_name')`.

Blocks are a new concept that Magento implements in order to reduce the controller load. They are basically data resources that communicate directly with the models that manipulate the data if needed and then pass it to the views.

Finally, we have our .phtml files. Templates contain HTML and PHP tags and are in charge of formatting and displaying the data from our models. Let’s take a look at a snippet from the product view template:

```
<div class="product-view"
...
<div class="product-name"
  
  <h1><?php echo $_helper->productAttribute($_product, $_product->getName(), 'name') ?></h1>
  
  </div>

...
Magento Fundamentals for Developers

```php
<?php echo $this->getReviewsSummaryHtml($_product, false, true)?>
<?php echo $this->getChildHtml('alert_urls') ?>
<?php echo $this->getChildHtml('product_type_data') ?>
<?php echo $this->getTierPriceHtml() ?>
<?php echo $this->getChildHtml('extrahint') ?>
...

<?php if ($_product->getShortDescription()):?>
    <div class="short-description">
        <h2><?php echo $this->__('Quick Overview') ?></h2>
        <div class="std">
            <?php echo $helper->productAttribute($_product, nl2br($_product->getShortDescription()), 'short_description') ?></div>
    </div>
<?php endif;?
...
</div>

The following is a diagram displaying the MVC model:

[Diagram of the MVC model]

---

[20]
Controllers
The Magento MVC controllers are designed to be thin controllers. Thin controllers have little business logic and are mostly used to drive the application requests. A basic Magento controller action will just load and render the layout:

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

From here, it is the job of the blocks to handle the *display logic* to get the data from our models, prepare the data, and send it to the views.

Websites and store scopes
One of Magento's core features is the ability to handle multiple websites and stores with a single Magento installation. Internally, Magento refers to each of these instances as scopes.
Values for certain elements such as products, categories, attributes, and configuration are scope-specific and can differ on different scopes. This gives Magento tremendous flexibility. For example, a product can be set up on two different websites with different prices but still share the rest of the attribute configuration.

As developers, one of the areas where we will be using scopes the most is when working with configuration. The different configuration scopes available in Magento are as follows:

- **Global**: As the name implies, this applies across all scopes.
- **Website**: These are defined by a domain name and are composed by one or more stores. Websites can be set up to share customer data or be completely isolated.
- **Store**: These are used to manage products and categories and to group store views. Stores also have a root category that allows us to have separated catalogs per store.
- **Store view**: By using store views, we can set up multiple languages on our store frontend.

Configuration options in Magento can store values on three scopes (global, website, and store views). By default, all the values are set on the global scope. Using system.xml on our modules, we can specify the scopes on which the configuration options can be set. Let’s revisit our previous system.xml file:

```xml
<field_name translate="label comment">
    <label>Enabled</label>
    <comment>
        <![CDATA[Comments can contain <strong>HTML</strong>]]>
    </comment>
    <frontend_type>select</frontend_type>
    <source_model>adminhtml/system_config_source_yesno</source_model>
    <sort_order>10</sort_order>
    <show_in_default>1</show_in_default>
    <show_in_website>1</show_in_website>
    <show_in_store>1</show_in_store>
</field_name>
```

...
Factory names and functions

Magento makes use of factory methods to instantiate models, helpers, and block classes. A factory method is a design pattern that allows us to instantiate an object without using the exact class name and using a class alias instead.

Magento implements the following factory methods:

- Mage::getModel()  
- Mage::getResourceModel()  
- Mage::helper()  
- Mage::getSingleton()  
- Mage::getResourceSingleton()  
- Mage::getResourceHelper()

Each of these methods takes a class alias that is used to determine the real class name of the object that we are trying to instantiate. For example, if we want to instantiate a product object, we can do so by calling the `getModel()` method:

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

Notice that we are passing a factory name composed of `group_classname/model_name`. Magento will resolve this to the actual class name of `Mage_Catalog_Model_Product`. Let's take a closer look at the inner workings of `getModel()`:

```
public static function getModel($modelClass = '', $arguments = array())
{
    return self::getConfig()->getModelInstance($modelClass, $arguments);
}
```

`getModel` calls the `getModelInstance` from the `Mage_Core_Model_Config` class.

```
public function getModelInstance($modelClass='', $constructArguments=array())
{
    $className = $this->getModelClassName($modelClass);
    if (class_exists($className)) {
        Varien_Profiler::start('CORE ::create_object_of::'.$className);
        $obj = new $className($constructArguments);
    }
```

In return, `getModelInstance()` calls the `getModelClassName()` method that takes a class alias as a parameter. Then, it tries to validate the existence of the returned class, and if the class exists, it creates a new instance of that class and returns it to our `getModel()` method:

```php
public function getModelClassName($modelClass) {
    $modelClass = trim($modelClass);
    if (strpos($modelClass, '/')===false) {
        return $modelClass;
    }
    return $this->getGroupedClassName('model', $modelClass);
}
```

The `getModelClassName()` method calls the `getGroupedClassName()` method, which is actually in charge of returning the real class name of our model.

The `getGroupedClassName()` method takes two parameters, namely `$groupType` and `$classId`. The `$groupType` parameter refers to the type of object that we are trying to instantiate. Currently, only models, blocks, and helpers are supported. The `$classId` that we are trying to instantiate is as follows:

```php
public function getGroupedClassName($groupType, $classId, $groupRootNode=null) {
    if (empty($groupRootNode)) {
        $groupRootNode = 'global/'.$groupType.'s';
    }
    $classArr = explode('/', trim($classId));
    $group = $classArr[0];
    $class = !empty($classArr[1]) ? $classArr[1] : null;
    
    if (isset($this->_classNameCache[$groupRootNode][$group][$class])) {
        return $this->_classNameCache[$groupRootNode][$group][$class];
    }
```
$config = $this->_xml->global->{($groupType.'s')->{($group)};
$className = null;
if (isset($config->rewrite->$class)) {
    $className = (string)$config->rewrite->$class;
} else {
    if (isset($config)->deprecatedNode) {
        $deprecatedNode = $config->deprecatedNode;
        $configOld = $this->_xml->global->{($groupType.'s')->{($deprecatedNode)};
        if (isset($configOld->rewrite->$class)) {
            $className = (string) $configOld->rewrite->$class;
        }
    }
}
if (empty($className)) {
    if (!empty($config)) {
        $className = $config->getClassName();
    }
    if (empty($className)) {
        $className = 'mage_'.$group.'_'.$groupType;
    }
    if (!empty($class)) {
        $className .= '_'.$class;
    }
    $className = uc_words($className);
}
$this->_classNameCache[$groupRootNode][$group][$class] = $className;
return $className;

As we can see, getGroupedClassName() is actually doing all the work. It grabs our class alias catalog/product and creates an array by exploding the string on the slash character.

Then, it loads an instance of Varien_Simplexml_Element and passes the first value in our array (group_classname). It also checks if the class has been rewritten, and if it has, we will use the corresponding group name.

Magento also uses a custom version of the uc_words() function that will capitalize the first letters and convert separators of the class alias if needed.
Finally, the function will return the real class name to the `getModelInstance()` function. In our example case, this will return `Mage_Catalog_Model_Product`.

**Events and observers**

The event and observer pattern is probably one of Magento's more interesting features, as it allows developers to extend Magento in critical parts of the application flow.

In order to provide more flexibility and facilitate interaction between the different modules, Magento implements an **Event/Observer** pattern. This pattern allows modules to be loosely coupled.
There are two parts to this system, an **Event** dispatch with the object and event information and an **Observer** listening to a particular event:

**Event dispatch**

Events are created or dispatched using the `Mage::dispatchEvent()` function. The core team has already created several events on critical parts of the core. For example, the Model abstract class `Mage_Core_Model_Abstract` calls two protected functions every time a model is saved: `_beforeSave()` and `_afterSave()` on each of these methods two event are fired.

```php
protected function _beforeSave()
{
    if (!$this->getId()) {
        $this->isObjectNew(true);
    }
    Mage::dispatchEvent('model_save_before',
        array('object'=>$this));
    Mage::dispatchEvent($this->_eventPrefix.'_save_before',
        $this->_getEventData());
    return $this;
}
```

---

[27]
protected function _afterSave()
{
    $this->cleanModelCache();
    Mage::dispatchEvent('model_save_after',
    array('object'=>$this));
    Mage::dispatchEvent($this->_eventPrefix.'_save_after',
    $this->_getEventData());
    return $this;
}

Each function fires a generic model_save_after event, and then a dynamic version based on the type of object being saved. This gives us a wide range of possibilities to manipulate objects through observers.

The Mage::dispatchEvent() method takes two parameters, the first is the event name and the second is an array of data that is received by the observer. We can pass values or objects in this array. This comes in handy if we want to manipulate the objects.

In order to understand the details of the event system, let's take a look at the dispatchEvent() method:

    public static function dispatchEvent($name, array $data = array())
    {
        $result = self::app()->dispatchEvent($name, $data);
        return $result;
    }

This function is actually an alias to the dispatchEvent() function inside the App core class, located in Mage_Core_Model_App:

    public function dispatchEvent($eventName, $args)
    {
        foreach ($this->_events as $area=>$events) {
            if (!isset($events[$eventName])) {
                $eventConfig = $this->getConfig()->getEventConfig(
                $area, $eventName);
                if (!$eventConfig)
                { $this->_events[$area][$eventName] = false;
                    continue;
                }
                $observers = array();
                foreach ($eventConfig->observers->children() as
                $obsName=>$obsConfig) {
                    $observers[$obsName] = array(  
                        'type' => (string)$obsConfig->type,  
                        'arguments' => array(  
                            ...
                        )
                    );
                }
                $result = $this->dispatchEvent($eventName, $data, $observers);
            } else {  
                $result = $this->dispatchEvent($eventName, $data);
            }
        }
        return $result;
    }

This method allows for more fine-grained control over the event system, with support for custom event arguments and observers.
'model' => $obsConfig->class ? (string)$obsConfig->class : $obsConfig->getClassName(),
'method' => (string)$obsConfig->method, 'args' => (array)$obsConfig->args,
);

$events[$eventName]['observers'] = $observers;$this->_events[$area][$eventName]['observers'] = $observers;

if (false===$events[$eventName]) {
    continue;
} else {
    $event = new Varien_Event($args);
    $event->setName($eventName);
    $observer = new Varien_Event_Observer();
    foreach ($events[$eventName]['observers'] as $obsName=>$obs) {
        $observer->setData(array('event'=>$event));
        Varien_Profiler::start('OBSERVER: '.$obsName);
        switch ($obs['type']) {
            case 'disabled':
                break;
            case 'object':
                case 'model':
                    $object = Mage::getModel($obs['model']);
                    $this->_callObserverMethod($object, $method, $observer);
                    break;
            default:
                $object = Mage::getSingleton($obs['model']);
                $this->_callObserverMethod($object, $method, $observer);
                break;
        }
        Varien_Profiler::stop('OBSERVER: '.$obsName);
    }
}
return $this;
The `dispatchEvent()` method actually does all the work on the Event/Observer model:

1. It gets the Magento configuration object.
2. Then, it walks through the observer's node children, checking if the defined observer is listening to the current event.
3. For each of the available observers, the dispatch event tries to instantiate the observer object.
4. Lastly, Magento tries to call the corresponding observer function mapped to this particular event.

**Observer bindings**

Now, dispatching an event is only part of the equation. We also need to tell Magento which observer is listening to each event. Not to our surprise, observers are specified through the `config.xml` file. As we saw before, the `dispatchEvent()` function queries the configuration object for available observers. Let's take a look at an example `config.xml` file:

```xml
<events>
  <event_name>
    <observers>
      <observer_identifier>
        <class>module_name/observer</class>
        <method>function_name</method>
      </observer_identifier>
    </observers>
  </event_name>
</events>
```

The event node can be specified in each of the configuration sections (admin, global, frontend, and so on) and we can specify multiple `event_name` children nodes. The `event_name` node has to match the event name used in the `dispatchEvent()` function.

Inside each `event_name` node, we have a single observer node that can contain multiple observers, each with a unique identifier.

Observer nodes have two properties, `<class>`, which points to our observer model class, and `<method>`, which points to the actual method inside the observer class. Let's analyze an example observer class definition:

```php
class Namespace_Modulename_Model_Observer
{
```
public function methodName(Varien_Event_Observer $observer)
{
    //some code
}
}

[ ] One interesting thing about observer models is that they don't extend to any other Magento class.

**Summary**

In this chapter, we covered many important and fundamental topics about Magento:

- Architecture
- Folder structure
- Routing system
- MVC patterns
- Events and observers
- Configuration scope

While this may seem overwhelming at first sight, it is just the tip of the iceberg. There is a lot more to learn about each of these topics and Magento. The purpose of this chapter is to make developers aware of all the important components of the platform, from the configuration object to the way the event/object pattern is implemented.

Magento is a powerful and flexible system and much more than an e-commerce platform. The core team has put a lot of effort in making Magento a powerful framework.

In later chapters, we will not only review all these concepts in more detail, but we will also apply them in a practical manner by building our own extensions.
Where to buy this book


Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.