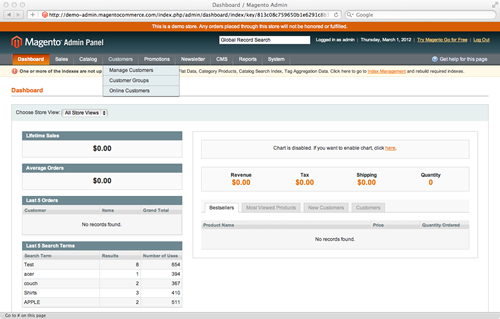
**[The Basics Of Creating A Magento Module](http://coding.smashingmagazine.com/2012/03/01/basics-creating-magento-module/" \o "Read 'The Basics Of Creating A Magento Module')**

A lot of community extensions (or modules) are available for the feature-rich open-source e-commerce solution [Magento](http://ampersandcommerce.com/what-is-magento/), but what if they don’t quite work as you want them to? What if you could understand the structure of a Magento module a little better, to the point that you could modify it to suit your needs or, better yet, write your own module from scratch?



In this tutorial, we will introduce the coding of Magento in the form of a “Hello World”-style module. The goal of the module will be simply to write some information to a log file every time a product is saved. This very basic module will allow us to cover a number of interesting topics, including:

* The app/code directories,
* The structure and creation of a Magento module,
* Event observers,
* Logging.

### Before We Begin

This tutorial assumes that you already have an installation of Magento up and running, either locally or on a development server, that you can add new files to. The version of Magento that you use doesn’t really matter, because we will be covering fundamental aspects that exist across all versions and editions: Community, Professional and Enterprise.

#### Disable the Cache

This is one of the first lessons a Magento developer should learn: disable the cache! You can do this by going to Admin Panel > System > Cache Management > Select All > Actions: Disable > Submit.

While very good at boosting performance in a production environment, the cache is a developer’s enemy. Leave it enabled at your peril! Every Magento developer I have met has on more than one occasion spent an hour or so wondering why their latest update is not showing up, only to find that Magento is still displaying the version of the website that it conveniently cached earlier that day.

### The app/code Directory

The brains of Magento can be found in individual modules inside the app/code directory, which is split in to three areas: core, community and local.

#### Core

The app/code/core directory contains all of the functionality for products, categories, customers, payments, etc. Until you know what you are doing (and even afterwards), keep app/code/core off limits because these files **should not be modified**.

Magento is structured in such a way that you can alter the functionality of any of these core files without modifying them directly, which ensures that your application remains upgrade-proof. By all means, look in order to better understand how Magento works, but do not touch.

#### Community

As the name suggests, app/code/community is where you will find modules that have been provided by third parties (i.e. not Magento’s core team). Hundreds of modules are available through Magento Connect, and when you install them through the built-in “Package Manager,” this is where they end up.

#### Local

Magento ships with an empty app/code/local directory, ready for you to add bespoke modules for your own Magento installation. This is where we will be working for the duration of this tutorial.

### Structuring Our Directory

Open your favorite editor, and navigate to app/code/local to add some new directories and files.

#### Module Namespace

The first directory we will create is a “namespace.” This can be called anything you like, but the convention is some form of the name of the company or module’s author. Magento uses “Mage” as its namespace. Here at Ampersand Commerce, we use “Ampersand.” For this tutorial, we will use “SmashingMagazine” as our namespace. So, create the directory app/code/local/SmashingMagazine.

#### Module Name

For the next directory, we will give our module a descriptive name. The module we are creating will write log entries each time a product is saved, so a logical name would be LogProductUpdate. Create the directory app/code/local/SmashingMagazine/LogProductUpdate.

We should now have the following directory structure for our module. These directory and file names are **case-sensitive**, so capitalize where appropriate.

app

- code

- local

- SmashingMagazine

- LogProductUpdate

### Configuring Our Module

Next, we will begin to configure our module. The configuration files belong inside our module in a directory named etc, so let’s create that along with a new XML file: app/code/local/SmashingMagazine/LogProductUpdate/etc/config.xml. This XML file will inform Magento of the location of the files in our module, as well as many other things, such as version number and events to observe. For now, we will create a simple config.xml file, which contains comments that explain the meaning of each section.

<?xml version="1.0" encoding="UTF-8"?>

<!-- The root node for Magento module configuration -->

<config>

<!--

The module's node contains basic

information about each Magento module

-->

<modules>

<!--

This must exactly match the namespace and module's folder

names, with directory separators replaced by underscores

-->

<SmashingMagazine\_LogProductUpdate>

<!-- The version of our module, starting at 0.0.1 -->

<version>0.0.1</version>

</SmashingMagazine\_LogProductUpdate>

</modules>

</config>

### Activating Our Module

The next step is to inform our Magento installation that our module exists, which we do by creating a new XML file in app/etc/modules. The name of this XML file can be anything you like, since Magento will read all XML files in this directory and will be interested only in the content. However, by convention we should give the file and module the same name. Let’s create app/etc/modules/SmashingMagazine\_LogProductUpdate.xml with the following content:

<?xml version="1.0" encoding="UTF-8"?>

<config>

<modules>

<SmashingMagazine\_LogProductUpdate>

<!-- Whether our module is active: true or false -->

<active>true</active>

<!-- Which code pool to use: core, community or local -->

<codePool>local</codePool>

</SmashingMagazine\_LogProductUpdate>

</modules>

</config>

### Sanity Check: Is The Module Enabled?

We now have a fully functional module that is enabled in Magento. It doesn’t do anything, but it is a valid module. This is our first opportunity to see whether we have correctly configured everything so far. If we log into the Magento admin panel and navigate to System > Configuration > Advanced > Advanced and view the “Disable Modules Output” listing, we should see our SmashingMagazine\_LogProductUpdate module listed as enabled. If it is not listed, then something has gone wrong, so carefully run through the steps up to this point again. This is usually when new Magento developers discover the cache!

Our module’s structure now looks like this:

app

- code

- local

- SmashingMagazine

- LogProductUpdate

- etc

- config.xml

- etc

- modules

- SmashingMagazine\_LogProductUpdate.xml

### Defining An Event Observer

Event observers are extremely powerful and are one of the cleanest ways to extend Magento’s functionality without having to rewrite or override any core methods or classes. We want to **observe the event that Magento dispatches just after a product is saved**, so the code for the event we are interested in is catalog\_product\_save\_after. Determining which event code to use when defining a new observer requires a basic understanding of Magento’s model layer, which is beyond the scope of this tutorial. Don’t worry, though: we’ll cover it another time!

We now need to modify our config.xml to include the event observer definition:

<?xml version="1.0" encoding="UTF-8"?>

<config>

<modules>

<SmashingMagazine\_LogProductUpdate>

<version>0.0.1</version>

</SmashingMagazine\_LogProductUpdate>

</modules>

<!-- Configure our module's behavior in the global scope -->

<global>

<!-- Defining an event observer -->

<events>

<!-- The code of the event we want to observe -->

<catalog\_product\_save\_after>

<!-- Defining an observer for this event -->

<observers>

<!--

Unique identifier within the

catalog\_product\_save\_after node.

By convention, we write the module's

name in lowercase.

-->

<smashingmagazine\_logproductupdate>

<!-- The model to be instantiated -->

<class>smashingmagazine\_logproductupdate/observer</class>

<!-- The method of the class to be called -->

<method>logUpdate</method>

<!-- The type of class to instantiate -->

<type>singleton</type>

</smashingmagazine\_logproductupdate >

</observers>

</catalog\_product\_save\_after>

</events>

</global>

</config>

### Configuring Our Model’s Directory

In the event observer defined above, we made reference to a model that we have not yet created. We need to inform Magento where to find models in our module by updating config.xml with the following:

<?xml version="1.0" encoding="UTF-8"?>

<config>

<modules>

<SmashingMagazine\_LogProductUpdate>

<version>0.0.1</version>

</SmashingMagazine\_LogProductUpdate>

</modules>

<!-- Configure our module's behavior in the global scope -->

<global>

<!-- Defining models -->

<models>

<!--

Unique identifier in the model's node.

By convention, we put the module's name in lowercase.

-->

<smashingmagazine\_logproductupdate>

<!--

The path to our models directory, with directory

separators replaced by underscores

-->

<class>SmashingMagazine\_LogProductUpdate\_Model</class>

</smashingmagazine\_logproductupdate>

</models>

<events>

<catalog\_product\_save\_after>

<observers>

<smashingmagazine\_logproductupdate>

<class>smashingmagazine\_logproductupdate/observer</class>

<method>logUpdate</method>

<type>singleton</type>

</smashingmagazine\_logproductupdate >

</observers>

</catalog\_product\_save\_after>

</events>

</global>

</config>

### Creating An Observer Model

We will now create the model to be instantiated when the event is dispatched. Create a new PHP file in app/code/local/SmashingMagazine/LogProductUpdate/Model/Observer.php with the following content:

<?php

/\*\*

\* Our class name should follow the directory structure of

\* our Observer.php model, starting from the namespace,

\* replacing directory separators with underscores.

\* i.e. app/code/local/SmashingMagazine/

\* LogProductUpdate/Model/Observer.php

\*/

class SmashingMagazine\_LogProductUpdate\_Model\_Observer

{

/\*\*

\* Magento passes a Varien\_Event\_Observer object as

\* the first parameter of dispatched events.

\*/

public function logUpdate(Varien\_Event\_Observer $observer)

{

// Retrieve the product being updated from the event observer

$product = $observer->getEvent()->getProduct();

// Write a new line to var/log/product-updates.log

$name = $product->getName();

$sku = $product->getSku();

Mage::log(

"{$name} ({$sku}) updated",

null,

'product-updates.log'

);

}

}

### We’re done! Try it out.

The directory structure for our completed module should now look like this:

app

- code

- local

- SmashingMagazine

- LogProductUpdate

- Model

- Observer.php

- etc

- config.xml

- etc

- modules

- SmashingMagazine\_LogProductUpdate.xml

Now that our module is complete, it’s time to try it out! Log into the Magento admin panel, create or update a product in your catalog, and then check the var/log folder to see your product-updates.log file populated.

If nothing appears or the directory does not exist, ensure that the correct permissions are set to allow Magento to write to this directory, and that logging is enabled in Admin Panel > System > Configuration > Developer > Log Settings > Enabled.

This basic tutorial is meant to give you an overall understanding of how Magento modules work. After completing this tutorial, spend some time exploring the Magento modules in app/code/core and see if you now have a better idea of how it all works.

Feel free to [download the source code](http://ampersandcommerce.com/assets/SmashingMagazine_LogProductUpdate.zip) (ZIP).

We welcome any questions and would love to hear any feedback in the comments area below.