A Developer's Guide to Magento Backend

Publisher- Silver Touch Technologies Ltd.

Editor- Deepa Ranganathan

© Copyright 2017 Silver Touch Technologies Ltd.

All rights reserved. No part of this Guide shall be reproduced, stored in a retrieval system, or transmitted by any means, electronic, mechanical, photocopying, recording, or otherwise, without written permission from Silver Touch Technologies Ltd.

Note : Please take a backup of your existing Magento website before working on these codes given in this e-book

Contents

Introduction

Magento is a developer's favorite e-commerce platform, as it not just allows them to create solutions designed to meet the business needs, but also offers flexibility and scalability. Magento's growing popularity can be attributed to the performance, community support and easy integration features it offers. This platform has been specifically designed for e-commerce, and that's the reason it offers many functionalities and features.

As a developer, many of you want to customize the backend to make things manageable and easy for you. If you want to customize the core, or even want to manage your extensions from the core, you can do so by tweaking the available code a bit. With the backend customized, you will be able to design the site to suit your e-store needs.

How to Customize the Core of your Magento Site in a Few Steps

Do you want to make some relevant changes to the core of Magento Installation? While it is possible, you will need to take extra care of what you are changing. You should ensure that no extra changes are made; else it will change the basics of your website.

Few basic things that you should check on before changing the core would be

i. You can make relevant changes later

ii. Nothing can be overwritten

iii. You can customize your core once again whenever the need arises

The Configuration Files

Magento merges different XML files into one XML file which is both global and cached. With this you can easily edit all the core files manually and customize to your needs.

Below is the code using which two XML configuration files will be combined to form a single XML file which is both cached and global.

i. First Configuration File

```
<?xml version="1.0"?>
<config>
  <node1>
    <node1_1>Data1</node1_1>
  </node1>
  <node2>Data2</node2>
</config>
```

ii. Second Configuration File

```
<?xml version="1.0"?>
<config>
  <node1>
    <node1_2>Data3</node1_2>
  </node1>
  <node2>Data4</node2>
  <node3>Data5</node3>
</config>
```

iii. Combination of the Two Configuration Files

```
<?xml version="1.0"?>
<config>
  <node1>
    <node1_1>Data1</node1_1>
    <node1_2>Data3</node1_2>
  </node1>
```

```
  <node2>Data4</node2>
  <node3>Data5</node3>
</config>
```

Files Used for Configuration

Before beginning with the configuration, initialize the system with init(). You can read about how to use this function by following this path: app/code/core/Mage/core/Model/Config.php

Let's say config cache for Magento core is enabled, and it is loaded successfully to your system when you open it using the above mentioned path, then you can easily skip the below mentioned steps. Else you will need to manually load it to your system. Here are the steps you can use to manually load

- First load app/etc/config.xml. If you want to successfully load all the elements present in the Magento core, you will need to make sure this file is loaded
- Now merge all the files present in app/etc/modules/(app/etc/modules.xml till you reach 0.6.14100). if there are any modules declarations they are removed by the core or community or custom installations. They are then named using Package name (Mage_All.xml)
- Once you have installed, it is time to merge them using Merge app/etc/local.xml. This file has all the settings for the local database along with installation date and the encryption key
- In case you are unable to locate the local file, app/etc/distro.xml will get merged. This contains all the auto values needed for installation
- Finally merge config.xml using the different modules declared earlier in app/etc/modules/*. The modules declared include: app/code/{codepool}/{companyName}/etc/config.xml

Defining the Model

Model is a set of specific classes that offer resource specific logic. It is important to know how to define or create a new model in Magento core

Let's begin with Sample_module

```
<?xml version="1.0"?>
<config>
  <modules>
    <Sample_Module>
      <codePool>local</codePool>
      <active>true</active>
    </Sample_Module>
  </modules>
</config>
```

Once you have defined the module, you will need to create the config file:
app/code/local/Test/Module/etc/config.xml

Here's the code that will help create the config xml

```
<?xml version="1.0"?>
<config>
  <global>
    <models>
```

```
<sample_mod>
  <class>Sample_Module_Model</class>
</sample_mod>
```

Using Mage function and the above code, you will be returned the value Sample Module

```
Mage::getModel('sample_mod/test_model')
```

The value returned will actually be located in the following path

```
app/code/local/Sample/Module/Model/Test/Model.php
```

Along with test_model, Sample_Module_Model is able to generate the required class for customization

Customizing the Core

Customizing the core actually means changing the functionalities to suit your needs without really changing the basics of the file. A module sample_module has been created. Now let's override this with sample_custom to understand how core files are customized without really disturbing the functionalities.

You will define the new model in the same way as you will define the sample_module

You need to ensure that the application does not break and for this, you will need to offer all classes within the original inside the custom module.

```
<?xml version="1.0"?>
<config>
  <global>
    <models>
      <example_mod>
        <class>Example_Custom_Model</class>
      </example_mod>
```

Overriding a single class to understand customization

```
<?xml version="1.0"?>
<config>
  <global>
    <models>
      <example_mod>
        <rewrite>
          <test_model>Example_Custom_Model_Test_Model</test_model>
        </rewrite>
      </example_mod>
```

Post customization, app/code/local/Example/Custom/Model/Test/Model.php will appear as follows

```
<?php
        class Example_Custom_Model_Test_Model extends Sample_Module_Model_Test_Model
        {
                public function exampleMethod()  {    return 'custom result';  }
        }
```

Now in the code for $obj= Mage::getModel('example_mod/test_model')

The value returned would be

Example_Custom_Model_Test_Model which will include functionalities present within Sample_Module_Model_Test_Model along with some extended functionalities that were added

Setting up Helpers

Helpers are nothing but support classes. They help in manipulating the data and rendering them as you want

You can make the changes in *.phtml templates or *.xml files

Apart from manipulations you can also use helpers to cache your website

```
$this->helper('wishlist'); // wishlist module; data.php helper
$this->helper('catalog/image'); //catalog module; image.php helper
$this->helper(); //current module; data.php helper
```

Setting up Admin Menu

It is important to set up your admin menu when customizing the Magento core.

You will find the setup in app/code/core/Mage/Adminhtml/etc/config.xml

You can copy paste the xml file to this and then carry out customization. You can even add it to any other configuration file and merge the two files as mentioned earlier.

How to Use Magento Settings to Manage Extension Settings

Whenever you want to make small changes within a particular module or extension in Magento, you have to locate the module configuration tab, and make the necessary changes there. Locating, managing, and making the changes is a tedious process, which you can avoid if you transfer the necessary settings to the main Magento settings. So, you just need to make the modifications within the main settings file, and all the changes are registered.

Create System.xml File

Begin the whole process by creating system.xml file in the etc folder of Magento.

```
<config>
  <sections>
    <pos module="pos">
      <label>Example Pos Settings</label>
      <tab>service</tab>
      <frontend_type>text</frontend_type>
      <show_in_default>1</show_in_default>
      <show_in_website>1</show_in_website>
      <show_in_store>1</show_in_store>
      <sort_order>10</sort_order>
      <groups>
        <settings translate="label">
          <label>example Pos Settings</label>
          <frontend_type>text</frontend_type>
          <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>
          <fields>
            <sanalpos_url translate="label">
              <label>Service Url</label>
              <frontend_type>text</frontend_type>
              <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>
            </sanalpos_url>
            <user_id translate="label">
              <label>User Id</label>
              <frontend_type>text</frontend_type>
              <sort_order>20</sort_order>
              <show_in_default>1</show_in_default>
              <show_in_website>1</show_in_website>
              <show_in_store>1</show_in_store>
            </user_id>
          </fields>
        </settings>
      </groups>
    </pos>
  </sections>
</config>
```

Integrate to Config.xml File

This file contains all the field settings required to modify the extension you have created. While Magento is definitely open source, you don't want to make all the settings and code public. So, you will need to integrate the individual settings to the core settings page. Paste the following code to the config.xml file

```
<config>
   <!-- some more code here -->
   <adminhtml>
      <acl>
         <resources>
            <all>
               <title>Allow Everything</title>
            </all>
            <admin>
               <children>
                  <system>
                  <children>
                        <config>
                           <children>
                              <pos>
                                 <title>Kartaca Pos Settings</title>
                                 <sort_order>10</sort_order>
                              </pos>
                           </children>
                        </config>
                     </children>
                  </system>
               </children>
            </admin>
         </resources>
      </acl>
   </adminhtml>
   <!-- some more code here... -->
</config>
   </sections>
</config>
```

With this long xml code, you would enjoy not having to search through individual files and changing the codes there. Saves a lot of time!

How to Create Custom Grid on your Magento Website

There is a default grid for product display available with Magento store. What if you want to customize the grid and create a new layout, with some add-ons? Is it possible? Yes, of course you can create a custom grid, provided you can code your way into it.

Here is how coding will get you to deliver a custom grid for the e-Store

Let's say you want to change the grid layout for your orders grid.

Let's begin with creating a new module

Module Declaration

Create Sttl_Custom.xml inside /app/etc/modules

```
<?xml version="1.0"?>
<config>
        <modules>
                <Sttl_Custom>
                        <active>true</active>
                        <codePool>local</codePool>
                </Sttl_Custom>
        </modules>
</config>
```

Once you have declare a new module, it is time to create a new folder where you can create folders like block, controller, helper and other such elements.

Go to /app/code/local/Sttl/Custom,

Here you can create sub-folders like controller, helper, block etc.

Module configuration,

create config.xml inside /app/code/local/Sttl/Custom/etc

```
<?xml version="1.0"?>
<config>
  <modules>
    <Sttl_Custom>
      <version>0.1.0</version>
    </Sttl_Custom>
  </modules>
  <global>
    <models>
      <sttlcustom>
        <class>Sttl_Custom_Model</class>
      </sttlcustom>
    </models>
    <blocks>
      <sttlcustom>
```

```
            <class>Sttl_Custom_Block</class>
        </sttlcustom>
    </blocks>
    <helpers>
        <sttlcustom>
            <class>Sttl_Custom_Helper</class>
        </sttlcustom>
    </helpers>
  </global>
  <admin>
    <routers>
        <adminhtml>
            <args>
                <modules>
                    <sttlcustom before="Mage_Adminhtml">Sttl_Custom_Adminhtml</sttlcustom>
                </modules>
            </args>
        </adminhtml>
    </routers>
  </admin>
</config>
```

Create adminhtml.xml

Go to /app/code/local/Sttl/Custom/etc

```
<?xml version="1.0"?>
<config>
  <menu>
    <sales>
        <children>
            <sttlcustom translate="title" module="sttlcustom">
                <sort_order>10</sort_order>
                <title>Orders - Example</title>
                <action>adminhtml/order/</action>
            </sttlcustom>
        </children>
    </sales>
  </menu>
</config>
```

Helper

Once this is done, you will need to create a blank helper class and a controller grid to further actions

Create Data.php inside /app/code/local/Sttl/Custom/Helper

Blank Helper Class

```
<?php
        class Sttl_Custom_Helper_Data extends Mage_Core_Helper_Abstract
        {
        }
```

Controllers

Create OrderController.php inside /app/code/local/Sttl/Custom/controllers/Adminhtml

```
<?php
class Sttl_Custom_Adminhtml_OrderController extends Mage_Adminhtml_Controller_Action
{
    public function indexAction()
    {
        $this->_title($this->__('Sales'))->_title($this->__('Orders - Example'));
        $this->loadLayout();
        $this->_setActiveMenu('sales/sales');
        $this->_addContent($this->getLayout()->createBlock('sttlcustom/adminhtml_sales_order'));
        $this->renderLayout();
    }

    public function gridAction()
    {
        $this->loadLayout();
        $this->getResponse()->setBody(
            $this->getLayout()->createBlock('sttlcustom/adminhtml_sales_order_grid')->toHtml()
        );
    }

    public function exportCsvAction()
    {
        $fileName = 'orders.csv';
        $grid = $this->getLayout()->createBlock('sttlcustom/adminhtml_sales_order_grid');
        $this->_prepareDownloadResponse($fileName, $grid->getCsvFile());
    }

    public function exportExcelAction()
    {
        $fileName = 'orders.xml';
        $grid = $this->getLayout()->createBlock('sttlcustom/adminhtml_sales_order_grid');
        $this->_prepareDownloadResponse($fileName, $grid->getExcelFile($fileName));
    }
}
```

Create Grid Container

Create Order.php inside /app/code/local/Sttl/Custom/Block/Adminhtml/Sales

```
<?php
class Sttl_Custom_Block_Adminhtml_Sales_Order extends Mage_Adminhtml_Block_Widget_Grid_Container
{
    public function __construct()
    {
        $this->_blockGroup = 'sttlcustom';
        $this->_controller = 'adminhtml_sales_order';
        $this->_headerText = Mage::helper('sttlcustom')->__('Orders - Example');
        parent::__construct();
        $this->_removeButton('add');
    }
}
```

Create Grid Class

Create Grid.php inside /app/code/local/Sttl/Custom/Block/Adminhtml/Sales/Order

```
<?php
class Sttl_Custom_Block_Adminhtml_Sales_Order_Grid extends Mage_Adminhtml_Block_Widget_Grid
{
    public function __construct()
    {
        parent::__construct();
        $this->setId('tmporder_grid');
        $this->setDefaultSort('increment_id');
        $this->setDefaultDir('DESC');
        $this->setSaveParametersInSession(true);
        $this->setUseAjax(true);
    }

    protected function _prepareCollection()
    {
        $collection = Mage::getResourceModel('sales/order_collection')
            ->join(array('a' => 'sales/order_address'), 'main_table.entity_id = a.parent_id AND a.address_type !=
\'billing\'', array(
                'city'       => 'city',
                'country_id' => 'country_id'
            ))
            ->join(array('c' => 'customer/customer_group'), 'main_table.customer_group_id = c.customer_group_id',
array(
                'customer_group_code' => 'customer_group_code'
            ))
            ->addExpressionFieldToSelect(
                'fullname',
                'CONCAT({{customer_firstname}}, \' \', {{customer_lastname}})',
                array('customer_firstname' => 'main_table.customer_firstname', 'customer_lastname' =>
'main_table.customer_lastname'))
            ->addExpressionFieldToSelect(
                'products',
                '(SELECT GROUP_CONCAT(\' \', x.name)
                    FROM sales_flat_order_item x
                    WHERE {{entity_id}} = x.order_id
                        AND x.product_type != \'configurable\')',
                array('entity_id' => 'main_table.entity_id')
            )
        ;
         $this->setCollection($collection);
        parent::_prepareCollection();
        return $this;
    }
     protected function _prepareColumns()
    {
        $helper = Mage::helper('sttlcustom');
        $currency = (string) Mage::getStoreConfig(Mage_Directory_Model_Currency::XML_PATH_CURRENCY_BASE);
         $this->addColumn('increment_id', array(
            'header' => $helper->__('Order #'),
            'index'  => 'increment_id'
        ));
         $this->addColumn('purchased_on', array(
            'header' => $helper->__('Purchased On'),
            'type'   => 'datetime',
            'index'  => 'created_at'
```

```
        ));
        $this->addColumn('products', array(
            'header'       => $helper->__('Products Purchased'),
            'index'        => 'products',
            'filter_index' => '(SELECT GROUP_CONCAT(\' \', x.name) FROM sales_flat_order_item x WHERE
main_table.entity_id = x.order_id AND x.product_type != \'configurable\')'
        ));
        $this->addColumn('fullname', array(
            'header'       => $helper->__('Name'),
            'index'        => 'fullname',
            'filter_index' => 'CONCAT(customer_firstname, \' \', customer_lastname)'
        ));
        $this->addColumn('city', array(
            'header' => $helper->__('City'),
            'index'  => 'city'
        ));
        $this->addColumn('country', array(
            'header'   => $helper->__('Country'),
            'index'    => 'country_id',
            'renderer' => 'adminhtml/widget_grid_column_renderer_country'
        ));
        $this->addColumn('customer_group', array(
            'header' => $helper->__('Customer Group'),
            'index'  => 'customer_group_code'
        ));
        $this->addColumn('grand_total', array(
            'header'        => $helper->__('Grand Total'),
            'index'         => 'grand_total',
            'type'          => 'currency',
            'currency_code' => $currency
        ));
        $this->addColumn('shipping_method', array(
            'header' => $helper->__('Shipping Method'),
            'index'  => 'shipping_description'
        ));
        $this->addColumn('order_status', array(
            'header'  => $helper->__('Status'),
            'index'   => 'status',
            'type'    => 'options',
            'options' => Mage::getSingleton('sales/order_config')->getStatuses(),
        ));
        $this->addExportType('*/*/exportCsv', $helper->__('CSV'));
        $this->addExportType('*/*/exportExcel', $helper->__('Excel XML'));
        return parent::_prepareColumns();
    }
    public function getGridUrl()
    {
        return $this->getUrl('*/*/grid', array('_current'=>true));
    }
}
```

You can easily access the custom grid by visiting the admin panel and navigating to Sales>Orders-Example

How to Add Featured Products to your Magento Store Frontend

Magento Product Type

1) Simple Product

A simple product is a physical item with a single SKU. Simple products have a variety of pricing and of input controls which makes it possible to sell variations of the product. Simple products can be used in association with grouped, bundle, and configurable products.

2) Grouped Product

A grouped product presents multiple, standalone products as a group. You can offer variations of a single product, or group them for a promotion. The products can be purchased separately, or as a group.

3) Configurable Product

A configurable product appears to be a single product with lists of options for each variation. However, each option represents a separate, simple product with a distinct SKU, which makes it possible to track inventory for each variation.

4) Virtual Product

Virtual products do not have a physical presence, and are typically used for such things as services, warranties, and subscriptions. Virtual products can be used in association with grouped and bundle products.

5) Bundle Product

A bundle product let customers "build their own" from an assortment of options. The bundle could be a gift basket, computer, or anything else that can be customized. Each item in the bundle is a separate, standalone product.

6) Downloadable

A digitally downloadable product that consists of one or more files that are downloaded. The files can reside on your server or be provided as URLs to any other server.

Steps to Add Featured Products

Step 1: Create "is_featured" product attribute

Go to Catalog>Attributes>Manage Attributes>Add New Attribute

Here you can define the attribute properties for the new featured attribute that you are creating. Here are some default definitions

Attribute Properties

Attribute Code: is_featured

Scope: Store View (you want to enable this attribute only for front end)

Catalog input type for store owner: yes/no

Default Value: no

Unique Value: No (As the same material might be used for several other products)

Values Required: No (In case the material for the different products are different)

Input Validation for Store Owner: None (if no validation is required select none)

Apply to: All Product Types

Frontend Properties

Use in Quick Search: No

Use in Quick/Advanced Search: Yes (By selecting yes, you are enabling faster search on the front end for this attribute)

Comparable on Front End: Yes (So that two products with similar value can be compared)

Use in Layered Navigation: No

Visible on Catalog Pages along the front end: Yes

Settings for Manage Label/Options

Default: Featured Product

English: Featured Product

Once you have made changes in all the above mentioned settings, save the new attribute

Go to Catalog>Attributes>Manage Attribute Sets, and add the newly created attribute to Default attribute set.

Step 2: Creating a New Block Class

Once you have created the new featured attribute, the next step is to create a new block class which will feature the featured product

Go to app/code/local/Sttl/Custom/Block and create Featured.php block

```
<?php
class Sttl_Custom_Block_Featured extends Mage_Catalog_Block_Product_Abstract
{
        public function gteFeaturedProducts()    {
                $products = Mage::getModel('catalog/product')->getCollection()
                        ->addAttributeToFilter('is_featured', array(1))
                        ->addAttributeToFilter('status', array(1));
                return $products;                    }
}
```

Step 3: Reverberate Featured Products as HTML

Go to app/design/frontend/default/default/template/sttl/custom/ and create featured.phtml

```
<?php $featured_products = $this->gteFeaturedProducts();?>

<?php shuffle($featured_products); ?>

<div class="box recently" style="padding-left:15px; padding-right:15px;">

        <h3><?php echo $this->__('Featured Products') ?></h3>

        <div class="listing-type-grid  catalog-listing">

                <?php $_collectionSize = count($featured_products) ?>

                <table cellspacing="0" class="recently-list" id="product-list-table">

                        <?php $i=0;foreach ($featured_products as $_product): ?>

                                <?php if ($i++%3==0): ?><tr><?php endif ?>

                                        <td>
                                                <div>
                                                        <a href="<?php echo $_product->getProductUrl() ?>"
title="<?php echo $this->htmlEscape($_product->getName()) ?>">
                                                                <img class="product-image" src="<?php echo $this-
>helper('catalog/image')->init($_product, 'small_image')->resize(135, 135); ?>" width="135" height="135" alt="<?php
echo $this->htmlEscape($_product->getName()) ?>" />
                                                        </a>
                                                </div>

                                                <p>
                                                        <a class="product-name" href="<?php echo $_product-
>getProductUrl() ?>" title="<?php echo $this->htmlEscape($_product->getName()) ?>)"><?php echo $this-
>htmlEscape($_product->getName()) ?></a>
                                                </p>

                                                <?php echo $this->getReviewsSummaryHtml($_product, 'short') ?>

                                                </td>

                                <?php if ($i%3==0 && $i!=$_collectionSize): ?></tr><?php endif ?>

                        <?php endforeach ?>

                        <?php for($i;$i%3!=0;$i++): ?>

                                <td class="empty-product"> </td>

                        <?php endfor ?>

                        <?php if ($i%3==0): ?> <?php endif ?>

                </table>

                <script type="text/javascript">decorateTable('product-list-table')</script>

        </div>

</div>
```

Step 4: Add Featured Products to Frontend

Now place the featured product box to the frontend. To do this

Go to CMS>Manage Pages>Homepage

Place the following code at this location

{{block type="catalog/product_featured" name="product_featured" as="product_featured" template="catalog/product/featured.phtml}}

The featured products are added to the front end. Now, enable a good interface and let out an excellent experience to your customers

Conclusion

Customizing the backend would help you give your e-store the desired admin panel, and manageable core. This way whenever you need to insist on some change in the coding, you don’t need to change all the codes available on the website, thus making a chaos of the codes. Instead, you just need to take into account the block/controller that you are looking to change, and modify it accordingly. You not just save your website from crashing down unnecessarily, but also save a lot of time and efforts that go into modifying a huge code. Adding blocks, products, categories or even segregating and filtering the products is easy when you have created the backend as per your project needs.

THANKS FOR READING

Silver Touch™
TECHNOLOGIES

To learn more, Visit us on the web at

www.silvertouch.com

Email : info@silvertouch.com

© 2017Silver Touch Technologies LTD. All Rights Reserved