Exercise Solutions

Section 4 Lesson 2. Blocks

Exercise 1: Create a New Page with a Custom Block

Use the `Training_Practice` module for these exercises.

Step 1. Register and create a controller class

a) Refer to the exercise solutions for Section 3, Lesson 4, Exercise 1, Steps 3–5 for instructions regarding action controller class names and file location, as well as the syntax for modifying the response object.

b) Use the front name `render`, call the controller `BlockController`, and call the test action `testAction()`.

Step 2. Verify that the controller class works

- Invoke the controller by visiting the appropriate URL based on your front name, controller path, and action name (`/render/block/test`).

Step 3. Register a block class group

a) Refer to the exercise solutions for Section 3, Lesson 5, Exercise 1, Step 2 for an example of the XML structure to configure your class group and class prefix.

b) Use `training_practice` as the block class group.

Step 4. Create a block class extending from `Mage_Core_Model_Abstract`

- Based on the class prefix, create the appropriate folder structure to contain your class definition: `Training_Practice_Block`. Create the block class `Training_Practice_Block_Example` in the appropriate file *Training/Practice/Block/Example.php*.

```
<?php

class Training_Practice_Block_Example
      extends Mage_Core_Block_Abstract
{
}
```

Step 5. Override the `_toHtml()` method and return the string "HELLO WORLD"

- In the block class that was just created, override the abstract class '`_toHtml()` method in order to have the block return content as expected.

```
  protected function _toHtml()
  {
      return "HELLO WORLD";
  }
```

Step 6. Add the block output to the response body

a) In a new action `customBlockAction()` in your action controller class, use the block factory method to create an instance of the block class using the appropriate class group and class ID.

```
  public function customBlockAction()
  {
    $block = $this->getLayout()
        ->createBlock('training_practice/example');
  }
```


b) After the block instance has been created, use the instance's public rendering method to obtain the block's output and assign it to the response object.

```
public function customBlockAction()
{
    $block = $this->getLayout()
       ->createBlock('training_practice/example');

    $this->getResponse()->setBody($block->toHtml());
}
```

c) Visit the appropriate route based on your front name, controller path, and action (i.e., `/render/block/customBlock`). You should see the string "HELLO WORLD".

Exercise 2: Use the `core/template` Block to Display a Template for a Page

Use the `Training_Practice` module for these exercises.

Step 1. Create a new action

- Add a new action `templateBlockAction()` to your `RenderController` class.

Step 2. Create a block instance of `core/template`

- Add the following code to your action.

```
public function templateBlockAction()
{
  $block = $this->getLayout()->createBlock('core/template');
}
```

Step 3. Assign a template

- Assign a template to the `core/template` block.

```
$block->setTemplate('training/practice/example.phtml');
```


Step 4. Add the block output to the response body

- The full action method should look like this.

```
public function templateBlockAction()
{
  $block = $this->getLayout()->createBlock('core/template');
  $block->setTemplate('training/practice/example.phtml');
  $this->getResponse()->setBody($block->toHtml());
}
```

Step 5. Create the template with the output

a) Create the file *app/design/frontend/base/default/template/training/practice/example.phtml*.

b) Add the following content.

```
<h1><?php echo $this->getTemplateFile() ?></h1>
```

c) Visit the */render/block/templateBlock* path to see the result, which will be the full path within Magento to the template file you created.

Exercise 3: Pass a Parameter from the Controller to a Block and Output Its Value in a Template

Use the `Training_Practice` module for these exercises.

Step 1. In a controller action, register a parameter in the Mage registry

- Create a new action in the action controller class from the previous exercise and register a registry key with some string content. We'll add to this definition later.

```
public function registryAction()
{
    Mage::register('SOME_VALUE','Some custom string');
}
```


Step 2. Create a new block class extending `core/template` with a getter method that returns that value from the registry

a) Create a new file named *Registry.php* under your *Block* folder (for example, *Training/Practice/Block/Registry.php)* and define the class as follows.

```
  class Training_Practice_Block_Registry
extends Mage_Core_Block_Template
  {
  }
```

b) Define a getter method for use in the template file.

```
  class Training_Practice_Block_Registry
extends Mage_Core_Block_Template
  {
      public function getRegisteredVal()
{
          return Mage::registry('SOME_VALUE');
      }
  }
```

Note that this block will use the `_toHtml()` method defined in the parent class `Mage_Core_Block_Template` – to handle the rendering of the template when `toHtml()` is called.

Step 3. Create a template to output the value using the block getter method

a) Create a module-specific folder and a new template file under the `base/default` theme's template directory: *app/design/frontend/base/default/template/training/practice/registry.phtml.*

b) In the template file, add some markup and call the block getter method that you defined in Step 2b.

```
<div>
  <?php echo $this->__('The registered value is:') ?>
  <span style="color:red;">
    <?php echo $this->getRegisteredVal() ?>
  </span>
</div>
```


c) In the action defined above in Step 1a, instantiate the block class and assign the template using the `setTemplate()` method from `core/template`.

```
public function registryAction()
{
  Mage::register('SOME_VALUE','Some custom string');

  $block = $this->getLayout()
      ->createBlock('training_practice/registry')
      ->setTemplate('training/practice/registry.phtml');

  $this->getResponse()->setBody($block->toHtml());
}
```

d) Visit the appropriate URL and note that the output is composed of content that includes the registered value and the template. You should see the following output:

`The registered value is: Some custom string`

Exercise 4: Create a `core/text_list` Block, Assign Child Blocks, & Add Output to the Response Object

Use the `Training_Practice` module for these exercises.

Step 1. Create a text list block in a controller action

- Create a new action `textListAction()` in the action controller class from the previous exercise and assign a `core/text_list` (`Mage_Core_Block_Text_List`) block to a variable.

```
public function textListAction()
{
  $listBlock = $this->getLayout()
      ->createBlock('core/text_list');
}
```


Step 2. Instantiate two other blocks and add them as children to the text list block using the `insert()` method

a) For this step, we'll use the `core/text` class and assign some text for output using the `setText()` method of the class. When these blocks are rendered, they return the text that has been set on them

```
  public function textListAction()
  {
      $listBlock = $this->getLayout()
->createBlock('core/text_list');

      $blockA = $this->getLayout()
->createBlock('core/text','block.a')
->setText('<p>Example block A.</p>');

      $blockB = $this->getLayout()
->createBlock('core/text','block.b')
->setText('<p>Example block B.</p>');
  }
```

There are now a total of three block instances in the layout object. Note that we're electing to assign names to the text block instances using the second parameter to the `createBlock()` method.

b) Next, we'll use the `insert()` method from the `Mage_Core_Block_Abstract` superclass to add the two text block instances to the text list block's `_children` and `_sortedChildren` arrays, i.e., "make them children of" the text list block instance, and then add the text list block output to the response object.

```
public function textListAction()
{
    //... snip ... (block creation logic from above)

    $listBlock->insert( $blockA )
              ->insert( $blockB );
    /*
          Blocks can also be assigned by name, i.e.:

          $listBlock->insert( 'block.a' )
                    ->insert( 'block.b' );

   */

    $this->getResponse()->setBody($listBlock->toHtml());
}
```

c) Visit the textlist action in or your controller and you should see the following output:

```
Example block B.

Example block A.
```

When rendered, a text list block will simply render its children in order from its `_sortedChildren` array. This allows other blocks to be assigned and rendered without the need to edit template files.

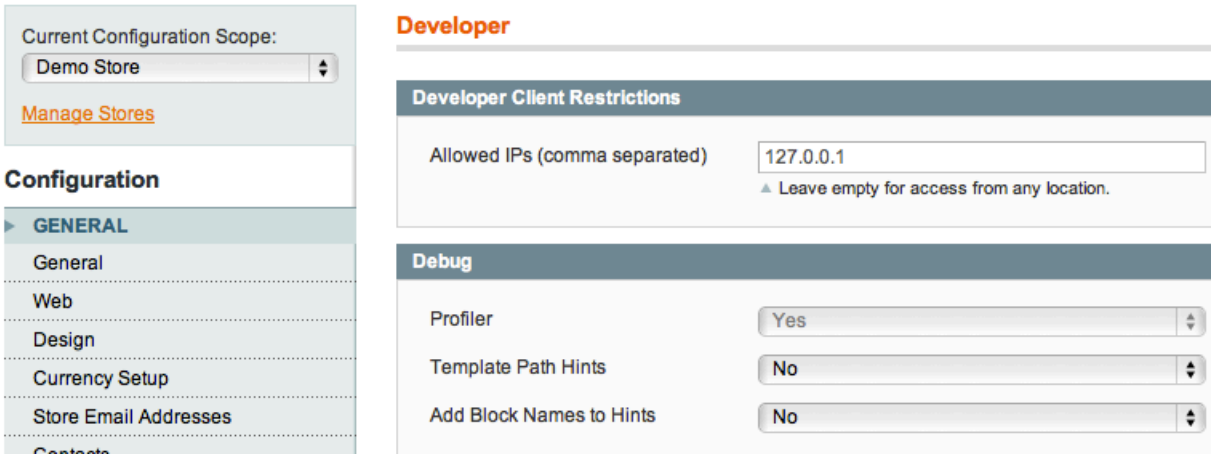
Note: The default behavior of the `insert()` method is to add the child instance to the first position of the `_sortedChildren` array. There are sorting parameters available, as well as an alternative `append()` method. Refer to the method definition to learn more.

Exercise 5: Use a Block Rewrite to Add a Hardcoded Breadcrumb

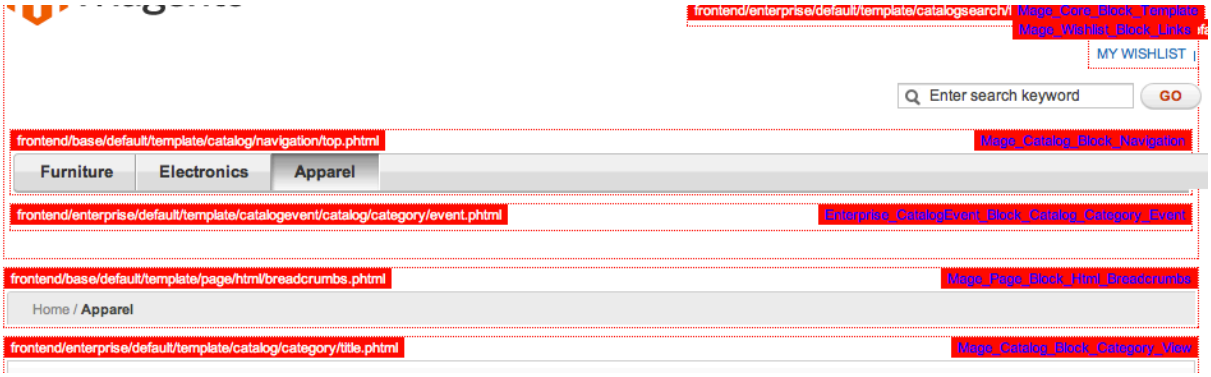
Use the Magento Admin and your IDE to do this exercise.

Step 1. Enable template hints and find the breadcrumbs block

a) Navigate to System > Configuration > Developer and change the **Current Configuration Scope** to a relevant scope *other than* "Default Config". Click to expand the Debug section; the **Template Path Hints** and **Add Block Names to Hints** fields should be visible. Change these to "Yes", save, and refresh the config cache if necessary.



b) Visit a page that renders breadcrumbs (for example, a category or product page). The breadcrumbs block should be headed by a red `<div>`, which contains the path to the template as well as the block class being used.



In the above example, "**Home / Apparel**" is the output of the breadcrumbs block, and the template path is *frontend/base/default/page/html/breadcrumbs.phtml*. The block class name is `Mage_Page_Block_Html_Breadcrumbs`, which in class group notation is `page/html_breadcrumbs`.

Step 3. Override the block using configuration XML

a) Merge the following XML structure to the `Training_Practice` module's *config.xml* file. If your XML structure is not the same as above, simply ensure that the XPath to the `html_breadcrumbs` class name is complete.

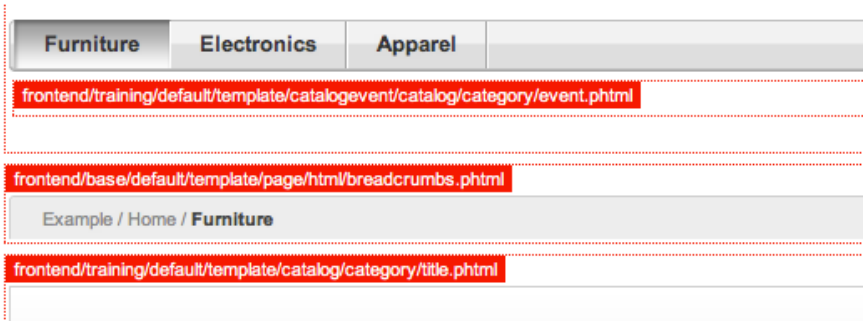
```
<?xml version="1.0"?>
<config>
 <global>
  <blocks>
   <page>
    <rewrite>
<html_breadcrumbs>Training_Practice_Block_Page_Breadcrumbs</html_breadcrumbs>
    </rewrite>
   </page>
  </blocks>
 </global>
</config>
```

Note the `<page>` and `<html_breadcrumbs>` nodes and how they match the class group notation. When the `createBlock()` factory invokes class name resolution, rather than getting a class name of `Mage_Page_Block` + `html_breadcrumbs`, the rewritten class name is used.

b) Create the class indicated in the configuration XML. Following the above example, create a class definition for `Training_Practice_Block_Page_Html_Breadcrumbs` at *Training/Practice/Block/Page/Html/Breadcrumbs.php.*

```
class Training_Practice_Block_Page_Html_Breadcrumbs
      extends Mage_Page_Block_Html_Breadcrumbs
{
  protected function _prepareLayout()
  {
    $this->addCrumb('practice', array(
      'link'  => 'http://example.com/',
      'label' => 'Example',
      'title' => 'This is a hardcoded crumb'
    ));
    return parent::_prepareLayout();
  }
}
```


c) Refresh the page. You should see something similar to the following.



Notice that along with the additional output, the rewritten class name, `Training_Practice_Block_Page_Html_Breadcrumbs`, is being displayed.

