Magento® U

Exercise Solutions

Fundamentals of Magento 2 Development
Unit One - Unit Three

magento.com/training/overview

v.1.0

Magento® U

training@magento.com

magento.com/training/overview

magento.com/training/catalog/certification

Contents

Unit One: Preparation & Configuration

Section 4: Magento 2 Overview

1.4.1) Create a new module. Make a mistake in its config. Create a second module dependent on the first.

Solution

1) Create a folder `app/code/Training/Test`.
2) Create a file `app/code/Training/Test/etc/module.xml`:

```
<?xml version="1.0"?>
<config xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="../../../../../lib/internal/Magento/Framework/Module
/etc/module.xsd">
    <module name="Training_Test" schema_version="2.0.0">
    </module>
</config>
```

3) Add your new module to the `app/etc/config.php` in the list of modules.
4) Make a mistake in the module.xml. For example change `</module>` to `</mod>`. Then clean the cache (using the command `rf -rf var/cache/*`) and load any page. You should get an error:
Warning: DOMDocument::loadXML(): Opening and ending tag mismatch: module line 8 and mod in Entity, line: 9 in /var/www/magento/m2/lib/internal/Magento/Framework/Module/ModuleList/Loader.php on line 56.
5) Fix the xml and clean the cache again.
6) Create a folder `app/code/Training/Test2` and file `app/code/Training/Test2/etc/module.xml`:

```
<?xml version="1.0"?>
<!--
/**
 * @copyright Copyright (c) 2014 X.commerce, Inc. (http://www.magentocommerce.com)
 */
-->
<config xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:noNamespaceSchemaLocation="../../../../../lib/internal/Magento/Framework/Module
   /etc/module.xsd">
       <module name="Training_Test2" schema_version="2.0.0">
           <sequence>
               <module name="Magento_Test"/>
           </sequence>
       </module>
</config>
```

7) Add `Training_Test2` to the list of modules in `app/etc/config.php`.
8) Clean the cache, and test whether your module is working.
9) You can disable `Training_Test` by setting its value to `0` in the `etc/config.php`
10) After cleaning the cache, there will be no visible change. In order to see a list of loaded modules, go into the class `Magento\Framework\Module\ModuleList`, method `getNames()`, and put `print_r($result); exit;` before the return from the method. It will show a list of loaded methods, and you will see `Training_Test2` but no `Training_Test`.

Section 7: DI & Object Manager

1.7.1) In the empty module you've created, create new xml/xsd files.

Solution

In order to create new xml/xsd files, we have to take the following steps:

Phase 1: Create `test.xml` and `test.xsd` files.
Phase 2: Create php files to process them: Config, ConfigInterface, Convertor, Reader, SchemaLocator.
Phase 3: Define a preference for `ConfigInterface`.
Phase 4: Test: In this example we will create a new controller to test this functionality out.

Let's follow through each step:

Phase 1:

1.1)Create etc/test.xml:

```
<?xml version="1.0"?>
<config xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="config.xsd">
  <mynode>HELLO</mynode>
  <mynode>HELLO 2</mynode>
</config>
```

1.2)Create etc/test.xsd:

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

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:element name="config">
        <xs:complexType>
            <xs:sequence>
            <xs:element name="mynode" type="xs:string" maxOccurs="10"/>
            </xs:sequence>
      </xs:complexType>
    </xs:element>
</xs:schema>
```

Phase 2:

2.1) Create an interface:

```
<?php
namespace Training\Test\Model\Config;

interface ConfigInterface {
    public function getMyNodeInfo();
}
```

2.2) Create Config class:

```
<?php
namespace Training\Test\Model;

class Config extends \Magento\Framework\Config\Data implements
\Training\Test\Model\Config\ConfigInterface
{
    public function __construct(
      \Training\Test\Model\Config\Reader $reader,
      \Magento\Framework\Config\CacheInterface $cache,
      $cacheId = 'training_test_config'
    ) {
          parent::__construct($reader, $cache, $cacheId);
    }

    public function getMyNodeInfo() {
      return $this->get();
    }
}
```

2.3) Create Reader class:

```
<?php
namespace Training\Test\Model\Config;

class Reader extends \Magento\Framework\Config\Reader\Filesystem
{
    /**
     * List of id attributes for merge
     *
     * @var array
     */
    protected $_idAttributes = []; //['/config/option' => 'name',
'/config/option/inputType' => 'name'];

    /**
     * @param \Magento\Framework\Config\FileResolverInterface $fileResolver
     * @param \Magento\Catalog\Model\ProductOptions\Config\Converter $converter
     * @param \Magento\Catalog\Model\ProductOptions\Config\SchemaLocator
$schemaLocator
     * @param \Magento\Framework\Config\ValidationStateInterface $validationState
     * @param string $fileName
     * @param array $idAttributes
     * @param string $domDocumentClass
     * @param string $defaultScope
     */
public function __construct(
      \Magento\Framework\Config\FileResolverInterface $fileResolver,
      \Training\Test\Model\Config\Converter $converter,
      \Training\Test\Model\Config\SchemaLocator $schemaLocator,
      \Magento\Framework\Config\ValidationStateInterface $validationState,
      $fileName = 'test.xml',
      $idAttributes = [],
      $domDocumentClass = 'Magento\Framework\Config\Dom',
      $defaultScope = 'global'
    ) {
            parent::__construct(
            $fileResolver,
            $converter,
            $schemaLocator,
            $validationState,
            $fileName,
            $idAttributes,
            $domDocumentClass,
            $defaultScope
      );
    }
}
```

2.4) Create schemaLocator class:

```
<?php
namespace Training\Test\Model\Config;

class SchemaLocator implements \Magento\Framework\Config\SchemaLocatorInterface
{
    /**
     * Path to corresponding XSD file with validation rules for merged config
     *
     * @var string
     */
    protected $_schema = null;

    /**
     * Path to corresponding XSD file with validation rules for separate config files
     *
     * @var string
     */
    protected $_perFileSchema = null;

    /**
     * @param \Magento\Framework\Module\Dir\Reader $moduleReader
     */
    public function __construct(\Magento\Framework\Module\Dir\Reader $moduleReader)
    {
      $etcDir = $moduleReader->getModuleDir('etc', 'Training_Test');
      $this->_schema        = $etcDir . '/test.xsd';
      $this->_perFileSchema = $etcDir . '/test.xsd';
    }

    /**
     * Get path to merged config schema
     *
     * @return string|null
     */
    public function getSchema()
    {
        return $this->_schema;
    }

    /**
     * Get path to pre file validation schema
     *
     * @return string|null
     */
    public function getPerFileSchema()
    {
      return $this->_perFileSchema;
    }
}
```

2.5) Create converter class:

```
<?php

namespace Training\Test\Model\Config;

class Converter implements \Magento\Framework\Config\ConverterInterface
{
    /**
     * Convert dom node tree to array
     *
     * @param \DOMDocument $source
     * @return array
     * @throws \InvalidArgumentException
     */
    public function convert($source)
    {
        $output = [];

        /** @var $optionNode \DOMNode */
      foreach ($source->getElementsByTagName('mynode') as $node) {
            $output[] = $node->textContent;
        }
        return $output;
    }
}
```

Phase 3:

In the di.xml set a preference:

```
 <preference for="Training\Test\Model\Config\ConfigInterface"
type="Training\Test\Model\Config" />
```

Phase 4:

4.1) Create a controller file (assuming you've set up routes.xml already):

```
<?php
/**
 * Product controller.
 *
 * @copyright Copyright (c) 2014 X.commerce, Inc. (http://www.magentocommerce.com)
 */
namespace Training\Test\Controller\Action;

class Config extends \Magento\Framework\App\Action\Action
{

    public function execute() {
        $testConfig = $this->_objectManager-
>get('Training\Test\Model\Config\ConfigInterface');
        $myNodeInfo = $testConfig->getMyNodeInfo();
        if (is_array($myNodeInfo)) {
            foreach($myNodeInfo as $str) {
                $this->getResponse()->appendBody($str . "<BR>");
            }
        }
    }
}
```

4.2) Hit a page /test/action/config. You will see:

```
HELLO
HELLO 2
```

Section 8: Plugins

1.8.1) For \Magento\Catalog\Model\Product getPriceMethod(), create a plugin and preference.

Solution

Option 1: Create a plugin.
Option 2: Create a preference.

Please note, in real-world situations, you should use option 1 OR option 2, but not both.

Option 1:

1.1) Add a plugin declaration into di.xml:

```
    <type name="Magento\Catalog\Model\Product">
        <plugin name="magento-catalog-product-plugin"
                type="Training\Test\Model\Product" sortOrder="10"/>
    </type>
```

1.2) Create a plugin class:

```
<?php

namespace Training\Test\Model;

class Product {
    public function afterGetPrice(\Magento\Catalog\Model\Product $product, $result) {
        return 5;
    }
}
```

1.3) Visit any page (after cleaning the cache). You should see every price being set to $5.

Option 2:

NOTE: Don’t forget to disable the declaration done in Option 1!

2.1) Create a preference declaration:

```
    <preference for="Magento\Catalog\Model\Product"
       type="Training\Test\Model\Testproduct" />
```

2.2) Create a new Product class:

```
<?php
namespace Training\Test\Model;

class Testproduct extends \Magento\Catalog\Model\Product
{
    public function getPrice() {
        return 3;
    }
}
```

2.3) Test. Now all prices should be set to $3.

Section 9: Events

1.9.1) Create an observer to the event controller_action_predispatch

Solution

1) Create an event declaration in the events.xml:

```
<event name="controller_action_predispatch">
    <observer name="training_test"
      instance="Training\Test\Model\Observer"
      method="changeRequestParams" shared="false" />
</event>
```

2) Create an Observer:

```
<?php
namespace Training\Test\Model;

class Observer {

    public function changeRequestParams(\Magento\Framework\Event\Observer $observer) {
      $request = $observer->getEvent()->getData('request');
      $request->setModuleName('catalog');
      $request->setControllerName('product');
      $request->setActionName('view');
      $request->setParams(array('id' => 1));
    }
}
```

Result: Now all pages are “Not found.”

3) Comment out `setModuleName()`, `setControllerName()`, `setActionName()` but be sure to leave `setParams()`. Now all product pages refer to the same product page.

```
    public function changeRequestParams(\Magento\Framework\Event\Observer $observer) {
      $request = $observer->getEvent()->getData('request');
      //$request->setModuleName('catalog');
      //$request->setControllerName('product');
      //$request->setActionName('view');
      $request->setParams(array('id' => 1));
    }
```

Unit Two: Request Flow

Section 2: Request Flow Overview:

2.2.1) Find a place in the code where output is flushed to the browser. Now, create an extension that captures and logs the file-generated page html.

Solution

1) Declare an event in the file etc/frontend/events.xml:

```
<config xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="../../../../../../lib/internal/Magento/Framework/Event/
etc/events.xsd">
    <event name="controller_front_send_response_before">
        <observer name="training_test" instance="Training\Test\Model\Observer"
method="logPageOutput" shared="false" />
    </event>
</config>
```

2) Create an observer class:

```
<?php
namespace Training\Test\Model;

class Observer {
    protected $_logger = null;

    public function __construct(\Psr\Log\LoggerInterface $logger) {
        $this->_logger = $logger;
    }

    public function logPageOutput(\Magento\Framework\Event\Observer $observer) {
      return;
      $response = $observer->getEvent()->getData('response');
      $body = $response->getBody();
      $this->_logger->addDebug("--------\n\n\n BODY \n\n\n ". $body);
    }
}
```

Section 3: Request Routing

2.3.1 Create an extension that logs into the file list of all available routers.

Solution

1) Create a preference in the di.xml:

```
  <preference for="Magento\Framework\App\FrontControllerInterface"
type="Training\Test\App\FrontController" />
```

2) Implement a front controller class:

```
<?php
namespace Training\Test\App;

class FrontController extends \Magento\Framework\App\FrontController
{

    protected $_routerList;
    protected $_logger;

    public function __construct(\Magento\Framework\App\RouterList $routerList,
\Psr\Log\LoggerInterface $logger)
    {
        $this->_routerList = $routerList;
         $this->_logger = $logger;
    }


    public function dispatch(\Magento\Framework\App\RequestInterface $request) {
      foreach ($this->_routerList as $router) {
            $this->_logger->addDebug(get_class($router));
      }
        return parent::dispatch($request);
    }
}
```

2.3.2) Create a new router which “understands” urls like /frontName-actionPath-action (and converts them to: /frontName/actionPath/action).

Solution

1) Declare your router. Add the following code to the etc/frontend/di.xml of your module (assuming your module is Training_Test):

```
    <type name="Magento\Framework\App\RouterList">
        <arguments>
            <argument name="routerList" xsi:type="array">
                <item name="training" xsi:type="array">
                    <item name="class"
                      xsi:type="string">Training\Test\Controller\Router</item>
                    <item name="disable" xsi:type="boolean">false</item>
                    <item name="sortOrder" xsi:type="string">70</item>
                </item>
            </argument>
        </arguments>
    </type>
```

2) Create a router class:

```
<?php

namespace Training\Test\Controller;

class Router implements \Magento\Framework\App\RouterInterface
{
    public function __construct(\Magento\Framework\App\ActionFactory $actionFactory) {
        $this->actionFactory = $actionFactory;
    }

    public function match(\Magento\Framework\App\RequestInterface $request) {
        $info = $request->getPathInfo();

        if (preg_match("%^/(test)-(.*?)-(.*?)$%", $info, $m)) {
            $request->setPathInfo(sprintf("/%s/%s/%s", $m[1], $m[2], $m[3]));
            return $this->actionFactory-
                >create('Magento\Framework\App\Action\Forward', ['request' =>
                $request]);
                }
        return null;
    }
}
```

In this example, the router only “understands” urls that start with “test”. To make it work with every url, remove the line:

```
if (preg_match("%^/(test)-(.*?)-(.*?)$%", $info, $m)) {
```

2.3.3) Modify Magento so a “not found” page will forward to the homepage.

Solution
There are many different ways to do this. The easiest is to change the config option: /web/default/noroute. This will change the 404 page for all requests. To make the code more flexible, you can create a new NoRouteHandler. For doing this:

1) Declare your handler in the di.xml:

```
<type name="Magento\Framework\App\Router\NoRouteHandlerList">
    <arguments>
        <argument name="handlerClassesList" xsi:type="array">
            <item name="default" xsi:type="array">
                <item name="class"
                      xsi:type="string">Training\Test\Controller\NoRouteHandler</item>
                <item name="sortOrder" xsi:type="string">200</item>
            </item>
        </argument>
    </arguments>
</type>
```

2) Create a handler class:

```
<?php
namespace Training\Test\Controller;

class NoRouteHandler implements \Magento\Framework\App\Router\NoRouteHandlerInterface
{

    public function process(\Magento\Framework\App\RequestInterface $request) {
        $moduleName      = 'cms';
        $controllerName  = 'index';
        $actionName      = 'index';

        $request
          ->setModuleName($moduleName)
          ->setControllerName($controllerName)
          ->setActionName($actionName);
        return true;
    }
}
```

Section 5: Working with Controllers

2.5.1) Create a frontend controller which renders “HELLO WORLD”.

Solution

1) Declare a route in etc/frontend/routes.xml:

```
<config xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="../../../../../../lib/internal/Magento/Framework/App/et
c/routes.xsd">
    <router id="standard">
        <route id="test" frontName="test">
            <module name="Training_Test" />
        </route>
    </router>
</config>
```

2) Create an action class:

```
<?php
/**
 * Product controller.
 *
 * @copyright Copyright (c) 2014 X.commerce, Inc. (http://www.magentocommerce.com)
 */
namespace Training\Test\Controller\Action;

class Index extends \Magento\Framework\App\Action\Action
{

    public function execute() {
        $this->getResponse()->appendBody("HELLO WORLD");
    }
}
```

2.5.2) Customize the catalog product view controller using plugins and preferences.

Solution

1) To add a plugin or preference, use the following code in di.xml:

```
  <preference for="Magento\Catalog\Controller\Product\View"
   type="Training\Test\Controller\Product\View" />
```

Or

```
<type name="Magento\Catalog\Controller\Product\View">
    <plugin name="product-view-controller-plugin"
            type="Training\Test\Controller\Product\View" sortOrder="10"/>
</type>
```

Note: You will create a preference OR plugin within one module.

2) Now you can implement your preference/plugin:

```
<?php

namespace Training\Test\Controller\Product;

class View extends \Magento\Framework\App\Action\Action
{
    /**
    public function execute() {
        echo "ONE"; exit;
    }
    public function beforeExecute() {
        //echo "BEFORE<BR>"; exit;
    }
    public function afterExecute(\Magento\Catalog\Controller\Product\View $controller,
$result) {
        //echo "AFTER"; exit;
    }
    */
}
```

3) Uncomment the appropriate method for testing: Uncomment "`execute`" for preferences, and "`beforeExecute`", "`afterExecute`" for plugins.

2.5.3) Create an adminhtml controller that allows access only if the GET parameter "secret" is set.

Solution

1) Create a file `Etc/adminhtml/routes.xml`:

```
<?xml version="1.0"?>
<!—
/**
 * Copyright © 2015 Magento. All rights reserved.
 * See COPYING.txt for license details.
 */
-->
<config xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="../../../../../../lib/internal/Magento/Framework/App/et
c/routes.xsd">
    <router id="admin">
        <route id="test" frontName="test">
            <module name="Training_Test" before="Magento_Adminhtml" />
        </route>
    </router>
</config>
```

2) Create an action class:

```
<?php
```

```
/**
 *
 * Copyright © 2015 Magento. All rights reserved.
 * See COPYING.txt for license details.
 */
namespace Training\Test\Controller\Adminhtml\Action;

class Index extends \Magento\Backend\App\Action
{

    /**
     * Product list page
     *
     * @return \Magento\Backend\Model\View\Result\Page
     */
    public function execute()
    {
      $this->getResponse()->appendBody("Hello world in admin");
    }

    protected function _isAllowed() {
      $secret = $this->getRequest()->getParam('secret');
      return isset($secret) && (int)$secret==1;
    }
}
```

2.5.4) Make the “Hello World” controller you just created redirect to a specific category page.

Solution

Put a line `$this->_redirect(‘catalog/category/view/id/_CATEGORY_ID_’)` into the execute method (but replace `_CATEGORY_ID_` with the real category_id).

Section 6: URL Rewrites

2.6.1) Create a url rewrite for the “Hello World” controller

Solution

Add one record to the url_rewrite table:

```
INSERT INTO url_rewrite SET request_path=’testpage.html’,
target_path=’test/action/index’, redirect_type=0, store_id=1, is_autogenerated=0;
```

Unit Three: Rendering

Section 3: Rendering Flow

3.3.1) In the core files, find and print out the layout xml for the product view page and the shopping cart page.

Solution

For both: `\Magento\Framework\View\Layout::generateXml()`

Section 4: Block Architecture &Lifecycle

3.5.1) Create a block extending AbstractBlock, and implement the _toHtml() method. Render that block in the new controller.

Solution

1) Create the block:

```
<?php
namespace Training\Test\Block;

class Test extends \Magento\Framework\View\Element\AbstractBlock
{
    protected function _toHtml() {
        return "<b>Hello world from block!</b>";
    }
}
```

2) Create an action class:

```
<?php
namespace Training\Test\Controller\Block;

class Index extends \Magento\Framework\App\Action\Action
{

    public function execute() {
        $layout = $this->_view->getLayout();
        $block = $layout->createBlock('Training\Test\Block\Test');
      $this->getResponse()->appendBody($block->toHtml());
    }
}
```

3.5.2) Create and render in controller text block.

Solution

Create an action class:

```
<?php

namespace Training\Test\Controller\Block;

class Text extends \Magento\Framework\App\Action\Action
{
    public function execute() {
        $block = $this->_view->getLayout()-
>createBlock('Magento\Framework\View\Element\Text');
        $block->setText("Hello world from text block !");
      $this->getResponse()->appendBody($block->toHtml());
    }
}
```

3.5.3) Customize the Catalog\Product\View\Description block, implement the _beforeToHtml() method, and set the custom description to the product here.

Solution

1) Declare a plugin in the etc/frontend/di.xml:

```
    <type name="Magento\Catalog\Block\Product\View\Description">
        <plugin name="product-view-desription-plugin"
                type="Training\Test\Block\Product\View\Description" sortOrder="10"/>
    </type>
```

2) Create a plugin class:

```
<?php
namespace Training\Test\Block\Product\View;

class Description extends \Magento\Framework\View\Element\Template
{
    public function beforeToHtml(\Magento\Catalog\Block\Product\View\Description
$originalBlock) {
        $originalBlock->getProduct()->setDescription('Test description');
    }
}
```

Section 5: Templates

3.6.1) Define which template is used in Catalog\Block\Product\View\Attributes.

Solution

`Magento/Catalog/view/frontend/templates/product/view/attributes.phtml`

3.6.2) Create a template block, and a custom template file for it. Render the block in the controller.

Solution

1) Create the block:

```
<?php
namespace Training\Test\Block;

class Template extends \Magento\Framework\View\Element\Template
{
}
```

Please note: You may not create your own block, but must use Magento\Framework\View\Element\Template, since it is not an abstract.

2) Create a template file `Training/Test/view/frontend/test.phtml`:

```
“Hello from template”.
```

3) Create an action class:

```
<?php
namespace Training\Test\Controller\Block;

class Template extends \Magento\Framework\App\Action\Action
{

    public function execute() {
      $block = $this->_view->getLayout()->createBlock('Training\Test\Block\Template');
      $block->setTemplate('test.phtml');
      $this->getResponse()->appendBody($block->toHtml());
    }
}
```

3.6.3) Customize the Catalog\Block\Product\View\Description block, and set a custom template to it.

Solution

1) Using the same declaration as in 2.3, change the `beforeToHtml` method to:

```
public function beforeToHtml(\Magento\Catalog\Block\Product\View\Description
$originalBlock) {
        $originalBlock->setTemplate('Training_Test::description.phtml');
    }
```

2) Create a template `Training/Test/view/frontend/templates/description.phtml`:

```
<h1>Custom description template!</h1>
```

Section 7: Layout XML

You will be provided with a code archive containing the solutions for the exercises in this section.