4

Creating a module

In this chapter, we will cover

* Creating the module files
* Creating a controller
* Adding layout updates
* Adding a translation file
* Adding a block of new products
* Adding a dependency injection

# Introduction

When you look in the app/code folder (the core of Magento), you see the modular architecture. Every concept in the e-commerce flow is stored in a module. The Magento application is the combination of all these modules.

The advantage of a modular architecture is the extendibility. It is easy to add modules that adds or modifies the behaviour of Magento.

In this chapter, we will create a module with the most important things you need to know when writing code in Magento.

# Creating the module files

When creating a module, the first step is to create the files and folders to register the module. At the end of this recipe, we will have a registered module but without functionality.

In the next recipes, we will add extra features to that module.

## Getting ready

Open your Magento 2 website in the app/code/ folder. This is the folder where all the module development needs to be done.

Access to a command line is also recommended.

## How to do it

In the following steps, we will create the required files to register a Magento module.

1. We will create a HelloWorld module in the Packt namespace. Create the following folders:

app/code/Packt

app/code/Packt/HelloWorld

app/code/Packt/HelloWorld/etc

1. In the etc folder of the module, create a file called module.xml with the following content:

<?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="Packt\_HelloWorld" setup\_version="2.0.0.0">

<sequence>

<module name="Magento\_Catalog"/>

</sequence>

</module>

</config>

1. Open your terminal and go to the Magento directory. In that directory, run the following commands:

php bin/magento cache:clean --all

php bin/magento setup:upgrade

1. When everything is OK, you can see the name of the module in the output of the last command.
2. To test that the module is installed, open the backend and go to the page Stores | Configuration | Advanced | Advanced and check that the module is present in the list. Make sure you have cleaned the Magento caches.

## How it works

Module development in Magento 2 is much easier than in Magento 1. The concept of code pools is gone, everything is stored in 1 folder (code, translations, templates, CSS, …). These things makes it a lot easier to develop and maintain a Magento module.

To initialize, we have to create the folders and the module.xml file in the etc folder of the module. In the module.xml file, we initialized the name Packt\_HelloWorld, the version number and the dependencies.

When we created the module files, we executed the command setup:upgrade. By running this command, we will run the install or upgrade procedure of all the modules. In this process, a lot of dependencies are generated in the folder var/generation.

We used the tool bin/magento for cleaning the cache and running the upgrade scripts. This tool is introduced in Magento 2 and is a replacement of 3rd party tools from Magento 1 (like n98magerun and wiz).

When running the command php bin/magento, you see a list of all available commands and it is easy to add your own commands in a module.

# Creating a controller

The first thing that we will do extend our module is something very visible. We will add an extra page that we can use for several purposes.

## Getting ready

We build further on the Packt\_HelloWorld module that we created in the previous recipe. Make sure you have this module in your Magento instance.

## How to do it

The following steps shows you how to add extra pages using controllers and controller actions.

1. Create the following folders:

app/code/Packt/HelloWorld/etc/frontend

app/code/Packt/HelloWorld/Controller

app/code/Packt/HelloWorld/Controller/Index

1. In the folder app/code/Packt/HelloWorld/etc/frontend, create a routes.xml file with the following content:

<?xml version="1.0"?>

<config xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="../../../../../../lib/internal/Magento/Framework/App/etc/routes.xsd">

<router id="standard">

<route id="helloworld" frontName="helloworld">

<module name="Packt\_HelloWorld" />

</route>

</router>

</config>

1. In that last folder app/code/Packt/HelloWorld/Controller/Index, create the file Index.php with the following content:

<?php

namespace Packt\HelloWorld\Controller\Index;

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

{

/\*\*

\* Index action

\*

\* @return $this

\*/

public function execute()

{

}

}

1. Clean the cache using the command php bin/magento cache:clean --all and generate the dependencies using php bin/magento setup:upgrade.
2. Open your browser and navigate to /helloworld url of the shop. You will see a white page. This is normal because the controller action is empty.
3. To load the layout of the shop, add the following code in the execute() action:

/\*\* @var \Magento\Framework\View\Result\PageFactory \*/

protected $resultPageFactory;

public function \_\_construct(

\Magento\Framework\App\Action\Context $context,

\Magento\Framework\View\Result\PageFactory $resultPageFactory

) {

$this->resultPageFactory = $resultPageFactory;

parent::\_\_construct($context);

}

public function execute()

{

$resultPage = $this->resultPageFactory->create();

return $resultPage;

}

1. We will now create an extra action that redirects us to the HelloWorld page. Create the file app/code/Packt/HelloWorld/Controller/Index/Forward.php.
2. In that file, add the following content:

<?php

namespace Packt\HelloWorld\Controller\Index;

class Forward extends \Magento\Framework\App\Action\Action

{

public function execute()

{

$this->\_redirect('helloworld');

}

}

1. When going to the URL /helloworld/index/forward, we will be forwarded to the index action.
2. We can also change the content of the execute() function to $this->\_forward('index');. We will see the same output but the URL is doesn’t change in the forward.

## How it works

All pages in Magento are executed by controller actions. All the controllers are placed in modules and every controller can have multiple controller actions.

This gives us the following structure of URL: <modulename or frontname>/<controllerName>/<actionName>.

When you compare the controller part with Magento 1, a lot of things are changed and made easier.

In Magento 2, every controller action is written in a separate class. This class extends the Magento\Framework\App\Action\Action class. The controller is the folder where the actions are placed in.

It is also possible that the controller is in a separate class but this is only done when there are generic functions that the actions will use. A good example can you find in ProductController of the Magento\_Catalog module.

In a controller action, the execute() function is used to start the rendering of the page. When nothing is in this page, the page will have an empty output.

If we wanted to render the layout, we initialized the resultPageFactory in the \_\_construct() method of the controller. This factory class is used to start the layout rendering of the page.

The second controller action we created was one that does a redirect to another page. When calling the \_redirect() function in a controller action, a 301 redirect will be returned to the given URL.

The \_forward() function does likely the same but this internally forwards the action to another controller. This means that the output of another controller action will be rendered on the page but the URL doesn’t change.

This method is used to translate an SEO friendly URL (like a product URL) to the right controller action with the right parameters.

# Adding Layout updates

In the previous recipe, we created a page without content. In this recipe, we will modify the content of that page with layout updates.

With layout updates, we can arrange the structure of the page like we have seen in the recipe Customizing the HTML output of Chapter 3. But here we will see how we can do that in a module.

## Getting ready

This recipe builds further on the previous recipe. You need the module installed that we have created in the previous recipes.

## How to do it

In the next steps, we will see how we can modify the block layout with our module.

1. Create the folder app/code/Packt/HelloWorld/view/frontend/layout.
2. In that folder, create a file called default.xml with the following content:

<?xml version="1.0"?>

<page xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="../../../../../../../lib/internal/Magento/Framework/View/Layout/etc/page\_configuration.xsd">

<body>

<referenceBlock name="footer\_links">

<block class="Magento\Framework\View\Element\Html\Link\Current" ifconfig="contact/contact/enabled" name="helloworld-link">

<arguments>

<argument name="label" xsi:type="string">Helloworld landing</argument>

<argument name="path" xsi:type="string">helloworld</argument>

</arguments>

</block>

</referenceBlock>

</body>

</page>

1. Clean the cache using the command php bin/magento cache:clean --all and reload the frontend. In the footer, you will see an extra link leading to the page that we have created in the previous recipe.
2. The layout update we have just created, is applied on all pages. If we want updates on the helloworld index page, we have to create the file app/code/Packt/HelloWorld/view/frontend/layout/helloworld\_index\_index.xml.
3. In that file, paste the following content:

<?xml version="1.0"?>

<page xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" layout="2columns-left" xsi:noNamespaceSchemaLocation="../../../../../../../lib/internal/Magento/Framework/View/Layout/etc/page\_configuration.xsd">

<head>

<title>Helloworld Landingspage</title>

</head>

<body>

<remove name="wishlist\_sidebar" />

</body>

</page>

1. We also need to register the page. For this, create the file app/code/Packt/HelloWorld/etc/frontend/page\_types.xml with the following content:

<?xml version="1.0"?>

<page\_types xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="../../../../../../lib/internal/Magento/Framework/View/Layout/etc/page\_types.xsd">

<type id="helloworld\_index\_index" label="HelloWorld landing page"/>

</page\_types>

1. Clean the cache and reload the /helloworld page. You will see that the title is like we have configured and the wishlist block is not present in the left column.
2. To finish this recipe, we will add a custom template with a custom Block class. Create the file app/code/Packt/HelloWorld/Block/Landingspage.php with the following content:

<?php

namespace Packt\HelloWorld\Block;

use Magento\Framework\View\Element\Template;

class Landingspage extends Template

{

public function getLandingsUrl() {

return $this->getUrl('helloworld');

}

public function getForwardUrl() {

return $this->getUrl('helloworld/index/forward');

}

}

1. Now we have to create the template where we will call function from the Landingspage class. Create the file app/code/Packt/HelloWorld/view/frontend/templates/landingspage.phtml with the following content:

<h2>Hello World</h2>

<p>

<a href="<?php echo $block->getLandingsUrl(); ?>">Go to landings URL</a>

</p>

<p>

<a href="<?php echo $block->getForwardUrl(); ?>">Go to forward URL</a>

</p>

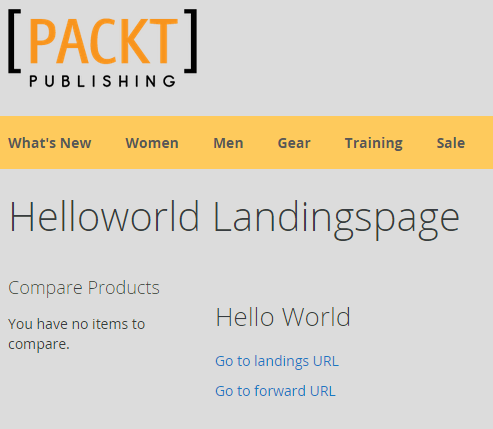
1. As a last step, we have to add the block with our layout XML. Add the following configuration to the file app/code/Packt/HelloWorld/view/frontend/layout/helloworld\_index\_index.xml as child of the <body> tag.

<referenceContainer name="content">

<block class="Packt\HelloWorld\Block\Landingspage" name="landingsblock" template="Packt\_HelloWorld::landingspage.phtml" />

</referenceContainer>

1. Clean the cache and reload the /landingspage URL. You will see something like the following screenshot:



Insert Image 4869\_04\_01.png

## How it works

Layout updates can be placed in modules and themes. In the recipe Customizing the HTML output of Chapter 3, is explained how layout updates works in modules but it is also possible to do it in a module.

Every Magento 2 folder has a view folder. In this folder all the stuff to render the page is stored like LESS (CSS), JavaScript, templates and layout files.

In the view folder, we can have the following subfolders:

* adminhtml
* base
* frontend

Like the names said, the adminhtml is used for the Magento backend, the frontend folder is used for the frontend and the base folder is used for both (frontend and backend).

In those folders, the following structure is used:

* layout (for layout update XML’s
* templates (for .phtml templates)
* web (for static files (LESS, JavaScript, images)

In the layout folder, we can place layout XML files. For every layout handle, we can apply layout updates in a separate file.

We have placed a layout file for the default handle (these instructions are loaded on all pages). Every page has also its own handle in the following structure <modulename>\_<controllername>\_<actionname>. For the helloworld landingspage, is this helloworld\_index\_index. In that file (helloworld\_index\_index.xml) we have placed the layout instructions of that page.

In that file, we created a layout instruction that adds a custom template with block to the page. The landingspage.phtml template of the Packt\_HelloWorld module is used to render the output. With the $block variable, we can call the methods of the Packt\HelloWorld\Block\Landingspage class.

# Adding a translation file

Magento is made to run in multiple language. That means that the interface needs to be translatable in the configured languages.

In this recipe, we will learn how we can make the strings in our module translatable in different languages.

## Getting ready

We will create translation files for the module that we have created in the previous recipes of this chapter. Make sure you have the code in your Magento instance.

## How to do it

The following procedure shows us how we can manage translations in our module.

1. To make a test translation, we can create a test translation in the template file that we created in the previous recipe. Add the following code at the end of the file app/code/Packt/HelloWorld/view/frontend/templates/landingspage.phtml.

<p>

<?php echo \_\_('Test translation') ?>

</p>

1. Go to the /helloworld page and you will see that the text Test translation is added on the page.
2. To translate this string, we have to create the folder app/code/Packt/HelloWorld/i18n.
3. In this folder, create the file en\_US.csv.
4. Add the following line in that csv file:

"Test translation","Translation to test"

1. Clean the cache and reload the page. If the language of your shop is set to English (United States), you will see that the output is set to Translation to test.
2. If we want for example a French translation, we have to create the file fr\_FR.csv with the following content:

"Test translation","Test traduction"

1. Change the language of the store to French, clean the cache and you will see the French translation.

If you want to know all the translations of a module, you can run the command php bin/magento i18n:collect-phrases app/code/<Vendor name>/<Module name> and you will get a CSV list of all the translations.

## How it works

When calling the \_\_('translate string') function, Magento will search for a translation for that string in the current language. Magento will look for the strings in the following order:

* The database table translation
* The theme translation files (app/design/fronted/<Package>/<theme>/i18n/<locale\_code>.csv)
* The module translation files (app/code/<Vendor>/<Module>/i18n/<locale\_code>.csv)

When a string is found, Magento don’t look further for other matching strings.

If no matching string is found for the current language, Magento will return the string that is present in the first parameter of the translate function.

The implementation of translations in Magento 2 is much easier than in Magento 1. Everything is stored in the module folder and you don’t have to add configuration XML instructions to the module where you can do mistakes with.

Also the translate function is now moved to a global function. You don’t need a helper class to call the \_\_() function. The \_\_() function is implemented as a global function that is available everywhere in the application.

# Adding a block of new products

In the previous recipes, we prepared the module for the real work. We added the most common features to the module so that we can easily extend it with new functionality.

In this recipe, we will add a block of new product to the page we created in the previous recipes.

## Getting ready

We will create a block that extends from the Magento\_Catalog module. Later, we have to add a layout XML instruction and a template.

Make sure you have the module of the previous recipe installed.

## How to do it

The following steps shows you how to start with adding the block with new products.

1. To create the block class, we have to create the file Newproducts.php in the folder app/code/Packt/HelloWorld/Block/.
2. Add the following content to that file:

<?php

namespace Packt\HelloWorld\Block;

use Magento\Framework\View\Element\Template;

class Newproducts extends Template

{

}

1. Create a template in the module folder. We can do this by creating the file newproducts.phtml in the folder app/code/Packt/HelloWorld/view/frontend/templates/.
2. Add some HTML content to that template file like <h2>New Products</h2>.
3. To add the block to the page, we have to create a layout update. In the file app/code/Packt/HelloWorld/view/frontend/layout/helloworld\_index\_index.xml. Add the following code as child of the <referenceContainer name="content">.

<block class="Packt\HelloWorld\Block\Newproducts" name="new\_products" template="Packt\_HelloWorld::newproducts.phtml" />

1. Clean the cache and reload the /helloworld page. You will see that New Products title is visible.
2. Create a contructor in the block class that initializes the product collection factory. We can do this by adding the following code in that class:

private $\_productCollectionFactory;

public function \_\_construct(

Template\Context $context,

\Magento\Catalog\Model\Resource\Product\CollectionFactory $productCollectionFactory,

array $data = [])

{

parent::\_\_construct($context, $data);

$this->\_productCollectionFactory = $productCollectionFactory;

}

1. Create the getProducts() function in the same block class. This function will return the five latest products of the shop. The code for the getProducts() function will look as follows:

public function getProducts() {

$collection = $this->\_productCollectionFactory->create();

$collection

->addAttributeToSelect('\*')

->setOrder('created\_at')

->setPageSize(5);

return $collection;

}

1. The last step is to call the function in the template and generate a HTML for it. The code of the template is the following:

<h2>New products</h2>

<ul>

<?php foreach ($block->getProducts() as $product): ?>

<li><?php echo $product->getName() ?></li>

<?php endforeach; ?>

</ul>

1. Reload the /helloworld page and you will see a list with the names of the latest products.

## How it works

What we have done in this recipe is a basic extension of Magento. We added a custom block that uses the Magento framework to render the content.

We created a block class that has the getProducts() function. This function returns the latest 5 products of the webshop.

In that function, we created a query using the Magento Collections. With Magento collections, we can get data from the database. A collection builds an SQL query in the background.

The purpose of collections is that there is an easy interface to get the right entities. A product is not stored in 1 database table and the Magento collections generates an SQL query that returns the values of that tables. This saves us the programming of very complex SQL queries.

To work with the collections, we used the product CollectionFactory to work with the functions. We initialized this class in the constructor and used it in the getProducts() function.

# Adding a dependency injection

One of the big things that is changed in Magento 2 is that there is no Mage class. To replace this, all object are passed to the classes with dependency injection.

Dependency injection is a powerful tool that adds a lot of flexibility to add or change behaviour in Magento.

## Getting ready

For exploring the possibilities of Dependency Injection of Magento 2, we need the module that we have created in the previous recipes.

## How to do it

The following steps describes how we can modify the behavior of some classes.

1. Create the file app/code/Packt/HelloWorld/etc/di.xml and paste the following content in it:

<?xml version="1.0"?>

<config xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="../../../../../lib/internal/Magento/Framework/ObjectManager/etc/config.xsd">

<type name="Magento\Catalog\Model\Product">

<plugin name="Packt\_HelloWorld::productName" type="Packt\HelloWorld\Plugin\Catalog\ProductAround" sortOrder="1" />

</type>

</config>

1. Create a plugin class by creating the file app/code/Packt/HelloWorld/Plugin/Catalog/ProductAround.php with the following contents:

<?php

namespace Packt\HelloWorld\Plugin\Catalog;

use Magento\Catalog\Model\Product;

class ProductAround

{

public function aroundGetName($interceptedInput)

{

return "Name of product";

}

}

1. Clean the caches and build the dependencies by running the command php bin/magento setup:upgrade.
2. Reload a product page and you will see that every product name is now Name of product.
3. To undo this, comment the <type> tag and contents of the di.xml file and regenerate the dependencies using php bin/magento setup:upgrade.
4. Reload a product page and you will see the normal product names.

## How it works

In this recipe, we added a dependency injection into the Magento\Catalog\Model\Product class. We did an override of an existing function in Magento.

With dependency injection, we can execute code before, after and around any function of a class. This gives a lot of possibilities to add behaviour to Magento.

In the di.xml file, we initialized a plugin that could override functions of the Magento\Catalog\Model\Product class.

The overrides are done in the class Packt\HelloWorld\Plugin\Catalog\ProductAround. In that class, we did a modification of the getName() function of the original class. This was done using the aroundGetName() function.

To test our code, we had to generate the dependencies using the command php bin/magento setup:upgrade.

This command generates the dependencies that will be placed in the var/generation folder. Without generating dependencies, the changed dependencies will not load. This is also the reason why you have to run this command when installing or upgrading a new module.