4
Creating a Module

In this chapter, we will cover the following recipes:

- Creating the module files
- Creating a controller
- Adding layout updates
- Adding a translation file
- Adding a block of new products
- Adding an interceptor
- Adding a console command

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 a combination of all these modules.

One of the advantages of a modular architecture is the extendibility. It is easy to add modules that add to or modify the native behavior 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 the root folder of your Magento 2 website. The `app/code/` folder is the folder where all the module development needs to be done.

Access to a command line is also recommended because Magento 2 has a built-in console tool with a lot of commands that we can use during the development.

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`
2. 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=
"urn:magento:framework:Module/etc/module.xsd">
    <module name="Packt_HelloWorld" setup_version="2.0.0">
        <sequence>
            <module name="Magento_Catalog"/>
        </sequence>
    </module>
</config>
```

In Magento 2, there are **XML Style Definition** (**XSD**) files that describes the structure of the configuration XML files. In the `<config>` tag, the correct XSD file is configured.

3. To register the module, we have to create a `registration.php` file in the `app/code/Packt/HelloWorld/` folder with the following content:

   ```
   <?php

   \Magento\Framework\Component\ComponentRegistrar::register(
       \Magento\Framework\Component\ComponentRegistrar::
       MODULE,
       'Packt_HelloWorld',
       __DIR__
   );
   ```

4. Open your terminal and go to the Magento directory. In this directory, run the following commands:
   - `composer install`
   - `php bin/magento cache:clean`
   - `php bin/magento setup:upgrade`
5. When everything is OK, you can see the name of the module in the output of the last command.
6. To test that the module is installed, open the backend and navigate to **Stores** | **Configuration** | **Advanced** | **Advanced**, and check that the module is present in the list. Ensure that 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 a single folder (code, translations, templates, CSS, and more). These things make 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 initialize the `Packt_HelloWorld` name, the version number, and the sequence.

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

We used the `bin/magento` tool for cleaning the cache and running the upgrade scripts. This tool was introduced in Magento 2 and is a replacement of third-party tools from Magento 1 (such as `n98magerun` and `wiz`).

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

Creating a controller

The first thing that we will do to 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. Ensure that you have this module in your Magento instance. Also, ensure that the full page cache is disabled when you are developing. You can disable this in the backend by navigating to **System** | **Cache Management**.

How to do it...

The following steps show 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`
2. In the `app/code/Packt/HelloWorld/etc/frontend` folder, 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=
"urn:magento:framework:App/etc/routes.xsd">
    <router id="standard">
        <route id="helloworld" frontName="helloworld">
            <module name="Packt_HelloWorld" />
        </route>
    </router>
</config>
```

3. In the last folder, that is, `app/code/Packt/HelloWorld/Controller/Index`, create the `Index.php` file 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()
    {

    }
}
```

4. Clean the cache using the `php bin/magento cache:clean` command.
5. Open your browser and navigate to the `/helloworld` URL of the shop. You will see a white page. This is normal because the controller action is empty.
6. To load the layout of the shop, add the following code in the `index.php` file:

```
/** @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;
}
```

If you still see a white page, the page is cached. You have to flush the cache using the `php bin/magento cache:flush` command. It is recommended that you disable the Full Page Cache, as explained in the beginning of this recipe.

7. We will now create an extra action that redirects us to the `HelloWorld` page and create the `app/code/Packt/HelloWorld/Controller/Index/Redirect.php` file.

8. In this file, add the following content:

   ```
   <?php
   namespace Packt\HelloWorld\Controller\Index;

   class Redirect extends \Magento\Framework\App\Action\Action
   {
       public function execute()
       {
           $this->_redirect('helloworld');
       }
   }
   ```

9. Clean the cache and go to the URL `/helloworld/index/redirect`. We will be redirected to the index action.

10. We can also change the content of the `execute()` method to `$this->_forward('index')`. We will see the same output but the URL 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 each controller can have multiple controller actions. This gives us the following structure of the URL: `<modulename or frontname>/<controllerName>/<actionName>`.

When you compare the controller part with Magento 1, a lot of things have been 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.

> 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 be found in `ProductController` of the `Magento_Catalog` module (`app/code/Magento/Catalog/Controller/Product.php`).

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

If we want to render the layout, we will initialize the `resultPageFactory` instance 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()` method in a controller action, a `301` redirect will be returned to the given URL.

The `_forward()` method 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 won't change. This method is used to translate an SEO-friendly URL (such as a product URL) to the right controller action with the right parameters.

There's more...

When things are not working as you expect, you can use the following tips to make it work:

- Clean the cache. You can do this using the `php bin/magento cache:clean` command.
- Flush the cache. You can do this using the `php bin/magento cache:flush` command.
- Remove the `var/generation` folder. Sometimes, the classes needs to be regenerated.

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 as we have seen in the *Customizing the HTML output* recipe of *Chapter 3*, *Theming*. 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 install the module that we 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 `app/code/Packt/HelloWorld/view/frontend/layout` folder.

2. In this 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="urn:magento:framework:View/
Layout/etc/page_configuration.xsd">
    <body>
        <referenceBlock name="footer_links">
            <block class="Magento\Framework\View\Element\
            Html\Link\Current" name="helloworld-link">
                <arguments>
                    <argument name="label" translate="true"
                    xsi:type="string">Helloworld
                    landing</argument>
                    <argument name="path"
                    xsi:type="string">helloworld/
                    index/index</argument>
                </arguments>
            </block>
        </referenceBlock>
    </body>
</page>
```

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

```
<?xml version="1.0"?>
<page xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
layout="2columns-left" xsi:noNamespaceSchemaLocation=
"urn:magento:framework:View/Layout/etc/
page_configuration.xsd">
    <head>
        <title>Helloworld Landingspage</title>
    </head>
    <body>
        <remove name="wishlist_sidebar" />
    </body>
</page>
```

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

```
<?xml version="1.0"?>
<page_types xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation=
"urn:magento:framework:View/Layout/etc/page_types.xsd">
    <type id="helloworld_index_index" label="HelloWorld
    landing page"/>
</page_types>
```

7. Clean the cache and reload the `/helloworld` page. You will see that the title is similar to what we configured in the XML file and the wishlist block is not present in the left-hand side column.
8. To finish this recipe, we will add a custom template with a custom `Block` class. Create the `app/code/Packt/HelloWorld/Block/Landingspage.php` file 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 getRedirectUrl()
    {
        return $this->getUrl('helloworld/index/redirect');
    }
```

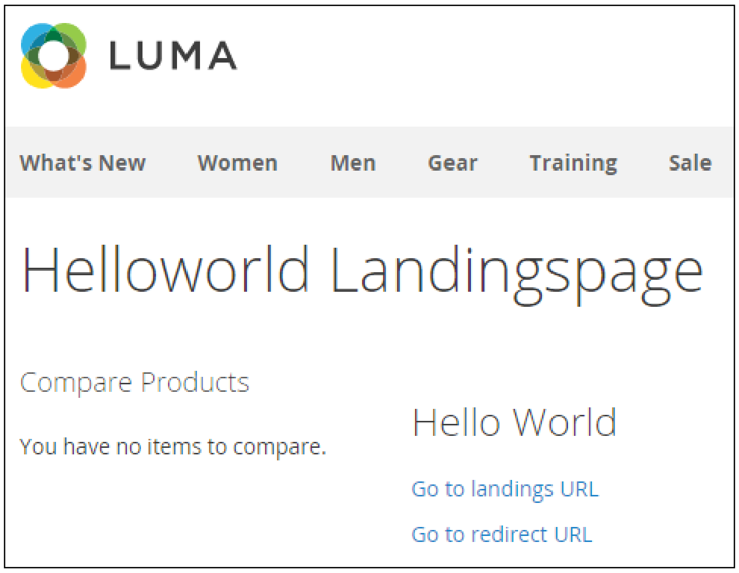
9. Now, we have to create the template where we will call the method from the `Landingspage` class. Create the `app/code/Packt/HelloWorld/view/frontend/templates/landingspage.phtml` file with the following content:

```
<h2>Hello World</h2>
<p>
    <a href="<?php echo $block->getLandingsUrl(); ?>">Go to
    landings URL</a>
</p>
<p>
    <a href="<?php echo $block->getRedirectUrl(); ?>">Go to
    redirect URL</a>
</p>
```

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

```
<referenceContainer name="content">
    <block class="Packt\HelloWorld\Block\Landingspage"
    name="landingsblock" template="Packt_HelloWorld::
    landingspage.phtml" />
</referenceContainer>
```

11. Clean the cache and reload the `/helloworld` URL. You will see something like the following:



Helloworld Landingspage

Compare Products

You have no items to compare.

Hello World

Go to landings URL

Go to redirect URL

How it works...

Layout updates can be placed in modules and themes. In the *Customizing the HTML* recipe out of *Chapter 3*, *Theming*, we explained how layout updates work in themes, but it is also possible to do the same principle in a module.

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

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

- `adminhtml`
- `base`
- `frontend`

As the name suggests, the `adminhtml` folder 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 these folders, the following structure is the internal folder structure that is used:

- `layout` (for layout update XML files)
- `templates` (for `.phtml` templates)
- `web` (for static files, such as LESS, JavaScript, and 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 also has its own handle in the structure `<module front name>_<controllername>_<actionname>`. For the `helloworld` landingspage, is this `helloworld_index_index` file. In the `helloworld_index_index.xml` file, we have placed the layout instructions of that page. The default handle, `default.xml`, is loaded on all pages.

In that file, we created a layout instruction that defines 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.

In Magento 1, we used the `$this` command to call methods from the block class. In Magento 2, we will use the `$block` variable for this.

The guideline is to use the `.phtml` files for the rendering of the HTML output. These files may not contain a log of th ePHP code. The PHP code is written in the block files and the HTML code in the `.phtml` files. In the `.phtml` files, we can call methods from the block class.

Adding a translation file

Magento is made to run in multiple languages. This means that the interface and content needs to be translatable in the configured languages.

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

Getting ready

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

How to do it...

The following procedure demonstrates 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>
   ```

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

   ```
   "Test translation","Translation to test"
   ```

6. 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`.
7. If we want, for example, a French translation, we have to create the `fr_FR.csv` file with the following content:

   ```
   "Test translation","Test traduction"
   ```

8. 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 `php bin/magento i18n:collect-phrases app/code/<Vendor name>/<Module name>` command 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 `translation` table
- 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 doesn'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 (that is, the untranslated string).

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 has now been 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 the new functionality.

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

Getting ready

In our module, we will create a block that will load a product collection. This product collection will be used in the template, which will show the newest products of the shop.

Ensure that you have the module of the previous recipe installed.

How to do it...

The following steps demonstrate how to start with adding the block with new products:

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

    ```
    <?php
    namespace Packt\HelloWorld\Block;

    use Magento\Framework\View\Element\Template;

    class Newproducts extends Template
    {

    }
    ```

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

    ```
    <block class="Packt\HelloWorld\Block\Newproducts"
    name="new_products" template="Packt_HelloWorld::
    newproducts.phtml"/>
    ```

6. Clean the cache and reload the `/helloworld` page. You will see that the **New Products** title is visible.
7. Create a constructor in the block class that initializes the product collection factory. We can do this by adding the following code in that class (the `app/code/Packt/HelloWorld/Block/Newproducts.php` file):

    ```
    private $_productCollectionFactory;

    public function __construct(
        Template\Context $context,
        \Magento\Catalog\Model\ResourceModel\Product\
        CollectionFactory $productCollectionFactory,
        array $data = [])
    {
    ```

```
        parent::__construct($context, $data);

        $this->_productCollectionFactory = $productCollectionFactory;
    }
```

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

```
public function getProducts() {
    $collection = $this->_productCollectionFactory->create();

    $collection
        ->addAttributeToSelect('*')
        ->setOrder('created_at')
        ->setPageSize(5);

    return $collection;
}
```

9. The last step is to call the method in the template and generate an HTML file for it. The code of the template is as follows:

```
<h2>New products</h2>

<ul>
<?php foreach ($block->getProducts() as $product): ?>
    <li><?php echo $product->getName() ?></li>
<?php endforeach; ?>
</ul>
```

10. 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()` method. This method returns the latest five products of the webshop. In this method, we created a query that uses the Magento collections. With Magento collections, we can get data from the database. A collection builds a 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 one database table because it uses the **Entity Attribute Value** system (**EAV**). The Magento collections generate 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 `CollectionFactory` product to work with the collection functions. We initialized this class in the constructor and used it in the `getProducts()` method.

When we run the `create()` function on `CollectionFactory`, a product collection will be returned. It is like doing a collection using the `getCollection()` method on a Magento model, but because this method is deprecated, we have to use `CollectionFactory`.

Adding an interceptor

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

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

Getting ready

To explore the possibilities of dependency injection of Magento 2, we need the module that we created in the previous recipes.

How to do it...

The following steps describe how we can modify the behavior of some classes, which is a new concept in Magento 2 that replaces the rewriting of classes in Magento 1:

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

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

[The letters `di` in the `di.xml` file stands for Dependency Injection.]

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

```
<?php
namespace Packt\HelloWorld\Plugin\Catalog;

use Magento\Catalog\Model\Product;

class ProductAround
{
    public function aroundGetName($interceptedInput)
    {
        return "Name of product";
    }
}
```

[It is highly recommended that you use around in the method name because you can also write interceptors that are executed before or after a method.]

3. Clean the caches and regenerate the classes by removing the `var/generation` folder.
4. Reload a product page and you will see that every product name is now the name of product.
5. To undo this, comment the `<type>` tag and contents of the `di.xml` file and regenerate the classes by removing the `var/generation` folder. Also, don't forget to clean the cache.
6. Reload the 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 method in Magento.

With interception, we can execute the code before, after, and around any method of a class. This gives a lot of possibilities to add behavior to Magento.

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

The overrides are done in the `Packt\HelloWorld\Plugin\Catalog\ProductAround` class. In this class, we did a modification of the `getName()` method of the original class using the `aroundGetName()` method.

To test our code, we had to create the generated classes. We can do this by removing the `var/generation` folder or by running the `php bin/magento setup:di:compile` command. The cache also needs to be cleaned because we changed things in the configuration XML files.

This command creates generated classes that will be placed in the `var/generation` folder. Without generating the classes, the configuration in the `di.xml` file will not load. This is also the reason why you have to do this when installing or upgrading a new module.

Dependency injection replaces the class rewrite system in Magento 1. With dependency injection, you can intercept every method that is called in a class. With the rewrite system of Magento 1, you could not do this with abstract classes.

It is also possible to execute code before and after a method is called.

See also

A lot of things are possible with Dependency Injection. For more information how it is integrated in Magento, you can read the documentation on the Magento site:

`http://devdocs.magento.com/guides/v2.0/extension-dev-guide/depend-inj.html`.

More information about the dependency injection design pattern can be found on the following URL:

`https://en.wikipedia.org/wiki/Dependency_injection`.

Adding a console command

Another new thing in Magento 2 is the built-in command-line tool. In this chapter, we used this tool to clean the cache, for example.

Within a module, it is possible to extend this tool with custom commands, and this is the thing that we will do in this recipe.

Getting ready

This recipe will build further on the module that we have created in this chapter. If you don't have the code, you can install the starter files.

How to do it...

In the next steps, we will create a simple console command that will print some output to the console. Using this principle, you can create your own commands to automate some tasks:

1. For a custom console command, we have to add the following configuration in the `di.xml` file of the module. Paste the following code in that file as child of the `<config>` tag:

```
<type name="Magento\Framework\Console\CommandList">
    <arguments>
        <argument name="commands" xsi:type="array">
            <item name="helloWorldCommand"
            xsi:type="object">Packt\HelloWorld\Console\
            Command\HelloWorldCommand</item>
        </argument>
    </arguments>
</type>
```

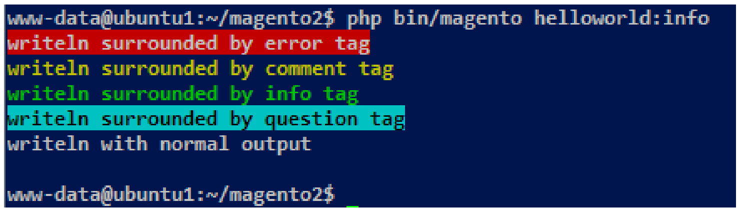
2. Next, we will create the `app/code/Packt/HelloWorld/Console/Command/HelloWorldCommand.php` file with the following content:

```
<?php

namespace Packt\HelloWorld\Console\Command;

use Symfony\Component\Console\Command\Command;
use Symfony\Component\Console\Input\InputInterface;
use Symfony\Component\Console\Output\OutputInterface;
use Symfony\Component\Console\Input\InputOption;

class HelloWorldCommand extends Command
{

}
```

3. In the previous step, we created the class for the command. To initialize the command, we have to add the following content to it:

```
const INPUT_KEY_EXTENDED = 'extended';

protected function configure()
{
    $options = [
        new InputOption(
            self::INPUT_KEY_EXTENDED,
            null,
            InputOption::VALUE_NONE,
            'Get extended info'
        ),
    ];
    $this->setName('helloworld:info')
        ->setDescription('Get info about installation')
        ->setDefinition($options);
    parent::configure();
}

protected function execute(InputInterface $input,
OutputInterface $output)
{
    $output->writeln('<error>' . 'writeln surrounded by
    error tag' . '</error>');
    $output->writeln('<comment>' . 'writeln surrounded by
    comment tag' . '</comment>');
    $output->writeln('<info>' . 'writeln surrounded by info
    tag' . '</info>');
    $output->writeln('<question>' . 'writeln surrounded by
    question tag' . '</question>');
    $output->writeln('writeln with normal output');

    if ($input->getOption(self::INPUT_KEY_EXTENDED)) {
        $output->writeln('');
        $output->writeln('<info>'.'Extended parameter is
        given'.'</info>');
    }

    $output->writeln('');
}
```

4. Clean the cache, remove the `var/generation` folder, and run the `php bin/magento` command. You will see that the `helloworld:info` command will be in the list.
5. When you run the command, you will see the following output:



```
www-data@ubuntu1:~/magento2$ php bin/magento helloworld:info
writeln surrounded by error tag
writeln surrounded by comment tag
writeln surrounded by info tag
writeln surrounded by question tag
writeln with normal output

www-data@ubuntu1:~/magento2$ _
```

6. When you run the command with the `extended` parameter, you will see some extra output. To do this, we have to run the command as follows:

   ```
   php bin/magento helloworld:info --extended
   ```

How it works...

To register the console class to the command list, we had to create an extra argument for the `Magento\Framework\Console\CommandList` class in the `di.xml` file. In this file, we refer to the `Packt\HelloWorld\Console\Command\HelloWorldCommand` class for our custom command.

In the `configure()` method, we registered the name of the command, the description, and the other options. In this case, we initialized an optional input option.

The `execute()` method is made to execute the command. The `$input` parameter contains the input of the command, such as the options and arguments. With the `$output` parameter, we can modify the output of the command. This parameter is used to write output to the console with the `write()` and `writeln()` methods.

In this recipe, we worked with some colors to style the console output. The text between the error, comment, information, and question tags will be rendered in a different color, as we have seen in this recipe.

We also had an optional parameter called extended. To get the value of this parameter, we can use the `getOption()` method of the `$input` parameter. When the parameter is set without value, it will return `true`. If the parameter isn't set, it will return `false`. If a text is given to the parameter, it will return the text.

See also...

The Magento console is built using the **Symfony console** component. More information about how to use the Symfony console can be found at the following URL:

`http://symfony.com/doc/current/components/console/introduction.html`