## 1.Module

### 1.1 Create module

Create module which is  extended from Magento\_Catalog module.Please refer this article for Module Creation.  
<https://ktree.com/create-magento-2-module-custom.html>

### 1.2 Folder structure

### 

### 1.3 List of created files other then default files

1. app/code/KTree/Custom/view/frontend/layout/catalog\_product\_view.xml
2. app/code/KTree/Custom/view/frontend/web/js/mymodal-component.js
3. app/code/KTree/Custom/view/frontend/requirejs-config.js
4. app/code/KTree/Custom/view/frontend/templates/catalog/product/view/post\_requirements.phtml

### 2. Add Popup template to product view page

File: app/code/KTree/Custom/view/frontend/layout/catalog\_product\_view.xml

<?xml version="1.0"?>

<page layout="2columns-right" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:noNamespaceSchemaLocation="urn:magento:framework:View/Layout/etc/page\_configuration.xsd">

<body>

   <referenceContainer name="product.info.form.content">

        <block name="product.info.post.requirements"

          template="KTree\_Custom::catalog/product/view/post\_requirements.phtml"

           after="product.info.addtocart" />

    </referenceContainer>

</body>

</page>

### 2.1.Add Popup custom js to Custom Module

File : app/code/KTree/Custom/view/frontend/web/js/mymodal-component.js

require(

        [

            'jquery',

            'Magento\_Ui/js/modal/modal'

        ],

        function(

            $,

            modal

        ) {

            var options = {

                type: 'popup',

                responsive: true,

                innerScroll: true,

                buttons: [{

                    text: $.mage.\_\_('Continue'),

                    class: 'mymodal1',

                    click: function () {

                        this.closeModal();

                    }

                }]

            };

            var popup = modal(options, $('#myModal'));

            $("#popupButton").on('click',function(){

                $("#myModal").modal("openModal");

            });

        });

Add above js-component to product view page. There are 2 ways to add custom js file to custom module.

* Using RequireJs-config Js file
* Using template

#### Using RequireJs-config Js file

#### By using requirejs  configurations we can add  custom js  file to custom module

**2.1.1 Paths**

Paths is used when you want to relate your base url path with custom module path.Path have the same properties as the baseUrl.If the path is started from “/” or any url “http://” then it will not relate to the base url.

var config = {

    "baseUrl": "KTree/Custom",

    "paths": {

        "sample": "web/js"

    },

};

require( ["sample/sample1"],

    function(samplemodule) {

    }

);

#### Now, samplemodule is reffered to the file at path “KTree/Custom/web/js/sample1.js”.

**2.1.2 deps**

If require js configurations depends upon some dependencies, i.e. you want to load some dependencies before your requires js called.

**2.1.3 shim**

It used when there are some scripts, to whome you want to declare globally, and those scripts not using define() to define values, if those scripts already used the define() then it will not work correctly.  
In our example the final requirejs-config.js like below  
FIle:app/code/KTree/Custom/view/frontend/requirejs-config.js

var config = {

    paths: {

         'mypopup': "KTree\_Custom/js/mymodal-component"

      },

    shim: {

    'mypopup': {

        deps: ['jquery']

    }

  }

};

Here, we are loading mymodal-component js component to our custom module using requirejs configurations   
Initialize or call custom js using mage-init attribute

<button data-mage-init='{"mypopup": {}}' type="button" class="btn btn-primary btn-lg" id="popupButton" data-toggle="modal"

  data-target="#myModal"><?php

echo \_\_('Post Requirement');

?></button>

### 2.2 Using phtml File

Add following script to app/code/KTree/Custom/view/frontend/templates/catalog/product/view/post\_requirements.phtml template file

<script>

  require(['jquery', 'KTree\_Custom/js/mymodal-component'],

      function ($) {

          return;

      }

  );

  </**script**>

Note: In our example we added custom js file using Phtml file,the script like below

File: app/code/KTree/Custom/view/frontend/templates/catalog/product/view/post\_requirements.phtml

<button type="button" class="btn btn-primary btn-lg" id="popupButton" data-toggle="modal"

  data-target="#myModal"><?php echo \_\_('Post Requirement')?></button>

  <!-- Modal -->

  <div class="modal fade" id="myModal" role="dialog" style="display:none;">

    <div class="modal-dialog">

      <!-- Modal content-->

      <div class="modal-content">

        <div class="modal-header">

          <h4 class="modal-title"><?php echo \_\_('Post Requirement')?></h4>

        </div>

        <div class="modal-body">

          <form action="">

      <div class="form-group">

        <label for="name"><?php echo \_\_('Name');?>:</label>

        <input type="email" class="form-control" id="name" placeholder="<?php echo \_\_('Enter name')?>" name="name">

      </div>

      <div class="form-group">

        <label for="requirement"><?php echo \_\_('Requirement') ?> :</label>

      <textarea class="form-control" rows="5" id="requirement"></textarea>

  </div>

      <button type="submit" class="btn btn-default"><?php echo \_\_('Submit')?></button>

    </form>

        </div>

      </div>

    </div>

  </div>

<script>

  require(['jquery', 'KTree\_Custom/js/mymodal-component'],

      function ($) {

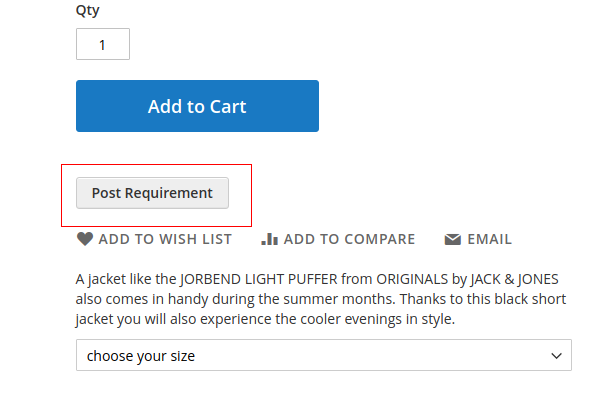
          return;

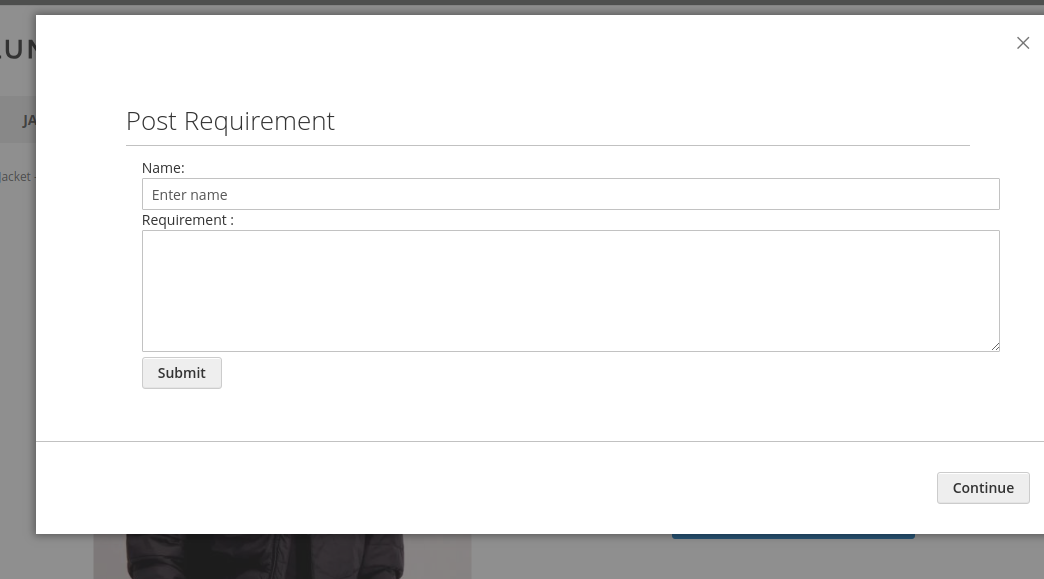
      }

  );

  </script>

* The full source code can be downloaded from this Github link [Add popup to product view page](https://github.com/KtreeOpenSource/Magento2Examples/tree/master/PopupPDP)
* After completion above steps, please run the "php bin/magento setup:upgrade" command and "php bin/magento setup:static-content:deploy"
* The final output like below





I am going to explain how to create and call modal popup widget in Magento 2.

First, we have a button. Clicking it will display a pop-up.

<button id="modal-btn">Open Modal</button>

## ****Create content for popup****

<div id="modal-content">

<div class="modal-inner-content">

<h2>Modal Title</h2>

<p>Modal content.....</p>

</div>

</div>

## ****Use Magento 2 modal widget****

We now need to require both jQuery and jQuery UI. To do this, within your JS script, add the following code:

<script>

require(

[

'jquery',

'Magento\_Ui/js/modal/modal'

],

function($, modal) {

// Your JS

}

);

</script>

## ****Set options for modal**** ****widget****

var options = {

type: 'popup',

responsive: true,

innerScroll: true,

title: 'Pop-up title',

buttons: [{

text: $.mage.\_\_('Close'),

class: 'modal-close',

click: function (){

this.closeModal();

}

}]

};

There are a lot of options you can set. You can see the full list [here](https://devdocs.magento.com/guides/v2.4/javascript-dev-guide/widgets/widget_modal.html#modal_options).

## ****Initialize modal**** ****widget****

modal(options, $('#modal-content'));

Creating an event when clicking the button will display a popup:

$("#modal-btn").on('click',function(){

$("#modal-content").modal("openModal");

});

## ****Finish****

When all of the above steps are done, your script should look like this:

<script>

require(

[

'jquery',

'Magento\_Ui/js/modal/modal'

],

function($, modal) {

var options = {

type: 'popup',

responsive: true,

innerScroll: true,

title: 'Pop-up title',

buttons: [{

text: $.mage.\_\_('Close'),

class: 'modal-close',

click: function (){

this.closeModal();

}

}]

};

modal(options, $('#modal-content'));

$("#modal-btn").on('click',function(){

$("#modal-content").modal("openModal");

});

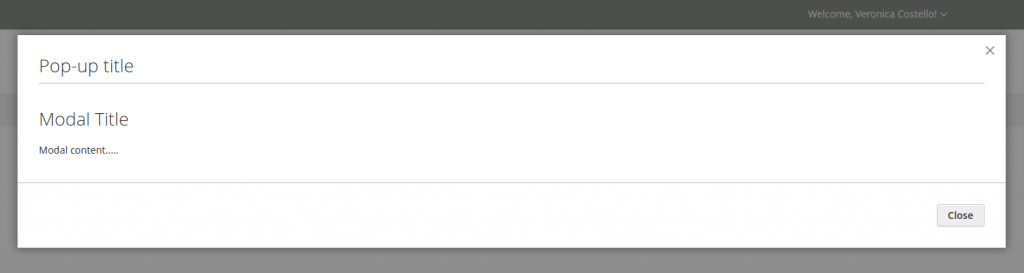
}

);

</script>

## ****Result****

We have a pop-up showed up when the button is clicked:

Popup is created using Magento 2 modal widget

Hopefully, this post is useful to you.

2. Component File Structure

After providing several basic knowledge about the definition as well as the structure, in

this part, I will go through the different file structure for types of components. The application of Magento will look for the files which create a component including configuration files in specific positions inside the component file structure. In order to ensure the component type works properly, you need to follow the predefined file structures.

Root directory

A root directory of the component matches its name and also includes all its subdirectories and files. Depend on the way you installed Magento, your component’s root directory can be put in one of the two following places:

* <Magento install directory>/app: This is an advisable location for component development. By using Cloning the Magento 2 GitHub repository, you can set up this environment.
  + Apply app/code for modules
  + Apply app/design/frontend for storefront themes
  + Apply app/design/adminhtml for Admin themes
  + Apply app/i18n for language packages
* <Magento install directory>/vendor: This location can be found in the installations which the composer create-project is used to install the Magento 2 metapackage, with which the CE or EE code is downloaded. This location can also be found if you extract the compressed Magento 2 archive to install Magento.

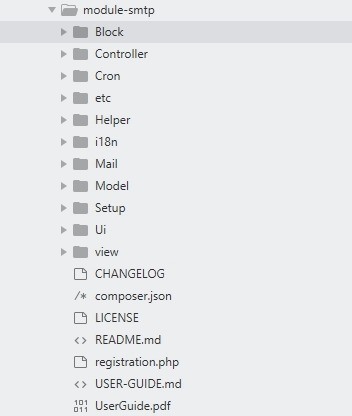
Required files

All components require the following files:

* registration.php: This is the file which will register your component with Magento. In this file, the component’s root directory name is used as the component name. Therefore, by default, components are installed in the <Magento root dir>/vendor directory.
* etc/module.xml: This is the file which defines the basic information about the component, for example, component dependencies and version number. The version number will be used to determine which schema and data in order to update when you execute bin/magento setup:upgrade.
* composer.json: This is the file which defines the dependencies that the component will need at runtime.

3. Module file structure

In the picture below, you will see how a typical file structure looks like:



Learn more: [How to create a custom module for Magento 2](https://www.mageplaza.com/magento-2-module-development/)

Common directories

Here are some module directories which are common:

* Block: includes PHP view classes as part of module logic Model View Controller(MVC) vertical implementation.
* Controller: includes PHP controller classes as part of module logic MVC vertical implementation.
* etc: includes configuration files; especially the module.xml, which is required.
* Model: includes PHP model classes as part of module logic MVC vertical implementation.
* Setup: includes classes for module database structure and data setup. These data are invoked when installing or upgrading.

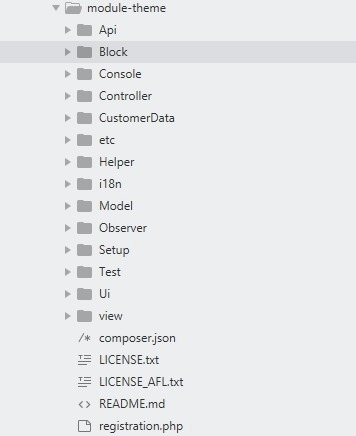
Additional directories

These folders can be added for configuration or other items ancillary functions such as localization,plugin-ins, and layout files.

* Api: includes any PHP classes which are exposed to the API.
* Console: includes CLI commands.
* Cron: includes cron job definitions.
* CustomerData: includes section files.
* Helper: includes aggregated functionality.
* i18n: includes localization files.
* Observer: includes files for executing commands which are from the listener.
* Plugin: includes any needed plug-ins.
* UI: includes data generation files.
* view: includes view files, containing static view files, email templates, design templates, and layout files.

4. Theme file structure

Below are how a typical theme file structure looks like:



Learn more: [How to create a custom theme for Magento 2](https://www.mageplaza.com/devdocs/how-to-create-magento-2-theme.html)

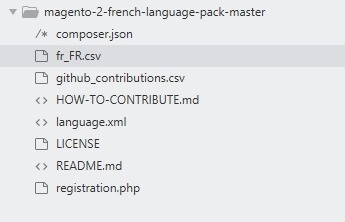
Common directories

Following are the typical theme directories:

* etc: Includes configuration files like the view.xml file that has image configurations for all images and thumbnails.
* i18n: Translation dictionaries, if there any available.
* media: Theme preview images can be put in this directory.
* web: Optional directory which has static files organized into the following files:
  + css/source: Includes a theme’s less configuration files which invoke mixins for global elements from the Magento UI library, and also the theme.less file which overrides the default variables values.
  + css/source/lib: Includes view files which override the UI library files that are stored in lib/web/css/source/lib.
  + fonts: you can place the different fonts for your theme in this folder.
  + images: folder of static images.
  + js: The folder which is used for your JavaScript files.

5. Language package file structure

Below are how a typical directory structure for [the French language](https://www.mageplaza.com/magento-2-french-language-pack.html) looks like:

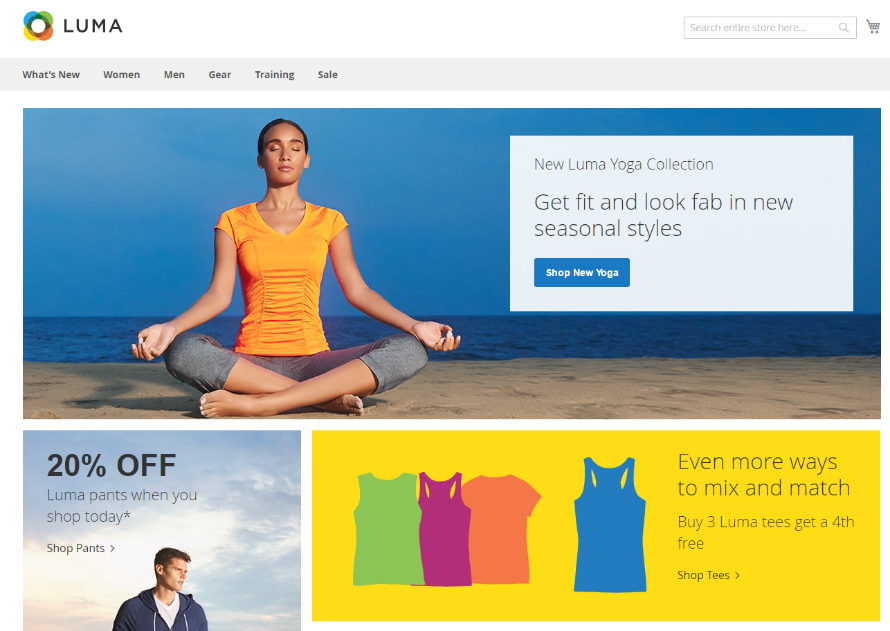


The only directory which is required for a language pack is the top-level directory. For better performance, I recommend you use the directory name that matches the ISO code to identify the locale.

# **Create Custom Theme in Magento 2 - Theme Development Tutorial Step by Step**

A theme determines how the elements and the entire website are presented to the user. It combines a combination of custom templates, layouts, styles, and graphics to give a Magento store an uniform experience. There are number of ways to **create a custom themes in Magento 2**. It uses theme.xml, introduced in Magento 1.9, and a new folder structure in Magento 2 that works in a similar way to Magento 1.x, but has the added advantage that you can select unlimited parent themes to inherit from, and you can configure the theme.xml file in your theme.

Let’s say you want to **create a brand new theme** based on the new Magento “Blank” theme. First, you would create a new folder in app/design/frontend, for example Mageplaza/simple. You would then create a theme.xml file in this directory, path: app/design/frontend/Mageplaza/simple (it is probably best to copy it from app/design/frontend/Magento/blank/theme.xml), In this case, we want to base on Magento 2’s Blank theme.



## 9 Simple Steps To Create Magento 2 Theme:

* [1. Creating a Magento theme folder](https://www.mageplaza.com/devdocs/how-to-create-magento-2-theme.html#1-creating-a-magento-theme-folder)
* [2. Declare your theme](https://www.mageplaza.com/devdocs/how-to-create-magento-2-theme.html#2-declare-your-theme)
* [3. Composer package](https://www.mageplaza.com/devdocs/how-to-create-magento-2-theme.html#3-composer-package)
* [4. registration.php file](https://www.mageplaza.com/devdocs/how-to-create-magento-2-theme.html#4-registrationphp-file)
* [5. Creating static files, folders](https://www.mageplaza.com/devdocs/how-to-create-magento-2-theme.html#5-creating-static-files-folders)
* [6. Configure catalog product images](https://www.mageplaza.com/devdocs/how-to-create-magento-2-theme.html#6-configure-catalog-product-images)
* [7. Declare Theme Logo](https://www.mageplaza.com/devdocs/how-to-create-magento-2-theme.html#7-declare-theme-logo)
* [8. Basic layout elements](https://www.mageplaza.com/devdocs/how-to-create-magento-2-theme.html#8-basic-layout-elements)
* [9. Layout files types and conventions](https://www.mageplaza.com/devdocs/how-to-create-magento-2-theme.html#9-layout-files-types-and-conventions)

So your app/design/frontend/mageplaza/theme.xml file should look something like this:

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

<title>Mageplaza Simple</title>

<parent>Magento/blank</parent>

</theme>

### Theme structure

#### **Theme Folder structure**

app/design/frontend/Mageplaza/

├── simple/

│ ├── etc/

│ │ ├── view.xml

│ ├── web/

│ │ ├── images

│ │ │ ├── logo.svg

│ ├── registration.php

│ ├── theme.xml

│ ├── composer.json

<Vendor> is theme vendor. e.g: Mageplaza

<theme> is theme name. e.g: simple

Ok, let’s go

## 1. Creating a Magento theme folder

Creating a folder for the theme:

* Go to app/design/frontend
* Creating a vendor folder app/design/frontend/<vendor> e.g: app/design/frontend/Mageplaza
* Create a theme folder app/design/frontend/<vendor>/<theme> e.g: app/design/frontend/Mageplaza/simple

app/design/frontend/

├── Mageplaza/

│ │ ├──simple/

│ │ │ ├── ...

│ │ │ ├── ...

## 2. Declare your theme

Now we have folder app/design/frontend/Mageplaza/simple , now create a file named theme.xml that define basic information about theme such as: Name, parent theme (in case your theme inherits from an existing theme), preview image

<theme xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="urn:magento:framework:Config/etc/theme.xsd">

<title>Mageplaza Simple</title> <!-- your theme's name -->

<parent>Magento/blank</parent> <!-- the parent theme, in case your theme inherits from an existing theme -->

<media>

<preview\_image>media/preview.jpg</preview\_image> <!-- the path to your theme's preview image -->

</media>

</theme>

## 3. Composer package

**Composer** is a tool for dependency management in PHP. It allows you to declare the libraries your project depends on and it will manage (install/update) them for you.

To distribute your theme as a package, add a composer.json file to the theme directory and register the package on a packaging server.

composer.json example:

{

"name": "mageplaza/simple",

"description": "N/A",

"require": {

"php": "~5.5.0|~5.6.0|~7.0.0",

"magento/theme-frontend-blank": "100.0.\*",

"magento/framework": "100.0.\*"

},

"type": "magento2-theme",

"version": "100.0.1",

"license": [

"OSL-3.0",

"AFL-3.0"

],

"autoload": {

"files": [

"registration.php"

]

}

}

## 4. registration.php file

You can add the following content to register the theme to Magento 2

<?php

/\*\*

\* Copyright © 2015 Magento. All rights reserved.

\* See COPYING.txt for license details.

\*/

\Magento\Framework\Component\ComponentRegistrar::register(

\Magento\Framework\Component\ComponentRegistrar::THEME,

'frontend/Mageplaza/simple',

\_\_DIR\_\_

);

You should change Mageplaza, simple to your vendor, theme name.

## 5. Creating static files, folders

In a design, there are many static files such as javascript, css, images and fonts. They are stored in separate folders in web of theme package.

Here are the structure

app/design/<area>/Mageplaza/simple/

├── web/

│ ├── css/

│ │ ├── source/

│ ├── fonts/

│ ├── images/

│ ├── js/

#### **Tips**

In Magento 2, theme or extension development, when you update any files in app/design/<area>/Mageplaza/simple/web folder, you have to static folders which located at pub/static and var/view\_preprocessed Otherwise, you still there is no change in frontend.

## 6. Configure catalog product images

As you can see in the theme structure I mentioned above, there is a file called etc/view.xml. This is configuration file. This file is required for the [Magento 2 theme](https://www.mageplaza.com/magento-2-themes/) but it is optional if it exists in the parent theme.

Go to app/design/<area>/Mageplaza/simple/ and create a folder etc and file view.xml You can copy the view.xml file in parent theme such as **Blank theme** app/design/frontend/Magento/blank/etc/view.xml

Ok, let update the image configuration for catalog product grid page.

<image id="category\_page\_grid" type="small\_image">

<width>250</width>

<height>250</height>

</image>

In view.xml, image properties are configured in the scope of element:

<images module="Magento\_Catalog">

...

<images/>

Image properties are configured for each image type defined by the id and type attributes of the <image> element:

<images module="Magento\_Catalog">

<image id="unique\_image\_id" type="image\_type">

<width>100</width> <!-- Image width in px -->

<height>100</height> <!-- Image height in px -->

</image>

<images/>

## 7. Declare Theme Logo

In Magento 2 default, it uses <theme\_dir>/web/images/logo.svg, in your theme, you can change to different file formats such as png, jpg but you have to declare it.

Logo size should be sized 300x300px and you open file <theme\_dir>/Magento\_Theme/layout/default.xml

<page xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="urn:magento:framework:View/Layout/etc/page\_configuration.xsd">

<body>

<referenceBlock name="logo">

<arguments>

<argument name="logo\_file" xsi:type="string">images/custom\_logo.png</argument>

<argument name="logo\_img\_width" xsi:type="number">300</argument>

<argument name="logo\_img\_height" xsi:type="number">300</argument>

</arguments>

</referenceBlock>

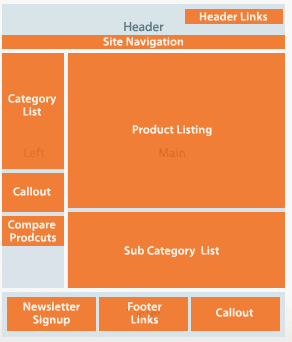
</body>

</page>

## 8. Basic layout elements

The basic components of Magento page design are blocks and containers.

A container exists for the sole purpose of assigning content structure to a page. A container has no additional content except the content of included elements. Examples of containers include the header, left column, main column, and footer.



## 9. Layout files types and conventions

#### **Module and theme layout files**

The following terms are used to distinguish layouts provided by different application components:

* Base layouts: Layout files provided by modules. Conventional location:
  + Page configuration and generic layout files: <module\_dir>/view/frontend/layout
  + Page layout files: <module\_dir>/view/frontend/page\_layout
* Theme layouts: Layout files provided by themes. Conventional location:
  + Page configuration and generic layout files: <theme\_dir>/<Namespace>\_<Module>/layout
  + Page layout files: <theme\_dir>/<Namespace>\_<Module>/page\_layout

### Create a theme extending file

Rather than copy extensive page layout or page configuration code and then modify what you want to change, in the [Magento system](https://www.mageplaza.com/kb/magento-2-system-requirements.html), you only need to create an extending layout file that contains the changes you want.

To add an extending page configuration or generic layout file:

<theme\_dir>

|\_\_/<Namespace>\_<Module>

|\_\_/layout

|--<layout1>.xml

|--<layout2>.xml

For example, to customize the layout defined in <Magento\_Catalog\_module\_dir>/view/frontend/layout/catalog\_product\_view.xml, you need to add a layout files with the same name in your custom theme, like following:

<theme\_dir>/Magento\_Catalog/layout/catalog\_product\_view.xml

<theme\_dir>

|\_\_/<Namespace>\_<Module>

|\_\_/page\_layout

|--<layout1>.xml

|--<layout2>.xml

### Override base layouts

Overriding is not necessary if a block has a method that cancels the effect of the originally invoked method. In this case, you can customize the layout by adding a layout file where the canceling method is invoked.

To add an overriding base layout file (to override a base layout provided by the module): Put a layout file with the same name in the following location:

<theme\_dir>

|\_\_/<Namespace\_Module>

|\_\_/layout

|\_\_/override

|\_\_/base

|--<layout1>.xml

|--<layout2>.xml

These files override the following layouts:

* <module\_dir>/view/frontend/layout/<layout1>.xml
* <module\_dir>/view/frontend/layout/<layout2>.xml

### Override theme layouts

To add an overriding theme file (to override a parent theme layout):

<theme\_dir>

|\_\_/<Namespace\_Module>

|\_\_/layout

|\_\_/override

|\_\_/theme

|\_\_/<Parent\_Vendor>

|\_\_/<parent\_theme>

|--<layout1>.xml

|--<layout2>.xml

These files override the following layouts:

* <parent\_theme\_dir>/<Namespace>\_<Module>/layout/<layout1>.xml
* <parent\_theme\_dir>/<Namespace>\_<Module>/layout/<layout1>.xml

#### **Tips!**

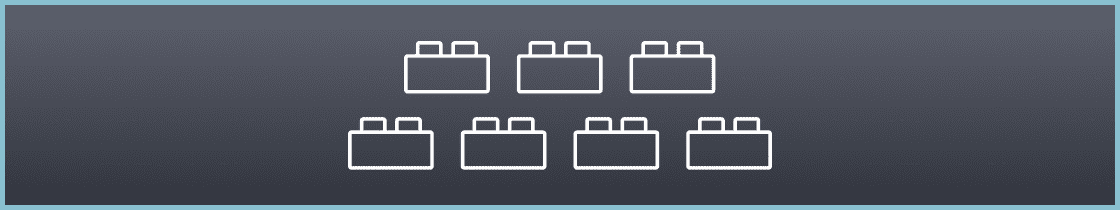
- Changing block name or alias. The name of a block should not be changed, and neither should the alias of a block remaining in the same parent element.

- Changing handle inheritance. For example, you should not change the page type parent handle.

Congrats! Now you have your first simple Magento 2 theme. You can try to create a complexible theme later. Remember to consider all the possible aspects that we share [here](https://www.mageplaza.com/blog/best-magento-enterprise-themes.html) to create an amazing theme, as well as take a look at the below theme collection suggestions. And we are always here to help you, so please comment down below or give us a direct message [here](https://www.mageplaza.com/contact.html) if you have any questions.

## What is a Magento 2 Module?

A Module in Magento 2 is an independent component or a set of components that provide business logic and features. Magento 2 Open Source is a modular application and it supports modularity. It means that all functionality is implemented and delivered in components that are known as Modules. A Component is also known and called an Extension. A module and an extension is basically the same thing.

[](https://magemastery.net/static/45c75129ca87ceff8db3c4ffb545a39e/f43e4/modules-magento-2.png)

Each extension or module in Magento 2 provides a set of capabilities that support the business and functional logic of a module. It allows bringing additional features for the e-commerce store which is based on Magento 2.

### Example

Let's take a Catalog module for example. The Catalog module provides a different set of features which include Category Management, Product Management, Product Types support for Simple and Virtual products, Catalog browsing capability and so on.

Also, the module provides extensibility points for other modules which allow to extend the Catalog module with additional functionality. An example of such an additional module is a Downloadable Product module.

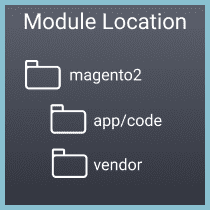
A module in Magento 2 could bring a complex set of features from one side and from the other side it could be a tiny feature that adds an icon to a Product page for example.

## A Module Name

Each module's name consists of two parts. A vendor name and the name of a module. These two parts are concatenated with a \_(underscore) or a \(backslash). As a result, the module name should follow the pattern VendorName\_ModuleName. For example, a Catalog module name is going to be the name of a Magentovendor name and a Catalogmodule name, which is a Magento\_Catalogor Magento\Catalog.

## Where should a Module be located in Magento 2?

That's a very interesting question. There are two locations where a module can be located in a Magento 2 application.

[](https://magemastery.net/static/66406e69a542513a34792680d2b7437d/65ed1/module-location-in-magento-2.png)

The first location is an app/codedirectory. This directory is used to add all custom and 3rd-party Magento 2 modules. This location is usually used by development agencies, internal or in-house developers in order to simplify a development process.

The second location is a vendordirectory. This directory is used during the [composer package management installation](https://magemastery.net/courses/magento-2-beginner/getting-started-with-magento-2). A module installation in Magento 2 with a composer package manager is the recommended way of installing a Magento 2 module.

Learn more about Magento 2 Open Source directory structure in the [*Project Structure Overview*](https://magemastery.net/courses/magento-2-beginner/project-structure-overview) lesson.

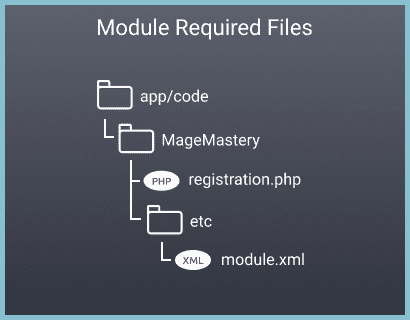
## How to create a Module?

The first thing we have to do in order to create a module is to create a module directory. Usually, all modules are located under the app/codedirectory. If you follow the previous lesson, the codedirectory should be missing under the appdirectory of a Magento 2 project.

Inside the appdirectory, let's create the codedirectory. This is a directory where all modules should be located. Every Magento 2 module should be located under the app/code/VendorName/ModuleNamedirectory location.

Let's create a vendor directory. In this lesson, the vendor is a MageMastery. Inside the app/code/MageMasterydirectory, let's create a module directory. For the module name, this is going to be a FirstModuledirectory name.

There are two required files that have to be created in each Magento 2 module. These files are used to notify to Magento 2 about a module. The registration.phpand module.xmlfiles.

[](https://magemastery.net/static/18be6315c3d0336b09b8197afecd3110/d68e4/module-files.png)

### The registration.php file

The registration.phpfile is used to notify to Magento 2 application about a module in a file system. It also used to provide a location to a Magento 2 module. Place the new registration.phpfile inside the app/code/MageMastery/FirstModuledirectory.

The content of the file is as follows:

**<?php**

use Magento\Framework\Component\ComponentRegistrar;

ComponentRegistrar::register(

ComponentRegistrar::MODULE,

'MageMastery\_FirstModule',

\_\_DIR\_\_

);

The Magento\Framework\Component\ComponentRegistrarclass is used to register a new module in a Magento 2 application.

The static register()method accepts three arguments. The $componentTypeargument accepts a type of a component, which in case of a module is ComponentRegistrar::MODULE. The ComponentRegistrar::MODULEis a constant of the Magento\Framework\Component\ComponentRegistrarclass and it is recommended to use this constant for the module registration.

The second argument of the register()method is the name of the module. As you can see the name is a VendorName\_ModuleName. In the above code sample, the name is MageMastery\_FirstModule.

The final argument is a PHP \_\_DIR\_\_constant. It provides an actual location of the module and is interpreted into an /var/www/magento/app/code/MageMastery/FirstModulein case a Magento 2 application is installed in the /var/www/magentodirectory.

**How does this work?**

One of the allowed locations of the registration.phpfile is app/code/VendorName/ModuleNamedirectory (in case of the current lesson the location is app/code/MageMastery/FirstModule/). The registration.phpfile is scanned and executed during the Magento 2 Open Source application start. The method register()of the ComponentRegistrarclass adds the module name and its location into the global pool of all module's locations. Once added, Magento 2 Open Source then goes through all registered modules paths and reads the configuration of the module.xmlfile.

That is why it is important to have both files to successfully register a Module in Magento 2.

### The module.xml file

The module.xmlfile is used to declare and provide information about the module name and any dependencies to other 3rd-party or out-of-box Magento 2 modules.

The module.xmlfile is a main module's configuration file. It is required to place this file into the module's etcdirectory. Let's create the directory inside the app/code/MageMastery/FirstModuledirectory module.

An example of the XML configuration file is as follow:

<?xml version="1.0"?>

<config>

</config>

This is a basic file content that is used across almost all Magento 2 configuration files. We are going to learn more about XML configuration files in the next lessons.

Additionally, all XML files in Magento 2 contain 2 attributes in the <config>node. The xmlnsattribute provides a location to the XMLSchema instance and the xsi:noNamespaceSchemaLocationattribute provides a path to an XSD (XML Schema Definition) file of a Magento 2 framework. The last one is used to provide XML validation and syntax highlighting capability to a configuration file.

These attributes are not required, however, it is recommended to follow the standard Magento practices and include it into XML files.

As a result, the final module.xmlfile should look like follow:

<?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="MageMastery\_FirstModule" />

</config>

The only required XML node which has to be added is a <module />one. The nameattribute should include the name of the current module, which is MageMastery\_FirstModule.

## How to register a Module?

The last step is to run the command which will enable and install the MageMastery\_Firstmodule.

bin/magento setup:upgrade

You may notice a new module added to the list of modules.

In order to check if the module has been added and enabled in a Magento 2 application. Here I use a grepcommand to check a string in an app/etc/config.phpfile.

grep MageMastery\_FirstModule app/etc/config.php

As a result, a terminal screen should show a

The value 1means that the module is enabled, and 0means that the module is disabled. As we can see from the output, the MageMastery\_FirstModuleis set to 1, which means that the module has been successfully registered in a Magento 2 Open Source application.

Lesson Overview

In this lesson we are going to learn the following:

* How to create a page in Magento 2?
* How to create an Action Controller?
* How to register a new route?
* How to return a JSON data from a Controller?

Before we begin

For this tutorial, I've created the MageMastery\_FirstPagemodule with the registration.phpand module.xmlfiles. And the module has been successfully registered with the bin/magento setup:upgradeCLI command.

The registration.phpfile:

**<?php**

use Magento\Framework\Component\ComponentRegistrar;

ComponentRegistrar::register(

ComponentRegistrar::MODULE,

'MageMastery\_FirstPage',

\_\_DIR\_\_

);

The module.xmlfile:

<?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="MageMastery\_FirstPage" />

</config>

For more information about the module creation and registration, please refer to the previous lesson [A Module in Magento 2](https://magemastery.net/courses/magento-2-beginner/module-in-magento-2).

Registering a new route

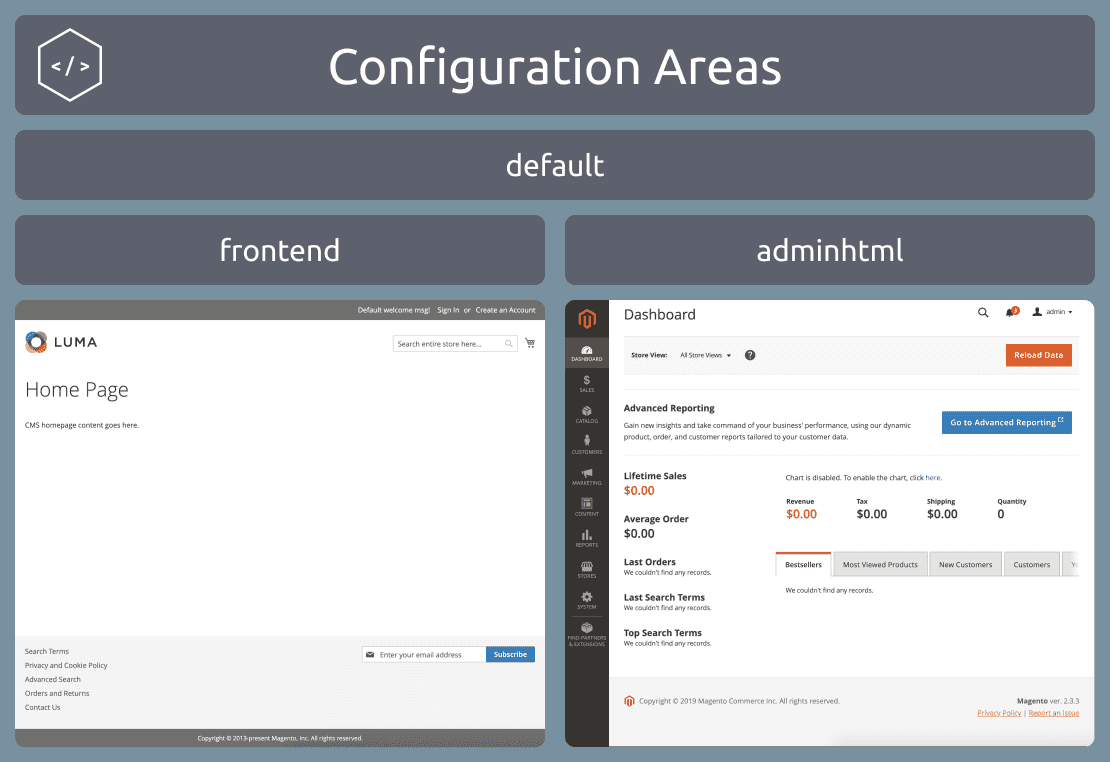
In order to create a new page in Magento 2, which is accessible via a browser, there are two files that have to be created in a Magento 2 module. The first file is routes.xml. The file is responsible for providing information to Magento 2 where to look for a particular controller module's name and additional rules to match a URI.

The second file is a controller class, that should be executed and process the incoming request.

Let's go ahead and create a new routes.xmlfile under the MageMastery/FirstPage/etc/directory.

Configuration in Magento 2

The configuration can be provided for the two different areas: **frontend** and **adminhtml** areas. There is also a **default** that is used to provide a configuration for both areas.

[](https://magemastery.net/static/862b113289708d3e1ed35a13f7660b6d/73caa/1-configuration-areas-in-magento-2.png)

For the storefront configuration changes, we have to add a new frontenddirectory inside the MageMastery/FirstPage/etc/directory. We will talk more about the adminhtmldirectory in the upcoming lessons.

If you haven't created the etc/frontenddirectory this is the right time to do it.

The routes.xml file

Inside the etc/frontenddirectory let's create the routes.xmlconfiguration file. All routes configuration files from all Magento 2 registered modules including the MageMastery/FirstPagemodule will be merged and provided for the execution.

The routes.xmlinitial file state:

<?xml version="1.0"?>

<config xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:noNamespaceSchemaLocation="urn:magento:framework:App/etc/routes.xsd">

</config>

Magento 2 Open Source provides so-called routers, and each router has its own identifier. We are not going to go deeply into the router implementation just now. We are going to use the standard router that is identified as "standard". This is what you see in the code below:

<?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">

</router>

</config>

Inside the standard router node, we can add routes configuration. Each route is responsible for providing a front name that is used to match the route with a request URL. Also, the route should provide a module name, that includes an action controller class.

The route configuration for the MageMastery\_FirstPagemodule and the magemasteryfront name looks as follow:

<route id="magemastery\_firstpage" frontName="magemastery">

<module name="MageMastery\_FirstPage" />

</route>

Note the id argument of the route. It has to be unique across the Magento 2 application. The recommended naming convention for the route id is lower-case module name e.g. magemastery\_firstpageto ensure the uniqueness.

Let's see the final state of the routes.xmlfile:

<?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="magemastery\_firstpage" frontName="magemastery">

<module name="MageMastery\_FirstPage" />

</route>

</router>

</config>

Magento 2 standard router reads the route configuration that starts with the magemasteryrequest URI part, finds a module name with the name MageMastery\_FirstPage, and will check for a controller, which can process the request.

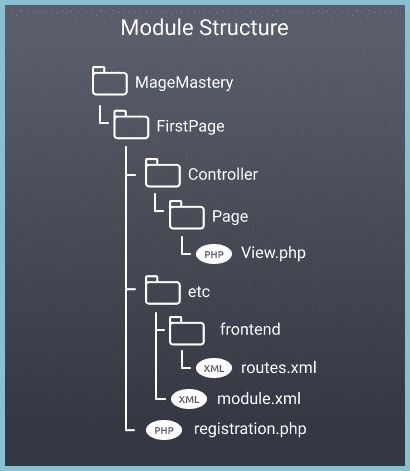
A controller in a module is the second bit or file to be created in this lesson.

Action Controller

All controllers of a module should be located in a directory called Controller. There is another directory level that should exist inside the Controllerdirectory before the class can be located. For this lesson, we are going to create a Pagedirectory in the Controller.

Inside the Pagedirectory, let's create the View.phpfile.

As a result, you should have the following module structure.

[](https://magemastery.net/static/7593e75acaa6cab1ba1c1bb979da299d/d68e4/2-module-structure.png)

The View.phpfile is a MageMastery\FirstPage\Controller\Page\Viewclass. The Viewclass should extend the Magento\Framework\App\Action\Actionclass.

**<?php**

declare(strict\_types=1);

namespace MageMastery\FirstPage\Controller\Page;

use Magento\Framework\App\Action\Action;

class View extends Action

{

public function execute()

{

}

}

The Actionclass requires to provide implementation of the execute()method. Once Magento 2 will resolve the route it will trigger the View::execute()method. The execute()method's logic should be added in order to render something on a front end. For this lesson, we are going to return a JSON response with the message.

In order to return anything from the execute()method, there is a resultFactoryvariable that is provided to us from the Actionparent class. The create()method of the Magento\Framework\Controller\ResultFactoryclass creates an instance of a Magento\Framework\Controller\Result\Jsonclass. The Jsonclass then can be used to convert an array into a JSON and return back from the execute()method.

$jsonResult = $this->resultFactory->create(ResultFactory::TYPE\_JSON);

We can use the $jsonResultto set an array with the key messageand the value to be My First Page.

$jsonResult->setData([

'message' => 'My First Page'

]);

The resulting code of the Viewcontroller class looks as follow:

**<?php**

declare(strict\_types=1);

namespace MageMastery\FirstPage\Controller\Page;

use Magento\Framework\Controller\Result\Json;

use Magento\Framework\App\Action\Action;

use Magento\Framework\Controller\ResultFactory;

class View extends Action

{

public function execute()

{

/\*\* @var Json $jsonResult \*/

$jsonResult = $this->resultFactory->create(ResultFactory::TYPE\_JSON);

$jsonResult->setData([

'message' => 'My First Page'

]);

return $jsonResult;

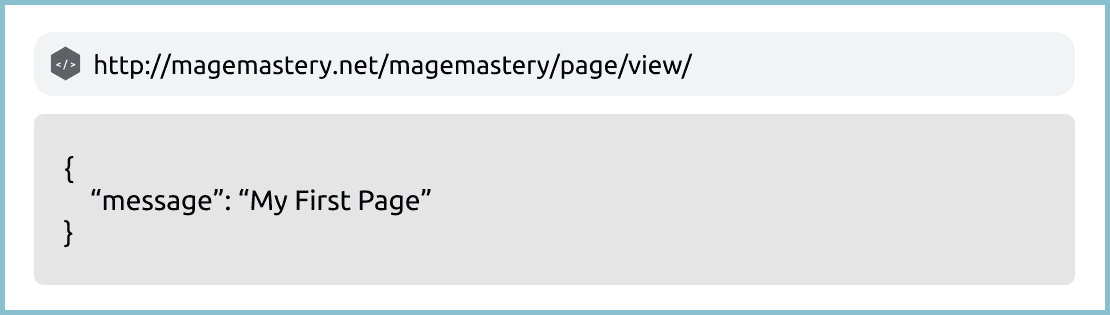
}

}

In order to a result in a browser, the Magento 2 configuration cache has to be cleaned. The cache cleanup can be done with the Magento CLI command cache:clean. The command has to be executed from the Magento 2 project root directory.

bin/magento cache:clean

Navigate to the http://magento2ce.magemastery.net/magemastery/page/viewurl, where the magento2ce.magemastery.net domain name can be different one, depending on what domain name you provided during the [Magento 2 installation](https://magemastery.net/courses/magento-2-beginner/getting-started-with-magento-2).

[](https://magemastery.net/static/3219a3ebac5ce66ca208d1c8fca1b9e1/73caa/3-magento-2-page.png)

Let's have a closer look at /magemastery/page/viewpath for the page. The first part is the magemastery, this is a front name that is configured in the routes.xmlfile. The pageis the name of a directory Pageinside which there is a Viewclass. The View::execute()method returns the JSON object.

Homework

The homework for this lesson. Create a custom controller with the custom JSON message which has to be rendered on a page with the route /magemastery/custom/page.

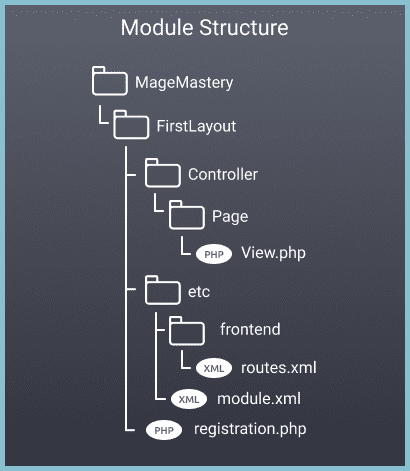
Lesson Overview

In this lesson we are going to learn the following:

* How to create a template?
* How to render a template on a custom Magento 2 page?
* How to add a template via layout configuration file?

Before we begin

Here is the structure of the module, we are going to start with:

[](https://magemastery.net/static/a7b15b731420530d295d9bcee35b9712/d68e4/lesson-6-structure.png)

For this tutorial, I've created the MageMastery\_FirstLayoutmodule with the registration.phpand module.xmlfiles. And the module has been successfully registered with the bin/magento setup:upgradeCLI command.

The registration.phpfile:

**<?php**

use Magento\Framework\Component\ComponentRegistrar;

ComponentRegistrar::register(

ComponentRegistrar::MODULE,

'MageMastery\_FirstLayout',

\_\_DIR\_\_

);

The module.xmlfile:

<?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="MageMastery\_FirstLayout" />

</config>

For more information about the module creation and registration, please refer to the previous lesson [A Module in Magento 2](https://magemastery.net/courses/magento-2-beginner/module-in-magento-2).

In addition to this, I've added a route.xmlfile with an Action Controller class. These two files allows to access a custom page that has been declared in the MageMastery\_FirstLayoutmodule.

The routes.xmlfile content is the following:

<?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="magemastery\_firstlayout" frontName="firstlayout">

<module name="MageMastery\_FirstLayout" />

</route>

</router>

</config>

The routes.xml file is located in the MageMastery/FirstLayout/etc/frontenddirectory. This configuration allows to access a module's controller by using the "firstlayout" URI path of the URL.

And the Action Controller class located in the MageMastery/FirstLayout/Controller/Page/View.phpfile, the content is:

**<?php**

declare(strict\_types=1);

namespace MageMastery\FirstLayout\Controller\Page;

use Magento\Framework\App\Action\Action;

use Magento\Framework\Controller\ResultFactory;

use Magento\Framework\View\Result\Page;

class View extends Action

{

public function execute()

{

/\*\* @var Page $resultPage \*/

return $this->resultFactory->create(ResultFactory::TYPE\_PAGE);

}

}

New Page Type Object

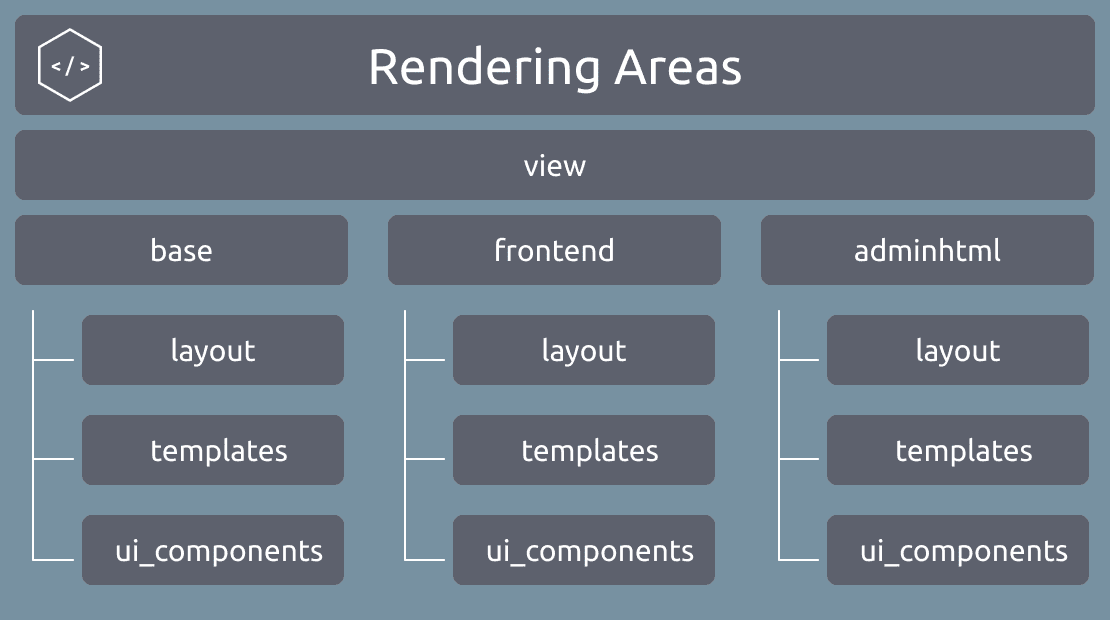
From the [Create a Page in Magento 2](https://magemastery.net/courses/magento-2-beginner/create-page-magento-2) lesson we've learned how to create a custom page and return a JSON-format data. As you've noticed, there is an instance of the Magento\Framework\View\Result\Pageclass returned from the execute()method.

return $this->resultFactory->create(ResultFactory::TYPE\_PAGE);

The idea behind creating a TYPE\_PAGEtype of a result object is to trigger the Magento 2 rendering mechanism.

Rendering

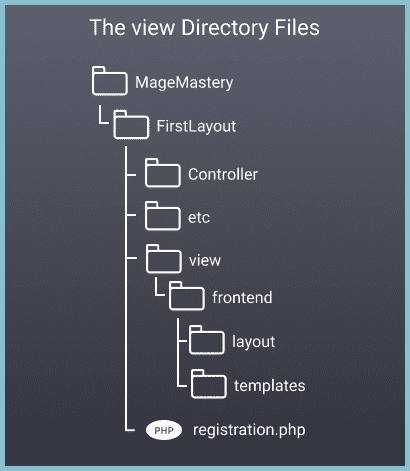
There is a new directory that we've to learn in this lesson. The directory is called view. As we've discussed in the [previous tutorial](https://magemastery.net/courses/magento-2-beginner/create-page-magento-2), Magento 2 application uses different configuration areas. It is also applicable for rendering purposes. The frontend and adminhtml are the two areas we can use in order to render a custom content on a Storefront and Admin area accordingly.

[](https://magemastery.net/static/4c42b842bb04aa4503fcef9f824bef6c/73caa/rendering-areas-with-base.png)

There are other files and directories might be located in the viewdirectories, however, this is outside of the lesson. In the upcoming lessons we will learn about different rendering areas and how it can be used to provide different files for different areas.

In case we want to render something on a Storefront, we have to create a view/frontenddirectory. Inside this directory, all the files should be added and further to be executed in a frontend area. This happens upon Magento 2 rendering. Inside frontenddirectory, there are two directories. The layoutdirectory is used to place layout configuration files of the MageMastery\_FirstLayoutmodule.

The templatesdirectory, contains PHTML templates that have to be rendered on a Magento 2 Storefront.

[](https://magemastery.net/static/06962dafb1c8c68156f039faf3414104/d68e4/lesson-6-view-structure.png)

Layout Configuration File

The new magemastery\_firstlayout\_page\_view.xmllayout configuration file has to be added to the MageMastery/FirstLayout/view/frontend/layoutdirectory.

<?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>

<referenceContainer name="content">

<block class="Magento\Framework\View\Element\Template"

name="magemastery.first.layout.example"

template="MageMastery\_FirstLayout::example.phtml" />

</referenceContainer>

</body>

</page>

The format of the layout file is pretty much similar to all the XML files in Magento 2. The root node <page />allows to add configuration and templates that have to be rendered on a page.

In order to add a custom example.phtmltemplate from the view/frontend/templatesdirectory, the XML declaration should be created and added to the layout configuration file.

There are 3 pieces of information or node arguments have to be declared as part of the blocknode. Let's have a look at the blockdeclaration example:

<block class="Magento\Framework\View\Element\Template"

name="magemastery.first.layout.example"

template="MageMastery\_FirstLayout::example.phtml" />

The classargument is responsible for providing a Block class name. The Block class is used to render a PHTML template file. The nameof the blockis magemastery.first.layout.example. The name has to be unique and can be used to reference to a block from the other layout configuration files and templates. For example, the contentname or alias of a block is defined in one of the Magento 2 modules. We use the contentblock name to reference and add a block for rendering the example.phtmltemplate file.

The path to a template. It usually includes a name of a module joined with "::" template file name. The example.phtmlfile should be located in the MageMastery/FirstLayout/view/frontend/templatesdirectory. As you may notice, there is no mention in the layout configuration file about the rendering area. However, Magento 2 rendering mechanism knows how to locate the corresponding area of a layout and find a path to the template.

As you may notice, there is also no templatesdirectory specified for the template="MageMastery\_FirstLayout::example.phtml". In case we want to add a sub-directory inside the templatesdirectory, let's say a viewdirectory, the configuration of the template should include the sub-directory template="MageMastery\_FirstLayout::view/example.phtml".

The example.phtmlfile is a simple and static template that provides a few HTML tags.

<h1>My First Template</h1>

<p>With Layout support</p>

The content of the example.phtmlfile is rendered by the Magento\Framework\View\Element\Templateclass.

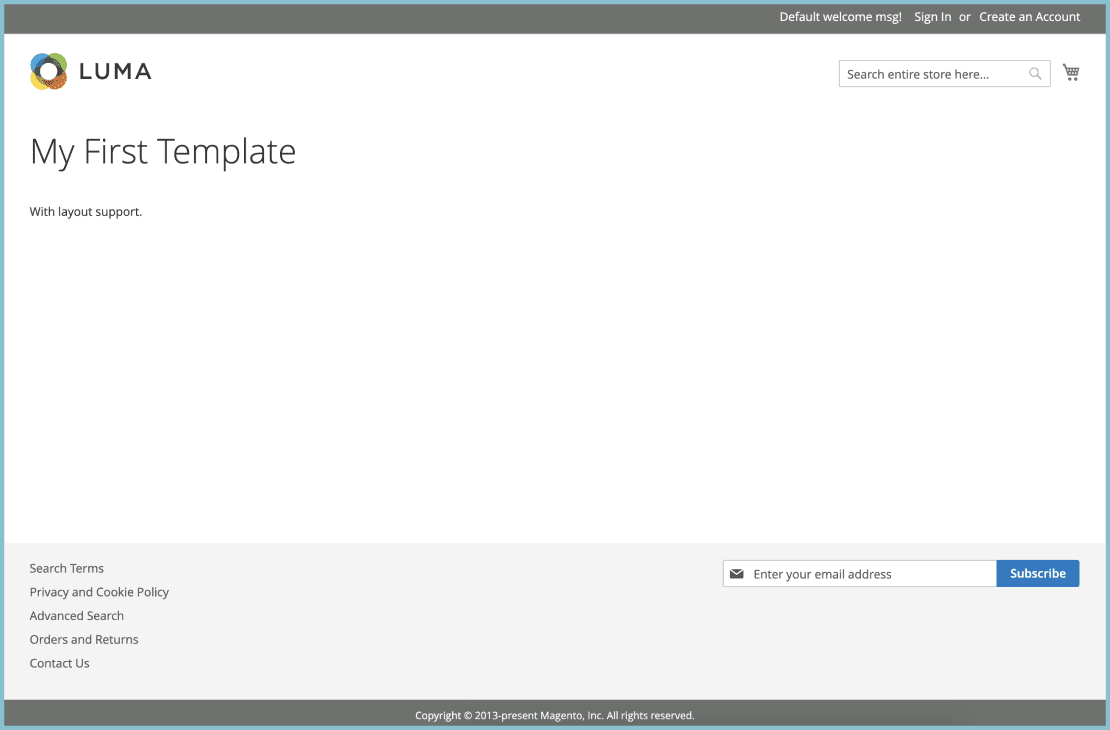
Layout File Name

Let's have a look at the layout file name. The magemastery\_firstlayout\_page\_view.xmlfile name is not an easy-to-guess name. There is some naming convention comes into place.

First of all, a name of the layout file should include the ID of the route. The ID of the route is located in the etc/frontend/routes.xmlfile. In case of this lesson this is the magemastery\_firstlayoutID. The pagepart of the layout name comes from a name of the controller directory. Finally, the last viewpath is the name of a controller class, which is View.

Once, we have a custom controller and we want to render a page with the custom layout. The magemastery\_firstlayout\_page\_viewname should be provided. Magento 2 will match the the layout configuration file, load the declared list of templates and classes, and render the content of the template on a page.

As the result, we can navigate to the https:\/\/magento2ce.magemastery.net/firstlayout/page/view URL and observe a custom page with the custom content rendered on the page.

[](https://magemastery.net/static/2867aa331d0e86a3159ed8e8e7e78544/73caa/layout-and-template-custom-page.png)

Homework

The homework for this lesson.

1. Create a hello.phtmltemplate under the MageMastery/FirstTemplate/view/frontend/templates/viewdirectory.
2. Add the template into the magemastery\_firstlayout\_page\_view.xmlfile as an additional block declaration, that should go after the example.phtmlblock.
3. Check that the new content of the hello.phtmltemplate is rendered on the page.

# **How to Create Controller in Magento 2**

[**Controller**](https://www.mageplaza.com/magento-2-module-development/how-to-create-controllers-magento-2.html) specially is one of the important thing in [Module development series](https://www.mageplaza.com/magento-2-module-development/), and PHP MVC Framework in general. It functionarity is that received request, process and render page.

In **Magento 2 Controller** has one or more files in Controller folder of module, it includes actions of class which contain execute() method. There are 2 different controllers, they are **frontend controller** and **backend controller**. They are generally similar of workflow, but admin controller is a little different. There is a checking permission method in admin controller, it calls form key.

## How controller work?

It receive an request from end-user (browser or comamnd line), for example:

http://example.com/route\_name/controller/action

* route\_name is a unique name which is set in [routes.xml](https://www.mageplaza.com/magento-2-module-development/magento-2-routing.html).
* controller is the folder inside **Controller** folder.
* action is a class with execute method to process request.

One of the important in [Magento system](https://www.mageplaza.com/kb/magento-2-system-requirements.html) is **frontController** (Magento\Framework\App\FrontController), it alway receives request then route controller, action by route\_name Let’s take an example of routing an request:

foreach ($this->\_routerList as $router) {

try {

$actionInstance = $router->match($request);

…

}

If there is an action of controller class found, execute() method will be run.

## How to create a controller in Magento 2?

**To Create Controller in Magento 2:**

* [Step 1: Create routes.xml file](https://www.mageplaza.com/magento-2-module-development/how-to-create-controllers-magento-2.html#step-1-create-routesxml-file)
* [Step 2: Create controller file](https://www.mageplaza.com/magento-2-module-development/how-to-create-controllers-magento-2.html#step-2-create-controller-file)
* [Step 3: Create controller Layout file](https://www.mageplaza.com/magento-2-module-development/how-to-create-controllers-magento-2.html#step-3-create-controller-layout-file)
* [Step 4: Create controller Block file](https://www.mageplaza.com/magento-2-module-development/how-to-create-controllers-magento-2.html#step-4-create-controller-block-file)
* [Step 5: Create controller template file](https://www.mageplaza.com/magento-2-module-development/how-to-create-controllers-magento-2.html#step-5-create-controller-template-file)
* [Step 6: Flush Magento cache](https://www.mageplaza.com/magento-2-module-development/how-to-create-controllers-magento-2.html#step-6-flush-magento-cache)
* [Step 7: Run a test new controller](https://www.mageplaza.com/magento-2-module-development/how-to-create-controllers-magento-2.html#step-7-run-a-test-new-controller)

**To create a controller**, we need to create a folder inside Controller folder of module and declare an action class inside it. For example, we create a index controller and a index action for module Mageplaza\_HelloWorld:

### Step 1: Create routes.xml file.

File: app/code/Mageplaza/HelloWorld/etc/frontend/routes.xml

<?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 frontName="helloworld" id="helloworld">

<module name="Mageplaza\_HelloWorld"/>

</route>

</router>

</config>

In the previous [How to Create Module in Magento 2](https://www.mageplaza.com/magento-2-module-development/how-create-hello-world-module-magento-2.html) , we created file routes.xml. If you created it, you can ignore this step.

### Step 2: Create controller file

File: app/code/Mageplaza/HelloWorld/Controller/Index/Index.php

<?php

namespace Mageplaza\HelloWorld\Controller\Index;

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

{

protected $\_pageFactory;

public function \_\_construct(

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

\Magento\Framework\View\Result\PageFactory $pageFactory)

{

$this->\_pageFactory = $pageFactory;

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

}

public function execute()

{

return $this->\_pageFactory->create();

}

}

As you see, all controllers must be extended \Magento\Framework\App\Action\Action class which has dispatch method which will call execute() method in action class. In this execute() method, we will write all of our controller logic and will return response for the request.

### Step 3: Create Layout file

File: app/code/Mageplaza/HelloWorld/view/frontend/layout/helloworld\_index\_index.xml

<?xml version="1.0"?>

<page xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" layout="1column" xsi:noNamespaceSchemaLocation="urn:magento:framework:View/Layout/etc/page\_configuration.xsd">

<referenceContainer name="content">

<block class="Mageplaza\HelloWorld\Block\Index" name="helloworld\_index\_index" template="Mageplaza\_HelloWorld::index.phtml" />

</referenceContainer>

</page>

### Step 4: Create Block file

File: app/code/Mageplaza/HelloWorld/Block/Index.php

<?php

namespace Mageplaza\HelloWorld\Block;

class Index extends \Magento\Framework\View\Element\Template

{

}

### Step 5: Create template file

File: app/code/Mageplaza/HelloWorld/view/frontend/templates/index.phtml

<h2>Welcome to Mageplaza.com</h2>

We can learn more View: Layout, Block, Template in [this topic](https://www.mageplaza.com/magento-2-module-development/view-block-layout-template-magento-2.html).

### Step 6: Flush Magento cache

[How to flush Magento cache here](https://www.mageplaza.com/kb/how-flush-enable-disable-cache.html)

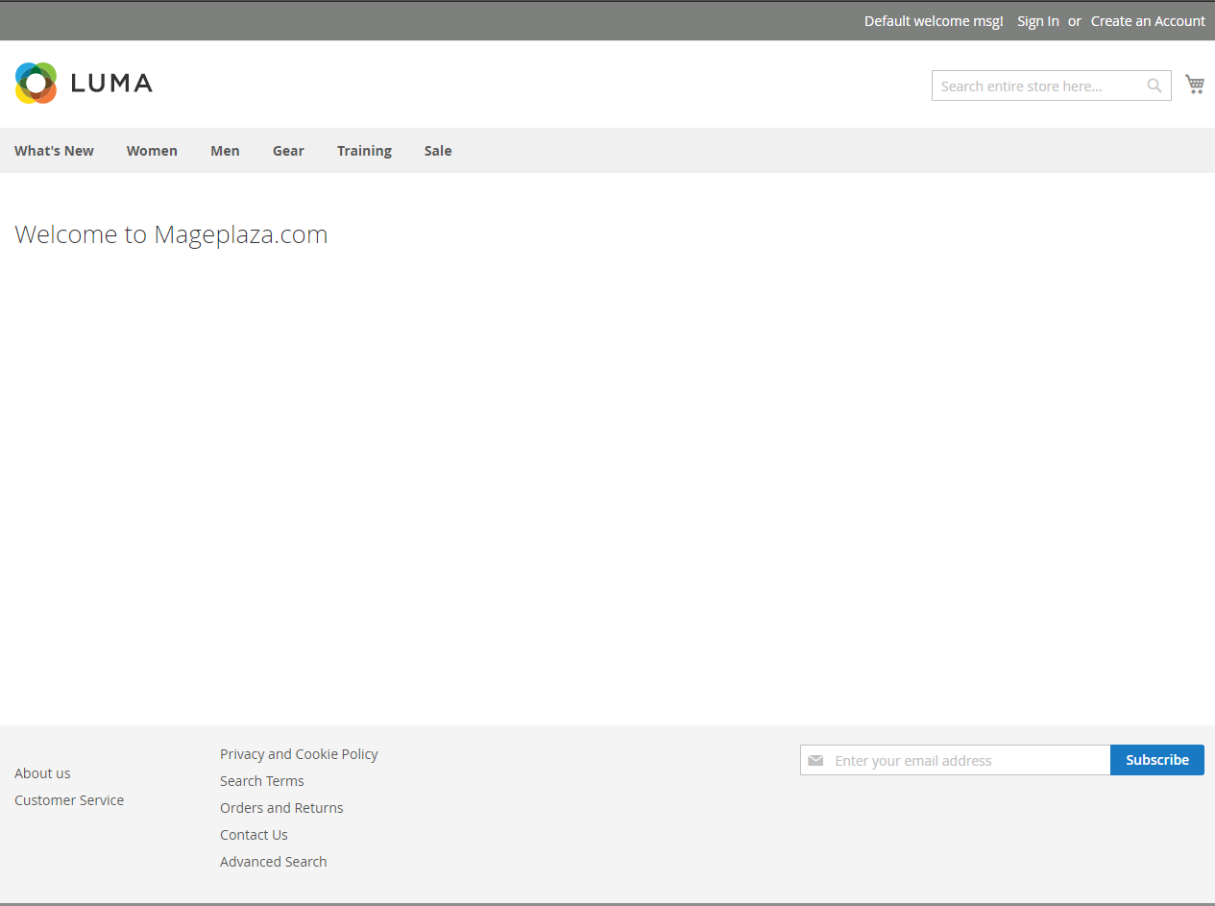
### Step 7: Run a test

Let’s open browser and navigate to

http://<yourhost.com>/helloworld/index/index

or

http://<yourhost.com>/helloworld/



## Permission - ACL

There is a checking permission method in admin controller. Let’s take an example:

protected function \_isAllowed()

{

return $this->\_authorization->isAllowed('Magento\_AdminNotification::show\_list');

}

It will check the current user has right to access this action or not, learn more [Admin ACL Access Control Lists](https://www.mageplaza.com/magento-2-module-development/magento-2-acl-access-control-lists.html)

## Other methods in Magento 2 Controller

### \_forward() and \_redirect() action.

\Magento\Framework\App\Action\Action class provide us 2 important methods: \_forward and \_redirect.

**Forward method**

\_forward() protected function will edit the request to transfer it to another controller/action class. This will not change the request url. For example, we have 2 actions Forward and [Hello World](https://www.mageplaza.com/magento-2-module-development/) like this:

namespace Mageplaza\HelloWorld\Controller\Test;

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

{

public function execute()

{

$this->\_forward('hello');

}

}

If you make a request to http://example.com/route\_name/test/forward , here are result will be displied on the screen.

Hello World! Welcome to Mageplaza.com

You can also change the controller, module and set param for the request when forward. Please check the \_forward() function for more information:

protected function \_forward($action, $controller = null, $module = null, array $params = null)

{

$request = $this->getRequest();

$request->initForward();

if (isset($params)) {

$request->setParams($params);

}

if (isset($controller)) {

$request->setControllerName($controller);

// Module should only be reset if controller has been specified

if (isset($module)) {

$request->setModuleName($module);

}

}

$request->setActionName($action);

$request->setDispatched(false);

}

#### **Redirect method**

This method will transfer to another controller/action class and also change the response header and the request url. With above example, if we replace \_forward() method by this \_redirect() method:

$this->\_redirect('\*/\*/hello');

Then after access from the url http://example.com/route\_name/test/forward, the url will be change to http://example.com/route\_name/test/hello and show the message Hello World! Welcome to Mageplaza.com on the screen.

If you got this error message: **Exception printing is disabled by default for security reasons**, [this topic](https://www.mageplaza.com/kb/exception-printing-is-disabled-by-default-for-security-reasons-magento-1-2.html) may help.

While we are working on Magento2 store we need few of commands to be executed. But remembering all the commands is little bit tough task.  
Here, I will provide you all useful command list for Magneto.

## Admin

#### **1. Create an admin user:**

php bin/magento admin:user:create

Example:

php bin/magento admin:user:create --admin-user='admin' --admin-password='admin123' --admin-email='admin@xyz.com' --admin-firstname='Aryan' --admin-lastname='Srivastava'

#### **2. Unlock admin user account**

php bin/magento admin:user:unlock username

Example:

php bin/magento admin:user:unlock admin

## Cache

#### **1. Cleans cache**

php bin/magento cache:clean

Short code:

php bin/magento c:c

#### **2. Flushes cache storage**

php bin/magento cache:flush

Short code:

php bin/magento c:f

#### **3. Disables cache**

php bin/magento cache:disable

Short code:

php bin/magento c:d

#### **4. Enable cache**

php bin/magento cache:enable

Short code:

php bin/magento c:e

#### **5. Check cache status**

php bin/magento cache:status

## Setup

#### **1. Setup upgrades**

Upgrades the Magento application, DB data, and schema

php bin/magento setup:upgrade

Short code:

php bin/magento s:up

#### **2. Setup compile**

Generates DI configuration and all missing classes that can be auto-generated.

php bin/magento setup:di:compile

Short code:

php bin/magento s:d:c

#### **3. Setup deploy**

Deploys static view files.

php bin/magento setup:static-content:deploy

Short code:

php bin/magento s:s:d

Note: In developer/default mode we need to force deploy flag (-f) to deploy static view files.

Example:

php bin/magento setup:static-content:deploy -f

Or

php bin/magento s:s:d -f

#### **4. Uninstall magento application**

php bin/magento setup:uninstall

#### **5. Takes backup of Magento Application code base, media and database**

php bin/magento setup:backup --code --db --media

## Deploy

#### **1. Set deploy mode**

Set application mode.

php bin/magento deploy:mode:set <value>

Note: Value can be:

a. developer

b. production

c. default

Example:

php bin/magento deploy:mode:set production

#### **2. Show deploy mode**

Displays current application mode.

php bin/magento deploy:mode:show

## Dev

#### **1. Enable profiler**

php bin/magento dev:profiler:enable

#### **2. Disable profiler**

php bin/magento dev:profiler:disable

## Maintenance

#### **1. Show status of maintenance mode**

php bin/magento maintenance:status

#### **2. Enable maintenance mode**

php bin/magento maintenance:enable

Enable maintenance mode except specific IP(s):

php bin/magento maintenance:enable --ip=127.0.0.1 --ip=127.0.0.2

#### **3. Allow specific IP(s)**

Sets maintenance mode exempt IPs.

php bin/magento maintenance:allow-ips --ip=127.0.0.1 --ip=127.0.0.2

#### **4. Disable maintenance mode**

php bin/magento maintenance:disable

## Catalog

#### **1. Create resized product image**

php bin/magento catalog:images:resize

#### **2. Removes unused product attributes in magento application using cli**

php bin/magento catalog:product:attributes:cleanup

## Indexer

#### **1. Show indexer info**

php bin/magento indexer:info

Short code:

php bin/magento i:info

#### **2. Show indexer status**

php bin/magento indexer:status

Short code:

php bin/magento i:status

#### **3. Reset indexer**

php bin/magento indexer:reset

Short code:

php bin/magento i:reset

#### **4. Reindexes data**

php bin/magento indexer:reindex

Short code:

php bin/magento i:rei

## Module

#### **1. Displays status of modules**

php bin/magento module:status

#### **2. Enable modules**

php bin/magento module:enable <vendorname\_modulename>

Example:

php bin/magento module:enable Magento\_TwoFactorAuth Magento\_CmsPageBuilderAnalytics

#### **3. Disable modules**

php bin/magento module:disable <vendorname\_modulename>

Example:

php bin/magento module:disable Magento\_TwoFactorAuth Magento\_CmsPageBuilderAnalytics

## Store

#### **1. Displays the list of stores**

php bin/magento store:list

#### **2. Displays the list of websites**

php bin/magento store:website:list

## Cron

#### **1. Install crontab in magento**

php bin/magento cron:install

#### **2. Run cron in magento**

php bin/magento cron:run

#### **3. Remove crontab in magento**

php bin/magento cron:remove

## Info

#### **1. Displays the Magento Admin URI**

php bin/magento info:adminuri

#### **2. Prints list of available backup files**

php bin/magento info:backups:list

#### **3. Displays the list of available currencies**

php bin/magento info:currency:list

#### **4. Displays the list of available language locales**

php bin/magento info:language:list

#### **5. Displays the list of available timezones**

php bin/magento info:timezone:list

In our everyday activity, while working on Magento2, we need a few things, but remembering all this is a bit of a hectic task. So I’ll try to list one possible Magento2 Cheat Sheet for you.

You may also like:

1. [Magento2 Useful Command-List](https://desssigner.in/magento2-useful-command-list/)

Let’s start with Magento2 Cheat Sheet:

## Magento2: Get URL(s)

#### **1. Magento2: Get Url in phtml file**

**<?php** echo $this->getUrl(); **?>**

For example:

1. Add contact page URL in phtml file

<a href="**<?php** echo $this->getUrl('contact'); **?>**" title="**<?=** \_\_('Contact Us');**?>**">**<?=** \_\_('Contact Us');**?>**</a>

#### **2. Magento2: Get Web / Base URL in Static block:**

{{config path="web/unsecure/base\_url"}}

For example:

1. Add contact page URL

<a href="{{config path="web/unsecure/base\_url"}}contact" title="Contact Us">Contact Us</a>

#### **3. Magento2: Get Web / Secure Base URL in Static block:**

{{config path="web/secure/base\_url"}}

For example:

1. Add contact page URL

<a href="{{config path="web/secure/base\_url"}}contact" title="Contact Us">Contact Us</a>

#### **4. Magento2: Get General / Store Information / Store Name in Static block:**

{{config path="general/store\_information/name"}}

For example:

1. Show store name:

<span>{{config path="general/store\_information/name"}}</span>

#### **5. Magento2: Get image URL in phtml file**

**<?php** echo $this->getViewFileUrl('images/demo.jpg'); **?>**

For example:

If the image stored under **app/design/frontend/<vendor>/<theme>/web/images/** folder

<img src="**<?php** echo $this->getViewFileUrl('images/demo.jpg'); **?>**" alt="demo">

## Magento2: Add a custom class on the body tag

Sometimes we need to add a custom class to the body tag. If you think it is a tough task, let me show you it is very easy.

<attribute name="class" value="custom-simple-product"/>

For example:

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

<page xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" layout="1column" xsi:noNamespaceSchemaLocation="urn:magento:framework:View/Layout/etc/page\_configuration.xsd">

<body>

<attribute name="class" value="custom-class"/>

</body>

</page>

**Note:**

1. If you want to add a custom body class for all pages, then write above code in default.xml
2. If you want to add a custom body class for a specific page, you need to write above to specific layout handle.
   1. Homepage – cms\_index\_index.xml
   2. Product page – catalog\_product\_view.xml
   3. Category page – catalog\_category\_view.xml and so on.

## Magento2: Add a custom css file(s)

Sometimes we need to add a custom css file.

<css src="css/custom.css" />

For example:

If the css file stored under **app/design/frontend/<vendor>/<theme>/web/css/** folder

<?xml version="1.0"?>

<page layout="3columns" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="urn:magento:framework:View/Layout/etc/page\_configuration.xsd">

<head>

<css src="css/custom.css" />

</head>

</page>

If the css file stored under **app/code/<vendor>/<module>/view/frontend/web/css** folder

<?xml version="1.0"?>

<page layout="3columns" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="urn:magento:framework:View/Layout/etc/page\_configuration.xsd">

<head>

<css src="Vendor\_Module::css/custom.css" />

</head>

</page>

## Magento2: Add a custom JS file(s)

Sometimes we need to add a custom Js file.

<link src="js/custom.js" />

For example:

If the JS file stored under **app/design/frontend/<vendor>/<theme>/web/js/** folder

<?xml version="1.0"?>

<page layout="3columns" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="urn:magento:framework:View/Layout/etc/page\_configuration.xsd">

<head>

<link src="js/custom.js" />

</head>

</page>

If the js file stored under **app/code/<vendor>/<module>/view/frontend/web/js** folder

<?xml version="1.0"?>

<page layout="3columns" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="urn:magento:framework:View/Layout/etc/page\_configuration.xsd">

<head>

<link src="Vendor\_Module::js/custom.js" />

</head>

</page>

## Magento2: List of layout handles

List of common layout handles:

| **Pages** | **Layout Handle** |
| --- | --- |
| Homepage | cms\_index\_index.xml |
| Shopping Cart page | checkout\_cart\_index.xml |
| Product page | catalog\_product\_view.xml |
| Product page (simple product) | catalog\_product\_view\_type\_simple.xml |
| Product page (configurable product) | catalog\_product\_view\_type\_configurable.xml |
| Product page (virtual product) | catalog\_product\_view\_type\_virtual.xml |
| Category page | catalog\_category\_view.xml |
| Checkout page | checkout\_index\_index.xml |
| Order Success page | checkout\_onepage\_success.xml |
| Order Failure page | checkout\_onepage\_failure.xml |
| Signin/Login page | customer\_account\_login.xml |
| Signup page | customer\_account\_create.xml |
| Forget password page | customer\_account\_forgotpassword.xml |
| Customer account dashboard | customer\_account.xml |
| Contact us page | contact\_index\_index.xml |
| 404 Error page | cms\_noroute\_index.xml |
| Wishlist | wishlist\_index\_index.xml |
| Search result | catalogsearch-result-index |

Common used layout handles in Magento2

## Magento2: Call Static Block in CMS Page

{{block class="Magento\Cms\Block\Block" block\_id="your\_block\_identifier"}}

## Magento2: Call Static Block in Template (.phtml) file

**<?php**

echo $this->getLayout()

->createBlock('Magento\Cms\Block\Block')

->setBlockId('your\_block\_identifier')

->toHtml();

?

## Magento2: Call Static Block in Layout (XML) File

<referenceContainer name="content">

<block class="Magento\Cms\Block\Block" name="block\_identifier">

<arguments>

<argument name="block\_id" xsi:type="string">block\_identifier</argument>

</arguments>

</block>

</referenceContainer>

## Magento2: Call template(.phtml) file in static block / cms page

{{block class="Magento\Framework\View\Element\Template" template="Vendor\_Module::custom.phtml"}}

Hope this article helped you